## Contents

### Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vii</td>
<td></td>
</tr>
</tbody>
</table>

### About ILE C/C++ Run-time Library Functions (SC41-5607)

<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subcategory</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ix</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A note about examples</th>
<th>x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Prerequisite and related information</th>
<th>x</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>How to send your comments</th>
<th>x</th>
</tr>
</thead>
</table>

### Part 1. Run-Time Library Functions

#### Chapter 1. Include Files

- `<assert.h>`: 3
- `<ctype.h>`: 3
- `<decimal.h>`: 3
- `<errno.h>`: 4
- `<except.h>`: 4
- `<float.h>`: 7
- `<langinfo.h>`: 7
- `<limits.h>`: 7
- `<locale.h>`: 7
- `<math.h>`: 7
- `<mallocinfo.h>`: 8
- `<monetary.h>`: 8
- `<nl_types.h>`: 8
- `<pointer.h>`: 8
- `<recio.h>`: 8
- `<regex.h>`: 12
- `<setjmp.h>`: 13
- `<signal.h>`: 13
- `<stdarg.h>`: 13
- `<stddef.h>`: 13
- `<stdio.h>`: 14
- `<stdlib.h>`: 15
- `<string.h>`: 16
- `<strings.h>`: 16
- `<time.h>`: 16
- `<wchar.h>`: 17
- `<wcstr.h>`: 17
- `<wctype.h>`: 17
- `<xxcvt.h>`: 18
- `<xxdtaa.h>`: 18
- `<xxenv.h>`: 18
- `<xxdbk.h>`: 18

#### Machine Interface (MI) Include Files

| MI Include Files | 19 |

### Chapter 2. Library Functions

#### The C/C++ Library

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort()</td>
<td>38</td>
</tr>
<tr>
<td>abs()</td>
<td>39</td>
</tr>
<tr>
<td>acos()</td>
<td>40</td>
</tr>
<tr>
<td>asctime()</td>
<td>41</td>
</tr>
<tr>
<td>asctime_r()</td>
<td>43</td>
</tr>
<tr>
<td>atexit()</td>
<td>48</td>
</tr>
<tr>
<td>atof()</td>
<td>50</td>
</tr>
<tr>
<td>atoi()</td>
<td>51</td>
</tr>
<tr>
<td>atol()</td>
<td>52</td>
</tr>
<tr>
<td>bsearch()</td>
<td>54</td>
</tr>
<tr>
<td>btowc()</td>
<td>55</td>
</tr>
<tr>
<td>_C_Get_Ssn_Handle()</td>
<td>57</td>
</tr>
<tr>
<td>calloc()</td>
<td>58</td>
</tr>
<tr>
<td>catclose()</td>
<td>59</td>
</tr>
<tr>
<td>catgets()</td>
<td>60</td>
</tr>
<tr>
<td>catopen()</td>
<td>61</td>
</tr>
<tr>
<td>ceil()</td>
<td>62</td>
</tr>
<tr>
<td>clearerr()</td>
<td>63</td>
</tr>
<tr>
<td>clock()</td>
<td>64</td>
</tr>
<tr>
<td>cos()</td>
<td>65</td>
</tr>
<tr>
<td>cosh()</td>
<td>66</td>
</tr>
<tr>
<td>_C_TS_malloc_debug()</td>
<td>67</td>
</tr>
<tr>
<td>_C_TS_malloc_info()</td>
<td>68</td>
</tr>
<tr>
<td>difftime()</td>
<td>69</td>
</tr>
<tr>
<td>div()</td>
<td>70</td>
</tr>
<tr>
<td>erfc()</td>
<td>71</td>
</tr>
<tr>
<td>exit()</td>
<td>72</td>
</tr>
<tr>
<td>exp()</td>
<td>73</td>
</tr>
<tr>
<td>exp()</td>
<td>74</td>
</tr>
<tr>
<td>exp()</td>
<td>75</td>
</tr>
<tr>
<td>exp()</td>
<td>76</td>
</tr>
<tr>
<td>exp()</td>
<td>77</td>
</tr>
<tr>
<td>exp()</td>
<td>78</td>
</tr>
<tr>
<td>exp()</td>
<td>79</td>
</tr>
<tr>
<td>exp()</td>
<td>80</td>
</tr>
<tr>
<td>exp()</td>
<td>81</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>remove() — Delete File</td>
<td>257</td>
</tr>
<tr>
<td>rename() — Rename File</td>
<td>257</td>
</tr>
<tr>
<td>rewind() — Adjust Current File Position</td>
<td>259</td>
</tr>
<tr>
<td>_Rfseek() — Force the End-of-Data</td>
<td>260</td>
</tr>
<tr>
<td>_Rfseek0() — Force the End-of-File</td>
<td>261</td>
</tr>
<tr>
<td>_Rfmax() — Set the Record Format Name</td>
<td>262</td>
</tr>
<tr>
<td>_Rindata() — Set Separate Indicator Area</td>
<td>264</td>
</tr>
<tr>
<td>_Riofbk() — Obtain I/O Feedback Information</td>
<td>266</td>
</tr>
<tr>
<td>_Rlocate() — Position a Record</td>
<td>268</td>
</tr>
<tr>
<td>_Ropen() — Open a Record File for I/O Operations</td>
<td>271</td>
</tr>
<tr>
<td>_Ropenfbk() — Obtain Open Feedback Information</td>
<td>275</td>
</tr>
<tr>
<td>_Rpgmdev() — Set Default Program Device</td>
<td>276</td>
</tr>
<tr>
<td>_Rread() — Read a Record by Relative Record</td>
<td>277</td>
</tr>
<tr>
<td>_Readf() — Read the First Record</td>
<td>279</td>
</tr>
<tr>
<td>_Readindv() — Read from an Invited Device</td>
<td>281</td>
</tr>
<tr>
<td>_Readk() — Read a Record by Key</td>
<td>284</td>
</tr>
<tr>
<td>_Readl() — Read the Last Record</td>
<td>288</td>
</tr>
<tr>
<td>_Readn() — Read the Next Record</td>
<td>289</td>
</tr>
<tr>
<td>_Readnc() — Read the Next Changed Record in a Subfile</td>
<td>292</td>
</tr>
<tr>
<td>_Rreadp() — Read the Previous Record</td>
<td>293</td>
</tr>
<tr>
<td>_Reads() — Read the Same Record</td>
<td>296</td>
</tr>
<tr>
<td>_Rrelease() — Release a Program Device</td>
<td>297</td>
</tr>
<tr>
<td>_Rlslock() — Release a Record Lock</td>
<td>299</td>
</tr>
<tr>
<td>_Rrollbck() — Roll Back Commitment Control Changes</td>
<td>300</td>
</tr>
<tr>
<td>_Rupdate() — Update a Record</td>
<td>302</td>
</tr>
<tr>
<td>_Rupfb() — Provide Information on Last I/O Operation</td>
<td>304</td>
</tr>
<tr>
<td>_Rwrite() — Write the Next Record</td>
<td>305</td>
</tr>
<tr>
<td>_Rwritten() — Write a Record Directly</td>
<td>307</td>
</tr>
<tr>
<td>_Rwiterd() — Write and Read a Record</td>
<td>310</td>
</tr>
<tr>
<td>_Rwread() — Write and Read a Record (separate buffers)</td>
<td>311</td>
</tr>
<tr>
<td>scanf() — Read Data</td>
<td>313</td>
</tr>
<tr>
<td>setbuf() — Control Buffering</td>
<td>319</td>
</tr>
<tr>
<td>set jmp() — Preserve Environment</td>
<td>321</td>
</tr>
<tr>
<td>setlocale() — Set Locale</td>
<td>322</td>
</tr>
<tr>
<td>setbufv() — Control Buffering</td>
<td>328</td>
</tr>
<tr>
<td>signal() — Handle Interrupt Signals</td>
<td>330</td>
</tr>
<tr>
<td>sin() — Calculate Sine</td>
<td>332</td>
</tr>
<tr>
<td>sinh() — Calculate Hyperbolic Sine</td>
<td>333</td>
</tr>
<tr>
<td>snprintf() — Format and Write Wide Characters and to a Stream</td>
<td>334</td>
</tr>
<tr>
<td>sprintf() — Print Formatted Data to Buffer</td>
<td>335</td>
</tr>
<tr>
<td>sqrt() — Calculate Square Root</td>
<td>337</td>
</tr>
<tr>
<td>srand() — Set Seed for rand() Function</td>
<td>338</td>
</tr>
<tr>
<td>sscanf() — Read Data</td>
<td>339</td>
</tr>
<tr>
<td>strftime() — Convert Date/Time to String</td>
<td>354</td>
</tr>
<tr>
<td>strcspn() — Find Offset of First Character Match</td>
<td>350</td>
</tr>
<tr>
<td>strerror() — Set Pointer to Run-Time Error Message</td>
<td>350</td>
</tr>
<tr>
<td>strfmon() — Convert Monetary Value to String</td>
<td>351</td>
</tr>
<tr>
<td>stricmp() — Compare Strings without Case</td>
<td>357</td>
</tr>
<tr>
<td>strl() — Determine String Length</td>
<td>358</td>
</tr>
<tr>
<td>strncasecmp() — Compare Strings without Case</td>
<td>358</td>
</tr>
<tr>
<td>strcoll() — Compare Strings</td>
<td>346</td>
</tr>
<tr>
<td>strcmp() — Compare Strings</td>
<td>347</td>
</tr>
<tr>
<td>strcmpi() — Compare Strings without Case</td>
<td>348</td>
</tr>
<tr>
<td>strcpy() — Copy Strings</td>
<td>349</td>
</tr>
<tr>
<td>strncpy() — Copy Strings</td>
<td>350</td>
</tr>
<tr>
<td>strncmp() — Compare Strings without Case</td>
<td>351</td>
</tr>
<tr>
<td>strncasecmp() — Compare Strings without Case</td>
<td>352</td>
</tr>
<tr>
<td>strncmp() — Compare Strings</td>
<td>353</td>
</tr>
<tr>
<td>strncpy() — Copy Strings</td>
<td>354</td>
</tr>
<tr>
<td>strncmp() — Compare Strings without Case</td>
<td>355</td>
</tr>
<tr>
<td>strlen() — Determine String Length</td>
<td>356</td>
</tr>
<tr>
<td>strpbrk() — Find Characters in String</td>
<td>357</td>
</tr>
<tr>
<td>strptime() — Convert String to Date/Time</td>
<td>358</td>
</tr>
<tr>
<td>strchr() — Locate Last Occurrence of Character in String</td>
<td>359</td>
</tr>
<tr>
<td>strn() — Find Offset of First Non-matching Character.</td>
<td>360</td>
</tr>
<tr>
<td>str() — Locate Substring</td>
<td>361</td>
</tr>
<tr>
<td>strtok() — Convert Character String to Double</td>
<td>362</td>
</tr>
<tr>
<td>strtok() — Tokenize String</td>
<td>363</td>
</tr>
<tr>
<td>strtok_r() — Tokenize String (Restartable)</td>
<td>364</td>
</tr>
<tr>
<td>strtol() — Convert Character String to Long and Long Long Integer</td>
<td>365</td>
</tr>
<tr>
<td>stroul() — Convert Character String to Unsigned Long and Signed Long Integer</td>
<td>366</td>
</tr>
<tr>
<td>strxfrm() — Transform String</td>
<td>367</td>
</tr>
<tr>
<td>swprintf() — Format and Write Wide Characters to Buffer</td>
<td>368</td>
</tr>
<tr>
<td>_ultoa() — Convert Unsigned Long Integer to String</td>
<td>369</td>
</tr>
<tr>
<td>_tolower() — Convert Character Case Representable by ASCII</td>
<td>370</td>
</tr>
<tr>
<td>toupper() — Convert Character to Character Case</td>
<td>371</td>
</tr>
<tr>
<td>towcase() — Translate Wide Character to Wide Character Case</td>
<td>372</td>
</tr>
<tr>
<td>strto() — Convert String to Double</td>
<td>373</td>
</tr>
<tr>
<td>str() — Locate Substring</td>
<td>374</td>
</tr>
<tr>
<td>strncmp() — Compare Strings</td>
<td>375</td>
</tr>
<tr>
<td>strstr() — Locate Substring</td>
<td>376</td>
</tr>
<tr>
<td>str() — Locate Substring</td>
<td>377</td>
</tr>
<tr>
<td>system() — Execute a Command</td>
<td>378</td>
</tr>
<tr>
<td>tan() — Calculate Tangent</td>
<td>379</td>
</tr>
<tr>
<td>tanh() — Calculate Hyperbolic Tangent</td>
<td>380</td>
</tr>
<tr>
<td>time() — Determine Current Time</td>
<td>381</td>
</tr>
<tr>
<td>tmpfile() — Create Temporary File</td>
<td>382</td>
</tr>
<tr>
<td>tmpnam() — Produce Temporary File Name</td>
<td>383</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>384</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>385</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>386</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>387</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>388</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>389</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>390</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>391</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>392</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>393</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>394</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>395</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>396</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>397</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>398</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>399</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>400</td>
</tr>
<tr>
<td>toascii() — Convert Character to Character Case Representable by ASCII</td>
<td>401</td>
</tr>
<tr>
<td>va_arg() — va_end() — va_start() — Access Function Arguments</td>
<td>402</td>
</tr>
<tr>
<td>vfscanf() — Read Formatted Data from Stream</td>
<td>403</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>404</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data from Stream</td>
<td>405</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>406</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>407</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>408</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>409</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>410</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>411</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>412</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>413</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>414</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>415</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>416</td>
</tr>
<tr>
<td>vslogfmt() — Print Formatted Data as Wide Characters and to a Stream</td>
<td>417</td>
</tr>
</tbody>
</table>
### Tables

1. Grouping Example ........................................... 170
2. Monetary Formatting Example .............................. 171
3. Monetary Fields .............................................. 171
4. Values of Precision ........................................... 217
5. ................................................................. 270
6. Return values of `strcasecmp()` ......................... 340
7. Flags .......................................................... 352
8. Conversion Characters ........................................ 353
9. Return values of `strncasecmp()` ......................... 360
10. Return values of `wcsicmp()` ............................. 439
11. Return values of `__wcsicmp()` ......................... 446
12. `errno` Macros ............................................... 485
13. `errno` Values for IFS Enabled C Stream I/O .......... 487
14. Record Input and Output Error Macro to Exception Mapping ........................................... 489
15. Handling Action Definitions for Signal Values ....... 490
16. Default Actions for Signal Values ....................... 491
17. Signal to iSeries Exception Mapping .................... 492
18. Determining Canceled Invocation Reason Codes ........... 493
20. Exception Classes ............................................. 495
21. ILE C Data Type Compatibility with ILE RPG ........ 496
22. ILE C Data Type Compatibility with ILE COBOL .......... 497
23. ILE C Data Type Compatibility with ILE CL .......... 499
24. ILE C Data Type Compatibility with OPM RPG/400 .... 499
25. ILE C Data Type Compatibility with OPM COBOL/400 .... 500
26. ILE C Data Type Compatibility with CL ............... 501
27. Arguments Passed From a Command Line CL Call to an ILE C Program .......................... 502
28. CL Constants Passed from a Compiled CL Program to an ILE C Program ......................... 502
29. CL Variables Passed from a Compiled CL Program to an ILE C Program ......................... 502
30. Invariant Characters ......................................... 503
31. Variant Characters in Different CCSIDs ............... 503
32. Standard C Library Functions ............................. 513
33. ILE C Library Extensions .................................. 529

© Copyright IBM Corp. 1999, 2004
About ILE C/C++ Run-time Library Functions (SC41-5607)

This book provides reference information about:
- Include files
- Run-time functions
- Run-time considerations

Use this book as a reference when you write Integrated Language Environment (ILE) C and C++ applications.

This book does not describe how to program in the C or C++ programming languages, nor does it explain the concepts of ILE. Companion publications for this reference are:
- C/C++ Legacy Class Libraries Reference, SC09-7652-01
- ILE Concepts, SC41-5606-07
- ILE C/C++ for AS/400 MI Library Reference, SC09-2418-00
- Standard C/C++ Library Reference, SC09-4949-01
- WebSphere Development Studio: ILE C/C++ Compiler Reference, SC09-4816-02
- WebSphere Development Studio: ILE C/C++ Language Reference, SC09-7852-00

For other prerequisite and related information, see “Prerequisite and related information” on page x and the “Bibliography” on page 537.

Who should read this book

This book is intended for programmers who are familiar with the C/C++ programming language and who want to write or maintain ILE C/C++ applications. You must have experience in using applicable iSeries menus, and displays or control language (CL) commands. You also need knowledge of Integrated Language Environment as explained in the ILE Concepts manual.

A note about examples

The examples in this book that illustrate the use of library functions are written in a simple style. The examples do not demonstrate all possible uses of C/C++ language constructs. Some examples are only code fragments and do not compile without additional code.

All complete runnable examples for library functions and machine interface instructions are in library QCPLLE, in source file QACSRC. Each example name is the same as the function name or instruction name. For example, the source code for the example illustrating the use of the _Rcommit() function in this book is in library QCPLLE, file QACSRC, member RCOMMIT. The QSYSINC library must be installed.
Prerequisite and related information

Use the iSeries Information Center as your starting point for iSeries technical information.

You can access the Information Center two ways:

- From the following Web site:
  http://www.ibm.com/eserver/iseries/infocenter

- From the iSeries Information Center, SK3T-4091-04 CD-ROM. This CD-ROM ships with your new iSeries hardware or IBM Operating System/400 software upgrade order. You can also order the CD-ROM from the IBM Publications Center:
  http://www.ibm.com/shop/publications/order

The iSeries Information Center contains new and updated iSeries information such as software and hardware installation, Linux, WebSphere, Java, high availability, database, logical partitions, CL commands, and system application programming interfaces (APIs). In addition, it provides advisors and finders to assist in planning, troubleshooting, and configuring your iSeries hardware and software.

With every new hardware order, you receive the iSeries Setup and Operations CD-ROM, SK3T-4098-02. This CD-ROM contains IBM @server IBM e(logo)server iSeries Access for Windows and the EZ-Setup wizard. iSeries Access Family offers a powerful set of client and server capabilities for connecting PCs to iSeries servers. The EZ-Setup wizard automates many of the iSeries setup tasks.

For other related information, see the “Bibliography” on page 537.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other iSeries documentation, fill out the readers’ comment form at the back of this book.

- If you prefer to send comments by mail, use the readers’ comment form with the address that is printed on the back. If you are mailing a readers’ comment form from a country or region other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use either of the following numbers:
  - United States, Canada, and Puerto Rico: 1-800-937-3430
  - Other countries or regions: 1-507-253-5192
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:
    RCHCLERK@us.ibm.com
  - Comments on the iSeries Information Center:
    RCHINFOC@us.ibm.com

Be sure to include the following:

- The name of the book or iSeries Information Center topic.
- The publication number of a book.
- The page number or topic of a book to which your comment applies.
Part 1. Run-Time Library Functions
Chapter 1. Include Files

The include files that are provided with the run time library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from include files to work properly. The inclusion of files is optional, as long as the necessary statements from the files are coded directly into the source.

This section describes each include file, explains its contents, and lists the functions that are declared in the file.

The QSYSINC (system openness includes) library must be installed on your iSeries system. QSYSINC contains include files useful for C/C++ users, such as system API, Dynamic Screen Manager (DSM), and LE header files. The QSYSINC library contains header files that include the prototypes and templates for the MI built-ins and the ILE C/C++ MI functions. See the ILE C/C++ for AS/400 MI Library Reference for more information about these header files.

<assert.h>

The <assert.h> include file defines the `assert` macro. You must include assert.h when you use assert.

The definition of assert is in an `#ifndef` preprocessor block. If you have not defined the identifier NDEBUG through a `#define` directive or on the compilation command, the `assert` macro tests the assertion expression. If the assertion is false, the system prints a message to stderr, and raises an abort signal for the program. The system also does a Dump Job (DMPJOB) OUTPUT(*PRINT) when the assertion is false.

If NDEBUG is defined, assert is defined to do nothing. You can suppress program assertions by defining NDEBUG.

<ctype.h>

The <ctype.h> include file defines functions that are used in character classification. The functions that are defined in <ctype.h> are:

<table>
<thead>
<tr>
<th>isascii</th>
<th>isblank</th>
<th>isgraph</th>
<th>ispunct</th>
<th>toascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum</td>
<td>iscntrl</td>
<td>islower</td>
<td>isspace</td>
<td>tolower</td>
</tr>
<tr>
<td>isalpha</td>
<td>isdigit</td>
<td>isprint</td>
<td>isupper</td>
<td>toupper</td>
</tr>
<tr>
<td>isxdigit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: 1 These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command.
Note: 2 This function is applicable to C++ only.

<decimal.h>

The <decimal.h> include file contains definitions of constants that specify the ranges of the packed decimal type and its attributes. The <decimal.h> file must be included with a `#include` directive in your source code if you use the keywords decimal, digitsof, or precisionof.
**<errno.h>**

The `<errno.h>` include file defines macros that are set to the `errno` variable. The `<errno.h>` include file defines macros for values that are used for error reporting in the C library functions and defines the macro `errno`. An integer value can be assigned to `errno`, and its value can be tested during run time. See "Checking the Errno Value" in the *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for information about displaying the current `errno` value.

**Note:** To test the value of `errno` after library function calls, set it to 0 before the call because its value may not be reset during the call.

**<except.h>**

The `<except.h>` include file declares types and macros that are used in ILE C exception handling.

The definition of `_INTRPT_Hndlr_Parms_T` is:

```c
typedef _Packed struct {
  unsigned int    Block_Size;
  _INVFLAGS_T     Tgt_Flags;
  char            reserved[8];
  INVPTR          Target;
  INVPTR          Source;
  SPCPTR          Com_Area;
  char            Compare_Data[32];
  char            Msg_Id[7];
  char            reserved1;
  _INTRPT_Mask_T  Mask;
  unsigned int    Msg_Ref_Key;
  unsigned short  Exception_Id;
  unsigned short  Compare_Data_Len;
  char            Signal_Class;
  char            Priority;
  short           Severity;
  char            reserved3[4];
  int             Msg_Data_Len;
  char            Mch_Dep_Data[10];
  char            Tgt_Inv_Type;
  SUSPENDPTR      Tgt_Suspend;
  char            Ex_Data[48];
} _INTRPT_Hndlr_Parms_T;
```

**Element**

**Description**

**Block_Size**

The size of the parameter block passed to the exception handler.

**Tgt_Flags**

Contains flags that are used by the system.

**reserved**

An eight byte reserved field.

**Target**

An invocation pointer to the call stack entry that enabled the exception handler.

**Source**

An invocation pointer to the call stack entry that caused the exception. If that call stack entry no longer exists, then this is a pointer to the call stack entry where control resumes when the exception is handled.
Com_Area
A pointer to the communications area variable specified as the second parameter on the #pragma exception_handler. If a communication area was not specified, this value is NULL.

Compare_Data
The compare data consists of 4 bytes of message prefix, for example CPF, MCH, followed by 28 bytes which are taken from the message data of the related message. In the case where the message data is greater than 28 these are the first 28 bytes. For MCH messages, these are the first 28 bytes of the exception related data that is returned by the system (substitution text).

Msg_Id
A message identifier, for example CPF123D. *STATUS message types are not updated in this field.

reserved1
A 1 byte pad.

Mask
This is an 8-byte exception mask, identifying the type of the exception that occurred, for example a decimal data error. The possible types are shown in Table 20 on page 495.

Msg_Ref_Key
A key used to uniquely identify the message.

Exception_Id
Binary value of the exception id, for example, 0x123D. To display value, use conversion specifier %x as information is stored in hex value.

Compare_Data_Len
The length of the compare data.

Signal_Class
Internal signal class.

Priority
The handler priority.

Severity
The message severity.

reserved3
A four-byte reserved field.

Msg_Data_Len
The length of available message data.

Mch_Dep_Data
Machine-dependent data.

Tgt_Inv_Type
Invocation type. Macros are defined in <mimchobs.h>.

Tgt_Suspend
Suspend pointer of the target.

Ex_Data
The first 48 bytes of exception data.

The definition of _CNL_Hndlr_Parms_T is:
typedef _Packed struct {
    unsigned int    Block_Size;
    _INVFLAGS_T     Inv_Flags;
    char            reserved[8];
    _INVPTR         Invocation;
    _SPCPRTR        Com_Area;
    _CNL_Mask_T     Mask;
} _CNL_Hndlr_Parms_T;

Element
    Description

Block_Size
    The size of the parameter block passed to the cancel handler.

Inv_Flags
    Contains flags that are used by the system.

reserved
    An eight byte reserved field.

Invocation
    An invocation pointer to the invocation that is being cancelled.

Com_Area
    A pointer to the handler communications area defined by the cancel
    handler.

Mask
    A 4 byte value indicating the cancel reason.

The following built-ins are defined in <except.h>:

Built-in
    Description

__EXBDY
    The purpose of the __EXBDY built-in or _EXBDY macro is to act as a
    boundary for exception-sensitive operations. An exception-sensitive
    operation is one that may signal an exception. An EXBDY enables
    programmers to selectively suppress optimizations that do code motion.
    For example, a divide is an exception-sensitive operation because it can
    signal a divide-by-zero. An execution path containing both an EXBDY and
    a divide will perform the two in the same order with or without
    optimization. For example:
    
    b = exp1;
    c = exp2;
    ...
    __EXBDY();
    a = b/c;

__VBDY
    The purpose of a __VBDY built-in or _VBDY macro is to ensure the home
    storage locations are current for variables that are potentially used on
    exception paths. This ensures the visibility of the current values of
    variables in exception handlers. A VBDY enables programmers to
    selectively suppress optimizations, such as redundant store elimination
    and forward store motion to enforce sequential consistency of variable
    updates. In the following example, the VBDYs ensure that state is in it's
    home storage location before each block of code that may signal an exception.
    A VBDY is often used in combination with an EXBDY to ensure that earlier
    assignments to state variables really update home storage locations and
    that later exception sensitive operations are not moved before these
    assignments.
state = 1;
_VBDY();
/* Do stuff that may signal an exception. */
state = 2;
_VBDY();
/* More stuff that may signal an exception. */
state = 3;
_VBDY();
For more information on built-ins, see the ILE C/C++ for AS/400 MI Library Reference.

<float.h>

The <float.h> include file defines constants that specify the ranges of floating-point data types. For example, the maximum number of digits for objects of type `double` or the minimum exponent for objects of type `float`.

<langinfo.h>

The <langinfo.h> include file contains the declarations and definitions that are used by `nl_langinfo`.

<limits.h>

The <limits.h> include file defines constants that specify the ranges of integer and character data types. For example, the maximum value for an object of type `char`.

<locale.h>

The <locale.h> include file declares the `setlocale()`, `localeconv()` and `wcslocaleconv()` library functions. These functions are useful for changing the C locale when you are creating applications for international markets.

The <locale.h> include file also declares the type `struct lconv` and the following macro definitions:

```
NULL LC_ALL LC_C LC_C_FRANCE
LC_C_GERMANY LC_C_ITALY LC_C_SPAIN LC_C_UK
LC_C_USA LC_COLLATE LC_CTYPE LC_MESSAGES
LC_MONETARY LC_NUMERIC LC_TIME LC_TOD
LC_UCLS2_ALL LC_UCLS2_COLLATE LC_UCLS2_CTYPE LC_UNI_ALL
LC_UNI_COLLATE LC_UNI_CTYPE LC_UNI_TIME LC_UNI_NUMERIC
LC_UNI_MESSAGES LC_UNI_MONITARY LC_UNI_TOD
```

<math.h>

The <math.h> include file declares all the floating-point math functions:

```
acos cos floor log sqrt
asin cosh fmod log10 tan
atan erf frexp modf tanh
atan2 erfc gamma pow
Bessel exp hypot sin
ceil fabs ldexp sinh
```
Notes:
1. The Bessel functions are a group of functions named $j_0$, $j_1$, $j_n$, $y_0$, $y_1$, and $y_n$.
2. Floating point numbers are only guaranteed 15 significant digits. This can greatly affect expected results if multiple floating point numbers are used in a calculation.

<math.h> defines the macro HUGE_VAL, which expands to a positive double expression, and possibly to infinity on systems that support infinity.

For all mathematical functions, a domain error occurs when an input argument is outside the range of values that are allowed for that function. In the event of a domain error, errno is set to the value of EDOM.

A range error occurs if the result of the function cannot be represented in a double value. If the magnitude of the result is too large (overflow), the function returns the positive or negative value of the macro HUGE_VAL, and sets errno to ERANGE. If the result is too small (underflow), the function returns zero.

<mallocinfo.h>

Include file with _C_TS_malloc_info and _C_TS_malloc_debug.

<money.h>

The <money.h> header file contains declarations and definitions that are related to the output of monetary quantities. The following monetary functions are defined: strfmon() and wcsfmon(). The strfmon() function is not available when LOCALETYPE(*CLD) is specified on the compilation command. The wcsfmon() function is available only when LOCALETYPE(*LOCALEUTF) is specified on the compilation command.

<nl_types.h>

The <nl_types.h> header file contains catalog definitions and the following catalog functions: catclose(), catgets(), and catopen(). These definitions are not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

<pointer.h>

The <pointer.h> include file contains typedefs and pragma directives for the iSeries™ pointer types: space pointer, open pointer, invocation pointer, label pointer, system pointer, and suspend pointer. The typedefs _ANYPTR and _SPCPTRCN are also defined in <pointer.h>.

<recio.h>

The <recio.h> include file defines the types and macros, and prototypes functions for all the ILE C record input and output (I/O) operations.

The following functions are defined in <recio.h>:

_Racquire  _Rclose  _Rcommit  _Rdelete  
_Rdevatr  _Rfend  _Rfdev  _Rformat  
_Rindara  _Riofbk  _Rlocate  _Ropen
The following positioning macros are defined in recio.h:

__END__ __END_FRC__ __FIRST__ __KEY_EQ__
__KEY_GE__ __KEY_GT__ __KEY_LE__ __KEY_LT__
__KEY_NEXTEQ__ __KEY_NEXTUNQ__ __KEY_PREVEQ__ __KEY_PREVUNQ__
__KEY_LAST__ __KEY_NEXT__ __NO_POSITION__ __PREVIOUS__
__PRIOR__ __RRN_EQ__ __START__ __START_FRC__
__LAST__ __NEXT__

The following macros are defined in recio.h:

__DATA_ONLY__ __DFT__ __NO_LOCK__ __NULL_KEY_MAP__

The following directional macros are defined in recio.h:

__READ_NEXT__ __READ_PREV__

The following functions and macros support locate or move mode:

_Rreadd__ _Rreadf__ _Rreadindv__ _Rreadk__
_Rreadl__ _Rreadn__ _Rreadnc__ _Rreadp__
_Rreads__ _Rupdate__ _Rwriterd__ _Rwrited__

Any of the record I/O functions that include a buffer parameter may work in move mode or locate mode. In move mode, data is moved between the user user-supplied buffer and the system buffer. In locate mode, the user must access the data in the system buffer. Pointers to the system buffers are exposed in the _RFILE structure. To specify that locate mode is being used, the buffer parameter of the record I/O function is coded as NULL.

A number of the functions include a size parameter. For move mode, this is the number of data bytes that are copied between the user-supplied buffer and the system buffer. All of the record I/O functions work with one record at a time regardless of the size that is specified. The size of this record is defined by the file description. It may not be equal to the size parameter that is specified by the user on the call to the record I/O functions. The amount of data that is moved between buffers is equal to the record length of the current record format or specified minimum size, whichever is smaller. The size parameter is ignored for locate mode.

The following types are defined in recio.h:

Information for controlling opened record I/O operations
typedef _Packed struct {
    char reserved1[16];
    volatile void *const *const in_buf;
};
volatile void *const *const out_buf;
char reserved2[48];
_RIOFB_T riofb;
char reserved3[32];
const unsigned int buf_length;
char reserved4[28];
volatile char *const in_null_map;
volatile char *const out_null_map;
volatile char *const null_key_map;
char reserved5[48];
const int min_length;
short null_map_len;
short null_key_map_len;
char reserved6[8];
}_RFILE;

Element Description

in_null_map Specifies which fields are to be considered NULL when you read from a database file.

out_null_map Specifies which fields are to be considered NULL when you write to a database file.

null_key_map Specifies which fields contain NULL if you are reading a database by key.

null_map_len Specifies the lengths of the in_null_map and out_null_map.

null_key_map_len Specifies the length of the null_key_map.

Record I/O Feedback Information
typedef struct {
    unsigned char    *key;
    _Sys_Struct_T    *sysparm;
    unsigned long    rrn;
    long num_bytes;
    short blk_count;
    char blk_filled_by;
    int dup_key :1;
    int icf_locate :1;
    int reserved1 :6;
    char reserved2[20];
} _RIOFB_T;

Element Description

key If you are processing a file using a keyed sequence access path, this field contains a pointer to the key value of the record successfully positioned to, read or written.

sysparm This field is a pointer to the major and minor return code for ICF, display, and printer files.

rrn This field contains the relative record number of the record that was successfully positioned to, read or written.

num_bytes This field contains the number of bytes that are read or are written.

blk_count This field contains the number of records that remain in the block. If the file is open for input, blkrcd=y is specified, and a read function is called,
this field will be updated with the number of records remaining in the block.

**blk_filled_by**

This field indicates the operation that filled the block. If the file is open for input, blkrcd=y is specified, and a read function is called. This field will be set to the __READ_NEXT macro if the _Rreadn function filled the block or to the __READ_PREV macro if the _Rreadp function filled the block.

**System-Specific Information**

typedef struct {
    void *sysparm_ext;
    _Maj_Min_rc_T _Maj_Min;
    char reserved1[12];
} _Sys_Struct_T;

**Major and Minor Return Codes**

typedef struct {
    char major_rc[2];
    char minor_rc[2];
} _Maj_Min_rc_T;

The following macros are defined in recio.h:

- **FILENAME_MAX**
  Expands to an integral constant expression that is the size of a character array large enough to hold the longest file name. This is the same as the stream I/O macro.

- **ROPEN_MAX**
  Expands to an integral constant expression that is the maximum number of files that can be opened simultaneously.

The following null field macros are defined in recio.h:

**Element**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLEAR_NULL_MAP(file, type)</strong></td>
</tr>
<tr>
<td>Clears the null output field map that indicates that there are no null fields in the record to be written to file. type is a typedef that corresponds to the null field map for the current record format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLEAR_UPDATE_NULL_MAP(file, type)</strong></td>
</tr>
<tr>
<td>Clears the null input field map that indicates that no null fields are in the record to be written to file. type is a typedef that corresponds to the null field map for the current record format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QRY_NULL_MAP(file, type)</strong></td>
</tr>
<tr>
<td>Returns the number of fields that are null in the previously read record. type is a typedef that corresponds to the null field map for the current record format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLEAR_NULL_KEY_MAP(file, type)</strong></td>
</tr>
<tr>
<td>Clears the null key field map so that it indicates no null key fields in the record to be written to file. type is a typedef that corresponds to the null key field map for the current record format.</td>
</tr>
</tbody>
</table>
_SET_NULL_MAP_FIELD(file, type, field)
Sets the specified field in the output null field map so that field is considered NULL when the record is written to file.

_SET_UPDATE_NULL_MAP_FIELD(file, type, field)
Sets the specified field in the input null field map so that field is considered null when the record is written to file. type is a typedef that corresponds to the null key field map for the record format.

_QRY_NULL_MAP_FIELD(file, type, field)
Returns 1 if the specified field in the null input field map indicates that the field is to be considered null in the previously read record. If field is not null, it returns zero. type is a typedef that corresponds to the NULL key field map for the current record format.

_SET_NULL_KEY_MAP_FIELD(file, type, field)
Sets the specified field map that indicates that the field will be considered null when the record is read from file. type is a typedef that corresponds to the null key field map for the current record format.

_QRY_NULL_KEY_MAP(file, type)
Returns the number of fields that are null in the key of the previously read record. type is a typedef that corresponds to the null field map for the current record format.

_QRY_NULL_KEY_MAP_FIELD(file, type, field)
Returns 1 if the specified field in the null key field map indicates that field is to be considered null in the previously read record. If field is not null, it returns zero. type is a typedef that corresponds to the null key field map for the current record format.

<regex.h>

The <regex.h> include file defines the following regular expression functions:
regcomp() regerror() regexec() regfree()

The <regex.h> include file also declares the regmatch_t type, the regex_t type, which is capable of storing a compiled regular expression, and the following macros:

Values of the cflags parameter of the regcomp() function:
REG_BASIC
REG_EXTENDED
REG_ICASE
REG_NEWLINE
REG_NOSUB

Values of the eflags parameter of the regexec() function:
REG_NOTBOL
REG_NOTEOL

Values of the errcode parameter of the regerror() function:
REG_NOMATCH
REG_BADPAT
REG_ECOLLATE
These declarations and definitions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**<setjmp.h>**

The `<setjmp.h>` include file declares the `setjmp()` macro and `longjmp()` function. It also defines a buffer type, `jmp_buf`, that the `setjmp()` macro and `longjmp()` function use to save and restore the program state.

**<signal.h>**

The `<signal.h>` include file defines the values for signals and declares the `signal()` and `raise()` functions.

The `<signal.h>` include file also defines the following macros:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>SIG_ERR</td>
</tr>
<tr>
<td>SIGALL</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>SIG_DFL</td>
<td>SIG_IIGN</td>
</tr>
<tr>
<td>SIGILL</td>
<td>SIGINT</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>SIGIO</td>
<td>SIGTERM</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>SIGUSR2</td>
</tr>
</tbody>
</table>

`<signal.h>` also declares the function `_GetExcData`, an iSeries extension to the C standard library.

**<stdarg.h>**

The `<stdarg.h>` include file defines macros that allow you access to arguments in functions with variable-length argument lists: `va_arg()`, `va_start()`, and `va_end()`.

The `<stdarg.h>` include file also defines the type `va_list`.

**<stddef.h>**

The `<stddef.h>` include file declares the commonly used pointers, variables, and types as listed below:

- `ptrdiff_t`: typedef for the type of the difference of two pointers
- `size_t`: typedef for the type of the value that is returned by `sizeof`
### wchar_t

typedef for a wide character constant.

The `<stddef.h>` include file also defines the macros `NULL` and `offsetof`. `NULL` is a pointer that is guaranteed not to point to a data object. The `offsetof` macro expands to the number of bytes between a structure member and the start of the structure. The `offsetof` macro has the form:

\[
\text{offsetof(\text{\texttt{structure\_type}}, \text{\texttt{member}})}
\]

The `<stddef.h>` include file also declares the `extern` variable `_EXCP_MSGID`, an iSeries extension to C.

---

### `<stdio.h>`

The `<stdio.h>` include file defines constants, macros, and types, and declares stream input and output functions. The stream I/O functions are:

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>C_Get_Ssn_Handle</code></td>
</tr>
<tr>
<td><code>clearerr</code></td>
</tr>
<tr>
<td><code>fclose</code></td>
</tr>
<tr>
<td><code>fdopen</code>&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>feof</code></td>
</tr>
<tr>
<td><code>ferror</code></td>
</tr>
<tr>
<td><code>fflush</code></td>
</tr>
<tr>
<td><code>fgetpos</code></td>
</tr>
<tr>
<td><code>fgetc</code></td>
</tr>
<tr>
<td><code>fgets</code></td>
</tr>
<tr>
<td><code>fputws</code>&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>fscanf</code></td>
</tr>
<tr>
<td><code>fgetwc</code>&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>fgetws</code>&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>fileno</code>&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td><code>fopen</code></td>
</tr>
</tbody>
</table>

**Note:**

<sup>1</sup> These functions are not available when either `LOCALETYPE(*CLD)` or `SYSIFOPT(*NOIFSIO)` is specified on the compilation command.

<sup>2</sup> These functions are available when `SYSIFOPT(*IFSIO)` is specified on the compilation command.

The `<stdio.h>` include file also defines the macros that are listed below. You can use these constants in your programs, but you should not alter their values.

- **BUFSIZ** Specifies the buffer size that the `setbuf` library function will use when you are allocating buffers for stream I/O. This value establishes the size of system-allocated buffers and is used with `setbuf`.

- **EOF** The value that is returned by an I/O function when the end of the file (or in some cases, an error) is found.

- **FOPEN_MAX** The number of files that can be opened simultaneously.

- **FILENAME_MAX** The longest file name that is supported. If there is no reasonable limit, `FILENAME_MAX` will be the recommended size.

- **_L_tmpnam** The size of the longest temporary name that can be generated by the `tmpnam` function.
**TMP_MAX**

The minimum number of unique file names that can be generated by the `tmpnam` function.

**NULL**

A pointer guaranteed not to point to a data object.

The `FILE` structure type is defined in `<stdio.h>`. Stream I/O functions use a pointer to the `FILE` type to get access to a given stream. The system uses the information in the `FILE` structure to maintain the stream.

When integrated file system is enabled with a compilation parameter `SYSIFCOPT(*IFSIO)`, `ifs.h` is included into `<stdio.h>`.

The C standard streams `stdin`, `stdout`, and `stderr` are also defined in `<stdio.h>`.

The macros `SEEK_CUR`, `SEEK_END`, and `SEEK_SET` expand to integral constant expressions and can be used as the third argument to `fseek()`.

The macros `_IOFBF`, `_IOLBF`, and `_IONBF` expand to integral constant expressions with distinct values suitable for use as the third argument to the `setvbuf` function.

The type `fpos_t` is defined in `<stdio.h>` for use with `fgetpos()` and `fsetpos()`.

See “<stddef.h>” on page 13 for more information on NULL.

---

**<stdlib.h>**

The `<stdlib.h>` include file declares the following functions:

```
abort     div          labs        qsort        strtoul
abs       exit         ldiv        rand        strtol
atexit    free         ldiv        rand_r      system
atof      gcvt         malloc      realloc     ultoa
atoi      getenv       mbcnw      srand        _ultoa
atol      mbtowc      mbstowcs     strtol      wctomb
bsearch   labs        mbstowcs     wctoi
calloc     mbwcstombs  _ultoa      wcstombs
```

**Note:** 1 These functions are applicable to C++ only.

The `<stdlib.h>` include file also contains definitions for the following macros:

**NULL**

The NULL pointer value.

**EXIT_SUCCESS**

Expands to 0; used by the `atexit` function.

**EXIT_FAILURE**

Expands to 8; used by the `atexit` function.

**RAND_MAX**

Expands to an integer that represents the largest number that the `rand` function can return.

**MB_CUR_MAX**

Expands to an integral expression to represent the maximum number of bytes in a multibyte character for the current locale.
For more information on NULL and the types size_t and wchar_t, see “<stddef.h>” on page 13.

### <string.h>

The `<string.h>` include file declares the string manipulation functions:

<table>
<thead>
<tr>
<th>memchr</th>
<th>strcat</th>
<th>strcspn</th>
<th>strcmp</th>
<th>setlocale</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcmp</td>
<td>strchr</td>
<td>strlen</td>
<td>strcmpi</td>
<td></td>
</tr>
<tr>
<td>memcpy</td>
<td>strcmp</td>
<td>strncasep</td>
<td>strnicmp</td>
<td></td>
</tr>
<tr>
<td>memmove</td>
<td>strcoll</td>
<td>strlen</td>
<td>strncasecmp</td>
<td></td>
</tr>
<tr>
<td>memset</td>
<td>strpbrk</td>
<td>strstr</td>
<td>strpbrk</td>
<td></td>
</tr>
<tr>
<td>memmove</td>
<td>strspn</td>
<td>strstr</td>
<td>strspn</td>
<td></td>
</tr>
<tr>
<td>memmove</td>
<td>strtok</td>
<td>strset</td>
<td>strset</td>
<td></td>
</tr>
</tbody>
</table>

Note:  
1 These functions are available for C++ programs. They are available for C only when the program defines the __cplusplus__ macro.

The `<string.h>` include file also defines the macro NULL, and the type size_t.

For more information on NULL and the type size_t, see “<stddef.h>” on page 13.

### <strings.h>

Contains the functions strcasecmp and strncasecmp

### <time.h>

The `<time.h>` include file declares the time and date functions:

<table>
<thead>
<tr>
<th>asctime</th>
<th>ctime</th>
<th>gmtime</th>
<th>localtime</th>
<th>strftime</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime_r</td>
<td>ctime_r</td>
<td>gmtime_r</td>
<td>localtime_r</td>
<td>strftime</td>
</tr>
<tr>
<td>clock</td>
<td>difftime</td>
<td>mktime</td>
<td>strptime</td>
<td></td>
</tr>
</tbody>
</table>

Note: These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

The `<time.h>` include file also provides:

- A structure `tm` that contains the components of a calendar time. See “gmtime() — Convert Time” on page 151 for a list of the `tm` structure members.
- A macro CLOCKS_PER_SEC equal to the number per second of the value that is returned by the `clock` function.
- Types clock_t, time_t, and size_t.
- The NULL pointer value.

For more information on NULL and the type size_t, see “<stddef.h>” on page 13.
The <wchar.h> header file contains declarations and definitions that are related to the manipulation of wide character strings. Any functions which deal with files are accessible if SYSIFCOPT(*IFSIO) is specified.

```
btorc1  mbren1  vswscant1  wcstoull1  wcsxfrm1
fgetwc2  mbttowc1  vswprintf1  wscant2  wcrtomb1
fgetws2  mbsinit1  wvprintf2  wprintf2  wcwidth1
fputwc2  mbsrtowcs1  wcrtomb1  wcsrtombs1  wmemchr
fputws2  putwc2  wcsctl1  wcstr  wmemcmp
fwide2  putwchar2  wcsftime1  wcstod1  wmemcpy
fwprintf2  swwprintf1  _wcscmp1  wcstok  wmemmove
fwscanf2  swscanf1  _wcsncmp1  wcstol1  wmemset
getwc2  ungetwc2  wcsptime1  wcstourl1  wmemset
getwchar2  vfwprintf2  wcsctl1  wcswidth1
```

**Note:** 1 These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Note:** 2 These functions are available only when SYSIFCOPT(*IFSIO) and LOCALETYPE(*LOCALE) are specified on the compilation command.

**Note:** 3 These functions are available only when LOCALETYPE(*LOCALEUTF) is specified on the compilation command.

The <wcstr.h> include file declares the multibyte functions:

```
wccat  wcscpy  wcsncat  wcssbrk  wcsxcs
wcschr  wcsncpy  wcsncmp  wcspan  wcsstr
wcsncmp  wmemset  wcswcat  wcsncpy
```

<wcstr.h> also defines the types size_t, NULL, and wchar_t.

For more information on NULL and the types size_t and wchar_t, see "<stddef.h>" on page 13.

The <wctype.h> header file declares the following wide character functions:

```
iswalnum  iswgraph  iswspace  towlower  wctrans
iswalpha  iswlower  iswupper  towupper
iswcntrl  iswprint  iswxdigit  towctrans
iswdigit  iswpunct  iswctype  wctype
```

The <wctype.h> header file also contains declarations and definitions for wide character classification. These declarations and definitions are not available when LOCALETYPE(*CLD) is specified on the compilation command.
The `<xxcvt.h>` include file contains the declarations that are used by the QXXDTOP, QXXDTOZ, QXXITOP, QXXITOZ, QXXPTOI, QXXPTOD, QXXZTOD, and QXXZTOI conversion functions.

The `<xxdtaa.h>` include file contains the declarations for the data area interface functions QXXCHGDA, QXXRTVDA, and the type _DTAA_NAME_T.

The definition of _DTAA_NAME_T is:
```c
typedef struct _DTAA_NAME_T {
    char dtaa_name[10];
    char dtaa_lib[10];
} _DTAA_NAME_T;
```

The `<xxenv.h>` include file contains the declarations for the QPXXCALL and QPXXDLTE EPM environment handling program. ILE procedures cannot be called from this interface.

The definition of _ENVPGM_T is:
```c
typedef struct _ENVPGM_T {
    char pgmname[10];
    char pgmlib[10];
} _ENVPGM_T;
```

The `<xxfdbk.h>` include file contains the declarations that are used by the OS/400® system feedback areas. To retrieve information from feedback areas, see "_Riofbk() — Obtain I/O Feedback Information" on page 260 and "_Ropnfbk() — Obtain Open Feedback Information" on page 275.

The following is an example of a type that is defined in the `<xxfdbk.h>` include file:
```c
typedef _Packed struct _XXIOFB_T {
    short file_dep_fb_offset;
    int write_count;
    int read_count;
    int write_read_count;
    int other_io_count;
    char reserved1;
    char cur_operation;
    char rec_format[10];
    char dev_class[2];
    char dev_name[10];
    int last_io_rec_len;
    char reserved2[80];
    short num_recs_retrieved;
    short last_io_rec_len2;
    char reserved3[2];
    int cur_blk_count;
    char reserved4[8];
} _XXIOFB_T;
```
For further information on the open feedback areas, see the File Management topic in the Information Center.

**Machine Interface (MI) Include Files**

See the *ILE C/C++ for AS/400 MI Library Reference* for a description of the MI header files.
Chapter 2. Library Functions

This chapter describes the standard C/C++ library functions and the ILE C/C++ extensions to the library functions, except for the ILE C/C++ MI functions. See the ILE C/C++ for AS/400 MI Library Reference for more information about the MI functions.

Each library function that is listed in this section contains:

• A format description that shows the include file that declares the function.
• The data type that is returned by the function.
• The required data types of the arguments to the function.

This example shows the format of the `log()` function:

```c
#include <math.h>
double log(double x);
```

The example shows that:

• you must include the file `math.h` in the program.
• the `log()` function returns type double.
• the `log()` function requires an argument `x` of type double.

Examples throughout the section illustrate the use of library functions and are not necessarily complete.

This chapter lists the library functions in alphabetic order. If you are unsure of the function you want to use, see the summary of the library functions in “The C/C++ Library.”

Note: All functions are considered threadsafe unless noted otherwise.

The C/C++ Library

This chapter summarizes the available C/C++ library functions and their location in this book. It also briefly describes what the function does. Each library function is listed according to the type of function it performs.

Error Handling

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert()</td>
<td>assert.h</td>
<td>45</td>
<td>Prints diagnostic messages.</td>
</tr>
<tr>
<td>atexit()</td>
<td>stdlib.h</td>
<td>48</td>
<td>Registers a function to be executed at program end.</td>
</tr>
<tr>
<td>clearerr()</td>
<td>stdio.h</td>
<td>64</td>
<td>Resets error indicators.</td>
</tr>
<tr>
<td>feof()</td>
<td>stdio.h</td>
<td>86</td>
<td>Tests end-of-file indicator for stream input.</td>
</tr>
<tr>
<td>ferror()</td>
<td>stdio.h</td>
<td>87</td>
<td>Tests the error indicator for a specified stream.</td>
</tr>
</tbody>
</table>
### Function Details

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_GetExcData()</code></td>
<td>signal.h</td>
<td>145</td>
<td>Retrieves information about an exception from within a C signal handler. This function is not defined when SYSIFOPT(*SYNCSIGNAL) is specified on the compilation command.</td>
</tr>
<tr>
<td>perror()</td>
<td>stdio.h</td>
<td>209</td>
<td>Prints an error message to stderr.</td>
</tr>
<tr>
<td>raise()</td>
<td>signal.h</td>
<td>238</td>
<td>Initiates a signal.</td>
</tr>
<tr>
<td>signal()</td>
<td>signal.h</td>
<td>330</td>
<td>Allows handling of an interrupt signal from the operating system.</td>
</tr>
<tr>
<td>strerror()</td>
<td>string.h</td>
<td>350</td>
<td>Retrieves pointer to system error message.</td>
</tr>
</tbody>
</table>

### Searching and Sorting

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch()</td>
<td>stdlib.h</td>
<td>54</td>
<td>Performs a binary search of a sorted array.</td>
</tr>
<tr>
<td>qsort()</td>
<td>stdlib.h</td>
<td>228</td>
<td>Performs a quick sort on an array of elements.</td>
</tr>
</tbody>
</table>

### Mathematical

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>stdlib.h</td>
<td>39</td>
<td>Calculates the absolute value of an integer.</td>
</tr>
<tr>
<td>ceil()</td>
<td>math.h</td>
<td>63</td>
<td>Calculates the double value representing the smallest integer that is greater than or equal to a number.</td>
</tr>
<tr>
<td>div()</td>
<td>stdlib.h</td>
<td>77</td>
<td>Calculates the quotient and remainder of an integer.</td>
</tr>
<tr>
<td>erf()</td>
<td>math.h</td>
<td>79</td>
<td>Calculates the error function.</td>
</tr>
<tr>
<td>erfc()</td>
<td>math.h</td>
<td>79</td>
<td>Calculates the error function for large numbers.</td>
</tr>
<tr>
<td>exp()</td>
<td>math.h</td>
<td>81</td>
<td>Calculates an exponential function.</td>
</tr>
<tr>
<td>fabs()</td>
<td>math.h</td>
<td>81</td>
<td>Calculates the absolute value of a floating-point number.</td>
</tr>
<tr>
<td>floor()</td>
<td>math.h</td>
<td>89</td>
<td>Calculates the double value representing the largest integer that is less than or equal to a number.</td>
</tr>
<tr>
<td>fmod()</td>
<td>math.h</td>
<td>89</td>
<td>Calculates the floating point remainder of one argument divided by another.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>frexp()</td>
<td>math.h</td>
<td>122</td>
<td>Separates a floating-point number into its mantissa and exponent.</td>
</tr>
<tr>
<td>gamma()</td>
<td>math.h</td>
<td>140</td>
<td>Calculates the gamma function.</td>
</tr>
<tr>
<td>hypot()</td>
<td>math.h</td>
<td>155</td>
<td>Calculates the hypotenuse.</td>
</tr>
<tr>
<td>labs()</td>
<td>stdlib.h</td>
<td>164</td>
<td>Calculates the absolute value of a long integer.</td>
</tr>
<tr>
<td>llabs()</td>
<td>stdlib.h</td>
<td>164</td>
<td>Calculates the absolute value of a long long integer.</td>
</tr>
<tr>
<td>ldexp()</td>
<td>math.h</td>
<td>166</td>
<td>Multiplies a floating-point number by an integral power of 2.</td>
</tr>
<tr>
<td>ldiv()</td>
<td>stdlib.h</td>
<td>166</td>
<td>Calculates the quotient and remainder of a long integer.</td>
</tr>
<tr>
<td>lldiv()</td>
<td>stdlib.h</td>
<td>166</td>
<td>Calculates the quotient and remainder of a long long integer.</td>
</tr>
<tr>
<td>log()</td>
<td>math.h</td>
<td>175</td>
<td>Calculates natural logarithm.</td>
</tr>
<tr>
<td>log10()</td>
<td>math.h</td>
<td>176</td>
<td>Calculates base 10 logarithm.</td>
</tr>
<tr>
<td>modf()</td>
<td>math.h</td>
<td>205</td>
<td>Calculates the signed fractional portion of the argument.</td>
</tr>
<tr>
<td>pow()</td>
<td>math.h</td>
<td>210</td>
<td>Calculates the value of an argument raised to a power.</td>
</tr>
<tr>
<td>sqrt()</td>
<td>math.h</td>
<td>337</td>
<td>Calculates the square root of a number.</td>
</tr>
</tbody>
</table>

**Trigonometric Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos()</td>
<td>math.h</td>
<td>40</td>
<td>Calculates the arc cosine.</td>
</tr>
<tr>
<td>asin()</td>
<td>math.h</td>
<td>44</td>
<td>Calculates the arc sine.</td>
</tr>
<tr>
<td>atan()</td>
<td>math.h</td>
<td>47</td>
<td>Calculates the arc tangent.</td>
</tr>
<tr>
<td>atan2()</td>
<td>math.h</td>
<td>47</td>
<td>Calculates the arc tangent.</td>
</tr>
<tr>
<td>cos()</td>
<td>math.h</td>
<td>66</td>
<td>Calculates the cosine.</td>
</tr>
<tr>
<td>cosh()</td>
<td>math.h</td>
<td>67</td>
<td>Calculates the hyperbolic cosine.</td>
</tr>
<tr>
<td>sin()</td>
<td>math.h</td>
<td>332</td>
<td>Calculates the sine.</td>
</tr>
<tr>
<td>sinh()</td>
<td>math.h</td>
<td>333</td>
<td>Calculates the hyperbolic sine.</td>
</tr>
<tr>
<td>tan()</td>
<td>math.h</td>
<td>389</td>
<td>Calculates the tangent.</td>
</tr>
<tr>
<td>tanh()</td>
<td>math.h</td>
<td>390</td>
<td>Calculates the hyperbolic tangent.</td>
</tr>
</tbody>
</table>

**Bessel Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j0()</td>
<td>math.h</td>
<td>52</td>
<td>0 order differential equation of the first kind.</td>
</tr>
<tr>
<td>j1()</td>
<td>math.h</td>
<td>52</td>
<td>1st order differential equation of the first kind.</td>
</tr>
</tbody>
</table>
### Function Headers

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jn()</td>
<td>math.h</td>
<td>32</td>
<td>$n$th order differential equation of the first kind.</td>
</tr>
<tr>
<td>y0()</td>
<td>math.h</td>
<td>32</td>
<td>0 order differential equation of the second kind.</td>
</tr>
<tr>
<td>y1()</td>
<td>math.h</td>
<td>32</td>
<td>1st order differential equation of the second kind.</td>
</tr>
<tr>
<td>yn()</td>
<td>math.h</td>
<td>32</td>
<td>$n$th order differential equation of the second kind.</td>
</tr>
</tbody>
</table>

### Time Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime()</td>
<td>time.h</td>
<td>41</td>
<td>Converts time stored as a structure to a character string in storage.</td>
</tr>
<tr>
<td>asctime_r()</td>
<td>time.h</td>
<td>43</td>
<td>Converts time stored as a structure to a character string in storage. (Restartable version of asctime())</td>
</tr>
<tr>
<td>clock()</td>
<td>time.h</td>
<td>65</td>
<td>Determines processor time.</td>
</tr>
<tr>
<td>ctime()</td>
<td>time.h</td>
<td>68</td>
<td>Converts time stored as a long value to a character string.</td>
</tr>
<tr>
<td>ctime_r()</td>
<td>time.h</td>
<td>69</td>
<td>Converts time stored as a long value to a character string. (Restartable version of ctime())</td>
</tr>
<tr>
<td>difftime()</td>
<td>time.h</td>
<td>76</td>
<td>Calculates the difference between two times.</td>
</tr>
<tr>
<td>gmtime()</td>
<td>time.h</td>
<td>151</td>
<td>Converts time to Coordinated Universal Time structure.</td>
</tr>
<tr>
<td>gmtime_r()</td>
<td>time.h</td>
<td>153</td>
<td>Converts time to Coordinated Universal Time structure. (Restartable version of gmtime())</td>
</tr>
<tr>
<td>localtime()</td>
<td>time.h</td>
<td>173</td>
<td>Converts time to local time.</td>
</tr>
<tr>
<td>localtime_r()</td>
<td>time.h</td>
<td>174</td>
<td>Converts time to local time. (Restartable version of localtime())</td>
</tr>
<tr>
<td>mktime()</td>
<td>time.h</td>
<td>204</td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>time()</td>
<td>time.h</td>
<td>391</td>
<td>Returns the time in seconds.</td>
</tr>
</tbody>
</table>

### Type Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atof()</td>
<td>stdlib.h</td>
<td>49</td>
<td>Converts a character string to a floating-point value.</td>
</tr>
<tr>
<td>atoi()</td>
<td>stdlib.h</td>
<td>50</td>
<td>Converts a character string to an integer.</td>
</tr>
<tr>
<td>atol()</td>
<td>stdlib.h</td>
<td>51</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>atoll()</td>
<td>stdlib.h</td>
<td>50</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>_gcvt()</td>
<td>stdlib.h</td>
<td>141</td>
<td>Converts a floating point value to a string.</td>
</tr>
<tr>
<td>_itoa()</td>
<td>stdlib.h</td>
<td>163</td>
<td>Converts an integer to a string.</td>
</tr>
<tr>
<td>_ltoa()</td>
<td>stdlib.h</td>
<td>177</td>
<td>Converts a long integer to a string.</td>
</tr>
<tr>
<td>strtol()</td>
<td>stdlib.h</td>
<td>380</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>strtol()</td>
<td>stdlib.h</td>
<td>380</td>
<td>Converts a character string to a long long integer.</td>
</tr>
<tr>
<td>strto1()</td>
<td>stdlib.h</td>
<td>382</td>
<td>Converts a string to an unsigned long integer.</td>
</tr>
<tr>
<td>strtoul()</td>
<td>stdlib.h</td>
<td>382</td>
<td>Converts a string to an unsigned long long integer.</td>
</tr>
<tr>
<td>toascii()</td>
<td>ctype.h</td>
<td>394</td>
<td>Converts a character to the corresponding ASCII value.</td>
</tr>
<tr>
<td>_ultoa()</td>
<td>stdlib.h</td>
<td>398</td>
<td>Converts an unsigned long integer to a string.</td>
</tr>
<tr>
<td>wcstod()</td>
<td>wchar.h</td>
<td>455</td>
<td>Converts a wide-character string to a double floating point.</td>
</tr>
<tr>
<td>wcstol()</td>
<td>wchar.h</td>
<td>458</td>
<td>Converts a wide-character string to a long integer.</td>
</tr>
<tr>
<td>wcstoll()</td>
<td>wchar.h</td>
<td>458</td>
<td>Converts a wide-character string to a long long integer.</td>
</tr>
<tr>
<td>wcstoul()</td>
<td>wchar.h</td>
<td>463</td>
<td>Converts a wide-character string to an unsigned long integer.</td>
</tr>
<tr>
<td>wcstoull()</td>
<td>wchar.h</td>
<td>463</td>
<td>Converts a wide-character string to an unsigned long long integer.</td>
</tr>
</tbody>
</table>

## Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QXXDTOP()</td>
<td>xxcvt.h</td>
<td>231</td>
<td>Converts a floating-point value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXDTOZ()</td>
<td>xxcvt.h</td>
<td>232</td>
<td>Converts a floating-point value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXITOP()</td>
<td>xxcvt.h</td>
<td>233</td>
<td>Converts an integer value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXITOZ()</td>
<td>xxcvt.h</td>
<td>233</td>
<td>Converts an integer value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXPTOD()</td>
<td>xxcvt.h</td>
<td>234</td>
<td>Converts a packed decimal value to a floating-point value.</td>
</tr>
<tr>
<td>QXXPTOI()</td>
<td>xxcvt.h</td>
<td>235</td>
<td>Converts a packed decimal value to an integer value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>QXXZTOD()</td>
<td>xxcvt.h</td>
<td>237</td>
<td>Converts a zoned decimal value to a floating-point value.</td>
</tr>
<tr>
<td>QXXZTOI()</td>
<td>xxcvt.h</td>
<td>238</td>
<td>Converts a zoned decimal value to an integer value.</td>
</tr>
</tbody>
</table>

**Record Input/Output**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Racquire()</td>
<td>recio.h</td>
<td>240</td>
<td>Prepares a device for record I/O operations.</td>
</tr>
<tr>
<td>_Rclose()</td>
<td>recio.h</td>
<td>241</td>
<td>Closes a file that is opened for record I/O operations.</td>
</tr>
<tr>
<td>_Rcommit()</td>
<td>recio.h</td>
<td>243</td>
<td>Completes the current transaction, and establishes a new commitment boundary.</td>
</tr>
<tr>
<td>_Rdelete()</td>
<td>recio.h</td>
<td>244</td>
<td>Deletes the currently locked record.</td>
</tr>
<tr>
<td>_Rdevatr()</td>
<td>recio.h</td>
<td>246</td>
<td>Returns a pointer to a copy of the device attributes feedback area for the file reference by fp and the device pgmdev.</td>
</tr>
<tr>
<td>_Rfeod()</td>
<td>recio.h</td>
<td>260</td>
<td>Forces an end-of-file condition for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rfeov()</td>
<td>recio.h</td>
<td>261</td>
<td>Forces an end-of-volume condition for tapes.</td>
</tr>
<tr>
<td>_Rformat()</td>
<td>recio.h</td>
<td>262</td>
<td>Sets the record format to fmt for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rindara()</td>
<td>recio.h</td>
<td>264</td>
<td>Sets up the separate indicator area to be used for subsequent record I/O operations.</td>
</tr>
<tr>
<td>_Riofbk()</td>
<td>recio.h</td>
<td>266</td>
<td>Returns a pointer to a copy of the I/O feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rlocate()</td>
<td>recio.h</td>
<td>268</td>
<td>Positions to the record in the files associated with fp and specified by the key, klen_rrn and opt parameters.</td>
</tr>
<tr>
<td>_Ropen()</td>
<td>recio.h</td>
<td>271</td>
<td>Opens a file for record I/O operations.</td>
</tr>
<tr>
<td>_Ropnfbk()</td>
<td>recio.h</td>
<td>275</td>
<td>Returns a pointer to a copy of the open feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rpgmdev()</td>
<td>recio.h</td>
<td>276</td>
<td>Sets the default program device.</td>
</tr>
<tr>
<td>_Rreadd()</td>
<td>recio.h</td>
<td>277</td>
<td>Reads a record by relative record number.</td>
</tr>
<tr>
<td>_Rreadf()</td>
<td>recio.h</td>
<td>279</td>
<td>Reads the first record.</td>
</tr>
<tr>
<td>_Rreadindv()</td>
<td>recio.h</td>
<td>281</td>
<td>Reads data from an invited device.</td>
</tr>
<tr>
<td>_Rreadk()</td>
<td>recio.h</td>
<td>284</td>
<td>Reads a record by key.</td>
</tr>
<tr>
<td>_Rreadl()</td>
<td>recio.h</td>
<td>288</td>
<td>Reads the last record.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>_Rreadn()</td>
<td>recio.h</td>
<td>289</td>
<td>Reads the next record.</td>
</tr>
<tr>
<td>_Rreadnc()</td>
<td>recio.h</td>
<td>292</td>
<td>Reads the next changed record in the subfile.</td>
</tr>
<tr>
<td>_Rreadp()</td>
<td>recio.h</td>
<td>293</td>
<td>Reads the previous record.</td>
</tr>
<tr>
<td>_Rreads()</td>
<td>recio.h</td>
<td>296</td>
<td>Reads the same record.</td>
</tr>
<tr>
<td>_Rrelease()</td>
<td>recio.h</td>
<td>297</td>
<td>Makes the specified device ineligible for record I/O operations.</td>
</tr>
<tr>
<td>_Rrlslck()</td>
<td>recio.h</td>
<td>299</td>
<td>Releases the currently locked record.</td>
</tr>
<tr>
<td>_Rrollback()</td>
<td>recio.h</td>
<td>300</td>
<td>Reestablishes the last commitment boundary as the current commitment boundary.</td>
</tr>
<tr>
<td>_Rupdate()</td>
<td>recio.h</td>
<td>302</td>
<td>Writes to the record that is currently locked for update.</td>
</tr>
<tr>
<td>_Rupfb()</td>
<td>recio.h</td>
<td>304</td>
<td>Updates the feedback structure with information about the last record I/O operation.</td>
</tr>
<tr>
<td>_Rwrite()</td>
<td>recio.h</td>
<td>305</td>
<td>Writes a record to the end of the file.</td>
</tr>
<tr>
<td>_Rwritten()</td>
<td>recio.h</td>
<td>307</td>
<td>Writes a record by relative record number. It will only write over deleted records.</td>
</tr>
<tr>
<td>_Rwriterd()</td>
<td>recio.h</td>
<td>310</td>
<td>Writes and reads a record.</td>
</tr>
<tr>
<td>_Rrrread()</td>
<td>recio.h</td>
<td>311</td>
<td>Functions as _Rwriterd(), except separate buffers may be specified for input and output data.</td>
</tr>
</tbody>
</table>

**Stream Input/Output**

*Formatted Input/Output*

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf()</td>
<td>stdio.h</td>
<td>108</td>
<td>Formats and prints characters to the output stream.</td>
</tr>
<tr>
<td>fscanf()</td>
<td>stdio.h</td>
<td>123</td>
<td>Reads data from a stream into locations given by arguments.</td>
</tr>
<tr>
<td>fwprintf()</td>
<td>stdio.h</td>
<td>133</td>
<td>Formats data as wide characters, and writes to a stream.</td>
</tr>
<tr>
<td>fwscanf()</td>
<td>stdio.h</td>
<td>137</td>
<td>Reads wide data from stream into locations given by arguments.</td>
</tr>
<tr>
<td>printf()</td>
<td>stdio.h</td>
<td>211</td>
<td>Formats and prints characters to stdout.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>scanf()</td>
<td>stdio.h</td>
<td>313</td>
<td>Reads data from stdin into locations given by arguments.</td>
</tr>
<tr>
<td>snprintf()</td>
<td>stdio.h</td>
<td>334</td>
<td>Same as sprintf, except that the snprintf() function will stop after ( n ) characters have been written to a buffer.</td>
</tr>
<tr>
<td>sprintf()</td>
<td>stdio.h</td>
<td>335</td>
<td>Formats and writes characters to a buffer.</td>
</tr>
<tr>
<td>sscanf()</td>
<td>stdio.h</td>
<td>339</td>
<td>Reads data from a buffer into locations given by arguments.</td>
</tr>
<tr>
<td>swprintf()</td>
<td>wchar.h</td>
<td>385</td>
<td>Formats and writes wide characters to buffer.</td>
</tr>
<tr>
<td>swscanf()</td>
<td>wchar.h</td>
<td>387</td>
<td>Reads wide data from a buffer into locations given by arguments.</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>stdio.h</td>
<td>404</td>
<td>Formats and prints characters to the output stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfscanf()</td>
<td>stdarg.h</td>
<td>406</td>
<td>Reads data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vfwprintf()</td>
<td>stdio.h</td>
<td>407</td>
<td>Formats argument data as wide characters and writes to a stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfwscanf()</td>
<td>stdarg.h</td>
<td>409</td>
<td>Reads wide data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
<td>------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>vprintf()</td>
<td>stdarg.h stdio.h</td>
<td>411</td>
<td>Formats and writes characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>vscanf()</td>
<td>stdarg.h stdio.h</td>
<td>412</td>
<td>Reads data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vsnprintf()</td>
<td>stdio.h stdarg.h</td>
<td>414</td>
<td>Same as vprintf, except that the vsnprintf function will stop after n characters have been written to a buffer.</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>stdarg.h stdio.h</td>
<td>415</td>
<td>Formats and writes characters to a buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vsscanf()</td>
<td>stdarg.h stdio.h</td>
<td>416</td>
<td>Reads data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vswprintf()</td>
<td>wchar.h stdarg.h</td>
<td>418</td>
<td>Formats and writes wide characters to buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vswscanf()</td>
<td>wchar.h stdarg.h</td>
<td>412</td>
<td>Reads wide data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vwprintf()</td>
<td>wchar.h stdarg.h</td>
<td>422</td>
<td>Formats and writes wide characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>vwscanf()</td>
<td>stdarg.h stdio.h</td>
<td>423</td>
<td>Reads wide data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>wprintf()</td>
<td>stdio.h</td>
<td>481</td>
<td>Formats and writes wide characters to stdout.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wscanf()</td>
<td>stdio.h</td>
<td>182</td>
<td>Reads wide data from stdin into locations given by arguments.</td>
</tr>
</tbody>
</table>

**Character and String Input/Output**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetc()</td>
<td>stdio.h</td>
<td>89</td>
<td>Reads a character from a specified input stream.</td>
</tr>
<tr>
<td>fgets()</td>
<td>stdio.h</td>
<td>92</td>
<td>Reads a string from a specified input stream.</td>
</tr>
<tr>
<td>fgetwc()</td>
<td>stdio.h</td>
<td>94</td>
<td>Reads a wide character from a specified stream.</td>
</tr>
<tr>
<td>fgets()</td>
<td>stdio.h</td>
<td>96</td>
<td>Reads a wide-character string from a specified stream.</td>
</tr>
<tr>
<td>fputc()</td>
<td>stdio.h</td>
<td>109</td>
<td>Prints a character to a specified output stream.</td>
</tr>
<tr>
<td>_fputchar()</td>
<td>stdio.h</td>
<td>111</td>
<td>Writes a character to stdout.</td>
</tr>
<tr>
<td>fputs()</td>
<td>stdio.h</td>
<td>112</td>
<td>Prints a string to a specified output stream.</td>
</tr>
<tr>
<td>fputwc()</td>
<td>stdio.h</td>
<td>113</td>
<td>Writes a wide character to a specified stream.</td>
</tr>
<tr>
<td>fputws()</td>
<td>stdio.h</td>
<td>115</td>
<td>Writes a wide-character string to a specified stream.</td>
</tr>
<tr>
<td>getc()</td>
<td>stdio.h</td>
<td>142</td>
<td>Reads a character from a specified input stream.</td>
</tr>
<tr>
<td>getchar()</td>
<td>stdio.h</td>
<td>142</td>
<td>Reads a character from stdin.</td>
</tr>
<tr>
<td>gets()</td>
<td>stdio.h</td>
<td>146</td>
<td>Reads a line from stdin.</td>
</tr>
<tr>
<td>getwc()</td>
<td>stdio.h</td>
<td>147</td>
<td>Reads a wide character from a specified stream.</td>
</tr>
<tr>
<td>getwchar()</td>
<td>stdio.h</td>
<td>149</td>
<td>Gets a wide character from stdin.</td>
</tr>
<tr>
<td>putc()</td>
<td>stdio.h</td>
<td>221</td>
<td>Prints a character to a specified output stream.</td>
</tr>
<tr>
<td>putchar()</td>
<td>stdio.h</td>
<td>221</td>
<td>Prints a character to stdout.</td>
</tr>
<tr>
<td>puts()</td>
<td>stdio.h</td>
<td>224</td>
<td>Prints a string to stdout.</td>
</tr>
<tr>
<td>putwc()</td>
<td>stdio.h</td>
<td>225</td>
<td>Writes a wide character to a specified stream.</td>
</tr>
<tr>
<td>putwchar()</td>
<td>stdio.h</td>
<td>227</td>
<td>Writes a wide character to stdout.</td>
</tr>
<tr>
<td>ungetc()</td>
<td>stdio.h</td>
<td>399</td>
<td>Pushes a character back onto a specified input stream.</td>
</tr>
<tr>
<td>ungetwc()</td>
<td>stdio.h</td>
<td>401</td>
<td>Pushes a wide character back onto a specified input stream.</td>
</tr>
</tbody>
</table>
Direct Input/Output

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fread()</td>
<td>stdio.h</td>
<td>116</td>
<td>Reads items from a specified input stream.</td>
</tr>
<tr>
<td>fwrite()</td>
<td>stdio.h</td>
<td>136</td>
<td>Writes items to a specified output stream.</td>
</tr>
</tbody>
</table>

File Positioning

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetpos()</td>
<td>stdio.h</td>
<td>91</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>fseek()</td>
<td>stdio.h</td>
<td>124</td>
<td>Moves the file pointer to a new location.</td>
</tr>
<tr>
<td>fseeko()</td>
<td>stdio.h</td>
<td>124</td>
<td>Same as fseek().</td>
</tr>
<tr>
<td>fsetpos()</td>
<td>stdio.h</td>
<td>126</td>
<td>Moves the file pointer to a new location.</td>
</tr>
<tr>
<td>ftell()</td>
<td>stdio.h</td>
<td>128</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>ftello()</td>
<td>stdio.h</td>
<td>128</td>
<td>Same as ftell().</td>
</tr>
<tr>
<td>rewind()</td>
<td>stdio.h</td>
<td>259</td>
<td>Repositions the file pointer to the beginning of the file.</td>
</tr>
</tbody>
</table>

File Access

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclose()</td>
<td>stdio.h</td>
<td>82</td>
<td>Closes a specified stream.</td>
</tr>
<tr>
<td>fdopen()</td>
<td>stdio.h</td>
<td>83</td>
<td>Associates an input or output stream with a file.</td>
</tr>
<tr>
<td>fflush()</td>
<td>stdio.h</td>
<td>87</td>
<td>Causes the system to write the contents of a buffer to a file.</td>
</tr>
<tr>
<td>fopen()</td>
<td>stdio.h</td>
<td>100</td>
<td>Opens a specified stream.</td>
</tr>
<tr>
<td>freopen()</td>
<td>stdio.h</td>
<td>120</td>
<td>Closes a file and reassigns a stream.</td>
</tr>
<tr>
<td>fwide()</td>
<td>stdio.h</td>
<td>130</td>
<td>Determines stream orientation.</td>
</tr>
<tr>
<td>setbuf()</td>
<td>stdio.h</td>
<td>319</td>
<td>Allows control of buffering.</td>
</tr>
<tr>
<td>setvbuf()</td>
<td>stdio.h</td>
<td>328</td>
<td>Controls buffering and buffer size for a specified stream.</td>
</tr>
<tr>
<td>wfopen()</td>
<td>stdio.h</td>
<td>476</td>
<td>Opens a specified stream, accepting file name and mode as wide characters.</td>
</tr>
</tbody>
</table>
### File Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileno()</td>
<td>stdio.h</td>
<td>98</td>
<td>Determines the file handle.</td>
</tr>
<tr>
<td>remove()</td>
<td>stdio.h</td>
<td>257</td>
<td>Deletes a specified file.</td>
</tr>
<tr>
<td>rename()</td>
<td>stdio.h</td>
<td>257</td>
<td>Renames a specified file.</td>
</tr>
<tr>
<td>tmpfile()</td>
<td>stdio.h</td>
<td>392</td>
<td>Creates a temporary file and returns a pointer to that file.</td>
</tr>
<tr>
<td>tmpnam()</td>
<td>stdio.h</td>
<td>393</td>
<td>Produces a temporary file name.</td>
</tr>
</tbody>
</table>

### Handling Argument Lists

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_arg()</td>
<td>stdarg.h</td>
<td>402</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
<tr>
<td>va_end()</td>
<td>stdarg.h</td>
<td>402</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
<tr>
<td>va_start()</td>
<td>stdarg.h</td>
<td>402</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
</tbody>
</table>

### Pseudorandom Numbers

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand(), rand_r()</td>
<td>stdlib.h</td>
<td>338</td>
<td>Returns a pseudorandom integer. (rand_r() is the restartable version of rand().)</td>
</tr>
<tr>
<td>srand()</td>
<td>stdlib.h</td>
<td>338</td>
<td>Sets the starting point for pseudorandom numbers.</td>
</tr>
</tbody>
</table>

### Dynamic Memory Management

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calloc()</td>
<td>stdlib.h</td>
<td>57</td>
<td>Reserves storage space for an array and initializes the values of all elements to 0.</td>
</tr>
<tr>
<td>_C_TS_malloc_debug</td>
<td>mallocinfo.h</td>
<td>71</td>
<td>Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by malloc functions when compiled with teraspace.</td>
</tr>
<tr>
<td>_C_TS_malloc_info</td>
<td>mallocinfo.h</td>
<td>73</td>
<td>Returns the current memory usage information.</td>
</tr>
<tr>
<td>free()</td>
<td>stdlib.h</td>
<td>118</td>
<td>Frees storage blocks.</td>
</tr>
<tr>
<td>malloc()</td>
<td>stdlib.h</td>
<td>180</td>
<td>Reserves storage blocks.</td>
</tr>
<tr>
<td>realloc()</td>
<td>stdlib.h</td>
<td>247</td>
<td>Changes storage size allocated for an object.</td>
</tr>
</tbody>
</table>
### Memory Objects

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr()</td>
<td>string.h</td>
<td>197</td>
<td>Searches a buffer for the first occurrence of a given character.</td>
</tr>
<tr>
<td>memcmp()</td>
<td>string.h</td>
<td>198</td>
<td>Compares two buffers.</td>
</tr>
<tr>
<td>memcpy()</td>
<td>string.h</td>
<td>199</td>
<td>Copies a buffer.</td>
</tr>
<tr>
<td>memicmp()</td>
<td>string.h</td>
<td>200</td>
<td>Compare two buffers without regard to case.</td>
</tr>
<tr>
<td>memmove()</td>
<td>string.h</td>
<td>202</td>
<td>Moves a buffer.</td>
</tr>
<tr>
<td>memset()</td>
<td>string.h</td>
<td>203</td>
<td>Sets a buffer to a given value.</td>
</tr>
<tr>
<td>wmemchr()</td>
<td>wchar.h</td>
<td>476</td>
<td>Locates a wide character in a wide-character buffer.</td>
</tr>
<tr>
<td>wmemcmp()</td>
<td>wchar.h</td>
<td>477</td>
<td>Compares two wide-character buffers.</td>
</tr>
<tr>
<td>wmemcpy()</td>
<td>wchar.h</td>
<td>478</td>
<td>Copies a wide-character buffer.</td>
</tr>
<tr>
<td>wmemmove()</td>
<td>wchar.h</td>
<td>479</td>
<td>Moves a wide-character buffer.</td>
</tr>
<tr>
<td>wmemset()</td>
<td>wchar.h</td>
<td>480</td>
<td>Sets a wide-character buffer to a given value.</td>
</tr>
</tbody>
</table>

### Environment Interaction

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort()</td>
<td>stdlib.h</td>
<td>38</td>
<td>Ends a program abnormally.</td>
</tr>
<tr>
<td>_C_Get_Ssn_Handle()</td>
<td>stdio.h</td>
<td>57</td>
<td>Returns a handle to the C session for use with DSM APIs.</td>
</tr>
<tr>
<td>exit()</td>
<td>stdlib.h</td>
<td>80</td>
<td>Ends the program normally if called in the initial thread.</td>
</tr>
<tr>
<td>getenv()</td>
<td>stdlib.h</td>
<td>144</td>
<td>Searches environment variables for a specified variable.</td>
</tr>
<tr>
<td>localeconv()</td>
<td>locale.h</td>
<td>168</td>
<td>Formats numeric quantities in struct lconv according to the current locale.</td>
</tr>
<tr>
<td>longjmp()</td>
<td>setjmp.h</td>
<td>178</td>
<td>Restores a stack environment.</td>
</tr>
<tr>
<td>nl_langinfo()</td>
<td>langinfo.h</td>
<td>206</td>
<td>Retrieves information from the current locale.</td>
</tr>
<tr>
<td>putenv()</td>
<td>stdlib.h</td>
<td>223</td>
<td>Sets the value of an environment variable by altering an existing variable or creating a new one.</td>
</tr>
<tr>
<td>setjmp()</td>
<td>setjmp.h</td>
<td>321</td>
<td>Saves a stack environment.</td>
</tr>
</tbody>
</table>
**Function** | **Header File** | **Page** | **Description**
--- | --- | --- | ---
setlocale() | locale.h | 322 | Changes or queries locale.
system() | stdlib.h | 388 | Passes a string to the operating system’s command interpreter.
wcslocaleconv() | locale.h | 441 | Formats numeric quantities in struct wcslconv according to the current locale.

---

### String Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcasecmp()</td>
<td>strings.h</td>
<td>340</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strcat()</td>
<td>string.h</td>
<td>341</td>
<td>Concatenates two strings.</td>
</tr>
<tr>
<td>strchr()</td>
<td>string.h</td>
<td>342</td>
<td>Locates the first occurrence of a specified character in a string.</td>
</tr>
<tr>
<td>strcmp()</td>
<td>string.h</td>
<td>343</td>
<td>Compares the value of two strings.</td>
</tr>
<tr>
<td>strcmpi()</td>
<td>string.h</td>
<td>345</td>
<td>Compares the value of two strings without regard to case.</td>
</tr>
<tr>
<td>strcoll()</td>
<td>string.h</td>
<td>346</td>
<td>Compares the locale-defined value of two strings.</td>
</tr>
<tr>
<td>strcpy()</td>
<td>string.h</td>
<td>347</td>
<td>Copies one string into another.</td>
</tr>
<tr>
<td>strcspn()</td>
<td>string.h</td>
<td>348</td>
<td>Finds the length of the first substring in a string of characters not in a second string.</td>
</tr>
<tr>
<td>strdup()</td>
<td>string.h</td>
<td>350</td>
<td>Duplicates a string.</td>
</tr>
<tr>
<td>strftime()</td>
<td>time.h</td>
<td>351</td>
<td>Converts date and time to a formatted string.</td>
</tr>
<tr>
<td>stricmp()</td>
<td>string.h</td>
<td>357</td>
<td>Compares the value of two strings without regard to case.</td>
</tr>
<tr>
<td>strlen()</td>
<td>string.h</td>
<td>358</td>
<td>Calculates the length of a string.</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>strings.h</td>
<td>359</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strncat()</td>
<td>string.h</td>
<td>361</td>
<td>Adds a specified length of one string to another string.</td>
</tr>
<tr>
<td>strncmp()</td>
<td>string.h</td>
<td>362</td>
<td>Compares two strings up to a specified length.</td>
</tr>
<tr>
<td>strncpy()</td>
<td>string.h</td>
<td>364</td>
<td>Copies a specified length of one string into another.</td>
</tr>
<tr>
<td>strnicmp()</td>
<td>string.h</td>
<td>365</td>
<td>Compares the value of two substrings without regard to case.</td>
</tr>
<tr>
<td>strnset()</td>
<td>string.h</td>
<td>366</td>
<td>Sets character in a string.</td>
</tr>
<tr>
<td>strpbrk()</td>
<td>string.h</td>
<td>367</td>
<td>Locates specified characters in a string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>strftime()</td>
<td>time.h</td>
<td>369</td>
<td>Converts string to formatted time.</td>
</tr>
<tr>
<td>strrchr()</td>
<td>string.h</td>
<td>373</td>
<td>Locates the last occurrence of a character within a string.</td>
</tr>
<tr>
<td>strchr()</td>
<td>string.h</td>
<td>374</td>
<td>Locates the first character in a string that is not part of specified set of characters.</td>
</tr>
<tr>
<td>strstr()</td>
<td>string.h</td>
<td>375</td>
<td>Locates the first occurrence of a string in another string.</td>
</tr>
<tr>
<td>strtok()</td>
<td>string.h</td>
<td>378</td>
<td>Locates a specified token in a string.</td>
</tr>
<tr>
<td>strtok_r()</td>
<td>string.h</td>
<td>379</td>
<td>Locates a specified token in a string. (Restartable version of strtok()).</td>
</tr>
<tr>
<td>strxfrm()</td>
<td>string.h</td>
<td>384</td>
<td>Transforms strings according to locale.</td>
</tr>
<tr>
<td>wcsftime()</td>
<td>wchar.h</td>
<td>437</td>
<td>Converts to formatted date and time.</td>
</tr>
<tr>
<td>wcsptime()</td>
<td>wchar.h</td>
<td>448</td>
<td>Converts string to formatted time.</td>
</tr>
<tr>
<td>wcsstr()</td>
<td>wchar.h</td>
<td>455</td>
<td>Locates a wide-character substring.</td>
</tr>
<tr>
<td>wcstok()</td>
<td>wchar.h</td>
<td>457</td>
<td>Tokenizes a wide-character string.</td>
</tr>
</tbody>
</table>

### Character Testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for alphanumeric characters.</td>
</tr>
<tr>
<td>isalpha()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for alphabetic characters.</td>
</tr>
<tr>
<td>isascii()</td>
<td>ctype.h</td>
<td>158</td>
<td>Tests for ASCII values.</td>
</tr>
<tr>
<td>isblank()</td>
<td>ctype.h</td>
<td>159</td>
<td>Tests for blank or tab characters.</td>
</tr>
<tr>
<td>iscntrl()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for control characters.</td>
</tr>
<tr>
<td>isdigit()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for decimal digits.</td>
</tr>
<tr>
<td>isgraph()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for printable characters excluding the space.</td>
</tr>
<tr>
<td>islower()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for lowercase letters.</td>
</tr>
<tr>
<td>isprint()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for printable characters including the space.</td>
</tr>
<tr>
<td>ispunct()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for punctuation characters as defined in the locale.</td>
</tr>
<tr>
<td>isspace()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for white-space characters.</td>
</tr>
<tr>
<td>isupper()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for uppercase letters.</td>
</tr>
<tr>
<td>isxdigit()</td>
<td>ctype.h</td>
<td>156</td>
<td>Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.</td>
</tr>
</tbody>
</table>
# Multibyte Character Testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswalnum()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide alphanumeric characters.</td>
</tr>
<tr>
<td>iswalpha()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide alphabetic characters.</td>
</tr>
<tr>
<td>iswcntrl()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide control characters.</td>
</tr>
<tr>
<td>iswctype()</td>
<td>wctype.h</td>
<td>162</td>
<td>Tests for character property.</td>
</tr>
<tr>
<td>iswdigit()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide decimal digits.</td>
</tr>
<tr>
<td>iswgraph()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide printing characters excluding the space.</td>
</tr>
<tr>
<td>iswlower()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide lowercase letters.</td>
</tr>
<tr>
<td>iswprint()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide printing characters.</td>
</tr>
<tr>
<td>iswpunct()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide punctuation characters as defined in the locale.</td>
</tr>
<tr>
<td>iswspace()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide whitespace characters.</td>
</tr>
<tr>
<td>iswupper()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide uppercase letters.</td>
</tr>
<tr>
<td>iswxdigit()</td>
<td>wctype.h</td>
<td>160</td>
<td>Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.</td>
</tr>
</tbody>
</table>

# Character Case Mapping

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolower()</td>
<td>ctype.h</td>
<td>395</td>
<td>Converts a character to lowercase.</td>
</tr>
<tr>
<td>toupper()</td>
<td>ctype.h</td>
<td>395</td>
<td>Converts a character to uppercase.</td>
</tr>
<tr>
<td>towlower()</td>
<td>ctype.h</td>
<td>397</td>
<td>Converts a wide character to lowercase.</td>
</tr>
<tr>
<td>towupper()</td>
<td>ctype.h</td>
<td>397</td>
<td>Converts a wide character to uppercase.</td>
</tr>
</tbody>
</table>

# Multibyte Character Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>btowc()</td>
<td>stdio.h</td>
<td>35</td>
<td>Converts a single byte to a wide character.</td>
</tr>
<tr>
<td>mblen()</td>
<td>stdlib.h</td>
<td>182</td>
<td>Determines the length of a multibyte character.</td>
</tr>
<tr>
<td>mbrlen()</td>
<td>stdlib.h</td>
<td>183</td>
<td>Determines the length of a multibyte character. (Restartable version of mblen())</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>stdlib.h</td>
<td>186</td>
<td>Converts a multibyte character to a wide character. (Restartable version of mbtowc())</td>
</tr>
<tr>
<td>mbsinit()</td>
<td>stdlib.h</td>
<td>190</td>
<td>Tests state object for initial state.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mbstowcs()</td>
<td>stdlib.h</td>
<td>191</td>
<td>Converts a multibyte string to a wide character string. (Restartable version of mbstowcs())</td>
</tr>
<tr>
<td>mbstowcs()</td>
<td>stdlib.h</td>
<td>192</td>
<td>Converts a multibyte string to a wide character string.</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>stdlib.h</td>
<td>196</td>
<td>Converts multibyte characters to a wide character.</td>
</tr>
<tr>
<td>towctrans()</td>
<td>wctype.h</td>
<td>396</td>
<td>Translates wide character.</td>
</tr>
<tr>
<td>wctomb()</td>
<td>stdlib.h</td>
<td>425</td>
<td>Converts a wide character to a multibyte character. (Restartable version of wctomb())</td>
</tr>
<tr>
<td>wcscat()</td>
<td>wcstr.h</td>
<td>430</td>
<td>Concatenates wide character strings.</td>
</tr>
<tr>
<td>wcschr()</td>
<td>wcstr.h</td>
<td>431</td>
<td>Searches a wide character string for a wide character.</td>
</tr>
<tr>
<td>wcscmp()</td>
<td>wcstr.h</td>
<td>433</td>
<td>Compares two wide character strings.</td>
</tr>
<tr>
<td>wcscoll()</td>
<td>wcstr.h</td>
<td>434</td>
<td>Compares the locale-defined value of two wide-character strings.</td>
</tr>
<tr>
<td>wcscpy()</td>
<td>wcstr.h</td>
<td>435</td>
<td>Copies a wide character string.</td>
</tr>
<tr>
<td>wcscspn()</td>
<td>wcstr.h</td>
<td>436</td>
<td>Searches a wide character string for characters.</td>
</tr>
<tr>
<td>__wcsicmp()</td>
<td>wcstr.h</td>
<td>439</td>
<td>Compares two wide character strings without regard to case.</td>
</tr>
<tr>
<td>wcslen()</td>
<td>wcstr.h</td>
<td>440</td>
<td>Finds length of a wide character string.</td>
</tr>
<tr>
<td>wcscatn()</td>
<td>wcstr.h</td>
<td>442</td>
<td>Concatenates a wide character string segment.</td>
</tr>
<tr>
<td>wcscmpn()</td>
<td>wcstr.h</td>
<td>443</td>
<td>Compares wide character string segments.</td>
</tr>
<tr>
<td>wcscpy()</td>
<td>wcstr.h</td>
<td>445</td>
<td>Copies wide character string segments.</td>
</tr>
<tr>
<td>__wcsnicmp()</td>
<td>wcstr.h</td>
<td>446</td>
<td>Compares two wide character substrings without regard to case.</td>
</tr>
<tr>
<td>wcspbrk()</td>
<td>wcstr.h</td>
<td>447</td>
<td>Locates wide characters in string.</td>
</tr>
<tr>
<td>wcsrchr()</td>
<td>wcstr.h</td>
<td>450</td>
<td>Locates wide character in string.</td>
</tr>
<tr>
<td>wcsrtombs()</td>
<td>stdlib.h</td>
<td>451</td>
<td>Converts a wide character string to a multibyte character string. (Restartable version of wcsrtombs())</td>
</tr>
<tr>
<td>wcsspn()</td>
<td>wcstr.h</td>
<td>453</td>
<td>Finds offset of first non-matching wide character.</td>
</tr>
<tr>
<td>wcstombs()</td>
<td>stdlib.h</td>
<td>460</td>
<td>Converts a wide character string to a multibyte character string.</td>
</tr>
<tr>
<td>wcswcs()</td>
<td>wcstr.h</td>
<td>465</td>
<td>Locates a wide character string in another wide character string.</td>
</tr>
<tr>
<td>wcswidth()</td>
<td>wchar.h</td>
<td>466</td>
<td>Determines the display width of a wide character string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>wcsxfrm()</td>
<td>wcstr.h</td>
<td>167</td>
<td>Transforms wide-character strings according to locale.</td>
</tr>
<tr>
<td>wctob()</td>
<td>stdlib.h</td>
<td>168</td>
<td>Converts a wide character to a single byte.</td>
</tr>
<tr>
<td>wctomb()</td>
<td>stdlib.h</td>
<td>169</td>
<td>Converts a wide character to multibyte characters.</td>
</tr>
<tr>
<td>wctrans()</td>
<td>wctype.h</td>
<td>170</td>
<td>Gets a handle for character mapping.</td>
</tr>
<tr>
<td>wctype()</td>
<td>wchar.h</td>
<td>172</td>
<td>Obtains a handle for character property classification.</td>
</tr>
<tr>
<td>wcwidth()</td>
<td>wchar.h</td>
<td>175</td>
<td>Determines the display width of a wide character.</td>
</tr>
</tbody>
</table>

**Data Areas**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QXXCHGDA()</td>
<td>xxdtaa.h</td>
<td>230</td>
<td>Changes the data area.</td>
</tr>
<tr>
<td>QXXRTVDA()</td>
<td>xxdtaa.h</td>
<td>236</td>
<td>Retrieves a copy of the data area specified by dname.</td>
</tr>
</tbody>
</table>

**Message Catalogs**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>catclose()</td>
<td>nl_types.h</td>
<td>59</td>
<td>Closes a message catalog.</td>
</tr>
<tr>
<td>catgets()</td>
<td>nl_types.h</td>
<td>60</td>
<td>Reads a message from an opened message catalog.</td>
</tr>
<tr>
<td>catopen()</td>
<td>nl_types.h</td>
<td>61</td>
<td>Opens a message catalog.</td>
</tr>
</tbody>
</table>

**Regular Expression**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regcomp()</td>
<td>regex.h</td>
<td>250</td>
<td>Compiles a regular expression.</td>
</tr>
<tr>
<td>regerror()</td>
<td>regex.h</td>
<td>252</td>
<td>Returns error message for regular expression.</td>
</tr>
<tr>
<td>regexec()</td>
<td>regex.h</td>
<td>253</td>
<td>Executes a compiled regular expression.</td>
</tr>
<tr>
<td>regfree()</td>
<td>regex.h</td>
<td>256</td>
<td>Frees memory for regular expression.</td>
</tr>
</tbody>
</table>

**abort() — Stop a Program**

**Format**

```c
#include <stdlib.h>
void abort(void);
```

**Language Level**: ANSI
Threadsafe: Yes.

Description

The abort() function causes an abnormal end of the program and returns control to the host environment. Like the exit() function, the abort() function deletes buffers and closes open files before ending the program.

Calls to the abort() function raise the SIGABRT signal. The abort() function will not result in the ending of the program if SIGABRT is caught by a signal handler, and the signal handler does not return.

Note: When compiled with SYSIFCOPT(*ASYNCSIGNAL), the abort() function cannot be called in a signal handler.

Return Value

There is no return value.

Example that uses abort()

This example tests for successful opening of the file myfile. If an error occurs, an error message is printed, and the program ends with a call to the abort() function.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *stream;

    if ((stream = fopen("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        abort();
    }
}
```

Related Information

- “exit() — End Program” on page 80
- “signal() — Handle Interrupt Signals” on page 330
- “<stdlib.h>” on page 15
- See the signal() API in the APIs topic in the iSeries Information Center.

abs() — Calculate Integer Absolute Value

Format

```c
#include <stdlib.h>
int abs(int n);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The abs() function returns the absolute value of an integer argument n.
Return Value

There is no error return value. The result is undefined when the absolute value of the argument cannot be represented as an integer. The value of the minimum allowable integer is defined by INT_MIN in the <limits.h> include file.

Example that uses abs()

This example calculates the absolute value of an integer x and assigns it to y.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int x = -4, y;
    y = abs(x);
    printf("The absolute value of x is %d.\n", y);
    /*********** Output ************
    The absolute value of x is 4.
    ***********************************/
}
```

Related Information

- “fabs() — Calculate Floating-Point Absolute Value” on page 81
- “labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 164
- “<limits.h>” on page 7
- “<stdlib.h>” on page 15

acos() — Calculate Arccosine

Format

```c
#include <math.h>
double acos(double x);
```

Language Level: ANSI

Threading Safe: Yes.

Description

The acos() function calculates the arccosine of x, expressed in radians, in the range 0 to π.

Return Value

The acos() function returns the arccosine of x. The value of x must be between -1 and 1 inclusive. If x is less than -1 or greater than 1, acos() sets errno to EDOM and returns 0.

Example that uses acos()

This example prompts for a value for x. It prints an error message if x is greater than 1 or less than -1; otherwise, it assigns the arccosine of x to y.
```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX 1.0
#define MIN -1.0

int main(void)
{
    double x, y;

    printf( "Enter x
" );
    scanf( "%lf", &x );

    /* Output error if not in range */
    if ( x > MAX )
        printf( "Error: %lf too large for acos\n", x );
    else if ( x < MIN )
        printf( "Error: %lf too small for acos\n", x );
    else {
        y = acos( x );
        printf( "acos( %lf ) = %lf\n", x, y );
    }
}

/******  Expected output if 0.4 is entered:  *********
Enter x
acos( 0.400000 ) = 1.159279
*/
```

Related Information
- "asin() — Calculate Arcsine" on page 44
- "atan() — atan2() — Calculate Arctangent" on page 47
- "cos() — Calculate Cosine" on page 66
- "cosh() — Calculate Hyperbolic Cosine" on page 67
- "sin() — Calculate Sine" on page 332
- "sinh() — Calculate Hyperbolic Sine" on page 333
- "tan() — Calculate Tangent" on page 389
- "tanh() — Calculate Hyperbolic Tangent" on page 390
- "<math.h>" on page 7

### asctime() — Convert Time to Character String

#### Format
```
#include <time.h>
char *asctime(const struct tm *time);
```

#### Language Level: ANSI

#### Threadsafe: No. Use asctime_r() instead.

#### Description
The asctime() function converts time, stored as a structure pointed to by `time`, to a character string. You can obtain the `time` value from a call to the `gmtime()` function or the `localtime()` function; either returns a pointer to a `tm` structure defined in `<time.h>`.
The string result that `asctime()` produces contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

The following are examples of the string returned:
```
Sat Jul 16 02:03:55 1994\n0
```
or
```
Sat Jul 16  2:03:55 1994\n0
```

The `asctime()` function uses a 24-hour-clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

**Return Value**

The `asctime()` function returns a pointer to the resulting character string. If the function is unsuccessful, it returns NULL.

**Note:** The `asctime()`, `ctime()` functions, and other time functions may use a common, statically allocated buffer to hold the return string. Each call to one of these functions may destroy the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically-allocated buffer to hold the return string. These functions can be used in the place of the `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions if reentrancy is desired.

**Example that uses asctime()**

This example polls the system clock and prints a message that gives the current time.
```
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

    /* Get the time in seconds */
    time(&ltime);
    /* Convert it to the structure tm */
    newtime = localtime(&ltime);

    /* Print the local time as a string */
    printf("The current date and time are %s",
            asctime(newtime));
}
```

****** Output should be similar to: ******
The current date and time are Fri Sep 16 13:29:51 1994

**Related Information**

- ["asctime_r() — Convert Time to Character String (Restartable)" on page 43](#)
asctime_r() — Convert Time to Character String (Restartable)

Format

```
#include <time.h>
char *asctime_r(const struct tm *tm, char *buf);
```

Language Level: XPG4

Threading: Yes.

Description

This function is the restartable version of the asctime() function.

The asctime_r() function converts time, stored as a structure pointed to by tm, to a character string. You can obtain the tm value from a call to gmtime_r() or localtime_r(); either returns a pointer to a tm structure defined in <time.h>.

The string result that asctime_r() produces contains exactly 26 characters and has the format:

```
%.3s %.3s%3d %.2d:%.2d:%.2d %d
```

The following are examples of the string returned:

- Sat Jul 16 02:03:55 1994
- Sat Jul 16  2:03:55 1994

The asctime_r() function uses a 24-hour-clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Return Value

The asctime_r() function returns a pointer to the resulting character string. If the function is unsuccessful, it returns NULL.
Example that uses asctime_r()

This example polls the system clock and prints a message giving the current time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;
    char  mybuf[50];

    /* Get the time in seconds */
    time(&ltime);
    /* Convert it to the structure tm */
    newtime = localtime_r(&ltime);
    /* Print the local time as a string */
    printf("The current date and time are \%s",
        asctime_r(newtime, mybuf));
}
```

/*************** Output should be similar to ******************
The current date and time are Fri Sep 16 132951 1994
*/

Related Information

- “asctime() — Convert Time to Character String” on page 41
- “ctime() — Convert Time to Character String” on page 68
- “ctime_r() — Convert Time to Character String (Restartable)” on page 69
- “gmtime() — Convert Time” on page 151
- “gmtime_r() — Convert Time (Restartable)” on page 153
- “localtime() — Convert Time” on page 173
- “localtime_r() — Convert Time (Restartable)” on page 174
- “mktime() — Convert Local Time” on page 204
- “strftime() — Convert Date/Time to String” on page 354
- “time() — Determine Current Time” on page 391
- "printf() — Print Formatted Characters” on page 211
- "<time.h>” on page 16

asinc() — Calculate Arcsine

**Format**

```c
#include <math.h>
double asind(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The asind() function calculates the arcsine of x, in the range -π/2 to π/2 radians.

**Return Value**
The `asin()` function returns the arcsine of `x`. The value of `x` must be between -1 and 1. If `x` is less than -1 or greater than 1, the `asin()` function sets `errno` to `EDOM`, and returns a value of 0.

**Example that uses `asin()`**

This example prompts for a value for `x`. It prints an error message if `x` is greater than 1 or less than -1; otherwise, it assigns the arcsine of `x` to `y`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAX  1.0
#define MIN -1.0

int main(void)
{
    double x, y;
    printf( "Enter x\n" );
    scanf( "%lf", &x );

    /* Output error if not in range */
    if ( x > MAX )
        printf( "Error: %lf too large for asin\n", x );
    else if ( x < MIN )
        printf( "Error: %lf too small for asin\n", x );
    else
    {
        y = asin( x );
        printf( "asin( %lf ) = %lf\n", x, y );
    }
}
```

/********************  Output should be similar to  ******************
Enter x
asin( 0.200000 ) = 0.201358
*/

**Related Information**

- ["acos() — Calculate Arccosine" on page 40](#)
- ["atan() — atan2() — Calculate Arctangent" on page 47](#)
- ["cos() — Calculate Cosine" on page 66](#)
- ["cosh() — Calculate Hyperbolic Cosine" on page 67](#)
- ["sin() — Calculate Sine" on page 332](#)
- ["sinh() — Calculate Hyperbolic Sine" on page 333](#)
- ["tan() — Calculate Tangent" on page 389](#)
- ["tanh() — Calculate Hyperbolic Tangent" on page 390](#)
- ["<math.h>" on page 7](#)

**assert() — Verify Condition**

**Format**

```c
#include <assert.h>
void assert(int expression);
```

**Language Level:** ANSI
Threadsafe: No.

Description

The assert() function prints a diagnostic message to stderr and aborts the program if expression is false (zero). The diagnostic message has the format: Assertion failed: expression, file filename, line line-number.

The assert() function takes no action if the expression is true (nonzero).

Use the assert() function to identify program logic errors. Choose an expression that holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier NDEBUG to remove the assert() calls from the program. If you define NDEBUG to any value with a #define directive, the C preprocessor expands all assert calls to void expressions. If you use NDEBUG, you must define it before you include <assert.h> in the program.

Return Value

There is no return value.

Note: The assert() function is defined as a macro. Do not use the #undef directive with assert().

Example that uses assert()

In this example, the assert() function tests string for a null string and an empty string, and verifies that length is positive before processing these arguments.

```
#include <stdio.h>
#include <assert.h>

void analyze (char *, int);

int main(void)
{
    char *string = "ABC";
    int length = 3;

    analyze(string, length);
    printf("The string %s is not null or empty, 
        "and has length %d \n", string, length);
}

void analyze(char *string, int length)
{
    assert(string != NULL); /* cannot be NULL */
    assert(*string != '\0'); /* cannot be empty */
    assert(length > 0); /* must be positive */
}
```

******************** Output should be similar to ********************
The string ABC is not null or empty, and has length 3

Related Information

• "abort() — Stop a Program" on page 38
• "<assert.h>" on page 3
atan() – atan2() — Calculate Arctangent

Format

#include <math.h>
double atan(double x);
double atan2(double y, double x);

Language Level: ANSI

Threadsafe: Yes.

Description

The atan() and atan2() functions calculate the arctangent of x and y/x, respectively.

Return Value

The atan() function returns a value in the range \(-\pi/2\) to \(\pi/2\) radians. The atan2() function returns a value in the range \(-\pi\) to \(\pi\) radians. If both arguments of the atan2() function are zero, the function sets errno to EDOM, and returns a value of 0.

Example that uses atan()

This example calculates arctangents using the atan() and atan2() functions.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double a,b,c,d;
    c = 0.45;
    d = 0.23;
    a = atan(c);
    b = atan2(c,d);
    printf("atan( %lf ) = %lf/n", c, a);
    printf("atan2( %lf, %lf ) = %lf/n", c, d, b);
}
```

/******************** Output should be similar to ******************
atan( 0.450000 ) = 0.422854
atan2( 0.450000, 0.230000 ) = 1.098299
******************************************************************/

Related Information

- “acos() — Calculate Arccosine” on page 40
- “asin() — Calculate Arcsine” on page 44
- “cos() — Calculate Cosine” on page 66
- “cosh() — Calculate Hyperbolic Cosine” on page 67
- “sin() — Calculate Sine” on page 332
- “sinh() — Calculate Hyperbolic Sine” on page 333
- “tan() — Calculate Tangent” on page 389
atexit() — Record Program Ending Function

Format

```c
#include <stdlib.h>
int atexit(void (*func)(void));
```

Language Level: ANSI

Threadsafe: Yes.

Description

The atexit() function records the function, pointed to by `func`, that the system calls at normal program end. For portability, you should use the atexit() function to register a maximum of 32 functions. The functions are processed in a last-in, first-out order. The atexit() function cannot be called from the OPM default activation group. Most functions can be used with the atexit function; however, if the exit function is used the atexit function will fail.

Return Value

The atexit() function returns 0 if it is successful, and nonzero if it fails.

Example that uses atexit()

This example uses the atexit() function to call goodbye() at program end.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    void goodbye(void);
    int rc;
    rc = atexit(goodbye);
    if (rc != 0)
        perror("Error in atexit");
    exit(0);
}

void goodbye(void)
/* This function is called at normal program end */
{
    printf("The function goodbye was called at program end\n");
}

/*************** Output should be similar to: ***************
The function goodbye was called at program end
*/
```

Related Information

- "exit() — End Program" on page 80
- "signal() — Handle Interrupt Signals" on page 330
- "<stdlib.h>" on page 15
atof() — Convert Character String to Float

Format
#include <stdlib.h>
double atof(const char *string);

Language Level: ANSI

Threading: Yes.

Description

The `atof()` function converts a character string to a double-precision floating-point value.

The input `string` is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atof()` function expects a `string` in the following form:

```
whitespace [ + ] [ - ] [ . ] digits [ . ] digits [ e ] [ + ] [ - ] digits
```

The white space consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atof()` function ignores leading white-space characters.

For the `atof()` function, `digits` is one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter `e` or `E`. The exponent is a decimal integer, which may be signed.

The `atof()` function will not fail if a character other than a digit follows an `E` or if `e` is read in as an exponent. For example, 100elf will be converted to the floating-point value 100.0. The accuracy is up to 17 significant character digits.

Return Value

The `atof()` function returns a double value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. In case of overflow, the function sets `errno` to `ERANGE` and returns the value `-HUGE_VAL` or `+HUGE_VAL`.

Example that uses `atof()`

This example shows how to convert numbers that are stored as strings to numeric values.
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    double x;
    char *s;
    s = " -2309.12E-15";
    x = atof(s);     /* x = -2309.12E-15 */
    printf("x = %.4e\n",x);
}

/********************  Output should be similar to:  ***************
x = -2.3091e-12
*/

Related Information
• "atoi() — Convert Character String to Integer"
• "atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 51
• “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 380
• “strtol() — Convert Character String to Double” on page 376
• “<stdlib.h>” on page 15

atoi() — Convert Character String to Integer

Format
#include <stdlib.h>
int atoi(const char *string);

Language Level: ANSI

Threadsafe: Yes.

Description

The atoi() function converts a character string to an integer value. The input string is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The atoi() function does not recognize decimal points or exponents. The string argument for this function has the form:

```plaintext
whitespace digits
```

where whitespace consists of the same characters for which the isspace() function is true, such as spaces and tabs. The atoi() function ignores leading white-space characters. The value digits represents one or more decimal digits.
Return Value

The `atoi()` function returns an int value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. The return value is undefined in the case of an overflow.

Example that uses `atoi()`

This example shows how to convert numbers that are stored as strings to numeric values.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i;
    char *s;
    s = " -9885";
    i = atoi(s);    /* i = -9885 */
    printf("i = %d\n",i);
}
```

```
/************** Output should be similar to: **************
i = -9885
*/
```

Related Information

- “atof() — Convert Character String to Float” on page 49
- “atol() — atoll() — Convert Character String to Long or Long Long Integer”
- “strtod() — Convert Character String to Double” on page 376
- “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 380
- “<stdlib.h>” on page 15

---

**atol() — atoll() — Convert Character String to Long or Long Long Integer**

**Format**

```c
#include <stdlib.h>
long int atol(const char *string);
```

```c
#include <stdlib.h>
long long int atoll(const char *string);
```

**Language Level**: ANSI

**Threading safe**: Yes.

**Description**

The `atol()` function converts a character string to a long value. The `atoll()` function converts a character string to a long long value.
The input string is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The atol() and atoll() functions do not recognize decimal points or exponents. The string argument for this function has the form:

```
whitespace + - digits
```

where whitespace consists of the same characters for which the isspace() function is true, such as spaces and tabs. The atol() and atoll() functions ignore leading white-space characters. The value digits represents one or more decimal digits.

Return Value

The atol() and atoll() functions return a long or a long long value that is produced by interpreting the input characters as a number. The return value is 0L if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

Example that uses atol()

This example shows how to convert numbers that are stored as strings to numeric values.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    long l;
    char *s;

    s = "98854 dollars";
    l = atol(s); /* l = 98854 */
    printf("l = %.ld\n",l);
}

/*******************  Output should be similar to:  *******************
1 = 98854
*/
```

Related Information

- "atof() — Convert Character String to Float" on page 49
- "atoi() — Convert Character String to Integer" on page 50
- "strtod() — Convert Character String to Double" on page 376
- "strtol() — strtoll() — Convert Character String to Long and Long Long Integer" on page 380
- "<stdlib.h>" on page 15

Bessel Functions

Format
Language Level: ILE C Extension

Threatsafe: Yes.

Description

Bessel functions solve certain types of differential equations. The j0(), j1(), and jn() functions are Bessel functions of the first kind for orders 0, 1, and n, respectively. The y0(), y1(), and yn() functions are Bessel functions of the second kind for orders 0, 1, and n, respectively.

The argument x must be positive. The argument n should be greater than or equal to zero. If n is less than zero, it will be a negative exponent.

Return Value

For j0(), j1(), y0(), or y1(), if the absolute value of x is too large, the function sets errno to ERANGE, and returns 0. For y0(), y1(), or yn(), if x is negative, the function sets errno to EDOM and returns the value -HUGE_VAL. For y0, y1(), or yn(), if x causes overflow, the function sets errno to ERANGE and returns the value -HUGE_VAL.

Example that uses Bessel Functions

This example computes y to be the order 0 Bessel function of the first kind for x. It also computes z to be the order 3 Bessel function of the second kind for x.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = 4.27;
    y = j0(x);       /* y = -0.3660 is the order 0 bessel */
    /* function of the first kind for x */
    z = yn(3,x);     /* z = -0.0875 is the order 3 bessel */
    /* function of the second kind for x */

    printf("y = %lf\n", y);
    printf("z = %lf\n", z);
}
```

Output should be similar to:

```
y = -0.366022
z = -0.087482
```

Related Information

- “erf() – erfc() — Calculate Error Functions” on page 79
- “gamma() — Gamma Function” on page 140
bsearch() — Search Arrays

Format
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
    size_t num, size_t size,
    int (*compare)(const void *key, const void *element));

Language Level: ANSI

Threading: Yes.

Description
The bsearch() function performs a binary search of an array of num elements, each
of size bytes. The array must be sorted in ascending order by the function pointed
to by compare. The base is a pointer to the base of the array to search, and key is the
value being sought.

The compare argument is a pointer to a function you must supply that compares
two items and returns a value specifying their relationship. The first item in the
argument list of the compare() function is the pointer to the value of the item that
is being searched for. The second item in the argument list of the compare() function
is a pointer to the array element being compared with the key. The
compare() function must compare the key value with the array element and then
return one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>key less than element</td>
</tr>
<tr>
<td>0</td>
<td>key identical to element</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>key greater than element</td>
</tr>
</tbody>
</table>

Return Value
The bsearch() function returns a pointer to key in the array to which base points. If
two keys are equal, the element that key will point to is unspecified. If the
bsearch() function cannot find the key, it returns NULL.

Example that uses bsearch()
This example performs a binary search on the argv array of pointers to the
program parameters and finds the position of the argument PATH. It first removes
the program name from argv, and then sorts the array alphabetically before calling
bsearch(). The compare1() and compare2() functions compare the values pointed
to by arg1 and arg2 and return the result to the bsearch() function.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int compare1(const void *, const void *);
int compare2(const void *, const void *);

main(int argc, char *argv[])
{    /* This program performs a binary */
```
char **result;       /* search on the argv array of pointers */
char *key = "PATH";  /* to the program parameters. It first */
int i;              /* removes the program name from argv */
/* then sorts the array alphabetically */
argv++;              /* before calling bsearch. */
argc--;              
qsort((char *)argv, argc, sizeof(char *), compare1);
result = (char**)bsearch(&key, (char *)argv, argc, sizeof(char *), compare2);
if (result != NULL) {
  printf("result = <s>n",*result);
} else printf("result is null\n");

/*This function compares the values pointed to by arg1 and arg2 and returns the result to qsort. arg1 and arg2 are both pointers to elements of the argv array. */
int compare1(const void *arg1, const void *arg2)
{
  return (strcmp(*(char **)arg1, *(char **)arg2));
}

/*This function compares the values pointed to by arg1 and arg2 and returns the result to bsearch */
/*arg1 is a pointer to the key value, arg2 points to the element of argv that is being compared to the key */
/*value. */
int compare2(const void *arg1, const void *arg2)
{
  return (strcmp(*(char **)arg1, *(char **)arg2));
}
/
result = <PATH>

/******************** When the input on the iSeries command line is *******
CALL BSEARCH PARM(WHERE IS PATH IN THIS PHRASE?')
*/

Related Information
* "qsort() — Sort Array" on page 228
* "<stdlib.h>" on page 15

btowc() — Convert Single Byte to Wide Character

Format
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);

Language Level: ANSI

Threading safe: Yes.

Description
The `btowc()` function converts the single byte value \( c \) to the wide-character representation of \( c \). If \( c \) does not constitute a valid (one-byte) multibyte character in the initial shift state, the `btowc()` function returns `WEOF`.

The behavior of the `btowc()` function is affected by the LC_CTYPE category of the current locale. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command.

**Return Value**

The `btowc()` function returns `WEOF` if \( c \) has the value EOF, or if (unsigned char) \( c \) does not constitute a valid (one-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

**Note:** When the program is compiled using `LOCALETYPE(*LOCALE)`, the wide character produced is wide EBCDIC. When the program is compiled using `LOCALETYPE(*LOCALEUCS2)`, the wide character produced is UCS2. When the program is compiled using `LOCALETYPE(*LOCALEUTF)`, the wide character produced is UTF32.

**Example that uses `btowc()`**

This example scans various types of data.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define UPPER_LIMIT   0xFF

int main(void)
{
  int wc;
  int ch;
  if (NULL == setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE")) {
    printf("Locale could not be loaded\n");
    exit(1);
  }
  for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
    wc = btowc(ch);
    if (wc==WEOF) {
      printf("%#04x is not a one-byte multibyte character\n", ch);
    } else {
      printf("%#04x has wide character representation: %#06x\n", ch, wc);
    }
  }
  wc = btowc(EOF);
  if (wc==WEOF) {
    printf("The character is EOF.\n", ch);
  } else {
    printf("EOF has wide character representation: %#06x\n", wc);
  }
  return 0;
}
```

If the locale is bound to SBCS, the output should be similar to:

- 0000 has wide character representation: 000000
- 0x01 has wide character representation: 0x0001
- ...
- 0xfe has wide character representation: 0x00fe
Related Information
- “mblen() — Determine Length of a Multibyte Character” on page 182
- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 196
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- “mbstowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 191
- “setlocale() — Set Locale” on page 322
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 425
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 451
- “<stdio.h>” on page 14
- “<wchar.h>” on page 17

_C_Get_Ssn_Handle() — Handle to C Session

Format
#include <stdio.h>

_SSN_HANDLE_T _C_Get_Ssn_Handle (void)

Language Level: ILE C Extension

Threading: Yes.

Description

Returns a handle to the C session for use with Dynamic Screen Manager (DSM) APIs.

Return Value

The _C_Get_Ssn_Handle() function returns a handle to the C session. If an error occurs, _SSN_HANDLE_T is set to zero. See the APIs topic in the iSeries Information Center for more information about using the _C_Get_Ssn_Handle() function with DSM APIs.

calloc() — Reserve and Initialize Storage

Format
#include <stdlib.h>
void *calloc(size_t num, size_t size);

Language Level: ANSI

Threading: Yes.

Description
The calloc() function reserves storage space for an array of num elements, each of length size bytes. The calloc() function then gives all the bits of each element an initial value of 0.

**Return Value**

The calloc() function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage, or if num or size is 0.

**Note:** To use Teraspace storage instead of heap storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the compiler command. This maps the calloc() library function to _C_TS calloc(), its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to _C_TS calloc() is 2GB - 224, or 2147483424 bytes.

For more information about Teraspace, see the ILE Concepts manual.

**Example that uses calloc()**

This example prompts for the number of array entries required, and then reserves enough space in storage for the entries. If calloc() is successful, the example prints out each entry; otherwise, it prints out an error.
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array; /* start of the array */
    long * index; /* index variable */
    int    i;    /* index variable */
    int    num; /* number of entries of the array */

    printf( "Enter the size of the array\n" );
    scanf( "%i", &num );
    /* allocate num entries */
    if ( (index = array = (long *) calloc( num, sizeof( long ))) != NULL )
    {
        for ( i = 0; i < num; ++i ) /* put values in arr */
            *index++ = i; /* using pointer no */
        for ( i = 0; i < num; ++i ) /* print the array out */
            printf( "array[ %i ] = %i\n", i, array[i] );
    }
    else /* out of storage */
    {
        perror( "Out of storage" );
        abort();
    }
}

/******************  Output should be similar to:  *******************/
Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2

Related Information
• "free() — Release Storage Blocks" on page 118
• "malloc() — Reserve Storage Block" on page 180
• "realloc() — Change Reserved Storage Block Size" on page 247
• "<stdlib.h>" on page 15

catclose() — Close Message Catalog

Format
#include <nl_types.h>
int catclose (nl_catd catd);

Language Level: XPG4

Threading safe: Yes.

Description
The catclose() function closes the previously opened message catalog that is identified by catd.
**Note:** This function is not available when either **LOCALETYPE(*CLD)** or **SYSIFOPT(*NOIFSIO)** is specified on the compilation command.

### Return Value

If the close is performed successfully, 0 is returned. Otherwise, -1 is returned indicating failure, which may happen if **catd** is not a valid message catalog descriptor.

The value of **errno** may be set to:

**EBADF**
- The catalog descriptor is not valid.

**EINTR**
- The function was interrupted by a signal.

### Example that uses **catclose()**

```c
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {

    nl_catd msg_file;
    char * my_msg;
    char * my_locale;

    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);

    if (msg_file != CATD_ERR)  {

        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);

    }

}
```

### Related Information

- "**catopen() — Open Message Catalog**” on page 61
- "**catgets() — Retrieve a Message from a Message Catalog**”

---

**catgets() — Retrieve a Message from a Message Catalog**

### Format

```c
#include <nl_types.h>
char *catgets(nl_catd catd, int set_id, int msg_id, char *s);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**
The catgets() function retrieves message msg_id, in set set_id from the message catalog that is identified by catd. catd is a message catalog descriptor that is returned by a previous call to catopen(). The s argument points to a default message which will be returned by catgets() if the identified message cannot be retrieved.

Note: This function is not available when either LOCALETYP(*CLD) or SYSIFOPT(*NOIFSIO) is specified on the compilation command.

Return Value

If the message is retrieved successfully, then catgets() returns a pointer to the message string that is contained in the message catalog. If the message is retrieved unsuccessfully, then a pointer to the default string s is returned.

The value of errno may be set to the following:

EBADF
The catalog descriptor is not valid.

ECONVERT
A conversion error occurred.

EINTR
The function was interrupted by a signal.

Example that uses catgets()

#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;
    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);
    if (msg_file != CATD_ERR) {
        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);
    }
}

Related Information

• “catclose() — Close Message Catalog” on page 59
• “catopen() — Open Message Catalog”
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**

The `catopen()` function opens a message catalog, which must be done before a message can be retrieved. The NLSPATH environment variable and the LC_MESSAGES category are used to find the specified message catalog if no slash (/) characters are found in the name. If the name contains one or more slash (/) characters, then the name is interpreted as a path name of the catalog to open.

If there is no NLSPATH environment variable, or if a message catalog cannot be found in the path specified by NLSPATH, then a default path will be used. The default path may be affected by the setting of the LC_MESSAGES locale category if the value of `oflag` is NL_CAT_LOCALE, or by the LANG environment variable if the value of `oflag` is zero.

The message catalog descriptor will remain valid until it is closed by a call to `catclose()`. If the LC_MESSAGES locale category is changed, it may invalidate existing open message catalogs.

**Note:** This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command. The name of the message catalog must be a valid Integrated File System file name.

**Return Value**

If the message catalog is opened successfully, then a valid catalog descriptor is returned. If `catopen()` is unsuccessful, then it returns CATD_ERR ((nl_catd)-1).

The `catopen()` function might fail under the following conditions, and the value of errno may be set to:

- **EACCES**
  Insufficient authority to read the message catalog specified, or to search the component of the path prefix of the message catalog specified.

- **ECONVERT**
  A conversion error occurred.

- **EMFILE**
  NL_MAXOPEN message catalogs are currently open.

- **ENAMETOOLONG**
  The length of the path name of the message catalog exceeds PATH_MAX, or a path name component is longer than NAME_MAX.

- **ENFILE**
  Too many files are currently open in the system.

- **ENOENT**
  The message catalog does not exist, or the name argument points to an empty string.
Example that uses `catopen()`

```c
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */
int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;
    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);
    if (msg_file != CATD_ERR) {
        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);
    }
}
```

Related Information

- "`catclose()` — Close Message Catalog" on page 59
- "`catgets()` — Retrieve a Message from a Message Catalog" on page 60

---

`ceil()` — Find Integer >= Argument

**Format**

```c
#include <math.h>
double ceil(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `ceil()` function computes the smallest integer that is greater than or equal to `x`.

**Return Value**

The `ceil()` function returns the integer as a double value.

**Example that uses `ceil()`**

This example sets `y` to the smallest integer greater than 1.05, and then to the smallest integer greater than -1.05. The results are 2.0 and -1.0, respectively.
```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double y, z;
    y = ceil(1.05);    /* y = 2.0 */
    z = ceil(-1.05);   /* z = -1.0 */
    printf("y = %.2f ; z = %.2f\n", y, z);
}

/***************** Output should be similar to: ***********************
 y = 2.00 ; z = -1.00
**********************************************************************/
```

Related Information

- “floor() — Find Integer <= Argument” on page 99
- “fmod() — Calculate Floating-Point Remainder” on page 99
- “<math.h>” on page 7

---

clearerr() — Reset Error Indicators

Format

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The clearerr() function resets the error indicator and end-of-file indicator for the specified stream. Once set, the indicators for a specified stream remain set until your program calls the clearerr() function or the rewind() function. The fseek() function also clears the end-of-file indicator. The ILE C/C++ run-time environment does not automatically clear error or end of file indicators.

Return Value

There is no return value.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
EIORECERR

A recoverable I/O error occurred.

Example that uses clearerr()

This example reads a data stream, and then checks that a read error has not occurred.

```c
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

int main(void)
{
  if ((stream = fopen("mylib/myfile", "r")) != NULL)
    {
      if ((c=getc(stream)) == EOF)
      {
        if (ferror(stream))
        {
          perror("Read error");
          clearerr(stream);
        }
      }
    }
  else
    exit(0);
}
```

Related Information

- "feof() — Test End-of-File Indicator" on page 86
- "ferror() — Test for Read/Write Errors" on page 87
- “fseek() — fseeko() — Reposition File Position” on page 124
- "perror() — Print Error Message" on page 209
- "rewind() — Adjust Current File Position” on page 259
- "strerror() — Set Pointer to Run-Time Error Message” on page 350
- "<stdio.h>” on page 14

---

clock() — Determine Processor Time

Format

```c
#include <time.h>
clock_t clock(void);
```

Language Level: ANSI

Threatsafe: Yes.

Description

The clock() function returns an approximation of the processor time used by the program since the beginning of an implementation-defined time-period that is related to the process invocation. To obtain the time in seconds, divide the value that is returned by clock() by the value of the macro CLOCKS_PER_SEC.

Return Value
If the value of the processor time is not available or cannot be represented, the `clock()` function returns the value `(clock_t)-1`.

To measure the time spent in a program, call `clock()` at the start of the program, and subtract its return value from the value returned by subsequent calls to `clock()`. On other platforms, you can not always rely on the `clock()` function because calls to the `system()` function may reset the clock.

**Example that uses clock()**

This example prints the time that has elapsed since the program was called.

```c
#include <time.h>
#include <stdio.h>

double timel, timedif;  /* use doubles to show small values */

int main(void)
{
    int i;
    timel = (double) clock();  /* get initial time */
    timel = timel / CLOCKS_PER_SEC;  /* in seconds */

    /* running the FOR loop 10000 times */
    for (i=0; i<10000; i++);
    /* call clock a second time */
    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - timel;
    printf("The elapsed time is %lf seconds\n", timedif);
}
```

**Related Information**
- "difftime() — Compute Time Difference" on page 76
- "time() — Determine Current Time" on page 391
- "<time.h>" on page 16

---

**cos() — Calculate Cosine**

**Format**

```
#include <math.h>
double cos(double x);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `cos()` function calculates the cosine of `x`. The value `x` is expressed in radians. If `x` is too large, a partial loss of significance in the result may occur.

**Return Value**

The `cos()` function returns the cosine of `x`. The value of `errno` may be set to either `EDOM` or `ERANGE`.

**Example that uses cos()**
This example calculates $y$ to be the cosine of $x$.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = 7.2;
    y = cos(x);
    printf("cos( %lf ) = %lf\n", x, y);
}
```

Example that uses `cosh()`

This example calculates $y$ to be the hyperbolic cosine of $x$.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = 7.2;
    y = cosh(x);
    printf("cosh( %lf ) = %lf\n", x, y);
}
```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x,y;
    x = 7.2;
    y = cosh(x);
    printf("cosh( %lf ) = %lf
", x, y);
}

****************** Output should be similar to: ******************
cosh( 7.200000 ) = 669.715755
*/

Related Information

- “acos() — Calculate Arccosine” on page 40
- “cos() — Calculate Cosine” on page 66
- “sin() — Calculate Sine” on page 332
- “sinh() — Calculate Hyperbolic Sine” on page 333
- “tan() — Calculate Tangent” on page 389
- “tanh() — Calculate Hyperbolic Tangent” on page 390
- “<math.h>” on page 7

### ctime() — Convert Time to Character String

#### Format

```c
#include <time.h>
char *ctime(const time_t *time);
```

**Language Level**: ANSI

**Threadsafe**: No. Use `ctime_r()` instead.

**Description**

The `ctime()` function converts the time value pointed to by `time` to local time in the form of a character string. A time value is usually obtained by a call to the `time()` function.

The string result that is produced by `ctime()` contains exactly 26 characters and has the format:

"%3s %3s%3d %2d:%2d:%2d %d\n"

For example:

Mon Jul 16 02:03:55 1987\n\0

The `ctime()` function uses a 24-hour clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have a constant width. Dates with only one digit are preceded with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

**Return Value**
The `ctime()` function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to the `ctime()` function is equivalent to:

```
asctime(localtime(&anytime))
```

**Note:** The `asctime()` and `ctime()` functions, and other time functions may use a common, statically allocated buffer to hold the return string. Each call to one of these functions may destroy the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically-allocated buffer to hold the return string. These functions can be used in the place of `asctime()`, `ctime()`, `gmtime()`, and `localtime()` if reentrancy is desired.

**Example that uses `ctime()`**

This example polls the system clock using `time()`. It then prints a message giving the current date and time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    time(&ltime);

    printf("the time is %s", ctime(&ltime));
}
```

**Related Information**

- “`asctime()` — Convert Time to Character String” on page 41
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 43
- “`ctime_r()` — Convert Time to Character String (Restartable)"
- “`gmtime()` — Convert Time” on page 151
- “`gmtime_r()` — Convert Time (Restartable)” on page 153
- “`localtime()` — Convert Time” on page 173
- “`localtime_r()` — Convert Time (Restartable)” on page 174
- “`mktime()` — Convert Local Time” on page 204
- “`setlocale()` — Set Locale” on page 322
- “`strftime()` — Convert Date/Time to String” on page 354
- “`time()` — Determine Current Time” on page 391
- “`printf()` — Print Formatted Characters” on page 211
- “<time.h>” on page 16

**`ctime_r()` — Convert Time to Character String (Restartable)**

**Format**

```
#include <time.h>
char *ctime_r(const time_t *time, char *buf);
```

**Language Level:** XPG4

**Threadsafe:** Yes.
Description

This function is the restartable version of the `ctime()` function.

The `ctime_r()` function converts the time value pointed to by `time` to local time in the form of a character string. A time value is usually obtained by a call to the `time()` function.

The string result that is produced by the `ctime_r()` function contains exactly 26 characters and has the format:

```
%.3s %.3s%3d %.2d:%.2d:%.2d %d
```

For example:

```
Mon Jul 16 02:03:55 1987
```

The `ctime_r()` function uses a 24-hour clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have a constant width. Dates with only one digit are preceded with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

Return Value

The `ctime_r()` function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to `ctime_r()` is equivalent to:

```
asctime_r(localtime_r(&anytime, buf2), buf)
```

where `buf` is a pointer to char.

Example that uses `ctime_r()`

This example polls the system clock using `ctime_r()`. It then prints a message giving the current date and time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    char buf[50];

    time(&ltime);
    printf("the time is %s, ctime_r(&ltime, buf)\n"),
}
```

Related Information

- "asctime() — Convert Time to Character String” on page 41
- "asctime_r() — Convert Time to Character String (Restartable)” on page 43
- "ctime() — Convert Time to Character String” on page 68
- "gmtime() — Convert Time” on page 151
- "gmtime_r() — Convert Time (Restartable)” on page 153
- "localtime_r() — Convert Time (Restartable)” on page 174
- "localtime() — Convert Time” on page 173
- "mktime() — Convert Local Time” on page 204
_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)

Format

```c
#include <mallocinfo.h>
int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level,
                      struct _C_mallinfo_t *output_record, size_t size_of_output);
```

Language Level: Extended

Threadsafe: Yes.

Description

The _C_TS_malloc_debug() function determines the amount of teraspace memory used and returns the information within the given output_record structure. If the given dump_level parameter is greater than 0, it also dumps the internal memory structures used to stdout. If the given verify_level parameter is greater than 0, it also performs verification checks for the internal memory structures. If a verification fails, a message is generated to stdout indicating the failure. If both the dump_level and verify_level parameters are 0, this function provides the same behavior as the _C_TS_malloc_info function.

The following macros are defined within the <mallocinfo.h> include file to be specified for the dump_level parameter:

<table>
<thead>
<tr>
<th>_C_NO_DUMPS</th>
<th>No information is dumped</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_DUMP_TOTALS</td>
<td>Overall totals and totals for each chunk are printed</td>
</tr>
<tr>
<td>_C_DUMP_CHUNKS</td>
<td>Additional information about each chunk is printed</td>
</tr>
<tr>
<td>_C_DUMP_NODES</td>
<td>Additional information for all nodes within each chunk is printed</td>
</tr>
<tr>
<td>_C_DUMP_TREE</td>
<td>Additional information for the cartesian tree used to track free nodes is printed</td>
</tr>
<tr>
<td>_C_DUMP_ALL</td>
<td>All available information is printed</td>
</tr>
</tbody>
</table>

The following macros are defined within the <mallocinfo.h> include file to be specified for the verify_level parameter:

<table>
<thead>
<tr>
<th>_C_NO_CHECKS</th>
<th>No verification checks are performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_CHECK_TOTALS</td>
<td>Totals are verified for correctness</td>
</tr>
<tr>
<td>_C_CHECK_CHUNKS</td>
<td>Additional verifications are performed for each chunk</td>
</tr>
<tr>
<td>_C_CHECK_NODES</td>
<td>Additional verifications are performed for all nodes within each chunk</td>
</tr>
<tr>
<td>_C_CHECK_TREE</td>
<td>Additional verifications are performed for the cartesian tree used to track free nodes</td>
</tr>
</tbody>
</table>

Chapter 2. Library Functions 71
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_CHECK_ALL</td>
<td>All verifications are performed</td>
</tr>
<tr>
<td>_C_CHECK_ALL_AND_ABORT</td>
<td>All verifications are performed, and if any verification fails, the abort() function is called</td>
</tr>
</tbody>
</table>

**Note:** This function is for low-level debug of teraspace memory usage within an application.

**Return Value**

If successful, the function returns 0. If an error occurs, the function returns a negative value.

**Example that uses _C_TS_malloc_debug()**

This example prints the information returned from _C_TS_malloc_debug() to stdout. This program is compiled with TERASPACE(*YES *TSIFC).
```c
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int           rc;
    void         *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);

    rc = _C_TS_malloc_debug(_C_DUMP_TOTALS,
                   _C_NO_CHECKS,
                   &info, sizeof(info));

    if (rc == 0) {
        printf("_C_TS_malloc_debug successful\n");
    } else {
        printf("_C_TS_malloc_debug failed (rc = %d)\n", rc);
    }

    free(m);
}

/******************************************************************************
 The output should be similar to:

 total_bytes         = 524288
 allocated_bytes     = 688
 unallocated_bytes   = 523600
 allocated_blocks    = 1
 unallocated_blocks  = 1
 requested_bytes     = 500
 pad_bytes           = 12
 overhead_bytes      = 176
 Number of memory chunks = 1
 Total bytes          = 524288
 Total allocated bytes = 688
 Total unallocated bytes = 523600
 Total allocated blocks = 1
 Total unallocated blocks = 1
 Total requested bytes = 500
 Total pad bytes      = 12
 Total overhead bytes = 176
 _C_TS_malloc_debug successful
******************************************************************************

Related Information
• "_C_TS_malloc_info() — Determine amount of teraspace memory used"
• "calloc() — Reserve and Initialize Storage“ on page 57
• "free() — Release Storage Blocks” on page 118
• “malloc() — Reserve Storage Block” on page 180
• “realloc() — Change Reserved Storage Block Size” on page 247
• “<mallocinfo.h>” on page 8

_C_TS_malloc_info() — Determine amount of teraspace memory used
Format
#include <mallocinfo.h>
int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);

**Language Level**: Extended

**Threadsafe**: Yes.

**Description**

The 

_C_TS_malloc_info() function determines the amount of teraspace memory used and returns the information within the given output_record structure.

**Note**: This function is for low-level debug of teraspace memory usage within an application.

**Return Value**

If successful, the function returns 0. If an error occurs, the function returns a negative value.

**Example that uses _C_TS_malloc_info()**

This example prints the information returned from _C_TS_malloc_info() to stdout. This program is compiled with TERASPACE(*YES *TSIFC).
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int           rc;
    void         *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);

    rc = _C_TS_malloc_info(&info, sizeof(info));
    if (rc == 0) {
        printf("Total bytes = %llu\n", info.total_bytes);
        printf("Total allocated bytes = %llu\n", info.allocated_bytes);
        printf("Total unallocated bytes = %llu\n", info.unallocated_bytes);
        printf("Total allocated blocks = %llu\n", info.allocated_blocks);
        printf("Total unallocated blocks = %llu\n", info.unallocated_blocks);
        printf("Total requested bytes = %llu\n", info.requested_bytes);
        printf("Total pad bytes = %llu\n", info.pad_bytes);
        printf("Total overhead bytes = %llu\n", info.overhead_bytes);
    } else {
        printf("_C_TS_malloc_info failed (rc = %d)\n", rc);
    }

    free(m);
}

/* ************************************************************
   The output should be similar to:

   Total bytes = 524288
   Total allocated bytes = 688
   Total unallocated bytes = 523600
   Total allocated blocks = 1
   Total unallocated blocks = 1
   Total requested bytes = 500
   Total pad bytes = 12
   Total overhead bytes = 176
   ************************************************************

Related Information

- "_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)" on page 71
- "calloc() — Reserve and Initialize Storage" on page 57
- "free() — Release Storage Blocks" on page 118
- "malloc() — Reserve Storage Block" on page 180
- "realloc() — Change Reserved Storage Block Size" on page 247
- "<mallocinfo.h>" on page 8
*/
difftime() — Compute Time Difference

Format

```c
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Language Level: ANSI

Threading: Yes.

Description

The `difftime()` function computes the difference in seconds between `time2` and `time1`.

Return Value

The `difftime()` function returns the elapsed time in seconds from `time1` to `time2` as a double precision number. Type `time_t` is defined in `<time.h>`.

Example that uses `difftime()`

This example shows a timing application that uses `difftime()`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10000.
#include <time.h>
#include <stdio.h>

#define RUNS 1000
#define SIZE 10000

int mark[SIZE];

int main(void)
{
    time_t start, finish;
    int i, loop, n, num;

time(&start);

    /* This loop finds the prime numbers between 2 and SIZE */
    for (loop = 0; loop < RUNS; ++loop)
    {
        for (n = 0; n < SIZE; ++n)
            mark[n] = 0;

        /* This loop marks all the composite numbers with -1 */
        for (num = 0, n = 2; n < SIZE; ++n)
            if ( ! mark[n])
            {
                for (i = 2 * n; i < SIZE; i += n)
                    mark[i] = -1;

                ++num;
            }
    }

time(&finish);
    printf("Program takes an average of %f seconds 
" to find %d primes.\n",
difftime(finish,start)/RUNS, num);
}

/******************** Output should be similar: ********************
The program takes an average of 0.106000 seconds to find 1229 primes. 
*/

Related Information

• "asctime() — Convert Time to Character String" on page 41
• "asctime_r() — Convert Time to Character String (Restartable)" on page 43
• "ctime() — Convert Time to Character String" on page 68
• "ctime_r() — Convert Time to Character String (Restartable)" on page 69
• "gmtime() — Convert Time" on page 151
• "gmtime_r() — Convert Time (Restartable)" on page 153
• "localtime() — Convert Time" on page 173
• "localtime_r() — Convert Time (Restartable)" on page 174
• "mktime() — Convert Local Time" on page 204
• "strftime() — Convert Date/Time to String" on page 354
• "time() — Determine Current Time" on page 391
• "<time.h>" on page 16

div() — Calculate Quotient and Remainder

Format
#include <stdlib.h>
div_t div(int numerator, int denominator);

Language Level: ANSI

Threadsafe: Yes. However, only the function version is threadsafe. The macro version is NOT threadsafe.

Description

The `div()` function calculates the quotient and remainder of the division of `numerator` by `denominator`.

Return Value

The `div()` function returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. If the return value cannot be represented, its value is undefined. If `denominator` is 0, an exception will be raised.

Example that uses `div()`

This example uses `div()` to calculate the quotients and remainders for a set of two dividends and two divisors.

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int num[2] = {45,-45};
    int den[2] = {7,-7};
    div_t ans;   /* div_t is a struct type containing two ints:
                   'quot' stores quotient; 'rem' stores remainder */
    short i,j;

    printf("Results of division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            ans = div(num[i],den[j]);
            printf("Dividend: %6d  Divisor: %6d", num[i], den[j]);
            printf("  Quotient: %6d  Remainder: %6d\n", ans.quot, ans.rem);
        }

}  

/********************  Output should be similar to:  ********************
Results of division:
Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3

******************************************************************/

Related Information

- "ldiv() — lldiv() — Perform Long and Long Long Division” on page 166
- "<stdlib.h>” on page 15
erf() – erfc() — Calculate Error Functions

Format
#include <math.h>
double erf(double x);
double erfc(double x);

Language Level: ILE C Extension

Threading: Yes.

Description
The erf() function calculates the error function of:

\[ 2\pi^{-1/2} \int_0^x e^{-t^2} dt \]

The erfc() function computes the value of 1.0 - erf(x). The erfc() function is used in place of erf() for large values of x.

Return Value
The erf() function returns a double value that represents the error function. The erfc() function returns a double value representing 1.0 - erf.

Example that uses erf()

This example uses erf() and erfc() to compute the error function of two numbers.

```c
#include <stdio.h>
#include <math.h>

double smallx, largex, value;

int main(void)
{
    smallx = 0.1;
    largex = 10.0;

    value = erf(smallx); /* value = 0.112463 */
    printf("Error value for 0.1: %.10f\n", value);

    value = erfc(largex); /* value = 2.088488e-45 */
    printf("Error value for 10.0: %e\n", value);
}
```

/* Output should be similar to: */
Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45

Related Information
- "Bessel Functions" on page 52
- "gamma() — Gamma Function" on page 140
exit() — End Program

Format
#include <stdlib.h>
void exit(int status);

Language Level: ANSI

Threadsafe: Yes.

Description

The exit() function returns control to the host environment from the program. It first calls all functions that are registered with the atexit() function, in reverse order; that is, the last one that is registered is the first one called. It deletes all buffers and closes all open files before ending the program.

The argument status can have a value from 0 to 255 inclusive, or be one of the macros EXIT_SUCCESS or EXIT_FAILURE. A status value of EXIT_SUCCESS or 0 indicates a normal exit; otherwise, another status value is returned.

Note: When compiled with SYSIFCOPT(*ASYNCSIGNAL), exit() cannot be called in a signal handler.

Return Value

The exit() function returns both control and the value of status to the operating system.

Example that uses exit()

This example ends the program after deleting buffers and closing any open files if it cannot open the file myfile.

#include <stdio.h>
#include <stdlib.h>
FILE *stream;
int main(void)
{
    if ((stream = fopen("mylib/myfile", "r")) == NULL) {
        perror("Could not open data file");
        exit(EXIT_FAILURE);
    }
}

Related Information

- "abort() — Stop a Program" on page 38
- "atexit() — Record Program Ending Function" on page 48
- "signal() — Handle Interrupt Signals" on page 330
- "<stdlib.h>" on page 15
exp() — Calculate Exponential Function

Format

#include <math.h>
double exp(double x);

Language Level: ANSI

Threadsafe: Yes.

Description

The exp() function calculates the exponential value of a floating-point argument \( x \) ( \( e^x \), where \( e \) equals 2.17128128...).

Return Value

If an overflow occurs, the exp() function returns HUGE_VAL. If an underflow occurs, it returns 0. Both overflow and underflow set errno to ERANGE. The value of errno may also be set to EDOM.

Example that uses exp()

This example calculates \( y \) as the exponential function of \( x \):

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = 5.0;
    y = exp(x);
    printf("exp( %lf ) = %lf\n", x, y);
}
```

/******************** Output should be similar to: *********************/

exp( 5.000000 ) = 148.413159

Related Information

- “log() — Calculate Natural Logarithm” on page 175
- “log10() — Calculate Base 10 Logarithm” on page 176
- “<math.h>” on page 7

fabs() — Calculate Floating-Point Absolute Value

Format

#include <math.h>
double fabs(double x);

Language Level: ANSI

Threadsafe: Yes.
**Description**

The `fabs()` function calculates the absolute value of the floating-point argument `x`.

**Return Value**

The `fabs()` function returns the absolute value. There is no error return value.

**Example that uses fabs()**

This example calculates `y` as the absolute value of `x`:

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = -5.6798;
    y = fabs(x);
    printf("fabs( %lf ) = %lf\n", x, y);
}
```

/******************** Output should be similar to: *******************/
fabs( -5.679800 ) = 5.679800
*/

**Related Information**

- "abs() — Calculate Integer Absolute Value" on page 39
- "labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer" on page 164
- "<math.h>" on page 7

---

**fclose() — Close Stream**

**Format**

```c
#include <stdio.h>
int fclose(FILE *stream);
```

**Language Level**: ANSI

**Threading**: Yes.

**Description**

The `fclose()` function closes a stream pointed to by `stream`. This function deletes all buffers that are associated with the stream before closing it. When it closes the stream, the function releases any buffers that the system reserved. When a binary stream is closed, the last record in the file is padded with null characters (`\0`) to the end of the record.

**Return Value**

The `fclose()` function returns 0 if it successfully closes the stream, or EOF if any errors were detected.
The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
<tr>
<td>ESCANFAILURE</td>
<td>The file was marked with a scan failure.</td>
</tr>
</tbody>
</table>

Note: The storage pointed to by the FILE pointer is freed by the fclose() function. After the use of the fclose() function, any attempt to use the FILE pointer is not valid.

Example that uses fclose()

This example opens a file myfile for reading as a stream; then it closes this file.

```c
#include <stdio.h>

#define NUM_ALPHA 26

int main(void)
{
    FILE *stream;
    char buffer[NUM_ALPHA];

    if (( stream = fopen("mylib/myfile", "r")) != NULL )
    {
        fread(buffer, sizeof(char), NUM_ALPHA, stream);
        printf("buffer = %s\n", buffer);
    }

    if (fclose(stream)) /* Close the stream. */
        perror("fclose error");
    else printf("File mylib/myfile closed successfully.\n");
}
```

Related Information
- "fflush() — Write Buffer to File" on page 87
- "fopen() — Open Files" on page 100
- "freopen() — Redirect Open Files" on page 120
- "<stdio.h>" on page 14

fdopen() — Associates Stream With File Descriptor

Format

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

Language Level: XPG4

Threading Safe: Yes.
The `fdopen()` function is made available by specifying SYSIFCOPT(*IFSIO) on the compilation command.

The `fdopen()` function associates an input or output stream with the file that is identified by `handle`. The `type` variable is a character string specifying the type of access that is requested for the stream.

**Mode**  **Description**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Create a stream to read a text file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>w</td>
<td>Create a stream to write to a text file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>a</td>
<td>Create a stream to write, in append mode, at the end of the text file. The file pointer is set to the end of the file.</td>
</tr>
<tr>
<td>r+</td>
<td>Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>w+</td>
<td>Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>a+</td>
<td>Create a stream for reading or writing, in append mode, at the end of the text file. The file pointer is set to the end of the file.</td>
</tr>
<tr>
<td>rb</td>
<td>Create a stream to read a binary file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>wb</td>
<td>Create a stream to write to a binary file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>ab</td>
<td>Create a stream to write to a binary file in append mode. The file pointer is set to the end of the file.</td>
</tr>
<tr>
<td>r+b</td>
<td>Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>w+b</td>
<td>Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.</td>
</tr>
<tr>
<td>a+b</td>
<td>Create a stream for reading and writing to a binary file in append mode. The file pointer is set to the end of the file.</td>
</tr>
</tbody>
</table>

**Note:** Use the w, w+, wb, wb+, and w+b modes with care; they can destroy existing files.

The specified `type` must be compatible with the access method you used to open the file. If the file was opened with the O_APPEND flag, the stream mode must be a, a+, ab, a+b, or ab+. To use the `fdopen()` function you need a file descriptor. To get a descriptor use the POSIX function `open()`. The O_APPEND flag is a mode for `open()`. Modes for `open()` are defined in QSYSINC/H/FCNTL. For further information see the APIs topic in the iSeries Information Center.

If `fdopen()` returns NULL, use `close()` to close the file. If `fdopen()` is successful, you must use `fclose()` to close the stream and file.

**Return Value**
The `fdopen()` function returns a pointer to a file structure that can be used to access the open file. A NULL pointer return value indicates an error.

**Example that uses `fdopen()`**

This example opens the file sample.dat and associates a stream with the file using `fdopen()`. It then reads from the stream into the buffer.

```c
/* compile with SYSIFCLOPT(*IFSIO) */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
    long length;
    int fh;
    char buffer[20];
    FILE *fp;

    printf("Creating sample.dat.\n");
    if ((fp = fopen("/sample.dat", "w")) == NULL) {
        perror(" File was not created: ");
        exit(1);
    }
    fputs("Sample Program", fp);
    fclose(fp);
    memset(buffer, '\0', 20); /* Initialize buffer*/

    if (-1 == (fh = open("/sample.dat", O_RDWR|O_APPEND))) {
        perror("Unable to open sample.dat");
        exit(1);
    }
    if (NULL == (fp = fdopen(fh, "r"))) {
        perror("fdopen failed");
        close(fh);
        exit(1);
    }
    if (14 != fread(buffer, 1, 14, fp)) {
        perror("fread failed");
        fclose(fp);
        exit(1);
    }
    printf("Successfully read from the stream the following:\n%s.\n", buffer);
    fclose(fp);
    return 1;
}
```

**Related Information**

- “`fclose()` — Close Stream” on page 82
- “`fopen()` — Open Files” on page 100
- “`fseek()` — `fseeko()` — Reposition File Position” on page 124
- “`fsetpos()` — Set File Position” on page 126
feof() — Test End-of-File Indicator

Format

```
#include <stdio.h>
int feof(FILE *stream);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `feof()` function indicates whether the end-of-file flag is set for the given stream. The end-of-file flag is set by several functions to indicate the end of the file. The end-of-file flag is cleared by calling the `rewind()`, `fsetpos()`, `fseek()`, or `clearerr()` functions for this stream.

Return Value

The `feof()` function returns a nonzero value if and only if the EOF flag is set; otherwise, it returns 0.

Example that uses `feof()`

This example scans the input stream until it reads an end-of-file character.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char string[100];
    FILE *stream;
    memset(string, 0, sizeof(string));
    stream = fopen("qcpple/qacsrc(feof)", "r");
    fscanf(stream, "%s", string);
    while (!feof(stream))
    {
        printf("%s\n", string);
        memset(string, 0, sizeof(string));
        fscanf(stream, "%s", string);
    }
}
```

Related Information

- “`clearerr() — Reset Error Indicators” on page 64
- “`ferror() — Test for Read/Write Errors” on page 87
- “`fseeko() — Reposition File Position” on page 124
- “`fsetpos() — Set File Position” on page 126
- “`perror() — Print Error Message” on page 209
- “`rewind() — Adjust Current File Position” on page 259
ferror() — Test for Read/Write Errors

Format
#include <stdio.h>
int ferror(FILE *stream);

Language Level: ANSI

Threadsafe: Yes.

Description
The ferror() function tests for an error in reading from or writing to the given stream. If an error occurs, the error indicator for the stream remains set until you close stream, call the rewind() function, or call the clearerr() function.

Return Value
The ferror() function returns a nonzero value to indicate an error on the given stream. A return value of 0 means that no error has occurred.

Example that uses ferror()
This example puts data out to a stream, and then checks that a write error has not occurred.

#include <stdio.h>

int main(void)
{
    FILE *stream;
    char *string = "Important information";
    stream = fopen("mylib/myfile", "w");
    fprintf(stream, "%s\n", string);
    if (ferror(stream))
    {
        printf("write error\n");
        clearerr(stream);
    }
    if (fclose(stream))
    perror("fclose error");
}

Related Information
- “clearerr() — Reset Error Indicators” on page 64
- “feof() — Test End-of-File Indicator” on page 86
- “fopen() — Open Files” on page 100
- “perror() — Print Error Message” on page 209
- “strerror() — Set Pointer to Run-Time Error Message” on page 350
- “<stdio.h>” on page 14

fflush() — Write Buffer to File

Format
#include <stdio.h>

fflush(FILE *stream);

Related Information
- “<stdio.h>” on page 14
```c
#include <stdio.h>
int fflush(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `fflush()` function causes the system to empty the buffer that is associated with the specified output `stream`, if possible. If the `stream` is open for input, the `fflush()` function undoes the effect of any `ungetc()` function. The `stream` remains open after the call.

If `stream` is NULL, the system flushes all open streams.

**Note:** The system automatically deletes buffers when you close the stream, or when a program ends normally without closing the stream.

**Return Value**

The `fflush()` function returns the value 0 if it successfully deletes the buffer. It returns EOF if an error occurs.

The value of `errno` may be set to:

**Value** | **Meaning**
---|---
ENOTOPEN | The file is not open.
ERECIO | The file is opened for record I/O.
ESTDIN | `stdin` cannot be opened.
EIOERROR | A non-recoverable I/O error occurred.
EIORECERR | A recoverable I/O error occurred.

The `fflush()` function is not supported for files that are opened with type=record.

**Example that uses `fflush()`**

This example deletes a stream buffer.
Related Information

- “fclose() — Close Stream” on page 82
- “fopen() — Open Files” on page 100
- “setbuf() — Control Buffering” on page 319
- “ungetc() — Push Character onto Input Stream” on page 399
- “<stdio.h>” on page 14

### fgetc() — Read a Character

**Format**

```c
#include <stdio.h>
int fgetc(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `fgetc()` function reads a single unsigned character from the input `stream` at the current position and increases the associated file pointer, if any, so that it points to the next character.

**Note:** The `fgetc()` function is identical to `getc()` but it is always defined as a function call; it is never replaced by a macro.

**Return Value**

The `fgetc()` function returns the character that is read as an integer. An EOF return value indicates an error or an end-of-file condition. Use the `feof()` or the `ferror()` function to determine whether the EOF value indicates an error or the end of the file.

The value of `errno` may be set to:

**Value** | **Meaning**
---|---
...|...
EBADF
The file pointer or descriptor is not valid.

ECONVERT
A conversion error occurred.

ENOTREAD
The file is not open for read operations.

EGETANDPUT
An read operation that was not allowed occurred after a write operation.

ERECIO
The file is open for record I/O.

ESTDIN
stdin cannot be opened.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

The fgetc() function is not supported for files that are opened with type=record.

Example that uses fgetc()

This example gathers a line of input from a stream.

```c
#include <stdio.h>
#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int i, ch;
    stream = fopen("mylib/myfile","r");
    for (i = 0; (i < (sizeof(buffer)-1) &&
        ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        buffer[i] = ch;
    buffer[i] = '\0';
    if (fclose(stream))
        perror("fclose error");
    printf("line: %s\n", buffer);
}
```

If FILENAME contains: one two three
The output should be:
line: one two three

Related Information
- “feof() — Test End-of-File Indicator” on page 86
- “ferror() — Test for Read/Write Errors” on page 87
- “fgetwc() — Read Wide Character from Stream” on page 94
fgetpos() — Get File Position

Format
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);

Language Level: ANSI

Threadsafe: YES

Description
The fgetpos() function stores the current position of the file pointer that is associated with stream into the object pointed to by pos. The value pointed to by pos can be used later in a call to fsetpos() to reposition the stream.

Return Value
The fgetpos() function returns 0 if successful; on error, it returns nonzero and sets errno to a nonzero value.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>EBADSEEK</td>
<td>Bad offset for a seek operation.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation was attempted on a wrong device.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
The `fgetpos()` function is not supported for files that are opened with type=record.

**Example that uses `fgetpos()`**

This example opens the file `myfile` for reading and stores the current file pointer position into the variable `pos`.

```c
#include <stdio.h>

FILE *stream;

int main(void)
{
    int retcode;
    fpos_t pos;

    stream = fopen("mylib/myfile", "rb");

    /* The value returned by fgetpos can be used by fsetpos */
    /* to set the file pointer if 'retcode' is 0 */
    if ((retcode = fgetpos(stream, Point-of-Sale)) == 0)
        printf("Current position of file pointer found\n");
    fclose(stream);
}
```

**Related Information**
- "fseek() — fseeko() — Reposition File Position" on page 124
- "fsetpos() — Set File Position" on page 126
- "ftell() — ftello() — Get Current Position" on page 128
- "<stdio.h>" on page 14

---

**fgets() — Read a String**

**Format**

```c
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

**Language Level**: ANSI

**Threading**: Yes.

**Description**

The `fgets()` function reads characters from the current `stream` position up to and including the first new-line character (`\n`), up to the end of the stream, or until the number of characters read is equal to `n`-1, whichever comes first. The `fgets()` function stores the result in `string` and adds a null character (`\0`) to the end of the string. The `string` includes the new-line character, if read. If `n` is equal to 1, the `string` is empty.

**Return Value**

The `fgets()` function returns a pointer to the `string` buffer if successful. A NULL return value indicates an error or an end-of-file condition. Use the `feof()` or `ferror()` functions to determine whether the NULL value indicates an error or the end of the file. In either case, the value of the string is unchanged.
The `fgets()` function is not supported for files that are opened with type=record.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An operation that was not allowed occurred after a write operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td><code>stdin</code> cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses `fgets()`

This example gets a line of input from a data stream. The example reads no more than `MAX_LEN - 1` characters, or up to a new-line character from the stream.

```c
#include <stdio.h>
#define  MAX_LEN  100

int main(void)
{
    FILE *stream;
    char line[MAX_LEN], *result;
    stream = fopen("mylib/myfile","rb");
    if ((result = fgets(line,MAX_LEN,stream)) != NULL)
        printf("The string is %s\n", result);
    if (fclose(stream))
        perror("fclose error");
}
```

Related Information

- “feof() — Test End-of-File Indicator” on page 86
- “ferror() — Test for Read/Write Errors” on page 87
- “fgetws() — Read Wide-Character String from Stream” on page 96
- “fputs() — Write String” on page 112
- “gets() — Read a Line” on page 146
- “puts() — Write a String” on page 224
- “<stdio.h>” on page 14
fgetwc() — Read Wide Character from Stream

Format
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE *stream);

Language Level: ANSI

Threadsafe: Yes.

Description
The fgetwc() reads the next multibyte character from the input stream pointed to by stream, converts it to a wide character, and advances the associated file position indicator for the stream (if defined).

Using non-wide-character functions with fgetwc() on the same stream results in undefined behavior. After calling fgetwc(), flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling fgetwc().

Notes:
1. The behavior of the fgetwc() function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.
2. This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value
The fgetwc() function returns the next wide character that corresponds to the multibyte character from the input stream pointed to by stream. If the stream is at EOF, the EOF indicator for the stream is set, and fgetwc() returns WEOF.

If a read error occurs, the error indicator for the stream is set, and the fgetwc() function returns WEOF. If an encoding error occurs (an error converting the multibyte character into a wide character), the fgetwc() function sets errno to EILSEQ and returns WEOF.

Use the ferror() and feof() functions to distinguish between a read error and an EOF. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An read operation that was not allowed occurred after a write operation.</td>
</tr>
</tbody>
</table>
ERECIO
The file is open for record I/O.

ESTDIN
stdin cannot be opened.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

EILSEQ
An invalid multibyte character sequence was encountered.

ECONVERT
A conversion error occurred.

Example that uses fgetwc()

This example opens a file, reads in each wide character, and prints out the characters.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    wchar_t wc;
    if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
        printf("Unable to open: \"fgetwc.dat\"\n");
        exit(1);
    }
    errno = 0;
    while (WEOF != (wc = fgetwc(stream)))
    {
        printf("wc = %lc\n", wc);
    }
    fclose(stream);
    return 0;
}
```

Related Information
- "fgetc() — Read a Character” on page 89
- "fputwc() — Write Wide Character” on page 113
- "fgetws() — Read Wide-Character String from Stream” on page 96
- "getc() – getchar() — Read a Character” on page 142
- "getwc() — Read Wide Character from Stream” on page 147
- "getwchar() — Get Wide Character from stdin” on page 149
- "<stdio.h>” on page 14
- "<wchar.h>” on page 17
fgetws() — Read Wide-Character String from Stream

Format

#include <wchar.h>
#include <stdio.h>

wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);

Language Level: ANSI

Threading: Yes.

Description

The fgetws() function reads at most one less than the number of wide characters specified by n from the stream pointed to by stream. The fgetws() function stops reading characters after WEOF, or after it reads a new-line wide character (which is retained). It adds a null wide character immediately after the last wide character read into the array. The fgetws() function advances the file position unless there is an error. If an error occurs, the file position is undefined.

Using non-wide-character functions with the fgetws() function on the same stream results in undefined behavior. After calling the fgetws() function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless WEOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the fgetws() function.

Notes:

1. The behavior of the fgetws() function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur.

2. This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value

If successful, the fgetws() function returns a pointer to the wide-character string wcs. If WEOF is encountered before any wide characters have been read into wcs, the contents of wcs remain unchanged and the fgetws() function returns a null pointer. If WEOF is reached after data has already been read into the string buffer, the fgetws() function returns a pointer to the string buffer to indicate success. A subsequent call would return NULL because WEOF would be reached without any data being read.

If a read error occurs, the contents of wcs are indeterminate, and the fgetws() function returns NULL. If an encoding error occurs (in converting a wide character to a multibyte character), the fgetws() function sets errno to EILSEQ and returns NULL.

If n equals 1, the wcs buffer has only room for the ending null character, and nothing is read from the stream. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless the buffer is flushed or the stream pointer repositioned first.) If n is greater than 1, the fgetws() function fails only if an I/O error occurs, or if WEOF is reached before data is read from the stream.
Use the `ferror()` and `feof()` functions to distinguish between a read error and a WEOF. A WEOF error is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the WEOF indicator.

For information about `errno` values for `fgetws()`, see "fgetwc() — Read Wide Character from Stream" on page 94.

Example that uses `fgetws()`

This example opens a file, reads in the file contents, then prints the file contents.

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE *stream;
    wchar_t wcs[100];

    if (NULL == (stream = fopen("fgetws.dat", "r"))) {
        printf("Unable to open: "fgetws.dat"
    }

    errno = 0;
    if (NULL == fgetws(wcs, 100, stream)) {
        if (EILSEQ == errno) {
            printf("An invalid wide character was encountered.\n");  
            exit(1);
        }
        else if (feof(stream))
            printf("End of file reached.\n");
        else
            perror("Read error.\n");
    }
    printf("wcs = "\n", wcs);
    fclose(stream);
    return 0;
}
```

Assuming the file `fgetws.dat` contains:

This test string should not return -1

The output should be similar to:

`wcs = "This test string should not return -1"
`
fileno() — Determine File Handle

Format

```c
#include <stdio.h>
int fileno(FILE *stream);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `fileno()` function only works in ILE C when the Integrated File System has been enabled. The `fileno()` function determines the file handle that is currently associated with `stream`.

Return Value

If the environment variable QIBM_USE_DESCRIPTOR_STDIO is set to Yes, the `fileno()` function returns 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`.

With QIBM_USE_DESCRIPTOR_STDIO set to No, the ILE C session files `stdin`, `stdout`, and `stderr` do not have a file descriptor associated with them. The `fileno()` function will return a value of -1 in this case.

The value of `errno` can be set to EBADF.

Example that uses `fileno()`

This example determines the file handle of the `stderr` data stream.

```c
/* Compile with SYSIFCOPT(*IFSIO) */
#include <stdio.h>

int main (void)
{
    FILE *fp;
    int result;
    fp = fopen ("stderr","w");
    result = fileno(fp);
    printf("The file handle associated with stderr is %d.\n", result);
    return 0;
}
```

Related Information

- “fopen() — Open Files” on page 100
- “freopen() — Redirect Open Files” on page 120
- “<stdio.h>” on page 14
floor() — Find Integer <= Argument

Format

```c
#include <math.h>
double floor(double x);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `floor()` function calculates the largest integer that is less than or equal to `x`.

Return Value

The `floor()` function returns the floating-point result as a double value. The result of `floor()` cannot have a range error.

Example that uses `floor()`

This example assigns `y` value of the largest integer less than or equal to 2.8 and `z` the value of the largest integer less than or equal to -2.8.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double y, z;
    y = floor(2.8);
    z = floor(-2.8);
    printf("floor(  2.8 ) = %lf
", y);
    printf("floor( -2.8 ) = %lf\n", z);
}
```

```c
/*******************  Output should be similar to:  ***************
floor(  2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
*/
```

Related Information

- “ceil() — Find Integer >= Argument” on page 63
- “fmod() — Calculate Floating-Point Remainder”
- “<math.h>” on page 7

fmod() — Calculate Floating-Point Remainder

Format

```c
#include <math.h>
double fmod(double x, double y);
```

Language Level: ANSI

Threadsafe: Yes.
Description

The `fmod()` function calculates the floating-point remainder of \( x/y \). The absolute value of the result is always less than the absolute value of \( y \). The result will have the same sign as \( x \).

Return Value

The `fmod()` function returns the floating-point remainder of \( x/y \). If \( y \) is zero or if \( x/y \) causes an overflow, `fmod()` returns 0. The value of errno may be set to EDOM.

Example that uses `fmod()`

This example computes \( z \) as the remainder of \( x/y \); here, \( x/y \) is -3 with a remainder of -1.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = -10.0;
    y = 3.0;
    z = fmod(x,y); /* z = -1.0 */
    printf("fmod( %lf, %lf) = %lf\n", x, y, z);
}

/******************** Output should be similar to: ***************
fmod( -10.000000, 3.000000) = -1.000000
*/
```

Related Information

- "ceil() — Find Integer =Argument" on page 63
- "fabs() — Calculate Floating-Point Absolute Value" on page 81
- "floor() —Find Integer <=Argument" on page 99
- "<math.h>" on page 7

**fopen() — Open Files**

Format

```c
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `fopen()` function opens the file that is specified by `filename`. The `mode` parameter is a character string specifying the type of access that is requested for the file. The `mode` variable contains one positional parameter followed by optional keyword parameters.
Note: When the program is compiled with SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO), and fopen() creates an IFS file, the owner of the file, the owner’s group, and public is given read, write, and execute authority to the file.

The possible values for the positional parameters are:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open a text file for reading. The file must exist.</td>
</tr>
<tr>
<td>w</td>
<td>Create a text file for writing. If the given file exists, its contents are destroyed unless it is a logical file.</td>
</tr>
<tr>
<td>a</td>
<td>Open a text file in append mode for writing at the end of the file. The fopen() function creates the file if it does not exist and is not a logical file.</td>
</tr>
<tr>
<td>r+</td>
<td>Open a text file for both reading and writing. The file must exist.</td>
</tr>
<tr>
<td>w+</td>
<td>Create a text file for both reading and writing. If the given file exists, its contents are cleared unless it is a logical file.</td>
</tr>
<tr>
<td>a+</td>
<td>Open a text file in append mode for reading or updating at the end of the file. The fopen() function creates the file if it does not exist.</td>
</tr>
<tr>
<td>rb</td>
<td>Open a binary file for reading. The file must exist.</td>
</tr>
<tr>
<td>wb</td>
<td>Create an empty binary file for writing. If the file exists, its contents are cleared unless it is a logical file.</td>
</tr>
<tr>
<td>ab</td>
<td>Open a binary file in append mode for writing at the end of the file. The fopen function creates the file if it does not exist.</td>
</tr>
<tr>
<td>r+b  or rb+</td>
<td>Open a binary file for both reading and writing. The file must exist.</td>
</tr>
<tr>
<td>w+b  or wb+</td>
<td>Create an empty binary file for both reading and writing. If the file exists, its contents will be cleared unless it is a logical file.</td>
</tr>
<tr>
<td>a+b  or ab+</td>
<td>Open a binary file in append mode for writing at the end of the file. The fopen() function creates the file if it does not exist.</td>
</tr>
</tbody>
</table>

Notes:

1. The fopen() function is not supported for files that are opened with the attributes type=record and ab+, rb+, or wb+
2. Use the w, w+, wb, w+b, and wb+ parameters with care; data in existing files of the same name will be lost.

Text files contain printable characters and control characters that are organized into lines. Each line ends with a new-line character, except possibly the last line, depending on the compiler. The system may insert or convert control characters in an output text stream. The fopen() function mode "a" and "a+" can not be used for the QSYS.LIB file system. There are implementation restrictions when using the QSYS.LIB file system for text files in all modes. Seeking beyond the start of files cannot be relied on to work with streams opened in text mode.

Note: When you use fopen() to create a file in the QSYS.LIB file system, specifying a library name of *LIBL or blank causes the file to be created in QTEMP library.
If a text file does not exist, you can create one using the following command:

```
CRTSRCPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR)
SYSTEM(*FILETYPE)
```

**Note:** Data output to a text stream may not compare as equal to the same data on input. The QSYS.LIB file system treats database files as a directory of members. The database file must exist before a member can be dynamically created when using the `fopen()` function.

See [Large file support](#) in the Files and file systems, Integrated File System topic in the iSeries Information Center for the current file system limit for the Integrated File System. There are three methods you can use to allow your application programs access to 64-bit C runtime functions for IFS files larger than 2 gigabytes:

- Specify `SYSIFCOPT(*IFS64IO)` on a compilation command, which causes the native C compiler to define `_IFS64_IO`. This causes the macros `_LARGE_FILES` and `_LARGE_FILE_API` to be defined.
- Define the macro `_LARGE_FILES`, either in the program source or by specifying `DEFINE(_LARGE_FILES)` on a compilation command. The existing C runtime functions and the relevant data types in the code will all be automatically mapped or redefined to their 64-bit versions.
- Define the macro `_LARGE_FILE_API`, either in the program source or by specifying `DEFINE(_LARGE_FILE_API)` on a compilation command. This makes visible the set of new 64-bit C runtime functions and data types. The application must explicitly specify the name of the C runtime functions, both existing version and 64-bit version, to use.

The 64-bit C runtime functions include the following:

- `fgetpos64()`, `FILE *fopen64()`, `FILE *freopen64()`, `FILE *wfopen64()`, `int fsetpos64(FILE *, const fpost64_t *)`, `FILE *tmpfile64()`, `int fseeko(FILE *, off_t, int)`, `int fseeko64(FILE *, off64_t, int)`, `off_t ftello(FILE *)`, and `off64_t ftello64()`.

**Binary files** contain a series of characters. For binary files, the system does not translate control characters on input or output.

If a binary file does not exist, you can create one using the following command:

```
CRTPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR)
MAXMBRS(*NOMAX) SYSTEM(*FILETYPE)
```

When you open a file with a, a+, ab, a+b or ab+ mode, all write operations take place at the end of the file. Although you can reposition the file pointer using the `fseek()` function or the `rewind()` function, the write functions move the file pointer back to the end of the file before they carry out any operation. This action prevents you from overwriting existing data.

When you specify the update mode (using + in the second or third position), you can both read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as the `fseek()`, `fsetpos()`, `rewind()`, or `fflush()`. Output may immediately follow input if the end-of-file was detected.

**Keyword parameters for non-Integrated File System**

`blksize=value`

Specifies the maximum length, in bytes, of a physical block of records.
lrecl=value
Specifies the length, in bytes, for fixed-length records and the maximum
length for variable-length records.

recfm=value
value can be:
F fixed-length, deblocked records
FB fixed-length, blocked records
V variable-length, deblocked records
VB variable-length, blocked records
VBS variable-length, blocked, spanned records for tape files
VS variable-length, deblocked, spanned records for tape files
D variable-length, deblocked, unspanned records for ASCII D format
for tape files
DB variable-length, blocked, unspanned records for ASCII D format for
tape files
U undefined format for tape files
FA fixed-length that uses first character forms control data for printer
files

Note: If the file is created using CTLCHAR(*FCFC), the first character
form control will be used. If it is created using CTLCHAR(*NONE),
the first character form control will not be used.

commit=value
value can be:
N This parameter identifies that this file is not opened under commitment
control. This is the default.
Y This parameter identifies that this file is opened under commitment
control.

ccsid=value
If a CCSID that is not supported by the iSeries is specified, it is ignored by
data management.

When LOCALETYPE(*LOCALEUTF) is specified on the compilation
command, the default value is the LC_CTYPE CCSID value, which is
determine by your current locale setting. See “setlocale() — Set Locale” on
page 322 for further information about locale settings. When
LOCALETYPE(*LOCALEUTF) is not specified on the compilation
command, the default value is the job CCSID value.

arrseq=value
value can be:
N This parameter identifies that this file is processed in the way it was
created. This is the default.
Y This parameter identifies that this file is processed in arrival sequence.

indicators=value
value can be:
N This parameter identifies that indicators in display, ICF, or printer files are stored in the file buffer. This is the default.

Y This parameter identifies that indicators in display, ICF, or printer files are stored in a separate indicator area, not in the file buffer. A file buffer is the area the iSeries system uses to transfer data to and from the user program and the operating system when writing and reading. You must store indicators in a separate indicator area when processing ICF files.

ty\text{pe}=\text{value}

\text{value} can be:

- \text{memory} This parameter identifies this file as a memory file that is available only from C programs. This is the default.

- \text{record} This parameter specifies that the file is to be opened for sequential record I/O. The file must be opened as a binary file; otherwise, the \text{fopen()} function fails. Read and write operations are done with the \text{fread()} function and the \text{fwrite()} functions.

\textbf{Keyword parameters for Integrated File System only}

\text{type}=\text{value}

\text{value} can be:

- \text{record} The file is opened for sequential record I/O. (File has to be opened as binary stream.)

\text{ccsid}=\text{value}

\text{ccsid} is converted to a code page value. The default is to use the job CCSID value as the code page. The CCSID and codepage option cannot both be specified. The CCSID option provides compatibility with iSeries and Data management based stream I/O.

\textbf{Note:} Mixed data (the data contains both single and double-byte characters) is not supported for a file data processing mode of text. Mixed data is supported for a file processing mode of binary.

If you specify the \text{ccsid} keyword, you cannot specify the \text{o_ccsid} keyword or the codepage keyword.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions such as \text{ftell()}, \text{fseek()}, \text{fgetpos()}, and \text{fsetpos()} will not work. These functions will fail with ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the \text{setvbuf} function with the _IONBF keyword.

The \text{fopen()} function will fail with the ECONVERT error when all of the following three conditions occur:

- The file data processing mode is text.
- The code page is not specified.
- The CCSID of the job is ‘mixed-data’ (the data contains both single– and double–byte characters).
When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the default value is the LC_CTYPE CCSID value, which is determined by your current locale setting. See "setlocale() — Set Locale" on page 322 for further information about locale settings. When LOCALETYPE(*LOCALEUTF) is not specified on the compilation command, the default value is the job CCSID value.

This parameter is similar to the ccsid parameter, except that the value specified is not converted to a code page. Also, mixed data is supported. If the file is created, it is tagged with the specified CCSID. If the file already exists, data will be converted from the CCSID of the file to the specified CCSID on read operations. On write operations, the data is assumed to be in the specified CCSID, and is converted to the CCSID of the file.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions such as ftell(), fseek(), fgetpos(), and fsetpos() will not work. These functions will fail with ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the setvbuf function with the _IONBF keyword.

Example that uses o_ccsid

```c
/* Create a file that is tagged with CCSID 37 */
if ((fp = fopen("/MYFILE", "w, o_ccsid=37")) == NULL) {
    printf("Failed to open file with o_ccsid=37\n");
}
fclose(fp);

/* Now reopen the file with CCSID 13488, because your application wants to deal with the data in UNICODE */
if ((fp = fopen("/MYFILE", "r+, o_ccsid=13488")) == NULL) {
    printf("Failed to open file with o_ccsid=13488\n");
}
/* Turn buffering off because read functions do not work when buffering is on */
if (setbuf(fp, NULL, _IONBF, 0) != 0){
    printf("Unable to turn buffering off\n");
}
/* Because you opened with o_ccsid = 13488, you must provide all input data as unicode. If this program is compiled with LOCALETYPE(*LOCALEUCS2), _L constants will be unicode. */
funcreturn = fputws(L"ABC", fp); /* Write a unicode ABC to the file. */
if (funcreturn < 0) {
    printf("Error with 'fputws' on line %d\n", __LINE__);}
/* Because the file was tagged with CCSID 37, the unicode ABC was converted to EBCDIC ABC when it was written to the file. */
```

codepage=value

The code page that is specified by value is used.
If you specify the codepage keyword, you cannot specify the ccsid keyword or the o_ccsid keyword.

If the file to be opened does not exist, and the open mode specifies that the file should be created, the file is created and tagged with the calculated code page. If the file already exists, the data read from the file is converted from the files code page to the calculated code page during the read operation. Data written to the file is assumed to be in the calculated code page and is converted to the code page of the file during the write operation.

\texttt{crln=value}

\textit{value} can be:

\textbf{Y} The line terminator to be used is carriage return [CR], new line [NL] combination. When data is read, all carriage returns [CR] are stripped for string functions. When data is written to a file, carriage returns [CR] are added before each new line [NL] character. Line terminator processing only occurs when a file is open with text mode. This is the default.

\textbf{N} The line terminator to be used is new line [NL] only.

The keyword parameters are not case sensitive and should be separated by a comma.

The \texttt{fopen()} function generally fails if parameters are mismatched.

\textbf{Return Value}

The \texttt{fopen()} function returns a pointer to a \texttt{FILE} structure type that can be used to access the open file.

\textbf{Note:} To use stream files (type = record) with record I/O functions, you must cast the \texttt{FILE} pointer to an \texttt{RFILE} pointer.

A NULL pointer return value indicates an error.

The value of \texttt{errno} may be set to:

\textbf{Value} \textbf{ Meaning}

\texttt{EBADMODE}

The file mode that is specified is not valid.

\texttt{EBADNAME}

The file name that is specified is not valid.

\texttt{ECONVRT}

Conversion error.

\texttt{ENOENT}

No file or library.

\texttt{ENOMEM}

Storage allocation request failed.

\texttt{ENOTOPEN}

The file is not open.

\texttt{EIOERROR}

A non-recoverable I/O error occurred.
A recoverable I/O error occurred.

The file was marked with a scan failure.

If the mode string passed to fopen() is correct, fopen() will not set errno to EBADMODE, regardless of the file type.

If the mode string that is passed to fopen() is not valid, fopen() will set errno to EBADMODE, regardless of the file type.

If the mode string passed to fopen() is correct, but is invalid to that specific type of file, fopen() will set errno to ENOTOPEN, EIOERROR, or EIORECERR, regardless of the file type.

Example that uses fopen()

This example attempts to open a file for reading.

```c
#include <stdio.h>
#define MAX_LEN  60

int main(void)
{
    FILE *stream;
    fpos_t pos;
    char line1[MAX_LEN];
    char line2[MAX_LEN];
    char *result;
    char ch;
    int num;

    /* The following call opens a text file for reading. */
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
        printf("Could not open data file\n");
    else if ((result = fgets(line1,MAX_LEN,stream)) != NULL)
    {
        printf("The string read from myfile: %s\n", result);
        fclose(stream);
    }

    /* The following call opens a fixed record length file */
    /* for reading and writing. */
    if ((stream = fopen("mylib/myfile2", "rb+, lrecl=80, \
            blksize=240, recfm=f") == NULL)
        printf("Could not open data file\n");
    else {
        fgetpos(stream, Point-of-Sale);
        if (!fread(line2,sizeof(line2),1,stream))
            perror("fread error");
        else printf("1st record read from myfile2: %s\n", line2);

        fsetpos(stream, Point-of-Sale);  /* Reset pointer to start of file */
        fputs(result, stream);  /* The line read from myfile is */
        /* written to myfile2. */
        fclose(stream);
    }
}
```

Related Information

- "fclose() — Close Stream" on page 82
fprintf() — Write Formatted Data to a Stream

Format
#include <stdio.h>
int fprintf(FILE *stream, const char *format-string, argument-list);

Language Level: ANSI

Threadsafe: Yes.

Description

The fprintf() function formats and writes a series of characters and values to the output stream. The fprintf() function converts each entry in argument-list, if any, and writes to the stream according to the corresponding format specification in the format-string.

The format-string has the same form and function as the format-string argument for the printf() function.

Return Value

The fprintf() function returns the number of bytes that are printed or a negative value if an output error occurs.

For information about errno values for fprintf(), see “printf() — Print Formatted Characters” on page 211

Example that uses fprintf()

This example sends a line of asterisks for each integer in the array count to the file myfile. The number of asterisks that are printed on each line corresponds to an integer in the array.
```c
#include <stdio.h>

int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};

int main(void)
{
    int i, j;
    FILE *stream;

    stream = fopen("mylib/myfile", "w");
    /* Open the stream for writing */
    for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fprintf(stream,"*");
        /* Print asterisk */
        fprintf(stream,"\n");
        /* Move to the next line */
    }
    fclose (stream);
}

/******************** Output should be similar to: ********************

* 
***** 
*******
*** 
***** 
******* 
******** 
********** 
*/
```

Related Information

- “fscanf() — Read Formatted Data” on page 123
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 133
- “printf() — Print Formatted Characters” on page 211
- “sprintf() — Print Formatted Data to Buffer” on page 335
- “vfprintf() — Print Argument Data to Stream” on page 404
- “vprintf() — Print Argument Data” on page 411
- “vsprintf() — Print Argument Data to Buffer” on page 415
- “<stdio.h>” on page 14

---

**fputc() — Write Character**

**Format**

```c
#include <stdio.h>
int fputc(int c, FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**
The `fputc()` function converts `c` to an unsigned char and then writes `c` to the output stream at the current position and advances the file position appropriately. If the stream is opened with one of the append modes, the character is appended to the end of the stream.

The `fputc()` function is identical to `putc()`; it always is defined as a function call; it is never replaced by a macro.

**Return Value**

The `fputc()` function returns the character that is written. A return value of EOF indicates an error.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>A write operation that was not permitted occurred after a read operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td><code>stderr</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td><code>stdout</code> cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fputc()` function is not supported for files that are opened with type=record.

**Example that uses `fputc()`**

This example writes the contents of buffer to a file that is called `myfile`.

**Note:** Because the output occurs as a side effect within the second expression of the for statement, the statement body is null.
```c
#include <stdio.h>

#define NUM_ALPHA 26

int main(void)
{
    FILE * stream;
    int i;
    int ch;

    char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if (( stream = fopen("mylib/myfile", "w"))) != NULL )
    {
        /* Put buffer into file */
        for ( i = 0; ( i < sizeof(buffer) ) &&
             ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
        fclose( stream );
    } else
        perror( "Error opening myfile" );
}
```

**Related Information**

- [“fgetc() — Read a Character” on page 89](#)
- [“putc() – putchar() — Write a Character” on page 221](#)
- [“<stdio.h>” on page 14](#)

### _fputchar - Write Character

**Format**

```c
#include <stdio.h>
int _fputchar(int c);
```

**Language Level**: Extension

**Threadsafe**: Yes.

**Description**

_fputchar writes the single character c to the stdout stream at the current position. It is equivalent to the following fputc call:

```c
fputc(c, stdout);
```

For portability, use the ANSI/ISO fputc function instead of _fputchar.

**Return Value**

_fputchar returns the character written. A return value of EOF indicates that a write error has occurred. Use ferror and feof to tell whether this is an error condition or the end of the file.

For information about errno values for _fputchar, see “fputc() — Write Character” on page 109.

**Example that uses _fputchar()**

This example writes the contents of buffer to stdout:
```c
#include <stdio.h>
int main(void)
{
    char buffer[80];
    int i, ch = 1;
    for (i = 0; i < 80; i++)
        buffer[i] = 'c';
    for (i = 0; (i < 80) && (ch != EOF); i++)
        ch = _fputchar(buffer[i]);
    printf("\n");
    return 0;
}
```

The output should be similar to:
```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

**Related Information:**
- [“getc() – getchar() — Read a Character” on page 142](#)
- [“fputc() — Write Character” on page 109](#)
- [“putc() – putchar() — Write a Character” on page 221](#)
- [“<stdio.h>” on page 14](#)

---

**fputs() — Write String**

**Format**
```c
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

**Language Level:** ANSI

**Threatsafe:** Yes.

**Description**

The `fputs()` function copies `string` to the output `stream` at the current position. It does not copy the null character (\0) at the end of the string.

**Return Value**

The `fputs()` function returns EOF if an error occurs; otherwise, it returns a non-negative value.

The `fputs()` function is not supported for files that are opened with type=record.

For information about errno values for `fputs()`, see [“fputc() — Write Character” on page 109](#).

**Example that uses fputs()**

This example writes a string to a stream.
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE *stream;
    int num;

    /* Do not forget that the \0 char occupies one character */
    static char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if ((stream = fopen("mylib/myfile", "w")) != NULL )
    {
        /* Put buffer into file */
        if ( (num = fputs( buffer, stream )) != EOF )
        {
            /* Note that fputs() does not copy the \0 character */
            printf( "Total number of characters written to file = %i\n", num );
            fclose( stream );
        }
        else /* fputs failed */
            perror("fputs failed");
    }
    else
        perror("Error opening myfile");
}

Related Information

- "fgets() — Read a String" on page 92
- "fputws() — Write Wide-Character String" on page 115
- "gets() — Read a Line" on page 146
- "puts() — Write a String" on page 224
- "<stdio.h>" on page 14

### fputwc() — Write Wide Character

**Format**

```c
#include <wchar.h>
#include <stdio.h>
wint_t fputwc(wint_t wc, FILE *stream);
```

**Language Level:** ANSI

**Threading:** Yes.

**Description**

The `fputwc()` function writes the wide character `wc` to the output stream pointed to by `stream` at the current position. It also advances the file position indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the stream.

Using non-wide-character functions with the `fputwc()` function on the same stream will result in undefined behavior. After calling the `fputwc()` function, delete the buffer or reposition the stream pointer before calling a read function for the stream. After reading from the stream, delete the buffer or reposition the stream pointer before calling the `fputwc()` function, unless EOF has been reached.
Notes:
1. The behavior of the `fputwc()` function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur.
2. This function is not available when either LOCALETYPE(*CLD) or SYSIFOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The `fputwc()` function returns the wide character that is written. If a write error occurs, the error indicator for the stream is set, and the `fputwc()` function returns WEOF. If an encoding error occurs during conversion from wide character to a multibyte character, `fputwc()` sets errno to EILSEQ and returns WEOF.

For information about errno values for `putc()`, see "putc() — Write Character" on page 109.

Example that uses `fputwc()`

This example opens a file and uses the `fputwc()` function to write wide characters to the file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"A character string."
    int     i;

    if (NULL == (stream = fopen("fputwc.out", "w")))
    {
        printf("Unable to open: \"fputwc.out\".
exit(1);
    }

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == fputwc(wcs[i], stream))
        {
            printf("Unable to fputwc() the wide character.\n" wcs[\%d] = 0x%.4lx\n\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\nexit(1);
        }
    }
    fclose(stream);
    return 0;
}
```

Related Information

- "fgetwc() — Read Wide Character from Stream" on page 94
fputws() — Write Wide-Character String

Format

```
#include <wchar.h>
#include <stdio.h>
int fputws(const wchar_t *wcs, FILE *stream);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `fputws()` function writes the wide-character string `wcs` to a `stream`. It does not write the ending null wide characters.

Using non-wide-character functions with the `fputws()` function on the same stream will result in undefined behavior. After calling the `fputws()` function, flush the buffer or reposition the stream pointer before calling a read function for the stream. After a read operation, flush the buffer or reposition the stream pointer before calling the `fputws()` function, unless EOF has been reached.

Notes:

1. The behavior of the `fputws()` function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent operations on the same stream, undefined results can occur.

2. This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The `fputws()` function returns a non-negative value if successful. If a write error occurs, the error indicator for the stream is set, and the `fputws()` function returns -1. If an encoding error occurs in converting the wide characters to multibyte characters, the `fputws()` function sets errno to EILSEQ and returns -1.

For information about errno values for `fputws()`, see “fputc() — Write Character” on page 109.

Example that uses `fputws()`

This example opens a file and writes a wide-character string to the file using the `fgetws()` function.
Related Information

- "fgetws() — Read Wide-Character String from Stream" on page 96
- "fputs() — Write String" on page 112
- "fputwc() — Write Wide Character" on page 113
- "puts() — Write a String" on page 224
- "<stdio.h>" on page 14
- "<wchar.h>" on page 17

fread() — Read Items

Format

```c
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `fread()` function reads up to `count` items of `size` length from the input `stream` and stores them in the given `buffer`. The position in the file increases by the number of bytes read.

Return Value

The `fread()` function returns the number of full items successfully read, which can be less than `count` if an error occurs, or if the end-of-file is met before reaching
count. If size or count is 0, the fread() function returns zero, and the contents of the array and the state of the stream remain unchanged.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGETANDPUT</td>
<td>A read operation that was not permitted occurred after a write operation.</td>
</tr>
<tr>
<td>ENOREC</td>
<td>Record is not found.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on the operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Use the ferror() and feof() functions to distinguish between a read error and an end-of-file.

When using fread() for record input, set size to 1 and count to the maximum expected length of the record, to obtain the number of bytes. If you do not know the record length, you should set size to 1 and count to a large value. You can read only one record at a time when using record I/O.

Example that uses fread()

This example attempts to read NUM_ALPHA characters from the file myfile. If there are any errors with either fread() or fopen(), a message is printed.
free() — Release Storage Blocks

Format

```c
#include <stdlib.h>
void free(void *ptr);
```

Language Level: ANSI

Threading: Yes

Description

The `free()` function frees a block of storage. The `ptr` argument points to a block that is previously reserved with a call to the `calloc()`, `malloc()`, `realloc()`, `_C_TS_calloc()`, `_C_TS_malloc()`, `_C_TS_realloc()`, or `_C_TS_malloc64()` functions. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of the `realloc()` function) the block of storage. If `ptr` is NULL, `free()` simply returns.
**Note:** Attempting to free a block of storage not allocated with `calloc()`, `malloc()`, or `realloc()` (or previously freed storage) can affect the subsequent reserving of storage and lead to undefined results. Storage that is allocated with the ILE bindable API CEEGTST can be freed with `free()`.

To use **Teraspace** storage instead of heap storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the CRTCMOD compiler command. This maps the `free()` library function to `_C_TS_free()`, its Teraspace storage counterpart.

**Note:** Whenever an invalid pointer is passed to the `free()` function, a C2M1212 message is signalled. Usually an MCH message associated with the C2M1212 message appears immediately prior to the C2M1212 message in the job log. The MCH message usually has additional information that can be used to debug the problem.

If a C2M1212 message is signalled and the data area QGPL/QC2M1212 exists, the program stack at the point that the C2M1212 message is signalled is spooled to a spool file. You can create the QGPL/QC2M1212 data area using the CRTDTAARA (Create Data Area) command. You can specify any type and length for the data area.

Following are some of the problems that can cause a C2M1212 message to be signalled:

- A pointer that was never set to point to storage reserved by the `malloc()` function is passed to the `free()` function.
- A pointer was set to point to storage reserved by the `malloc()` function, was subsequently modified, and then is passed to the `free()` function.
- A pointer was set to point to storage reserved by the `malloc()` function, was passed to the `free()` function, and then is passed to the `free()` function.

When a C2M1212 message is generated, the hexadecimal value of the pointer passed to the `free()` function is included as part of the message description. This hexadecimal value can provide clues as to the origin of the problem. The `malloc()` function returns only pointers that end in hexadecimal 0. Any pointer that does not end in hexadecimal 0 was either never set to point to storage reserved by the `malloc()` function or was modified since it was set to point to storage reserved by the `malloc()` function. If the pointer ends in hexadecimal 0, then the cause of the C2M1212 message is uncertain, and the program code that calls `free()` should be examined.

**Return Value**

There is no return value.

**Example that uses `free()`**

This example uses the `calloc()` function to allocate storage for `x` array elements, and then calls the `free()` function to free them.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{ 
```

Chapter 2. Library Functions  119
long * array;    /* start of the array */
long * index;    /* index variable */
int i;          /* index variable */
int num;        /* number of entries of the array */

printf( "Enter the size of the array\n" );
scanf( "%i", &num );

/* allocate num entries */
if ( (index = array = calloc( num, sizeof( long ))) != NULL )
{
    for ( i = 0; i < num; ++i )           /* put values in array */
        *index++ = i;                      /* using pointer notation */

    free( array );                        /* deallocates array */
}
else
    { /* Out of storage */
        perror( "Error: out of storage" );
        abort();
    }
}

Related Information
- "calloc() — Reserve and Initialize Storage” on page 57
- "malloc() — Reserve Storage Block” on page 180
- "realloc() — Change Reserved Storage Block Size” on page 247
- "<stdlib.h>” on page 15

freopen() — Redirect Open Files

Format
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);

Language Level: ANSI

Threading Safe: Yes.

Description
The freopen() function closes the file that is currently associated with stream and reassigns stream to the file that is specified by filename. The freopen() function opens the new file associated with stream with the given mode, which is a character string specifying the type of access requested for the file. You can also use the freopen() function to redirect the standard stream files stdin, stdout, and stderr to files that you specify.

For database files, if filename is an empty string, the freopen() function closes and reopens the stream to the new open mode, rather than reassigning it to a new file or device. You can use the freopen() function with no file name specified to change the mode of a standard stream from text to binary without redirecting the stream, for example:

    fp = freopen("", "rb", stdin);

You can use the same method to change the mode from binary back to text.
You cannot use the `freopen()` function with `filename` as an empty string in modules created with `SYSIFCOPT(*IFSIO)`.

You can use the same method to change the mode from binary back to text.

**Return Value**

The `freopen()` function returns a pointer to the newly opened stream. If an error occurs, the `freopen()` function closes the original file and returns a NULL pointer value.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>EBADMODE</td>
<td>The file mode that is specified is not valid.</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>The file name that is specified is not valid.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No file or library.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

**Example that uses `freopen()`**

This example closes the `stream1` data stream and reassigns its stream pointer. `stream1` and `stream2` will have the same value, but they will not necessarily have the same value as `stream`. 
```c
#include <stdio.h>
define MAX_LEN 100

int main(void)
{
  FILE *stream, *stream1, *stream2;
  char line[MAX_LEN], *result;
  int i;

  stream = fopen("mylib/myfile","r");
  if ((result = fgets(line,MAX_LEN,stream)) != NULL)
    printf("The string is %s\n", result);

  /* Change all spaces in the line to '*'. */
  for (i=0; i<sizeof(line); i++)
    if (line[i] == ' ')
      line[i] = '*';

  stream1 = stream;
  stream2 = freopen("", "w+", stream1);
  fputs( line, stream2 );
  fclose( stream2);
}
```

**Related Information**
- "fclose() — Close Stream" on page 82
- "fopen() — Open Files" on page 100
- "<stdio.h>" on page 14

---

### frexp() — Separate Floating-Point Value

**Format**
```
#include <math.h>
double frexp(double x, int *expptr);
```

**Language Level**: ANSI

**Threadingafe**: Yes.

**Description**

The `frexp()` function breaks down the floating-point value `x` into a term `m` for the mantissa and another term `n` for the exponent. It is done such that `x=m*2^n`, and the absolute value of `m` is greater than or equal to 0.5 and less than 1.0 or equal to 0. The `frexp()` function stores the integer exponent `n` at the location to which `expptr` points.

**Return Value**

The `frexp()` function returns the mantissa term `m`. If `x` is 0, `frexp()` returns 0 for both the mantissa and exponent. The mantissa has the same sign as the argument `x`. The result of the `frexp()` function cannot have a range error.

**Example that uses frexp()**

This example separates the floating-point value of `x`, 16.4, into its mantissa 0.5125, and its exponent 5. It stores the mantissa in `y` and the exponent in `n`. 
```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, m;
    int n;
    x = 16.4;
    m = frexp(x, n);
    printf("The mantissa is \%lf and the exponent is \%d\n", m, n);
}

/*************** Output should be similar to: **************
The mantissa is 0.512500 and the exponent is 5
*/
```

Related Information
- "ldexp() — Multiply by a Power of Two" on page 166
- "modf() — Separate Floating-Point Value" on page 205
- "<math.h>" on page 7

---

**fscanf() — Read Formatted Data**

**Format**

```c
#include <stdio.h>
int fscanf (FILE *stream, const char *format-string, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `fscanf()` function reads data from the current position of the specified stream into the locations that are given by the entries in argument-list, if any. Each entry in argument-list must be a pointer to a variable with a type that corresponds to a type specifier in format-string.

The format-string controls the interpretation of the input fields and has the same form and function as the format-string argument for the `scanf()` function.

**Return Value**

The `fscanf()` function returns the number of fields that it successfully converted and assigned. The return value does not include fields that the `fscanf()` function read but did not assign.

The return value is EOF if an input failure occurs before any conversion, or the number of input items assigned if successful.

**Example that uses fscanf()**

This example opens the file myfile for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.
#include <stdio.h>

#define  MAX_LEN  80

int main(void) {
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;

    stream = fopen("mylib/myfile", "r");

    /* Put in various data. */
    fscanf(stream, "%s", &s[0]);
    fscanf(stream, "%ld", &l);
    fscanf(stream, "%c", &c);
    fscanf(stream, "%f", &fp);

    printf("string = %s
", s);
    printf("long double = %ld
", l);
    printf("char = %c
", c);
    printf("float = %f
", fp);
}

/**
 * If myfile contains
 ** abcdedefghijklmnopqrstuvwxyz 343.2 
 **
 * expected output is: 
 **
 * string = abcdefghijklmnopqrstuvwxyz
 * long double = 343
 * char = .
 * float = 2.000000
 */

Related Information
- "fprintf() — Write Formatted Data to a Stream" on page 108
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "scanf() — Read Data" on page 313
- "sscanf() — Read Data" on page 339
- "swscanf() — Read Wide Character Data" on page 387
- "wscanf() — Read Data Using Wide-Character Format String" on page 482
- "<stdio.h>" on page 14

fseek() — fseeko() — Reposition File Position

Format
#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
int fseeko(FILE *stream, off_t offset, int origin);

Language Level: ANSI

Threading: Yes.

Description
The fseek() and fseeko() functions change the current file position that is associated with stream to a new location within the file. The next operation on stream takes place at the new location. On a stream open for update, the next operation can be either a reading or a writing operation.

The fseeko() function is identical to fseek() except that the offset argument is of type off_t.

The origin must be one of the following constants that are defined in <stdio.h>:

**Origin Definition**

- **SEEK_SET**
  Beginning of file

- **SEEK_CUR**
  Current position of file pointer

- **SEEK_END**
  End of file

For a binary stream, you can also change the position beyond the end of the file. An attempt to position before the beginning of the file causes an error. If successful, the fseek() or fseeko() function clears the end-of-file indicator, even when origin is SEEK_END, and undoes the effect of any preceding the ungetc() function on the same stream.

**Note:** For streams opened in text mode, the fseek() and fseeko() functions have limited use because some system translations (such as those between carriage-return-line-feed and new line) can produce unexpected results. The only fseek() and fseeko() operations that can be relied upon to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to the ftell() or ftello() functions. Calls to the ftell() and ftello() functions are subject to their restrictions.

**Return Value**

The fseek() or fseeko function returns 0 if it successfully moves the pointer. A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

The value of errno may be set to:

**Value**  **Meaning**

- **EBADF**
  The file pointer or descriptor is invalid.

- **EBADSEEK**
  Bad offset for a seek operation.

- **ENODEV**
  Operation was attempted on a wrong device.

- **ENOTOPEN**
  The file is not open.

- **ERECHO**
  The file is open for record I/O.
ESTDERR
stderr cannot be opened.

ESTDIN
stdin cannot be opened.

ESTDOUT
stdout cannot be opened.

EIOERROR
A non-recoverable I/O error occurred.

EIORECCERR
A recoverable I/O error occurred.

The fseek() and fseko() functions are not supported for files that are opened with type=record.

Example that uses fseek()

This example opens a file myfile for reading. After performing input operations, fseek() moves the file pointer to the beginning of the file.

```
#include <stdio.h>
#define MAX_LEN 10

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int result;
    int i;
    char ch;
    
    stream = fopen("mylib/myfile", "r");
    for (i = 0; (i < (sizeof(buffer)-1) &&
        ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        buffer[i] = ch;
    
    result = fseek(stream, 0L, SEEK_SET); /* moves the pointer to the */
    /* beginning of the file */
    if (result == 0)
        printf("Pointer successfully moved to the beginning of the file.\n");
    else
        printf("Failed moving pointer to the beginning of the file.\n");

    return 0;
}
```

Related Information
- "ftell() — ftello() — Get Current Position” on page 128
- "fgetpos() — Get File Position" on page 91
- "fsetpos() — Set File Position"
- "rewind() — Adjust Current File Position” on page 259
- "ungetc() — Push Character onto Input Stream” on page 399
- "fseek() — fseko() — Reposition File Position” on page 124
- "<stdio.h>” on page 14

fsetpos() — Set File Position

Format
```c
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `fsetpos()` function moves any file position that is associated with `stream` to a new location within the file according to the value pointed to by `pos`. The value of `pos` was obtained by a previous call to the `fgetpos()` library function.

If successful, `fsetpos()` clears the end-of-file indicator, and undoes the effect of any previous `ungetc()` function on the same stream.

After the `fsetpos()` call, the next operation on a stream in update mode may be input or output.

**Return Value**

If `fsetpos()` successfully changes the current position of the file, it returns 0. A nonzero return value indicates an error.

The value of `errno` may be set to:

**Value    Meaning**

**EBADF**

The file pointer or descriptor is invalid.

**EBADPOS**

The position that is specified is not valid.

**EINVAL**

The value specified for the argument is not correct. You may receive this `errno` when you compile your program with `*IFSIO`, and you are working with a file in the QSYS file system. For example, `/qsys.lib/qtemp.lib/myfile.file/mymem.mbr`

**ENODEV**

Operation was attempted on a wrong device.

**ENOPOS**

No record at the specified position.

**ERECIO**

The file is open for record I/O.

**ESTDERR**

`stderr` cannot be opened.

**ESTDIN**

`stdin` cannot be opened.

**ESTDOUT**

`stdout` cannot be opened.

**EIOERROR**

A non-recoverable I/O error occurred.

---

Chapter 2. Library Functions  127
A recoverable I/O error occurred.

The `fsetpos()` function cannot be used for files that are opened with type=record. Also, the `fsetpos()` function can only support setting the position to the beginning of the file if:

- your program is compiled with *IFSIO, and
- you are working on a file in the QSYS file system.

**Example that uses fsetpos()**

This example opens a file mylib/myfile for reading. After performing input operations, `fsetpos()` moves the file pointer to the beginning of the file and rereads the first byte.

```c
#include <stdio.h>

FILE *stream;
int main(void)
{
    int retcode;
    fpos_t pos;
    char ptr[20];  /* existing file 'mylib/myfile' has 20 byte records */
    int i;
    /* Open file, get position of file pointer, and read first record */
    stream = fopen("mylib/myfile", "rb");
    fgetpos(stream,Point-of-Sale);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record: %s\n", ptr);
    /* Perform another read operation on the second record */
    /* - the value of 'pos' changes */
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("2nd record: %s\n", ptr);
    /* Re-set pointer to start of file and re-read first record */
    fsetpos(stream,Point-of-Sale);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record again: %s\n", ptr);
    fclose(stream);
}
```

**Related Information**
- "fgetpos() — Get File Position" on page 91
- "fseek() — fseeko() — Reposition File Position" on page 124
- "ftell() — ftello() — Get Current Position"
- "rewind() — Adjust Current File Position" on page 259
- "<stdio.h>” on page 14

### ftell() — ftello() — Get Current Position

**Format**

```c
#include <stdio.h>

long int ftell(FILE *stream);
off_t ftello(FILE *stream);
```
Language Level: ANSI

Threadsafe: Yes.

Description

The ftell() and ftello() functions find the current position of the file associated with stream. For a fixed-length binary file, the value that is returned is an offset relative to the beginning of the stream.

For file in the QSYS library system, the ftell() and ftello() functions return a relative value for fixed-format binary files and an encoded value for other file types. This encoded value must be used in calls to the fseek() and fseeko() functions to positions other than the beginning of the file.

Note: The ftello() function is available only when SYSIFCOPT(*IFSIO) is specified on the compilation command.

Return Value

The ftell() and ftello() functions return the current file position. On error, ftell() and ftello() return -1, cast to long and off_t respectively, and set errno to a nonzero value.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENODEV</td>
<td>Operation was attempted on a wrong device.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ENUMMBRS</td>
<td>The file is open for multi-member processing.</td>
</tr>
<tr>
<td>ENUMRECS</td>
<td>Too many records.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The ftell() and ftello() functions are not supported for files that are opened with type=record.
Example that uses ftell(

This example opens the file mylib/myfile for reading. It reads enough characters to fill half of the buffer and prints out the position in the stream and the buffer.

```c
#include <stdio.h>

#define NUM_ALPHA 26
#define NUM_CHAR 6

int main(void)
{
    FILE *stream;
    int i;
    char ch;

    char buffer[NUM_ALPHA];
    long position;

    if ((stream = fopen("mylib/myfile", "r")) != NULL )
    {
        /* read into buffer */
        for (i = 0; (i < NUM_ALPHA/2) && ((buffer[i] = fgetc(stream)) != EOF); ++i)
        {
            if (i==NUM_CHAR-1) /* We want to be able to position the */
                /* file pointer to the character in */
                /* position NUM_CHAR */
                position = ftell(stream);

            buffer[i] = '\0';
        } printf("Current file position is %ld\n", position);
        printf("Buffer contains: %s\n", buffer);
    }
}
```

Related Information
- "fseek() — fseeko() — Reposition File Position" on page 124
- "fgetpos() — Get File Position" on page 91
- "fopen() — Open Files" on page 100
- "fsetpos() — Set File Position" on page 126
- "ftell() — ftello() — Get Current Position" on page 128
- "<stdio.h>" on page 14

fwide() — Determine Stream Orientation

Format
```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Language Level: ANSI

Threadsafe: Yes.

Description
The fwide() function determines the orientation of the stream pointed to by stream. If mode is greater than 0, the fwide() function first attempts to make the stream wide oriented. If mode is less than 0, the fwide() function first attempts to make the stream byte oriented. Otherwise, mode is 0, and the fwide() function does not alter the orientation of the stream.
Notes:
1. If the orientation of the stream has already been determined, the `fwide()` function does not change it.
2. This function is not available when either `LOCALETYPExC1D)` or `SYSIFCOPTExNOIFSIO)` is specified on the compilation command.

Return Value

If, after the call, the stream has wide orientation, the `fwide()` function returns a value greater than 0. If the stream has byte orientation, it returns a value less than 0. If the stream has no orientation, it returns 0.

Example that uses `fwide()`
#include <stdio.h>
#include <math.h>
#include <wchar.h>

void check_orientation(FILE *stream)
{
    int rc;
    rc = fwide(stream,0); /* check the orientation */
    if (rc<0) {
        printf("Stream has byte orientation.\n");
    } else if (rc>0) {
        printf("Stream has wide orientation.\n");
    } else {
        printf("Stream has no orientation.\n");
    }
    return;
}

int main(void)
{
    FILE *stream;
    /* Demonstrate that fwide can be used to set the orientation,
       but cannot change it once it has been set. */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwide(stream, -1); /* Make the stream byte oriented */
    printf("After fwide(stream, -1): ");
    check_orientation(stream);
    fwide(stream, 1); /* Try to make the stream wide oriented */
    printf("After fwide(stream, 1): ");
    check_orientation(stream);
    fclose(stream);
    printf("Close the stream\n");
    /* Check that a wide character output operation sets the orientation
       as expected. */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwprintf(stream, L"pi = %.5f\n", 4* atan(1.0));
    printf("After fwprintf( ): ");
    check_orientation(stream);
    fclose(stream);
    return 0;
}

The output should be similar to :
After opening the file: Stream has no orientation.
After fwide(stream, -1): Stream has byte orientation.
After fwide(stream, 1): Stream has byte orientation.
Close the stream
After opening the file: Stream has no orientation.
After fwprintf( ): Stream has wide orientation.
******************************************************************************

Related Information

- “fgetwc() — Read Wide Character from Stream” on page 94
- “fgetws() — Read Wide-Character String from Stream” on page 96
- “fputwc() — Write Wide Character” on page 113
- “fputws() — Write Wide-Character String” on page 115
- “<stdio.h>” on page 14
- “<wchar.h>” on page 17
fwprintf() — Format Data as Wide Characters and Write to a Stream

Format

#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, argument-list);

Language Level: ANSI

Threadsafe: Yes.

Description

The fwprintf() function writes output to the stream pointed to by stream, under control of the wide string pointed to by format. The format string specifies how subsequent arguments are converted for output.

The fwprintf() function converts each entry in argument-list according to the corresponding wide-character format specifier in format.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the fwprintf() function evaluates the excess arguments, but otherwise ignores them. The fwprintf() function returns when it encounters the end of the format string.

The format comprises zero or more directives: ordinary wide characters (not %) and conversion specifications. Conversion specifications are processed as if they were replaced in the format string by wide-character strings. The wide-character strings are the result of fetching zero or more subsequent arguments and then converting them, if applicable, according to the corresponding conversion specifier. The fwprintf() function then writes the expanded wide-character format string to the output stream.

The format for the fwprintf() function has the same form and function as the format string for printf(), with the following exceptions:

- %%c (without an l prefix) converts an integer argument to wchar_t, as if by calling the btowc() function.
- %%s (without an l prefix) converts an array of multibyte characters to an array of wchar_t, as if by calling the mbtowc() function. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.
- %%ls and %%S write an array of wchar_t. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, an union or an aggregate (except for an array of char type using %s conversion, an array of wchar_t type using %%ls conversion, or a pointer using %p conversion), the behavior is undefined.

In no case does a nonexistent, or small field width, cause truncation of a field; if the conversion result is wider than the field width, the field is expanded to contain the conversion result.
Note: This function is not available when either LOCALETYPE(*CLD) or SYSFIOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The `fwprintf()` function returns the number of wide characters transmitted. If an output error occurred, it returns a negative value.

Example that uses `fwprintf()`

Note: When writing wide characters, the file should be opened in binary mode, or the file must be opened with the `o CCSID` or `codepage` parameters. This ensures that no conversions occur on your wide characters. Since there are no valid double-byte job CCSIDS on the iSeries, any conversion would be invalid.

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};
int main(void)
{
    int i,j;
    FILE *stream; /* Open the stream for writing */
    if (NULL == (stream = fopen("/QSYS.LIB/LIB.LIB/WCHAR.FILE/WCHAR.MBR","wb")))
        perror("fopen error");
    for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fwprintf(stream, L"*");  /* Print asterisk */
        fwprintf(stream, L"
");  /* Move to the next line */
    }
    fclose (stream);
}
/* The member WCHAR of file WCHAR will contain:

* 
***** 
********
*** 
**** 
******
*******
********** */
```

Unicode example that uses `fwprintf()`
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALEUCS2) and SYSIFOPT(*IFSIO) */
int main(void)
{
    FILE *stream;
    wchar_t wc = 0x0058; /* UNICODE X */
    char c1 = 'c';
    char *s1 = "123";
    wchar_t ws[4];
    setlocale(LC_ALL, 
            "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041; /* UNICODE A */
    ws[1] = (wchar_t)0x0042; /* UNICODE B */
    ws[2] = (wchar_t)0x0043; /* UNICODE C */
    ws[3] = (wchar_t)0x0000;
    stream = fopen("myfile.dat", "wb+");
    /* 1c and 1s take wide char as input and just copies then */
    /* to the file. So the file would look like this */
    /* after the below fwprintf statement: */
    /* 0058002000200020004100420043 */
    /* 0020 is UNICODE blank */
    fwprintf(stream, L"%lc   %ls", wc, ws);
    /* c and s take multibyte as input and produce UNICODE */
    /* In this case c1 and s1 are CCSID 37 characters based */
    /* on the setlocale above. So the characters are */
    /* converted from CCSID 37 to UNICODE and will look */
    /* like this in hex after the following fwprintf */
    /* statement: 0663002000200020003100320033 */
    /* 0063 is a UNICODE c 0031 is a UNICODE 1 and so on */
    fwprintf(stream, L"%c   %s", c1, s1);
    /* Now lets try width and precision. 6ls means write */
    /* 6 wide characters so we will pad with 3 UNICODE */
    /* blanks and %.2s means write no more then 2 wide */
    /* characters. So we get an output that looks like */
    /* this: 00200020002000410042004300310032 */
    fwprintf(stream, L"%6ls%.2s", ws, s1);
}

Related Information
- “fprintf() — Write Formatted Data to a Stream” on page 108
- “printf() — Print Formatted Characters” on page 211
- “vfprintf() — Print Argument Data to Stream” on page 404
- “vprintf() — Print Argument Data” on page 411
- “btowc() — Convert Single Byte to Wide Character” on page 55
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 407
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 418
- “wprintf() — Format Data as Wide Characters and Print” on page 481
- "<stdarg.h>” on page 13
- "<wchar.h>” on page 17
fwrite() — Write Items

Format

```c
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
              FILE *stream);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `fwrite()` function writes up to `count` items, each of `size` bytes in length, from `buffer` to the output `stream`.

Return Value

The `fwrite()` function returns the number of full items successfully written, which can be fewer than `count` if an error occurs.

When using `fwrite()` for record output, set `size` to 1 and `count` to the length of the record to obtain the number of bytes written. You can only write one record at a time when using record I/O.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>EPAD</td>
<td>Padding occurred on a write operation.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td><code>stderr</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td><code>stdin</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td><code>stdout</code> cannot be opened.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses `fwrite()`

This example writes NUM long integers to a stream in binary format.
fwscanf() — Read Data from Stream Using Wide Character

Format
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, argument-list);

Language Level: ANSI

Threadsafe: Yes.

Description

The fwscanf() function reads input from the stream pointed to by stream, under control of the wide string pointed to by format. The format string specifies the admissible input sequences and how they are to be converted for assignment. To receive the converted input, the fwscanf() function uses subsequent arguments as pointers to the objects.

Each argument in argument-list must point to a variable with a type that corresponds to a type specifier in format.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the fwscanf() function evaluates the excess arguments, but otherwise ignores them.

The format consists of zero or more directives: one or more white-space wide characters; an ordinary wide character (neither % nor a white-space wide character); or a conversion specification. Each conversion specification is introduced by a %.

The format has the same form and function as the format string for the scanf() function, with the following exceptions:

#include <stdio.h>
define NUM 100

int main(void)
{
    FILE *stream;
    long list[NUM];
    int numwritten;
    int i;

    stream = fopen("MYLIB/MYFILE", "w+b");

    /* assign values to list[] */
    for (i=0; i<=NUM; i++)
        list[i]=i;

    numwritten = fwrite(list, sizeof(long), NUM, stream);
    printf("Number of items successfully written = %d\n", numwritten);
}
%c (with no l prefix) converts one or more wchar_t characters (depending on precision) to multibyte characters, as if by calling wcrtomb().

%lc and %C convert one or more wchar_t characters (depending on precision) to an array of wchar_t.

%s (with no l prefix) converts a sequence of non-white-space wchar_t characters to multibyte characters, as if by calling the wcrtomb() function. The array includes the ending null character.

%ls and %S copy an array of wchar_t, including the ending null wide character, to an array of wchar_t.

If the data is from stdin, and stdin has not been overridden, the data is assumed to be in the CCSID of the job. The data is converted as required by the format specifications. If the file that is being read is not opened with file mode rb, then invalid conversion can occur as there are no valid wide job CCSIDs.

If a conversion specification is invalid, the behavior is undefined. If the fscanf() function encounters end-of-file during input, conversion is ended. If end-of-file occurs before the fscanf() function reads any characters matching the current directive (other than leading white space, where permitted), execution of the current directive ends with an input failure. Otherwise, unless execution of the current directive terminates with a matching failure, execution of the following directive (other than %n, if any) ends with an input failure.

The fscanf() function leaves trailing white space (including new-line wide characters) unread, unless matched by a directive. You cannot determine the success of literal matches and suppressed assignments other than through the %n directive.

Note: This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The fscanf() function returns the number of input items assigned, which can be fewer than provided for, in the event of an early matching failure.

If an input failure occurs before any conversion, the fscanf() function returns EOF.

Example that uses fscanf()

This example opens the file myfile.dat for input, and then scans this file for a string, a long integer value, a character, and a floating point value.
This example reads a Unicode string from `unicode.dat` and prints it to the screen.
The example is compiled with `LOCALETYPEn(LOCALEUCS2) SYSIFCOPTn(IFSIO):`
gamma() — Gamma Function

Format
#include <math.h>
double gamma(double x);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The gamma() function computes the natural logarithm of the absolute value of G(x) (\(\ln(\lvert G(x) \rvert)\)), where

\[
G(x) = \int_{0}^{\infty} e^{-t} \times t^{x-1} \, dt
\]

The argument \(x\) must be a positive real value.

Return Value
The gamma() function returns the value of \( \ln(|G(x)|) \). If \( x \) is a negative value, errno is set to EDOM. If the result causes an overflow, gamma() returns HUGE_VAL and sets errno to ERANGE.

Example that uses gamma()

This example uses gamma() to calculate \( \ln(|G(x)|) \), where \( x = 42 \).

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x=42, g_at_x;
    g_at_x = exp(gamma(x)); /* g_at_x = 3.345253e+49 */
    printf("The value of G(44.21f) is %7.2e\n", x, g_at_x);
}
```

Related Information
- “Bessel Functions” on page 52
- “erf() – erfc() — Calculate Error Functions” on page 79
- “<math.h>” on page 7

_gcvt - Convert Floating-Point to String

Format
```c
#include <stdlib.h>
char *gcvt(double value, int ndec, char *buffer);
```

Note: The _gcvt function is supported only for C++, not for C.

Language Level: Extension

Threading: Yes.

Description

_gcvt() converts a floating-point value to a character string pointed to by buffer. The buffer should be large enough to hold the converted value and a null character (\0) that _gcvt() automatically adds to the end of the string. There is no provision for overflow.

_gcvt() attempts to produce ndec significant digits in FORTRAN F format. Failing that, it produces ndec significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion if they are insignificant.

A FORTRAN F number has the following format:

```
  +----+    +----+    +----+
  | digit|    | digit|    | digit|
  +----+    +----+    +----+
```
A FORTRAN E number has the following format:

```
-digit-.digit-E-digit-digit
```

_gcvt also converts NaN and infinity values to the strings NAN and INFINITY, respectively.

Return Value

_gcvt() returns a pointer to the string of digits. If it cannot allocate memory to perform the conversion, _gcvt() returns an empty string and sets errno to ENOMEM.

Example that uses _gcvt()

This example converts the value -3.1415e3 to a character string and places it in the character array buffer1.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer1[10];
    _gcvt(-3.1415e3, 7, buffer1);
    printf("The first result is %s \n", buffer1);
    return 0;
}
```

The output should be:

```
The first result is -3141.5
```

Related Information:

- Infinity and NaN Support
- `<stdlib.h>` on page 15

getc() – getchar() — Read a Character

Format

```c
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

Language Level: ANSI

Threadsafe: No. #undef getc or #undef getchar allows the getc or getchar function to be called instead of the macro version of these functions. The functions are threadsafe.

Description

The getc() function reads a single character from the current stream position and advances the stream position to the next character. The getchar() function is identical to getc(stdin).
The difference between the getc() and fgetc() functions is that getc() can be implemented so that its arguments can be evaluated multiple times. Therefore, the stream argument to getc() should not be an expression with side effects.

Return Value

The getc() and getchar() functions return the character read. A return value of EOF indicates an error or end-of-file condition. Use ferror() or feof() to determine whether an error or an end-of-file condition occurred.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An illegal read operation occurred after a write operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The getc() and getchar() functions are not supported in record mode.

Example that uses getc()

This example gets a line of input from the stdin stream. You can also use getc(stdin) instead of getchar() in the for statement to get a line of input from stdin.
```
#include <stdio.h>
#define LINE 80

int main(void)
{
    char buffer[LINE+1];
    int i;
    int ch;

    printf( "Please enter string\n" );

    /* Keep reading until either:
     1. the length of LINE is exceeded or
     2. the input character is EOF or
     3. the input character is a new-line character
    */

    for ( i = 0; ( i < LINE ) && (( ch = getchar()) != EOF) &&
         ( ch !='\n' ); ++i )
        buffer[i] = ch;

    buffer[i] = '\0';  /* a string should always end with '\0' ! */

    printf( "The string is %s\n", buffer );
}
```

getenv() — Search for Environment Variables

Format

```c
#include <stdlib.h>
char *getenv(const char *varname);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The getenv() function searches the list of environment variables for an entry corresponding to *varname.*

Note: This function accepts and returns only EBCDIC character strings. When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the user may need to convert any non-EBCDIC character strings to EBCDIC before calling this function and convert any resulting EBCDIC character strings to the correct CCSID.
Return Value

The `getenv()` function returns a pointer to the string containing the value for the specified `varname` in the current environment. If `getenv()` cannot find the environment string, NULL is returned, and errno is set to indicate the error.

Example that uses `getenv()`

```c
#include <stdlib.h>
#include <stdio.h>

/* Where the environment variable 'PATH' is set to a value. */
int main(void)
{
    char *pathvar;
    pathvar = getenv("PATH");
    printf("pathvar=%s",pathvar);
}
```

Related Information

- "<stdlib.h>" on page 15
- "putenv() — Change/Add Environment Variables" on page 223
- Environment Variable APIs in the [APIs] topic in the iSeries Information Center.

_GetExcData() — Get Exception Data

Format

```c
#include <signal.h>
void _GetExcData(_INTRPT_Hndlr_Parms_T *parms);
```

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The `_GetExcData()` function returns information about the current exception from within a C signal handler. The caller of the `_GetExcData()` function must allocate enough storage for a structure of type `_INTRPT_Hndlr_Parms_T`. If the `_GetExcData()` function is called from outside a signal handler, the storage pointed to by `parms` is not updated.

This function is not available when SYSIFCOPT(*ASYNCSIGNAL) is specified on the compilation commands. When SYSIFCOPT(*ASYNCSIGNAL) is specified, a signal handler established with the ILE C signal() function has no way to access any exception information that may have caused the signal handler to be invoked. An extended signal handler established with the `sigaction()` function, however, does have access to this exception information. The extended signal handler has the following function prototype:

```c
void func( int signo, siginfo_t *info, void *context )
```

The exception information is appended to the `siginfo_t` structure, which is then passed as the second parameter to the extended signal handler.
The `siginfo_t` structure is defined in `signal.h`. The exception-related data follows the `si_sigdata` field in the `siginfo_t` structure. You can address it from the `se_data` field of the `sigdata_t` structure.

The format of the exception data appended to the `siginfo_t` structure is defined by the `_INTRPT_Hndlr_Parms_T` structure in `except.h`.

**Note:** The character data returned within the exception data structure is in EBCDIC. When `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command, the user may need to convert any resulting EBCDIC character strings to the correct CCSID.

**Return Value**

There is no return value.

**Example that uses _GetExcData()**

This example shows how exceptions from MI library functions can be monitored and handled using a signal handling function. The signal handler `my_signal_handler` is registered before the `rslvsp()` function signals a 0x2201 exception. When a SIGSEGV signal is raised, the signal handler is called. If an 0x2201 exception occurred, the signal handler calls the QUSRCRTS API to create a space.

```c
#include <signal.h>
#include <QSYSINC/MIH/RSLVSP>
#include <QSYSINC/H/QUSCRTUS>
#include <string.h>

#define CREATION_SIZE 65500

void my_signal_handler(int sig) {
    _INTRPT_Hndlr_Parms_T excp_data;
    int error_code = 0;

    /* Check the message id for exception 0x2201 */
    _GetExcData(&excp_data);

    if (!memcmp(excp_data.Msg_Id, "MCH3401", 7))
        QUSCRTUS("MYSPACE QTEMP " ,
                 "MYSPACE " ,
                 CREATION_SIZE,
                 "\0",
                 "*ALL ",
                 "MYSAPCE example for Programmer's Reference ",
                 "*YES ",
                 &error_code);
}
```

**Related Information**

- "signal() — Handle Interrupt Signals" on page 330
- "<except.h>" on page 4

---

**gets() — Read a Line**

**Format**
Language Level: ANSI

Threadsafe: Yes.

Description

The `gets()` function reads a line from the standard input stream `stdin` and stores it in `buffer`. The line consists of all characters up to but not including the first new-line character (\n) or EOF. The `gets()` function then replaces the new-line character, if read, with a null character (\0) before returning the line.

Return Value

If successful, the `gets()` function returns its argument. A NULL pointer return value indicates an error, or an end-of-file condition with no characters read. Use the `ferror()` function or the `feof()` function to determine which of these conditions occurred. If there is an error, the value that is stored in `buffer` is undefined. If an end-of-file condition occurs, `buffer` is not changed.

Example that uses `gets()`

This example gets a line of input from `stdin`.

```c
#include <stdio.h>

#define MAX_LINE 100

int main(void)
{
    char line[MAX_LINE];
    char *result;

    printf("Please enter a string:\n");
    if ((result = gets(line)) != NULL)
        printf("The string is: %s\n", line);
    else if (ferror(stdin))
        perror("Error");
}
```

Related Information

- "fgets() — Read a String" on page 92
- "fgetws() — Read Wide-Character String from Stream” on page 96
- “feof() — Test End-of-File Indicator” on page 86
- “ferror() — Test for Read/Write Errors” on page 87
- “fputs() — Write String” on page 112
- “getc() — getchar() — Read a Character” on page 142
- “puts() — Write a String” on page 224
- “<stdio.h>” on page 14
```c
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `getwc()` function reads the next multibyte character from stream, converts it to a wide character, and advances the associated file position indicator for stream.

The `getwc()` function is equivalent to the `fgetwc()` function except that, if it is implemented as a macro, it can evaluate stream more than once. Therefore, the argument should never be an expression with side effects.

The behavior of the `getwc()` function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur. Using non-wide-character functions with the `getwc()` function on the same stream results in undefined behavior.

After calling the `getwc()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `getwc()` function.

**Note:** This function is not available when either LOCALETYPE("CLD") or SYSICLOPT("NOIFSIO) is specified on the compilation command.

**Return Value**

The `getwc()` function returns the next wide character from the input stream, or WEOF. If an error occurs, the `getwc()` function sets the error indicator. If the `getwc()` function encounters the end-of-file, it sets the EOF indicator. If an encoding error occurs during conversion of the multibyte character, the `getwc()` function sets errno to EILSEQ.

Use the `ferror()` or `feof()` functions to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

For information about errno values for `getwc()`, see "fgetwc() — Read Wide Character from Stream" on page 94.

**Example that uses getwc()**
Related Information

- “fgetwc() — Read Wide Character from Stream” on page 94
- “getwchar() — Get Wide Character from stdin” on page 17
- “getc() – getchar() — Read a Character” on page 142
- “putwc() — Write Wide Character” on page 225
- “ungetwc() — Push Wide Character onto Input Stream” on page 401
- “<stdio.h>” on page 14
- “<wchar.h>” on page 17

getwchar() — Get Wide Character from stdin

Format

```c
#include <wchar.h>
wint_t getwchar(void);
```

Language Level: ANSI

Threadsafe: Yes.
Description

The `getwchar()` function reads the next multibyte character from `stdin`, converts it to a wide character, and advances the associated file position indicator for `stdin`. A call to the `getwchar()` function is equivalent to a call to `getwc(stdin)`.

The behavior of the `getwchar()` function is affected by the LC_CTYPE category of the current locale. If you change the category between subsequent read operations on the same stream, undefined results can occur. Using non-wide-character functions with the `getwchar()` function on `stdin` results in undefined behavior.

Note: This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The `getwchar()` function returns the next wide character from `stdin` or WEOF. If the `getwchar()` function encounters EOF, it sets the EOF indicator for the stream and returns WEOF. If a read error occurs, the error indicator for the stream is set, and the `getwchar()` function returns WEOF. If an encoding error occurs during the conversion of the multibyte character to a wide character, the `getwchar()` function sets errno to EILSEQ and returns WEOF.

Use the `ferror()` or `feof()` functions to determine whether an error or an EOF condition occurred. EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does not turn on the EOF indicator.

For information about errno values for `getwchar()`, see "fgetwc() — Read Wide Character from Stream" on page 94.

Example that uses `getwchar()`

This example uses the `getwchar()` to read wide characters from the keyboard, then prints the wide characters.
gmtime() — Convert Time

Format
#include <time.h>
struct tm *gmtime(const time_t *time);

Language Level: ANSI

Threadsafe: No. Use gmtime_r() instead.

Description
The gmtime() function breaks down the time value, in seconds, and stores it in a tm structure, defined in <time.h>. The value time is usually obtained by a call to the time() function.

The fields of the tm structure include:

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wint_t wc;
    errno = 0;
    while (WEOF != (wc = getwchar()))
        printf("wc = \%lc\n", wc);
    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    return 0;

    // gmtime() function
    struct tm *tm;
    tm = gmtime(time);
    // tm fields include
    // - tm_year
    // - tm_mon
    // - tm_mday
    // - tm_hour
    // - tm_min
    // - tm_sec
    // - tm_wday
    // - tm_yday
    // - tm_isdst
    // tm_zone
    // tm_gmtoff

    // Print the time information
    printf("gmtime() returns an tm structure.
    tm time:
    tm_year = %d
    tm_mon = %d
    tm_mday = %d
    tm_hour = %d
    tm_min = %d
    tm_sec = %d
    tm_wday = %d
    tm_yday = %d
    tm_isdst = %d
    tm_gmtoff = %d
    tm_zone = %s
    ");
    // Further processing...
}
```
tm_sec
  Seconds (0-61)

tm_min
  Minutes (0-59)

tm_hour
  Hours (0-23)

tm_mday
  Day of month (1-31)

tm_mon
  Month (0-11; January = 0)

tm_year
  Year (current year minus 1900)

tm_wday
  Day of week (0-6; Sunday = 0)

tm_yday
  Day of year (0-365; January 1 = 0)

tm_isdst
  Zero if Daylight Saving Time is not in effect; positive if Daylight Saving Time is in effect; negative if the information is not available.

Return Value

The gmtime() function returns a pointer to the resulting tm structure.

Notes:

1. The range (0-61) for tm_sec allows for as many as two leap seconds.
2. The gmtime() and localtime() functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Note: This function is locale sensitive.

Example that uses gmtime()

This example uses the gmtime() function to adjust a time_t representation to a Coordinated Universal Time character string, and then converts it to a printable string using the asctime() function.
Related Information

- "asctime() — Convert Time to Character String" on page 41
- "asctime_r() — Convert Time to Character String (Restartable)" on page 43
- "ctime() — Convert Time to Character String" on page 68
- "ctime_r() — Convert Time to Character String (Restartable)" on page 69
- "gmtime_r() — Convert Time (Restartable)
- "localtime() — Convert Time (Restartable)"
- "localtime_r() — Convert Time (Restartable)" on page 174
- "mktime() — Convert Local Time" on page 204
- "setlocale() — Set Locale" on page 322
- "time() — Determine Current Time" on page 391
- "<time.h>" on page 16

**gmtime_r() — Convert Time (Restartable)**

**Format**

```c
#include <time.h>
struct tm *gmtime_r(const time_t *time, struct tm *result);
```

**Language Level:** XPG4

**Threadsafe:** Yes,

**Description**

This function is the restartable version of gmtime().

The gmtime_r() function breaks down the time value, in seconds, and stores it in result. result is a pointer to the tm structure, defined in <time.h>. The value time is usually obtained by a call to the time() function.

The fields of the tm structure include:

- **tm_sec**
  Seconds (0-61)

- **tm_min**
  Minutes (0-59)
tm_hour
   Hours (0-23)

tm_mday
   Day of month (1-31)

tm_mon
   Month (0-11; January = 0)

tm_year
   Year (current year minus 1900)

tm_wday
   Day of week (0-6; Sunday = 0)

tm_yday
   Day of year (0-365; January 1 = 0)

tm_isdst
   Zero if Daylight Saving Time is not in effect; positive if Daylight Saving
   Time is in effect; negative if the information is not available.

Return Value

The gmtime_r() function returns a pointer to the resulting tm structure.

Notes:
1. The range (0-61) for tm_sec allows for as many as two leap seconds.
2. The gmtime() and localtime() functions may use a common, statically
   allocated buffer for the conversion. Each call to one of these functions may alter
   the result of the previous call. The asctime_r(), ctime_r(), gmtime_r(), and
   localtime_r() functions do not use a common, statically-allocated buffer to
   hold the return string. These functions can be used in the place of the
   asctime(), ctime(), gmtime(), and localtime() functions if reentrancy is
   desired.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which
   is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Example that uses gmtime_r()

This example uses the gmtime_r() function to adjust a time_t representation to a
Coordinated Universal Time character string, and then converts it to a printable
string using the asctime_r() function.
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t ltime;
    struct tm mytime;
    char buf[50];
    
    time(&ltime)
    printf("Coordinated Universal Time is %s\n",
        asctime_r(gmtime_r(&ltime, &mytime), buf));
}

/**************************** Output should be similar to: **********
Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/

Related Information
• "asctime() — Convert Time to Character String" on page 41
• "asctime_r() — Convert Time to Character String (Restartable)" on page 43
• "ctime() — Convert Time to Character String" on page 68
• "ctime_r() — Convert Time to Character String (Restartable)" on page 69
• "gmtime() — Convert Time" on page 151
• "localtime() — Convert Time" on page 173
• "localtime_r() — Convert Time (Restartable)" on page 174
• "mktime() — Convert Local Time" on page 204
• "time() — Determine Current Time" on page 391
• "<time.h>" on page 16

hypot() — Calculate Hypotenuse

Format
#include <math.h>
double hypot(double side1, double side2);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The hypot() function calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides side1 and side2. A call to the hypot() function is equivalent to:
   sqrt(side1 * side1 + side2 * side2);

Return Value

The hypot() function returns the length of the hypotenuse. If an overflow results, hypot() sets errno to ERANGE and returns the value HUGE_VAL. If an underflow results, hypot() sets errno to ERANGE and returns zero. The value of errno may also be set to EDOM.

Example that uses hypot()
This example calculates the hypotenuse of a right-angled triangle with sides of 3.0 and 4.0.

```c
#include <math.h>

int main(void)
{
    double x, y, z;
    x = 3.0;
    y = 4.0;
    z = hypot(x,y);

    printf("The hypotenuse of the triangle with sides %lf and %lf
" " is %lf\n", x, y, z);
}

/********************** Output should be similar to: **************/
The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
*/

Related Information
- "sqrt() — Calculate Square Root" on page 337
- "<math.h>” on page 7

isalnum() - isxdigit() — Test Integer Value

Format
#include <ctype.h>
int isalnum(int c);
/* Test for upper- or lowercase letters, or decimal digit */
int isalpha(int c);
/* Test for alphabetic character */
int iscntrl(int c);
/* Test for any control character */
isdigit(int c);
/* Test for decimal digit */
isgraph(int c);
/* Test for printable character excluding space */
islower(int c);
/* Test for lowercase */
isprint(int c);
/* Test for printable character including space */
ispunct(int c);
/* Test for any nonalphanumeric printable character */
/* excluding space */
isspace(int c);
/* Test for whitespace character */
isupper(int c);
/* Test for uppercase */
isxdigit(int c);
/* Test for hexadecimal digit */

Language Level: ANSI

Threadsafe: Yes.

Description
The `<ctype.h>` functions listed test a character with an integer value. The LC_CTYPE category of the active locale determines the results returned from these functions.

**Return Value**

These functions return a nonzero value if the integer satisfies the test condition, or a zero value if it does not. The integer variable `c` must be representable as an unsigned char.

**Note:** EOF is a valid input value.

**Example that uses `<ctype.h>` functions**

This example analyzes all characters between code 0x0 and code `UPPER_LIMIT`, printing A for alphabetic characters, AN for alphanumerics, U for uppercase, L for lowercase, D for digits, X for hexadecimal digits, S for spaces, PU for punctuation, PR for printable characters, G for graphics characters, and C for control characters. This example prints the code if printable.

The output of this example is a 256-line table showing the characters from 0 to 255 that possess the attributes tested.

```c
#include <stdio.h>
#include <ctype.h>
#define UPPER_LIMIT   0xFF

int main(void)
{   int ch;
    for ( ch = 0; ch <= UPPER_LIMIT; ++ch )
    {   printf("%3d ", ch);
        printf("%04x ", ch);
        if ( isalnum(ch) ) printf("AN");
        else printf(" ");
        if ( isalpha(ch) ) printf("A");
        else printf(" ");
        if ( iscntrl(ch) ) printf("C");
        else printf(" ");
        if ( isdigit(ch) ) printf("D");
        else printf(" ");
        if ( isgraph(ch) ) printf("G");
        else printf(" ");
        if ( islower(ch) ) printf("L");
        else printf(" ");
        if ( isprint(ch) ) printf(" ");
        else printf("%c", ch);
        if ( ispunct(ch) ) printf("PU");
        else printf(" ");
        if ( isspace(ch) ) printf("S");
        else printf(" ");
        if ( isprint(ch) ) printf("PR");
        else printf(" ");
        if ( isupper(ch) ) printf("U");
        else printf(" ");
        if ( isxdigit(ch) ) printf("X");
        else printf(" ");
        putchar('
');
    }
}
```

**Related Information**

- "tolower() – toupper() — Convert Character Case” on page 395
- "isblank() — Test for Blank or Tab Character” on page 159
- "<ctype.h>” on page 3
isascii() — Test for Character Representable as ASCII Value

Format

```
#include <ctype.h>
int isascii(int c);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `isascii()` function tests if a given character, in the current locale, can be represented as a valid 7-bit US-ASCII character.

Return Value

The `isascii()` function returns nonzero if `c`, in the current locale, can be represented as a character in the 7-bit US-ASCII character set. Otherwise, it returns 0.

Example that uses `isascii()`

This example tests the integers from 0x7c to 0x82, and prints the corresponding character if the integer can be represented as a character in the 7-bit US-ASCII character set.

```
#include <stdio.h>
#include <ctype.h>

int main(void) {
  int ch;
  for (ch = 0x7c; ch <= 0x82; ch++) {
    printf("%#04x    ", ch);
    if (isascii(ch))
      printf("The character is %c\n", ch);
    else
      printf("Cannot be represented by an ASCII character\n");
  }
  return 0;
}
```

The output should be:

```
0x7c    The character is @
0x7d    The character is '
0x7e    The character is =
0x7f    The character is "
0x80    Cannot be represented by an ASCII character
0x81    The character is a
0x82    The character is b
```

Related Information

- "isalnum() - isxdigit() — Test Integer Value” on page 156
- "iswalnum() to iswxdigit() — Test Wide Integer Value” on page 160
isblank() — Test for Blank or Tab Character

Format
#include <ctype.h>
int isblank(int c);

Note: The isblank() function is supported only for C++, not for C.

Language Level: Extended

Threadsafe: Yes.

Description
The isblank() function tests if a character is either the EBCDIC space or EBCDIC tab character.

Return Value
The isblank() function returns nonzero if c is either the EBCDIC space character or the EBCDIC tab character, otherwise it returns 0.

Example that uses isblank()

This example tests several characters using isblank().
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char *buf = "a b\tc";
    int i;

    for (i = 0; i < 5; i++) {
        if (isblank(buf[i]))
            printf("Character %d is not a blank.\n", i);
        else
            printf("Character %d is a blank\n", i);
    }
    return 0;
}

/*****************************/
The output should be
Character 0 is not a blank.
Character 1 is a blank.
Character 2 is not a blank.
Character 3 is a blank.
Character 4 is not a blank.
/*****************************/
iswalnum() to iswxdigit() — Test Wide Integer Value

Format

```c
#include <wctype.h>

int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The functions listed above, which are all declared in `<wctype.h>`, test a given wide integer value.

The value of `wc` must be a wide-character code corresponding to a valid character in the current locale, or must equal the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

If the program is compiled with LOCALETYPE("LOCALEUCS2) or LOCALETYPE("LOCALEUTF), the wide character properties are those defined by the LC_UNI_CTYPE category of the current locale.

Here are descriptions of each function in this group.

iswalnum()  
Test for a wide alphanumeric character.

iswalpha()  
Test for a wide alphabetic character, as defined in the alpha class of the LC_CTYPE category of the current locale.

iswcntrl()  
Test for a wide control character, as defined in the cntrl class of the LC_CTYPE category of the current locale.

iswdigit()  
Test for a wide decimal-digit character: 0 through 9, as defined in the digit class of the LC_CTYPE category of the current locale.
iswgraph()
Test for a wide printing character, not a space, as defined in the graph class of the LC_CTYPE category of the current locale.

iswlower()
Test for a wide lowercase character, as defined in the lower class of the LC_CTYPE category of the current locale or for which none of the iswcntrl(), iswdigit(), iswspace() function are true.

iswprint()
Test for any wide printing character, as defined in the print class of the LC_CTYPE category of the current locale.

iswpunct()
Test for a wide non-alphanumeric, non-space character, as defined in the punct class of the LC_CTYPE category of the current locale.

iswspace()
Test for a wide whitespace character, as defined in the space class of the LC_CTYPE category of the current locale.

iswupper()
Test for a wide uppercase character, as defined in the upper class of the LC_CTYPE category of the current locale.

iswxdigit()
Test for a wide hexadecimal digit 0 through 9, a through f, or A through F, as defined in the xdigit class of the LC_CTYPE category of the current locale.

The LC_CTYPE or LC_UNI_CTYPE category of the active locale determines the results returned from these functions.

**Note:** The `<wctype.h>` functions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Returned Value**

These functions return a nonzero value if the wide integer satisfies the test value, or a 0 value if it does not. The value for `wc` must be representable as a wide unsigned char. WEOF is a valid input value.

**Example**
iswctype() — Test for Character Property

Format
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);

Language Level: ANSI

Threadsafe: Yes.

Description
The iswctype() function determines whether the wide character wc has the
property wc_prop. If the value of wc is neither WEOF nor any value of the wide
characters that corresponds to a multibyte character, the behavior is undefined. If
the value of wc_prop is incorrect (that is, it is not obtained by a previous call to the
wctype() function, or wc_prop has been invalidated by a subsequent call to the
setlocale() function), the behavior is undefined.

If the program is compiled with LOCALETYPE(*LOCALEUCS2) or
LOCALETYPE(*LOCALEUTF), the wide character properties are those defined by
the LC_UNI_CTYPE category of the current locale.

Note: This function is not available when LOCALETYPE(*CLD) is specified on the
compilation command.

Return Value
The iswctype() function returns true if the value of the wide character wc has the
property wc_prop.
The following strings, alnum through to xdigit are reserved for the standard character classes. The functions are shown as follows with their equivalent isw*() function:

```c
iswctype(wc, wctype("alnum"));  /* is equivalent to */          iswalnum(wc);
iswctype(wc, wctype("alpha"));  /* is equivalent to */          iswalpha(wc);
iswctype(wc, wctype("cntrl"));  /* is equivalent to */          iswcntrl(wc);
iswctype(wc, wctype("digit"));  /* is equivalent to */          iswdigit(wc);
iswctype(wc, wctype("graph"));  /* is equivalent to */          iswgraph(wc);
iswctype(wc, wctype("lower"));  /* is equivalent to */          iswlower(wc);
iswctype(wc, wctype("print"));  /* is equivalent to */          iswprint(wc);
iswctype(wc, wctype("punct"));  /* is equivalent to */          iswpunct(wc);
iswctype(wc, wctype("space"));  /* is equivalent to */          iswspace(wc);
iswctype(wc, wctype("upper"));  /* is equivalent to */          iswupper(wc);
iswctype(wc, wctype("xdigit"));  /* is equivalent to */         iswxdigit(wc);
```

**Example that uses iswctype()**

```c
#include <stdio.h>
#include <wctype.h>

int main(void)
{
    int wc;

    for (wc=0; wc <= 0xFF; wc++) {
        printf("%3d", wc);
        printf(" %#4x ", wc);
        printf("%3s", iswctype(wc, wctype("alnum"))  ? "AN" : " ");
        printf("%2s", iswctype(wc, wctype("alpha"))  ? "A"  : " ");
        printf("%2s", iswctype(wc, wctype("cntrl"))  ? "C"   : " ");
        printf("%2s", iswctype(wc, wctype("digit"))  ? "D"   : " ");
        printf("%2s", iswctype(wc, wctype("graph"))  ? "G"   : " ");
        printf("%2s", iswctype(wc, wctype("lower"))  ? "L"   : " ");
        printf("%C", iswctype(wc, wctype("print"))  ? wc   : ' ');
        printf("%3s", iswctype(wc, wctype("punct"))  ? "PU" : " ");
        printf("%3s", iswctype(wc, wctype("space"))  ? "S"  : " ");
        printf("%3s", iswctype(wc, wctype("print"))  ? "PR" : " ");
        printf("%2s", iswctype(wc, wctype("upper"))  ? "U"  : " ");
        printf("%2s", iswctype(wc, wctype("xdigit")) ? "X"  : " ");
        putchar(\'\n\');
    }
}
```

**Related Information**

- "wctype() — Get Handle for Character Property Classification” on page 472
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 160
- “<wctype.h>” on page 17

---

**_itoa - Convert Integer to String**

**Format**

```c
#include <stdlib.h>
char *_itoa(int value, char *string, int radix);
```

**Note:** The _itoa function is supported only for C++, not for C.

**Language Level:** Extension

**Threadsafe:** Yes.
Description

_itoa() converts the digits of the given value to a character string that ends with a null character and stores the result in string. The radix argument specifies the base of value; it must be in the range 2 to 36. If radix equals 10 and value is negative, the first character of the stored string is the minus sign (-).

Note: The space reserved for string must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

Return Value

_itoa returns a pointer to string. There is no error return value.

When the string argument is NULL or the radix is outside the range 2 to 36, errno will be set to EINVAL.

Example that uses _itoa()

This example converts the integer value -255 to a decimal, a binary, and a hex number, storing its character representation in the array buffer.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _itoa(-255, buffer, 10);
    printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
    p = _itoa(-255, buffer, 2);
    printf("The result of _itoa(-255) with radix of 2 is %s\n", p);
    p = _itoa(-255, buffer, 16);
    printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _itoa(-255) with radix of 10 is -255
The result of _itoa(-255) with radix of 2 is 11111111111111111111
The result of _itoa(-255) with radix of 16 is ffffff01
```

Related Information:

- "_gcvt - Convert Floating-Point to String" on page 141
- "_itoa - Convert Integer to String" on page 163
- "_ltoa - Convert Long Integer to String" on page 177
- "_ultoa - Convert Unsigned Long Integer to String" on page 398
- "<stdlib.h>" on page 15

labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer

Format (labs())
#include <stdlib.h>
long int labs(long int n);

**Format** (llabs())
#include <stdlib.h>
long long int llabs(long long int i);

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `labs()` function produces the absolute value of its long integer argument `n`. The result may be undefined when the argument is equal to `LONG_MIN`, the smallest available long integer. The value `LONG_MIN` is defined in the `<limits.h>` include file.

The `llabs()` function returns the absolute value of its long long integer operand. The result may be undefined when the argument is equal to `LONG_LONG_MIN`, the smallest available long integer. The value `LONG_LONG_MIN` is defined in the `<limits.h>` include file.

**Return Value**

The `labs()` function returns the absolute value of `n`. There is no error return value.

The `llabs()` function returns the absolute value of `i`. There is no error return value.

**Example that uses labs()**

This example computes `y` as the absolute value of the long integer `-41567`.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    long x, y;
    x = -41567L;
    y = labs(x);
    printf("The absolute value of %ld is %ld\n", x, y);
}
```

`/********************  Output should be similar to:  **************`

The absolute value of -41567 is 41567

`*/`

**Related Information**

- "abs() — Calculate Integer Absolute Value" on page 39
- "fabs() — Calculate Floating-Point Absolute Value" on page 81
- "<limits.h>" on page 7
ldexp() — Multiply by a Power of Two

Format

```c
#include <math.h>
double ldexp(double x, int exp);
```

Language Level: ANSI

Threatsafe: Yes.

Description

The ldexp() function calculates the value of $x \times (2^{exp})$.

Return Value

The ldexp() function returns the value of $x \times (2^{exp})$. If an overflow results, the function returns +HUGE_VAL for a large result or -HUGE_VAL for a small result, and sets errno to ERANGE.

Example that uses ldexp()

This example computes $y$ as 1.5 times 2 to the fifth power ($1.5 \times 2^5$):

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    int p;
    x = 1.5;
    p = 5;
    y = ldexp(x, p);
    printf("%lf times 2 to the power of %d is %lf\n", x, p, y);
}
```

/********************  Output should be similar to:  **************
1.500000 times 2 to the power of 5 is 48.000000
***************/

Related Information

- "exp() — Calculate Exponential Function" on page 81
- "frexp() — Separate Floating-Point Value" on page 122
- "modf() — Separate Floating-Point Value" on page 205
- "<math.h>" on page 7

ldiv() — lldiv() — Perform Long and Long Long Division

Format (ldiv())

```c
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

Format (lldiv())

```c
#include <math.h>
#include <stdio.h>
int main(void)
{
    double x, y;
    int p;
    x = 1.5;
    p = 5;
    y = ldexp(x, p);
    printf("%lf times 2 to the power of %d is %lf\n", x, p, y);
}
```

/********************  Output should be similar to:  **************
1.500000 times 2 to the power of 5 is 48.000000
***************/

Related Information

- "exp() — Calculate Exponential Function" on page 81
- "frexp() — Separate Floating-Point Value" on page 122
- "modf() — Separate Floating-Point Value" on page 205
- "<math.h>" on page 7
#include <stdlib.h>

```
lldiv_t lldiv(long long int numerator, long long int denominator);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, only the function version is threadsafe. The macro version is NOT threadsafe.

**Description**

The `ldiv()` function calculates the quotient and remainder of the division of `numerator` by `denominator`.

**Return Value**

The `ldiv()` function returns a structure of type `ldiv_t`, containing both the quotient (`long int quot`) and the remainder (`long int rem`). If the value cannot be represented, the return value is undefined. If `denominator` is 0, an exception is raised.

The `lldiv()` subroutine computes the quotient and remainder of the `numerator` parameter by the `denominator` parameter.

The `lldiv()` subroutine returns a structure of type `lldiv_t`, containing both the quotient and the remainder. The structure is defined as:

```c
struct lldiv_t {
    long long int quot; /* quotient */
    long long int rem; /* remainder */
};
```

If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and magnitude of the resulting quotient is the largest long long integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the `denominator` is 0), the behavior is undefined.

**Example that uses ldiv()**

This example uses `ldiv()` to calculate the quotients and remainders for a set of two dividends and two divisors.
Related Information

- “div() — Calculate Quotient and Remainder” on page 77
- “<stdlib.h>” on page 15

localeconv() — Retrieve Information from the Environment

Format

#include <locale.h>
struct lconv *localeconv(void);

Language Level: ANSI

Threading: Yes.

Description

The `localeconv()` sets the components of a structure having type `struct lconv` to values appropriate for the current locale. The structure may be overwritten by another call to `localeconv()`, or by calling the `setlocale()` function and passing `LC_ALL`, `LC_COLLATE`, `LC_MONETARY`, or `LC_NUMERIC`.

The structure contains the following elements (defaults shown are for the C locale):

<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose of Element</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *decimal_point</td>
<td>Decimal-point character used to format non-monetary quantities.</td>
<td>“.”</td>
</tr>
<tr>
<td>char *thousands_sep</td>
<td>Character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.</td>
<td>“…”</td>
</tr>
<tr>
<td>Element</td>
<td>Purpose of Element</td>
<td>Default</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>char *grouping</td>
<td>String indicating the size of each group of digits in formatted non-monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The characters following this define succeeding groups to the left of the previous group. If the last character is not UCHAR_MAX, the grouping is repeated using the last character as the size. If the last character is UCHAR_MAX, grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 170 for an example of how grouping works.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *int_curr_symbol</td>
<td>International currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) is the character used to separate the international currency symbol from the monetary quantity.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *currency_symbol</td>
<td>Local currency symbol of the current locale.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_decimal_point</td>
<td>Decimal-point character used to format monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_thousands_sep</td>
<td>Separator for digits in formatted monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_grouping</td>
<td>String indicating the size of each group of digits in formatted monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The following characters define succeeding groups to the left of the previous group. If the last character is not UCHAR_MAX, the grouping is repeated using the last character as the size. If the last character is UCHAR_MAX, grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 170 for an example of how grouping works.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *positive_sign</td>
<td>String indicating the positive sign used in monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *negative_sign</td>
<td>String indicating the negative sign used in monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Element</td>
<td>Purpose of Element</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>char int_frac_digits</td>
<td>The number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char frac_digits</td>
<td>Number of digits to the right of the decimal place in monetary quantities.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_cs_precedes</td>
<td>1 if the currency_symbol precedes the value for a nonnegative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_sep_by_space</td>
<td>1 if the currency_symbol is separated by a space from the value of a nonnegative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_cs_precedes</td>
<td>1 if the currency_symbol precedes the value for a negative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_sep_by_space</td>
<td>1 if the currency_symbol is separated by a space from the value of a negative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_sign_posn</td>
<td>Value indicating the position of the positive_sign for a nonnegative formatted monetary quantity.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_sign_posn</td>
<td>Value indicating the position of the negative_sign for a negative formatted monetary quantity.</td>
<td>UCHAR_MAX</td>
</tr>
</tbody>
</table>

Pointers to strings with a value of "" indicate that the value is not available in the C locale or is of zero length. Elements with char types with a value of UCHAR_MAX indicate that the value is not available in the current locale.

The n_sign_posn and p_sign_posn elements can have the following values:

**Value**  **Meaning**
0    The quantity and currency_symbol are enclosed in parentheses.
1    The sign precedes the quantity and currency_symbol.
2    The sign follows the quantity and currency_symbol.
3    The sign precedes the currency_symbol.
4    The sign follows the currency_symbol.

**Grouping Example**

*Table 1. Grouping Example*

<table>
<thead>
<tr>
<th>Locale Source</th>
<th>Grouping String</th>
<th>Number</th>
<th>Formatted Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>0x00</td>
<td>123456789</td>
<td>123456789</td>
</tr>
<tr>
<td>3</td>
<td>0x0300</td>
<td>123456789</td>
<td>123,456,789</td>
</tr>
<tr>
<td>3;−1</td>
<td>0x03FF00</td>
<td>123456789</td>
<td>123456,789</td>
</tr>
<tr>
<td>3;2;1</td>
<td>0x03020100</td>
<td>123456789</td>
<td>1,2,3,4,56,789</td>
</tr>
</tbody>
</table>
Monetary Formatting Example:

Table 2. Monetary Formatting Example

<table>
<thead>
<tr>
<th>Country</th>
<th>Positive Format</th>
<th>Negative Format</th>
<th>International Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italy</td>
<td>L.1.230</td>
<td>-L.1.230</td>
<td>ITL1.230</td>
</tr>
<tr>
<td>Netherlands</td>
<td>F 1.234,56</td>
<td>F -1.234,56</td>
<td>NLG 1.234,56</td>
</tr>
<tr>
<td>Norway</td>
<td>kr1.234,56</td>
<td>kr1.234,56-</td>
<td>NOK1.234,56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>SFRs.1,234,56</td>
<td>SFrx.1,234,56C</td>
<td>CHF 1,234,56</td>
</tr>
</tbody>
</table>

The above table was generated by locales with the following monetary fields:

Table 3. Monetary Fields

<table>
<thead>
<tr>
<th></th>
<th>Italy</th>
<th>Netherlands</th>
<th>Norway</th>
<th>Switzerland</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_curr_symbol</td>
<td>&quot;ITL.&quot;</td>
<td>&quot;NLG&quot;</td>
<td>&quot;NOK&quot;</td>
<td>&quot;CHF&quot;</td>
</tr>
<tr>
<td>currency_symbol</td>
<td>&quot;L.&quot;</td>
<td>&quot;F&quot;</td>
<td>&quot;kr&quot;</td>
<td>&quot;SFrs.&quot;</td>
</tr>
<tr>
<td>mon_decimal_point</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>mon_thousands_sep</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
</tr>
<tr>
<td>mon_grouping</td>
<td>&quot;&quot;\3&quot;</td>
<td>&quot;&quot;\3&quot;</td>
<td>&quot;&quot;\3&quot;</td>
<td>&quot;&quot;\3&quot;</td>
</tr>
<tr>
<td>positive_sign</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>negative_sign</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>int_frac_digits</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>frac_digits</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p_sep_by_space</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_sep_by_space</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p_sep_posn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_sign_posn</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Return Value

The `localeconv()` function returns a pointer to the structure.

Example that uses *CLD locale objects

This example prints out the default decimal point for your locale and then the decimal point for the LC_C_FRANCE locale.
Example that uses *LOCALE objects

/*@*************************************************************************/
This example prints out the default decimal point for
the C locale and then the decimal point for the French
locale using a *LOCALE object called
"QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE".

Step 1: Create a French *LOCALE object by entering the command
CRTLOCALE LOCALE('QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE') +
SRCFILE('QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +
FR_FR.MBR') CCSID(297)                            *
Step 2: Compile the following C source, specifying
 LOCALETYPE(*LOCALE) on the compilation command.
Step 3: Run the program.
*************************************************************************/

#include <stdio.h>
#include <locale.h>
int main(void) {
    char * string;
    struct lconv * mylocale;
    mylocale = localeconv();

    /* Display default decimal point */
    printf("Default decimal point is a %s\n", mylocale->decimal_point);
    if (NULL != (string = setlocale(LC_ALL, "QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE"))) {
        mylocale = localeconv();

        /* A comma is set to be the decimal point in the French locale */
        printf("France's decimal point is a %s\n", mylocale->decimal_point);
    } else {
        printf("setlocale(LC_ALL, "QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE") returned <NULL>\n\n");
    }
    return 0;
}
localtime() — Convert Time

Format
#include <time.h>
struct tm *localtime(const time_t *timeval);

Language Level: ANSI

Threadsafe: No. Use localtime_r() instead.

Description

The localtime() function converts a time value, in seconds, to a structure of type tm.

Note: This function is locale sensitive.

The localtime() function takes a timeval assumed to be Universal Coordinate Time (UTC) and converts it to job locale time. For this conversion localtime() checks the current locale setting for local time zone and Daylight Saving Time (DST). If these values are not set in the current locale, localtime() gets the local time zone and Daylight Saving Time (DST) settings from the current job. Once converted, the time is returned in a structure of type tm. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored.

The time value is usually obtained by a call to the time() function.

Notes:

1. The gmtime() and localtime() functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may destroy the result of the previous call. The ctime_r(), gmtime_r(), and localtime_r() functions do not use a common, statically-allocated buffer. These functions can be used in the place of the asctime(), ctime(), gmtime() and localtime() functions if reentrancy is desired.

2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Return Value

The localtime() function returns a pointer to the structure result. There is no error return value.

Example that uses localtime()

This example queries the system clock and displays the local time.
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

    ltime = time(&ltime);
    newtime = localtime(&ltime);
    printf("The date and time is %s", asctime(newtime));
}

/******************* If the local time is 3:00 p.m. May 31, 1993, **********
************************* output should be: ********************
The date and time is Mon May 31 15:00:00 1993
*/

Related Information
- "asctime() — Convert Time to Character String" on page 41
- "asctime_r() — Convert Time to Character String (Restartable)" on page 43
- "ctime() — Convert Time to Character String" on page 68
- "ctime_r() — Convert Time to Character String (Restartable)" on page 69
- "gmtime() — Convert Time" on page 151
- "gmtime_r() — Convert Time (Restartable)" on page 153
- "mktime() — Convert Local Time" on page 204
- "localtime_r() — Convert Time (Restartable)"
- "setlocale() — Set Locale" on page 322
- "time() — Determine Current Time" on page 391
- "<time.h>" on page 16

localtime_r() — Convert Time (Restartable)

Format
#include <time.h>
struct tm *localtime_r(const time_t *timeval, struct tm *result);

Language Level: XPG4

Threading: Yes,

Description
This function is the restartable version of localtime(). It is the same as
localtime() except that it passes in the place to store the returned structure result.

Return Value
The localtime_r() returns a pointer to the structure result. There is no error return
value.

Example that uses localtime_r()
This example queries the system clock and displays the local time.
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm newtime;
    time_t ltime;
    char buf[50];

    ltime=time(&ltime);
    localtime_r(&ltime, &newtime);
    printf("The date and time is %s", asctime_r(&newtime, buf));
}

/blob comment
The date and time is Mon May 31 15:00:00 1993
/blob comment

Related Information

- "asctime() — Convert Time to Character String” on page 41
- "asctime_r() — Convert Time to Character String (Restartable)” on page 43
- "ctime() — Convert Time to Character String” on page 68
- "ctime_r() — Convert Time to Character String (Restartable)” on page 69
- "gmtime() — Convert Time” on page 151
- "gmtime_r() — Convert Time (Restartable)” on page 153
- "localtime() — Convert Time” on page 173
- "mktime() — Convert Local Time” on page 204
- "time() — Determine Current Time” on page 391
- "<time.h>” on page 16

log() — Calculate Natural Logarithm

Format

#include <math.h>
double log(double x);

Language Level: ANSI

Threadsafe: Yes.

Description

The log() function calculates the natural logarithm (base e) of x.

Return Value

The log() function returns the computed value. If x is negative, log() sets errno to EDOM and may return the value -HUGE_VAL. If x is zero, log() returns the value -HUGE_VAL, and may set errno to ERANGE.

Example that uses log()

This example calculates the natural logarithm of 1000.0.
Related Information

• “exp() — Calculate Exponential Function” on page 81
• “log10() — Calculate Base 10 Logarithm”
• “pow() — Compute Power” on page 210
• “<math.h>” on page 7

log10() — Calculate Base 10 Logarithm

Format

#include <math.h>
double log10(double x);

Language Level: ANSI

Thesafet: Yes.

Description

The log10() function calculates the base 10 logarithm of x.

Return Value

The log10() function returns the computed value. If x is negative, log10() sets errno to EDOM and may return the value -HUGE_VAL. If x is zero, the log10() function returns the value -HUGE_VAL, and may set errno to ERANGE.

Example that uses log10()

This example calculates the base 10 logarithm of 1000.0.
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x = 1000.0, y;
    y = log10(x);
    printf("The base 10 logarithm of %lf is %lf
", x, y);
}

/********************** Output should be similar to: **************/
The base 10 logarithm of 1000.000000 is 3.000000

Related Information
- “exp() — Calculate Exponential Function” on page 81
- “log() — Calculate Natural Logarithm” on page 175
- “pow() — Compute Power” on page 210
- “<math.h>” on page 7

_ltoa - Convert Long Integer to String

Format
#include <stdlib.h>
char *_ltoa(long value, char *string, int radix);

Note: The _ltoa function is supported only for C++, not for C.

Language Level: Extension

Threading: Yes.

Description
_ltoa converts the digits of the given long integer value to a character string that ends with a null character and stores the result in string. The radix argument specifies the base of value; it must be in the range 2 to 36. If radix equals 10 and value is negative, the first character of the stored string is the minus sign (-).

Note: The space allocated for string must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

Return Value

_ltoa returns a pointer to string. There is no error return value.

When the string argument is NULL or the radix is outside the range 2 to 36, errno will be set to EINVAL.

Example that uses _ltoa()

This example converts the integer value -255L to a decimal, a binary, and a hex value, and stores its character representation in the array buffer.
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ltoa(-255L, buffer, 10);
    printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
    p = _ltoa(-255L, buffer, 2);
    printf("The result of _ltoa(-255) with radix of 2\n is %s\n", p);
    p = _ltoa(-255L, buffer, 16);
    printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
    return 0;
}

The output should be:
The result of _ltoa(-255) with radix of 10 is -255
The result of _ltoa(-255) with radix of 2
is 11111111111111111111111000000001
The result of _ltoa(-255) with radix of 16 is ffffffff1

Related Information:
- "atol() — atoll() — Convert Character String to Long or Long Long Integer" on page 51
- "_gcvt - Convert Floating-Point to String" on page 141
- "_itoa - Convert Integer to String" on page 163
- "strtol() — strtoll() — Convert Character String to Long and Long Long Integer" on page 380
- "_ultoa - Convert Unsigned Long Integer to String" on page 398
- "wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer" on page 458
- "<stdlib.h>" on page 15

**longjmp() — Restore Stack Environment**

**Format**
#include <setjmp.h>
void longjmp(jmp_buf env, int value);

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `longjmp()` function restores a stack environment previously saved in `env` by the `setjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function causes the current stack environment to be saved in `env`. A subsequent call to `longjmp()` restores the saved environment and returns control to a point in the program corresponding to the `setjmp()` call. Processing resumes as if the `setjmp()` call had just returned the given `value`.

All variables (except register variables) that are available to the function that receives control contain the values they had when `longjmp()` was called. The
values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to the `setjmp()` and `longjmp()` functions are also unpredictable.

**Note:** Ensure that the function that calls the `setjmp()` function does not return before you call the corresponding `longjmp()` function. Calling `longjmp()` after the function calling the `setjmp()` function returns causes unpredictable program behavior.

The *value* argument must be nonzero. If you give a zero argument for *value*, `longjmp()` substitutes 1 in its place.

**Return Value**

The `longjmp()` function does not use the normal function call and return mechanisms; it has no return value.

**Example that uses `longjmp()`**

This example saves the stack environment at the statement:

```c
if(setjmp(mark) != 0) ... 
```

When the system first performs the if statement, it saves the environment in mark and sets the condition to FALSE because the `setjmp()` function returns a 0 when it saves the environment. The program prints the message:

```
setjmp has been called
```

The subsequent call to function `p()` tests for a local error condition, which can cause it to call the `longjmp()` function. Then, control returns to the original `setjmp()` function using the environment saved in mark. This time, the condition is TRUE because -1 is the return value from the `longjmp()` function. The example then performs the statements in the block, prints `longjmp()` has been called, calls the `recover()` function, and leaves the program.
`malloc()` — Reserve Storage Block

**Format**

```c
#include <stdlib.h>

void *malloc(size_t size);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `malloc()` function reserves a block of storage of `size` bytes. Unlike the `calloc()` function, `malloc()` does not initialize all elements to 0. The maximum size for a non-Teraspace `malloc()` is 16711568 bytes.

---

```c
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
    if (setjmp(mark) != 0)
    {
        printf("longjmp has been called\n");
        recover();
        exit(1);
    }
    printf("setjmp has been called\n");
    printf("Calling function p()\n");
    p();
    printf("This point should never be reached\n");
}

void p(void)
{
    printf("Calling longjmp() from inside function p()\n");
    longjmp(mark, -1);
    printf("This point should never be reached\n");
}

void recover(void)
{
    printf("Performing function recover()\n");
}

/********************Output should be as follows: ********************/
setjmp has been called
Calling function p()
Calling longjmp() from inside function p()
longjmp has been called
Performing function recover()
/****************************END OF PROGRAM***********************/
```
Notes:
1. To use Teraspace storage instead of heap storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the CRTCMOD compiler command. This maps the malloc() library function to _C_TS_malloc(), its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to _C_TS_malloc() is 2GB - 224, or 2147483424 bytes. If more than 2147483408 bytes are needed on a single request, call _C_TS_malloc64(unsigned long long int);
   For more information, see the ILE Concepts manual.
2. For current statistics on the teraspace storage being used by MI programs in an activation group, call the _C_TS_malloc_info function. This function returns information including total bytes, allocated bytes and blocks, unallocated bytes and blocks, requested bytes, pad bytes, and overhead bytes. To get more detailed information about the memory structures used by the _C_TS_malloc() and _C_TS_malloc64() functions, call the _C_TS_malloc_debug function. You can use the information this function returns to identify memory corruption problems.

Return Value

The malloc() function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. The return value is NULL if not enough storage is available, or if size was specified as zero.

Example that uses malloc()

This example prompts for the number of array entries you want and then reserves enough space in storage for the entries. If malloc() was successful, the example assigns values to the entries and prints out each entry; otherwise, it prints out an error.
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable */
  int i;            /* index variable */
  int num;          /* number of entries of the array */

  printf( "Enter the size of the array\n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long *) malloc( num * sizeof( long ))) != NULL )
  {
    for ( i = 0; i < num; ++i )          /* put values in array */
      *index++ = i;                        /* using pointer notation */

    for ( i = 0; i < num; ++i )          /* print the array out */
      printf( "array[ %i ] = %i\n", i, array[i] );
  }
  else { /* malloc error */
    perror( "Out of storage" );
    abort();
  }
}

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4

Related Information

- “calloc() — Reserve and Initialize Storage” on page 57
- “free() — Release Storage Blocks” on page 118
- “realloc() — Change Reserved Storage Block Size” on page 247
- “<stdlib.h>” on page 15

mblen() — Determine Length of a Multibyte Character

Format
#include <stdlib.h>
int mblen(const char *string, size_t n);

Language Level: ANSI

Threadsafe: No. Use mbrlen() instead.

Description

The mblen() function determines the length in bytes of the multibyte character pointed to by string. n represents the maximum number of bytes examined.

Return Value
If \textit{string} is NULL, the \texttt{mblen()} function returns:
- Non-zero if the active locale allows mixed-byte strings. The function initializes
  the state variable.
- Zero otherwise.

If \textit{string} is not NULL, \texttt{mblen()} returns:
- Zero if \textit{string} points to the null character.
- The number of bytes comprising the multibyte character.
- -1 if \textit{string} does not point to a valid multibyte character.

\textbf{Note:} The \texttt{mblen()}, \texttt{mbtowc()}, and \texttt{wctomb()} functions use their own statically
allocated storage and are therefore not restartable. However, \texttt{mbrlen()},
\texttt{mbrtowc()}, and \texttt{wcrtomb()} are restartable.

\textbf{Example that uses \texttt{mblen()}}

This example uses \texttt{mblen()} and \texttt{mbtowc()} to convert a multibyte character into a
single wide character.

```c
#include <stdio.h>
#include <stdlib.h>

int length, temp;
char string[6] = "w";
wchar_t arr[6];

int main(void)
{
    /* Initialize internal state variable */
    length = mblen(NULL, MB_CUR_MAX);

    /* Set string to point to a multibyte character */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr,string,length);
    arr[1] = L'\0';
    printf("wide character string: %ls\n", arr);
}
```

\textbf{Related Information}
- \texttt{"mbrlen() — Determine Length of a Multibyte Character (Restartable)"}
- \texttt{"mbtowc() — Convert Multibyte Character to a Wide Character" on page 196}
- \texttt{"mbstowcs() — Convert a Multibyte String to a Wide Character String" on page 192}
- \texttt{"strlen() — Determine String Length" on page 358}
- \texttt{"wcslen() — Calculate Length of Wide-Character String" on page 440}
- \texttt{"wctomb() — Convert Wide Character to Multibyte Character" on page 469}
- \texttt{"<stdlib.h>" on page 15}

\textbf{mbrlen() — Determine Length of a Multibyte Character (Restartable)}

\textbf{Format}
```
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

\textbf{Language Level: ANSI}
**Threadsafe:** Yes, if *ps* is not NULL.

**Description**

This function is the restartable version of *mblen()*.

The *mbrlen()* function determines the length of a multibyte character.

*n* is the number of bytes (at most) of the multibyte string to examine.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, *ps* of type pointer to *mbstate_t* that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a NULL pointer, *mbrlen()* behaves like *mblen()*.

*mbrlen()* is a restartable version of *mblen()* in other words, shift-state information is passed as one of the arguments (*ps* represents the initial shift) and is updated on exit. With *mbrlen()* you can switch from one multibyte string to another, provided that you have kept the shift-state information.

This function is affected by the LC_CTYPE of the current locale.

**Note:** This function is not available when you specify LOCALETYPE("CLD) on the compilation command.

**Return Value**

If *s* is a null pointer and if the active locale allows mixed-byte strings, the *mbrlen()* function returns nonzero. If *s* is a null pointer and if the active locale does not allow mixed-byte strings, zero will be returned.

If *s* is not a null pointer, the *mbrlen()* function returns one of the following:

- **0** If *s* is a NULL string (*s* points to the NULL character).
- **positive** If the next *n* or fewer bytes comprise a valid multibyte character. The value returned is the number of bytes that comprise the multibyte character.
- **(size_t)-1** If *s* does not point to a valid multibyte character.
- **(size_t)-2** If the next *n* or fewer bytes contribute to an incomplete but potentially valid character and all *n* bytes have been processed.

**Example that uses mbrlen()**

```c
/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"
```
int main(void) {
    int length, sl = 0;
    char string[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will find the length of */
    /* of a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    /* For single byte cases the state will always */
    /* remain in the initial state 0 */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");
    length = mbrlen(string, MB_CUR_MAX, &ps);
    /* In this case length is 1, which is always the case for */
    /* single byte CCSID */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");
    length = mbrlen(string, MB_CUR_MAX, &ps);
    /* The first is single byte so length is 1 and */
    /* the state is still the initial state 0 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);
    sl += length;
    length = mbrlen(&string[sl], MB_CUR_MAX, &ps);
    /* The next character is a mixed byte. Length is 3 to */
    /* account for the shiftout 0x0E. State is */
    /* changed to double byte state. */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;
    length = mbrlen(&string[sl], MB_CUR_MAX, &ps);
    /* The next character is also a double byte character. */
    /* The state is changed to initial state since this was */
    /* the last double byte character. Length is 3 to */
    /* account for the ending 0x0F shiftin. */
    printf("length = %d, state = %d\n", length, ps);
}

Chapter 2. Library Functions  185
```c
sl += length;
length = mbrlen(&string[sl], MB_CUR_MAX, &ps);

/* The next character is single byte so length is 1 and */
/* state remains in initial state. */
printf("length = %d, state = %d\n\n", length, ps);
}
/* The output should look like this:
length = 1, state = 0
MB_CUR_MAX: 1
length = 1, state = 0
MB_CUR_MAX: 4
length = 3, state = 2
length = 3, state = 0
length = 1, state = 0
*/
```

**Related Information**

- “mblen() — Determine Length of a Multibyte Character” on page 182
- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 196
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)”
- “mbstowc() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 191
- “setlocale() — Set Locale” on page 322
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 425
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 451
- “<locale.h>” on page 7
- “<wchar.h>” on page 17

---

**mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)**

**Format**

```c
#include <wchar.h>
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

**Language Level:** ANSI

**Threadsafe:** Yes, if ps is not NULL

**Description**

This function is the restartable version of the mbtowc() function.
If \( s \) is a null pointer, the \( 	ext{mbrtowc}() \) function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). In this situation, the value of the \( pwc \) parameter will be ignored and the resulting shift state described will be the initial conversion state.

If \( s \) is not a null pointer, the \( 	ext{mbrtowc}() \) function determines the number of bytes that are in the multibyte character (and any leading shift sequences) pointed to by \( s \), produces the value of the corresponding multibyte character and if \( pwc \) is not a null pointer, stores that value in the object pointed to by \( pwc \). If the corresponding multibyte character is the null wide character, the resulting state will be reset to the initial conversion state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, \( ps \) of type pointer to \text{mbstate_t} that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If \( ps \) is NULL, this function uses an internal static variable for the state.

At most, \( n \) bytes of the multibyte string are examined.

**Note:** This function is not available when you specify LOCALETYPE(CLD) on the compilation command.

**Return Value**

If \( s \) is a null pointer, the \( 	ext{mbrtowc}() \) function returns the number of bytes necessary to enter the initial shift state. The value returned must be less than the MB_CUR_MAX macro.

If a conversion error occurs, \( \text{errno} \) may be set to \text{ECONVERT}.

If \( s \) is not a null pointer, the \( 	ext{mbrtowc}() \) function returns one of the following:

- 0 If the next \( n \) or fewer bytes form the multibyte character that corresponds to the null wide character.
- positive If the next \( n \) or fewer bytes form a valid multibyte character. The value returned is the number of bytes that constitute the multibyte character.
- \( (\text{size}_0)-2 \) If the next \( n \) bytes form an incomplete (but potentially valid) multibyte character, and all \( n \) bytes have been processed. It is unspecified whether this can occur when the value of \( n \) is less than the value of the MB_CUR_MAX macro.
- \( (\text{size}_0)-1 \) If an encoding error occurs (when the next \( n \) or fewer bytes do not form a complete and correct multibyte character). The value of the macro EILSEQ is stored in \( \text{errno} \), but the conversion state is unchanged.

**Note:** When a -2 value is returned, the string could contain redundant shift-out and shift-in characters or a partial UTF-8 character. To continue processing the multibyte string, increment the pointer by the value \( n \), and call \( 	ext{mbrtowc}() \) again.

**Example that uses \text{mbrtowc}()**
/* This program is compiled with LOCALETYPE(*LOCALE) and SYSICOPT(*IFSIO) */

#include  <stdio.h>
#include  <stdlib.h>
#include  <locale.h>
#include  <wchar.h>
#include  <errno.h>

#define LOCNAME     "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    wchar_t buffer[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    /* For single byte cases the state will always */
    /* remain in the initial state 0 */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.
" );
    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);
    /* In this case length is 1, and C1 is converted 0x00C1 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);
    sl += length;
    length = mbrtowc(&buffer[1], &string[sl], MB_CUR_MAX, &ps);
    /* The first is single byte so length is 1 and */
    /* the state is still the initial state 0. C1 is converted*/
    /* to 0x00C1 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.
" );
    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);
    /* The first is single byte so length is 1 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);
    sl += length;
    length = mbrtowc(&buffer[1], &string[sl], MB_CUR_MAX, &ps);
    /* The next character is a mixed byte. Length is 3 to */
    /* account for the shiftout 0x0e. State is */
    /* changed to double byte state. 0x4171 is copied into */
    /* the buffer */
}
printf("length = %d, state = %d\n\n", length, ps);

sl += length;
length = mbrtowc(&buffer[2], &string[sl], MB_CUR_MAX, &ps);

/* The next character is also a double byte character. */
/* The state is changed to initial state since this was */
/* the last double byte character. Length is 3 to */
/* account for the ending 0x0f shiftin. 0x4172 is copied */
/* into the buffer. */
printf("length = %d, state = %d\n\n", length, ps);
sl += length;
length = mbrtowc(&buffer[3], &string[sl], MB_CUR_MAX, &ps);

/* The next character is single byte so length is 1 and */
/* state remains in initial state. 0xC2 is converted to */
/* 0x00C2. The buffer now has the value: */
/* 0x00C14171417200C2 */
printf("length = %d, state = %d\n\n", length, ps);
}

/* The output should look like this:
length = 1, state = 0
MB_CUR_MAX: 1
length = 1, state = 0
MB_CUR_MAX: 4
length = 3, state = 2
length = 3, state = 0
length = 1, state = 0 */

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 182
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 183
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 191
- “setlocale() — Set Locale” on page 322
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 425
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 451
- “<locale.h>” on page 7
- “<wchar.h>” on page 17
mbsinit() — Test State Object for Initial State

Format

#include <wchar.h>
int mbsinit (const mbstate_t *ps);

Language Level: ANSI

Threading: Yes

Description

If ps is not a null pointer, the mbsinit() function specifies whether the pointed to mbstate_t object describes an initial conversion state.

Note: This function is not available when you specify LOCALETYPE("CLD") on the compilation command.

Return Value

The mbsinit() function returns nonzero if ps is a null pointer or if the pointed to object describes an initial conversion state. Otherwise, it returns zero.

Example that uses mbsinit()

This example checks the conversion state to see if it is the initial state.

#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>
main()
{
    char *string = "ABC";
    mbstate_t state = 0;
    wchar_t wc;
    int rc;
    rc = mbrtowc(&wc, string, MB_CUR_MAX, &state);
    if (mbsinit(&state))
        printf("In initial conversion state\n");
}

Related Information

- "mbrlen() — Determine Length of a Multibyte Character (Restartable)" on page 183
- "mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)" on page 186
- "mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)" on page 191
- "setlocale() — Set Locale" on page 322
- "wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)" on page 425
- "wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)" on page 451
- "<locale.h>" on page 7
mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)

Format

#include <wchar.h>

size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len,
               mbstate_t *ps);

Language Level: ANSI

Threadsafe: Yes, if ps is not NULL.

Description

This function is the restartable version of mbstowcs().

The mbsrtowcs() function converts a sequence of multibyte characters that begins
in the conversion state described by ps from the array indirectly pointed to by src
into a sequence of corresponding wide characters. It then stores the converted
characters into the array pointed to by dst.

Conversion continues up to and including an ending null character, which is also
stored. Conversion will stop earlier in two cases: when a sequence of bytes are
reached that do not form a valid multibyte character, or (if dst is not a null pointer)
when len wide characters have been stored into the array pointed to by dst. Each
conversion takes place as if by a call to mbtowc() function.

If dst is not a null pointer, the pointer object pointed to by src will be assigned
either a null pointer (if conversion stopped due to reaching an ending null
character) or the address just past the last multibyte character converted. If
conversion stopped due to reaching an ending null character, the initial conversion
state is described.

The behavior of mbsrtowcs() is affected by the LC_CTYPE category of the current
locale.

Note: This function is not available when you specify LOCALETYPE("CLD) on the
compilation command.

Return Value

If the input string does not begin with a valid multibyte character, an encoding
error occurs, the mbsrtowcs() function stores the value of the macro EILSEQ in
errno, and returns (size_t) -1, but the conversion state will be unchanged.
Otherwise, it returns the number of multibyte characters successfully converted,
which is the same as the number of array elements modified when dst is not a null
pointer.

If a conversion error occurs, errno may be set to ECONVERT.

Example that uses mbsrtowcs()
Also see the examples for "mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)" on page 186.

Related Information
- "mblen() — Determine Length of a Multibyte Character" on page 182
- "mbrlen() — Determine Length of a Multibyte Character (Restartable)" on page 183
- "mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)" on page 186
- "mbstowcs() — Convert a Multibyte String to a Wide Character String"
- "setlocale() — Set Locale" on page 322
- "wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)" on page 425
- "wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)" on page 451
- "<locale.h>" on page 7
- "<wchar.h>" on page 17

mbstowcs() — Convert a Multibyte String to a Wide Character String

Format
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);

**Language Level:** ANSI  
**Threadsafe:** Yes.

**Description**

The `mbstowcs()` function determines the length of the sequence of the multibyte characters pointed to by `string`. It then converts the multibyte character string that begins in the initial shift state into a wide character string, and stores the wide characters into the buffer that is pointed to by `pwc`. A maximum of `n` wide characters are written.

The behavior of this function is affected by the LC_CTYPE category of the current locale. If the program is compiled with `LOCALETYPE(*LOCALE)`, and the CCSID that is associated with locale is single-byte, all the multibyte characters are assumed to be single-byte, and are converted to wide characters like this: 0xC1 is converted to 0x00C1. If CCSID of the locale is a multibyte CCSID, then if the multibyte character is a single-byte character, it is converted as is above. If the multibyte character is a double-byte character, (characters between `shiftout` 0x0e and `shiftin` 0x0f), the double-byte characters are copied directly, stripping off the `shiftout` and `shiftin` characters. For example, 0x0e41710f is converted to 0x4171.

If the program is compiled using `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)`, the multibyte character string is converted from the CCSID of the locale to UNICODE (13488) or UTF32 (1232) as if `iconv()` were called.

**Return Value**

The `mbstowcs()` function returns the number of wide characters generated, not including any ending null wide characters. If a multibyte character that is not valid is encountered, the function returns `(size_t)-1`.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

**Examples that use `mbstowcs()`**
/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and SYSIFCOPT(*IFSIO) */
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    char string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert a multibyte character when the CCSID of locale associated with LC_CTYPE is 37. */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string2, 10);
    /* In this case length ABC is converted to UNICODE ABC or 0x004100420043. Length will be 3. */
    printf("length = %d\n", length);

    /* Now lets try a multibyte example. We first must set the locale to a multibyte locale. We choose a locale with CCSID 5026 */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);
    /* The buffer now has the value: */
    /* 0x004103A103A30042 length is 4 */
    printf("length = %d\n", length);
}
/* The output should look like this:
length = 3
length = 4 */
/* This program is compiled with LOCALETYPE(*LOCALE) and SYSIFOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define  LOCNAME     "qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char  string[10];
    char  string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;

    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string2, 10);
    /* In this case length ABC is converted to */
    /* 0x00C100C200C3. Length will be 3. */
    printf("length = %d\n\n", length);

    /* Now lets try a multibyte example. We first must set the * */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026 */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);
    /* The buffer now has the value: */
    /* 0x00C14171417200C2 */
    printf("length = %d\n\n", length);
}

/* The output should look like this: */
length = 3
length = 4

Related Information
• "mblen() — Determine Length of a Multibyte Character" on page 182
mbtowc() — Convert Multibyte Character to a Wide Character

Format

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

Language Level: ANSI

Threading: No. Use mbtowcs() instead.

Description

The mbtowc() function first determines the length of the multibyte character pointed to by string. It then converts the multibyte character to a wide character as described in mbstowcs. A maximum of n bytes are examined.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Value

If string is NULL, the mbtowc() function returns:
- Nonzero when the active locale is mixed byte. The function initializes the state variable.
- 0 otherwise.

If string is not NULL, the mbtowc() function returns:
- 0 if string points to the null character
- The number of bytes comprising the converted multibyte character
- -1 if string does not point to a valid multibyte character.

If a conversion error occurs, errno may be set to ECONVERT.

Example that uses mbtowc()

This example uses the mblen() and mbtowc() functions to convert a multibyte character into a single wide character.
#include <stdio.h>
#include <stdlib.h>

#define LOCNAME "qsys.lib/mylib.lib/ja_jp959.locale"

int length, temp;
char string[] = "\x0e\x41\x71\x0f";
wchar_t arr[6];

int main(void)
{
    /* initialize internal state variable */
    temp = mbtowc(arr, NULL, 0);

    setlocale (LC_ALL, LOCNAME);
    /* Set string to point to a multibyte character. */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr, string, length);
    arr[1] = L'\0';
    printf("wide character string: %ls", arr);
}

Related Information
- "mblen() — Determine Length of a Multibyte Character” on page 182
- "mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 192
- "wcslen() — Calculate Length of Wide-Character String” on page 440
- "wctomb() — Convert Wide Character to Multibyte Character” on page 469
- "<stdlib.h>” on page 15

memchr() — Search Buffer

Format
#include <string.h>
void *memchr(const void *buf, int c, size_t count);

Language Level: ANSI

Threading safe: Yes.

Description
The memchr() function searches the first count bytes of buf for the first occurrence of c converted to an unsigned character. The search continues until it finds c or examines count bytes.

Return Value
The memchr() function returns a pointer to the location of c in buf. It returns NULL if c is not within the first count bytes of buf.

Example that uses memchr() 

This example finds the first occurrence of “x” in the string that you provide. If it is found, the string that starts with that character is printed.
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    char * result;
    if ( argc != 2 )
        printf( "Usage: %s string\n", argv[0] );
    else
    {
        if ((result = (char *) memchr( argv[1], 'x', strlen(argv[1])) ) != NULL)
            printf( "The string starting with x is %s\n", result );
        else
            printf( "The letter x cannot be found in the string\n" );
    }
}

/****************************** Output should be similar to:  ****************************
The string starting with x is xing
*/

Related Information

- "memcmp() — Compare Buffers"
- "memcpy() — Copy Bytes" on page 199
- "memmove() — Copy Bytes" on page 202
- "wmemchr() — Locate Wide Character in Wide-Character Buffer" on page 476
- "memset() — Set Bytes to Value" on page 203
- "strchr() — Search for Character" on page 342
- "<string.h>" on page 16

memcmp() — Compare Buffers

Format
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);

Language Level: ANSI

Threadsafe: Yes.

Description

The memcmp() function compares the first count bytes of buf1 and buf2.

Return Value

The memcmp() function returns a value indicating the relationship between the two buffers as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>buf1 less than buf2</td>
</tr>
<tr>
<td>0</td>
<td>buf1 identical to buf2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>buf1 greater than buf2</td>
</tr>
</tbody>
</table>
Example that uses `memcmp()`

This example compares first and second arguments passed to `main()` to determine which, if either, is greater.

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int len;
    int result;

    if ( argc != 3 )
    {
        printf( "Usage: %s string1 string2\n", argv[0] );
    }
    else
    {
        /* Determine the length to be used for comparison */
            len = strlen( argv[1] );
        else
            len = strlen( argv[2] );

        result = memcmp( argv[1], argv[2], len );

        printf( "When the first %i characters are compared,\n", len );
        if ( result == 0 )
            printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2] );
        else if ( result < 0 )
            printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
        else
            printf( "\"%s\" is greater than \"%s\"\n", argv[1], argv[2] );
    }
}

/****************  If the program is passed the arguments  **************
*****************        firststring and secondstring,  ************
*****************        output should be:                 ************

When the first 11 characters are compared, 
"firststring" is less than "secondstring"
**************************************************************************/
```

Related Information

- "memchr() — Search Buffer" on page 197
- "memcpy() — Copy Bytes"
- "wmemcmp() — Compare Wide-Character Buffers" on page 477
- "memmove() — Copy Bytes" on page 202
- "memset() — Set Bytes to Value" on page 203
- "strcmp() — Compare Strings" on page 343
- "<string.h>" on page 16

memcpy() — Copy Bytes

Format

```c
#include <string.h>
void *memcpy(void *dest, const void *src, size_t count);
```

Language Level: ANSI
Threadsafe: Yes.

Description

The memcpy() function copies count bytes of src to dest. The behavior is undefined if copying takes place between objects that overlap. The memmove() function allows copying between objects that may overlap.

Return Value

The memcpy() function returns a pointer to dest.

Example that uses memcpy()

This example copies the contents of source to target.

```c
#include <string.h>
#include <stdio.h>

#define MAX_LEN 80

char source[ MAX_LEN ] = "This is the source string";
char target[ MAX_LEN ] = "This is the target string";

int main(void)
{
    printf( "Before memcpy, target is \"%s\"\n", target );
    memcpy( target, source, sizeof(source) );
    printf( "After memcpy, target becomes \"%s\"\n", target );
}
```

/****************************** Expected output: ******************************/

Before memcpy, target is "This is the target string"
After memcpy, target becomes "This is the source string"
*/

Related Information

- "memchr() — Search Buffer" on page 197
- "memcmp() — Compare Buffers" on page 198
- "wmemcpy() — Copy Wide-Character Buffer" on page 478
- "memmove() — Copy Bytes" on page 202
- "memset() — Set Bytes to Value" on page 203
- "strcpy() — Copy Strings" on page 347
- "<string.h>" on page 16

memicmp() - Compare Bytes

Format

```c
#include <string.h>    // also in <memory.h>
int memicmp(void *buf1, void *buf2, unsigned int cnt);
```

Note: The memicmp function is available for C++ programs. It is available for C only when the program defines the __cplusplus_strings__ macro.

Language Level: Extension
Threadsafe: Yes.

Description

The `memicmp` function compares the first `cnt` bytes of `buf1` and `buf2` without regard to the case of letters in the two buffers. The function converts all uppercase characters into lowercase and then performs the comparison.

Return Value

The return value of `memicmp` indicates the result as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>buf1</code> less than <code>buf2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>buf1</code> identical to <code>buf2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>buf1</code> greater than <code>buf2</code></td>
</tr>
</tbody>
</table>

Example that uses `memicmp()`

This example copies two strings that each contain a substring of 29 characters that are the same except for case. The example then compares the first 29 bytes without regard to case.

```c
#include <stdio.h>
#include <string.h>
char first[100], second[100];
int main(void)
{
    int result;
    strcpy(first, "Those Who Will Not Learn From History");
    strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
    result = memicmp(first, second, 29);
    printf("Comparing the first 29 characters of two strings.
");
    if (result < 0)
        printf("The first 29 characters of String 1 are less than String 2.\n");
    else
        if (0 == result)
            printf("equal to String 2.\n");
        else
            printf("greater than String 2.\n");
    return 0;
}
```

The output should be:

Comparing the first 29 characters of two strings.
The first 29 characters of String 1 are equal to String 2

Related Information:

- “`memchr()` — Search Buffer” on page 197
- “`memicmp()` — Compare Buffers” on page 198
- “`memcpy()` — Copy Bytes” on page 199
- “`memmove()` — Copy Bytes” on page 202
- “`memset()` — Set Bytes to Value” on page 203
- “`strcmp()` — Compare Strings” on page 343
- “`strcmpi()` - Compare Strings Without Case Sensitivity” on page 345
- “`stricmp()` - Compare Strings without Case Sensitivity” on page 357
memmove() — Copy Bytes

Format

```
#include <string.h>
void *memmove(void *dest, const void *src, size_t count);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `memmove()` function copies `count` bytes of `src` to `dest`. This function allows copying between objects that may overlap as if `src` is first copied into a temporary array.

Return Value

The `memmove()` function returns a pointer to `dest`.

Example that uses `memmove()`

This example copies the word “shiny” from position `target + 2` to position `target + 8`.

```
#include <string.h>
#include <stdio.h>

#define SIZE    21
char target[SIZE] = "a shiny white sphere";

int main( void )
{
    char * p = target + 8; /* p points at the starting character
    of the word we want to replace */
    char * source = target + 2; /* start of "shiny" */

    printf( "Before memmove, target is "%s\n", target );
    memmove( p, source, 5 );
    printf( "After memmove, target becomes "%s\n", target );
}
```

/*---------------------- Expected output: ----------------------*
Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
*/

Related Information

- "memchr() — Search Buffer” on page 197
- "memcmp() — Compare Buffers” on page 198
- "wmemmove() — Copy Wide-Character Buffer” on page 479
- "memcpy() — Copy Bytes” on page 199
- "memset() — Set Bytes to Value” on page 203
memset() — Set Bytes to Value

Format

```c
#include <string.h>
void *memset(void *dest, int c, size_t count);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `memset()` function sets the first `count` bytes of `dest` to the value `c`. The value of `c` is converted to an unsigned character.

Return Value

The `memset()` function returns a pointer to `dest`.

Example that uses `memset()`

This example sets 10 bytes of the buffer to A and the next 10 bytes to B.

```c
#include <stdio.h>
#define BUF_SIZE 20

int main(void)
{
    char buffer[BUF_SIZE + 1];
    char *string;

    memset(buffer, 0, sizeof(buffer));
    string = (char *) memset(buffer, 'A', 10);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+10, 'B', 10);
    printf("\nBuffer contents: %s\n", buffer);
}
```

/******************** Output should be similar to: ********************

Buffer contents: AAAAAAAAAA

Buffer contents: AAAAAAAAAABBBBBBBBBB

*/

Related Information

- "memchr() — Search Buffer” on page 197
- "memcmp() — Compare Buffers” on page 198
- "memcpy() — Copy Bytes” on page 199
- "memmove() — Copy Bytes” on page 202
- "wmemset() — Set Wide Character Buffer to a Value” on page 480
- “<string.h>” on page 16
**mktime() — Convert Local Time**

**Format**
```
#include <time.h>

time_t mktime(struct tm *time);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `mktime()` function converts a stored `tm` structure (assume to be in job local time) pointed to by `time`, into a `time_t` structure suitable for use with other time functions. After the conversion, the `time_t` structure will be considered Universal Coordinate Time (UTC). For this conversion, `mktime()` checks the current locale setting for local time zone and Daylight Savings Time (DST). If these values are not set in the current locale, `mktime()` gets the local time zone and Daylight Saving Time settings from the current job. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored. `mktime()` then uses the current time zone information to determine UTC.

The values of some structure elements pointed to by `time` are not restricted to the ranges shown for `gmtime()`.

The values of `tm_wday` and `tm_yday` passed to `mktime()` are ignored and are assigned their correct values on return.

**Return Value**

The `mktime()` function returns Universal Coordinate Time (UTC) having type `time_t`. The value `(time_t)(-1)` is returned if the Universal Coordinate Time cannot be represented.

**Example that uses mktime()**

This example prints the day of the week that is 40 days and 16 hours from the current date.
Related Information

- "asctime() — Convert Time to Character String" on page 41
- "asctime_r() — Convert Time to Character String (Restartable)" on page 43
- "ctime() — Convert Time to Character String" on page 68
- "ctime_r() — Convert Time to Character String (Restartable)" on page 69
- "gmtime() — Convert Time" on page 151
- "gmtime_r() — Convert Time (Restartable)" on page 153
- "localtime() — Convert Time" on page 173
- "localtime_r() — Convert Time (Restartable)" on page 174
- "time() — Determine Current Time" on page 391
- "<time.h>" on page 16

---

modf() — Separate Floating-Point Value

**Format**

```
#include <math.h>
double modf(double x, double *intptr);
```

**Language Level:** ANSI

**Threading:** Yes.

**Description**

The **modf()** function breaks down the floating-point value \( x \) into fractional and integral parts. The signed fractional portion of \( x \) is returned. The integer portion is stored as a double value pointed to by \( intptr \). Both the fractional and integral parts are given the same sign as \( x \).

**Return Value**

The **modf()** function returns the signed fractional portion of \( x \).
Example that uses modf()

This example breaks the floating-point number -14.876 into its fractional and integral components.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x, y, d;
  x = -14.876;
  y = modf(x, &d);

  printf("x = %.2f\n", x);
  printf("Integral part = %.2f\n", d);
  printf("Fractional part = %.2f\n", y);
}
```

/******************** Output should be similar to: ********************

x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
*

Related Information

- "fmod() — Calculate Floating-Point Remainder" on page 99
- "frexp() — Separate Floating-Point Value" on page 122
- "ldexp() — Multiply by a Power of Two" on page 166
- "<math.h>" on page 7

nl_langinfo() —Retrieve Locale Information

Format

```c
#include <langinfo.h>
#include <nl_types.h>
char *nl_langinfo(nl_item item);
```

Language Level: XPG4

Threadsafe: No.

Description

The nl_langinfo() function retrieves from the current locale the string that describes the requested information specified by item.

The retrieval of the following information from the current locale is supported:

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODESET</td>
<td>CCSID of locale in character form</td>
</tr>
<tr>
<td>D_T_FMT</td>
<td>string for formatting date and time</td>
</tr>
<tr>
<td>D_FMT</td>
<td>date format string</td>
</tr>
<tr>
<td>T_FMT</td>
<td>time format string</td>
</tr>
<tr>
<td>T_FMT_AMPM</td>
<td>a.m. or p.m. time format string</td>
</tr>
</tbody>
</table>

206 ILE C/C++ Run-Time Library Functions V5R3
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM_STR</td>
<td>Ante Meridian affix</td>
</tr>
<tr>
<td>PM_STR</td>
<td>Post Meridian affix</td>
</tr>
<tr>
<td>DAY_1</td>
<td>name of the first day of the week (for example, Sunday)</td>
</tr>
<tr>
<td>DAY_2</td>
<td>name of the second day of the week (for example, Monday)</td>
</tr>
<tr>
<td>DAY_3</td>
<td>name of the third day of the week (for example, Tuesday)</td>
</tr>
<tr>
<td>DAY_4</td>
<td>name of the fourth day of the week (for example, Wednesday)</td>
</tr>
<tr>
<td>DAY_5</td>
<td>name of the fifth day of the week (for example, Thursday)</td>
</tr>
<tr>
<td>DAY_6</td>
<td>name of the sixth day of the week (for example, Friday)</td>
</tr>
<tr>
<td>DAY_7</td>
<td>name of the seventh day of the week (for example, Saturday)</td>
</tr>
<tr>
<td>ABDAY_1</td>
<td>abbreviated name of the first day of the week</td>
</tr>
<tr>
<td>ABDAY_2</td>
<td>abbreviated name of the second day of the week</td>
</tr>
<tr>
<td>ABDAY_3</td>
<td>abbreviated name of the third day of the week</td>
</tr>
<tr>
<td>ABDAY_4</td>
<td>abbreviated name of the fourth day of the week</td>
</tr>
<tr>
<td>ABDAY_5</td>
<td>abbreviated name of the fifth day of the week</td>
</tr>
<tr>
<td>ABDAY_6</td>
<td>abbreviated name of the sixth day of the week</td>
</tr>
<tr>
<td>ABDAY_7</td>
<td>abbreviated name of the seventh day of the week</td>
</tr>
<tr>
<td>MON_1</td>
<td>name of the first month of the year</td>
</tr>
<tr>
<td>MON_2</td>
<td>name of the second month of the year</td>
</tr>
<tr>
<td>MON_3</td>
<td>name of the third month of the year</td>
</tr>
<tr>
<td>MON_4</td>
<td>name of the fourth month of the year</td>
</tr>
<tr>
<td>MON_5</td>
<td>name of the fifth month of the year</td>
</tr>
<tr>
<td>MON_6</td>
<td>name of the sixth month of the year</td>
</tr>
<tr>
<td>MON_7</td>
<td>name of the seventh month of the year</td>
</tr>
<tr>
<td>MON_8</td>
<td>name of the eighth month of the year</td>
</tr>
<tr>
<td>MON_9</td>
<td>name of the ninth month of the year</td>
</tr>
<tr>
<td>MON_10</td>
<td>name of the tenth month of the year</td>
</tr>
<tr>
<td>MON_11</td>
<td>name of the eleventh month of the year</td>
</tr>
<tr>
<td>MON_12</td>
<td>name of the twelfth month of the year</td>
</tr>
<tr>
<td>ABMON_1</td>
<td>abbreviated name of the first month of the year</td>
</tr>
<tr>
<td>ABMON_2</td>
<td>abbreviated name of the second month of the year</td>
</tr>
<tr>
<td>ABMON_3</td>
<td>abbreviated name of the third month of the year</td>
</tr>
<tr>
<td>ABMON_4</td>
<td>abbreviated name of the fourth month of the year</td>
</tr>
<tr>
<td>ABMON_5</td>
<td>abbreviated name of the fifth month of the year</td>
</tr>
<tr>
<td>ABMON_6</td>
<td>abbreviated name of the sixth month of the year</td>
</tr>
<tr>
<td>ABMON_7</td>
<td>abbreviated name of the seventh month of the year</td>
</tr>
<tr>
<td>ABMON_8</td>
<td>abbreviated name of the eighth month of the year</td>
</tr>
<tr>
<td>ABMON_9</td>
<td>abbreviated name of the ninth month of the year</td>
</tr>
<tr>
<td>ABMON_10</td>
<td>abbreviated name of the tenth month of the year</td>
</tr>
<tr>
<td>ABMON_11</td>
<td>abbreviated name of the eleventh month of the year</td>
</tr>
<tr>
<td>ABMON_12</td>
<td>abbreviated name of the twelfth month of the year</td>
</tr>
<tr>
<td>ERA</td>
<td>era description segments</td>
</tr>
<tr>
<td>ERA_D_FMT</td>
<td>era date format string</td>
</tr>
<tr>
<td>ERA_D_T_FMT</td>
<td>era date and time format string</td>
</tr>
<tr>
<td>ERA_T_FMT</td>
<td>era time format string</td>
</tr>
<tr>
<td>ALT_DIGITS</td>
<td>alternative symbols for digits</td>
</tr>
<tr>
<td>RADIXCHAR</td>
<td>radix character</td>
</tr>
<tr>
<td>THOUSEP</td>
<td>separator for thousands</td>
</tr>
<tr>
<td>YESEXPR</td>
<td>affirmative response expression</td>
</tr>
<tr>
<td>NOEXPR</td>
<td>negative response expression</td>
</tr>
<tr>
<td>YESSTR</td>
<td>affirmative response for yes/no queries</td>
</tr>
<tr>
<td>NOSTR</td>
<td>negative response for yes/no queries</td>
</tr>
<tr>
<td>CRNCYSTR</td>
<td>currency symbol, preceded by ‘−’ if the symbol should appear before the value, ‘+’ if the symbol should appear after the value, or ‘.’ if the symbol should replace the radix character</td>
</tr>
</tbody>
</table>

**Returned Value**

The `nl_langinfo()` function returns a pointer to a null-ended string containing information concerning the active language or cultural area. The active language or cultural area is determined by the most recent `setlocale()` call. The array pointed to by the returned value is modified by subsequent calls to the function. The array should not be changed by the user’s program.

If the item is not valid, the function returns a pointer to an empty string.

**Example that uses `nl_langinfo()`**

This example retrieves the name of the codeset using the `nl_langinfo()` function.

```c
#include <langinfo.h>
#include <locale.h>
#include <nl_types.h>
#include <stdio.h>

int main(void)
{
    printf("Current codeset is %s\n", nl_langinfo(CODESET));
    return 0;
}

/************************************************************************
The output should be similar to:
Current codeset is 37
************************************************************************/
```

**Related Information**

- “`localeconv()` — Retrieve Information from the Environment” on page 168
- “`setlocale()` — Set Locale” on page 322
- “`<langinfo.h>`” on page 7
- “`<nl_types.h>`” on page 8
perror() — Print Error Message

Format
#include <stdio.h>
void perror(const char *string);

Language Level: ANSI

Threadsafe: Yes.

Description
The perror() function prints an error message to stderr. If string is not NULL and does not point to a null character, the string pointed to by string is printed to the standard error stream, followed by a colon and a space. The message associated with the value in errno is then printed followed by a new-line character.

To produce accurate results, you should ensure that the perror() function is called immediately after a library function returns with an error; otherwise, subsequent calls may alter the errno value.

Return Value
There is no return value.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADDATA</td>
<td>The message data is not valid.</td>
</tr>
<tr>
<td>EBUSY</td>
<td>The record or file is in use.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The file or library cannot be found.</td>
</tr>
<tr>
<td>EPERM</td>
<td>Insufficient authorization for access.</td>
</tr>
<tr>
<td>ENOREC</td>
<td>Record not found.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses perror()

This example tries to open a stream. If fopen() fails, the example prints a message and ends the program.
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fh;
    if ((fh = fopen("mylib/myfile","r")) == NULL)
    {
        perror("Could not open data file");
        abort();
    }
}

Related Information
- "clearerr() — Reset Error Indicators" on page 64
- "ferror() — Test for Read/Write Errors" on page 87
- "strerror() — Set Pointer to Run-Time Error Message" on page 350
- "<stdio.h>" on page 14

pow() — Compute Power

Format
#include <math.h>
double pow(double x, double y);

Language Level: ANSI

Threaddafe: Yes.

Description
The pow() function calculates the value of x to the power of y.

Return Value
If y is 0, the pow() function returns the value 1. If x is 0 and y is negative, the pow() function sets errno to EDOM and returns 0. If both x and y are 0, or if x is negative and y is not an integer, the pow() function sets errno to EDOM, and returns 0. The errno variable may also be set to ERANGE. If an overflow results, the pow() function returns +HUGE_VAL for a large result or -HUGE_VAL for a small result.

Example that uses pow()
This example calculates the value of 2^3.
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, y, z;
    x = 2.0;
    y = 3.0;
    z = pow(x, y);

    printf("%.lf to the power of %.lf is %.lf\n", x, y, z);
}

/*
************** Output should be similar to: **************
2.000000 to the power of 3.000000 is 8.000000
*/

Related Information
- “exp() — Calculate Exponential Function” on page 81
- “log() — Calculate Natural Logarithm” on page 175
- “log10() — Calculate Base 10 Logarithm” on page 176
- “sqrt() — Calculate Square Root” on page 337
- “<math.h>” on page 7

printf() — Print Formatted Characters

Format
#include <stdio.h>
int printf(const char *format-string, argument-list);

Language Level: ANSI

Threa$safe$: Yes.

Description

The printf() function formats and prints a series of characters and values to the standard output stream stdout. Format specifications, beginning with a percent sign (%), determine the output format for any argument-list following the format-string. The format-string is a multibyte character string beginning and ending in its initial shift state.

The format-string is read left to right. When the first format specification is found, the value of the first argument after the format-string is converted and output according to the format specification. The second format specification causes the second argument after the format-string to be converted and output, and so on through the end of the format-string. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. Only 15 significant digits are guaranteed for conversions of floating point numbers.

A format specification has the following form:
Conversions can be applied to the \textit{nth} argument after the \textit{format-string} in the argument list, rather than to the next unused argument. In this case, the conversion character \% is replaced by the sequence \%n\$, where \(n\) is a decimal integer in the range 1 thru NL\_ARGMAX, giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

\textbf{Alternative format specification} has the following form:

As an alternative, specific entries in the argument-list may be assigned by using the format specification outlined in the diagram above. This format specification and the previous format specification may not be mixed in the same call to \texttt{scanf()}. Otherwise, unpredictable results may occur.

The arg-number is a positive integer constant where 1 refers to the first entry in the argument-list. Arg-number may not be greater than the number of entries in the argument-list, or else the results are undefined. Arg-number also may not be greater than NL\_ARGMAX.

In format strings containing the \%n\$ form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the \%n\$ form of a conversion specification, a field width or precision may be indicated by the sequence \*m\$, where \(m\) is a decimal integer in the range 1 thru NL\_ARGMAX giving the position of an integer argument containing the field width or precision, for example:

\begin{verbatim}
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
\end{verbatim}

The format-string can contain either numbered argument specifications (that is, \%n\$ and \*m\$), or unnumbered argument specifications (that is, \%, and \*), but normally not both. The only exception to this is that \%\% can be mixed with the \%n\$ form. The results of mixing numbered and unnumbered argument specifications in a format-string string are undefined. When numbered argument specifications are used, specifying the \(n\)th argument requires that all the leading arguments, from the first to the (n-1)th, are specified in the format string.

Each field of the format specification is a single character or number signifying a particular format option. The \texttt{type} character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a \texttt{type} character (for example, \%s).
The following optional fields control other aspects of the formatting:

**Field**  
**Description**

**flags**  
Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for wchar_t precision unit.

**width**  
Minimum number of bytes output.

**precision**  
See Table 4 on page 217

**h, l, ll, L**  
Size of argument expected:

- **h**  
A prefix with the integer types d, i, o, u, x, X, and n that specifies that the argument is short int or unsigned short int.

- **l**  
A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long int or unsigned long int.

- **ll**  
A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long long int or unsigned long long int.

- **L**  
A prefix with e, E, f, g, or G types that specifies that the argument is long double.

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to stdout. For example, to print a percent sign character, use %%. 

The **type** characters and their meanings are given in the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>Integer</td>
<td>Signed decimal integer.</td>
</tr>
<tr>
<td>u</td>
<td>Integer</td>
<td>Unsigned decimal integer.</td>
</tr>
<tr>
<td>o</td>
<td>Integer</td>
<td>Unsigned octal integer.</td>
</tr>
<tr>
<td>x</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using abcd.</td>
</tr>
<tr>
<td>X</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using ABCDEF.</td>
</tr>
<tr>
<td>D(n,p)</td>
<td>Packed decimal</td>
<td>It has the format [-] dddd.dddd where the number of digits after the decimal point is equal to the precision of the specification. If the precision is missing, the default is p; if the precision is zero, and the # flag is not specified, no decimal point character appears. If the n and the p are *, an argument from the argument list supplies the value. n and p must precede the value being formatted in the argument list. At least one character appears before a decimal point. The value is rounded to the appropriate number of digits.</td>
</tr>
<tr>
<td>f</td>
<td>Double</td>
<td>Signed value having the form [-]d.dddd e[sign]dd, where d is a single-decimal digit, dddd is one or more decimal digits, ddd is 2 or 3 decimal digits, and sign is + or -.</td>
</tr>
<tr>
<td>Character</td>
<td>Argument</td>
<td>Output Format</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>E</td>
<td>Double</td>
<td>Identical to the e format except that E introduces the exponent instead of e.</td>
</tr>
<tr>
<td>g</td>
<td>Double</td>
<td>Signed value printed in f or e format. The e format is used only when the exponent of the value is less than -4 or greater than <code>precision</code>. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.</td>
</tr>
<tr>
<td>G</td>
<td>Double</td>
<td>Identical to the g format except that E introduces the exponent (where appropriate) instead of e.</td>
</tr>
<tr>
<td>c</td>
<td>Character (byte)</td>
<td>Single character.</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Characters (bytes) printed up to the first null character (\0) or until <code>precision</code> is reached.</td>
</tr>
<tr>
<td>n</td>
<td>Pointer to integer</td>
<td>Number of characters (bytes) successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.</td>
</tr>
<tr>
<td>p</td>
<td>Pointer</td>
<td>Pointer converted to a sequence of printable characters. It can be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• space pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• system pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• invocation pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• procedure pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• open pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• suspend pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• data pointer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• label pointer</td>
</tr>
<tr>
<td>lc or C</td>
<td>Wide Character</td>
<td>The (wchar_t) character is converted to a multibyte character as if by a call to wcrtomb(), and this character is printed out.¹</td>
</tr>
<tr>
<td>ls or S</td>
<td>Wide Character</td>
<td>The (wchar_t) characters up to the first (wchar_t) null character (L\0), or until precision is reached, are converted to multibyte characters, as if by a call to wcstombs(), and these characters are printed out. If the argument is a null string, (null) is printed.¹</td>
</tr>
</tbody>
</table>

**Note:** ¹ If the program is compiled with LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) and SYSFCOPT(*IFSIO), then the wide characters are assumed to be UCS2 (CCSID 13488) or UTF32 (CCSID 1232) characters. When printing to the display in a UNICODE environment, the wide characters are converted to the CCSID of the job before they are printed out.

The following list shows the format of the printed values for iSeries pointers, and gives a brief description of the components of the printed values.

**Space pointer:** SPP:Context:Object:Offset:AG

- **Context:** type, subtype and name of the context
- **Object:** type, subtype and name of the object
- **Offset:** offset within the space
- **AG:** Activation group ID

  Context: type, subtype and name of the context
  Object: type, subtype and name of the object
  Auth: authority
  Index: Index associated with the pointer
  AG: Activation group ID

Invocation pointer:  IVP:Index:AG

  Index: Index associated with the pointer
  AG: Activation group ID

Procedure pointer:  PRP:Index:AG

  Index: Index associated with the pointer
  AG: Activation group ID

Suspend pointer:  SUP:Index:AG

  Index: Index associated with the pointer
  AG: Activation group ID

Data pointer:  DTP:Index:AG

  Index: Index associated with the pointer
  AG: Activation group ID

Label pointer:  LBP:Index:AG

  Index: Index associated with the pointer
  AG: Activation group ID

NULL pointer:  NULL

The following restrictions apply to pointer printing and scanning on the iSeries system:

- If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer printed out.
- If a scanf() family function scans a pointer that was printed out by a different activation group, the scanf() family function will set the pointer to NULL.
- If a pointer is printed out in a Teraspace environment, just the hexadecimal value of the pointer is printed out. These results are the same as when using %#p.


The flag characters and their meanings are as follows (notice that more than one flag can appear in a format specification):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left-justify the result within the field width.</td>
<td>Right-justify.</td>
</tr>
<tr>
<td>Flag</td>
<td>Meaning</td>
<td>Default</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>+</td>
<td>Prefix the output value with a sign (+ or −) if the output value is of a signed type.</td>
<td>Sign appears only for negative signed values (−).</td>
</tr>
<tr>
<td>blank(‘ ’)</td>
<td>Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the blank flag if both appear, and a positive signed value will be output with a sign.</td>
<td>No blank.</td>
</tr>
<tr>
<td>#</td>
<td>When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.</td>
<td>No prefix.</td>
</tr>
<tr>
<td></td>
<td>When used with the f, D(n,p), e, or E formats, the # flag forces the output value to contain a decimal point in all cases.</td>
<td>Decimal point appears only if digits follow it.</td>
</tr>
<tr>
<td></td>
<td>When used with the g or G formats, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.</td>
<td>Decimal point appears only if digits follow it; trailing zeros are truncated.</td>
</tr>
<tr>
<td></td>
<td>When used with the ls format, the # flag causes precision to be measured in characters, regardless of the size of the character. For example, if single-byte characters are being printed, a precision of 4 would result in 4 bytes being printed. If double-byte characters are being printed, a precision of 4 would result in 8 bytes being printed.</td>
<td>Precision indicates the maximum number of bytes to be output.</td>
</tr>
<tr>
<td></td>
<td>When used with the p format, the # flag converts the pointer to hex digits. These hex digits cannot be converted back into a pointer, unless in a Teraspace environment.</td>
<td>Pointer converted to a sequence of printable characters.</td>
</tr>
<tr>
<td>0</td>
<td>When used with the d, i, D(n,p) o, u, x, X, e, f, g, or G formats, the 0 flag causes leading 0s to pad the output to the field width. The 0 flag is ignored if precision is specified for an integer or if the − flag is specified.</td>
<td>Space padding. No space padding for D(n,p).</td>
</tr>
</tbody>
</table>

The # flag should not be used with c, lc, d, i, u, or s types.

*Width* is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters (bytes) in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the − flag is specified) until the minimum width is reached.

*Width* never causes a value to be truncated; if the number of characters (bytes) in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

For the ls type, *width* is specified in bytes. If the number of bytes in the output value is less than the specified *width*, single-byte blanks are added on the left or the right (depending on whether the − flag is specified) until the minimum width is reached.
The \textit{width} specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The \textit{width} argument must precede the value being formatted in the argument list.

\textit{Precision} is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed or the number of decimal places. Unlike the \textit{width} specification, the \textit{precision} can cause truncation of the output value or rounding of a floating-point or packed decimal value.

The \textit{precision} specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The \textit{precision} argument must precede the value being formatted in the argument list.

The interpretation of the \textit{precision} value and the default when the \textit{precision} is omitted depend on the \textit{type}, as shown in the following table:

\textbf{Table 4. Values of Precision}

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>\textit{Precision} specifies the minimum number of digits to be printed. If the number of digits in the argument is less than \textit{precision}, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds \textit{precision}.</td>
<td>If \textit{precision} is 0 or omitted entirely, or if the period (.) appears without a number following it, the \textit{precision} is set to 1.</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>\textit{Precision} specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.</td>
<td>Default \textit{precision} for \textit{f}, \textit{e} and \textit{E} is six. Default \textit{precision} for \textit{D(n,p)} is \textit{p}. If \textit{precision} is 0 or the period appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>D(n,p)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>\textit{Precision} specifies the maximum number of significant digits printed.</td>
<td>All significant digits are printed. Default \textit{precision} is six.</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>No effect.</td>
<td>The character is printed.</td>
</tr>
<tr>
<td>Ic</td>
<td>No effect.</td>
<td>The \text{wchar_t} character is converted and resulting multibyte character is printed.</td>
</tr>
<tr>
<td>s</td>
<td>\textit{Precision} specifies the maximum number of characters to be printed. Characters in excess of \textit{precision} are not printed.</td>
<td>Characters are printed until a null character is encountered.</td>
</tr>
<tr>
<td>ls</td>
<td>\textit{Precision} specifies the maximum number of bytes to be printed. Bytes in excess of \textit{precision} are not printed; however, multibyte integrity is always preserved.</td>
<td>\text{wchar_t} characters are converted and resulting multibyte characters are printed.</td>
</tr>
</tbody>
</table>

\textbf{Return Value}

The \texttt{printf()} function returns the number of bytes printed. The value of \texttt{errno} may be set to:

\begin{tabular}{|c|c|}
\hline
\textbf{Value} & \textbf{Meaning} \\
\hline
\end{tabular}
EBADMODE
The file mode that is specified is not valid.

ECONVERT
A conversion error occurred.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

EILSEQ
An invalid multibyte character sequence was encountered.

EPUTANDGET
An illegal write operation occurred after a read operation.

ESTDOUT
stdout cannot be opened.

Note: The radix character for the printf() function is locale sensitive. The radix character is the decimal point to be used for the #flag character of the format string parameter for the format types f, D(n,p), e, E, g, and G.

Example that uses printf()

This example prints data in a variety of formats.
Example that uses printf()

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and*/
/* SYSIFOPT(*IFSIO) */
/* We will assume the locale setting is the same as the CCSID of the */
/* job. We will also assume any files involved have a CCSID of * */
/* 65535 (no convert). This way if printf goes to the screen or */
/* a file the output will be the same. */
int main(void)
{
    wchar_t wc = 0x0058;   /* UNICODE X */
    wchar_t ws[4];
    setlocale(LC_ALL, 
              "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;        /* UNICODE A   */
    ws[1] = (wchar_t)0x0042;        /* UNICODE B   */
    ws[2] = (wchar_t)0x0043;        /* UNICODE C   */
    ws[3] = (wchar_t)0x0000;  /* The output displayed is CCSID 37 */
    printf("%1c  %ls

",wc,ws);
    printf("%.2ls

",fp,fp,fp,fp);
}
//************** Output should be similar to: ***************
234  +234  000234  EA   ea   352
12345678901234567890123456789
Value of count should be 13; count = 13
h   h
   computer
   comp
251.736600    251.74    2.517366e+02    2.517366E+02
16  8  10
*******************************************************************/

Chapter 2. Library Functions

Example that uses printf()

```
"/QSYS.LIB/JA_JP.LOCALE"); /* a CCSID 5026 locale */

/* big A means an A that takes up 2 bytes on the screen */
/* It will look bigger then single byte A */
ws[0] = (wchar_t)0xFF21; /* UNICODE big A */
ws[1] = (wchar_t)0xFF22; /* UNICODE big B */
ws[2] = (wchar_t)0xFF23; /* UNICODE big C */
ws[3] = (wchar_t)0x0800;
wc = 0xFF11; /* UNICODE big 1 */
printf("%lc %ls\n\n", wc, ws);

/* The output of this printf is not shown below and it */
/* will differ depending on the device you display it */
/* but if you looked at the string in hex it would look */
/* like this: 0E42F10F404040400E42C142C242C30F */
/* 0E is shift out, 0F is shift in, and 42F1 is the */
/* big 1 in CCSID 5026 */
printf("%lc .4ls\n\n", wc, ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C10F */
/* Since the precision is in bytes we only get 4 bytes */
/* of the string. */
printf("%lc %.2ls\n\n", wc, ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C20F */
/* The # means precision is in characters reguardless */
/* of size. So we get 2 characters of the string. */

/* ************ Output should be similar to: *************/

X ABC
X AB

*****************************************************************************
Example that uses printf()
*****************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */
int main(void)
{
  wchar_t wc = (wchar_t)0x00C4; /* D */
  wchar_t ws[4];
  ws[0] = (wchar_t)0x00C1; /* A */
  ws[1] = (wchar_t)0x00C2; /* B */
  ws[2] = (wchar_t)0x00C3; /* C */
  ws[3] = (wchar_t)0x0000;
  /* The output displayed is CCSID 37 */
  printf("%lc %ls\n", wc, ws);
  /* Now lets try a mixed byte CCSID example */
  /* You would need a device that can handle mixed bytes to */
  /* display this correctly. */
  setlocale(LC_ALL,
  ":/QSYS.lib/JA_JP.LOCALE"); /* a CCSID 5026 locale */
/* big A means an A that takes up 2 bytes on the screen */
/* It will look bigger then single byte A */
ws[0] = (wchar_t)0x42C1;        /* big A   */
ws[1] = (wchar_t)0x42C2;        /* big B   */
ws[2] = (wchar_t)0x42C3;        /* big C   */
ws[3] = (wchar_t)0x0000;        /* big 1   */
w = 0x42F1;                    /* big 1   */
printf("%lc   %ls\n\n",w,ws);
/* The output of this printf is not shown below and it */
/* will differ depending on the device you display it */
/* but if you looked at the string in hex it would look */
/* like this: 0E42F10F404040E42C142C242C30F */
/* 0E is shift out, 0F is shift in, and 42F1 is the */
/* big 1 in CCSID 5026 */
printf("%lc   %.4ls\n\n",w,ws);
/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040E42C10F */
/* Since the precision is in bytes we only get 4 bytes */
/* of the string. */
printf("%lc   %#.2ls\n\n",w,ws);
/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040E42C120F */
/* The # means precision is in characters regardless */
/* of size. So we get 2 characters of the string. */
#endif

/*****************  Output should be similar to:  *****************
D     ABC
*******************************************************************/

Related Information

- "fprintf() — Write Formatted Data to a Stream" on page 108
- "fscanf() — Read Formatted Data" on page 123
- "scanf() — Read Data" on page 313
- "sprintf() — Print Formatted Data to Buffer" on page 335
- "sscanf() — Read Data" on page 339
- "vfprintf() — Print Argument Data to Stream" on page 404
- "vprintf() — Print Argument Data" on page 411
- "vsprintf() — Print Argument Data to Buffer" on page 415
- "wprintf() — Format Data as Wide Characters and Print" on page 481
- "<stdio.h>" on page 14

putc() – putchar() — Write a Character

Format

```c
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```
Language Level: ANSI

Threadsafe: No. #undef putc or #undef putchar allows the putc or putchar
function to be called instead of the macro version of these functions. The functions
are threadsafe.

Description

The putc() function converts c to unsigned char and then writes c to the output
stream at the current position. The putchar() is equivalent to putc(c, stdout).

The putc() function can be defined as a macro so the argument can be evaluated
multiple times.

The putc() and putchar() functions are not supported for files opened with
type=record.

Return Value

The putc() and putchar() functions return the character written. A return value of
EOF indicates an error.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses putc()

This example writes the contents of a buffer to a data stream. In this example, the
body of the for statement is null because the example carries out the writing
operation in the test expression.
Related Information

- "fputc() — Write Character" on page 109
- "fwrite() — Write Items" on page 136
- "getc() – getchar() — Read a Character” on page 142
- "puts() — Write a String" on page 224
- "putwc() — Write Wide Character" on page 225
- "putwchar() — Write Wide Character to stdout” on page 227
- "<stdio.h>” on page 14

putenv() — Change/Add Environment Variables

Format

```c
#include <stdlib.h>
#include <string.h>
#define LENGTH 80
int main(void)
{
    FILE *stream = stdout;
    int i, ch;
    char buffer[LENGTH + 1] = "Hello world";

    /* This could be replaced by using the fwrite routine */
    for (i = 0;
        (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
          ++i);
}
```

Expected output: Hello world

Related Information

- "fputc() — Write Character” on page 109
- "fwrite() — Write Items” on page 136
- "getc() – getchar() — Read a Character” on page 142
- "puts() — Write a String” on page 224
- "putwc() — Write Wide Character” on page 225
- "putwchar() — Write Wide Character to stdout” on page 227
- "<stdio.h>” on page 14

putenv() — Change/Add Environment Variables

Format

```c
#include <stdlib.h>
#include <string.h>
#define LENGTH 80
int *putenv(const char *varname);
```

Language Level: XPG4

Threadsafe: Yes,

Description

The putenv() function sets the value of an environment variables by altering an existing variable or creating a new one. The `varname` parameter points to a string of the form `var=x`, where `x` is the new value for the environment variable `var`.

The name cannot contain a blank or an equal ( = ) symbol. For example,

```
PATH NAME=/my_lib/joe_user
```

is not valid because of the blank between PATH and NAME. Similarly,

```
PATH=NAME=/my_lib/joe_user
```

is not valid because of the equal symbol between PATH and NAME. The system interprets all characters following the first equal symbol as being the value of the environment variable.
Note: This function accepts only EBCDIC character strings. When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the user may need to convert any non-EBCDIC character strings to EBCDIC before calling this function.

Return Value

The putenv() function returns 0 is successful. If putenv() fails then -1 is returned and errno is set to indicate the error.

Example that uses putenv()

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *pathvar;
    if (-1 == putenv("PATH=/:/home/userid")) {
        printf("putenv failed \n");
        return EXIT_FAILURE;
    }
    /* getting and printing the current environment path */
    pathvar = getenv("PATH");
    printf("The current path is: %s\n", pathvar);
    return 0;
}

/**************************************************************************
The output should be:
The current path is: /:/home/userid

Related Information
- "getenv() — Search for Environment Variables" on page 144
- "<stdlib.h>" on page 15

puts() — Write a String

Format
#include <stdio.h>
int puts(const char *string);

Language Level: ANSI

Threading: Yes.

Description

The puts() function writes the given string to the standard output stream stdout; it also appends a new-line character to the output. The ending null character is not written.

Return Value

The puts() function returns EOF if an error occurs. A nonnegative return value indicates that no error has occurred.
The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses `puts()`

This example writes Hello World to stdout.

```c
#include <stdio.h>

int main(void)
{
  if ( puts("Hello World") == EOF )
    printf("Error in puts\n");
}
```

Hello World

```c
/******************** Expected output:  ******************
Hello World
*/
```

Related Information

- "fputs() — Write String" on page 112
- "fputws() — Write Wide-Character String" on page 115
- "gets() — Read a Line" on page 146
- "putc() — putchar() — Write a Character" on page 221
- "putwc() — Write Wide Character"
- "<stdio.h>" on page 14

**putwc() — Write Wide Character**

**Format**

```c
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
```

**Language Level:** ANSI

**Threading:** Yes.

**Description**

The `putwc()` function writes the wide character `wc` to the stream at the current position. It also advances the file position indicator for the stream appropriately. The `putwc()` function is equivalent to the `fputwc()` function except that some platforms implement `putwc()` as a macro. Therefore, for portability, the stream argument to `putwc()` should not be an expression with side effects.
The behavior of the putwc() function is affected by the LC_CTYPE category of the current locale. Using a non-wide-character function with the putwc() function on the same stream results in undefined behavior. After calling the putwc() function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the putwc() function.

**Note:** This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Return Value**

The putwc() function returns the wide character written. If a write error occurs, it sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the putwc() function sets errno to EILSEQ and returns WEOF.

For information about errno values for putwc(), see “fputc() — Write Character” on page 109.

**Example that uses putwc()**

The following example uses the putwc() function to convert the wide characters in wcs to multibyte characters and write them to the file putwc.out.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    wchar_t *wcs = L"A character string.";
    int i;
    if (NULL == (stream = fopen("putwc.out", "w"))) {
        printf("Unable to open: "putwc.out\n")
        exit(1);
    }
    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwc(wcs[i], stream)) {
            printf("Unable to putwc() the wide character.\n"
                "wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n"
                exit(1);
        }
    }
    fclose(stream);
    return 0;
}
```

The output file putwc.out should contain:

A character string.

---

ILE C/C++ Run-Time Library Functions V5R3
putwchar() — Write Wide Character to stdout

Format

```
#include <wchar.h>
wint_t putwchar(wint_t wc);
```

Language Level: ANSI

Threading: Yes.

Description

The `putwchar()` function converts the wide character `wc` to a multibyte character and writes it to stdout. A call to the `putwchar()` function is equivalent to `putwc(wc, stdout)`.

The behavior of this function is affected by the LC_CTYPE category of the current locale. Using a non-wide-character function with the `putwchar()` function on the same stream results in undefined behavior. After calling the `putwchar()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `putwchar()` function.

Note: This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Return Value

The `putwchar()` function returns the wide character written. If a write error occurs, the `putwchar()` function sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the `putwchar()` function sets errno to EILSEQ and returns WEOF.

For information about errno values for `putwc()`, see “fputc() — Write Character” on page 109

Example that uses `putwchar()`

This example uses the `putwchar()` function to write the string in `wcs`.

Related Information

- “fputc() — Write Character” on page 109
- “fputwc() — Write Wide Character” on page 113
- “fputws() — Write Wide-Character String” on page 115
- “getwc() — Read Wide Character from Stream” on page 147
- “putc() – putchar() — Write a Character” on page 221
- “putwchar() — Write Wide Character to stdout”
- “<stdio.h>” on page 14
- “<wchar.h>” on page 17
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
    wchar_t *wcs = L"A character string."
    int i;
    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwchar(wcs[i])) {
            printf("Unable to putwchar() the wide character.\n");
            printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}

Related Information
• "fputc() — Write Character" on page 109
• "fputwc() — Write Wide Character" on page 113
• "fputws() — Write Wide-Character String" on page 115
• "getwchar() — Get Wide Character from stdin" on page 149
• "putc() – putchar() — Write a Character" on page 221
• "putwc() — Write Wide Character" on page 225
• "<wchar.h>" on page 17

qsort() — Sort Array

Format
#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
           int (*compare)(const void *key, const void *element));

Language Level: ANSI

Threadsafe: Yes.

Description
The qsort() function sorts an array of num elements, each of width bytes in size. The base pointer is a pointer to the array to be sorted. The qsort() function overwrites this array with the sorted elements.

The compare argument is a pointer to a function you must supply that takes a pointer to the key argument and to an array element, in that order. The qsort() function calls this function one or more times during the search. The function must compare the key and the element and return one of the following values:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><em>key</em> less than <em>element</em></td>
</tr>
<tr>
<td>0</td>
<td><em>key</em> equal to <em>element</em></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><em>key</em> greater than <em>element</em></td>
</tr>
</tbody>
</table>

The sorted array elements are stored in ascending order, as defined by your `compare` function. You can sort in reverse order by reversing the sense of “greater than” and “less than” in `compare`. The order of the elements is unspecified when two elements compare equally.

**Return Value**

There is no return value.

**Example that uses qsort()**

This example sorts the arguments (argv) in ascending lexical sequence, using the comparison function `compare()` supplied in the example.
QXXCHGDA() — Change Data Area

Format
#include <xxdtaa.h>

void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len,
               char *dtaptr);

Language Level: ILE C Extension

Threatsafe: Yes.

Description
The QXXCHGDA() function allows you to change the data area specified by dtaname, starting at position offset, with the data in the user buffer pointed to by dtaptr of length len. The structure dtaname contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

Related Information
- “bsearch() — Search Arrays” on page 54
- “<stdlib.h>” on page 15
LDA Specifies that the contents of the local data area are to be changed. The library name dtaa_lib must be blank.

GDA Specifies that the contents of the group data area are to be changed. The library name dtaa_lib must be blank.

data-area-name Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be changed. The library name dtaa_lib must be either *LIBL, *CURLIB, or the name of the library where the data area (data-area-name) is located. The data area is locked while it is being changed.

QXXCHGDA can only be used to change character data.

Note: This function accepts and returns only EBCDIC character strings. When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the user may need to convert any non-EBCDIC character strings to EBCDIC before calling this function and convert any resulting EBCDIC character strings to the correct CCSID.

Example that uses QXXCHGDA()

```c
#include <stdio.h>
#include <xxdtaa.h>

#define START   1
#define LENGTH  8

int main(void)
{
    char newdata[LENGTH] = "new data";

    /* The local data area will be changed */
    _DTAA_NAME_T dtaname = {"*LDA      ", "          "};

    /* Use function to change the local data area. */
    QXXCHGDA(dтанame,START,LENGTH,newdata);

    /* The first 8 characters in the local data area */
    /* are: new data */
}
```

Related Information
- "QXXRTVDA() — Retrieve Data Area" on page 236

QXXDTOP() — Convert Double to Packed Decimal

Format
```c
#include <xxcvt.h>
void QXXDTOP(unsigned char *pptr, int digits, int fraction, double value);
```

Language Level: ILE C Extension

Threadsafe: Yes.

Description
The QXXDTOP function converts the double value specified in value to a packed decimal number with digits total digits, and fraction fractional digits. The result is stored in the array pointed to by pptr.

Example that uses QXXDTOP()  

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;

    QXXDTOP(pptr, digits, fraction, value);
}
```

Related Information

- “QXXDTOZ() — Convert Double to Zoned Decimal”
- “QXXITOP() — Convert Integer to Packed Decimal” on page 233
- “QXXITOD() — Convert Integer to Zoned Decimal” on page 233
- “QXXPTOD() — Convert Packed Decimal to Double” on page 234
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 235
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 237
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 238

### QXXDTOZ() — Convert Double to Zoned Decimal

**Format**

```c
#include <xxcvt.h>
void QXXDTOZ(unsigned char *zpstr, int digits, int fraction,
             double value);
```

**Language Level:** ILE C Extension

**Thredsafte:** Yes.

**Description**

The QXXDTOZ function converts the double value specified in value to a zoned decimal number with digits total digits, and fraction fractional digits. The result is stored in the array pointed to by zptr.

Example that uses QXXDTOZ()  

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;

    QXXDTOZ(zptr, digits, fraction, value);
}                             /* Zoned value is : 03141593 */
```
QXXITOP() — Convert Integer to Packed Decimal

Format

```c
#include <xxcvt.h>
void QXXITOP(unsigned char *pptr, int digits, int fraction, int value);
```

Language Level: ILE C Extension

Threading: Yes.

Description

The QXXITOP function converts the integer specified in `value` to a packed decimal number with `digits` total digits, and `fraction` fractional digits. The result is stored in the array pointed to by `pptr`.

Example that uses QXXITOP()

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;
    QXXITOP(pptr, digits, fraction, value);
}
```

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 231
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 232
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 234
- “QXXPTOD() — Convert Packed Decimal to Double” on page 235
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 237
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 238

QXXITOZ() — Convert Integer to Zoned Decimal

Format

```c
#include <xxcvt.h>
void QXXITOZ(unsigned char *pptr, int digits, int fraction, int value);
```
#include <xxcvt.h>

void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The QXXITOZ function converts the integer specified in value to a zoned decimal number with digits total digits, and fraction fractional digits. The result is stored in the array pointed to by zptr.

Example that uses QXXITOZ()

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 9, fraction = 0;
    int value = 111115;

    QXXITOZ(zptr, digits, fraction, value);
    /* Zoned value is : 000111115 */
}
```

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 231
- “QXXDTOZ() —Convert Double to Zoned Decimal” on page 232
- “QXXITOP() —Convert Integer to Packed Decimal” on page 233
- “QXXPTOD() —Convert Packed Decimal to Double”
- “QXXPTOI() —Convert Packed Decimal to Integer” on page 235
- “QXXZTOD() —Convert Zoned Decimal to Double” on page 237
- “QXXZTOI() —Convert Zoned Decimal to Integer” on page 238

QXXPTOD() —Convert Packed Decimal to Double

Format

```c
#include <xxcvt.h>
double QXXPTOD(unsigned char *pptr, int digits, int fraction);
```

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The QXXPTOD function converts a packed decimal number to a double.

Example that uses QXXPTOD()
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 6.123456, result;
    /* First convert an integer to a packed decimal, */
    QXXITOP(pptr, digits, fraction, value);
    /* then convert it back to a double. */
    result = QXXPTOD(pptr, digits, fraction);
    /* result = 6.123456 */
}

**QXXPTOI() — Convert Packed Decimal to Integer**

**Format**

```
#include <xxcvt.h>
int QXXPTOI(unsigned char *pptr, int digits, int fraction);
```

**Language Level**: ILE C Extension

**Threadsafe**: Yes.

**Description**

The QXXPTOI function converts a packed decimal number to an integer.

**Example that uses QXXPTOI()**

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0, value = 104, result;
    /* First convert an integer to a packed decimal, */
    QXXITOP(pptr, digits, fraction, value);
    /* then convert it back to an integer. */
    result = QXXPTOI(pptr, digits, fraction);
    /* result = 104 */
}
```
QXXRTVDA() —Retrieve Data Area

Format

```c
#include <xxdtaa.h>

void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset,
                 short int len, char *dtaptr);
```

Language Level: ILE C Extension

Threading: Yes.

Description

The following typedef definition is included in the `<xxdtaa.h>` header file. The character arrays are not null-ended strings so they must be blank filled.

```c
typedef struct _DTAA_NAME_T {
    char dtaa_name[10]; /* name of data area */
    char dtaa_lib[10];  /* library that contains data area */
} _DTAA_NAME_T;
```

The `QXXRTVDA()` function retrieves a copy of the data area specified by `dtaname` starting at position `offset` with a length of `len`. The structure `dtaname` contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

- **LDA** The contents of the local data area are to be retrieved. The library name `dtaa_lib` must be blank.
- **GDA** The contents of the group data area are to be retrieved. The library name `dtaa_lib` must be blank.
- **PDA** Specifies that the contents of the program initialization parameters (PIP) data area are to be retrieved. The PIP data area is created for each pre-started job and is a character area up to 2000 characters in length. You cannot retrieve the PIP data area until you have acquired the requester. The library name `dtaa_lib` must be blank.
- **data-area-name** Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be retrieved. The library name `dtaa_lib` must be either *LIBL, *CURLIB, or the name of the library where the data area (data-area-name) is located. The data area is locked while the data is being retrieved.

The parameter `dtaptr` is a pointer to the storage that receives the retrieved copy of the data area. Only character data can be retrieved using QXXRTVDA.

Note: This function accepts and returns only EBCDIC character strings. When `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command, the user may need to convert any non-EBCDIC character strings to EBCDIC before calling this function and convert any resulting EBCDIC character strings to the correct CCSID.
Example that uses QXXRTVDA()

#include <stdio.h>
#include <xxdtaa.h>

#define DATA_AREA_LENGTH 30
#define START 6
#define LENGTH 7

int main(void)
{
    char uda_area[DATA_AREA_LENGTH];

    /* Retrieve data from user-defined data area currently in MYLIB */
    _DTAA_NAME_T dtaname = {"USRDDA", "MYLIB"};

    /* Use the function to retrieve some data into uda_area. */
    QXXRTVDA(dtaname, START, LENGTH, uda_area);

    /* Print the contents of the retrieved subset. */
    printf("uda_area contains %7.7s\n", uda_area);
}

Related Information
• "QXXCHGDA() — Change Data Area" on page 230

QXXZTOD() — Convert Zoned Decimal to Double

Format
#include <xxcvt.h>
double QXXZTOD(unsigned char *zptr, int digits, int fraction);

Language Level: ILE C Extension

Threadsafe: Yes.

Description
The QXXZTOD function converts to a double, the zoned decimal number (with digits total digits, and fraction fractional digits) pointed to by zptr. The resulting double value is returned.

Example that uses QXXZTOD()

#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "06123456";
    int digits = 8, fraction = 6;
    double result;

    result = QXXZTOD(zptr, digits, fraction);
    /* result = 6.123456 */
}

Related Information
• "QXXDTOP() — Convert Double to Packed Decimal" on page 231
• "QXXDTOZ() — Convert Double to Zoned Decimal" on page 232
QXXZTOI() —Convert Zoned Decimal to Integer

Format
#include <xxcvt.h>
int QXXZTOI(unsigned char *zptr, int digits, int fraction);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The QXXZTOI function converts to an integer, the zoned decimal number (with digits total digits, and fraction fractional digits) pointed to by zptr. The resulting integer is returned.

Example that uses QXXZTOI()

#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "000111115";
    int digits = 9, fraction = 0, result;
    result = QXXZTOI(zptr, digits, fraction);
    /* result = 111115 */
}

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 231
- “QXXDTOZ() —Convert Double to Zoned Decimal” on page 232
- “QXXITOP() —Convert Integer to Packed Decimal” on page 233
- “QXXITOZ() —Convert Integer to Zoned Decimal” on page 233
- “QXXPTOD() —Convert Packed Decimal to Double” on page 234
- “QXXPTOI() —Convert Packed Decimal to Integer” on page 235
- “QXXZTOD() —Convert Zoned Decimal to Double” on page 237

raise() — Send Signal

Format
#include <signal.h>
int raise(int sig);

Language Level: ANSI

Threadsafe: Yes.
Description

The `raise()` function sends the signal `sig` to the running program. If compiled with `SYSIFCOPT(*ASYNC SIGNAL)` on the compilation command, this function uses asynchronous signals. The asynchronous version of this function throws a signal to the process or thread.

Return Value

The `raise()` function returns 0 if successful, nonzero if unsuccessful.

Example that uses `raise()`

This example establishes a signal handler called `sig_hand` for the signal `SIGUSR1`. The signal handler is called whenever the `SIGUSR1` signal is raised and will ignore the first nine occurrences of the signal. On the tenth raised signal, it exits the program with an error code of 10. Note that the signal handler must be reestablished each time it is called.

```c
#include <signal.h>
#include <stdio.h>

void sig_hand(int); /* declaration of sig_hand() as a function */

int main(void)
{
    signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */
    raise(SIGUSR1); /* signal SIGUSR1 is raised */
    /* sig_hand() is called */
}

void sig_hand(int sig)
{
    static int count = 0; /* initialized only once */

    count++;
    if (count == 10) /* ignore the first 9 occurrences of this signal */
        exit(10);
    else
        signal(SIGUSR1, sig_hand); /* set up the handler again */
}
/* This is a program fragment and not a complete program */
```

Related Information

- "signal() — Handle Interrupt Signals" on page 330
- "Signal Handling Action Definitions" on page 490
- "<signal.h>" on page 13
- Signal APIs in the APIs topic in the iSeries Information Center.
- POSIX thread APIs in the APIs topic in the iSeries Information Center.

---

rand(), rand_r() — Generate Random Number

Format

```c
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seed);
```

Language Level: ANSI
Threadsafe: No. `rand()` is not threadsafe, but `rand_r()` is.

Description

The `rand()` function generates a pseudo-random integer in the range 0 to `RAND_MAX` (macro defined in `<stdlib.h>`). Use the `srand()` function before calling `rand()` to set a starting point for the random number generator. If you do not call the `srand()` function first, the default seed is 1.

Note: The `rand_r()` function is the restartable version of `rand()`.

Return Value

The `rand()` function returns a pseudo-random number.

Example that uses `rand()`

This example prints the first 10 random numbers generated.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int x;
    for (x = 1; x <= 10; x++)
        printf("iteration %d, rand=%d\n", x, rand());
}
```

/*
iteration 1, rand=16838
iteration 2, rand=5758
iteration 3, rand=10113
iteration 4, rand=17515
iteration 5, rand=31051
iteration 6, rand=5627
iteration 7, rand=23010
iteration 8, rand=7419
iteration 9, rand=16212
iteration 10, rand=4086 */

Related Information

- "`srand()` — Set Seed for rand() Function" on page 338
- "`<stdlib.h>`" on page 15

_Racquire() —Acquire a Program Device

Format

```c
#include <recio.h>
int _Racquire(_RFILE *fp, char *dev);
```

Language Level: ILE C Extension

Threadsafe: No.

Description
The _Racquire() function acquires the program device specified by the dev parameter and associates it with the file specified by fp. The dev parameter is a null-ended C string. The program device name must be specified in uppercase. The program device must be defined to the file.

Note: The dev parameter string must be EBCDIC, even if the program is compiled using LOCALETYPE(*LOCALEUTF).

This function is valid for display and ICF files.

Return Value

The _Racquire() function returns 1 if it is successful or zero if it is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred).

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Racquire()

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    _RFILE     *fp;
    _RIOFB_T   *rfb;

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Racquire ( fp,"DEVICE1" ); /* Acquire another program device. */
    /* Replace with actual device name.*/

    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the * /
    /* display file. */

    rfb = _Rwrite ( fp, ",", 0 ); /* Set up the display. */

    /* Do some processing... */

    _Rclose ( fp );
}
```

Related Information

* "_Rrelease() — Release a Program Device" on page 297

_Rclose() —Close a File

Format

```c
#include <recio.h>

int _Rclose(_RFILE *fp);
```
Language Level: ILE C Extension

Threadsafe: Yes.

Description

The _Rclose() function closes the file specified by fp. Before this file is closed, all buffers associated with it are flushed and all buffers reserved for it are released. The file is closed even if an exception occurs. The _Rclose() function applies to all types of files.

Note: Closing a file more than once in a multi-threaded environment will cause undefined behavior.

Return Value

The _Rclose() function returns zero if the file is closed successfully, or EOF if the close operation failed or the file was already closed. The file is closed even if an exception occurs, and zero is returned.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECCERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rclose()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE *fp;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    else
    /* Do some processing */;

    _Rclose ( fp );
}
```

Related Information

- "_Ropen() — Open a Record File for I/O Operations" on page 271
_Rcommit() —Commit Current Record

Format
#include <recio.h>
int _Rcommit(char *cmtid);

Language Level: ILE C Extension

Threadsafe: No.

Description

The _Rcommit() function completes the current transaction for the job that calls it and establishes a new commitment boundary. All changes made since the last commitment boundary are made permanent. Any file or resource that is open under commitment control in the job is affected.

The cmtid parameter is a null-ended C string used to identify the group of changes associated with a commitment boundary. It cannot be longer than 4000 bytes.

Note: The cmtid parameter string must be EBCDIC, even if the program is compiled using LOCALETYPE(*LOCALEUTF).

The _Rcommit() function applies to database and DDM files.

Return Value

The _Rcommit() function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred).

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rcommit()
```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char     buf[40];
    int      rc = 1;
    _RIFILE  *purf;
    _RIFILE  *dailyf;

    /* Open purchase display file and daily transaction file */
    if (( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.\n"");
        exit ( 1 );
    }

    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n"");
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file */
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}
```

Related Information

- "_Rrollbck() — Roll Back Commitment Control Changes" on page 300

_Rdelete() — Delete a Record

Format

```
#include <recio.h>
_RIOFB_T *_Rdelete(_RIFILE *fp);
```

Language Level: ILE C Extension
Threadsaf e: Yes.

Description

The _Rdelete() function deletes the record that is currently locked for update in the file specified by fp. After the delete operation, the record is not locked. The file must be open for update.

A record is locked for update by reading or locating to it unless __NO_LOCK is specified on the read or locate option. If the __NO_POSITION option is specified on the locate operation that locked the record, the record deleted may not be the record that the file is currently positioned to.

This function is valid for database and DDM files.

Return Value

The _Rdelete() function returns a pointer to the _RIOFB_T structure associated with fp. If the operation is successful, the num_bytes field contains 1. If the operation is unsuccessful, the num_bytes field contains zero.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTDLT</td>
<td>The file is not open for delete operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rdelete()
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE *fp;
    _XXOPFB_T *opfb;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile: %10.10s\n", 
        opfb->library_name, 
        opfb->file_name);
    /* Get the first record. */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );
    /* Delete the first record. */
    _Rdelete ( fp );
    _Rclose ( fp );
}

Related Information
• "Rrlslck() — Release a Record Lock" on page 299

_Rdevatr() — Get Device Attributes

Format
#include <recio.h>
#include <xxfdbk.h>
_XXDEV_ATR_T *_Rdevatr(_RFILE *fp, char *dev);

Language Level: ILE C Extension

Threadsafe: No.

Description
The _Rdevatr() function returns a pointer to a copy of the device attributes feedback area for the file pointed to by fp, and the device specified by dev.

The dev parameter is a null-ended C string. The device name must be specified in uppercase.

Note: The dev parameter string must be EBCDIC, even if the program is compiled using LOCALETYP(*LOCALEUTF).

The _Rdevatr() function is valid for display and ICF files.

Return Value
The _Rdevatr() function returns NULL if an error occurs.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rdevatr()

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    _XXDEV_ATR_T *dv_atr; /* Pointer to a copy of the file's device attributes feedback area */

    /* Open the device file. */
    if (( fp = _Ropen ("MYLIB/T1677RD2", "ar+" ) ) == NULL )
    {
        printf ("Could not open file\n" );
        exit (1 );
    }

    dv_atr = _Rdevatr (fp, argv[1]);
    if (dv_atr == NULL)
        printf("Error occurred getting device attributes for %s.\n", argv[1]);
    _Rclose (fp);
}

Related Information

- "_Racquire() — Acquire a Program Device" on page 240
- "_Rrelease() — Release a Program Device" on page 297

realloc() — Change Reserved Storage Block Size

Format

#define <stdlib.h>
void *realloc (void *ptr, size_t size);

Language Level: ANSI

Threaasafe: Yes.

Description

The realloc() function changes the size of a previously reserved storage block. The ptr argument points to the beginning of the block. The size argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

If the ptr is NULL, realloc() reserves a block of storage of size bytes. It does not necessarily give all bits of each element an initial value of 0.

If size is 0 and the ptr is not NULL, realloc() frees the storage allocated to ptr and returns NULL.
Return Value

The realloc() function returns a pointer to the reallocated storage block. The storage location of the block may be moved by the realloc() function. Thus, the ptr argument to the realloc() function is not necessarily the same as the return value.

If size is 0, the realloc() function returns NULL. If there is not enough storage to expand the block to the given size, the original block is unchanged and the realloc() function returns NULL.

The storage to which the return value points is aligned for storage of any type of object.

To use Teraspace storage instead of heap storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the CRTCMOD compiler command. This maps the realloc() library function to _C_TS_realloc(), its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to _C_TS_realloc() is 2GB - 240, or 214743408 bytes. For additional information about Teraspace, see the ILE Concepts manual.

Example that uses realloc()

This example allocates storage for the prompted size of array and then uses realloc() to reallocate the block to hold the new size of the array. The contents of the array are printed after each allocation.
```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array;    /* start of the array */
    long * ptr;      /* pointer to array   */
    int i;        /* index variable     */
    int num1, num2; /* number of entries of the array */
    void print_array( long  *ptr_array, int size);
    printf( "Enter the size of the array\n" );
    scanf( "%i", &num1);
    /* allocate num1 entries using malloc() */
    if ( (array = (long *) malloc( num1 * sizeof( long ))) != NULL )
    {
        for ( ptr = array, i = 0; i < num1 ; ++i ) /* assign values */
            *ptr++ = i;
        print_array( array, num1 );
        printf("\n");
    }
    else { /* malloc error */
        perror("Out of storage");
        abort();
    }
    /* Change the size of the array ... */
    printf("Enter the size of the new array\n");
    scanf("%i", &num2);
    if ( (array = (long *) realloc( array, num2* sizeof( long ))) != NULL )
    {
        for ( ptr = array + num1, i = num1; i <= num2; ++i ) /* assign values to new elements */
            *ptr++ = i + 2000;
        print_array( array, num2 );
    }
    else { /* realloc error */
        perror("Out of storage");
        abort();
    }
}

void print_array( long  *ptr_array, int size)
{
    int i;
    long * index = ptr_array;
    printf("The array of size %d is:\n", size);
    for ( i = 0; i < size; ++i ) /* print the array out */
        printf( " array[ %i ] = %li\n", i, ptr_array[i] );
}
```

**** If the initial value entered is 2 and the second value entered
is 4, then the expected output is:

Enter the size of the array
The array of size 2 is:
array[ 0 ] = 0
array[ 1 ] = 1
Enter the size of the new array
The array of size 4 is:
array[ 0 ] = 0
array[ 1 ] = 1
array[ 3 ] = 2003 */

Related Information

- "calloc() — Reserve and Initialize Storage" on page 57
- "free() — Release Storage Blocks" on page 118
- "malloc() — Reserve Storage Block" on page 180
regcomp() — Compile Regular Expression

Format

```c
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `regcomp()` function compiles the source regular expression pointed to by `pattern` into an executable version and stores it in the location pointed to by `preg`. You can then use the `regexec()` function to compare the regular expression to other strings.

The `cflags` flag defines the attributes of the compilation process:

<table>
<thead>
<tr>
<th>cflag</th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_ALT_NL</td>
<td>When LOCALETYPE(*LOCALE) is specified, the IFS newline character will be matched by regular expressions that match the newline character. When LOCALETYPE(*LOCALEUTF) is specified, the database newline character will be matched. The default behavior (REG_ALT_NL flag is not set) for LOCALETYPE(*LOCALE) is to match the database newline, and the default for LOCALETYPE(*LOCALEUTF) is to match the IFS newline. Note that for CCSID 1208 and CCSID 1232 the IFS newline character and the database newline character are the same.</td>
</tr>
<tr>
<td>REG_EXTENDED</td>
<td>Support extended regular expressions.</td>
</tr>
<tr>
<td>REG_NEWLINE</td>
<td>Treat new-line character as a special end-of-line character; it then establishes the line boundaries matched by the ] and $ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the new-line character is treated like any other character.)</td>
</tr>
<tr>
<td>REG_ICASE</td>
<td>Ignore case in match.</td>
</tr>
<tr>
<td>REG_NOSUB</td>
<td>Ignore the number of subexpressions specified in <code>pattern</code>. When you compare a string to the compiled pattern (using <code>regexec()</code>), the string must match the entire pattern. The <code>regexec()</code> function then returns a value that indicates only if a match was found; it does not indicate at what point in the string the match begins, or what the matching string is.</td>
</tr>
</tbody>
</table>
Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, which can be interpreted differently depending on the current locale. The functions regcomp(), regerror(), regexec(), and regfree() use regular expressions in a similar way to the UNIX® awk, ed, grep, and egrep commands.

**Note:** This function is not available when you specify LOCALETYPEx(*CLD) on the compilation command.

**Return Value**

If the regcomp() function is successful, it returns 0. Otherwise, it returns an error code that you can use in a call to the regerror() function, and the content of preg is undefined.

**Example that uses regcomp()**

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t    preg;
    char       *string = "a very simple simple simple string";
    char       *pattern = "\(sim[a-z]le\) \1";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s',returning %d\n", string, pattern, rc);
    }
    else {
        printf("With the whole expression, "
        "a matched substring \"simple simple\" is found at position %d to %d.\n",
        pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
        pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
        "a matched substring \"simple\" is found at position %d to %d.\n",
        pmatch[1].rm_eo - pmatch[1].rm_so, "string[pmatch[1].rm_so],
        pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}
```

The output should be similar to:

With the whole expression, a matched substring "simple simple" is found at position 7 to 19.
With the sub-expression, a matched substring "simple" is found at position 7 to 12.

*************************************************************************/
```
regerror() — Return Error Message for Regular Expression

Format

```c
#include <regex.h>
size_t regerror(int errcode, const regex_t *preg,  
    char *errbuf, size_t errbuf_size);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `regerror()` function finds the description for the error code `errcode` for the regular expression `preg`. The description for `errcode` is assigned to `errbuf`. The `errbuf_size` value specifies the maximum message size that can be stored (the size of `errbuf`). The description strings for `errcode` are:

<table>
<thead>
<tr>
<th><code>errcode</code></th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_NOMATCH</td>
<td><code>regexec()</code> failed to find a match.</td>
</tr>
<tr>
<td>REG_BADPAT</td>
<td>Invalid regular expression.</td>
</tr>
<tr>
<td>REG_ECOLLATE</td>
<td>Invalid collating element referenced.</td>
</tr>
<tr>
<td>REG_ECTYPE</td>
<td>Invalid character class type referenced.</td>
</tr>
<tr>
<td>REG_EESCAPE</td>
<td>Last character in regular expression is a <code>\</code>.</td>
</tr>
<tr>
<td>REG_ESUBREG</td>
<td>Number in <code>\d</code> invalid, or error.</td>
</tr>
<tr>
<td>REG_EBRACK</td>
<td><code>[]</code> imbalance.</td>
</tr>
<tr>
<td>REG_EPAREN</td>
<td><code>\(</code> <code>\)</code> or <code>()</code> imbalance.</td>
</tr>
<tr>
<td>REG_EBRACE</td>
<td><code>\{</code> <code>\}</code> imbalance.</td>
</tr>
<tr>
<td>REG_BADBR</td>
<td>Expression between <code>\{</code> and <code>\}</code> is invalid.</td>
</tr>
<tr>
<td>REG_ERANGE</td>
<td>Invalid endpoint in range expression.</td>
</tr>
<tr>
<td>REG_ESPACE</td>
<td>Out of memory.</td>
</tr>
<tr>
<td>REG_BADRPT</td>
<td>`?, *, or + not preceded by valid regular expression.</td>
</tr>
<tr>
<td>REG_ECHAR</td>
<td>Invalid multibyte character.</td>
</tr>
<tr>
<td>REG_EBOL</td>
<td><code>^</code> anchor not at beginning of regular expression.</td>
</tr>
<tr>
<td>REG_EEOL</td>
<td><code>$</code> anchor not at end of regular expression.</td>
</tr>
<tr>
<td>REG_ECOMP</td>
<td>Unknown error occurred during <code>regcomp()</code> call.</td>
</tr>
<tr>
<td>REG_EEXEC</td>
<td>Unknown error occurred during <code>regexec()</code> call.</td>
</tr>
</tbody>
</table>
Note: This function is not available when you specify LOCATELYTYPE(*CLD) on the compilation command.

Return Value

The regerror() returns the size of the buffer needed to hold the string that describes the error condition. The value of errno may be set to ECONVERT (conversion error).

Example that uses regerror()

This example compiles an invalid regular expression, and prints an error message using the regerror() function.

```c
#include <regex.htm>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t preg;
    char *pattern = "a[missing.bracket";
    int rc;
    char buffer[100];

    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        regerror(rc, &preg, buffer, 100);
        printf("regcomp() failed with '%s'\n", buffer);
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

The output should be similar to:

```plaintext
regcomp() failed with '[] imbalance.'
```

Related Information

- “regcomp() — Compile Regular Expression” on page 250
- “regexec() — Execute Compiled Regular Expression”
- “regfree() — Free Memory for Regular Expression” on page 256
- “<regex.h>” on page 12

regexec() — Execute Compiled Regular Expression

Format

```c
#include <regex.h>
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

Language Level: XPG4

Threadsafe: Yes.

Description
The `regexec()` function compares the null-ended `string` against the compiled regular expression `preg` to find a match between the two.

The `nmatch` value is the number of substrings in `string` that the `regexec()` function should try to match with subexpressions in `preg`. The array you supply for `pmatch` must have at least `nmatch` elements.

The `regexec()` function fills in the elements of the array `pmatch` with offsets of the substrings in `string` that correspond to the parenthesized subexpressions of the original pattern given to the `regcomp()` function to create `preg`. The zeroth element of the array corresponds to the entire pattern. If there are more than `nmatch` subexpressions, only the first `nmatch - 1` are stored. If `nmatch` is 0, or if the REG_NOSUB flag was set when `preg` was created with the `regcomp()` function, the `regexec()` function ignores the `pmatch` argument.

The `eflags` flag defines customizable behavior of the `regexec()` function:

<table>
<thead>
<tr>
<th>errflag</th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_NOTBOL</td>
<td>Indicates that the first character of <code>string</code> is not the beginning of line.</td>
</tr>
<tr>
<td>REG_NOTEOL</td>
<td>Indicates that the first character of <code>string</code> is not the end of line.</td>
</tr>
</tbody>
</table>

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of `string`. The following rules determine which substrings are reported in `pmatch`:

1. If subexpression `i` in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in `pmatch[i]` will delimit the last such match.
2. If subexpression `i` is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in `pmatch[i]` will be -1. A subexpression does not participate in the match when any of the following conditions are true:
   - `*` or `\{ \}` appears immediately after the subexpression in a basic regular expression.
   - `*`, `?`, or `\{` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times).
   - `1` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.
3. If subexpression `i` is contained within another subexpression `j`, and `i` is not contained within any other subexpression that is contained within `j`, and a match of subexpression `j` is reported in `pmatch[j]`, then the match or non-match of subexpression `i` reported in `pmatch[i]` will be as described in 1. and 2. above, but within the substring reported in `pmatch[j]` rather than the whole string.
4. If subexpression `i` is contained in subexpression `j`, and the byte offsets in `pmatch[j]` are -1, then the offsets in `pmatch[i]` also will be -1.
5. If subexpression `i` matched a zero-length string, then both byte offsets in `pmatch[i]` will be the byte offset of the character or null terminator immediately following the zero-length string.
If the REG_NOSUB flag was set when `preg` was created by the `regcomp()` function, the contents of `pmatch` are unspecified. If the REG_NEWLINE flag was set when `preg` was created, new-line characters are allowed in string.

**Note:** This function is not available when you specify LOCALETYPE(*CLD) on the compilation command.

### Return Value

If a match is found, the `regexec()` function returns 0. If no match is found, the `regexec()` function returns REG_NOMATCH. Otherwise, it returns a nonzero value indicating an error. A nonzero return value can be used in a call to the `regerror()` function.

#### Example that uses `regexec()`

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t    preg;
    char       *string = "a very simple simple simple string";
    char       *pattern = "\(sim[a-z]le\) \1";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s', returning %d.\n", 
                string, pattern, rc);
    }
    else {
        printf("With the whole expression, 
" "a matched substring \"%s\" is found at position %d to %d.\n", 
                pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so], 
                pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, 
" "a matched substring \"%s\" is found at position %d to %d.\n", 
                pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so], 
                pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }

    regfree(&preg);
    return 0;
}
```

The output should be similar to:

With the whole expression, a matched substring "simple simple" is found at position 7 to 19.
With the sub-expression, a matched substring "simple" is found at position 7 to 12.

---

#### Related Information
regfree() — Free Memory for Regular Expression

Format

```
#include <regex.h>
void regfree(regex_t *preg);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The regfree() function frees any memory that was allocated by the
regcomp() function to implement the regular expression preg. After the call to the
regfree() function, the expression that is defined by preg is no longer a compiled
regular or extended expression.

Note: This function is not available when you specify LOCALETYPEN(CLD) on the
compilation command.

Return Value

There is no return value.

Example that uses regfree()

This example compiles an extended regular expression.

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t preg;
    char    *pattern = ".*(simple).*";
    int     rc;

    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    regfree(&preg);
    printf("regcomp() is successful.\n");
    return 0;

    /*******************************************************************************
    The output should be similar to:
    regcomp() is successful.
    /*******************************************************************************
```

256  ILE C/C++ Run-Time Library Functions V5R3
remove() — Delete File

Format

```c
#include <stdio.h>
int remove(const char *filename);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `remove()` function deletes the file specified by `filename`. If the filename contains the member name, the member is removed or the file is deleted.

Note: You cannot remove a nonexistent file or a file that is open.

Return Value

The `remove()` function returns 0 if it successfully deletes the file. A nonzero return value indicates an error.

The value of `errno` may be set to `ECONVERT` (conversion error).

Example that uses `remove()`

When you call this example with a file name, the program attempts to remove that file. It issues a message if an error occurs.

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
    if ( argc != 2 )
        printf( "Usage: %s fn\n", argv[0] );
    else
        if ( remove( argv[1] ) != 0 )
            perror( "Could not remove file" );
}
```

rename() — Rename File

Format

```c
#include <stdio.h>
```
#include <stdio.h>
int rename(const char *oldname, const char *newname);

Language Level: ANSI

Threading: Yes.

Description

The `rename()` function renames the file specified by `oldname` to the name given by `newname`. The `oldname` pointer must specify the name of an existing file. The `newname` pointer must not specify the name of an existing file. You cannot rename a file with the name of an existing file. You also cannot rename an open file.

The file formats that can be used to satisfy the new name depend on the format of the old name. The following table shows the valid file formats that can be used to specify the old file name and the corresponding valid file formats for the new name.

If the format for both new name and old name is `lib/file(member)`, then the file cannot change. If the file name changes, rename will not work. For example, the following is not valid: `lib/file1(member1) lib/file2(member1).

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lib/file(member)</code></td>
<td><code>lib/file(member), lib/file, file, file(member)</code></td>
</tr>
<tr>
<td><code>lib/file</code></td>
<td><code>lib/file, file</code></td>
</tr>
<tr>
<td><code>file</code></td>
<td><code>lib/file, file</code></td>
</tr>
<tr>
<td><code>file(member)</code></td>
<td><code>lib/file(member), lib/file, file, file(member)</code></td>
</tr>
</tbody>
</table>

Return Value

The `rename()` function returns 0 if successful. On an error, it returns a nonzero value.

The value of `errno` may be set to `ECONVERT` (conversion error).

Example that uses `rename()`

This example takes two file names as input and uses `rename()` to change the file name from the first name to the second name.

```c
#include <stdio.h>

int main(int argc, char ** argv )
{
    if ( argc != 3 )
        printf( "Usage: %s old_fn new_fn\n", argv[0] );
    else if ( rename( argv[1], argv[2] ) != 0 )
        perror ( "Could not rename file" );
}
```

Related Information

- "fopen() — Open Files" on page 100
- "remove() — Delete File" on page 257
- "<stdio.h>" on page 14
rewind() — Adjust Current File Position

Format

#include <stdio.h>
void rewind(FILE *stream);

Language Level: ANSI

Threading: Yes.

Description

The rewind() function repositions the file pointer associated with stream to the beginning of the file. A call to the rewind() function is the same as:

(\texttt{void})fseek\texttt{(stream, 0L, SEEK\_SET);}

except that the rewind() function also clears the error indicator for the stream.

The rewind() function is not supported for files opened with type=record.

Return Value

There is no return value.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation attempted on a wrong device.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example that uses \texttt{rewind()}

This example first opens a file myfile for input and output. It writes integers to the file, uses \texttt{rewind()} to reposition the file pointer to the beginning of the file, and then reads in the data.
#include <stdio.h>

FILE *stream;

int data1, data2, data3, data4;
int main(void)
{
    data1 = 1; data2 = -37;

    /* Place data in the file */
    stream = fopen("mylib/myfile", "w+");
    fprintf(stream, "%d %d\n", data1, data2);

    /* Now read the data file */
    rewind(stream);
    fscanf(stream, "%d", &data3);
    fscanf(stream, "%d", &data4);
    printf("The values read back in are: %d and %d\n",
           data3, data4);
}

The values read back in are: 1 and -37

Related Information

- "fgetpos() — Get File Position" on page 91
- "fseek() — fseeko() — Reposition File Position" on page 124
- "fsetpos() — Set File Position" on page 126
- "ftell() — ftello() — Get Current Position" on page 128
- "<stdio.h>" on page 14

_Rfeod() —Force the End-of-Data

Format

#include <recio.h>

int _Rfeod(_RFILE *fp);

Language Level: ILE C Extension

Thraedsafe: Yes.

Description

The _Rfeod() function forces an end-of-data condition for a device or member associated with the file specified by fp. Any outstanding updates, deletes or writes that the system is buffering will be forced to nonvolatile storage. If a database file is open for input, any outstanding locks will be released.

The _Rfeod() function positions the file to *END unless the file is open for multi-member processing and the current member is not the last member in the file. If multi-member processing is in effect and the current member is not the last member in the file, _Rfeod() will open the next member of the file and position it to *START.

The _Rfeod() function is valid for all types of files.
Return Value

The _Rfeod() function returns 1 if multi-member processing is taking place and the next member has been opened. EOF is returned if the file is positioned to *END. If the operation is unsuccessful, zero is returned. The value of errno may be set to EIOERROR (a non-recoverable error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rfeod()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR 1002022244";

    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    }

    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data. */
    _Rfeod(in);
}
```

_Rfeov() —Force the End-of-File

**Format**

```c
#include <recio.h>

int _Rfeov(_RFILE *fp);
```

**Language Level**: ILE C Extension

**Threadsafe**: Yes.

**Description**

The _Rfeov() function forces an end-of-volume condition for a tape file that is associated with the file that is specified by fp. The _Rfeov() function positions the file to the next volume of the file. If the file is open for output, the output buffers will be flushed.

The _Rfeov() function is valid for tape files.
Return Value

The _Rfeov() function returns 1 if the file has moved from one volume to the next. It will return EOF if it is called while processing the last volume of the file. It will return zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rfeov()

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char buf[92];
    int i, feov2;

    /* Open source physical file containing C source. */
    if ((_fp = _Ropen("QCSRC(T1677SRC)", "rr blkrcd=y")) == NULL)
    {
        printf("could not open C source file\n");
        exit(1);
    }

    /* Open tape file to receive C source statements */
    if ((_tape = _Ropen("T1677TPF", "wr lrec=92 blkrcd=y")) == NULL)
    {
        printf("could not open tape file\n");
        exit(2);
    }

    /* Read the C source statements, find their sizes */
    /* and add them to the tape file. */
    while ((_Rreadn(_fp, buf, sizeof(buf), __DFT)) -> num_bytes != EOF)
    {
        for (i = sizeof(buf) - 1; buf[i] == ' ' && i > 12; --i);
        i = (i == 12) ? 80 : (1-12);
        memmove(buf, buf+12, i);
        _Rwrite(_tape, buf, i);
    }

    feov2 = _Rfeov(_fp);

    _Rclose(_fp);
    _Rclose(_tape);
}
```

Related Information

- "_Racquire() —Acquire a Program Device” on page 240
- "_Rfeod() —Force the End-of-Data” on page 260

_Rformat() —Set the Record Format Name

Format
### Language Level: ILE C Extension

### Threadsafe: Yes.

### Description

The `_Rformat()` function sets the record format to `fmt` for the file specified by `fp`.

The `fmt` parameter is a null-ended C string. The `fmt` parameter must be in uppercase.

**Note:** The `fmt` parameter string must be EBCDIC, even if the program is compiled using `LOCALETYPE(*LOCALEUTF)`.

The `_Rformat()` function is valid for multi-format logical database, DDM files, display, ICF and printer files.

### Return Value

The `_Rformat()` function returns void. See Table 12 on page 485 and Table 14 on page 489 for errno settings.

### Example that uses `_Rformat()`

This example shows how `_Rformat()` is used.
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char       buf[40];
    int        rc = 1;
    _RFILE     *purf;
    _RFILE     *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }

    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction. */
    /* The _RWrite function writes the purchase display. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file */
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollback ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf);
    _Rclose ( dailyf);
}

Related Information

• "_Ropen() — Open a Record File for I/O Operations" on page 271

_Rindara() — Set Separate Indicator Area

Format
#include <recio.h>

void _Rindara(_RFILE *fp, char *indic_buf);

Language Level: ILE C Extension
**Threadsafe:** No.

**Description**

The `_Rindara()` function registers `indic_buf` as the separate indicator area to be used by the file specified by `fp`. The file must be opened with the keyword `indicators=Y` on the `_Ropen()` function. The DDS for the file should specify also that a separate indicator area is to be used. It is generally best to initialize a separate indicator area explicitly with '0' (character) in each byte.

**Note:** The `indic_buf` parameter string must be EBCDIC, even if the program is compiled using `LOCALETYPE(*LOCALEUTF)`.

The `_Rindara()` function is valid for display, ICF, and printer files.

**Return Value**

The `_Rindara()` function returns void. See Table 12 on page 485 and Table 14 on page 489 for `errno` settings.

**Example that uses `_Rindara()`**
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define PF03 2
#define IND_OFF '0'
define IND_ON '1'

int main(void)
{
    char buf[40];
    int rc = 1;
    _SYSindara ind_area;
    _RFILE *purf;
    _RFILE *dailyf;
    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.
" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.
" );
        exit ( 2 );
    }
    /* Associate separate indicator area with purchase file */
    _Rindara ( purf, ind_area );
    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );
    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, ",", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
    /* While user is entering transactions, update daily and */
    /* monthly transaction files. */
    while ( rc && ind_area[PF03] == IND_OFF )
    {
        rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );
        /* If the databases were updated, then commit transaction */
        /* otherwise, rollback the transaction and indicate to the */
        /* user that an error has occurred and end the application. */
        if ( rc )
        {
            _Rcommit ( "Transaction complete" );
        }
        else
        {
            _Rrollback ( );
            _Rformat ( purf, "ERROR" );
        }
    }
    _Rclose ( purf );
    _Rclose ( dailyf );
}

Related Information
• "_Ropen() — Open a Record File for I/O Operations” on page 271

_Riofbk() — Obtain I/O Feedback Information

Format
#include <recio.h>
#include <xxfdbk.h>

_XXIOFB_T *_Riofbk(_RFILE *fp);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The _Riofbk() function returns a pointer to a copy of the I/O feedback area for the file that is specified by fp.

The _Riofbk() function is valid for all types of files.

Return Value

The _Riofbk() function returns NULL if an error occurs. See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Riofbk()

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE   *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats  buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if ((fp = _Ropen( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Racquire( fp,"DEVICE1" ); /* Acquire another device. Replace */
    /* with actual device name. */
    _Rformat( fp,"FORMAT1" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite( fp, "", 0 ); /* Set up the display. */
    _Rpgmdev( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd( fp, &buf, sizeof(buf) );
    rfb = _Rwrread( fp, &in_buf, sizeof(in_buf), &out_buf,
```
_Rlocate() — Position a Record

**Format**

```c
#include <recio.h>

_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);
```

**Language Level:** ILE C Extension

**Threading safe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The _Rlocate() function positions to the record in the file associated with fp and specified by the key, klen_rrn and opts parameters. The _Rlocate() function locks the record specified by the key, klen_rrn and opts parameters unless __NO_LOCK is specified.

**Note:** The key parameter string must be EBCDIC, even if the program is compiled using LOCALETYPE(*LOCALEUTF).

The _Rlocate() function is valid for database and DDM files that are opened with the _Ropen() function. The following are valid parameters of the _Rlocate() function.

- **key** Points to a string containing the key fields to be used for positioning.
- **klen_rrn** Specifies the length of the key that is used if positioning by key or the relative record number if positioning by relative record number.
- **opts** Specifies positioning options to be used for the locate operation. The possible macros are:
  - __DFT Default to __KEY_EQ and lock the record for update if the file is open for updating.

---

**Related Information**

- "_Ropnfbk() — Obtain Open Feedback Information" on page 275
Positions to just after the last record in a file. There is no record that is associated with this position.

Positions to just after the last record in a file. All buffered changes are made permanent. There is no record that is associated with this position.

Positions to the first record in the access path that is currently being used by *fp*. The *key* parameter is ignored.

Positions to the first record with the specified key.

Positions to the first record that has a key greater than or equal to the specified key.

Positions to the first record that has a key greater than the specified key.

Positions to the first record that has a key less than or equal to the specified key.

Positions to the first record that has a key less than the specified key.

Positions to the next record that has a key equal to the key value with a length of *klen_rrn*, at the current position. The *key* parameter is ignored.

Positions to the next record with a unique key from the current position in the access path. The *key* parameter is ignored.

Positions to the previous record with a key equal to the key value with a length of *klen_rrn*, at the current position. The *key* parameter is ignored.

Positions to the previous record with a unique key from the current position in the access path. The *key* parameter is ignored.

Positions to the last record in the access path that is currently being used by *fp*. The *key* parameter is ignored.

Positions to the next record in the access path that is currently being used by *fp*. The *key* parameter is ignored.

Positions to the previous record in the access path that is currently being used by *fp*. The *key* parameter is ignored.
__RRN_EQ
Positions to the record that has the relative record number
specified on the _klen_rrn parameter.

__START
Positions to just before the first record in the file. There is no
record that is associated with this position.

__START_FRC
Positions to just before the first record in a file. There is no record
that is associated with this position. All buffered changes are made
permanent.

__DATA_ONLY
Positions to data records only. Deleted records will be ignored.

__KEY_NULL_MAP
The NULL key map is to be considered when locating to a record
by key.

__NO_LOCK
The record that is positioned will not be locked.

__NO_POSITION
The position of the file is not changed, but the located record will
be locked if the file is open for update.

__PRIOR
Positions to just before the requested record.

If you specify the start and end options (__START, __START_FRC, __END or
__END_FRC) with any other options, the other options are ignored.

If you are positioned to __START or __END and perform a _Rreads operation, errno is set to EIOERROR.

Return Value

The _Rlocate() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rlocate() operation is successful, the num_bytes field contains 1. If __START, __START_FRC, __END or __END_FRC are specified, the num_bytes field is set to EOF. If the _Rlocate() operation is unsuccessful, the num_bytes field contains zero. The key and rrr fields are updated, and the key field will contain the complete key even if a partial key is specified.

The value of errno may be set to:

Table 5.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADKEYLN</td>
<td>The key length that is specified is not valid.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.
Example that uses _Rlocate()

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE  *in;
    char     new_purchase[21] = "PEAR 1002022244";

    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data. */
    _Rfeod(in);
    _Rclose(in);
}

Related Information

• "_Ropen() — Open a Record File for I/O Operations"

_Ropen() — Open a Record File for I/O Operations

Format

#include <recio.h>

_RFILE *_Ropen(const char * filename, const char * mode, ...);

Language Level: ILE C Extension

Threading: Yes.

Description

The _Ropen() function opens the record file specified by filename according to the mode parameter, which may be followed by optional parameters, if the varparm keyword parameter is specified in the mode parameter. The open mode and keyword parameters may be separated by a comma and one or more spaces. The _Ropen() function does not dynamically create database files for you. All of the files you refer to in the _Ropen() function must exist, or the open operation will fail.

Files that are opened by the _Ropen() function are closed implicitly when the activation group they are opened in, is ended. If a pointer to a file opened in one activation group is passed to another activation group and the opening activation group is ended, the file pointer will no longer be valid.
The _Ropen() function applies to all types of files. The filename variable is any valid iSeries system file name.

The mode parameter specifies the type of access that is requested for the file. It contains an open mode that is followed by optional keyword parameters. The mode parameter may be one of the following values:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rr</td>
<td>Open an existing file for reading records.</td>
</tr>
<tr>
<td>wr</td>
<td>Open an existing file for writing records. If the file contains data, the content is cleared unless the file is a logical file.</td>
</tr>
<tr>
<td>ar</td>
<td>Open an existing file for writing records to the end of the file (append).</td>
</tr>
<tr>
<td>rr+</td>
<td>Open an existing file for reading, writing or updating records.</td>
</tr>
<tr>
<td>wr+</td>
<td>Open an existing file for reading, writing or updating records. If the file contains data, the content is cleared unless the file is a logical file.</td>
</tr>
<tr>
<td>ar+</td>
<td>Open an existing file for reading and writing records. All data is written to the end of the file.</td>
</tr>
</tbody>
</table>

The mode may be followed by any of the following keyword parameters:

**Keyword**

**Description**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrseq=value</td>
<td>Specifies that the file is processed in arrival sequence.</td>
</tr>
<tr>
<td>blkrcd=value</td>
<td>Specifies that the file is processed using the access path that is used when the file was created. This is the default.</td>
</tr>
<tr>
<td>ccsid=value</td>
<td>Performs record blocking. The iSeries system determines the most efficient block size for you. This parameter is valid for database, DDM, diskette and tape files. It is only valid for files opened for input-only or output-only (modes rr, wr, or ar).</td>
</tr>
<tr>
<td>commit=value</td>
<td>Specifies the CCSID that is used for translation of the file. The default is 0 which indicates that the job CCSID is used.</td>
</tr>
<tr>
<td>dupkey=value</td>
<td>Specifies that the database file is opened under commitment control. Commitment control must have been set up prior to this.</td>
</tr>
<tr>
<td></td>
<td>Specifies that the database file is not opened under commitment control. This is the default.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Duplicate key values will be flagged in the _RIOFB_T structure.</td>
</tr>
<tr>
<td>N</td>
<td>Duplicate key values will not be flagged. This is the default.</td>
</tr>
</tbody>
</table>
indicators=value
Indicators are valid for printer, display, and ICF files. value can be:
Y The indicators that are associated with the file are returned in a separate indicator area instead of in the I/O buffers.
N The indicators are returned in the I/O buffers. This is the default.

lrecl=value
The length, in bytes, for fixed length records, and the maximum length for variable length records. This parameter is valid for diskette, display, printer, tape, and save files.

nullcap=value
Where value can be:
Y The program is capable of handling null fields in records. This is valid for database and DDM files.
N The program cannot handle null fields in records. This is the default.

riofb=value
Where value can be:
Y All fields in the _RIOFB_T structure are updated by any I/O operation that returns a pointer to the _RIOFB_T structure. However, the blk_filled_by field is not updated when using the _Rreadk function. This is the default.
N Only the num_bytes field in the _RIOFB_T structure is updated.

rtncode=value
Where value can be:
Y Use this option to bypass exception generation and handling. This will improve performance in the end-of-file and record-not-found cases. If the end-of-file is encountered, num_bytes will be set to EOF, but no errno values will be generated. If no record is found, num_bytes will be set to zero, and errno will be set to EIORECERR. This parameter is only valid for database and DDM files. For DDM files, num_bytes is not updated for _Rfeod.
N The normal exception generation and handling process will occur for the cases of end-of-file and record-not-found. This is the default.

secure=value
Where value can be:
Y Secures the file from overrides.
N Does not secure the file from overrides. This is the default.

splfname=(value)
For spooled output only. Where value can be:
*FILE The name of the printer file is used for the spooled output file name.

specoll-file-name
Specify the name of the spooled output file. A maximum of 10 characters can be used.
usrda=\(value\)

To specify, for spooled output only, user-specified data that identifies the file.

**user-data**

Specify up to 10 characters of user-specified text.

varparm=\(list\)

Where \(list\) is a list of optional keywords indicating which optional parameters will be passed to _Ropen(). The order of the keywords within the list indicates the order that the optional parameters will appear after the mode parameter. The following is a valid optional keyword:

**lvlchk**

The lvlchk keyword is used in conjunction with the lvlchk option on #pragma mapinc. When this keyword is used, a pointer to an object of type _LVLCHK_T (generated by #pragma mapinc) must be specified after the mode parameter on the _Ropen() function. For more details on this pointer, see the lvlchk option of #pragma mapinc in the *WebSphere Development Studio: ILE C/C++ Programmer’s Guide*.

vlr=\(value\)

Variable length record, where \(value\) is the minimum length of bytes of a record to be written to the file. The value can equal -1, or range from 0 to the maximum record length of the file. This parameter is valid for database and DDM files.

When VLR processing is required, _Ropen() will set min_length field. If the default value is not used, the minimum value that is provided by the user will be directly copied into min_length field. If the default value is specified, _Ropen() gets the minimum length from DB portion of the open data path.

Return Value

The _Ropen() function returns a pointer to a structure of type _RFILE if the file is opened successfully. It returns NULL if opening the file is unsuccessful.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADMODE</td>
<td>The file mode that is specified is not valid.</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>The file name that is specified is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.
Example that uses _Ropen()

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE *fp;
    /* Open the file for processing in arrival sequence. */
    if ((fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL)
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    else
    /* Do some processing */;

    _Rclose ( fp );
}

Related Information

- "_Rclose() — Close a File" on page 241
- "<recio.h>" on page 8

_Ropnfbk() — Obtain Open Feedback Information

Format
#include <recio.h>
#include <xxfdbk.h>

 XXOPFB_T *Ropnfbk(_RFILE *fp);

Language Level: ILE C Extension

Thesafe: Yes.

Description

The _Ropnfbk() function returns a pointer to a copy of the open feedback area for the file that is specified by fp.

The _Ropnfbk() function is valid for all types of files.

Return Value

The _Ropnfbk() function returns NULL if an error occurs. See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Ropnfbk()
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:    %10.10s\n", 
             opfb->library_name,
             opfb->file_name);
    _Rclose ( fp );
}

Related Information
• "_Rupfb() — Provide Information on Last I/O Operation” on page 304

_Rpgmdev() — Set Default Program Device

Format
#include <recio.h>
int _Rpgmdev(_RFILE *fp, char *dev);

Language Level: ILE C Extension

Threading: No.

Description
The _Rpgmdev() function sets the current program device for the file that is associated with fp to dev. You must specify the device in uppercase.

The dev parameter is a null-ended C string.

Note: The dev parameter string must be EBCDIC, even if the program is compiled using LOCALETYPE(*LOCALEUTF).

The _Rpgmdev() function is valid for display, ICF, and printer files.

Return Value
The _Rpgmdev() function returns 1 if the operation is successful or zero if the device specified has not been acquired for the file. See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rpgmdev()
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE   *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    formats   buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    /* Change the default program device. */
    /* Replace with actual device name. */
    _Rpgmdev ( fp,"DEVICE2" );

    /* Set the record format for the display file. */
    _Rformat ( fp,"FORMAT2" );

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf, sizeof(out_buf) );

    /* Continue processing. */

    _Rclose ( fp );
}

Related Information
- "_Racquire() — Acquire a Program Device" on page 240
- "_Rrelease() — Release a Program Device" on page 297

_Rreadd() — Read a Record by Relative Record Number

**Format**

```c
#include <recio.h>

_RIOFB_T _Rreadd (_RFILE *fp, void *buf, size_t size,
    int opts, long rrn);
```

**Language Level:** ILE C Extension
**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The _Rread() function reads the record that is specified by *rrn* in the arrival sequence access path for the file that is associated with *fp*. The _Rread() function locks the record specified by the *rrn* unless _NO_LOCK is specified. If the file is a keyed file, the keyed access path is ignored. Up to *size* number of bytes are copied from the record into *buf* (move mode only).

The following parameters are valid for the _Rread() function.

- **buf**: Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size**: Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- **rrn**: The relative record number of the record to be read.
- **opts**: Specifies the processing and access options for the file. The possible options are:
  - **__DFT**: If the file is opened for updating, then the record being read is locked for update. The previously locked record will no longer be locked.
  - **__NO_LOCK**: Does not lock the record being positioned to.

The _Rread() function is valid for database, DDM and display (subfiles) files.

**Return Value**

The _Rread() function returns a pointer to the _RIOFB_T structure associated with *fp*. If the _Rread() operation is successful the *num_bytes* field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). If blkrcd=Y and riofb=Y are specified, the blk_count and the blk_filled_by fields of the _RIOFB_T structure are updated. The key and *rrn* fields are also updated. If the file associated with *fp* is a display file, the sysparm field is updated. If it is unsuccessful, the *num_bytes* field is set to a value less than *size* and errno will be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadf()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *fp;
    _XXOPFB_T *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:    %10.10s\n",
             opfb->library_name,
             opfb->file_name);

    /* Get the second record. */
    _Rreadd ( fp, NULL, 20, __DFT, 2 );
    printf ( "Second record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

Related Information

- "_Rreadf() — Read the First Record"
- "_Rreadindv() — Read from an Invited Device” on page 281
- "_Rreadk() — Read a Record by Key” on page 284
- "_Rreadl() — Read the Last Record” on page 288
- "_Rreadn() — Read the Next Record” on page 289
- "_Rreadnc() — Read the Next Changed Record in a Subfile” on page 292
- "_Rreadp() — Read the Previous Record” on page 293
- "_Rreads() — Read the Same Record” on page 296

_Rreadf() — Read the First Record

Format

```c
#include <recio.h>

_RIOFB_T *Rreadf (_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level: ILE C Extension

Threadsafe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rreadf() function reads the first record in the access path that is currently being used for the file specified by fp. The access path may be keyed sequence or...
arrival sequence. The _Rreadf() function locks the first record unless __NO_LOCK is specified. Up to size number of bytes are copied from the record into buf (move mode only).

The following are valid parameters for the _Rreadf() function.

- **buf**: This parameter points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size**: This parameter specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.
- **opts**: This parameter specifies the processing and access options for the file. The possible options are:
  - **__DFT**: If the file is opened for updating, then the record being read or positioned to is locked for update. The previously locked record will no longer be locked.
  - **__NO_LOCK**: Does not lock the record being positioned to.

The _Rreadf() function is valid for database and DDM files.

**Return Value**

The _Rreadf() function returns a pointer to the _RIOFB_T structure that is specified by fp. If the _Rreadf() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The key and rrr fields are updated. If record blocking is taking place, the blk_count and blk_filled_by fields are updated. The num_bytes field is set to EOF if the file is empty. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

**Example that uses _Rreadf()**
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE     *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if ((fp = _Ropen( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf( "Library: %10.10s
File:    %10.10s
", opfb->library_name,
            opfb->file_name);

    /* Get the first record. */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf( "First record: %10.10s
", *(fp->in_buf) );

    /* Delete the first record. */
    _Rdelete ( fp );

    _Rclose ( fp );
}

Related Information

- "_Rread() — Read a Record by Relative Record Number" on page 277
- "Rreadindv() — Read from an Invited Device"
- "_Rreadk() — Read a Record by Key" on page 284
- "_Rreadl() — Read the Last Record" on page 288
- "_Rreadn() — Read the Next Record" on page 289
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 292
- "_Rreadp() — Read the Previous Record" on page 293
- "_Rreads() — Read the Same Record" on page 296

_Rreadindv() — Read from an Invited Device

Format

```c
#include <recio.h>

_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level: ILE C Extension

Threadsafe: No.

Description

The _Rreadindv() function reads data from an invited device.

The following are valid parameters for the _Rreadindv() function.
buf Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

size Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

opts Specifies the processing options for the file. Possible values are:

__DFT If the file is opened for updating, then the record being read or positioned to is locked. Otherwise, the option is ignored.

The _Rreadindv() function is valid for display and ICF files.

Return Value

The _Rreadindv() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadindv() function is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The sysparm and rrn (for subfiles) fields are also updated. The num_bytes field is set to EOF if the file is empty. If the _Rreadindv() function is unsuccessful, the num_bytes field is set to a value less than the value of size and the errno will be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadindv()
```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE  *fp;               /* File pointer */
    _RIOFB_T *rfb;        /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb;          /* Pointer to the file's feedback area */
    formats  buf, in_buf, out_buf          /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" );    /* Acquire another device. Replace */
    /* with actual device name. */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display. */
    _Rpgmdv ( fp,"DEVICE2" );     /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" );   /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 );   /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rrrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                    sizeof(out_buf) );
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
    /* Read from the first device that */
    /* enters data - device becomes */
    /* default program device. */

    /* Determine which terminal responded first. */
    iofb = _Riofbk ( fp);
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ) )
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }

    /* Continue processing. */
    printf ( "Data displayed is %45.45s\n", &buf);
    _Rclose ( fp);
}
```

**Related Information**

- "_Rread() — Read a Record by Relative Record Number" on page 277
_Rreadk() — Read a Record by Key

Format

```
#include <recio.h>

_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size, 
    int opts, void *key, unsigned int keylen);
```

Language Level: ILE C Extension

Threa safely: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rreadk() function reads the record in the keyed access path that is currently being used for the file that is associated with fp. Up to size number of bytes are copied from the record into buf (move mode only). The _Rreadk() function locks the record positioned to unless __NO_LOCK is specified. You must be processing the file using a keyed sequence path.

The following parameters are valid for the _Rreadk() function.

- **buf**: Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size**: Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.
- **key**: Points to the key to be used for reading.
- **keylen**: Specifies the total length of the key to be used.
- **opts**: Specifies the processing options for the file. Possible values are:
  - __DFT: Default to __KEY_EQ.
  - __KEY_EQ: Positions to and reads the first record that has the specified key.
  - __KEY_GE: Positions to and reads the first record that has a key greater than or equal to the specified key.
  - __KEY_GT: Positions and reads to the first record that has a key greater than the specified key.
  - __KEY_LE: Positions to and reads the first record that has a key less than or equal to the specified key.
Positions to and reads the first record that has a key less than the specified key.

Positions to and reads the next record that has a key equal to the key value at the current position. The key parameter is ignored.

Positions to and reads the next record with a unique key from the current position in the access path. The key parameter is ignored.

Positions to and reads the last record that has a key equal to the key value at the current position. The key parameter is ignored.

Positions to and reads the previous record with a unique key from the current position in the access path. The key parameter is ignored.

Do not lock the record for updating.

The positioning options are mutually exclusive.

The following options may be combined with the positioning options using the bit-wise OR (|) operator.

The NULL key map is to be considered when reading a record by key.

The record that is positioned will not be locked.

The _Rreadk() function is valid for database and DDM files.

Return Value

The _Rreadk() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rreadk() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields will be updated. The key field will always contain the complete key if a partial key is specified. When using record blocking with _Rreadk(), only one record is read into the block. Thus there are zero records remaining in the block and the blk_count field of the _RIOFB_T structure will be updated with 0. The blk_filled_by field is not applicable to _Rreadk() and is not updated. If the record specified by key cannot be found, the num_bytes field is set to zero. If you are reading a record by a partial key, then the entire key is returned in the feedback structure. If it is unsuccessful, the num_bytes field is set to a value less than size and errno will be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADKEYLN</td>
<td>The key length specified is not valid.</td>
</tr>
</tbody>
</table>
ENOTREAD
   The file is not open for read operations.

ETRUNC
   Truncation occurred on an I/O operation.

EIOERROR
   A non-recoverable I/O error occurred.

EIORECERR
   A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadk()
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE   *fp;
    _RIOFB_T *fb;
    char      buf[4];
    /* Create a physical file */
    system("CRTPF FILE(QTEMP/MY_FILE)");
    /* Open the file for write */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails\n");
        exit(1);
    }
    /* write some records into the file */
    _Rwrite(fp, "KEY9", 4);
    _Rwrite(fp, "KEY8", 4);
    _Rwrite(fp, "KEY7", 4);
    _Rwrite(fp, "KEY6", 4);
    _Rwrite(fp, "KEY5", 4);
    _Rwrite(fp, "KEY4", 4);
    _Rwrite(fp, "KEY3", 4);
    _Rwrite(fp, "KEY2", 4);
    _Rwrite(fp, "KEY1", 4);
    /* Close the file */
    _Rclose(fp);
    /* Open the file for read */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr")) == NULL )
    {
        printf("open for read fails\n");
        exit(2);
    }
    /* Read the record with key KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_EQ, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key less than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_LT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key greater than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_GT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with different key */
    fb = _Rreadk(fp, buf, 4, __KEY_NEXTUNQ, "", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Close the file */
    _Rclose(fp);
}

Related Information

- "_Rread() — Read a Record by Relative Record Number" on page 277
- "_Rreadf() — Read the First Record" on page 279
- "_Rreadindv() — Read from an Invited Device" on page 281
- "_Rreadl() — Read the Last Record" on page 288
- "_Rreadn() — Read the Next Record" on page 289
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 292
- "_Rreadp() — Read the Previous Record" on page 293
- "_Rreads() — Read the Same Record" on page 296
Format
#include <recio.h>

_RIOFB_T * _Rreadl(_RFILE *fp, void *buf, size_t size, int opts);

Language Level: ILE C Extension

Threading safe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rreadl() function reads the last record in the access path currently being used for the file specified by fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). The _Rreadl() function locks the last record unless __NO_LOCK is specified.

The following parameters are valid for the _Rreadl() function.

buf Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

size Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

opts Specifies the processing options for the file. Possible values are:

__DFT If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

__NO_LOCK Do not lock the record being positioned to.

The _Rreadl() function is valid for database and DDM files.

Return Value

The _Rreadl() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadl() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The key and rrn fields will be updated. If record blocking is taking place, the blk_count and blk_filled_by fields will be updated. If the file is empty, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than size and errno will be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
</tbody>
</table>
EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadl()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:    %10.10s\n",
             opfb->library_name,
             opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );
    _Rclose ( fp );
}
```

Related Information
- "_Rread() — Read a Record by Relative Record Number” on page 277
- "_Rreadf() — Read the First Record” on page 279
- "_Rreadindv() — Read from an Invited Device” on page 281
- "_Rreadk() — Read a Record by Key” on page 284
- "_Rreadn() — Read the Next Record”
- "_Rreadnc() — Read the Next Changed Record in a Subfile” on page 292
- "_Rreadp() — Read the Previous Record” on page 293
- "_Rreads() — Read the Same Record” on page 296

_Rreadn() — Read the Next Record

Format
```
#include <recio.h>

_RIOFB_T *_Rreadn (_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level: ILE C Extension
**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The `_Rreadn()` function reads the next record in the access path that is currently being used for the file that is associated with `fp`. The access path may be keyed sequence or arrival sequence. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreadn()` function locks the record positioned to unless `__NO_LOCK` is specified.

If the file associated with `fp` is opened for sequential member processing and the current record position is the last record of any member in the file except the last, `_Rreadn()` will read the first record in the next member of the file.

If an `_Rlocate()` operation positioned to a record specifying the `__PRIOR` option, `_Rreadn()` will read the record positioned to by the `_Rlocate()` operation.

If the file is open for record blocking and a call to `_Rreadp()` has filled the block, the `_Rreadn()` function is not valid if there are records remaining in the block. You can check the `blk_count` in `_RIOFB_T` to see if there are any remaining records.

The following are valid parameters for the `_Rreadn()` function.

- **buf** Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size** Specifies the number of bytes that are to be read and stored in `buf`. If locate mode is used, this parameter is ignored.
- **opts** Specifies the processing options for the file. Possible values are:
  - `__DFT` If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.
  - `__NO_LOCK` Do not lock the record being positioned to.

The `_Rreadn()` function is valid for all types of files except printer files.

**Return Value**

The `_Rreadn()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreadn()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The `key` and `rrn` fields are updated. If the file that is associated with `fp` is a display file, the `sysparm` field is also updated. If record blocking is taking place, the `blk_count` and the `blk_filled_by` fields of the `_RIOFB_T` structure are updated. If attempts are made to read beyond the last record in the file, the `num_bytes` field is set to EOF. If it is unsuccessful, the `num_bytes` field is set to a value less than `size`, and `errno` is changed. If you are using device files and specify zero as the `size`, check `errno` to determine if the function was successful.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>290</td>
<td>ILE C/C++ Run-Time Library Functions V5R3</td>
</tr>
</tbody>
</table>
ENOTREAD
The file is not open for read operations.

ETRUNC
Truncation occurred on an I/O operation.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadn()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s
File:    %10.10s
", 
             opfb->library_name,
             opfb->file_name);

    /* Get the first record. */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record:  %10.10s
", *(fp->in_buf) );

    /* Delete the second record. */
    _Rreadn ( fp, NULL, 20, __DFT );
    _Rdelete ( fp );

    _Rclose ( fp );
}
```

Related Information
- "_Rreadd() — Read a Record by Relative Record Number” on page 277
- "_Rreadf() — Read the First Record” on page 279
- "_Rreadindv() — Read from an Invited Device” on page 281
- "_Rreadk() — Read a Record by Key” on page 284
- "_Rreadl() — Read the Last Record” on page 288
- "_Rreadnc() — Read the Next Changed Record in a Subfile” on page 292
- "_Rreadp() — Read the Previous Record” on page 293
- "_Rreadr() — Read the Same Record” on page 296
_Rreadnc() — Read the Next Changed Record in a Subfile

Format

#include <recio.h>

_RIOFB_T *_Rreadnc(_RFILE *fp, void *buf, size_t size);

Language Level: ILE C Extension

Threading: No.

Description

The _Rreadnc() function reads the next changed record from the current position in the subfile that is associated with _fp. The minimum size of data that is read from the screen are copied from the system buffer to _buf.

The following are valid parameters for the _Rreadnc() function.

- **buf** Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size** Specifies the number of bytes that are to be read and stored in _buf.

The _Rreadnc() function is valid for subfiles.

Return Value

The _Rreadnc() function returns a pointer to the _RIOFB_T structure that is associated with _fp. If the _Rreadnc() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The rrn and sysparm fields are updated. If there are no changed records between the current position and the end of the file, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than _size, and errno is changed.

The value of errno may be set to:

**Value**  **Meaning**

ENOTREAD  The file is not open for read operations.

ETRUNC  Truncation occurred on an I/O operation.

EIOERROR  A non-recoverable I/O error occurred.

EIORECERR  A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreadnc()
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define LEN 10
#define NUM_RECS 20
#define SUBFILENAME "MYLIB/T1677RD6"
#define PFFILENAME "MYLIB/T1677RDB"

typedef struct {
    char name[LEN];
    char phone[LEN];
} pf_t;

#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE *pf;
    _RFILE *subf;

    /**********************************************************************************
     * Open the subfile and the physical file.                                        *
     ***********************************************************************************/
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL) {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL) {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }

    /**********************************************************************************
     * Initialize the subfile with records from the physical file.                    *
     ***********************************************************************************/
    init_subfile(pf, subf);

    /**********************************************************************************
     * Write the subfile to the display by writing a record to the subfile control format. *
     ***********************************************************************************/
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, ",", 0);
    _Rreadnc(subf, ",", 0);

    /**********************************************************************************
     * Close the physical file and the subfile.                                     *
     ***********************************************************************************/
    _Rclose(pf);
    _Rclose(subf);
}

Related Information

- "_Rread() — Read a Record by Relative Record Number" on page 277
- "_Rreadf() — Read the First Record" on page 279
- "_Rreadindv() — Read from an Invited Device" on page 281
- "_Rreadk() — Read a Record by Key" on page 284
- "_Rreadl() — Read the Last Record" on page 288
- "_Rreadn() — Read the Next Record" on page 289
- "_Rreadp() — Read the Previous Record"
- "_Reads() — Read the Same Record" on page 296

_Rreadp() — Read the Previous Record

Format
#include <recio.h>

_RIOFB_T * _Rreadp(_RFILE *fp, void *buf, size_t size, int opts);

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The _Rreadp() function reads the previous record in the access path that is currently being used for the file that is associated with fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). The _Rreadp() function locks the record positioned to unless __NO_LOCK is specified.

If the file associated with fp is opened for sequential member processing and the current record position is the first record of any member in the file except the first, _Rreadp() will read the last record in the previous member of the file.

If the file is open for record blocking and a call to _Rreadn() has filled the block, the _Rreadp() function is not valid if there are records remaining in the block. You can check the blk_count in _RIOFB_T to see if there are any remaining records.

The following are valid parameters for the _Rreadp() function.

- **buf**
  - Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  - Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

- **opts**
  - Specifies the processing options for the file. Possible values are:
    - __DFT
      - If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.
    - __NO_LOCK
      - Do not lock the record being positioned to.

The _Rreadp() function is valid for database and DDM files.

**Return Value**

The _Rreadp() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadp() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user’s buffer (move mode) or the record length of the file (locate mode). The key and rrn fields are also updated. If record blocking is taking place, the blk_count and the blk_filled_by fields of the _RIOFB_T structure are updated. If attempts are made to read prior to the first record in the file, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:

---

ILE C/C++ Run-Time Library Functions V5R3

---
Value | Meaning
---|---
ENOTREAD | The file is not open for read operations.
ETRUNC | Truncation occurred on an I/O operation.
EIOERROR | A non-recoverable I/O error occurred.
EIORECERR | A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses `_Rreadp()`

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE  *fp;
    _XXOPFB_T *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed
" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfb ( fp );
    printf ( "Library: %10.10s
File:    %10.10s
",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s
", *(fp->in_buf) );

    /* Get the previous record. */
    _Rreadp ( fp, NULL, 20, __DFT );
    printf ( "Next to last record: %10.10s
", *(fp->in_buf) );

    _Rclose ( fp );
}
```

Related Information
- "`_Rread()` — Read a Record by Relative Record Number” on page 277
- "`_Rreadf()` — Read the First Record” on page 279
- "`_Rreadindv()` — Read from an Invited Device” on page 281
- "`_Rreadk()` — Read a Record by Key” on page 284
- "`_Rreadl()` — Read the Last Record“ on page 288
- "`_Rreadn()` — Read the Next Record” on page 289
- "`_Rreadnc()` — Read the Next Changed Record in a Subfile” on page 292
- "`_Rreads()` — Read the Same Record” on page 296

Chapter 2. Library Functions  295
_Rreads() — Read the Same Record

Format

```c
#include <recio.h>

_RIOFB_T *_Rreads(_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level: ILE C Extension

Threadsafe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rreads() function reads the current record in the access path that is currently being used for the file that is associated with *fp*. The access path may be keyed sequence or arrival sequence. Up to *size* number of bytes are copied from the record into *buf* (move mode only). The _Rreads() function locks the record positioned to unless __NO_LOCK is specified.

If the current position in the file that is associated with *fp* has no record associated with it, the _Rreads() function will fail.

The _Rreads() function is not valid when the file is open for record blocking.

The following are valid parameters for the _Rreads() function.

- **buf**: Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- **size**: Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- **opts**: Specifies the processing options for the file. Possible values are:
  - **__DFT**: If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.
  - **__NO_LOCK**: Do not lock the record being positioned to.

The _Rreads() function is valid for database and DDM files.

Return Value

The _Rreads() function returns a pointer to the _RIOFB_T structure that is associated with *fp*. If the _Rreads() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrrn fields are also updated. If it is unsuccessful, the num_bytes field is set to a value less than *size*, and errno is changed.

The value of errno may be set to:

- **ENOTREAD**: The file is not open for read operations.

296   ILE C/C++ Run-Time Library Functions V5R3
ETRUNC
Truncation occurred on an I/O operation.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rreads()

```c
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
        { printf ( "Open failed\n" );
          exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile: %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* Get the same record without locking it. */
    _Rreads ( fp, NULL, 20, __NO_LOCK); 
    printf ( "Same record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

Related Information
- "_Rreadl() — Read a Record by Relative Record Number" on page 277
- "_Rread() — Read the First Record" on page 279
- "_Rreadindv() — Read from an Invited Device" on page 281
- "_Rreadk() — Read a Record by Key" on page 284
- "_Rreadl() — Read the Last Record" on page 288
- "_Rreadn() — Read the Next Record" on page 289
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 292
- "_Rreadp() — Read the Previous Record" on page 293

_Rrelease() — Release a Program Device

Format
```c
#include <recio.h>

int _Rrelease(_RFILE *fp, char *dev);

Language Level: ILE C Extension

Threading: No.

Description

The _Rrelease() function releases the program device that is specified by dev from the file that is associated with fp. The device name must be specified in uppercase.

The dev parameter is a null-ended C string.

Note: The dev parameter string must be EBCDIC, even if the program is compiled using LOCALETYPE(*LOCALEUTF).

The _Rrelease() function is valid for display and ICF files.

Return Value

The _Rrelease() function returns 1 if it is successful or zero if it is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rrelease()
```
```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
  char name[20];
  char address[25];
} format1;

typedef struct {
  char name[8];
  char password[10];
} format2;

typedef union {
  format1 fmt1;
  format2 fmt2;
} formats;

int main(void)
{
  _RFILE *fp; /* File pointer */
  _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
  _XXIOFB_T *iofb; /* Pointer to the file's feedback area */

  formats buf, in_buf, out_buf; /* Buffers to hold data */

  /* Open the device file. */
  if (( fp = _Ropen("MYLIB/T1677RD2", "ar+" )) == NULL )
    { printf("Could not open file\n");
      exit(1);
    }

  _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace */
  _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
                       /* display file. */
  rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */

  /* Do some I/O. */

  /* Close the device. */
  _RClose ( fp );

  return (0);
}
```
_Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
/* Replace with actual device name. */
_Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
/* display file. */

rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */

rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
         sizeof(out_buf));
_Rreadindv ( fp, &buf, sizeof(buf), __DFT );
   /* Read from the first device that */
   /* enters data - device becomes */
   /* default program device. */

/* Determine which terminal responded first. */

iofb = _Riofbk ( fp );
if (!strncmp ( "FORMAT1", iofb -> rec_format, 10 ))
  {
    _Rrelease ( fp, "DEVICE1" );
  }
else
  {
    _Rrelease ( fp, "DEVICE2" );
  }

/* Continue processing. */

printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp );

Related Information

• "_Racquire() —Acquire a Program Device" on page 240

_Rrlslck() — Release a Record Lock

Format
#include <recio.h>

int _Rrlslck (_RFILE *fp);

Language Level: ILE C Extension

Threadsafe: Yes.

Description

The _Rrlslck() function releases the lock on the currently locked record for the file specified by *fp. The file must be open for update, and a record must be locked. If the _NO_POSITION option was specified on the _Rlocate() operation that locked the record, the record released may not be the record currently positioned to.

The _Rrlslck() function is valid for database and DDM files.

Return Value

The _Rrlslck() function returns 1 if the operation is successful, or zero if the operation is unsuccessful.

The value of errno may be set to:

Value Meaning

ENOTUPD
The file is not open for update operations.
EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rrlslck()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char        buf[21];
    _RFILE      *fp;
    _XXOPFB_T   *opfb;
    int         result;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:    %10.10s\n",
              opfb->library_name,
              opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* _Rrlslck example. */
    result = _Rrlslck ( fp );
    if ( result == 0 )
        printf("_Rrlslck failed.\n");

    _Rclose ( fp );
}
```

Related Information
- “_Rdelete() — Delete a Record” on page 244

_Rrollbck() — Roll Back Commitment Control Changes

Format
```c
#include <recio.h>

int _Rrollbck(void);
```

Language Level: ILE C Extension

Threading: No.

Description
The _Rrollbck() function reestablishes the last commitment boundary as the current commitment boundary. All changes that are made to the files under commitment control in the job, are reversed. All locked records are released. Any file that is open under commitment control in the job will be affected. You must specify the keyword parameter commit=y when the file is opened to be under commitment control. A commitment control environment must have been set up prior to this.

The _Rrollbck() function is valid for database and DDM files.

Return Value

The _Rrollbck() function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rrollbck()

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char       buf[40];
    int        rc = 1;
    _RFILE     *purf;
    _RFILE     *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, \"\", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file */
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ) )->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
```

Chapter 2. Library Functions 301
Related Information

- "_Rcommit() — Commit Current Record" on page 243
- Backup and Recovery manual

_Rupdate() — Update a Record

Format

#include <recio.h>

_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);

Language Level: ILE C Extension

Threading: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rupdate() function updates the record that is currently locked for update in the file that is specified by fp. The file must be open for update. A record is locked for update by reading or locating to it unless _NO_LOCK is specified on the read or locate operation. If the _NO_POSITION option is specified on a locate operation the record updated may not be the record currently positioned to. After the update operation, the updated record is no longer locked.

The number of bytes that are copied from buf to the record is the minimum of size and the record length of the file (move mode only). If size is greater than the record length, the data is truncated, and errno is set to ETRUNC. One complete record is always written to the file. If the size is less than the record length of the file, the remaining data in the record will be the original data that was read into the system buffer by the read that locked the record. If a locate operation locked the record, the remaining data will be what was in the system input buffer prior to the locate.

The _Rupdate() function can be used to update deleted records and key fields. A deleted record that is updated will no longer be marked as a deleted record. In both of these cases any keyed access paths defined for fp will be changed.

Note: If locate mode is being used, _Rupdate() works on the data in the file's input buffer.

The _Rupdate() function is valid for database, display (subfiles) and DDM files.

Return Value

The _Rupdate() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rupdate() function is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). If fp is a display file, the sysparm field is updated. If the _Rupdate() function is unsuccessful, the
num_bytes field is set to a value less than the size specified (move mode) or zero (locate mode). The errno value will also be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rupdate()

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";

    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    }

    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data. */
    _Rfeod(in);
    _Rclose(in);
}
```

Related Information

- "_Rreadd() — Read a Record by Relative Record Number" on page 277
- "_Rreadf() — Read the First Record" on page 279
- "_Rreadindv() — Read from an Invited Device" on page 281
- "_Rreadk() — Read a Record by Key" on page 284
- "_Rreadl() — Read the Last Record" on page 288
- "_Rreadn() — Read the Next Record" on page 289
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 292
- "_Rreadp() — Read the Previous Record" on page 293
- "_Rreads() — Read the Same Record" on page 296
_Rupfb() — Provide Information on Last I/O Operation

Format

#include <recio.h>

_RIOFB_T *_Rupfb(_RFILE *fp);

Language Level: ILE C Extension

Threadsafe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rupfb() function updates the feedback structure associated with the file specified by fp with information about the last I/O operation. The _RIOFB_T structure will be updated even if riofb=N was specified when the file was opened. The num_bytes field of the _RIOFB_T structure will not be updated. See "<recio.h>" on page 8 for a description of the _RIOFB_T structure.

The _Rupfb() function is valid for all types of files.

Return Value

The _Rupfb() function returns a pointer to the _RIOFB_T structure specified by fp. See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rupfb()  

#include <stdio.h>  
#include <recio.h>  
#include <stdlib.h>

int main(void)
{
    _RFILE *fp;  
    _RIOFB_T *fb;
    /* Create a physical file */  
    system("CRTPF FILE(QTEMP/MY_FILE) RCDLEN(80)");
    /* Open the file for write */  
    fp = _Ropen("QTEMP/MY_FILE", "wr");
    if (fp == NULL)
    {
        printf("open for write fails\n");
        exit(1);
    }
    /* Write some records into the file */  
    _Rwrite(fp, "This is record 1", 16);
    _Rwrite(fp, "This is record 2", 16);
    _Rwrite(fp, "This is record 3", 16);
    _Rwrite(fp, "This is record 4", 16);
    _Rwrite(fp, "This is record 5", 16);
    _Rwrite(fp, "This is record 6", 16);
    _Rwrite(fp, "This is record 7", 16);
    _Rwrite(fp, "This is record 8", 16);
    _Rwrite(fp, "This is record 9", 16);
    /* Close the file */  
    _Rclose(fp);
    /* Open the file for read */  
    fp = _Ropen("QTEMP/MY_FILE", "rr, blkrcd = y");
    if (fp == NULL)
    {
        printf("open for read fails\n");
        exit(2);
    }
/* Read some records                                    */
_Rreadn(fp, NULL, 80, __DFT);
_Rreadn(fp, NULL, 80, __DFT);
/* Call _Rupfb and print feedback information          */
fb = _Rupfb(fp);
printf("record number -------------------------- %d
", fb->rrn);
printf("number of bytes read ------------------- %d
", fb->num_bytes);
printf("number of records remaining in block --- %hd
", fb->blk_count);
if ( fb->blk_filled_by == __READ_NEXT )
{
    printf("block filled by ------------------------ __READ_NEXT\n");
}
else
{
    printf("block filled by ------------------------ __READ_PREV\n");
}
/* Close the file                                     */
_Rclose(fp);

Related Information

• "_Ropnfbk() — Obtain Open Feedback Information" on page 275

_Rwrite() — Write the Next Record

Format
#include <recio.h>

_RIOFB_T * _Rwrite(_RFILE *fp, void *buf, size_t size);

Language Level: ILE C Extension

Threadsafe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rwrite() function has two modes: move and locate. When buf points to a user buffer, _Rwrite() is in move mode. When buf is NULL, the function is in locate mode.

The _Rwrite() function appends a record to the file specified by fp. The number of bytes copied from buf to the record is the minimum of size and the record length of the file (move mode only). If size is greater than the record length, the data is truncated and errno is set to ETRUNC. One complete record is always written if the operation is successful.

If you are using _Ropen() and then _Rwrite() to output records to a source physical file, the sequence numbers must be manually appended.

The _Rwrite() function has no effect on the position of the file for a subsequent read operation.

If record blocking is taking place and the file associated with fp is nearing the limit of the number of records it can contain, records may be lost although the _Rwrite() function indicates success. This can happen if another file pointer is being used to write records to the file and it fills the file before the records in the
block are written to the file. In this case, the _Rwrite() function will indicate an error has occurred only on the call to the _Rwrite() function that sends the data to the database.

The _Rwrite() function is valid for all types of files.

Return Value

The _Rwrite() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rwrite() operation is successful the num_bytes field is set to the number of bytes written for both move mode and locate mode. The function transfers the bytes from the user’s buffer to the system buffer. If record blocking is taking place, the function only updates the rri and key fields when it sends the block to the database. If fp is a display, ICF or printer file, the function updates the sysparm field. If it is unsuccessful, the num_bytes field is set to a value less than size specified (move mode) or zero (locate mode) and errno is changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rwrite()

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE   *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats   buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
```
printf ( "Could not open file\n" );
exit ( 1 );

_Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace */
/* with actual device name. */
_Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
/* display file. */
rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
_Rpgmdv ( fp,"DEVICE2" ); /* Change the default program device. */
/* Replace with actual device name. */
_Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
/* display file. */
rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
    sizeof(out_buf));
_Rreadindv ( fp, &buf, sizeof(buf), __DFT );
    /* Read from the first device that */
    /* enters data - device becomes */
    /* default program device. */

/* Determine which terminal responded first. */
iofb = _Riofbk ( fp);
if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
else
    {
        _Rrelease(fp, "DEVICE2" );
    }

    /* Continue processing. */
    printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp);

Related Information

- "_Rwrited() — Write a Record Directly"
- "_Rwriterd() — Write and Read a Record” on page 310
- "_Rwrread() — Write and Read a Record (separate buffers)” on page 311

_Rwrited() — Write a Record Directly

Format

#include <recio.h>

_RIOFB_T *Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrn);

Language Level: ILE C Extension

Threadsafe: Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rwrited() function writes a record to the file associated with fp at the position specified by rrn. The _Rwrited() function will only write over deleted records. The number of bytes copied from buf to the record is the minimum of size and the record length of the file (move mode only). If size is greater than the record length, the data is truncated, and errno is set to ETRUNC. One complete record is always written if the operation is successful.
The _Rwrited() function has no effect on the position of the file for a read operation.

The _Rwrited() function is valid for database, DDM and subfiles.

**Return Value**

The _Rwrited() function returns a pointer to the _RIOFB_T structure associated with `fp`. If the _Rwrited() operation is successful the num.bytes field is set to the number of bytes transferred from the user’s buffer to the system buffer (move mode) or the record length of the file (locate mode). The `rrn` field is updated. If `fp` is a display file, the sysparm field is updated. If it is unsuccessful, the num_bytes field is set to a value less than `size` specified (move mode) or zero (locate mode) and `errno` is changed.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for `errno` settings.

**Example that uses _Rwrited()**
```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define LEN          10
#define NUM_RECS     20
#define SUBFILENAME  "MYLIB/T1677RD6"
#define PFFILENAME   "MYLIB/T1677RDB"

typedef struct  {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{    
    _RFILE           *pf;
    _RFILE           *subf;
    /* Open the subfile and the physical file. */    
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL)  {
        printf("can't open file %s\n", PFILENAME);    
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)  {
        printf("can't open file %s\n", SUBFILENAME);    
        exit(2);
    }
    /* Initialize the subfile with records */        
    init_subfile(pf, subf);
    /* Write the subfile to the display by writing */
    /* a record to the subfile control format. */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, ",", 0);
    _Rreadnc(subf, ",", 0);
    /* Close the physical file and the subfile. */
    _Rclose(pf);
    _Rclose(subf);
}

void init_subfile(_RFILE *pf, _RFILE *subf)
{    
    _RIOFB_T      *fb;
    int           i;
    pf_t          record;
    /* Select the subfile record format. */        
    _Rformat(subf, "SFL*");
    for (i = 1; i <= NUM_RECS; i++)  {
        fb = _Rreadn(pf, &record, RECLEN, __DFT);
        if (fb->num_bytes != RECLEN)  {
            printf("%d\n", fb->num_bytes);
            printf("%d\n", RECLEN);
            printf("error occurred during read\n");    
            exit(3);
        }
        fb = _Rwrited(subf, &record, RECLEN, i);
        if (fb->num_bytes != RECLEN)  {
            printf("error occurred during write\n");    
            exit(4);
        }
    }
}

Related Information
- "_Rwrite() — Write the Next Record" on page 305
- "_Rwriterd() — Write and Read a Record" on page 310
- "_Rwrread() — Write and Read a Record (separate buffers)" on page 311
```
_Rwriterd() — Write and Read a Record

Format

```c
#include <recio.h>
_RIOFB_T *_Rwriterd(_RFILE *fp, void *buf, size_t size);
```

Language Level: ILE C Extension

Threatsafe: No.

Description

The _Rwriterd() function performs a write and then a read operation on the file that is specified by fp. The minimum of size and the length of the current record format determines the amount of data to be copied between the system buffer and buf for both the write and read parts of the operation. If size is greater than the record length of the current format, errno is set to ETRUNC on the write part of the operation. If size is less than the length of the current record format, errno is set to ETRUNC on the read part of the operation.

The _Rwriterd() function is valid for display and ICF files.

Return Value

The _Rwriterd() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rwriterd() operation is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to buf on the read part of the operation (move mode) or the record length of the file (locate mode).

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

Example that uses _Rwriterd()
```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */

    rfb = _Rwrite ( fp, ", 0 ); /* Set up the display. */
    rfb = _Rwrited ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf, sizeof(out_buf) );

    /* Continue processing. */

    _Rclose ( fp );
}
```

Related Information
- "_Rwrite() — Write the Next Record" on page 305
- "_Rwrited() — Write a Record Directly" on page 307
- "_Rwrread() — Write and Read a Record (separate buffers)"

_Rwrread() — Write and Read a Record (separate buffers)

Format
#include <recio.h>

_RIOFB_T * _Rwrread(_RFILE *fp, void *in_buf, size_t in_buf_size,
    void *out_buf, size_t out_buf_size);
**Language Level:** ILE C Extension

**Threadsafe:** No.

**Description**

The _Rwread() function performs a write and then a read operation on the file that is specified by *fp*. Separate buffers may be specified for the input and output data. The minimum of size and the length of the current record format determines the amount of data to be copied between the system buffer and the buffers for both the write and read parts of the operation. If *out_buf_size* is greater than the record length of the current format, errno is set to ETRUNC on the write part of the operation. If *in_buf_size* is less than the length of the current record format, errno is set to ETRUNC on the read part of the operation.

The _Rwread() function is valid for display and ICF files.

**Return Value**

The _Rwread() function returns a pointer to the _RIOFB_T structure that is associated with *fp*. If the _Rwread() operation is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to *in_buf* in the read part of the operation (move mode) or the record length of the file (locate mode).

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 12 on page 485 and Table 14 on page 489 for errno settings.

**Example that uses _Rwread()**
#include <stdio.h>
#include <reco.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
  char name[20];
  char address[25];
} format1;

typedef struct {
  char name[8];
  char password[10];
} format2;

typedef union {
  format1 fmt1;
  format2 fmt2;
} formats;

int main(void) {

  _RFILE   *fp; /* File pointer */
  _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
  formats  buf, in_buf, out_buf; /* Buffers to hold data */

  /* Open the device file. */
  if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL ) {
    printf ( "Could not open file\n" );
    exit ( 1 );
  }

  _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
  _Rformat ( fp,"FORMAT2" ); /* Set the record format for the display file. */

  rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
  rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
  rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf, sizeof(out_buf) );

  /* Continue processing. */

  _Rclose ( fp );
}

Related Information

- "_Rwrite() — Write the Next Record" on page 305
- "_Rwrited() — Write a Record Directly" on page 307
- "_Rwriterd() — Write and Read a Record" on page 310

scanf() — Read Data

Format
#include <stdio.h>
int scanf(const char *format-string, argument-list);

Language Level: ANSI
Threadsafe: Yes.

Description

The `scanf()` function reads data from the standard input stream stdin into the locations that is given by each entry in argument-list. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format-string controls the interpretation of the input fields, and is a multibyte character string that begins and ends in its initial shift state.

The format-string can contain one or more of the following:

- White-space characters, as specified by the `isspace()` function (such as blanks and new-line characters). A white-space character causes the `scanf()` function to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in format-string matches any combination of white-space characters in the input.

- Characters that are not white space, except for the percent sign character (%). A non-whitespace character causes the `scanf()` function to read, but not to store, a matching non-whitespace character. If the next character in stdin does not match, the `scanf()` function ends.

- Format specifications, introduced by the percent sign (%). A format specification causes the `scanf()` function to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The `scanf()` function reads format-string from left to right. Characters outside of format specifications are expected to match the sequence of characters in stdin; the matched characters in stdin are scanned but not stored. If a character in stdin conflicts with format-string, `scanf()` ends. The conflicting character is left in stdin as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first entry in argument-list. The second format specification converts the second input field and stores it in the second entry in argument-list, and so on through the end of format-string.

An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field width is reached, whichever comes first. If there are too many arguments for the format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

A format specification has the following form:

```
%<option> [width] type
```

Each field of the format specification is a single character or a number signifying a particular format option. The type character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a
string, or a number. The simplest format specification contains only the percent sign and a type character (for example, %s).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format control character, that character and following characters up to the next percent sign are treated as an ordinary sequence of characters; that is, a sequence of characters that must match the input. For example, to specify a percent-sign character, use %%. 

The following restrictions apply to pointer printing and scanning:
- If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer that is printed out.
- If a scanf() family function scans a pointer that was printed out by a different activation group, the scanf() family function will set the pointer to NULL.


An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The width is a positive decimal integer controlling the maximum number of characters to be read from stdin. No more than width characters are converted and stored at the corresponding argument. Fewer than width characters are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before width is reached.

The optional size modifiers h, l, ll, and L indicate the size of the receiving object. The conversion characters d, i, and n must be preceded by h if the corresponding argument is a pointer to short int rather than a pointer to int, by l if it is a pointer to long int, or by ll if it is a pointer to long long int. Similarly, the conversion characters o, u, x, and X must be preceded by h if the corresponding argument is a pointer to unsigned short int rather than a pointer to unsigned int, by l if it is a pointer to unsigned long int, or by ll if it is a pointer to unsigned long long int. The conversion characters e, E, f, g, and G must be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to a long double. Finally, the conversion characters c, s, and [ must be preceded by l if the corresponding argument is a pointer to wchar_t rather than a pointer to a single byte character type. If an h, l, L or ll appears with any other conversion character, the behavior is undefined.

The type characters and their meanings are in the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Type of Input Expected</th>
<th>Type of Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Signed decimal integer</td>
<td>Pointer to int.</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>x, X</td>
<td>Unsigned hexadecimal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>i</td>
<td>Decimal, hexadecimal, or octal integer</td>
<td>Pointer to int.</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>Character</td>
<td>Type of Input Expected</td>
<td>Type of Argument</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>e, f, g, E, G</td>
<td>Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (e or E) followed by a possibly signed integer value.</td>
<td>Pointer to float.</td>
</tr>
<tr>
<td>D(n,p)</td>
<td>Packed decimal value consisting of an optional sign (+ or -); then a non-empty sequence of digits, optionally a series of one or more decimal digits possibly containing a decimal point, but not a decimal suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-whitespace character, in the expected form. It contains no characters if the input string is empty or consists entirely of white space, or if the first non-whitespace character is anything other than a sign, a digit, or a decimal point character.</td>
<td>Pointer to decimal(n,p). Since the internal representation of the binary coded decimal object is the same as the internal representation of the packed decimal data type, you can use the type character D(n,p).</td>
</tr>
<tr>
<td>c</td>
<td>Character; white-space characters that are ordinarily skipped are read when c is specified</td>
<td>Pointer to char large enough for input field.</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Pointer to character array large enough for input field plus a ending null character (\0), which is automatically appended.</td>
</tr>
<tr>
<td>n</td>
<td>No input read from stream or buffer</td>
<td>Pointer to int, into which is stored the number of characters successfully read from the stream or buffer up to that point in the call to scanf().</td>
</tr>
<tr>
<td>p</td>
<td>Pointer to void converted to series of characters</td>
<td>Pointer to void.</td>
</tr>
<tr>
<td>lc</td>
<td>Multibyte character constant</td>
<td>Pointer to wchar_t.</td>
</tr>
<tr>
<td>ls</td>
<td>Multibyte string constant</td>
<td>Pointer to wchar_t string.</td>
</tr>
</tbody>
</table>

To read strings not delimited by space characters, substitute a set of characters in brackets ([ ]) for the s (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification %ac, where a is a decimal integer. In this instance, the c type character means that the argument is a pointer to a character array. The next a characters are read from the input stream into the specified location, and no null character is added.

The input for a %x format specifier is interpreted as a hexadecimal number.

The scanf() function scans each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified width is reached, or when the next character cannot be converted as
specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on stdin.

For %lc and %ls, specifies the data that is read is a multibyte string and is converted to wide characters as if by calls to mbtowc.

Alternative format specification has the following form:

As an alternative, specific entries in the argument-list may be assigned by using the format specification outlined in the diagram above. This format specification and the previous format specification may not be mixed in the same call to scanf(). Otherwise, unpredictable results may occur.

The arg-number is a positive integer constant where 1 refers to the first entry in the argument-list. Arg-number may not be greater than the number of entries in the argument-list, or else the results are undefined. Arg-number also may not be greater than NL_ARGMAX.

Return Value

The scanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Error Conditions

If the type of the argument that is to be assigned into is different than the format specification, unpredictable results can occur. For example, reading a floating point value, but assigning it into a variable of type int, is incorrect and would have unpredictable results.

If there are more arguments than format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

If the format string contains an invalid format specification, and positional format specifications are being used, errno will be set to EILSEQ.

If positional format specifications are used and there are not enough arguments, errno will be set to EINVAL.

If a conversion error occurs, errno may be set to ECONVERT.

Examples using scanf()
This example scans various types of data.

```c
#include <stdio.h>

int main(void)
{
    int i;
    float fp;
    char c, s[81];

    printf("Enter an integer, a real number, a character "
           "and a string : \n");
    if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
        printf("Not all fields were assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}

/****************** If input is: 12 2.5 a yes, ******************
************** then output should be similar to: **************

Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

This example converts a hexadecimal integer to a decimal integer. The while loop ends if the input value is not a hexadecimal integer.

```c
#include <stdio.h>

int main(void)
{
    int number;
    printf("Enter a hexadecimal number or anything else to quit:
");
    while (scanf("%x", &number))
    {
        printf("Hexadecimal Number = %x\n", number);
        printf("Decimal Number     = %d\n", number);
    }
}

/******************** If input is: 0x231 0xf5e 0x1 q,  ********************
************** then output should be similar to: **************

Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number     = 561
Hexadecimal Number = f5e
Decimal Number     = 3934
Hexadecimal Number = 1
Decimal Number     = 1
*/
```

This example reads from stdin and assigns data by using the alternative positional format string.
This example reads in a multibyte character string into a wide Unicode string. The example can be compiled with either LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF).

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    char s[20];
    float f;
    scanf("%2$s %3$f %1$d", &i, s, &f);
    printf("The data read was %i\n%s\n%f\n,i,s,f);
    return 0;
}
/* If the input is: test 0.2 100
   then the output will be similar to: */
   The data read was
   100
   test
   0.20000
```

This example reads in a multibyte character string into a wide Unicode string. The example can be compiled with either LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF).

```
#include <locale.h>
#include <stdio.h>
#include <wchar.h>
void main(void)
{
    wchar_t uString[20];
    setlocale(LC_UNI_ALL, "");
    scanf("Enter a string %ls", uString);
    printf("String read was %ls\n", uString);
}
/* if the input is: ABC
   then the output will be similar to:
   String read was ABC
*/
```

Related Information
- "fscanf() — Read Formatted Data" on page 123
- "printf() — Print Formatted Characters" on page 211
- "sscanf() — Read Data" on page 339
- "wscanf() — Read Data Using Wide-Character Format String" on page 482
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "swscanf() — Read Wide Character Data" on page 387
- "<stdio.h>" on page 14
#include <stdio.h>
void setbuf(FILE *, char *buffer);

Language Level: ANSI

Threadsafe: Yes.

Description

If the operating system supports user-defined buffers, setbuf() controls buffering for the specified stream. The setbuf() function only works in ILE C when using the integrated file system. The stream pointer must refer to an open file before any I/O or repositioning has been done.

If the buffer argument is NULL, the stream is unbuffered. If not, the buffer must point to a character array of length BUFSIZ, which is the buffer size that is defined in the <stdio.h> include file. The system uses the buffer, which you specify, for input/output buffering instead of the default system-allocated buffer for the given stream. stdout, stderr, and stdin do not support user-defined buffers.

The setvbuf() function is more flexible than the setbuf() function.

Return Value

There is no return value.

Example that uses setbuf()

This example opens the file setbuf.dat for writing. It then calls the setbuf() function to establish a buffer of length BUFSIZ. When string is written to the stream, the buffer buf is used and contains the string before it is flushed to the file.

```
#include <stdio.h>

int main(void)
{
    char buf[BUFSIZ];
    char string[] = "hello world";
    FILE *stream;

    memset(buf, '\0', BUFSIZ); /* initialize buf to null characters */
    stream = fopen("setbuf.dat", "wb");
    setbuf(stream, buf);       /* set up buffer */
    fwrite(string, sizeof(string), 1, stream);
    printf("%s\n", buf);       /* string is found in buf now */
    fclose(stream);           /* buffer is flushed out to myfile.dat */
}
```

Related Information

- "fclose() — Close Stream” on page 82
- "fflush() — Write Buffer to File” on page 87
- "fopen() — Open Files” on page 100
- "setvbuf() — Control Buffering” on page 328
setjmp() — Preserve Environment

Format
#include <setjmp.h>
int setjmp(jmp_buf env);

Language Level: ANSI

Threadsafe: Yes.

Description

The `setjmp()` function saves a stack environment that can subsequently be restored by the `longjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function causes it to save the current stack environment in `env`. A subsequent call to the `longjmp()` function restores the saved environment and returns control to a point corresponding to the `setjmp()` call. The values of all variables (except register variables) available to the function receiving control contain the values they had when the `longjmp()` function was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to the `setjmp()` function and the `longjmp()` function are also unpredictable.

Return Value

The `setjmp()` function returns the value 0 after saving the stack environment. If the `setjmp()` function returns as a result of a `longjmp()` call, it returns the `value` argument of the `longjmp()` function, or 1 if the `value` argument of the `longjmp()` function is 0. There is no error return value.

Example that uses `setjmp()`

This example stores the stack environment at the statement if(setjmp(mark) != 0) ...

When the system first performs the if statement, it saves the environment in `mark` and sets the condition to FALSE because `setjmp()` returns a 0 when it saves the environment. The program prints the message:

`setjmp has been called`

The subsequent call to function p tests for a local error condition, which can cause it to perform the `longjmp()` function. Then, control returns to the original function using the environment that is saved in `mark`. This time, the condition is TRUE because -1 is the return value from the `longjmp()` function. The program then performs the statements in the block and prints:

`setjmp has been called`

Then the program calls the `recovery()` function and exits.

```c
#include <stdio.h>
#include <setjmp.h>
jmp_buf mark;
```
void p(void);
void recover(void);

int main(void)
{
  if (setjmp(mark) != 0)
  {
    printf("longjmp has been called\n");
    recover();
  }
  printf("setjmp has been called\n");
  p();
  exit(1);
}

void p(void)
{
  longjmp(mark, -1);
}

void recover(void)
{
  exit(1);
}

Related Information

• "longjmp() — Restore Stack Environment" on page 178
• "<setjmp.h>" on page 13

setlocale() — Set Locale

Format
#include <locale.h>
char *setlocale(int category, const char *locale);

Language Level: ANSI

Threading safe: No.

Description

The setlocale() function changes or queries variables that are defined in the
<locale.h> include file, that indicate location. The values for category are listed
below.

<table>
<thead>
<tr>
<th>Category</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>Names entire locale of program.</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>Affects behavior of the strcoll() and strcmp() functions.</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>Affects behavior of character handling functions.</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>Affects monetary information returned by localeconv() and n1_langinfo().functions.</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>Affects the decimal-point character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by the localeconv() and n1_langinfo() functions.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>Affects behavior of the strftime() function and the time formatting information returned by the n1_langinfo() function.</td>
</tr>
<tr>
<td>Category</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>Affects the behavior of the time functions.</td>
</tr>
<tr>
<td></td>
<td>The category LC_TOD has several fields in it. The TNAME field is the</td>
</tr>
<tr>
<td></td>
<td>time zone name. The TZDIFF field is the difference between local time</td>
</tr>
<tr>
<td></td>
<td>and Greenwich Meridian time. If the TNAME field is nonblank, then the</td>
</tr>
<tr>
<td></td>
<td>TZDIFF field is used when determining the values that are returned by</td>
</tr>
<tr>
<td></td>
<td>some of the time functions. This value takes precedence over the system</td>
</tr>
<tr>
<td></td>
<td>value, QUTCOFFSET.</td>
</tr>
<tr>
<td>LC_UNI_ALL*</td>
<td>This category causes setlocale() to load all of the the LC_UNI_</td>
</tr>
<tr>
<td></td>
<td>categories from the locale specified. This category accepts only a</td>
</tr>
<tr>
<td></td>
<td>locale with a UCS2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_COLLATE*</td>
<td>Affects behavior of the wcscol1() and wcscxfrm() functions. This</td>
</tr>
<tr>
<td></td>
<td>category accepts only a locale with a UCS2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td></td>
<td>Note: This category is not supported for UCS2.</td>
</tr>
<tr>
<td>LC_UNI_CTYPE*</td>
<td>Affects the behavior of the wide character handling functions. This</td>
</tr>
<tr>
<td></td>
<td>category accepts only a locale with a UCS2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_MESSAGES*</td>
<td>Affects the message formatting information returned by the <em>WCS_nl</em></td>
</tr>
<tr>
<td></td>
<td>langinfo() function. This category accepts only a locale with a UCS2</td>
</tr>
<tr>
<td></td>
<td>or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_MONETARY*</td>
<td>Affects the monetary information returned by the wcslocaleconv() and</td>
</tr>
<tr>
<td></td>
<td>_WCS_nl_langinfo() functions. This category accepts only a locale with</td>
</tr>
<tr>
<td></td>
<td>a UCS2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_NUMERIC*</td>
<td>Affects the decimal-point character for the wide character formatted</td>
</tr>
<tr>
<td></td>
<td>input/output and wide character string conversion functions, and the</td>
</tr>
<tr>
<td></td>
<td>non-monetary information returned by the wcslocaleconv() and <em>WCS_nl</em></td>
</tr>
<tr>
<td></td>
<td>langinfo() functions. This category accepts only a locale with a UCS2</td>
</tr>
<tr>
<td></td>
<td>or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_TIME*</td>
<td>Affects the behavior of the wcsftime() function and the time</td>
</tr>
<tr>
<td></td>
<td>formatting information returned by the _WCS_nl_langinfo() functions.</td>
</tr>
<tr>
<td></td>
<td>This category accepts only a locale with a UCS2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_TOD*</td>
<td>Affects the behavior of the wide character time functions. This</td>
</tr>
<tr>
<td></td>
<td>category accepts only a locale with a UCS2 or UTF-32 CCSID.</td>
</tr>
</tbody>
</table>

* To use categories with UNI in the name, LOCALETYPE(*LOCALEUCS2) or LOCATEYPE(*LOCALEUTF) must be specified on the compilation command. If LOCALETYPE(*LOCALEUCS2) is used, the locale specified must be a UCS2 locale. If LOCALETYPE(*LOCALEUTF) is used, the locale specified must be a UTF-32 locale.

Note: There are two ways of defining setlocale() and other locale-sensitive C functions on the iSeries server. The original way to define setlocale() uses *CLD locale objects to set the locale and retrieve locale-sensitive data. The second way to define setlocale() uses *LOCALE objects to set the locale and retrieve locale-sensitive data. The original way is accessed by specifying LOCALETYPE(*CLD) on the compilation command. The second way is accessed by specifying LOCALETYPE(*LOCALE), LOCALETYPE(*LOCALEUCS2), or LOCALETYPE(*LOCALEUTF) on the compilation command. For more information on the two methods of locale definition in ILE C, see "International Locale Support" in the WebSphere Development Studio: ILE C/C++ Programmer’s Guide.
Setlocale using *CLD locale objects

You can set the value of `locale` to "C", "", LC_C, LC_C_GERMANY, LC_C_FRANCE, LC_C_SPAIN, LC_C_ITALY, LC_C_USA or LC_C_UK. A `locale` value of "C" indicates the default C environment. A `locale` value of "" tells the `setlocale()` function to use the default locale for the implementation.

Setlocale with *LOCALE objects.

You can set the value of `locale` to "", "C", "POSIX", or the fully qualified Integrated File System path name of a *LOCALE object enclosed in double quotes. A `locale` value of "C" or "POSIX" indicates the default C *LOCALE object. A `locale` value of "" tells the `setlocale()` function to use the default locale for the process.

The default locale for the process is determined using the following table:

<table>
<thead>
<tr>
<th>Locale Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LC_ALL</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale for all POSIX locale categories. Otherwise, go to the next step.&lt;br&gt;2. For each POSIX locale category (LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MESSAGES, LC_MONETARY, and LC_TOD), check the environment variable with the same name. If it is defined and not null, use the locale specified. 3. Check the LANG environment variable. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_CTYPE</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_COLLATE</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_TIME</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_NUMERIC</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_MESSAGES</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_MONETARY</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td><strong>LC_TOD</strong></td>
<td>1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step. 2. Check the environment variable with the same name as the specified locale category. If it is defined and not null, use the locale specified. Otherwise, go to the next step. 3. Check the LANG environment variable. If it is defined and not null, set the locale category to the specified locale. Otherwise, go to the next step. 4. Set the locale category to the default C *LOCALE object.</td>
</tr>
</tbody>
</table>
If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:

1. Check the LC_UCS2_ALL environment variable. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.

2. For each Unicode locale category check the corresponding environment variable (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD). If it is defined and not null, use the locale specified.

3. Set the locale category to the default UCS2 *LOCALE object.

If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:

1. Check the LC_UTF_ALL environment variable. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.

2. For each Unicode locale category check the corresponding environment variable (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD). If it is defined and not null, use the locale specified.

3. Check the LANG environment variable. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.
<table>
<thead>
<tr>
<th>LC_UNI_CTYPE</th>
<th>LC_UNI_COLLATE</th>
<th>LC_UNI_TIME</th>
<th>LC_UNI_NUMERIC</th>
<th>LC_UNI_MESSAGES</th>
<th>LC_UNI_MONETARY</th>
<th>LC_UNI_TOD</th>
</tr>
</thead>
</table>

If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:

1. Check the environment variable corresponding to the specified locale category (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD)³. If it is defined and not null, use the locale specified. Otherwise, go to the next step.

2. Check the LC_UCS2_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step.

3. Set the locale category to the default UCS2 *LOCALE object.

If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:

1. Check the environment variable corresponding to the specified locale category (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD)². If it is defined and not null, use the locale specified. Otherwise, go to the next step.

2. Check the LC_UTF_ALL environment variable. If it is defined and not null, use the specified locale. Otherwise, go to the next step.

3. Check the LANG environment variable. If the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.

**Note:**¹ The environment variables with names corresponding to locale categories are created by the user. The LANG environment variable is automatically created during job initiation when you specify a locale path name for either of the following:

- **locale parameter in your user profile** (see the CHGUSRPRF (Change User Profile) command information in the iSeries Information Center).
- **QLOCALE system value** (see the QLOCALE system value information in the iSeries Information Center).

The locale environment variables are expected to contain a locale path name of the form /QSYS.LIB/<locname>.LOCALE or /QSYS.LIB/<libname>.LIB/<locname>.LOCALE. If your module is compiled with the LOCALETYPE(*LOCALEUTF) option, the environment variable will be ignored if the <locname> portion of the path exceeds 8 characters. This restriction exists because a 2 character suffix must be appended to the locale name to get the name of the corresponding UTF locale.

**Note:**² When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the setlocale() function appends a trailing “_8” to the LC_ALL, LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MESSAGES, LC_MONETARY, LC_TOD, and LANG environment variables. If this locale is not found, the UTF default locale object is used. For example, setlocale(LC_ALL, "") when LANG is set to /QSYS.LIB/EN_US.LOCALE.
causes `setlocale()` to attempt to load the locale `/QSYS.LIB/EN_US_8.LOCALE`. If the LANG environment variable is used to set one of the Unicode locale categories (LC_UNI_ALL, LC_UNI_CTYPE, LC_UNI_COLLATE, LC_UNI_TIME, LC_UNI_NUMERIC, LC_UNI_MESSAGES, LC_UNI_MONETARY, or LC_UNI_TOD), `setlocale()` appends a trailing "4" to the locale name stored in the environment variable. This is an attempt to locate the corresponding UTF32 locale. If this locale is not found, the default UTF32 locale object is used. For example, `setlocale(LC_UNI_TIME, "")` when LANG is set to `/QSYS.LIB/EN_US.LOCALE` causes `setlocale()` to attempt to load the locale `/QSYS.LIB/EN_US_4.LOCALE`. Locale names ending in "4" and "8" follow a naming convention introduced by the CRTLOCALE CL command (see the CRTLOCALE (Create Locale) command information in the iSeries Information Center) for locales created with CCSID(*UTF).

**Note:** The LC_UNI_ALL, LC_UNI_COLLATE, LC_UNI_CTYPE, LC_UNI_TIME, LC_UNI_NUMERIC, LC_UNI_MESSAGES, LC_UNI_MONETARY, and LC_UNI_TOD locale category names are shared between UCS2 and UTF. The environment variables corresponding to these categories cannot be shared, so the names of the environment variables do not exactly match the locale category names. For UCS2 environment variable names, “UNI” is replaced with “UCS2” (for example, LC_UNI_ALL locale category becomes LC_UCS2_ALL environment variable). For UTF environment variable names, “UNI” is replaced with “UTF” (for example, LC_UNI_ALL locale category becomes LC_UTF_ALL environment variable).

If compiled with LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF), the locale must be a pointer to a valid Unicode locale for the categories starting with LC_UNI_, and must not be a Unicode locale for the other categories.

**Return Value**

The `setlocale()` function returns a pointer to a string that, if passed back to the `setlocale()` function, would restore the locale to the previous setting. This string should be copied by the user as it will be overwritten on subsequent calls to `setlocale()`.

**Note:** Because the string to which a successful call to `setlocale()` points may be overwritten by subsequent calls to the `setlocale()` function, you should copy the string if you plan to use it later. The exact format of the locale string is different between locale types of *CLD, *LOCALE, *LOCALEUCS2, and *LOCALEUTF.

To query the locale, give a NULL as the second parameter. For example, to query all the categories of your locale, enter the following statement:

```c
char *string = setlocale(LC_ALL, NULL);
```

**Error Conditions**

On error, the `setlocale()` function returns NULL, and the program’s locale is not changed.

**Example that uses *CLD locale objects**
This example sets the locale of the program to
LC_C_FRANCE *CLD and prints the string
that is associated with the locale. This example must be compiled with
the LOCALETYPE(*CLD) parameter on the compilation command.

#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
  string = setlocale(LC_ALL, LC_C_FRANCE);
  if (string != NULL)
    printf(" %s 
",string);
}

Example that uses *LOCALE objects

#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
  string = setlocale(LC_ALL, "POSIX");
  if (string != NULL)
    printf(" %s 
",string);
}

Related Information
• "getenv() — Search for Environment Variables" on page 144
• "localeconv() — Retrieve Information from the Environment" on page 168
• "nl_langinfo() — Retrieve Locale Information" on page 206
• "<locale.h>" on page 7

setvbuf() — Control Buffering

Format
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);

Language Level: ANSI

Threadsafe: Yes.
Description

The `setvbuf()` function allows control over the buffering strategy and buffer size for a specified stream. The `setvbuf()` function only works in ILE C when using the integrated file system. The stream must refer to a file that has been opened, but not read or written to.

The array pointed to by `buf` designates an area that you provide that the C library may choose to use as a buffer for the stream. A `buf` value of NULL indicates that no such area is supplied and that the C library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

The `type` must be one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_IONBF</td>
<td>No buffer is used.</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>Full buffering is used for input and output. Use <code>buf</code> as the buffer and <code>size</code> as the size of the buffer.</td>
</tr>
<tr>
<td>_IOLBF</td>
<td>Line buffering is used. The buffer is deleted when a new-line character is written, when the buffer is full, or when input is requested.</td>
</tr>
</tbody>
</table>

If `type` is _IOFBF or _IOLBF, `size` is the size of the supplied buffer. If `buf` is NULL, the C library takes `size` as the suggested size for its own buffer. If `type` is _IONBF, both `buf` and `size` are ignored.

The value for `size` must be greater than 0.

Return Value

The `setvbuf()` function returns 0 if successful. It returns nonzero if a value that is not valid was specified in the parameter list, or if the request cannot be performed.

The `setvbuf()` function has no effect on `stdout`, `stdin`, or `stderr`.

Warning: The array that is used as the buffer must still exist when the specified `stream` is closed. For example, if the buffer is declared within the scope of a function block, the `stream` must be closed before the function is ended and frees the storage allocated to the buffer.

Example that uses `setvbuf()`

This example sets up a buffer of `buf` for `stream1` and specifies that input to `stream2` is to be unbuffered.
#include <stdio.h>

#define BUF_SIZE 1024

char buf[BUF_SIZE];
FILE *stream1, *stream2;

int main(void)
{
    stream1 = fopen("myfile1.dat", "r");
    stream2 = fopen("myfile2.dat", "r");

    /* stream1 uses a user-assigned buffer of BUF SIZE bytes */
    if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
        printf("Incorrect type or size of buffer
");

    /* stream2 is unbuffered */
    if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
        printf("Incorrect type or size of buffer
");

    /* This is a program fragment and not a complete function example */

    Related Information
    • "fclose() — Close Stream” on page 82
    • “fflush() — Write Buffer to File” on page 87
    • “fopen() — Open Files” on page 100
    • “setbuf() — Control Buffering” on page 319
    • "<stdio.h>” on page 14

signal() — Handle Interrupt Signals

Format
#include <signal.h>
void ( *signal (int sig, void(*func)(int)) )(int);

Language Level: ANSI

Threadsafe: Yes.

Description

The signal() function allows a program to choose one of several ways to handle an interrupt signal from the operating system or from the raise() function. If compiled with the SYSIFCOPT("ASYNCSIGNAL") option, this function uses asynchronous signals. The asynchronous version of this function behaves like sigaction() with SA_NODEFER and SA_RESETHAND options. Asynchronous signal handlers may not call abort() or exit(). The remainder of this function description will describe synchronous signals.

The sig argument must be one of the macros SIGABRT, SIGALL, SIGILL, SIGINT, SIGFPE, SIGIO, SIGOTHER, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2, defined in the signal.h include file. SIGALL, SIGIO, and SIGOTHER are only supported by the ILE C/C++ Run-time library. The func argument must be one of the macros SIG_DFL or SIG_IGN, defined in the <signal.h> include file, or a function address.

The meaning of the values of sig is as follows:
Value  Meaning
SIGABRT  Abnormal termination
SIGALL  Catch-all for signals whose current handling action is SIG_DFL.
          When SYSIFCOPT(*ASYNC SIGNAL) is specified, SIGALL is not a catch-all
          signal. A signal handler for SIGALL is only invoked for a user-raised
          SIGALL signal.
SIGILL  Detection of a function image that was not valid
SIGFPE  Arithmetic exceptions that are not masked, such as overflow, division by
          zero, and operations that are not valid
SIGINT  Interactive attention
SIGIO   Record file I/O error
SIGOTHER  ILE C signal
SIGSEGV  Access to memory that was not valid
SIGTERM  End request sent to the program
SIGUSR1  Intended for use by user applications. (extension to ANSI)
SIGUSR2  Intended for use by user applications. (extension to ANSI)
The action that is taken when the interrupt signal is received depends on the value
of func.
Value  Meaning
SIG_DFL  Default handling for the signal will occur.
SIG_IGN  The signal is to be ignored.
Return Value
A return value of SIG_ERR indicates an error in the call to signal(). If successful,
the call to signal() returns the most recent value of func. The value of errno may
be set to EINVAL (the signal is not valid).
Example that uses signal()
This example shows you how to establish a signal handler.
```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#define ONE_K 1024
#define OUT_OF_STORAGE (SIGUSR1)
    /* The SIGNAL macro does a signal() checking the return code */
#define SIGNAL(SIG, StrCln)       
    { 
      if (signal((SIG), (StrCln)) == SIG_ERR) { 
        perror("Could not signal user signal"); 
        abort(); 
      } 
    }

void StrCln(int);
void DoWork(char **, int);

int main(int argc, char *argv[]) { 
  int size;
  char *buffer;
  SIGNAL(OUT_OF_STORAGE, StrCln);
  if (argc != 2) { 
    printf("Syntax: %s size \n", argv[0]); 
    return(-1); 
  } 
  size = atoi(argv[1]); 
  DoWork(&buffer, size); 
  return(0); 
}

void StrCln(int SIG_TYPE) { 
  printf("Failed trying to malloc storage\n"); 
  SIGNAL(SIG_TYPE, SIG_DFL); 
  exit(0); 
}

void DoWork(char **buffer, int size) { 
  int rc; 
  *buffer = malloc(size*ONE_K);    /* get the size in number of K */
  if (*buffer == NULL) { 
    if (raise(OUT_OF_STORAGE)) { 
      perror("Could not raise user signal"); 
      abort(); 
    } 
  } 
  return; 
  /* This is a program fragment and not a complete function example */
}
```

Related Information

- "abort() — Stop a Program” on page 38
- "atexit() — Record Program Ending Function” on page 48
- "exit() — End Program” on page 80
- "raise() — Send Signal” on page 238
- "<signal.h>” on page 13
- signal() API in the APIs topic in the iSeries Information Center.

sin() — Calculate Sine

Format
#include <math.h>
double sin(double x);

Language Level: ANSI

Threadsafe: Yes.

Description

The `sin()` function calculates the sine of `x`, with `x` expressed in radians. If `x` is too large, a partial loss of significance in the result may occur.

Return Value

The `sin()` function returns the value of the sine of `x`. The value of `errno` may be set to either `EDOM` or `ERANGE`.

Example that uses `sin()`

This example computes `y` as the sine of `π/2`.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x, y;

    pi = 3.1415926535;
    x = pi/2;
    y = sin(x);

    printf("sin( %lf ) = %lf\n", x, y);
}
```

/******************** Output should be similar to: *************

sin( 1.570796 ) = 1.000000

*/

Related Information

- “acos() — Calculate Arccosine” on page 40
- “asin() — Calculate Arcsine” on page 44
- “atan() – atan2() — Calculate Arctangent” on page 47
- “cos() — Calculate Cosine” on page 66
- “cosh() — Calculate Hyperbolic Cosine” on page 67
- “sinh() — Calculate Hyperbolic Sine”
- “tan() — Calculate Tangent” on page 389
- “tanh() — Calculate Hyperbolic Tangent” on page 390
- “<math.h>” on page 7

**sinh() — Calculate Hyperbolic Sine**

Format

```c
#include <math.h>
double sinh(double x);
```

Language Level: ANSI
The sinh() function calculates the hyperbolic sine of \( x \), with \( x \) expressed in radians.

### Return Value

The sinh() function returns the value of the hyperbolic sine of \( x \). If the result is too large, the sinh() function sets errno to ERANGE and returns the value HUGE_VAL (positive or negative, depending on the value of \( x \)).

### Example that uses sinh()

This example computes \( y \) as the hyperbolic sine of \( \pi/2 \).

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x, y;
    pi = 3.1415926535;
    x = pi/2;
    y = sinh(x);
    printf("sinh( %lf ) = %lf\n", x, y);
}
```

Output should be similar to:

```
sinh( 1.570796 ) = 2.301299
```

### Related Information

- "acos() — Calculate Arccosine” on page 40
- "asin() — Calculate Arcsine” on page 44
- "atan() – atan2() — Calculate Arctangent” on page 47
- "cos() — Calculate Cosine” on page 66
- "cosh() — Calculate Hyperbolic Cosine” on page 67
- "sin() — Calculate Sine” on page 332
- "tan() — Calculate Tangent” on page 389
- "tanh() — Calculate Hyperbolic Tangent” on page 390
- “<math.h>” on page 7

### snprintf() — Print Formatted Data to Buffer

#### Format

```c
#include <stdio.h>
int snprintf(char *buffer, size_t n, const char *format-string, argument-list);
```

#### Language Level: ANSI

### Threadsafe: Yes.
Description

The `snprintf()` function formats and stores a series of characters and values in the array `buffer`. Any argument-list is converted and put out according to the corresponding format specification in the `format-string`. The `snprintf()` function is identical to the `sprintf()` function with the addition of the `n` argument, which indicates the maximum number of characters (including the ending null character) to be written to `buffer`.

The `format-string` consists of ordinary characters and has the same form and function as the format string for the `printf()` function.

Return Value

The `snprintf()` function returns the number of bytes that are written in the array, not counting the ending null character.

Example that uses `snprintf()`

This example uses `snprintf()` to format and print various data.

```c
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
    c = 'l';
    i = 35;
    fp = 1.7320508;

    /* Format and print various data */
    j = snprintf(buffer, 6, "%s
", s);
    j += snprintf(buffer+j, 6, "%c
", c);
    j += snprintf(buffer+j, 6, "%d
", i);
    j += snprintf(buffer+j, 6, "%f
", fp);
    printf("string: \n%s\ncharacter count = %d\n", buffer, j);
}
```

/******************** Output should be similar to:  *************

string:
balti
35
1.732
character count = 15  */

Related Information

- “fprintf() — Write Formatted Data to a Stream” on page 108
- “printf() — Print Formatted Characters” on page 211
- “sprintf() — Print Formatted Data to Buffer”
- “vsnprintf() — Print Argument Data to Buffer” on page 414
- “<stdio.h>” on page 14
Language Level: ANSI

Threadsafe: Yes.

Description

The `sprintf()` function formats and stores a series of characters and values in the array `buffer`. Any `argument-list` is converted and put out according to the corresponding format specification in the `format-string`.

The `format-string` consists of ordinary characters and has the same form and function as the `format-string` argument for the `printf()` function.

Return Value

The `sprintf()` function returns the number of bytes that are written in the array, not counting the ending null character.

Example that uses `sprintf()`

This example uses `sprintf()` to format and print various data.

```c
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
    c = 'l';
    i = 35;
    fp = 1.7320508;

    /* Format and print various data */
    j = sprintf(buffer, "%s\n", s);
    j += sprintf(buffer+j, "%c\n", c);
    j += sprintf(buffer+j, "%d\n", i);
    j += sprintf(buffer+j, "%f\n", fp);
    printf("string: \n%s\ncharacter count = %d\n", buffer, j);
}

/*@*********************  Output should be similar to:  *************
string: 
baltimore
1
35
1.732051
character count = 24           */
```

Related Information

- "fprintf() — Write Formatted Data to a Stream” on page 108
- “printf() — Print Formatted Characters” on page 211
- “sscanf() — Read Data” on page 339
sqrt() — Calculate Square Root

Format

```c
#include <math.h>
double sqrt(double x);
```

Language Level: ANSI

Threading: Yes.

Description

The `sqrt()` function calculates the nonnegative value of the square root of `x`.

Return Value

The `sqrt()` function returns the square root result. If `x` is negative, the function sets `errno` to `EDOM`, and returns 0.

Example that uses `sqrt()`

This example computes the square root of the quantity that is passed as the first argument to `main`. It prints an error message if you pass a negative value.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char ** argv)
{
    char * rest; 
    double value; 

    if ( argc != 2 )
        printf( "Usage: %s value\n", argv[0] );
    else 
    { 
        value = strtod( argv[1], &rest);
        if ( value < 0.0 )
            printf("sqrt of a negative number\n");
        else
            printf("sqrt( %lf ) = %lf\n", value, sqrt( value ));
    } 
}
```

/* *******************  If the input is 45,  **********************************
************** then the output should be similar to: **************

sqrt( 45.000000 ) = 6.708204
*/

Related Information

- “exp() — Calculate Exponential Function” on page 81
srand() — Set Seed for rand() Function

Format
#include <stdlib.h>
void srand(unsigned int seed);

Language Level: ANSI

Threadsafe: No.

Description

The srand() function sets the starting point for producing a series of
pseudo-random integers. If srand() is not called, the rand() seed is set as if
srand(1) were called at program start. Any other value for seed sets the generator to
a different starting point.

The rand() function generates the pseudo-random numbers.

Return Value

There is no return value.

Example that uses srand()

This example first calls srand() with a value other than 1 to initiate the random
value sequence. Then the program computes five random values for the array of
integers that are called ranvals.

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i, ranvals[5];

    srand(17);
    for (i = 0; i < 5; i++)
    {
        ranvals[i] = rand();
        printf("Iteration %d ranvals [ %d ] = %d\n", i+1, i, ranvals[i]);
    }

} /****************  Output should be similar to: ***************

Iteration 1 ranvals [0] = 24107
Iteration 2 ranvals [1] = 16552
Iteration 3 ranvals [2] = 12125
Iteration 4 ranvals [3] = 9427
Iteration 5 ranvals [4] = 13152
*/
sscanf() — Read Data

Format
#include <stdio.h>
int sscanf(const char *buffer, const char *format, argument-list);

Language Level: ANSI

Threadsafe: Yes.

Description

The sscanf() function reads data from buffer into the locations that are given by argument-list. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in the format-string.

Return Value

The sscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF when the end of the string is encountered before anything is converted.

Example that uses sscanf()

This example uses sscanf() to read various data from the string tokenstring, and then displays that data.

```c
#include <stdio.h>
#include <stddef.h>

int main(void)
{
    char    *tokenstring = "15 12 14";
    char    *string = "ABC Z";
    wchar_t  ws[81];
    wchar_t  wc;
    int   i;
    float fp;
    char  s[81];
    char  c;

    /* Input various data */
    /* In the first invocation of sscanf, the format string is */
    /* "%s %c%d%f". If there were no space between %s and %c, */
    /* sscanf would read the first character following the */
    /* string, which is a blank space. */
    sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    sscanf(string, "%ls %lc", ws,&wc);

    /* Display the data */
    printf("\nstring = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);

    return 0;
}
```

Chapter 2. Library Functions  339
printf("floating-point number = \%f\n", fp);
printf("wide-character string = \%S\n", ws);
printf("wide-character = \%C\n", wc);
}

/********************  Output should be similar to:  *******************/

string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z

/********************  Related Information  *******************/

* "fscanf() — Read Formatted Data" on page 123
* "scanf() — Read Data" on page 313
* "swscanf() — Read Wide Character Data" on page 387
* "fwscanf() — Read Data from Stream Using Wide Character" on page 137
* "wscanf() — Read Data Using Wide-Character Format String" on page 482
* "sprintf() — Print Formatted Data to Buffer" on page 335
* "<stdio.h>" on page 14

str case cmp() — Compare Strings without Case Sensitivity

Format
#include <strings.h>
int strcasecmp(const char *string1, const char *string2);

Language Level: XPG4

Threading: Yes.

Description
The strcasecmp() function compares string1 and string2 without sensitivity to case. All alphabetic characters in string1 and string2 are converted to lowercase before comparison.

The strcasecmp() function operates on null terminated strings. The string arguments to the function are expected to contain a null character ('\0') marking the end of the string.

Return Value
The strcasecmp() function returns a value indicating the relationship between the two strings, as follows:

Table 6. Return values of strcasecmp()

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 equivalent to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>
Example that uses `strcasecmp()`

This example uses `strcasecmp()` to compare two strings.

```c
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING";
    char_t *str2 = "string";
    int result;

    result = strcasecmp(str1, str2);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf(""%s" is less than %s\n", str1, str2);
    else
        printf(""%s" is greater than %s\n", str1, str2);

    return 0;
}

/******* The output should be similar to: **************/
Strings compared equal.

/********************/
The `strcat()` function operates on null-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string. No length checking is performed. You should not use a literal string for a `string1` value, although `string2` may be a literal string.

If the storage of `string1` overlaps the storage of `string2`, the behavior is undefined.

**Return Value**

The `strcat()` function returns a pointer to the concatenated string (`string1`).

**Example that uses `strcat()`**

This example creates the string "computer program" using `strcat()`.

```c
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer";
    char * ptr;

    ptr = strcat( buffer1, " program" );
    printf( "buffer1 = %s\n", buffer1 );
}

/*****************  Output should be similar to:  *****************/
buffer1 = computer program
*/
```

**Related Information**

- "`strchr()` — Search for Character" on page 342
- "`strcmp()` — Compare Strings" on page 343
- "`strcpy()` — Copy Strings" on page 347
- "`strcspn()` — Find Offset of First Character Match" on page 348
- "`strcat()` — Concatenate Strings" on page 361
- "`wcsstr()` — Search for Character" on page 430
- "`wcsstr()` — Concatenate Wide-Character Strings" on page 442
- "`<string.h>`" on page 16

**strchr() — Search for Character**

**Format**

```c
#include <string.h>
char *strchr(const char *string, int c);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**
The `strchr()` function finds the first occurrence of a character in a string. The character `c` can be the null character (`\0`); the ending null character of `string` is included in the search.

The `strchr()` function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) that marks the end of the string.

**Return Value**

The `strchr()` function returns a pointer to the first occurrence of `c` that is converted to a character in `string`. The function returns NULL if the specified character is not found.

**Example that uses `strchr()`**

This example finds the first occurrence of the character "p" in "computer program".

```c
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer program";
    char * ptr;
    int    ch = 'p';
    ptr = strchr( buffer1, ch );
    printf( "The first occurrence of %c in '%s' is '%s'
            ch, buffer1, ptr );
}
```

/******************** Output should be similar to: ********************/

The first occurrence of p in 'computer program' is 'puter program'
*/

**Related Information**

- "`strcat()` — Concatenate Strings" on page 341
- "`strcmp()` — Compare Strings"
- "`strcpy()` — Copy Strings" on page 347
- "`strcspn()` — Find Offset of First Character Match” on page 348
- "`strncmp()` — Compare Strings” on page 362
- "`strrchr()` — Find Characters in String” on page 367
- "`strchr()` — Locate Last Occurrence of Character in String” on page 373
- "`strspn()` — Find Offset of First Non-matching Character” on page 374
- "`wcschr()` — Search for Wide Character” on page 431
- "`wcsspn()` — Find Offset of First Non-matching Wide Character” on page 453
- "`<string.h>`” on page 16

**`strcmp()` — Compare Strings**

**Format**

#include <string.h>
int strcmp(const char *string1, const char *string2);

Language Level: ANSI

Threatsafe: Yes.

Description

The strcmp() function compares string1 and string2. The function operates on null-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string.

Return Value

The strcmp() function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 identical to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>

Example that uses strcmp()

This example compares the two strings that are passed to main() using strcmp().

#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int result;

    if ( argc != 3 )
    {
        printf( "Usage: %s string1 string2\n", argv[0] );
    }
    else
    {
        result = strcmp( argv[1], argv[2] );

        if ( result == 0 )
            printf( "%s" is identical to "%s", argv[1], argv[2] );
        else if ( result < 0 )
            printf( "%s" is less than "%s", argv[1], argv[2] );
        else
            printf( "%s" is greater than "%s", argv[1], argv[2] );
    }
}

/******* If the input is the strings *******
********** "is this first?" and "is this before that one?", **********
************** then the expected output is:  *******

"is this first?" is greater than "is this before that one?"  
***************************************************************************/

Related Information
**strcmpi() - Compare Strings Without Case Sensitivity**

**Format**
```
#include <string.h>
int strcmpi(const char *string1, const char *string2);
```

**Note:** The `strcmpi` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

**Description**

The `strcmpi` function compares `string1` and `string2` without sensitivity to case. All alphabetic characters in the two arguments `string1` and `string2` are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

**Return Value**

`strcmpi` returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

**Example that uses strcmpi()**

This example uses `strcmpi` to compare two strings.
```
#include <stdio.h>
#include <string.h>
int main(void)
{
    /* Compare two strings without regard to case */
```
```c
if (0 == strcmpi("hello", "HELLO"))
    printf("The strings are equivalent.\n");
else
    printf("The strings are not equivalent.\n");
return 0;
```

The output should be:
The strings are equivalent.

Related Information:
- "strcoll() — Compare Strings" on page 346
- "strcspn() — Find Offset of First Character Match” on page 348
- "strdup - Duplicate String” on page 350
- "strcmp() - Compare Strings without Case Sensitivity” on page 357
- "strncmp() — Compare Strings” on page 362
- "strncmp - Compare Substrings Without Case Sensitivity” on page 365
- "wcscmp() — Compare Wide-Character Strings” on page 433
- "wcsncmp() — Compare Wide-Character Strings” on page 443
- "strcasecmp() — Compare Strings without Case Sensitivity” on page 340
- "strncasecmp() — Compare Strings without Case Sensitivity” on page 359
- "<string.h>” on page 16

### strcoll() — Compare Strings

#### Format
```
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

#### Language Level: ANSI

#### Threadsafe: Yes.

#### Description

The `strcoll()` function compares two strings using the collating sequence that is specified by the program’s locale.

The behavior of this function is affected by the LC_COLLATE category of the current locale.

#### Return Value

The `strcoll()` function returns a value indicating the relationship between the strings, as listed below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>
If `strcoll()` is unsuccessful, errno is changed. The value of errno may be set to EINVAL (the `string1` or `string2` arguments contain characters that are not available in the current locale).

**Example that uses `strcoll()`**

This example compares the two strings that are passed to `main()` using `strcoll()`:

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int result;

    if ( argc != 3 )
    {
        printf( "Usage: %s string1 string2\n", argv[0] );
    }
    else
    {
        result = strcoll( argv[1], argv[2] );

        if ( result == 0 )
            printf( "%s" is identical to "%s"\n", argv[1], argv[2] );
        else if ( result < 0 )
            printf( "%s" is less than "%s"\n", argv[1], argv[2] );
        else
            printf( "%s" is greater than "%s"\n", argv[1], argv[2] );
    }
}
```

/*

************** If the input is the strings **************
************** "firststring" and "secondstring", **************
************** then the expected output is: **************

"firststring" is less than "secondstring"
*/

**Related Information**

- "setlocale() — Set Locale" on page 322
- "strcmp() — Compare Strings" on page 343
- "strnmcp() — Compare Strings" on page 362
- "wcsccoll() — Language Collation String Comparison" on page 434
- "<string.h>" on page 16

**strcpy() — Copy Strings**

**Format**

```c
#include <string.h>
char *strcpy(char *string1, const char *string2);
```

**Language Level:** ANSI

**Threaadsafe:** Yes.

**Description**
The strcpy() function copies string2, including the ending null character, to the location that is specified by string1.

The strcpy() function operates on null-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string. No length checking is performed. You should not use a literal string for a string1 value, although string2 may be a literal string.

Return Value

The strcpy() function returns a pointer to the copied string (string1).

Example that uses strcpy()

This example copies the contents of source to destination.

```c
#include <stdio.h>
#include <string.h>
#define SIZE 40

int main(void)
{
    char source[SIZE] = "This is the source string";
    char destination[SIZE] = "And this is the destination string";
    char * return_string;

    printf( "destination is originally = \"%s\n\n", destination );
    return_string = strcpy( destination, source );
    printf( "After strcpy, destination becomes \"%s\n\n", destination );
}

/*****************  Output should be similar to:  *****************

destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
*/
```

Related Information

- "strcat() — Concatenate Strings" on page 341
- "strchr() — Search for Character” on page 342
- "strcmp() — Compare Strings” on page 343
- "strcspn() — Find Offset of First Character Match”
- "strncpy() — Copy Strings” on page 364
- "strpbrk() — Find Characters in String” on page 367
- "strstr() — Locate Last Occurrence of Character in String” on page 373
- "strspn() — Find Offset of First Non-matching Character” on page 374
- "wcsncpy() — Copy Wide-Character Strings” on page 445
- "<string.h>” on page 16

strcspn() — Find Offset of First Character Match

Format
#include <string.h>

size_t strcspn(const char *string1, const char *string2);

Language Level: ANSI

Threadsafe: Yes.

Description

The `strcspn()` function finds the first occurrence of a character in `string1` that belongs to the set of characters that is specified by `string2`. Null characters are not considered in the search.

The `strcspn()` function operates on null-ended strings. The string arguments to the function should contain a null character (\0) marking the end of the string.

Return Value

The `strcspn()` function returns the index of the first character found. This value is equivalent to the length of the initial substring of `string1` that consists entirely of characters not in `string2`.

Example that uses `strcspn()`

This example uses `strcspn()` to find the first occurrence of any of the characters "a", "x", "l", or "e" in `string`.

```c
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char string[SIZE] = "This is the source string";
    char *substring = "axle";

    printf("The first %i characters in the string "string"
    "are not in the string "substring",
    strcspn(string, substring), string, substring);
}
```

/********** Output should be similar to: ***********/

```
The first 10 characters in the string "This is the source string"
are not in the string "axle"
```

Related Information

- "strcat() — Concatenate Strings" on page 341
- "strchr() — Search for Character" on page 342
- "strcmp() — Compare Strings" on page 343
- "strcpy() — Copy Strings" on page 347
- "strncpy() — Compare Strings" on page 362
- "strpbrk() — Find Characters in String" on page 367
- "strrchr() — Locate Last Occurrence of Character in String" on page 373
- "strspn() — Find Offset of First Non-matching Character" on page 374
strdup - Duplicate String

Format
#include <string.h>
char *strdup(const char *string);

Note: The strdup function is available for C++ programs. It is available for C only when the program defines the __cplusplus__strings__ macro.

Language Level: XPG4, Extension

Threatsafe: Yes.

Description

strdup reserves storage space for a copy of string by calling malloc. The string argument to this function is expected to contain a null character (\0) marking the end of the string. Remember to free the storage reserved with the call to strdup.

Return Value

strdup returns a pointer to the storage space containing the copied string. If it cannot reserve storage strdup returns NULL.

Example that uses strdup()

This example uses strdup to duplicate a string and print the copy.
#include <stdio.h>
#include <string.h>
int main(void)
{
  char *string = "this is a copy";
  char *newstr;
  /* Make newstr point to a duplicate of string */
  if ((newstr = strdup(string)) != NULL)
    printf("The new string is: %s\n", newstr);
  return 0;
}

The output should be:
The new string is: this is a copy

Related Information:
- "strcpy() — Copy Strings" on page 347
- "strncpy() — Copy Strings" on page 364
- "wcsncpy() — Copy Wide-Character Strings" on page 445
- "wcscspn() — Find Offset of First Wide-Character Match" on page 436
- "<string.h>" on page 16

strerror() — Set Pointer to Run-Time Error Message

Format
#include <string.h>
char *strerror(int errnum);

Language Level: ANSI

Threadsafe: Yes.

Description

The `strerror()` function maps the error number in `errnum` to an error message string.

Return Value

The `strerror()` function returns a pointer to the string. It does not return a NULL value. The value of `errno` may be set to `ECONVERT` (conversion error).

Example that uses `strerror()`

This example opens a file and prints a run-time error message if an error occurs.

```c
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(void)
{
    FILE *stream;

    if ((stream = fopen("mylib/myfile", "r")) == NULL)
        printf(" %s \n", strerror(errno));
}
/* This is a program fragment and not a complete function example */
```

Related Information

- "`clearerr()` — Reset Error Indicators" on page 64
- "`ferror()` — Test for Read/Write Errors" on page 87
- "`perror()` — Print Error Message" on page 209
- "`<string.h>`" on page 16

---

`strfmon()` — Convert Monetary Value to String

Format

```c
#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, argument_list);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `strfmon()` function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. No more than `maxsize` characters are placed into the array.
The character string _format_ contains two types of objects: plain characters, which are copied to the output stream, and directives, each of which results in the fetching of zero or more arguments, which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored. Only 15 significant digits are guaranteed on conversions involving double values.

A directive consists of a % character, optional conversion specifications, and a ending character that determines the directive’s behavior.

A directive consists of the following sequence:

- A % character.
- Optional flags.
- Optional field width.
- Optional left precision.
- Optional right precision.
- A required conversion character indicating the type of conversion to be performed.

### Table 7. Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=f</td>
<td>An = followed by a single character f which is used as the numeric fill character. By default the numeric fill character is a space character. This flag does not affect field width filling, which always uses a space character. This flag is ignored unless left precision is specified.</td>
</tr>
<tr>
<td>^</td>
<td>Do not use grouping characters when formatting the currency value. Default is to insert grouping characters as defined in the current locale.</td>
</tr>
<tr>
<td>+ or (</td>
<td>Specify the style representing positive and negative currency amounts. If + is specified, the locale’s equivalent of + and – for monetary quantities will be used. If ( is specified, negative amounts are enclosed within parenthesis. Default is +.</td>
</tr>
<tr>
<td>!</td>
<td>Do not output the currency symbol. Default is to output the currency symbol.</td>
</tr>
<tr>
<td>-</td>
<td>Use left justification for double arguments. Default is right justification.</td>
</tr>
</tbody>
</table>

### Field Width

_w_ A decimal digit string _w_ specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag - is specified). The default is 0.

### Left Precision

_n_ A # followed by a decimal digit string _n_ specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to _strfmon()_ aligned in the same columns. It can also be used to fill unused positions with a special character as in $***123.45. This option causes an
amount to be formatted as if it has the number of digits specified by \( n \). If more than \( n \) digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the \( =f \) flag above).

If grouping has not been suppressed with the \(^f\) flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit. To ensure alignment, any characters appearing before or after the number in the formatted output, such as currency or sign symbols, are padded as necessary with space characters to make their positive and negative formats an equal length.

### Right Precision

\( .p \) A period followed by a decimal digit string \( p \) specifies the number of digits after the radix character. If the value of the right precision \( p \) is 0, no radix character appears. If a right precision is not specified, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

### Table 8. Conversion Characters

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>The double argument is formatted according to the locale’s international currency format.</td>
</tr>
<tr>
<td>%n</td>
<td>The double argument is formatted according to the locale’s national currency format.</td>
</tr>
<tr>
<td>%%</td>
<td>Is replaced by %. No argument is converted.</td>
</tr>
</tbody>
</table>

**Note:** This function is not available when LOCALETYP(*CLD) is specified on the compilation command.

### Return Value

If the total number of resulting bytes including the ending null character is not more than \( \maxsize \), the \texttt{strfmon()} function returns the number of bytes placed into the array pointed to by \( s \), but excludes the ending null character. Otherwise, zero is returned, and the contents of the array are undefined.

The value of \texttt{errno} may be set to:

**E2BIG** Conversion stopped due to lack of space in the buffer.

### Example that uses \texttt{strfmon()}

```c
#include <stdio.h>
#include <monetary.h>
#include <locale.h>

int main(void)
{
    char string[100];
    double money = 1234.56;
    if (setlocale(LC_ALL, "/qsys.lib/en_us.locale") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }
    strfmon(string, 100, "\%i", money); /* USD 1,234.56 */
```
printf("%s
", string);
strfmon(string, 100, "%n", money); /* $1,234.56 */
printf("%s
", string);
}

The output should be similar to:
USD 1,234.56
$1,234.56

Related Information

- "localeconv() — Retrieve Information from the Environment" on page 168
- "<monetary.h>" on page 8

strftime() — Convert Date/Time to String

Format

```c
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format, 
    const struct tm *timeptr);
```

Language Level: ANSI

Threadsafe: Yes

Description

The strftime() function places bytes into the array pointed to by s as controlled by the string pointed to by format. The format string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character that determines the behavior of the conversion. All ordinary characters (including the terminating null byte, and multi-byte chars) are copied unchanged into the array. If copying takes place between objects that overlap, then the behavior is undefined. No more than maxsize bytes are placed in the array. The appropriate characters are determined by the values contained in the structure pointed to by timeptr, and by the values stored in the current locale.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name.</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name.</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name.</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name.</td>
</tr>
<tr>
<td>%c</td>
<td>Date/Time in the format of the locale.</td>
</tr>
<tr>
<td>%C</td>
<td>Century number [00-99], the year divided by 100 and truncated to an integer.</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month [01-31].</td>
</tr>
<tr>
<td>%D</td>
<td>Date Format, same as %m/%d/%y.</td>
</tr>
<tr>
<td>%e</td>
<td>Same as %d, except single digit is preceded by a space [1-31].</td>
</tr>
<tr>
<td>%g</td>
<td>2 digit year portion of ISO week date [00,99].</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%G</td>
<td>4 digit year portion of ISO week date. Can be negative.</td>
</tr>
<tr>
<td>%h</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24-hour format [00-23].</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12-hour format [01-12].</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year [001-366].</td>
</tr>
<tr>
<td>%m</td>
<td>Month [01-12].</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [00-59].</td>
</tr>
<tr>
<td>%n</td>
<td>Newline character.</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM string.</td>
</tr>
<tr>
<td>%r</td>
<td>Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.</td>
</tr>
<tr>
<td>%R</td>
<td>24-hour time format without seconds, same as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Second [00-61]. The range for seconds allows for a leap second and a double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Tab character.</td>
</tr>
<tr>
<td>%T</td>
<td>24-hour time format with seconds, same as %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>Weekday [1,7]. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year [00-53]. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%V</td>
<td>ISO week number of the year [01-53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year then it is considered week 1. Otherwise, it is the last week of the previous year, and the next year is week 1 of the new year.</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday [0,6], Sunday is 0.</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year [00-53]. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%x</td>
<td>Date in the format of the locale.</td>
</tr>
<tr>
<td>%X</td>
<td>Time in the format of the locale.</td>
</tr>
<tr>
<td>%y</td>
<td>2 digit year [00,99].</td>
</tr>
<tr>
<td>%Y</td>
<td>4-digit year. Can be negative.</td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.</td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name.</td>
</tr>
<tr>
<td>%%</td>
<td>% character.</td>
</tr>
</tbody>
</table>

**Modified Conversion Specifiers**

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specifier. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the **era** string is the empty string “”, which means that the string is unavailable, then %EY would act like %Y.
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Ec</td>
<td>Date/time for current era.</td>
</tr>
<tr>
<td>%EC</td>
<td>Era name.</td>
</tr>
<tr>
<td>%Ex</td>
<td>Date for current era.</td>
</tr>
<tr>
<td>%EX</td>
<td>Time for current era.</td>
</tr>
<tr>
<td>%Ey</td>
<td>Era year. This is the offset from the base year.</td>
</tr>
<tr>
<td>%EY</td>
<td>Year for current era.</td>
</tr>
<tr>
<td>%Od</td>
<td>Day of the month using alternate digits.</td>
</tr>
<tr>
<td>%Oe</td>
<td>Same as %Od.</td>
</tr>
<tr>
<td>%OH</td>
<td>Hour in 24 hour format using alternate digits.</td>
</tr>
<tr>
<td>%OI</td>
<td>Hour in 12 hour format using alternate digits.</td>
</tr>
<tr>
<td>%Om</td>
<td>Month using alternate digits.</td>
</tr>
<tr>
<td>%OM</td>
<td>Minutes using alternate digits.</td>
</tr>
<tr>
<td>%OS</td>
<td>Seconds using alternate digits.</td>
</tr>
<tr>
<td>%Ou</td>
<td>Weekday using alternate digits. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%OU</td>
<td>Week number of the year using alternate digits. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%OV</td>
<td>ISO week number of the year using alternate digits. See %V for explanation of ISO week number.</td>
</tr>
<tr>
<td>%Ow</td>
<td>Weekday using alternate digits. Sunday is 0.</td>
</tr>
<tr>
<td>%OW</td>
<td>Week number of the year using alternate digits. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%Oy</td>
<td>2-digit year using alternate digits.</td>
</tr>
</tbody>
</table>

**Note:** %C, %D, %e, %h, %n, %r, %R, %t, %T, %u, %V, and the modified conversion specifiers are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Return Value**

If the total number of resulting bytes including the terminating null byte is not more than maxsize, strftime() returns the number of bytes placed into the array pointed to by s, not including the terminating null byte. Otherwise, 0 is returned and the contents of the array are indeterminate.

If a conversion error occurs, errno may be set to ECONVERT.

**Example that uses strftime()**
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[100];
    int rc;
    time_t temp;
    struct tm *timeptr;

    temp = time(NULL);
    timeptr = localtime(&temp);

    rc = strftime(s,sizeof(s),"Today is %A, %b %d.
Time:  %r", timeptr);
    printf("%d characters written.
%s",rc,s);
    return 0;
}

/*************************************************
The output should be similar to:
46 characters written
Today is Wednesday, Oct 24.
Time:  01:01:15 PM
 **************************************************

Related Information
• ”asctime() — Convert Time to Character String” on page 41
• ”asctime_r() — Convert Time to Character String (Restartable)” on page 43
• ”ctime() — Convert Time to Character String” on page 68
• ”ctime_r() — Convert Time to Character String (Restartable)” on page 69
• ”gmtime() — Convert Time” on page 151
• ”gmtime_r() — Convert Time (Restartable)” on page 153
• ”localtime() — Convert Time” on page 173
• ”localtime_r() — Convert Time (Restartable)” on page 174
• ”setlocale() — Set Locale” on page 322
• ”strftime()— Convert String to Date/Time” on page 369
• ”time() — Determine Current Time” on page 391
• ”<time.h>” on page 16

stricmp() - Compare Strings without Case Sensitivity

Format
#include <string.h>
int stricmp(const char *string1, const char *string2);

Note: The stricmp function is available for C++ programs. It is available for C only when the program defines the __cplusplus_strings__ macro.

Language Level: Extension

Threadsafe: Yes.

Description
stricmp compares `string1` and `string2` without sensitivity to case. All alphabetic characters in the two arguments `string1` and `string2` are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

**Return Value**

`stricmp` returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

**Example that uses `stricmp()`**

This example uses `stricmp` to compare two strings.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    /* Compare two strings as lowercase */
    if (0 == stricmp("hello", "HELLO"))
        printf("The strings are equivalent.\n");
    else
        printf("The strings are not equivalent.\n");
    return 0;
}
```

The output should be:

```
The strings are equivalent.
```

**Related Information:**

- "strcmpl() - Compare Strings Without Case Sensitivity” on page 345
- "strcoll() — Compare Strings” on page 346
- "strcsnpl() — Find Offset of First Character Match” on page 348
- "strdup - Duplicate String” on page 350
- "strncmp() — Compare Strings” on page 362
- "strcasemcmp() — Compare Strings without Case Sensitivity” on page 340
- "strnケースcmp() — Compare Strings without Case Sensitivity” on page 359
- "strnicmp - Compare Substrings Without Case Sensitivity” on page 365
- "wcscmp() — Compare Wide-Character Strings” on page 433
- "wcsnccmp() — Compare Wide-Character Strings” on page 443
- "\"cstring.h\”” on page 16

---

**strlen() — Determine String Length**

**Format**
Included `<string.h>`

Size_t strlen(const char *string);

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `strlen()` function determines the length of `string` excluding the ending null character.

**Return Value**

The `strlen()` function returns the length of `string`.

**Example that uses `strlen()`**

This example determines the length of the string that is passed to `main()`.

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    if (argc != 2)
        printf( "Usage: %s string\n", argv[0]);
    else
        printf( "Input string has a length of %i\n", strlen(argv[1]));
}
```

```
/********************  If the input is the string  ********************
********************"How long is this string?", ********************
******************** then the expected output is: ********************
Input string has a length of 24
*/
```

**Related Information**

- “`mblen()` — Determine Length of a Multibyte Character” on page 182
- “`strncat()` — Concatenate Strings” on page 361
- “`strncpy()` — Copy Strings” on page 364
- “`wcslen()` — Calculate Length of Wide-Character String” on page 440
- “`<string.h>`” on page 16

**strncasecmp() — Compare Strings without Case Sensitivity**

**Format**

```c
#include <strings.h>
int strncasecmp(const char *string1, const char *string2, size_t count);
```

**Language Level**: XPG4

**Threadsafe**: Yes.

**Description**
The `strncasecmp()` function compares up to `count` characters of `string1` and `string2` without sensitivity to case. All alphabetic characters in `string1` and `string2` are converted to lowercase before comparison.

The `strncasecmp()` function operates on null terminated strings. The string arguments to the function are expected to contain a null character (`'\0'`) marking the end of the string.

**Return Value**

The `strncasecmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

**Example that uses `strncasecmp()`**

This example uses `strncasecmp()` to compare two strings.

```c
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING ONE";
    char_t *str2 = "string TWO";
    int result;

    result = strncasecmp(str1, str2, 6);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf(""%s" is less than "%s".\n", str1, str2);
    else
        printf(""%s" is greater than "%s".\n", str1, str2);

    return 0;
}

/******* The output should be similar to: **************
Strings compared equal.

*******/
```

**Related Information**

- "strcasecmp() — Compare Strings without Case Sensitivity" on page 340
- "strcmp() — Compare Strings" on page 362
- "strcmp() - Compare Strings without Case Sensitivity" on page 357
- "wcsmp() — Compare Wide-Character Strings" on page 433
- " wcsncmp() — Compare Wide-Character Strings" on page 443
- "__wcsicmp() — Compare Wide Character Strings without Case Sensitivity" on page 439
strncat() — Concatenate Strings

Format

```c
#include <string.h>
char *strncat(char *string1, const char *string2, size_t count);
```

Language Level: ANSI

Threading: Yes.

Description

The `strncat()` function appends the first `count` characters of `string2` to `string1` and ends the resulting string with a null character (`\0`). If `count` is greater than the length of `string2`, the length of `string2` is used in place of `count`.

The `strncat()` function operates on null-ended strings. The string argument to the function should contain a null character (`\0`) marking the end of the string.

Return Value

The `strncat()` function returns a pointer to the joined string (`string1`).

Example that uses `strncat()`

This example demonstrates the difference between `strcat()` and `strncat()`. The `strcat()` function appends the entire second string to the first, whereas `strncat()` appends only the specified number of characters in the second string to the first.
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer";
    char * ptr;
    /* Call strcat with buffer1 and " program" */
    ptr = strcat( buffer1, " program" );
    printf( "strcat : buffer1 = \"%s\"\n", buffer1 );

    /* Reset buffer1 to contain just the string "computer" again */
    memset( buffer1, '\0', sizeof( buffer1 ) );
    ptr = strcpy( buffer1, "computer" );
    /* Call strncat with buffer1 and " program" */
    ptr = strncat( buffer1, " program", 3 );
    printf( "strncat: buffer1 = \"%s\"\n", buffer1 );
}

/*****************  Output should be similar to:  *****************
strcat : buffer1 = "computer program"
strncat: buffer1 = "computer pr"
* /

Related Information
• "strcat() — Concatenate Strings“ on page 341
• "strcmp() — Compare Strings"
• "strcpy() — Copy Strings“ on page 364
• "strpbrk() — Find Characters in String” on page 367
• "strlen() — Find Length of String“ on page 368
• "strstr() — Locate First Occurrence of Character in String” on page 372
• "strrchr() — Locate Last Occurrence of Character in String” on page 373
• "strspn() —Find Offset of First Non-matching Character” on page 374
• "wcsstr() — Locate First Occurrence of Character in String” on page 376
• "wmemcpy() — Copy Characters“ on page 379
• "wmemmove() — Move Characters“ on page 381
• "wcsncpy() — Copy Wide-Character Strings” on page 395
• "wcscpy() — Copy Wide-Character Strings” on page 401
• "wcscstr() — Locate First Occurrence of Character in Wide-Character String” on page 407
• "wcscsrt() — Sort Wide-Character Strings” on page 410
• "wcsrchr() — Locate Last Occurrence of Character in Wide-Character String” on page 412
• "<string.h>” on page 16

strncmp() — Compare Strings

Format
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);

Language Level: ANSI

Threatsafe: Yes.

Description
The strncmp() function compares string1 and string2 to the maximum of count.

Return Value
The `strncmp()` function returns a value indicating the relationship between the strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

**Example that uses `strncmp()`**

This example demonstrates the difference between the `strcmp()` function and the `strncmp()` function.

```c
#include <stdio.h>
#include <string.h>
#define SIZE 10

int main(void)
{
    int result;
    int index = 3;
    char buffer1[SIZE] = "abcdefg";
    char buffer2[SIZE] = "abcfg";
    void print_result( int, char *, char * );
    result = strcmp( buffer1, buffer2 );
    printf( "Comparison of each character\n" );
    printf( "  strcmp: ");
    print_result( result, buffer1, buffer2 );
    result = strncmp( buffer1, buffer2, index);
    printf( "Comparison of only the first %i characters\n", index );
    printf( "  strncmp: ");
    print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
    if ( res == 0 )
        printf( "\"%s\" is identical to \"%s\"\n", p_buffer1, p_buffer2);        
    else if ( res < 0 )
        printf( "\"%s\" is less than \"%s\"\n", p_buffer1, p_buffer2 );
    else
        printf( "\"%s\" is greater than \"%s\"\n", p_buffer1, p_buffer2 );
}

/*****************  Output should be similar to:  *****************/
Comparison of each character
  strcmp: "abcdefg" is less than "abcfg"
Comparison of only the first 3 characters
  strncmp: "abcdefg" is identical to "abcfg"
*/

**Related Information**

- "`strcmp()` — Compare Strings" on page 343
- "`strcspn()` — Find Offset of First Character Match” on page 348
- "`strncat()` — Concatenate Strings” on page 361
strncpy() — Copy Strings

Format

```
#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```

Language Level: ANSI

Threading: Yes.

Description

The `strncpy()` function copies `count` characters of `string2` to `string1`. If `count` is less than or equal to the length of `string2`, a null character (\0) is not appended to the copied string. If `count` is greater than the length of `string2`, the `string1` result is padded with null characters (\0) up to length `count`.

Return Value

The `strncpy()` function returns a pointer to `string1`.

Example that uses `strncpy()`

This example demonstrates the difference between `strcpy()` and `strncpy()`.
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char source[SIZE] = "1234567890";
    char source1[SIZE] = "1234567890";
    char destination[SIZE] = "abcdefg";
    char destination1[SIZE] = "abcdefg";
    char *return_string;
    int index = 5;

    /* This is how strcpy works */
    printf( "destination is originally = '\%s\n', destination );
    return_string = strcpy( destination, source );
    printf( "After strcpy, destination becomes '\%s\n', destination );

    /* This is how strncpy works */
    printf( "destination1 is originally = '\%s\n', destination1 );
    return_string = strncpy( destination1, source1, index );
    printf( "After strncpy, destination1 becomes '\%s\n', destination1 );
}

/*****************  Output should be similar to:  *****************/
destination is originally = 'abcdefg'
After strcpy, destination becomes '1234567890'
destination1 is originally = 'abcdefg'
After strncpy, destination1 becomes '12345fg'

Related Information
• "strcpy() — Copy Strings” on page 347
• "strcspn() — Find Offset of First Character Match” on page 348
• "strncat() — Concatenate Strings” on page 361
• "strncmp() — Compare Strings” on page 362
• "strpbrk() — Find Characters in String” on page 367
• "strrchr() — Locate Last Occurrence of Character in String” on page 373
• "strspn() — Find Offset of First Non-matching Character” on page 374
• "<string.h>” on page 16

strnicmp - Compare Substrings Without Case Sensitivity

Format
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);

Note: The strnset and strset functions are available for C++ programs. They are available for C only when the program defines the __cplusplus__macro.

Language Level: Extension

Threadsafe: Yes.

Description
strnicmp compares, at most, the first \( n \) characters of \( string1 \) and \( string2 \) without sensitivity to case.

The function operates on null terminated strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

Return Value

\texttt{strnicmp} returns a value indicating the relationship between the substrings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>( substring1 ) less than ( substring2 )</td>
</tr>
<tr>
<td>0</td>
<td>( substring1 ) equivalent to ( substring2 )</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>( substring1 ) greater than ( substring2 )</td>
</tr>
</tbody>
</table>

Example that uses \texttt{strnicmp()}

This example uses \texttt{strnicmp} to compare two strings.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1 = "THIS IS THE FIRST STRING";
    char *str2 = "This is the second string";
    int numresult;
    /* Compare the first 11 characters of str1 and str2 without regard to case */
    numresult = strnicmp(str1, str2, 11);
    if (numresult < 0)
        printf("String 1 is less than string2.\n");
    else
        if (numresult > 0)
            printf("String 1 is greater than string2.\n");
        else
            printf("The two strings are equivalent.\n");
    return 0;
}
```

The output should be:

The two strings are equivalent.

Related Information:

- "\texttt{strcmp() — Compare Strings}" on page 343
- "\texttt{strcmpi() — Compare Strings Without Case Sensitivity}" on page 345
- "\texttt{stricmp() — Compare Strings without Case Sensitivity}" on page 357
- "\texttt{strn cmp() — Compare Strings}" on page 362
- "\texttt{wcs cmp() — Compare Wide-Character Strings}" on page 433
- "\texttt{wcsncmp() — Compare Wide-Character Strings}" on page 443
- "\texttt{<string.h>}" on page 16

\textbf{strnset - strset - Set Characters in String}

Format
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);

Note: The strnset and strset functions are available for C++ programs. They are available for C only when the program defines the __cplusplus_strings__ macro.

Language Level: Extension

Threadsafe: Yes.

Description

strnset sets, at most, the first n characters of string to c (converted to a char). If n is greater than the length of string, the length of string is used in place of n. strset sets all characters of string, except the ending null character (\0), to c (converted to a char). For both functions, the string is a null-terminated string.

Return Value

Both strset and strnset return a pointer to the altered string. There is no error return value.

Example that uses strnset() and strset()

In this example, strnset sets not more than four characters of a string to the character 'x'. Then the strset function changes any non-null characters of the string to the character 'k'.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "abcdefghi";
    printf("This is the string: %s\n", str);
    printf("This is the string after strnset: %s\n", strnset((char*)str, 'x', 4));
    printf("This is the string after strset: %s\n", strset((char*)str, 'k'));
    return 0;
}
```

The output should be:

```
This is the string: abcdefghi
This is the string after strnset: xxxxefghi
This is the string after strset: kkkkkkkkk
```

Related Information:

- “strchr() — Search for Character” on page 342
- “strpbrk() — Find Characters in String”
- “wcschr() — Search for Wide Character” on page 431
- “wcspbrk() — Locate Wide Characters in String” on page 447
- “<string.h>” on page 16

strpbrk() — Find Characters in String

Format
#include <string.h>
char *strpbrk(const char *string1, const char *string2);

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `strpbrk()` function locates the first occurrence in the string pointed to by `string1` of any character from the string pointed to by `string2`.

**Return Value**

The `strpbrk()` function returns a pointer to the character. If `string1` and `string2` have no characters in common, a NULL pointer is returned.

**Example that uses strpbrk()**

This example returns a pointer to the first occurrence in the array `string` of either a or b.

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *result, *string = "A Blue Danube";
    char *chars = "ab";

    result = strpbrk(string, chars);
    printf("The first occurrence of any of the characters " %s" in "
           "%s" is %s", chars, string, result);
}
```

/* Output should be similar to: */

The first occurrence of any of the characters "ab" in "The Blue Danube" is "anube"

**Related Information**

- "strchr() — Search for Character” on page 342
- "strcmp() — Compare Strings” on page 343
- "strcspn() — Find Offset of First Character Match” on page 348
- "strncmp() — Compare Strings” on page 362
- "strchr() — Locate Last Occurrence of Character in String” on page 373
- "strspn() —Find Offset of First Non-matching Character” on page 374
- "wcschr() — Search for Wide Character” on page 431
- "wcscspn() — Find Offset of First Wide-Character Match” on page 436
- "wcsstr() — Locate Wide Characters in String” on page 447
- "wcschr() — Locate Last Occurrence of Wide Character in String” on page 450
- "wcsnscs() — Locate Wide-Character Substring” on page 465
- "<string.h>” on page 16
strptime()— Convert String to Date/Time

Format
#include <time.h>
char *strptime(const char *buf, const char *format, struct tm *tm);

Language Level: XPG4

Threadsafe: Yes.

Description

The strptime() function converts the character string pointed to by buf to values that are stored in the tm structure pointed to by tm, using the format specified by format.

The format contains zero or more directives. A directive contains either an ordinary character (not % or a white space), or a conversion specification. Each conversion specification is composed of a % character followed by one or more conversion characters, which specify the replacement required. There must be a white space or other delimiter in both buf and format to be guaranteed that the function will behave as expected. There must be a delimiter between two string-to-number conversions, or the first number conversion may convert characters that belong to the second conversion specifier.

Any whitespace (as specified by isspace()) encountered before a directive is scanned in either the format string or the input string will be ignored. A directive that is an ordinary character must exactly match the next scanned character in the input string. Case is relevant when matching ordinary character directives. If the ordinary character directive in the format string does not match the character in the input string, strptime is not successful. No more characters will be scanned.

Any other conversion specification is matched by scanning characters in the input string until a character that is not a possible character for that specification is found or until no more characters can be scanned. If the specification was string-to-number, the possible character range is +,- or a character specified by isdigit(). Number specifiers do not require leading zeros. If the specification needs to match a field in the current locale, scanning is repeated until a match is found. Case is ignored when matching fields in the locale. If a match is found, the structure pointed to by tm will be updated with the corresponding locale information. If no match is found, strptime is not successful. No more characters will be scanned.

Missing fields in the tm structure may be filled in by strftime if given enough information. For example, if a date is given, tm_yday can be calculated.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Name of day of the week, can be either the full name or an abbreviation.</td>
</tr>
<tr>
<td>%A</td>
<td>Same as %a.</td>
</tr>
<tr>
<td>%b</td>
<td>Month name, can be either the full name or an abbreviation.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%B</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%c</td>
<td>Date/time, in the format of the locale.</td>
</tr>
<tr>
<td>%C</td>
<td>Century number [00-99]. Calculates the year if a two-digit year is used.</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month [1-31].</td>
</tr>
<tr>
<td>%D</td>
<td>Date format, same as %m/%d/%y.</td>
</tr>
<tr>
<td>%e</td>
<td>Same as %d.</td>
</tr>
<tr>
<td>%g</td>
<td>2 digit year portion of ISO week date [00-99].</td>
</tr>
<tr>
<td>%G</td>
<td>4 digit year portion of ISO week date. Can be negative.</td>
</tr>
<tr>
<td>%h</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24-hour format [0-23].</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12-hour format [1-12].</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year [1-366].</td>
</tr>
<tr>
<td>%m</td>
<td>Month [1-12].</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [0-59].</td>
</tr>
<tr>
<td>%n</td>
<td>Skip all whitespaces until a newline character is found.</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM string, used for calculating the hour if 12-hour format is used.</td>
</tr>
<tr>
<td>%r</td>
<td>Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.</td>
</tr>
<tr>
<td>%R</td>
<td>24-hour time format without seconds, same as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Second [00-61]. The range for seconds allows for a leap second and a double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Skip all whitespaces until a tab character is found.</td>
</tr>
<tr>
<td>%t</td>
<td>24 hour time format with seconds, same as %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>Weekday [1-7]. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year [0-53], Sunday is the first day of the week. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%V</td>
<td>ISO week number of the year [1-53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year, it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1 of the new year. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%W</td>
<td>Weekday [0-6]. Sunday is 0.</td>
</tr>
<tr>
<td>%w</td>
<td>Week number of the year [0-53]. Monday is the first day of the week. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%x</td>
<td>Date in the format of the locale.</td>
</tr>
<tr>
<td>%X</td>
<td>Time in the format of the locale.</td>
</tr>
<tr>
<td>%y</td>
<td>2-digit year [0-99].</td>
</tr>
<tr>
<td>%Y</td>
<td>4-digit year. Can be negative.</td>
</tr>
<tr>
<td>%Z</td>
<td>UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.</td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name.</td>
</tr>
</tbody>
</table>
### Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the `era` string is the empty string "", which means that `era` is unavailable, then `%EY` would act like `%Y`.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Ec</td>
<td>Date/time for current era.</td>
</tr>
<tr>
<td>%EC</td>
<td>Era name.</td>
</tr>
<tr>
<td>%Ex</td>
<td>Date for current era.</td>
</tr>
<tr>
<td>%EX</td>
<td>Time for current era.</td>
</tr>
<tr>
<td>%Ey</td>
<td>Era year. This is the offset from the base year.</td>
</tr>
<tr>
<td>%EY</td>
<td>Year for the current era.</td>
</tr>
<tr>
<td>%Od</td>
<td>Day of the month using alternate digits.</td>
</tr>
<tr>
<td>%Oe</td>
<td>Same as %Od.</td>
</tr>
<tr>
<td>%OH</td>
<td>Hour in 24-hour format using alternate digits.</td>
</tr>
<tr>
<td>%OI</td>
<td>Hour in 12-hour format using alternate digits.</td>
</tr>
<tr>
<td>%Om</td>
<td>Month using alternate digits.</td>
</tr>
<tr>
<td>%OM</td>
<td>Minutes using alternate digits.</td>
</tr>
<tr>
<td>%OS</td>
<td>Seconds using alternate digits.</td>
</tr>
<tr>
<td>%Ou</td>
<td>Day of the week using alternate digits. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%OU</td>
<td>Week number of the year using alternate digits. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%OV</td>
<td>ISO week number of the year using alternate digits. See %V for explanation of ISO week number.</td>
</tr>
<tr>
<td>%Ow</td>
<td>Weekday using alternate digit. Sunday is 0 and Saturday is 6.</td>
</tr>
<tr>
<td>%OW</td>
<td>Week number of the year using alternate digits. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%Oy</td>
<td>2-digit year using alternate digits.</td>
</tr>
</tbody>
</table>

**Note:** This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

### Return Value

On successful completion, the `strptime()` function returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned. The value of `errno` may be set to `ECONVERT` (conversion error).

**Example that uses `strptime()`**
```c
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
    char buf[100];
    time_t t;
    struct tm *timeptr,result;

    setlocale(LC_ALL,"/QSYS.LIB/EN_US.LOCALE");
    t = time(NULL);
    timeptr = localtime(&t);
    strftime(buf,sizeof(buf), "%a %m/%d/%Y %r", timeptr);

    if(strptime(buf, "%a %m/%d/%Y %r",&result) == NULL)
    printf("strftime failed\n");
    else
    {
        printf("tm_hour:  %d\n",result.tm_hour);
        printf("tm_min:  %d\n",result.tm_min);
        printf("tm_sec:  %d\n",result.tm_sec);
        printf("tm_mon:  %d\n",result.tm_mon);
        printf("tm_mday:  %d\n",result.tm_mday);
        printf("tm_year:  %d\n",result.tm_year);
        printf("tm_yday:  %d\n",result.tm_yday);
        printf("tm_wday:  %d\n",result.tm_wday);
    }

    return 0;
}
```

The output should be similar to:
Tue 10/30/2001 10:59:10 AM

Related Information:
- `asctime() — Convert Time to Character String” on page 41`
- `asctime_r() — Convert Time to Character String (Restartable)” on page 43`
- `ctime() — Convert Time to Character String” on page 68`
- `ctime_r() — Convert Time to Character String (Restartable)” on page 69`
- `gmtime() — Convert Time” on page 151`
- `gmtime_r() — Convert Time (Restartable)” on page 153`
- `localtime() — Convert Time” on page 173`
- `localtime_r() — Convert Time (Restartable)” on page 174`
- `setlocale() — Set Locale” on page 322`
- `strftime() — Convert Date/Time to String” on page 354`
- `time() — Determine Current Time” on page 391`
- `”<time.h>” on page 16`
- `wcsptime()— Convert Wide Character String to Date/Time” on page 448`
strrchr() — Locate Last Occurrence of Character in String

Format
#include <string.h>
char *strrchr(const char *string, int c);

Language Level: ANSI

Threadsafe: Yes.

Description
The strrchr() function finds the last occurrence of c (converted to a character) in string. The ending null character is considered part of the string.

Return Value
The strrchr() function returns a pointer to the last occurrence of c in string. If the given character is not found, a NULL pointer is returned.

Example that uses strrchr()

This example compares the use of strchr() and strrchr(). It searches the string for the first and last occurrence of p in the string.

#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buf[SIZE] = "computer program";
    char * ptr;
    int    ch = 'p';

    /* This illustrates strchr */
    ptr = strchr( buf, ch );
    printf( "The first occurrence of %c in '%s' is '%s'
", ch, buf, ptr );

    /* This illustrates strrchr */
    ptr = strrchr( buf, ch );
    printf( "The last occurrence of %c in '%s' is '%s'
", ch, buf, ptr );
}

/*************************  Output should be similar to:  ****************************
The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'

Related Information
- "strchr() — Search for Character" on page 342
- "strcmp() — Compare Strings" on page 343
- "strcspn() — Find Offset of First Character Match" on page 348
- "strncmp() — Compare Strings" on page 362
- "strpbrk() — Find Characters in String" on page 367
- "strspn() — Find Offset of First Non-matching Character" on page 374
strspn() —Find Offset of First Non-matching Character

Format

```
#include <string.h>
size_t strspn(const char *string1, const char *string2);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `strspn()` function finds the first occurrence of a character in `string1` that is not contained in the set of characters that is specified by `string2`. The null character (\0) that ends `string2` is not considered in the matching process.

Return Value

The `strspn()` function returns the index of the first character found. This value is equal to the length of the initial substring of `string1` that consists entirely of characters from `string2`. If `string1` begins with a character not in `string2`, the `strspn()` function returns 0. If all the characters in `string1` are found in `string2`, the length of `string1` is returned.

Example that uses `strspn()`

This example finds the first occurrence in the array `string` of a character that is not an a, b, or c. Because the string in this example is cabbage, the `strspn()` function returns 5, the length of the segment of cabbage before a character that is not an a, b, or c.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char * string = "cabbage";
    char * source = "abc";
    int index;

    index = strspn( string, "abc" );
    printf( "The first %d characters of "%s" are found in "%s"
            index, string, source );
}
```

/******************** Output should be similar to: ********************/

The first 5 characters of "cabbage" are found in "abc"
*/

Related Information

- “strcat() — Concatenate Strings” on page 341
- “strchr() — Search for Character” on page 342
- “strcmp() — Compare Strings” on page 343
- “strcpy() — Copy Strings” on page 347
- “strcspn() — Find Offset of First Character Match” on page 348
 strstr() — Locate Substring

Format
#include <string.h>
char *strstr(const char *string1, const char *string2);

Language Level: ANSI

Threadsafe: Yes.

Description

The strstr() function finds the first occurrence of string2 in string1. The function ignores the null character (\0) that ends string2 in the matching process.

Return Value

The strstr() function returns a pointer to the beginning of the first occurrence of string2 in string1. If string2 does not appear in string1, the strstr() function returns NULL. If string2 points to a string with zero length, the strstr() function returns string1.

Example that uses strstr()

This example locates the string "haystack" in the string "needle in a haystack".

#include <string.h>
#include <stdio.h>

int main(void)
{
    char *string1 = "needle in a haystack";
    char *string2 = "haystack";
    char *result;

    result = strstr(string1,string2);
    /* Result = a pointer to "haystack" */
    printf("%s\n", result);
}

/*****************  Output should be similar to: *****************/
haystack

Related Information

• ”strchr() — Search for Character” on page 342
strtod() — Convert Character String to Double

Format

```c
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Language Level: ANSI

Threading: Yes.

Description

The `strtod()` function converts a character string to a double-precision value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of the type double. This function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the null character at the end of the string.

The `strtod()` function expects `nptr` to point to a string with the following form:

```
whitespace [ + - ] digits [. digits] [e [ + - ] digits]
```

The first character that does not fit this form stops the scan.

The behavior of this function is affected by the `LC_NUMERIC` category of the current locale.

Return Value

The `strtod()` function returns the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, it returns -HUGE_VAL or +HUGE_VAL; for an underflow, it returns 0.
In both cases, errno is set to ERANGE, depending on the base of the value. If the string pointed to by nptr does not have the expected form, no conversion is performed and the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

The `strtod()` function does not fail if a character other than a digit follows an E or e read in as an exponent. For example, 100elf will be converted to the floating-point value 100.0.

**Example that uses `strtod()`**

This example converts the strings to a double value. It prints out the converted value and the substring that stopped the conversion.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    char *string, *stopstring;
    double x;

    string = "3.1415926\This stopped it";
    x = strtod(string, &stopstring);
    printf("string = %s\n", string);
    printf("    strtod = %lf\n", x);
    printf("    Stopped scan at %s\n", stopstring);

    string = "100\ergs";
    x = strtod(string, &stopstring);
    printf("string = "\n", string);
    printf("    strtod = %lf\n", x);
    printf("    Stopped scan at "%s\n", stopstring);
}
```

```
/*****************  Output should be similar to:  *****************/

string = 3.1415926\This stopped it
    strtod = 3.141593
    Stopped scan at This stopped it

string = 100\ergs
    strtod = 100.000000
    Stopped scan at \ergs
```

**Related Information**

- “`atof()` — Convert Character String to Float” on page 49
- “`atoi()` — Convert Character String to Integer” on page 50
- “`atol()` — Convert Character String to Long or Long Long Integer” on page 51
- “`strtol()` — Convert Character String to Long and Long Long Integer” on page 380
- “`strtoull()` — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 382
- “`wcstod()` — Convert Wide-Character String to Double” on page 455
- “`<stdlib.h>`” on page 15
strtok() — Tokenize String

Format
#include <string.h>
char *strtok(char *string1, const char *string2);

Language Level: ANSI

Threadsafe: No. Use strtok_r() instead.

Description

The strtok() function reads string1 as a series of zero or more tokens, and string2 as the set of characters serving as delimiters of the tokens in string1. The tokens in string1 can be separated by one or more of the delimiters from string2. The tokens in string1 can be located by a series of calls to the strtok() function.

In the first call to the strtok() function for a given string1, the strtok() function searches for the first token in string1, skipping over leading delimiters. A pointer to the first token is returned.

When the strtok() function is called with a NULL string1 argument, the next token is read from a stored copy of the last non-null string1 parameter. Each delimiter is replaced by a null character. The set of delimiters can vary from call to call, so string2 can take any value. Note that the initial value of string1 is not preserved after the call to the strtok() function.

Note that the strtok() function writes data into the buffer. The function should be passed to a non-critical buffer containing the string to be tokenized because the buffer will be damaged by the strtok() function.

Return Value

The first time the strtok() function is called, it returns a pointer to the first token in string1. In later calls with the same token string, the strtok() function returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-ended.

Note: The strtok() function uses an internal static pointer to point to the next token in the string being tokenized. A reentrant version of the strtok() function, strtok_r(), which does not use any internal static storage, can be used in place of the strtok() function.

Example that uses strtok()

Using a loop, this example gathers tokens, separated by commas, from a string until no tokens are left. The example prints the tokens, a string, of, and tokens.
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *token, *string = "a string, of, , tokens\0, after null terminator";

    /* the string pointed to by string is broken up into the tokens
    "a string", " of", " ", and "tokens" ; the null terminator (\0)
    is encountered and execution stops after the token "tokens" */
    token = strtok(string, ",");
    do
    {
        printf("token: \%s\n", token);
    }
    while (token = strtok(NULL, ","));
}

/* *************** Output should be similar to: ***************
token: a string
token: of
token: tokens */

Related Information

- "strcat() — Concatenate Strings" on page 341
- "strchr() — Search for Character" on page 342
- "strcmp() — Compare Strings" on page 343
- "strcpy() — Copy Strings" on page 347
- "strcspn() — Find Offset of First Character Match" on page 348
- "strspn() — Find Offset of First Non-matching Character" on page 374
- "strtok_r() — Tokenize String (Restartable)"
- "<string.h>" on page 16

**strtok_r() — Tokenize String (Restartable)**

**Format**

```
#include <string.h>
char *strtok_r(char *string, const char *seps,
               char **lasts);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**

This function is the restartable version of **strtok()**.

The **strtok_r()** function reads **string** as a series of zero or more tokens, and **seps** as the set of characters serving as delimiters of the tokens in **string**. The tokens in **string** can be separated by one or more of the delimiters from **seps**. The arguments **lasts** points to a user-provided pointer, which points to stored information necessary for the **strtok_r()** function to continue scanning the same string.
In the first call to the `strtok_r()` function for a given null-ended `string`, it searches for the first token in `string`, skipping over leading delimiters. It returns a pointer to the first character of the first token, writes a null character into `string` immediately following the returned token, and updates the pointer to which `lasts` points.

To read the next token from `string`, call the `strtok_r()` function with a NULL `string` argument. This causes the `strtok_r()` function to search for the next token in the previous token string. Each delimiter is replaced in the original `string` is replaced by a null character, and the pointer to which `lasts` points is updated. The set of delimiters in `seps` can vary from call to call, but `lasts` must remain unchanged from the previous call. When no tokens remain in `string`, a NULL pointer is returned.

**Return Value**

The first time the `strtok_r()` function is called, it returns a pointer to the first token in `string`. In later calls with the same token string, the `strtok_r()` function returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-ended.

**Related Information**

- "strcat() — Concatenate Strings” on page 341
- “strchr() — Search for Character” on page 342
- “strcmp() — Compare Strings” on page 343
- “strcpy() — Copy Strings” on page 347
- “strcspn() — Find Offset of First Character Match” on page 348
- “strspn() — Find Offset of First Non-matching Character” on page 374
- “strtok() — Tokenize String” on page 378
- “<string.h>” on page 16

### `strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer

**Format**

```c
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);

#include <stdlib.h>
long long int strtoll(char *string, char **endptr, int base);
```

**Language Level:** ANSI

**Threading Safe:** Yes.

**Description**

The `strtol()` function converts a character string to a long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type long int.

The `strtoll()` function converts a character string to a long long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type long long int.
When you use these functions, the `nptr` parameter should point to a string with the following form:

```
whitespace [+ - 0 0x 0X digits]
```

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the `base` parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base parameter are permitted. If the base parameter has a value of 16, the characters `0x` or `0X` optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the base. After an optional leading sign a leading 0 indicates octal conversion, a leading `0x` or `0X` indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the `base` parameter. This character may be the null character (`\0`) at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the `endptr` parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by `endptr`. If a value cannot be formed, the value pointed to by the `endptr` parameter is set to the value of the `nptr` parameter.

**Return Value**

If base has an invalid value (less than 0, 1, or greater than 36), `errno` is set to `EINVAL` and 0 is returned. The value pointed to by the `endptr` parameter is set to the value of the `nptr` parameter.

If the value is outside the range of representable values, `errno` is set to `ERANGE`. If the value is positive, the `strtol()` function will return `LONG_MAX`, and the `strtoll()` function will return `LONGLONG_MAX`. If the value is negative, the `strtol()` function will return `LONG_MIN`, and the `strtoll()` function will return `LONGLONG_MIN`.

If no characters are converted, the `strtoll()` and `strtol()` functions will set `errno` to `EINVAL` and 0 is returned. For both functions, the value pointed to by `endptr` is set to the value of the `nptr` parameter. Upon successful completion, both functions return the converted value.

**Example that uses `strtol()`**

This example converts the strings to a long value. It prints out the converted value and the substring that stopped the conversion.
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  char *string, *stopstring;
  long l;
  int bs;

  string = "10110134932";
  printf("string = %s\n", string);
  for (bs = 2; bs <= 8; bs *= 2)
  {
    l = strtol(string, &stopstring, bs);
    printf("   strtol = %ld (base %d)\n", l, bs);
    printf("   Stopped scan at %s\n", stopstring);
  }
}

/***************************  Output should be similar to:  *****************************/
string = 10110134932
   strtol = 45 (base 2)
           Stopped scan at 34932
   strtol = 4423 (base 4)
           Stopped scan at 4932

Related Information
• ”atof() — Convert Character String to Float” on page 49
• ”atoi() — Convert Character String to Integer” on page 50
• ”atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 51
• ”strtod() — Convert Character String to Double” on page 376
• ”strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer”
• ”wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 458
• ”<stdlib.h>” on page 15

strtol() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer

Format (strtol())
#include <stdlib.h>
unsigned long int strtol(const char *nptr, char **endptr, int base);

Format (strtoull())
#include <stdlib.h>
unsigned long long int strtoull(char *string, char **endptr, int base);

Language Level: ANSI

Threadsafe: Yes.

Description
The `strtoul()` function converts a character string to an unsigned long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type unsigned long int.

The `strtoull()` function converts a character string to an unsigned long long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type unsigned long long int.

When you use these functions, the `nptr` parameter should point to a string with the following form:

```
whitespace  +  0  - 0x  0X
digits
```

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the base parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35. Only letters whose ascribed values are less than that of the base parameter are permitted. If the `base` parameter has a value of 16 the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the `base`. After an optional leading sign a leading 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the base parameter. This character may be the null character (‘\0’) at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the `endptr` parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by `endptr`. If a value cannot be formed, the value pointed to by `endptr` is set to the `nptr` parameter.

**Return Value**

If `base` has an invalid value (less than 0, 1, or greater than 36), `errno` is set to EINVAL and 0 is returned. The value pointed to by the `endptr` parameter is set to the value of the `nptr` parameter.

If the value is outside the range of representable values, `errno` is set to ERANGE. The `strtoul()` function will return ULONG_MAX and the `strtoull()` function will return ULONGLONG_MAX.

If no characters are converted, the `strtoull()` function will set `errno` to EINVAL and 0 is returned. The `strtoul()` function will return 0 but will not set `errno` to EINVAL. In both cases the value pointed to by `endptr` is set to the value of the `nptr` parameter. Upon successful completion, both functions return the converted value.
Example that uses `strtoul()`

This example converts the string to an unsigned long value. It prints out the converted value and the substring that stopped the conversion.

```c
#include <stdio.h>
#include <stdlib.h>

#define BASE 2

int main(void)
{
    char *string, *stopstring;
    unsigned long ul;

    string = "1000e13 e";
    printf("string = %s\n", string);
    ul = strtoul(string, &stopstring, BASE);
    printf("   strtoul = %ld (base %d)\n", ul, BASE);
    printf("   Stopped scan at %s\n", stopstring);
}
```

```c
******* Output should be similar to: ************

string = 1000e13 e
   strtoul = 8 (base 2)
   Stopped scan at e13 e
*/
```

Related Information

- “[atof() — Convert Character String to Float” on page 49
- “atoi() — Convert Character String to Integer” on page 50
- “[atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 51
- “[strtod() — Convert Character String to Double” on page 376
- “[strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 380
- “[wcstoul() — wcstoull() — Convert WideCharacter String to Unsigned Long and Unsigned Long Long Integer” on page 463
- “[<stdlib.h>” on page 15

`strxfrm()` — Transform String

Format

```c
#include <string.h>
size_t strxfrm(char *string1, const char *string2, size_t count);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `strxfrm()` function transforms the string pointed to by `string2` and places the result into the string pointed to by `string1`. The transformation is determined by the program’s current locale. The transformed string is not necessarily readable, but can be used with the `strcmp()` or `strncmp()` functions.
The behavior of this function is affected by the LC_COLLATE category of the current locale.

**Return Value**

The `strxfrm()` function returns the length of the transformed string, excluding the ending null character. If the returned value is greater than or equal to `count`, the contents of the transformed string are indeterminate.

If `strxfrm()` is unsuccessful, errno is changed. The value of errno may be set to EINVAL (the `string1` or `string2` arguments contain characters which are not available in the current locale).

**Example that uses `strxfrm()`**

This example prompts the user to enter a string of characters, then uses `strxfrm()` to transform the string and return its length.

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1, buffer[80];
    int length;

    printf("Type in a string of characters.\n ");
    string1 = gets(buffer);
    length = strxfrm(NULL, string1, 0);
    printf("You would need a %d element array to hold the string
",length);
    printf("\n\n%\n\n transformed according",string1);
    printf(" to this program's locale. \n");
}
```

**Related Information**

- “`localeconv()` — Retrieve Information from the Environment” on page 168
- “`setlocale()` — Set Locale” on page 322
- “`strcmp()` — Compare Strings” on page 343
- “`strcoll()` — Compare Strings” on page 346
- “`strncmp()` — Compare Strings” on page 362
- “`<string.h>`” on page 16

---

**swprintf() — Format and Write Wide Characters to Buffer**

**Format**

```c
#include <wchar.h>
int swprintf(wchar_t *wcsbuffer, size_t n,
              const wchar_t *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**
The `swprintf()` function formats and stores a series of wide characters and values into the wide-character buffer `wcsbuffer`. The `swprintf()` function is equivalent to the `sprintf()` function, except that it operates on wide characters.

The value `n` specifies the maximum number of wide characters to be written, including the ending null character. The `swprintf()` function converts each entry in the argument-list according to the corresponding wide-character format specifier in `format`. The format has the same form and function as the format string for the `printf()` function, with the following exceptions:

- `%c` (without an `l` prefix) converts a character argument to `wchar_t`, as if by calling the `mbtowc()` function.
- `%lc` and `%C` copy a `wchar_t` to `wchar_t`. `%#lc` and `%#C` are equivalent to `%lc` and `%C`, respectively.
- `%s` (without an `l` prefix) converts an array of multibyte characters to an array of `wchar_t`, as if by calling the `mbstowcs()` function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- `%ls` and `%S` copy an array of `wchar_t` (no conversion). The array is written up to, but not including, the ending NULL character, unless the precision specifies a shorter output. `%#ls` and `%#S` are equivalent to `%ls` and `%S`, respectively.

Width and precision always are wide characters.

When the program is compiled with `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)`, and `SYSIFCOPT(*IFSIO)`, the wide characters that are written by this function are Unicode characters, and the inputs for `%lc` and `%ls` are assumed to be Unicode characters. To view Unicode examples, see "fwprintf() — Format Data as Wide Characters and Write to a Stream" on page 133.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

In extended mode, the `swprintf()` function also converts floating-point values of NaN and infinity to the strings "NAN" or "nan" and "INFINITY" or "infinity". The case and sign of the string is determined by the format specifiers.

Note: This function is not available when either `LOCALETYPE(*CLD)` or `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

Return Value

The `swprintf()` function returns the number of wide characters that are written in the array, not counting the ending null wide character.

The value of `errno` may be set to `EINVAL`, invalid argument.

Example that uses `swprintf()`

This example uses the `swprintf()` function to format and print several values to buffer.
```c
#include <wchar.h>
#define BUF_SIZE 100

int main(void)
{
    wchar_t wcsbuf[BUF_SIZE];
    wchar_t wstring[] = L"ABCDE";
    int num;

    num = swprintf(wcsbuf, BUF_SIZE, L"%s", L"xyz");
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%ls", wstring);
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%i", 100);
    printf("The array wcsbuf contains: \"%ls\"\n", wcsbuf);
    return 0;
}
```

---

**swscanf() — Read Wide Character Data**

**Format**

```c
#include <wchar.h>
int swscanf(const wchar_t *buffer, const wchar_t *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `swscanf()` function is equivalent of the `fwscanf()` function, except that the argument buffer specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the `fwscanf()` function.

**Return Value**

The `swscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF when the end of the string is encountered before anything is converted.

The value of `errno` may be set `EINVAL`, invalid argument.

**Example that uses swscanf()**

---

**Related Information**

- "printf() — Print Formatted Characters" on page 211
- "sprintf() — Print Formatted Data to Buffer" on page 335
- "vswprintf() — Format and Write Wide Characters to Buffer" on page 418
- "<wchar.h>" on page 17
This example uses the `swscanf()` function to read various data from the string `ltokenstring`, and then displays that data.

```c
#include <wchar.h>
#include <stdio.h>

wchar_t *ltokenstring = L"15 12 14";
int i;
float fp;
char s[10];
char c;

int main(void)
{
    /* Input various data */
    swscanf(ltokenstring, L "%s %c%d%f", s, &c, &i, &fp);

    /* If there were no space between %s and %c, swscanf would read the first character following */
    /* the string, which is a blank space. */
    printf("string = %s\n", s);
    printf("character = %c\n", c);
    printf("integer = %d\n", i);
    printf("floating-point number = %f\n", fp);
}
```

Related Information

- "`fscanf()` — Read Formatted Data" on page 123
- "`scanf()` — Read Data" on page 313
- "`fwscanf()` — Read Data from Stream Using Wide Character" on page 137
- "`wscanf()` — Read Data Using Wide-Character Format String" on page 482
- "`sscanf()` — Read Data" on page 339
- "`sprintf()` — Print Formatted Data to Buffer" on page 335
- "`<wchar.h>`" on page 17

---

**system() — Execute a Command**

**Format**

```c
#include <stdlib.h>
int system(const char *string);
```

**Language Level:** ANSI

**Thaadsafe:** Yes. However, the CL command processor and all CL commands are NOT threadsafe. Use this function with caution.

**Description**

The `system()` function passes the given `string` to the CL command processor for processing.

**Return Value**

If passed a non-NULL pointer to a string, the `system()` function passes the argument to the CL command processor. The `system()` function returns zero if the
command is successful. If passed a NULL pointer to a string, system() returns -1, and the command processor is not called. If the command fails, system() returns 1. If the system() function fails, the global variable _EXCP_MSGID in <stddef.h> is set with the exception message ID. The exception message ID set within the _EXCP_MSGID variable is in job CCSID.

Example that uses system()

```c
#include <stdlib.h>

int main(void)
{
    int result;

    /* A data area is created, displayed and deleted: */
    result = system("CRTDTAARA QTEMP/TEST TYPE(*CHAR) VALUE('Test')");
    result = system("DSPDTAARA TEST");
    result = system("DLTDATAARA TEST");
}
```

Related Information

- "exit() — End Program" on page 80
- "<stdlib.h>" on page 15

---

**tan() — Calculate Tangent**

**Format**

```c
#include <math.h>
double tan(double x);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The tan() function calculates the tangent of x, where x is expressed in radians. If x is too large, a partial loss of significance in the result can occur and sets errno to ERANGE. The value of errno may also be set to EDOM.

**Return Value**

The tan() function returns the value of the tangent of x.

**Example that uses tan()**

This example computes x as the tangent of π/4.
```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x;
    pi = 3.1415926;
    x = tan(pi/4.0);
    printf("tan( %lf ) is %lf\n", pi/4, x);
}

/*
*************** Output should be similar to: ***************
tan( 0.785398 ) is 1.000000
*/
```

**Related Information**
- [acos() — Calculate Arccosine](#)
- [asin() — Calculate Arcsine](#)
- [atan() — atan2() — Calculate Arctangent](#)
- [cos() — Calculate Cosine](#)
- [cosh() — Calculate Hyperbolic Cosine](#)
- [sin() — Calculate Sine](#)
- [sinh() — Calculate Hyperbolic Sine](#)
- [tanh() — Calculate Hyperbolic Tangent](#)
- [<math.h>](#)

---

### tanh() — Calculate Hyperbolic Tangent

**Format**
```
#include <math.h>
double tanh(double x);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `tanh()` function calculates the hyperbolic tangent of `x`, where `x` is expressed in radians.

**Return Value**

The `tanh()` function returns the value of the hyperbolic tangent of `x`. The result of `tanh()` cannot have a range error.

**Example that uses tanh()**

This example computes `x` as the hyperbolic tangent of `π/4`. 

---

390  ILE C/C++ Run-Time Library Functions V5R3
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x;
    pi = 3.1415926;
    x = tanh(pi/4);
    printf("tanh( %lf ) = %lf\n", pi/4, x);
}

/******************  Output should be similar to:  ****************
tanh( 0.785398 ) = 0.655794
*/

Related Information
• "acos() — Calculate Arccosine" on page 40
• "asin() — Calculate Arcsine" on page 44
• "atan() – atan2() — Calculate Arctangent" on page 47
• "cos() — Calculate Cosine" on page 66
• "cosh() — Calculate Hyperbolic Cosine" on page 67
• "sin() — Calculate Sine" on page 332
• "sinh() — Calculate Hyperbolic Sine" on page 333
• "tan() — Calculate Tangent" on page 389
• "<math.h>" on page 7

---

## time() — Determine Current Time

**Format**

```c
#include <time.h>

#include <stdio.h>

int main(void)
{
    double pi, x;
    pi = 3.1415926;
    x = tanh(pi/4);
    printf("tanh( %lf ) = %lf\n", pi/4, x);
}
```

---

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `time()` function determines the current calendar time, in seconds.

**Note:** Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

**Return Value**

The `time()` function returns the current calendar time. The return value is also stored in the location that is given by `timeptr`. If `timeptr` is NULL, the return value is not stored. If the calendar time is not available, the value `(time_t)(-1)` is returned.

**Example that uses time()**
This example gets the time and assigns it to `ltime`. The `ctime()` function then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    if(time(&ltime) == -1)
    {
        printf("Calendar time not available.\n");
        exit(1);
    }
    printf("The time is %s\n", ctime(&ltime));
}
```

Related Information
- "asctime() — Convert Time to Character String" on page 41
- "asctime_r() — Convert Time to Character String (Restartable)" on page 43
- "ctime() — Convert Time to Character String" on page 68
- "ctime_r() — Convert Time to Character String (Restartable)" on page 69
- "gmtime() — Convert Time" on page 151
- "gmtime_r() — Convert Time (Restartable)" on page 153
- "localtime() — Convert Time" on page 173
- "localtime_r() — Convert Time (Restartable)" on page 174
- "mktime() — Convert Local Time" on page 204
- "<time.h>" on page 16

---

tmpfile() — Create Temporary File

**Format**

```c
#include <stdio.h>
FILE *tmpfile(void);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `tmpfile()` function creates a temporary binary file. The file is automatically removed when it is closed or when the program is ended.

The `tmpfile()` function opens the temporary file in wb+ mode.

**Return Value**
The `tmpfile()` function returns a stream pointer, if successful. If it cannot open the file, it returns a NULL pointer. On normal end (`exit()`), these temporary files are removed.

On the iSeries Data Management system, the `tmpfile()` function creates a new file that is named QTEMP/QACXxxxx. If you specify the SYSFDCOPT(*IFSIO) option on the compilation command, the `tmpfile()` function creates a new file that is named /tmp/QACXaaaaaaa. At the end of the job, the file that is created with the filename from the `tmpfile()` function is discarded. You can use the `remove()` function to remove files.

Example that uses `tmpfile()`

This example creates a temporary file, and if successful, writes `tmpstring` to it. At program end, the file is removed.

```c
#include <stdio.h>

FILE *stream;
char tmpstring[ ] = "This is the string to be temporarily written";

int main(void)
{
    if((stream = tmpfile( )) == NULL)
        perror("Cannot make a temporary file");
    else
        fprintf(stream, "%s", tmpstring);
}
```

Related Information
- "`fopen() — Open Files" on page 100"
- "<stdio.h>" on page 14

---

**tmpnam() — Produce Temporary File Name**

**Format**

```
#include <stdio.h>
char *tmpnam(char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, using `tmpnam(NULL)` is NOT threadsafe.

**Description**

The `tmpnam()` function produces a valid file name that is not the same as the name of any existing file. It stores this name in `string`. If `string` is NULL, the `tmpnam()` function leaves the result in an internal static buffer. Any subsequent calls destroy this value. If `string` is not NULL, it must point to an array of at least `L_tmpnam` bytes. The value of `L_tmpnam` is defined in `<stdio.h>`.

The `tmpnam()` function produces a different name each time it is called within an activation group up to at least TMP_MAX names. For ILE C, TMP_MAX is 32 767. This is a theoretical limit; the actual number of files that can be opened at the same time depends on the available space in the system.

**Return Value**
The `tmpnam()` function returns a pointer to the name. If it cannot create a unique name then it returns NULL.

**Example that uses tmpnam()**

This example calls `tmpnam()` to produce a valid file name.

```c
#include <stdio.h>

int main(void)
{
    char *name1;
    if ((name1 = tmpnam(NULL)) != NULL)
        printf("%s can be used as a file name.\n", name1);
    else printf("Cannot create a unique file name\n");
}
```

**Related Information**
- "fopen() — Open Files" on page 100
- "remove() — Delete File" on page 257
- "<stdio.h>" on page 14

---

**toascii() — Convert Character to Character Representable by ASCII**

**Format**

```c
#include <ctype.h>
int toascii(int c);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**

The `toascii()` function determines to what character `c` would be mapped to in a 7-bit US-ASCII locale and returns the corresponding character encoding in the current locale.

**Return Value**

The `toascii()` function maps the character `c` according to a 7-bit US-ASCII locale and returns the corresponding character encoding in the current locale.

**Example that uses toascii()**

This example prints encodings of the 7-bit US-ASCII characters 0x7c to 0x82 are mapped to by `toascii()`.

```c
#include <stdio.h>

int main(void)
{
    for (int c = 0x7c; c <= 0x82; c++)
        printf("%d: \%c\n", c, toascii(c));
}
```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int ch;
    for (ch=0x7c; ch<=0x82; ch++) {
        printf("tolower(\%04x) = \%c\n", ch, toascii(ch));
    }
}

/***************And the output should be:**************************
toascii(0x7c) = @
toascii(0x7d) = '
toascii(0x7e) = =
toascii(0x7f) = "
toascii(0x80) = X
toascii(0x81) = a
toascii(0x82) = b
******************************/

Related Information
• "isascii() — Test for Character Representable as ASCII Value" on page 158
• "<ctype.h>" on page 3

tolower() — toupper() — Convert Character Case

Format
#include <ctype.h>
int tolower(int C);
int toupper(int c);

Language Level: ANSI

Threa dsafe: Yes.

Description
The tolower() function converts the uppercase letter C to the corresponding lowercase letter.

The toupper() function converts the lowercase letter c to the corresponding uppercase letter.

Return Value
Both functions return the converted character. If the character c does not have a corresponding lowercase or uppercase character, the functions return c unchanged.

Note: This function is locale sensitive.

Example that uses toupper() and tolower()

This example uses the toupper() and tolower() functions to change characters between code 0 and code 7f.
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;
    for (ch = 0; ch <= 0x7f; ch++)
    {
        printf("toupper=%#04x\n", toupper(ch));
        printf("tolower=%#04x\n", tolower(ch));
        putchar('\n');
    }
}

Related Information
- “isalnum() - isxdigit() — Test Integer Value” on page 156
- “towlower() – towupper() — Convert Wide Character Case” on page 397
- “<ctype.h>” on page 3


towctrans() — Translate Wide Character

Format
#include <wctype.h>
#wint_t towctrans(wint_t wc, wctrans_t desc);

Language Level: ANSI

Threatsage: Yes.

Description
The towctrans() function maps the wide character wc using the mapping that is described by desc. The current setting of the LC_CTYPE category (or the LC UNI_CTYPE category if a UNICODE LOCALETYPES was specified) will be the same as the one used during the call to the wctrans() function that returned the value desc.

A towctrans(wc, wctrans("tolower")) behaves in the same way as the call to the wide-character, case-mapping function tolower().

A towctrans(wc, wctrans("toupper")) behaves in the same way as the call to the wide-character, case-mapping function towupper().

Return Value
The towctrans() function returns the mapped value of wc using the mapping that is described by desc.

Example that uses towctrans()
Related Information

- "wcrtans() —Get Handle for Character Mapping" on page 470
- "<wchar.h>" on page 17

### towlower() —towupper() — Convert Wide Character Case

**Format**

```c
#include <wctype.h>

wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `towupper()` function converts the lowercase character `wc` to the corresponding uppercase letter. The `towlower()` function converts the uppercase character `wc` to the corresponding lowercase letter.

**Note:** This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Return Value**

If `wc` is a wide character for which `iswupper()` (or `iswlower()`) is true and there is a corresponding wide character for which `iswlower()` (or `iswupper()`) is true, `towlower()` (or `towupper()`) returns the corresponding wide character. Otherwise, the argument is returned unchanged.

**Example that uses towlower() and towupper()**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
    char *alpha = "abcdefghijklmnopqrstuvwxyz";
    char *tocase[2] = {"toupper", "tolower"};
wchar_t *wcalpha;
    int i, j;
size_t alphalen;

    alphalen = strlen(alpha)+1;
wcalpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);
    mbstowcs(wcalpha, alpha, 2*alphalen);

    for (i=0; i<2; ++i) {
        printf("Input string: %ls\n", wcalpha);
        for (j=0; j<alphalen; j)
            printf("%ls\n", wcalpha);
    }
}
```
This example uses `tolower()` and `toupper()` to convert characters between 0 and 0x7f.

```
#include <wctype.h>
#include <stdio.h>

int main(void)
{
    wint_t w_ch;
    for (w_ch = 0; w_ch <= 0xff; w_ch++) {
        printf("toupper : %#04x %#04x, ", w_ch, towupper(w_ch));
        printf("tolower : %#04x %#04x\n", w_ch, towlower(w_ch));
    }
    return 0;
}
```

Related Information

- "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 160
- "tolower() – toupper() — Convert Character Case" on page 395
- "<wctype.h>" on page 17

__ultoa - Convert Unsigned Long Integer to String

```
#include <stdlib.h>
char *ultoa(unsigned long value, char *string, int radix);
```

Note: The _ultoa function is supported only for C++, not for C.

Language Level: Extension

Threading: Yes.

Description

_ultoa converts the digits of the given unsigned long value to a character string that ends with a null character and stores the result in string. The radix argument specifies the base of value; it must be in the range 2 to 36.

Note: The space allocated for string must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).
Return Value

_ultoa returns a pointer to string. There is no error return value.

When the string argument is NULL or the radix is outside the range 2 to 36, errno will be set to EINVAL.

Example that uses _ultoa()

This example converts the integer value 255 to a decimal, binary, and hexadecimal representation.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ultoa(255UL, buffer, 10);
    printf("The result of _ultoa(255) with radix of 10 is %s
", p);
    p = _ultoa(255UL, buffer, 2);
    printf("The result of _ultoa(255) with radix of 2
    is %s
", p);
    p = _ultoa(255UL, buffer, 16);
    printf("The result of _ultoa(255) with radix of 16 is %s
", p);
    return 0;
}
```

The output should be:

The result of _ultoa(255) with radix of 10 is 255
The result of _ultoa(255) with radix of 2 is 11111111
The result of _ultoa(255) with radix of 16 is ff

Related Information

- "_gcvt - Convert Floating-Point to String" on page 141
- "_itoa - Convert Integer to String" on page 163
- "_ltoa - Convert Long Integer to String" on page 177
- "<stdlib.h>" on page 15

ungetc() — Push Character onto Input Stream

Format

```c
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The ungetc() function pushes the unsigned character c back onto the given input stream. However, only one consecutive character is guaranteed to be pushed back onto the input stream if you call ungetc() consecutively. The stream must be open for reading. A subsequent read operation on the stream starts with c. The character c cannot be the EOF character.
Characters placed on the stream by ungetc() will be erased if fseek(), fsetpos(),
rewind(), or fflush() is called before the character is read from the stream.

Return Value

The ungetc() function returns the integer argument c converted to an unsigned
char, or EOF if c cannot be pushed back.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The ungetc() function is not supported for files opened with type=record.

Example that uses ungetc()

In this example, the while statement reads decimal digits from an input data
stream by using arithmetic statements to compose the numeric values of the
numbers as it reads them. When a non-digit character appears before the end of
the file, ungetc() replaces it in the input stream so that later input functions can
process it.

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    FILE *stream;
    int ch;
    unsigned int result = 0;
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';
    if (ch != EOF)
        ungetc(ch, stream);
    /* Put the nondigit character back */
    printf("The result is: \%d\n", result);
    if ((ch = getc(stream)) != EOF)
        printf("The character is: \%c\n", ch);
}
```

Related Information

- “getc() – getchar() — Read a Character” on page 142
- “fflush() — Write Buffer to File” on page 87
- “fseek() — fseeko() — Reposition File Position” on page 124
- “fsetpos() — Set File Position” on page 126
- “putc() – putchar() — Write a Character” on page 221
- “rewind() — Adjust Current File Position” on page 259
- “<stdio.h>” on page 14
ungetwc() — Push Wide Character onto Input Stream

Format
#include <wchar.h>
#include <stdio.h>
wint_t ungetwc(wint_t wc, FILE *stream);

Language Level: ANSI

Threadsafe: Yes.

Description

The ungetwc() function pushes the wide character wc back onto the input stream. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (on the stream) to a file positioning function (fseek(), fsetpos(), or rewind()) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged. There is always at least one wide character of push-back. If the value of wc is WEOF, the operation fails and the input stream is unchanged.

A successful call to the ungetwc() function clears the EOF indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back. However, only one consecutive wide character is guaranteed to be pushed back onto the input stream if you call ungetwc() consecutively.

For a text stream, the file position indicator is backed up by one wide character. This affects the ftell(), fflush(), fseek() (with SEEK_CUR), and fgetpos() function. For a binary stream, the position indicator is unspecified until all characters are read or discarded, unless the last character is pushed back, in which case the file position indicator is backed up by one wide character. This affects the ftell(), fseek() (with SEEK_CUR), fgetpos(), and fflush() function.

Note: This function is available when SYSIFCOPT(*IFSIO) is specified on the compilation command, and LOCALETYPE(*CLD) is not specified on the compilation command.

Return Value

The ungetwc() function returns the wide character pushed back after conversion, or WEOF if the operation fails.

Example that uses ungetwc()
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE     *stream;
  wint_t   wc;
  wint_t   wc2;
  unsigned int result = 0;

  if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
printf("Unable to open file.\n");
exit(EXIT_FAILURE);
}

while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
result = result * 10 + wc - L'0';

if (WEOF != wc)
ungetwc(wc, stream);  /* Push the nondigit wide character back */

/* get the pushed back character */
if (WEOF != (wc = fgetwc(stream))) {
  if (wc != wc2) {
    printf("Subsequent fgetwc does not get the pushed back character.\n");
    exit(EXIT_FAILURE);
  }
  printf("The digits read are '\%i'\n"
    "The character being pushed back is '\%lc', result, wc2);}
return 0;

/********************************************
Assuming the file ungetwc.dat contains:
12345ABCDE67890XYZ
The output should be similar to :
The digits read are '12345'
The character being pushed back is 'A'
********************************************/}

Related Information
• “fflush() — Write Buffer to File” on page 87
• “fseek() — fseeko() — Reposition File Position” on page 124
• “fsetpos() — Set File Position” on page 126
• “getwc() — Read Wide Character from Stream” on page 147
• “putwc() — Write Wide Character” on page 225
• “ungetc() — Push Character onto Input Stream” on page 399
• “<wchar.h>” on page 17

va_arg() – va_end() – va_start() — Access Function Arguments

Format
#include <stdarg.h>
var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);

Language Level: ANSI

Thraedsafe: Yes.

Description
The va_arg(), va_end(), and va_start() functions access the arguments to a
function when it takes a fixed number of required arguments and a variable
number of optional arguments. You declare required arguments as ordinary
parameters to the function and access the arguments through the parameter names.
va_start() initializes the arg_ptr pointer for subsequent calls to va_arg() and va_end().

The argument variable_name is the identifier of the rightmost named parameter in the parameter list (preceding , ...). Use va_start() before va_arg(). Corresponding va_start() and va_end() macros must be in the same function.

The va_arg() function retrieves a value of the given var_type from the location given by arg_ptr, and increases arg_ptr to point to the next argument in the list. The va_arg() function can retrieve arguments from the list any number of times within the function. The var_type argument must be one of int, long, decimal, double, struct, union, or pointer, or a typedef of one of these types.

The va_end() function is needed to indicate the end of parameter scanning.

Return Value

The va_arg() function returns the current argument. The va_end and va_start() functions do not return a value.

Example that uses va_arg() – va_end() – va_start()

This example passes a variable number of arguments to a function, stores each argument in an array, and prints each argument.
Related Information

- "vfprintf() — Print Argument Data to Stream"
- "printf() — Print Argument Data" on page 411
- "vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream" on page 407
- "vsprintf() — Print Argument Data to Buffer" on page 415
- "<stdarg.h>" on page 13

vfprintf() — Print Argument Data to Stream

Format

```
#include <stdio.h>
#include <stdarg.h>

int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

Language Level: ANSI

Threading Safe: Yes.

Description

```c
#include <stdio.h>
#include <stdarg.h>

int vout(int max, ...);

int main(void)
{
    vout(3, "Sat", "Sun", "Mon");
    printf("\n");
    vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

int vout(int max, ...)
{
    va_list arg_ptr;
    int args = 0;
    char *days[7];

    va_start(arg_ptr, max);
    while(args < max)
    {
        days[args] = va_arg(arg_ptr, char *);
        printf("Day:  %s  \n", days[args++]);
    }
    va_end(arg_ptr);
}
```

Output should be similar to:

```
Day:  Sat
Day:  Sun
Day:  Mon
Day:  Mon
Day:  Tues
Day:  Wed
Day:  Thurs
Day:  Fri
*/
```
The `vfprintf()` function formats and writes a series of characters and values to the output stream. The `vfprintf()` function works just like the `fprintf()` function, except that `arg_ptr` points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `fprintf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vfprintf()` function converts each entry in the argument list according to the corresponding format specifier in `format`. The `format` has the same form and function as the format string for the `printf()` function.

**Return Value**

If successful, `vfprintf()` returns the number of bytes written to `stream`. If an error occurs, the function returns a negative value.

**Example that uses vfprintf()**

This example prints out a variable number of strings to the file `myfile`.

```c
#include <stdarg.h>
#include <stdio.h>

void vout(FILE *stream, char *fmt, ...);
char fmt1[] = "%s  %s  %s\n";

int main(void)
{
    FILE *stream;
    stream = fopen("mylib/myfile", "w");
    vout(stream, fmt1, "Sat", "Sun", "Mon");
}

void vout(FILE *stream, char *fmt, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vfprintf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
}

/******************  Output should be similar to:  ******************
Sat Sun Mon
*/
```

**Related Information**

- "fprintf() — Write Formatted Data to a Stream” on page 108
- "printf() — Print Formatted Characters” on page 211
- "va_arg() – va_end() – va_start() — Access Function Arguments” on page 402
- "vprintf() — Print Argument Data” on page 411
- “vsprintf() — Print Argument Data to Buffer” on page 415
- “vwprintf() — Format Argument Data as Wide Characters and Print” on page 422
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 14
vfscanf() — Read Formatted Data

Format
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg_ptr);

Language Level: ANSI

Threadsafe: Yes.

Description
The vfscanf() function reads data from a stream into locations specified by a variable number of arguments. The vfscanf() function works just like the scanf() function except that arg_ptr points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start for each call. In contrast, the scanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format has the same form and function as the format string for the scanf() function.

Return Value
The vfscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example that uses vfscanf()
This example opens the file myfile for input, and then scans this file for a string, a long integer value, and a floating point value.
#include <stdio.h>
#include <stdarg.h>

int vread(FILE *stream, char *fmt, ...) {
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void) {
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, "%s", &s[0]);
    vread(stream, "%ld", &l);
    vread(stream, "%c", &c);
    vread(stream, "%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}

/*************** If myfile contains ******************
************** expected output is: ***************
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/

Related Information
- "fprintf() — Write Formatted Data to a Stream" on page 108
- "fscanf() — Read Formatted Data" on page 123
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "scanf() — Read Data" on page 313
- "sscanf() — Read Data" on page 339
- "swscanf() — Read Wide Character Data" on page 387
- "wscanf() — Read Data Using Wide-Character Format String" on page 482
- "<stdio.h>" on page 14

tfwprintf() — Format Argument Data as Wide Characters and Write to a Stream

Format
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
**Language Level:** ANSI

**Threading:** Yes.

**Description**

The `vfwprintf()` function is equivalent to the `fwprintf()` function, except that the variable argument list is replaced by `arg`, which the `va_start` macro (and possibly subsequent `va_arg` calls) will have initialized. The `vfwprintf()` function does not invoke the `va_end` macro.

Because the functions `vfwprintf()`, `vswprintf()`, and `vwprintf()` invoke the `va_arg` macro, the value of `arg` after the return is unspecified.

**Note:** This function is available when `SYSIFCOPT(*IFSIO)` is specified on the compilation command, and `LOCALETYPE(*CLD)` is not specified on the compilation command.

**Return Value**

The `vfwprintf()` function returns the number of wide characters transmitted. If an output error occurred, it returns a negative value.

**Example that uses `vfwprintf()`**

This example prints the wide character `a` to a file. The printing is done from the `vout()` function, which takes a variable number of arguments and uses `vfwprintf()` to print them to a file.

```c
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (FILE *stream, wchar_t *fmt, ...);

const char ifs_path [] = "tmp/myfile";

int main(void) {
    FILE *stream;
    wchar_t format [] = L"%lc";
    setlocale(LC_ALL, "POSIX");
    if ((stream = fopen (ifs_path, "w")) == NULL) {
        printf("Could not open file.\n");
        return (-1);
    }
    vout (stream, format, L'a');
    fclose (stream);

    //******************************************************************************
    // The contents of output file tmp/myfile.dat should be a wide char 'a' which in the "POSIX" locale
    // is '0081'x.
    //******************************************************************************
    return (0);
}
```

void vout (FILE *stream, wchar_t *fmt, ...)
{ va_list arg_ptr;
  va_start (arg_ptr, fmt);
  vfwprintf (stream, fmt, arg_ptr);
  va_end (arg_ptr);
}

Related Information
- "printf() — Print Formatted Characters” on page 211
- “fprintf() — Write Formatted Data to a Stream” on page 108
- “vfprintf() — Print Argument Data to Stream” on page 404
- “vprintf() — Print Argument Data” on page 411
- “btowc() — Convert Single Byte to Wide Character” on page 55
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 133
- “vfwprintf() — Format and Write Wide Characters to Buffer” on page 418
- “vwpprintf() — Format Argument Data as Wide Characters and Print” on page 422
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 14
- “<wchar.h>” on page 17

**vfwscanf() — Read Formatted Wide Character Data**

Format
```
#include <stdarg.h>
#include <stdio.h>
int vfwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);
```

Language Level: ANSI

Thsadsafe: Yes.

Description

The **vfwscanf()** function reads wide data from a stream into locations specified by a variable number of arguments. The vfwscanf() function works just like the **fwscanf() function** except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by *va_start* for each call. In contrast, the fscanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the fscanf() function.

**Note:** This function is available when SYSIFCOPT(*IFSIO) is specified on the compilation command, and LOCALETYPE(*CLD) is not specified on the compilation command.

Return Value
The `vfwscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

**Example that uses vfwscanf()**

This example opens the file `myfile` for input, and then scans this file for a string, a long integer value, and a floating point value.

```c
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(FILE *stream, wchar_t *fmt, ...) {
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfwscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void) {
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, L"%s", &s[0]);
    vread(stream, L"%ld", &l);
    vread(stream, L"%c", &c);
    vread(stream, L"%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}
```

Related Information

- "fscanf() — Read Formatted Data" on page 123
- "fwprintf() — Format Data as Wide Characters and Write to a Stream" on page 133
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "scanf() — Read Data" on page 313
- "sscanf() — Read Data" on page 339
- "swprintf() — Format and Write Wide Characters to Buffer" on page 385
- "swscanf() — Read Wide Character Data" on page 387
vprintf() — Print Argument Data

Format
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);

Language Level: ANSI

Threadsafe: Yes.

Description

The vprintf() function formats and prints a series of characters and values to stdout. The vprintf() function works just like printf() function except that arg_ptr points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start for each call. In contrast, the printf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

Return Value

If successful, the vprintf() function returns the number of bytes written to stdout. If an error occurs, the vprintf() function returns a negative value. The value of errno may be set to ETRUNC.

Example that uses vprintf()

This example prints out a variable number of strings to stdout.
```c
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);
char fmt1[] = "%s %s %s %s %s \n";

int main(void)
{
    FILE *stream;
    stream = fopen("mylib/myfile", "w");
    vout(fmt1, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(char *fmt, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vprintf(fmt, arg_ptr);
    va_end(arg_ptr);
}

/******************  Output should be similar to:  ******************
Mon Tues Wed Thurs Fri
*/

Related Information

- "printf() — Print Formatted Characters" on page 211
- "va_arg() – va_end() – va_start() — Access Function Arguments" on page 402
- "vfprintf() — Print Argument Data to Stream" on page 404
- "vsprintf() — Print Argument Data to Buffer" on page 415
- "<stdarg.h>" on page 13
- "<stdio.h>" on page 14

vscanf() — Read Formatted Data

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg_ptr);
```
Return Value

The vscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example that uses vscanf()

This example uses the vscanf() function to read an integer, a floating point value, a character, and a string from stdin and then displays these values.

```c
#include <stdio.h>
#include <stdarg.h>
int vread(char *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vscanf(fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    int i, rc;
    float fp;
    char c, s[81];
    printf("Enter an integer, a real number, a character and a string:
" "and a string : 
\n");
    rc = vread("%d %f %c %s", &i, &fp, &c, s);
    if (rc != 4)
        printf("Not all fields are assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}
```

************** If input is: 12 2.5 a yes, **************
************** then output should be similar to: **************
Enter an integer, a real number, a character and a string:
integer = 12
real number = 2.500000
character = a
string = yes
*/

Related Information
- "fscanf() — Read Formatted Data" on page 123
- "fwprintf() — Format Data as Wide Characters and Write to a Stream" on page 133
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "scanf() — Read Data" on page 313
- "sscanf() — Read Data" on page 339
- "swprintf() — Format and Write Wide Characters to Buffer" on page 385
- "swscanf() — Read Wide Character Data" on page 387
vsnprintf() — Print Argument Data to Buffer

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char *target-string, size_t n, const char *format, va_list arg_ptr);
```

Language Level: ANSI

Threading Safe: Yes.

Description

The `vsnprintf()` function formats and stores a series of characters and values in the buffer `target-string`. The `vsnprintf()` function works just like the `snprintf()` function, except that `arg_ptr` points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by the `va_start` function for each call. In contrast, the `snprintf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vsnprintf()` function converts each entry in the argument list according to the corresponding format specifier in `format`. The `format` has the same form and function as the format string for the `printf()` function.

Return Value

The `vsnprintf()` function returns the number of bytes that are written in the array, not counting the ending null character.

Example that uses `vsnprintf()`

This example assigns a variable number of strings to `string` and prints the resultant string.

```c
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...); 
char fmt1[] = "%s %s %s\n";

int main(void)
{
    char string[100];

    vout(string, fmt1, "Sat", "Sun", "Mon");
    printf("The string is: \%s\n", string);
}
```
void vout(char *string, char *fmt, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vsnprintf(string, 8, fmt, arg_ptr);
    va_end(arg_ptr);
}

/********************  Output should be similar to: ********************
The string is: Sat Su
*/

Related Information
- "printf() — Print Formatted Characters” on page 211
- "sprintf() — Print Formatted Data to Buffer” on page 335
- "snprintf() — Print Formatted Data to Buffer” on page 334
- "va_arg() – va_end() – va_start() — Access Function Arguments” on page 402
- "vfprintf() — Print Argument Data to Stream” on page 404
- “vsprintf() — Print Argument Data to Buffer"
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 14

vsprintf() — Print Argument Data to Buffer

Format
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format, va_list arg_ptr);

Language Level: ANSI

Threading: Yes.

Description

The vsprintf() function formats and stores a series of characters and values in the buffer target-string. The vsprintf() function works just like the sprintf() function except that arg_ptr points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by the va_start function for each call. In contrast, the sprintf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vsprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

Return Value

If successful, the vsprintf() function returns the number of bytes written to target-string. If an error occurs, the vsprintf() function returns a negative value.
Example that uses vsprintf()

This example assigns a variable number of strings to string and prints the resultant string.

```c
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...);
char fmt1[] = "%s %s %s\n";

int main(void)
{
    char string[100];
    vout(string, fmt1, "Sat", "Sun", "Mon");
    printf("The string is: %s\n", string);
}

void vout(char *string, char *fmt, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vsprintf(string, fmt, arg_ptr);
    va_end(arg_ptr);
}
```

/ ******************  Output should be similar to: ******************

The string is: Sat Sun Mon
/

Related Information

- "printf() — Print Formatted Characters" on page 211
- "sprintf() — Print Formatted Data to Buffer" on page 335
- "va_arg() - va_end() - va_start() — Access Function Arguments" on page 402
- "vfprintf() — Print Argument Data to Stream" on page 404
- "vprintf() — Print Argument Data" on page 411
- "vswprintf() — Format and Write Wide Characters to Buffer" on page 418
- "<stdarg.h>" on page 13
- "<stdio.h>" on page 14

vsscanf() — Read Formatted Data

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char *buffer, const char *format, va_list arg_ptr);
```

Language Level: ANSI

Threading: Yes.

Description

The vsscanf() function reads data from a buffer into locations specified by a variable number of arguments. The vsscanf() function works just like the scanf() function except that arg_ptr points to a list of arguments whose number...
can vary from call to call in the program. These arguments should be initialized by
`va_start` for each call. In contrast, the `sscanf()` function can have a list of
arguments, but the number of arguments in that list is fixed when you compile the
program.

Each argument must be a pointer to a variable with a type that corresponds to a
type specifier in format-string. The `format` has the same form and function as the
format string for the `scanf()` function.

**Return Value**

The `vsscanf()` function returns the number of fields that were successfully
converted and assigned. The return value does not include fields that were read
but not assigned. The return value is EOF for an attempt to read at end-of-file if no
conversion was performed. A return value of 0 means that no fields were assigned.

**Example that uses `vsscanf()`**

This example uses `vsscanf()` to read various data from the string `tokenstring` and
then displays that data.
```c
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>

int vread(const char *buffer, char *fmt, ...) {  
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vsscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void) {  
    char *tokenstring = "15 12 14";
    wchar_t * idestring = L"ABC Z";
    wchar_t ws[81];
    wchar_t wc;
    int i;
    float fp;
    char s[81];
    char c;
    /* Input various data */
    /* In the first invocation of vsscanf, the format string is */
    /* "%s %c%d%f", If there were no space between %s and %c, */
    /* vsscanf would read the first character following the */
    /* string, which is a blank space. */
    vread(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    vread((char *) idestring, "%S %C", ws,&wc);
    /* Display the data */
    printf("\nstring = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
    printf("wide-character string = %S\n", ws);
    printf("wide-character = %C\n", wc);
}
```

Related Information

- "fscanf() — Read Formatted Data” on page 123
- "fwscanf() — Read Data from Stream Using Wide Character” on page 137
- "scanf() — Read Data” on page 313
- "sscanf() — Read Data” on page 339
- "sprintf() — Print Formatted Data to Buffer” on page 335
- "<stdio.h>” on page 14
- "swscanf() — Read Wide Character Data” on page 387
- "wscanf() — Read Data Using Wide-Character Format String” on page 482

---

**vswprintf() — Format and Write Wide Characters to Buffer**

**Format**
#include <stdarg.h>
#include <wchar.h>

int vswprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, va_list argptr);

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `vswprintf()` function formats and stores a series of wide characters and values in the buffer `wcsbuffer`. The `vswprintf()` function works just like the `swprintf()` function, except that `argptr` points to a list of wide-character arguments whose number can vary from call to call. These arguments should be initialized by `va_start` for each call. In contrast, the `swprintf()` function can have a list of arguments, but the number of arguments in that list are fixed when you compile in the program.

The value `n` specifies the maximum number of wide characters to be written, including the ending null character. The `vswprintf()` function converts each entry in the argument list according to the corresponding wide-character format specifier in format. The format has the same form and function as the format string for the `printf()` function, with the following exceptions:

- `%c` (without an `l` prefix) converts an integer argument to wchar_t, as if by calling the `mbtowc()` function.
- `%lc` converts a wint_t to wchar_t.
- `%s` (without an `l` prefix) converts an array of multibyte characters to an array of wchar_t, as if by calling the `mbrtowc()` function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- `%ls` writes an array of wchar_t. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

**Note:** This function is available when SYSIFOPT(*IFSIO) is specified on the compilation command, and LOCALETYPE(*CLD) is **not** specified on the compilation command.

**Return Value**

The `vswprintf()` function returns the number of bytes written in the array, not counting the ending null wide character.

**Example that uses vswprintf()**

This example creates a function `vout()` that takes a variable number of wide-character arguments and uses `vswprintf()` to print them to `wcstr`.
Related Information

- "swprintf() — Format and Write Wide Characters to Buffer" on page 385
- "vfprintf() — Print Argument Data to Stream" on page 404
- "vprintf() — Print Argument Data" on page 411
- "vsprintf() — Print Argument Data to Buffer" on page 415
- "<stdarg.h>" on page 13
- "<wchar.h>" on page 17

---

**vswscanf() — Read Formatted Wide Character Data**

**Format**

```c
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

wchar_t *format3 = L"%ls  %d  %ls";
wchar_t *format5 = L"%ls  %d  %ls  %d  %ls";

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...) {
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vswprintf(wcs, n, fmt, arg_ptr);
    va_end(arg_ptr);
    return;
}

int main(void) {
    wchar_t wcstr[100];
    vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
    printf("%ls\n", wcstr);
    vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
    printf("%ls\n", wcstr);
    return 0;
}
```

The output should be similar to:

```
ONE  2  THREE
ONE  2  THREE  4  FIVE
```

---

**Related Information**

- "swprintf() — Format and Write Wide Characters to Buffer" on page 385
- "vfprintf() — Print Argument Data to Stream" on page 404
- "vprintf() — Print Argument Data" on page 411
- "vsprintf() — Print Argument Data to Buffer" on page 415
- "<stdarg.h>" on page 13
- "<wchar.h>" on page 17
can vary from call to call in the program. These arguments should be initialized by
\texttt{va\_start} for each call. In contrast, the \texttt{swscanf()} function can have a list of
arguments, but the number of arguments in that list is fixed when you compile the
program.

Each argument must be a pointer to a variable with a type that corresponds to a
type specifier in format-string. The \textit{format} has the same form and function as the
format string for the \texttt{swscanf()} function.

\textbf{Return Value}

The \texttt{vswscanf()} function returns the number of fields that were successfully
converted and assigned. The return value does not include fields that were read
but not assigned. The return value is EOF for an attempt to read at end-of-file if no
conversion was performed. A return value of 0 means that no fields were assigned.

\textbf{Example that uses vswscanf()}

This example uses the \texttt{vswscanf()} function to read various data from the string
\texttt{tokenstring} and then displays that data.

```c
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(const wchar_t *buffer, wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vswscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    wchar_t *tokenstring = L"15 12 14.000000";
    char s[81];
    char c;
    int i;
    float fp;

    /* Input various data */
    vread(tokenstring, L"%s %c%d%f", s, &c, &i, &fp);

    /* Display the data */
    printf("\nstring = %s
character = %c
integer = %d
floating-point number = %f\n", s, c, i, fp);
}
```

***************** Output should be similar to: *****************

string = 15
character = 1
integer = 2
floating-point number = 14.000000

******************************************************************/

\textbf{Related Information}

\begin{itemize}
\item \texttt{"fscanf() — Read Formatted Data" on page 123}
\end{itemize}
vprintf() — Format Argument Data as Wide Characters and Print

Format

```c
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);
```

Language Level: ANSI

Threadsafe: Yes.

Description

The `vprintf()` function is equivalent to the `wprintf()` function, except that the variable argument list is replaced by `arg`, which the `va_start` macro (and possibly subsequent `va_arg` calls) will have initialized. The `vprintf()` function does not invoke the `va_end` macro.

Note: This function is available when SYSIFCOPT(*IFSIO) is specified on the compilation command, and LOCALETYP(*CLD) is not specified on the compilation command.

Return Value

The `vprintf()` function returns the number of wide characters transmitted. If an output error occurred, the `vprintf()` returns a negative value.

Example that uses `vprintf()`

This example prints the wide character `a`. The printing is done from the `vout()` function, which takes a variable number of arguments and uses the `vprintf()` function to print them to stdout.
Related Information

- “printf() — Print Formatted Characters” on page 211
- “vfprintf() — Print Argument Data to Stream” on page 404
- “vprintf() — Print Argument Data” on page 411
- “btowc() — Convert Single Byte to Wide Character” on page 55
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 133
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 418
- “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 407
- “<stdarg.h>” on page 13
- “<wchar.h>” on page 17

---

**vwscanf() — Read Formatted Wide Character Data**

**Format**

```c
#include <wchar.h>
#include <va_arg.h>
int vwscanf(const wchar_t *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threading:** Yes.

**Description**

The `vwscanf()` function reads data from `stdin` into locations specified by a variable number of arguments. The `vwscanf()` function works just like the `scanf()` function.
function, except that arg_ptr points to a list of arguments whose number can vary
from call to call in the program. These arguments should be initialized by va_start
for each call. In contrast, the wscanf() function can have a list of arguments, but
the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a
type specifier in format-string. The format has the same form and function as the
format string for the wscanf() function.

Note: This function is available when SYSIFCOPT("IFSIO") is specified on the
 compilation command, and LOCALETYPE("CLD") is not specified on the
 compilation command.

Return Value

The vwscanf() function returns the number of fields that were successfully
converted and assigned. The return value does not include fields that were read
but not assigned. The return value is EOF for an attempt to read at end-of-file if no
conversion was performed. A return value of 0 means that no fields were assigned.

Example that uses vwscanf()

This example scans various types of data from stdin.
Related Information

- "fscanf() — Read Formatted Data" on page 123
- "scanf() — Read Data" on page 313
- "sscanf() — Read Data" on page 339
- "swscanf() — Read Wide Character Data" on page 387
- "fwscanf() — Read Data from Stream Using Wide Character" on page 137
- "wscanf() — Read Data Using Wide-Character Format String" on page 482
- "sprintf() — Print Formatted Data to Buffer" on page 335
- "<stdio.h>" on page 14


crtomb() — Convert a Wide Character to a Multibyte Character (Restartable)

Format

```c
#include <wchar.h>
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);
```

Language Level: ANSI

Threadsafe: Yes, except when `ps` is NULL.
Description

This function is the restartable version of the wctomb() function.

The wcrtomb() function converts a wide character to a multibyte character.

If s is a null pointer, the wcrtomb() function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described will be the initial conversion state.

If s is not a null pointer, the wcrtomb() function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by wc (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by s. At most MB_CUR_MAX bytes will be stored. If wc is a null wide character, the resulting state described will be the initial conversions state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, ps of type pointer to mbstate_t that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If ps is NULL, an internal static variable will be used to keep track of the conversion state. Using the internal static variable is not thread safe.

When the program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO), the behavior of wcrtomb() is affected by the LC_CTYPE category of the current locale. Remember that the CCSID of the locale should match the CCSID of your job. If the CCSID of the locale is a single-byte CCSID, the wide characters are converted to single-byte characters. Any wide character whose value is greater than 256 would be invalid when converting to a single-byte CCSID. When the CCSID is a multibyte CCSID, the wide characters are converted to multibyte characters.

When the program is compiled with a UNICODE LOCALETYPE and SYSIFCOPT(*IFSIO), the wide characters are assumed to be Unicode characters. These Unicode characters are converted to the CCSID of the locale associated with LC_CTYPE.

Return Value

If s is a null pointer, the wcrtomb() function returns the number of bytes needed to enter the initial shift state. The value returned will not be greater than that of the MB_CUR_MAX macro.

If s is not a null pointer, the wcrtomb() function returns the number of bytes stored in the array object (including any shift sequences) when wc is a valid wide character; otherwise (when wc is not a valid wide character), an encoding error occurs, the value of the macro EILSEQ shall be stored in errno and -1 will be returned, but the conversion state will be unchanged.

If a conversion error occurs, errno may be set to ECONVERT.

Examples that use wcrtomb()

This program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO):
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define STRLENGTH   10
#define LOCNAME     "qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc = 0x4171;
    wchar_t wc2 = 0x00C1;
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;

    /* In this first example we will convert a wide character */
    /* to a single byte character.  We first set the locale   */
    /* to a single byte locale.  We choose a locale with      */
    /* CCSID 37.  For single byte cases the state will always */
    /* remain in the initial state 0 */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc, &ps);

    /* In this case since wc > 256 hex, length is -1 and */
    /* errno is set to EILSEQ (3492) */
    printf("errno = %d, length = %d\n", errno, length);
    length = wcrtomb(string, wc2, &ps);

    /* In this case wc2 00C1 is converted to C1 */
    printf("string = %s\n", string);

    /* Now lets try a multibyte example.  We first must set the */
    /* locale to a multibyte locale.  We choose a locale with */
    /* CCSID 5026 */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc_string[0], &ps);

    /* The first character is < 256 hex so is converted to */
    /* single byte and the state is still the initial state 0 */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;
    length = wcrtomb(&string[sl], wc_string[1], &ps);

    /* The next character is > 256 hex so we get a shift out */
    /* 0x0e followed by the double byte character.  State is */
    /* changed to double byte state.  Length is 3. */
    printf("length = %d, state = %d\n", length, ps);
}
sl += length;
length = wcrtomb(&string[sl], wc_string[2], &ps);

/* The next character is > 256 hex so we get another double byte character. The state is left in double byte state. Length is 2. */
printf("length = %d, state = %d\n\n", length, ps);
sl += length;
length = wcrtomb(&string[sl], wc_string[3], &ps);

/* The next character is < 256 hex so we close off the double byte characters with a shift in 0x0f and then get a single byte character. Length is 2. */
/* The hex look at string would now be: */
/* C10E417141720FC2 */
/* You would need a device capable of displaying multibyte characters to see this string. */
printf("length = %d, state = %d\n\n", length, ps);
/* In the last example we will show what happens if NULL is passed in for the state. */
memset(string, '\0', STRLENGTH);
length = wcrtomb(string, wc_string[1], NULL);

/* The second character is > 256 hex so a shift out followed by the double character is produced but since the state is NULL, the double byte character is closed off with a shift in right away. So string we look like this: 0E41710F and length is 4 and the state is left in the initial state. */
printf("length = %d, state = %d\n\n", length, ps);
}
/* The output should look like this:
errno = 3492, length = -1
string = A
length = 1, state = 0
length = 3, state = 2
length = 2, state = 2
length = 2, state = 0
length = 4, state = 0 */

This program is compiled with LOCALETYPE(*LOCALEUCS2) and SYSIFCOPT(*IFSIO):
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define STRLENGTH 10
#define LOCNAME     "qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc = 0x4171;
    wchar_t wc2 = 0x0041;
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;
    wc_string[1] = 0xFF31;
    wc_string[2] = 0xFF32;
    wc_string[3] = 0x0042;
    wc_string[4] = 0x0000;
    /* In this first example we will convert a UNICODE character */
    /* to a single byte character. We first set the locale    */
    /* to a single byte locale. We choose a locale with      */
    /* CCSID 37. For single byte cases the state will always */
    /* remain in the initial state 0                      */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc2, &ps);
    /* In this case wc2 0041 is converted to C1       */
    /* 0041 is UNICODE A, C1 is CCSID 37          */
    printf("string = %s\n", string);
    /* Now lets try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026                          */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc_string[0], &ps);
    /* The first character UNICODE character is converted to a */
    /* single byte and the state is still the initial state 0 */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;
    length = wcrtomb(&string[sl], wc_string[1], &ps);
    /* The next UNICODE character is converted to a shift out */
    /* 0x0e followed by the double byte character. State is */
    /* changed to double byte state. Length is 3.          */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;
    length = wcrtomb(&string[sl], wc_string[2], &ps);
    /* The UNICODE character is converted to another */
    /* double byte character. The state is left in */
    /* double byte state. Length is 2. */
    printf("length = %d, state = %d\n", length, ps);
}
sl += length;

length = wcrtomb(&string[sl], wc_string[3], &ps);

/* The next UNICODE character converts to single byte so */
/* we close off the */
/* double byte characters with a shiftin 0x0f and then */
/* get a single byte character. Length is 2. */
/* The hex look at string would now be: */
/* C10E42D842D90FC2 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d, state = %d\n\n", length, ps);

} /* The output should look like this:

string = A
length = 1, state = 0
length = 3, state = 2
length = 2, state = 2
length = 2, state = 0 */

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 182
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 183
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 191
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 451
- “wctomb() — Convert Wide Character to Multibyte Character” on page 469
- “<wchar.h>” on page 17

**wcscat() — Concatenate Wide-Character Strings**

**Format**

```c
#include <wctype.h>
wchar_t *wcsat(wchar_t *string1, const wchar_t *string2);
```

**Language Level:** XPG4

**Threaftsae:** Yes.

**Description**

The wcsat() function appends a copy of the string pointed to by string2 to the end of the string pointed to by string1.
The `wcscat()` function operates on null-ended wchar_t strings. The string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed.

**Return Value**

The `wcscat()` function returns a pointer to the concatenated `string1`.

**Example that uses `wcscat()`**

This example creates the wide character string "computer program" using the `wcscat()` function.

```c
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
    wchar_t * string     = L" program";
    wchar_t * ptr;

    ptr = wcscat( buffer1, string );
    printf( "buffer1 = %ls\n", buffer1 );
}
```

/**
 ** Output should be similar to: **********
 ** buffer1 = computer program
 ** ************************************************************************** **

**Related Information**

- "`strcat()` — Concatenate Strings” on page 341
- "`strncat()` — Concatenate Strings” on page 361
- "`wcschr()` — Search for Wide Character”
- "`wcscmp()` — Compare Wide-Character Strings” on page 433
- "`wcscpy()` — Copy Wide-Character Strings” on page 435
- "`wcscln()` — Calculate Length of Wide-Character String” on page 440
- "`wcscscan()` — Concatenate Wide-Character Strings” on page 442
- "`<wcstr.h>” on page 17

**`wcschr()` — Search for Wide Character**

**Format**

```c
#include <wcstr.h>
wchar_t *wcschr(const wchar_t *string, wchar_t character);
```

**Language Level**: XPG4

**Threadsafe**: Yes.

**Description**
The wcschr() function searches the wide-character string for the occurrence of character. The character can be a wchar_t null character (\0); the wchar_t null character at the end of string is included in the search.

The wcschr() function operates on null-ended wchar_t strings. The string argument to this function should contain a wchar_t null character marking the end of the string.

Return Value

The wcschr() function returns a pointer to the first occurrence of character in string. If the character is not found, a NULL pointer is returned.

Example that uses wcschr()

This example finds the first occurrence of the character "p" in the wide-character string "computer program".

```c
#include <stdio.h>
#include <wcstr.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wchar_t ch = L'p';

    ptr = wcschr( buffer1, ch );
    printf( "The first occurrence of %lc in '%ls' is '%ls'\n", ch, buffer1, ptr );
}
```

/* Output should be similar to: */

The Related Information

- "strchr() — Search for Character” on page 342
- "strcspn() — Find Offset of First Character Match” on page 348
- "strupr() — Find Characters in String” on page 367
- "strchr() — Locate Last Occurrence of Character in String” on page 373
- "strrchr() — Find Characters in String” on page 374
- "wcscat() — Concatenate Wide-Character Strings” on page 430
- "wcssp() — Compare Wide-Character Strings” on page 433
- "wcscpy() — Copy Wide-Character Strings” on page 435
- "wcssp() — Find Offset of First Wide-Character Match” on page 436
- "wcslen() — Calculate Length of Wide-Character String” on page 440
- "wcsncmp() — Compare Wide-Character Strings” on page 443
- "wcsncpy() — Locate Wide Characters in String” on page 447
- "wcschr() — Locate Last Occurrence of Wide Character in String” on page 450
- "wcssp() — Find Offset of First Non-matching Wide Character” on page 453
- "wcswcs() — Locate Wide-Character Substring” on page 465
wcscmp() — Compare Wide-Character Strings

Format
#include <wcstr.h>
int wcscmp(const wchar_t *string1, const wchar_t *string2);

Language Level: ANSI

Threadsafe: Yes.

Description

The wcscmp() function compares two wide-character strings. The wcscmp() function operates on null-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed when a string is added to or copied.

Return Value

The wcscmp() function returns a value indicating the relationship between the two strings, as follows:

Value  Meaning
Less than 0  string1 less than string2
0  string1 identical to string2
Greater than 0  string1 greater than string2.

Example that uses wcscmp()

This example compares the wide-character string string1 to string2 using wcscmp().
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    int result;
    wchar_t string1[] = L"abcdef";
    wchar_t string2[] = L"abcdefg";
    result = wcscmp( string1, string2 );
    if ( result == 0 )
        printf( "%ls" is identical to "%ls\n", string1, string2 );
    else if ( result < 0 )
        printf( "%ls" is less than "%ls\n", string1, string2 );
    else
        printf( "%ls" is greater than "%ls\n", string1, string2 );
}

/**************************** Output should be similar to: ****************************
"abcdef" is less than "abcdefg"
*/
wcscoll() — Language Collation String Comparison

Format

#include <wchar.h>

int wcscoll(const wchar_t *wcs1, const wchar_t *wcs2);

Language Level: XPG4

Threadsafe: Yes.

Description

The wcscoll() function compares the wide-character strings pointed to by wcs1 and wcs2, both interpreted as appropriate to the LC_COLLATE category of the current locale (or the LC_UNI_COLLATE category if a UNICODE LOCALTYPE was specified).

Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

The behavior of this wide-character function is affected by the LC_COLLATE category of the current locale (or the LC_UNI_COLLATE category if a UNICODE LOCALTYPE was specified).

Return Value

The wcscoll() function returns an integer value indicating the relationship between the strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>wcs1 less than wcs2</td>
</tr>
<tr>
<td>0</td>
<td>wcs1 equivalent to wcs2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>wcs1 greater than wcs2</td>
</tr>
</tbody>
</table>
If wcs1 or wcs2 contain characters outside the domain of the collating sequence, the wcscoll() function sets errno to EINVAL. If an error occurs, the wcscoll() function sets errno to an nonzero value. There is no error return value.

Example that uses wcscoll()

This example uses the default locale.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    int result;
    wchar_t *wcs1 = L"first_wide_string";
    wchar_t *wcs2 = L"second_wide_string";

    result = wcscoll(wcs1, wcs2);
    if ( result == 0 )
        printf(""\%S" is identical to \"%S"\n", wcs1, wcs2);
    else if ( result < 0 )
        printf(""\%S" is less than \"%S"\n", wcs1, wcs2);
    else
        printf(""\%S" is greater than \"%S"\n", wcs1, wcs2);
}
```

Related Information
- “strcoll() — Compare Strings” on page 346
- “setlocale() — Set Locale” on page 322
- “<wchar.h>” on page 17

**wcscpy() — Copy Wide-Character Strings**

Format
```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The wcscpy() function copies the contents of string2 (including the ending wchar_t null character) into string1.

The wcscpy() function operates on null-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Only string2 needs to contain a null character. Boundary checking is not performed.

Return Value

The wcscpy() function returns a pointer to string1.

Example that uses wcscpy()
This example copies the contents of source to destination.

```c
#include <stdio.h>
#include <wchar.h>

#define SIZE   40

int main(void)
{
  wchar_t source[ SIZE ] = L"This is the source string";
  wchar_t destination[ SIZE ] = L"And this is the destination string";
  wchar_t * return_string;

  printf( "destination is originally = \"%ls\"\n", destination );
  return_string = wcscpy( destination, source );
  printf("After wcscpy, destination becomes \"%ls\"
", destination );
}
```

Related Information

- "strcpy() — Copy Strings" on page 347
- "strncpy() — Copy Strings" on page 364
- "wcscat() — Concatenate Wide-Character Strings" on page 430
- "wcschr() — Search for Wide Character" on page 431
- "wcsncmp() — Compare Wide-Character Strings" on page 433
- "wcsnep() — Find Offset of First Wide-Character Match"
- "wcslen() — Calculate Length of Wide-Character String" on page 440
- "wcsnep() — Copy Wide-Character Strings" on page 445
- "<wcstr.h>" on page 17

### wcscspn() — Find Offset of First Wide-Character Match

**Format**

```c
#include <wcstr.h>
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

**Language Level**: XPG4

**Threadsafe**: Yes.

**Description**

The `wcscspn()` function determines the number of wchar_t characters in the initial segment of the string pointed to by `string1` that do not appear in the string pointed to by `string2`.

The `wcscspn()` function operates on null-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

**Return Value**
The `wcscspn()` function returns the number of wchar_t characters in the segment.

**Example that uses `wcscspn()`**

This example uses `wcscspn()` to find the first occurrence of any of the characters a, x, l, or e in string.

```c
#include <stdio.h>
#include <wctype.h>

#define SIZE 40

int main(void)
{
    wchar_t string[SIZE] = L"This is the source string";
    wchar_t *substring = L"axle";

    printf( "The first %i characters in the string "%ls" are not in the "string "%ls" \n", wcscspn( string, substring),
            string, substring );
}
```

/* Output should be similar to: */

The first 10 characters in the string "This is the source string" are not in the string "axle"

**Related Information**

- "`strcspn()` — Find Offset of First Character Match” on page 348
- "`strspn()` — Find Offset of First Non-matching Character” on page 374
- "`wcschr()` — Search for Wide Character” on page 431
- "`wcscmp()` — Compare Wide-Character Strings” on page 433
- "`wcschr()` — Search for Wide Character” on page 431
- "`wcsftime()` — Convert to Formatted Date and Time” on page 440
- "`wcscspn()` — Find Offset of First Non-matching Wide Character” on page 453
- "`wcscs()` — Locate Wide-Character Substring” on page 465
- "`<wctype.h>`” on page 17

---

**wcsftime() — Convert to Formatted Date and Time**

**Format**

```c
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
    const wchar_t *format, const struct tm *timeptr);
```

**Language Level**: ANSI

**Threadsafe**: Yes.

**Description**

The `wcsftime()` function converts the time and date specification in the `timeptr` structure into a wide-character string. It then stores the null-ended string in the array pointed to by `wdest` according to the format string pointed to by `format`. The
`maxsize` value specifies the maximum number of wide characters that can be copied into the array. This function is equivalent to `strftime()`, except that it uses wide characters.

The `wcsftime()` function works just like the `strftime()` function, except that it uses wide characters. The format string is a wide-character character string that contains:

- Conversion-specification characters.
- Ordinary wide characters, which are copied into the array unchanged.

**Note:** When `wcsftime()` is compiled with a UNICODE LOCALETYPE, both the `wdest` parameter and the `format` parameter are expected to be Unicode-encoded strings. Otherwise, they are expected to be wide EBCDIC strings.

This function is not available when LOCALETYPE("CLD") is specified on the compilation command.

This function uses the time structure pointed to by `timeptr`, and if the specifier is locale sensitive, then it will also use the LC_TIME category of the current locale to determine the appropriate replacement value of each valid specifier. The time structure pointed to by `timeptr` is usually obtained by calling the `gmtime()` or `localtime()` function.

**Return Value**

If the total number of wide characters in the resulting string, including the ending null wide character, does not exceed `maxsize`, `wcsftime()` returns the number of wide characters placed into `wdest`, not including the ending null wide character. Otherwise, the `wcsftime()` function returns 0 and the contents of the array are indeterminate.

If a conversion error occurs, `errno` may be set to ECONVERT.

**Example that uses wcsftime()**

This example obtains the date and time using `localtime()`, formats the information with the `wcsftime()`, and prints the date and time.

```c
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void) {
    struct tm *timeptr;
    wchar_t dest[100];
    time_t temp;
    size_t rc;

    temp = time(NULL);
    timeptr = localtime(&temp);
    rc = wcsftime(dest, sizeof(dest), L" Today is %A, "
                  L" %b %d.\n Time: %I:%M %p", timeptr);
    printf("%d characters placed in string to make:\n\n", rc, dest);
    return 0;
}
```

The output should be similar to:

```
438
```

ILE C/C++ Run-Time Library Functions V5R3
Related Information

- “gmtime() — Convert Time” on page 151
- “localtime() — Convert Time” on page 173
- “strftime() — Convert Date/Time to String” on page 354
- “strptime() — Convert String to Date/Time” on page 369
- “<wchar.h>” on page 17

__wcsicmp() — Compare Wide Character Strings without Case Sensitivity

Format

```c
#include <wchar.h>
int __wcsicmp(const wchar_t *string1, const wchar_t *string2);
```

Language Level: Extension

Threading: Yes.

Description

The __wcsicmp() function compares string1 and string2 without sensitivity to case. All alphabetic wide characters in string1 and string2 are converted to lowercase before comparison. The function operates on null terminated wide character strings. The string arguments to the function are expected to contain a wchar_t null character (L'\0') marking the end of the string.

Return Value

The __wcsicmp() function returns a value indicating the relationship between the two strings as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 equivalent to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>

Example that uses __wcsicmp()

This example uses __wcsicmp() to compare two wide character strings.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING";
    // ...
 wchar_t *str2 = L"string";
  int result;

  result = __wcsicmp(str1, str2);

  if (result == 0)
    printf("Strings compared equal.\n");
  else if (result < 0)
    printf("%ls is less than %ls.\n", str1, str2);
  else
    printf("%ls is greater than %ls.\n", str1, str2);

  return 0;

} /* The output should be similar to: */

Strings compared equal.

******************/

Related Information

- "strcmp() — Compare Strings" on page 343
- "strncmp() — Compare Strings" on page 362
- "wcsat() — Concatenate Wide-Character Strings" on page 430
- "wcschr() — Search for Wide Character" on page 431
- "wcsnscn() — Find Offset of First Wide-Character Match" on page 436
- "wcslen() — Calculate Length of Wide-Character String"
- "wcsncmp() — Compare Wide-Character Strings" on page 443
- "__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity" on page 446
- "<wchar.h>" on page 17

**wcslen() — Calculate Length of Wide-Character String**

**Format**

```c
#include <wctype.h>
size_t wcslen(const wchar_t *string);
```

**Language Level:** XPG4

**Threading Safe:** Yes.

**Description**

The wcslen() function computes the number of wide characters in the string pointed to by `string`.

**Note:** This function is not available when LOCALETYPE("CLD) is specified on the compilation command.

**Return Value**

The wcslen() function returns the number of wide characters in `string`, excluding the ending wchar_t null character.

**Example that uses wcslen()**
This example computes the length of the wide-character string string.

```c
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t * string = L"abcdef";
    printf( "Length of \"%ls\" is %lu\n", string, wcslen( string ));
}

/***************  Output should be similar to:  ***************
Length of "abcdef" is 6
*/

Related Information
- "mblen() — Determine Length of a Multibyte Character" on page 182
- "strlen() — Determine String Length" on page 358
- "wcsncat() — Concatenate Wide-Character Strings" on page 442
- "wcsncmp() — Compare Wide-Character Strings" on page 443
- "wcsncpy() — Copy Wide-Character Strings" on page 445
- "<wcstr.h>" on page 17

wcslocaleconv() — Retrieve Wide Locale Information

Format
#include <locale.h>
struct wcslocale *wcslocaleconv(void);

Language Level: Extended

Threaadsafe: Yes.

Description

The wcslocaleconv() function is the same as the localeconv function, except that it returns a pointer to a wcslocale structure, which is the wide version of a lconv structure. These elements are determined by the LC_UNI_MONETARY and LC_UNI_NUMERIC categories of the current locale.

Note: This function is available only when LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command.

Return Value

The wcslocaleconv() function returns a pointer to a wcslocale structure.

Example that uses wcslocaleconv()

This example prints out the Unicode currency symbol for a French locale.
wcsncat() — Concatenate Wide-Character Strings

**Format**

```c
#include <wcsstr.h>
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level**: XPG4

**Threading Safe**: Yes

**Description**

The `wcsncat()` function appends up to `count` wide characters from `string2` to the end of `string1`, and appends a wchar_t null character to the result.

The `wcsncat()` function operates on null-ending wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

**Return Value**

The `wcsncat()` function returns `string1`.

**Example that uses `wcsncat()`**

This example demonstrates the difference between the `wcscat()` and `wcsncat()` functions. The `wcscat()` function appends the entire second string to the
first; the wcsncat() function appends only the specified number of characters in the second string to the first.

```
#include <stdio.h>
#include <wstr.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
    wchar_t * ptr;
    /* Call wcscat with buffer1 and " program" */
    ptr = wcscat(buffer1, L" program");
    printf("wcscat : buffer1 = "%ls\n", buffer1);
    /* Reset buffer1 to contain just the string "computer" again */
    memset(buffer1, L'\0', sizeof(buffer1));
    ptr = wcscpy(buffer1, L"computer");
    /* Call wcscat with buffer1 and " program" */
    ptr = wcscat(buffer1, L" program", 3);
    printf("wcscat : buffer1 = "%ls\n", buffer1);
}
/****************  Output should be similar to: ***************
wcscat : buffer1 = "computer program"
wcsncat: buffer1 = "computer pr"
*/
```

**Related Information**
- "strcat() — Concatenate Strings" on page 341
- "strncpy() — Concatenate Strings" on page 361
- "wcscat() — Concatenate Wide-Character Strings" on page 430
- "wcsncmp() — Compare Wide-Character Strings"
- "wcsncpy() — Copy Wide-Character Strings" on page 445
- "<wcsstr.h>" on page 17

---

**wcsncmp() — Compare Wide-Character Strings**

**Format**
```
#include <wstr.h>
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**
The wcsncmp() function compares up to count wide characters in string1 to string2.

The wcsncmp() function operates on null-ended wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.
Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

Return Value

The wcsncmp() function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 identical to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>

Example that uses wcsncmp()

This example demonstrates the difference between the wcscmp() function, which compares the entire strings, and the wcsncmp() function, which compares only a specified number of wide characters in the strings.
#include <stdio.h>
#include <wcstr.h>

#define SIZE 10

int main(void)
{
    int result;
    int index = 3;
    wchar_t *buffer1[SIZE] = L"abcdefg";
    wchar_t *buffer2[SIZE] = L"abcfg";
    void print_result( int, wchar_t *, wchar_t * );
    result = wcscmp( buffer1, buffer2 );
    printf( "Comparison of each character\n" );
    printf( " wcscmp: " );
    print_result( result, buffer1, buffer2 );
    result = wcsncmp( buffer1, buffer2, index);
    printf( "Comparison of only the first %i characters\n", index );
    printf( " wcsncmp: " );
    print_result( result, buffer1, buffer2 );
}

void print_result( int res, wchar_t * p_buffer1, wchar_t * p_buffer2 )
{
    if ( res == 0 )
        printf( "%ls" is identical to %ls
", p_buffer1, p_buffer2 );
    else if ( res < 0 )
        printf( "%ls" is less than %ls
", p_buffer1, p_buffer2 );
    else
        printf( "%ls" is greater than %ls
", p_buffer1, p_buffer2 );
}

/**** Similar output ***/

Related Information

- "strcmp() — Compare Strings" on page 343
- "strcoll() — Compare Strings" on page 346
- "strncmp() — Compare Strings" on page 362
- "wcsncmp() — Compare Wide-Character Strings" on page 433
- "wcsncat() — Concatenate Wide-Character Strings" on page 442
- "wcscpy() — Copy Wide-Character Strings"
- "<wcstr.h>" on page 17

__wcsncpy__() — Copy Wide-Character Strings

Format

```
#include <wcstr.h>
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);
```

Language Level: XPG4

Threading: Yes,
Description

The wcsncpy() function copies up to count wide characters from string2 to string1. If string2 is shorter than count characters, string1 is padded out to count characters with wchar_t null characters.

The wcsncpy() function operates on null-ended wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Only string2 needs to contain a null character.

Return Value

The wcsncpy() returns a pointer to string1.

Related Information

- "strcpy() — Copy Strings" on page 347
- "strncpy() — Copy Strings" on page 364
- "wcsncpy() — Copy Wide-Character Strings" on page 435
- "wcsncat() — Concatenate Wide-Character Strings” on page 442
- “wcsncmp() — Compare Wide-Character Strings” on page 443
- "<wchstr.h>” on page 17

__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity

Format

#include <wchar.h>
int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);

Language Level: Extension

Thesafe: Yes.

Description

The __wcsnicmp() function compares up to count characters of string1 and string2 without sensitivity to case. All alphabetic wide characters in string1 and string2 are converted to lowercase before comparison.

The __wcsnicmp() function operates on null terminated wide character strings. The string arguments to the function are expected to contain a wchar_t null character (L'\0') marking the end of the string.

Return Value

The __wcsnicmp() function returns a value indicating the relationship between the two strings, as follows:

Table 11. Return values of __wcsicmp()

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 equivalent to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>
Example that uses \_wcsnicmp()

This example uses \_wcsnicmp() to compare two wide character strings.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING ONE";
    wchar_t *str2 = L"string TWO";
    int result;

    result = \_wcsnicmp(str1, str2, 6);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\%ls\" is less than \%ls\".\n", str1, str2);
    else
        printf("\%ls\" is greater than \%ls\".\n", str1, str2);
    return 0;
}

/******** The output should be similar to: **************
Strings compared equal.

******************************************************************************

Related Information

- "strcmp() — Compare Strings" on page 343
- "strncmp() — Compare Strings" on page 362
- “wcsstr() — Concatenate Wide-Character Strings” on page 430
- “wcschr() — Search for Wide Character” on page 431
- “wcscspn() — Find Offset of First Wide-Character Match” on page 436
- “wcslen() — Calculate Length of Wide-Character String” on page 440
- “wcsncmp() — Compare Wide-Character Strings” on page 443
- “\_wcsicmp() — Compare Wide Character Strings without Case Sensitivity” on page 439
- “<wchar.h>” on page 17

wcspbrk() — Locate Wide Characters in String

Format

```c
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The wcspbrk() function locates the first occurrence in the string pointed to by \texttt{string1} of any wide character from the string pointed to by \texttt{string2}.
Return Value

The wcspbrk() function returns a pointer to the character. If string1 and string2 have no wide characters in common, the wcspbrk() function returns NULL.

Example that uses wcspbrk()

This example uses wcspbrk() to find the first occurrence of either "a" or "b" in the array string.

```
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t *result;
    wchar_t * string = L"The Blue Danube";
    wchar_t *chars = L"ab";
    result = wcspbrk( string, chars);
    printf("The first occurrence of any of the characters \"%ls\" in "
           "\"%ls\" is \"%ls\"\n", chars, string, result);
}
```

The first occurrence of any of the characters "ab" in "The Blue Danube" is "anube"

Related Information
- "strchr() — Search for Character" on page 342
- "strcspn() — Find Offset of First Character Match" on page 348
- "strpbrk() — Find Characters in String" on page 367
- "strspn() — Find Offset of First Non-matching Character" on page 374
- "wcschr() — Search for Wide Character" on page 431
- "wcscmp() — Compare Wide-Character Strings" on page 433
- "wcsncmp() — Compare Wide-Character Strings" on page 443
- "wcsstr() — Locate Last Occurrence of Wide Character in String" on page 450
- "wcsstr() — Locate Wide-Character Substring" on page 465
- "<wcstr.h>" on page 17
The wcsptime() function converts the wide character string pointed to by buf to values that are stored in the tm structure pointed to by tm, using the format specified by format. This function is equivalent to strftime(), except that it uses wide characters.

See "strftime()—Convert String to Date/Time" on page 369 for a description of the format string.

Note: This function is available only when LOCALETYPE("LOCALEUTF") is specified on the compilation command.

Return Value

On successful completion, the wcsptime() function returns a pointer to the character following the last wide character parsed. Otherwise, a null pointer is returned. The value of errno may be set to ECONVERT (conversion error).

Example that uses wcsptime()

```c
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
    wchar_t buf[100];
    time_t t;
    struct tm *timeptr,result;

    t = time(NULL);
    timeptr = localtime(&t);
    wcsftime(buf, 100, L"%a %m/%d/%Y %r", timeptr);

    if (wcsptime(buf, L"%a %m/%d/%Y %r", &result) == NULL)
        printf("wcsptime failed\n");
    else
    {
        printf("tm_hour:  %d\n",result.tm_hour);
        printf("tm_min:  %d\n",result.tm_min);
        printf("tm_sec:  %d\n",result.tm_sec);
        printf("tm_mon:  %d\n",result.tm_mon);
        printf("tm_mday:  %d\n",result.tm_mday);
        printf("tm_year:  %d\n",result.tm_year);
        printf("tm_yday:  %d\n",result.tm_yday);
        printf("tm_wday:  %d\n",result.tm_wday);
    }

    return 0;
}
```

The output should be similar to:

```
tm_hour:  14
tm_min: 25
tm_sec: 34
tm_mon: 7
tm_mday: 19
tm_year: 103
tm_yday: 230
tm_wday: 2
```

Related Information
wcsrchr() — Locate Last Occurrence of Wide Character in String

Format

```c
#include <wctype.h>
wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

Language Level: ANSI

Threadsafe: Yes,

Description

The wcsrchr() function locates the last occurrence of `character` in the string pointed to by `string`. The ending wchar_t null character is considered to be part of the string.

Return Value

The wcsrchr() function returns a pointer to the character, or a NULL pointer if `character` does not occur in the string.

Example that uses wcsrchr()

This example compares the use of wcschr() and wcsrchr(). It searches the string for the first and last occurrence of `p` in the wide character string.
Related Information

- "strchr() — Search for Character" on page 342
- "strrchr() — Locate Last Occurrence of Character in String" on page 373
- "strcspn() — Find Offset of First Character Match" on page 348
- "strspn() — Find Offset of First Non-matching Character" on page 374
- "wcschr() — Search for Wide Character" on page 431
- "wcscmp() — Compare Wide-Character Strings" on page 433
- "wcscspn() — Find Offset of First Wide-Character Match" on page 436
- "wcsncmp() — Compare Wide-Character Strings" on page 443
- "wcswcs() — Locate Wide-Character Substring" on page 465
- "wcspbrk() — Locate Wide Characters in String" on page 447
- "<wcstr.h>" on page 17

**wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)**

Format

```c
#include <wchar.h>
size_t wcsrtombs (char *dst, const wchar_t **src, size_t len,
                 mbstate_t *ps);
```

Language Level: ANSI

Threadsafe: Yes, if the fourth parameter, *ps*, is not NULL.

Description

This function is the restartable version of wcstombs().
The wcsrtombs() function converts a sequence of wide characters from the array indirectly pointed to by src into a sequence of corresponding multibyte characters that begins in the shift state described by ps, which, if dst is not a null pointer, are then stored into the array pointed to by dst. Conversion continues up to and including the ending null wide character, which is also stored. Conversion will stop earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if dst is not a null pointer) when the next multibyte element would exceed the limit of len total bytes to be stored into the array pointed to by dst. Each conversion takes place as if by a call to wcrtomb().

If dst is not a null pointer, the object pointed to by src will be assigned either a null pointer (if conversion stopped due to reaching a ending null character) or the address of the code just past the last wide character converted. If conversion stopped due to reaching a ending null wide character, the resulting state described will be the initial conversion state.

Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

Return Value

If the first code is not a valid wide character, an encoding error will occur. wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t) -1, but the conversion state will be unchanged. Otherwise it returns the number of bytes in the resulting multibyte character sequence, which is the same as the number of array elements changed when dst is not a null pointer.

If a conversion error occurs, errno may be set to ECONVERR.

Example that uses wcsrtombs()
```c
#include <stdio.h>
#include <wchar.h>
#include <string.h>
#define SIZE 20

int main(void)
{
    char     dest[SIZE];
    wchar_t *wcs = L"string";
    wchar_t *ptr;
    size_t   count = SIZE;
    size_t   length;
    mbstate_t ps = 0;

    ptr = (wchar_t *) wcs;
    length = wcsrtoemb(dest, ptr, count, &ps);
    printf("%d characters were converted.\n", length);
    printf("The converted string is " L"string"\n", dest);

    /* Reset the destination buffer */
    memset(dest, '\0', sizeof(dest));

    /* Now convert only 3 characters */
    ptr = (wchar_t *) wcs;
    length = wcsrtoemb(dest, ptr, 3, &ps);
    printf("%d characters were converted.\n", length);
    printf("The converted string is " L"str"\n", dest);
}
```

/* Output should be similar to: */
6 characters were converted.
The converted string is "string"

3 characters were converted.
The converted string is "str"

Related Information
- "mblen() — Determine Length of a Multibyte Character" on page 182
- "mbrlen() — Determine Length of a Multibyte Character (Restartable)" on page 183
- "mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)" on page 186
- "mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)" on page 191
- "wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)" on page 425
- "wcstombs() — Convert Wide-Character String to Multibyte String" on page 460
- "<wchar.h>" on page 17
Description

The wcsppn() function computes the number of wide characters in the initial segment of the string pointed to by string1, which consists entirely of wide characters from the string pointed to by string2.

Return Value

The wcsppn() function returns the number of wide characters in the segment.

Example that uses wcsppn()

This example finds the first occurrence in the array string of a wide character that is not an a, b, or c. Because the string in this example is cabbage, the wcsppn() function returns 5, the index of the segment of cabbage before a character that is not an a, b, or c.

```c
#include <stdio.h>
#include <wcstr.h>

int main(void)
{
    wchar_t * string = L"cabbage";
    wchar_t * source = L"abc";
    int index;

    index = wcsppn( string, L"abc" );
    printf( "The first %d characters of \"%ls\" are found in \"%ls\"\n",
            index, string, source );
}
```

*************** Output should be similar to: ***************
The first 5 characters of "cabbage" are found in "abc" */

Related Information

- "strchr() — Search for Character” on page 342
- "strcspn() — Find Offset of First Character Match” on page 348
- "strpbrk() — Find Characters in String” on page 367
- "strchr() — Locate Last Occurrence of Character in String” on page 373
- "strlen() —Find Offset of First Non-matching Character” on page 374
- "wcsat() — Concatenate Wide-Character Strings” on page 430
- "wcschr() — Search for Wide Character” on page 431
- "wcsncmp() — Compare Wide-Character Strings” on page 433
- "wcsppn() — Find Offset of First Wide-Character Match” on page 436
- "wcssppn() — Locate Wide Characters in String” on page 447
- "wcsschr() — Locate Last Occurrence of Wide Character in String” on page 450
- "wcsppn() — Find Offset of First Non-matching Wide Character” on page 453
- "wcsswpn() — Locate Wide-Character Substring” on page 465
- "<wcstr.h>” on page 17
wcsstr() — Locate Wide-Character Substring

**Format**

#include <wchar.h>

wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The wcsstr() function locates the first occurrence of wcs2 in wcs1.

**Return Value**

The wcsstr() function returns a pointer to the beginning of the first occurrence of wcs2 in wcs1. If wcs2 does not appear in wcs1, the wcsstr() function returns NULL. If wcs2 points to a wide-character string with zero length, it returns wcs1.

**Example that uses wcsstr()**

This example uses the wcsstr() function to find the first occurrence of "hay" in the wide-character string "needle in a haystack".

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs1 = L"needle in a haystack";
    wchar_t *wcs2 = L"hay";
    printf("result: \"%ls\n", wcsstr(wcs1, wcs2));
    return 0;
}
```

The output should be similar to:

```
result: "haystack"
```

**Related Information**

- "strstr() — Locate Substring" on page 375
- "wcschr() — Search for Wide Character" on page 431
- "wcsrchr() — Locate Last Occurrence of Wide Character in String" on page 450
- "wcswcs() — Locate Wide-Character Substring" on page 465
- "<wchar.h>" on page 17

wcstod() — Convert Wide-Character String to Double

**Format**

#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);

**Language Level:** XPG4
Threadsafe: Yes.

Description

The `wcstod()` function converts the initial portion of the wide-character string pointed to by `nptr` to a double value. The `nptr` parameter points to a sequence of characters that can be interpreted as a numerical value of type double. The `wcstod()` function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the `wchar_t` null character at the end of the string.

The behavior of the `wcstod()` function is affected by the LC_CTYPE category of the current locale.

**Note:** This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

If the program is compiled with a UNICODE LOCALETYPE, then the wide characters being converted are assumed to be Unicode characters.

Return Value

The `wcstod()` function returns the converted double value. If no conversion could be performed, the `wcstod()` function returns 0. If the correct value is outside the range of representable values, the `wcstod()` function returns +HUGE_VAL or -HUGE_VAL (according to the sign of the value), and sets `errno` to ERANGE. If the correct value would cause underflow, the `wcstod()` function returns 0 and sets `errno` to ERANGE. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The value of `errno` may be set to ERANGE, range error.

Example that uses `wcstod()`

This example uses the `wcstod()` function to convert the string `wcs` to a floating-point value.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t *wcs = L"3.1415926This stopped it";
  wchar_t *stopwcs;

  printf("wcs = \"%ls\"
      \"\n", wcs);
  printf("%f\n", wcstod(wcs, &stopwcs));
  printf(" Stop scanning at \"%ls\"
      \"\n", stopwcs);
  return 0;

  /******************************************************************************************
   The output should be similar to:
  wcstod = 3.1415926This stopped it
     3.141593
     Stop scanning at "This stopped it"
  ******************************************************************************************/
}
```
wcstok() — Tokenize Wide-Character String

Format
#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);

Language Level: ANSI

Description
The wcstok() function reads wcs1 as a series of zero or more tokens and wcs2 as the set of wide characters serving as delimiters for the tokens in wcs1. A sequence of calls to the wcstok() function locates the tokens inside wcs1. The tokens can be separated by one or more of the delimiters from wcs2. The third argument points to a wide-character pointer that you provide where the wcstok() function stores information necessary for it to continue scanning the same string.

When the wcstok() function is first called for the wide-character string wcs1, it searches for the first token in wcs1, skipping over leading delimiters. The wcstok() function returns a pointer to the first token. To read the next token from wcs1, call the wcstok() function with NULL as the first parameter (wcs1). This NULL parameter causes the wcstok() function to search for the next token in the previous token string. Each delimiter is replaced by a null character to end the token.

The wcstok() function always stores enough information in the pointer ptr so that subsequent calls, with NULL as the first parameter and the unmodified pointer value as the third, will start searching right after the previously returned token. You can change the set of delimiters (wcs2) from call to call.

Return Value
The wcstok() function returns a pointer to the first wide character of the token, or a null pointer if there is no token. In later calls with the same token string, the wcstok() function returns a pointer to the next token in the string. When there are no more tokens, the wcstok() function returns NULL.

Example that uses wcstok()

This example uses the wcstok() function to locate the tokens in the wide-character string str1.
#include <stdio.h>
#include <wchar.h>
int main(void)
{
    static wchar_t str1[] = L"?a??b,,,#c";
    static wchar_t str2[] = L"\t \t";
    wchar_t *t, *ptr1, *ptr2;

    t = wcstok(str1, L"?", &ptr1);   /* t points to the token L"a" */
    printf("t = '%ls'
", t);
    t = wcstok(NULL, L"", &ptr1);   /* t points to the token L"?b"*/
    printf("t = '%ls'
", t);
    t = wcstok(str2, L" \t", &ptr2); /* t is a null pointer        */
    printf("t = '%ls'
", t);
    t = wcstok(NULL, L"#", &ptr1);  /* t points to the token L"c" */
    printf("t = '%ls'
", t);
    t = wcstok(NULL, L"?", &ptr1);     /* t is a null pointer          */
    printf("t = '%ls'
", t);
    return 0;
}

Related Information

- "strtok() — Tokenize String" on page 378
- "<wchar.h>" on page 17

wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer

Format (wcstol())
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

Format (wcstoll())
#include <wchar.h>
long long int wcstoll(const wchar_t *nptr, wchar_t **endptr, int base);

Language Level: ANSI

Threadsafe: Yes.

Description

The wcstol() function converts the initial portion of the wide-character string pointed to by *nptr to a long integer value. The *nptr parameter points to a sequence of wide characters that can be interpreted as a numerical value of type long int. The wcstol() function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The behavior of the wcstol() function is affected by the LC_CTYPE category of the current locale.
Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

If a program is compiled with a UNICODE LOCALETYPE, the wide characters being converted are assumed to be Unicode characters.

The wcstoll() subroutine converts a wide-character string to a long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the iswspace subroutine). Any non-space character signifies the start of a subject string that may form a long long int in the radix specified by the base parameter. The subject sequence is defined to be the longest initial substring that is a long long int of the expected form.

If the value of the endptr parameter is not null, then a pointer to the character that ended the scan is stored in endptr. If a long long integer cannot be formed, the value of the endptr parameter is set to that of the nptr parameter.

If the base parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing a long long integer whose radix is specified by the base parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base parameter are permitted. If the base parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the base parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

Return Value

The wcstol() function returns the converted long integer value. If no conversion could be performed, the wcstol() function returns 0. If the correct value is outside the range of representable values, the wcstol() function returns LONG_MAX or LONG_MIN (according to the sign of the value), and sets errno to ERANGE. If the string nptr points to is empty or does not have the expected form, no conversion is performed, and the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Upon successful completion, the wcstoll() subroutine returns the converted value. If no conversion could be performed, 0 is returned, and the errno global variable is set to indicate the error. If the correct value is outside the range of representable values, the wcstoll() subroutine returns a value of LONG_LONG_MAX or LONG_LONG_MIN.

The value of errno may be set to ERANGE (range error), or EINVAL (invalid argument).

Example that uses wcstol()

This example uses the wcstol() function to convert the wide-character string wcs to a long integer value.

```
#include <stdio.h>
#include <wchar.h>
```
int main(void)
{
    wchar_t *wcs = L"10110134932";
    wchar_t *stopwcs;
    long     l;
    int      base;

    printf("wcs = "%ls"
        , wcs);
    for (base=2; base<=8; base*=2) {
        l = wcstol(wcs, &stopwcs, base);
        printf("    wcstol = %ld
               Stopped scan at "%ls"

               ", l, stopwcs);
    }
    return 0;
}

/****************************************************
The output should be similar to:
wcs = "10110134932"
wcs[0] = 45
    wcstol = 45
          Stopped scan at "34932"
wcstol = 4423
    wcstol = 4423
          Stopped scan at "4932"
wcstol = 2134108
    wcstol = 2134108
          Stopped scan at "932"
****************************************************/

Related Information
• "strtol() — strtoll() — Convert Character String to Long and Long Long Integer" on page 380
• "strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer" on page 382
• "wcstod() — Convert Wide-Character String to Double" on page 455
• "wcstoul() — wcstoull() — Convert WideCharacter String to Unsigned Long and Unsigned Long Long Integer” on page 463
• “<wchar.h>” on page 17

wcstombs() — Convert Wide-Character String to Multibyte String

Format
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t count);

Language Level: ANSI

Thethreadsafe: Yes.

Description
The wcstombs() function converts the wide-character string pointed to by string into the multibyte array pointed to by dest. The converted string begins in the initial shift state. The conversion stops after count bytes in dest are filled up or a wchar_t null character is encountered.
Only complete multibyte characters are stored in dest. If the lack of space in dest would cause a partial multibyte character to be stored, wcstombs() stores fewer than n bytes and discards the invalid character.

When the program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO), the behavior of wcstombs() is affected by the LC_CTYPE category of the current locale. Remember that the CCSID of the locale should match the CCSID of your job. If the CCSID of the locale is a single-byte CCSID, the wide characters are converted to single-byte characters. Any wide character whose value is greater than 256 would be invalid when converting to single-byte CCSID. When the CCSID is multibyte CCSID, the wide characters are converted to multibyte characters.

When the program is compiled with a UNICODE LOCALETYPE and SYSIFCOPT(*IFSIO), the wide characters are assumed to be Unicode characters, and these Unicode characters are converted to the CCSID of the locale that is associated with LC_TYPE.

Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

Return Value

The wcstombs() function returns the length in bytes of the multibyte character string, not including a ending null character. The value (size_t)-1 is returned if an invalid multibyte character is encountered.

The value of errno may be set to EILSEQ (conversion stopped due to input character), or ECONVERT (conversion error).

Examples that use wcstombs()

This program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO):

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc2[] = L"ABC";
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;

    /* In this first example we will convert a wide character string */
    /* to a single byte character string. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. */
```
if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n");

length = wcstombs(string, wc2, 10);

/* In this case wide characters ABC are converted to */
/* single byte characters ABC, length is 3. */
printf("string = %s, length = %d\n", string, length);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */
if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wcstombs(string, wc_string, 10);

/* The hex look at string would now be: */
/* C10E417141720FC2 length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */
printf("length = %d\n", length);

/* The output should look like this:
string = ABC, length = 3
length = 8 */

This program is compiled with LOCALETYPE(*LOCALEUCS2) and
SYSIFOPT(*IFSIO):
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc2[] = L"ABC";
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;   /* UNICODE A */
    wc_string[1] = 0xFF41;
    wc_string[2] = 0xFF42;
    wc_string[3] = 0x0000;   /* UNICODE B */
    wc_string[4] = 0x0000;
    /* In this first example we will convert a wide character string */
    /* to a single byte character string. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
printf("setlocale failed.\n");

length = wcstombs(string, wc2, 10);

/* In this case wide characters ABC are converted to */
/* single byte characters ABC, length is 3. */
printf("string = %s, length = %d\n", string, length);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */
if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wcstombs(string, wc_string, 10);

/* The hex look at string would now be: */
/* C10E42B02B0FC2 length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */
printf("length = %d\n", length);
}
/* The output should look like this:

string = ABC, length = 3
length = 8
*/

Related Information

- "mbstowcs() — Convert a Multibyte String to a Wide Character String" on page 192
- "wcslen() — Calculate Length of Wide-Character String" on page 440
- "wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)" on page 451
- "wctomb() — Convert Wide Character to Multibyte Character" on page 469
- "<stdlib.h>" on page 15

**wcstoul() — wcstoull() — Convert WideCharacter String to Unsigned Long and Unsigned Long Long Integer**

**Format (wcstoul())**
#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);

**Format (wcstoull())**
#include <wchar.h>
unsigned long long int wcstoull(const wchar_t *nptr, wchar_t **endptr, int base);

**Language Level:** ANSI

**Threading:** Yes.

**Description**
The `wcstoul()` function converts the initial portion of the wide-character string pointed to by `nptr` to an unsigned long integer value. The `nptr` parameter points to a sequence of wide characters that can be interpreted as a numerical value of type unsigned long int. The `wcstoul()` function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the `wchar_t` null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The behavior of the `wcstoul()` function is affected by the LC_CTYPE category of the current locale.

**Note:** This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

The `wcstoull()` subroutine converts a wide-character string to an unsigned long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the `iswspace` subroutine). Any non-space character signifies the start of a subject string that may form an unsigned long long int in the radix specified by the `base` parameter. The subject sequence is defined to be the longest initial substring that is an unsigned long long int of the expected form.

If the value of the `endptr` parameter is not null, then a pointer to the character that ended the scan is stored in `endptr`. If an unsigned long long integer cannot be formed, the value of the `endptr` parameter is set to that of the `nptr` parameter.

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an unsigned long long integer whose radix is specified by the `base` parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the `base` parameter are permitted. If the `base` parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

The value of `errno` may be set to **EINV AL** (`endptr` is null, no numbers are found, or `base` is invalid), or **ERANGE** (converted value is outside the range).

**Return Value**

The `wcstoul()` function returns the converted unsigned long integer value. If no conversion could be performed, the `wcstoul()` function returns 0. If the correct value is outside the range of representable values, The `wcstoul()` function returns `ULONG_MAX` and sets `errno` to **ERANGE**. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Upon successful completion, the `wcstoull()` subroutine returns the converted value. If no conversion could be performed, 0 is returned, and the `errno` global variable is set to indicate the error. If the correct value is outside the range of representable values, `wcstoull()` subroutine returns a value of `ULONG_LONG_MAX`. 
Example that uses \texttt{wcstoul()}

This example uses the \texttt{wcstoul()} function to convert the string \texttt{wcs} to an unsigned long integer value.

```c
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
    wchar_t *wcs = L"1000e13 camels";
    wchar_t *endptr;
    unsigned long int answer;

    answer = wcstoul(wcs, &endptr, BASE);
    printf("The input wide string used: \`%ls\`
"      "The unsigned long int produced: %lu
"      "The substring of the input wide string that was not
" converted to unsigned long: \`%ls\`", wcs, answer, endptr);
    return 0;
}
```

```plaintext
The output should be similar to:
The input wide string used: 1000e13 camels
The unsigned long int produced: 8
The substring of the input wide string that was not converted to unsigned long: e13 camels
```

Related Information

- "\texttt{strtod()} — Convert Character String to Double" on page 376
- "\texttt{strtol()} — \texttt{strtol()} — Convert Character String to Long and Long Long Integer" on page 380
- "\texttt{wcstod()} — Convert Wide-Character String to Double" on page 455
- "\texttt{wcstoll()} — Convert Wide Character String to Long and Long Long Integer" on page 458
- "\texttt{<wchar.h>}" on page 17

\texttt{wcswcs()} — Locate Wide-Character Substring

\textbf{Format}

```c
#include <wctype.h>
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

\textbf{Language Level}: XPG4

\textbf{Threadsafe}: Yes.

\textbf{Description}

The \texttt{wcswcs()} function locates the first occurrence of \texttt{string2} in the wide-character string pointed to by \texttt{string1}. In the matching process, the \texttt{wcswcs()} function ignores the wchar_t null character that ends \texttt{string2}.

\textbf{Return Value}
The `wcswcs()` function returns a pointer to the located string or NULL if the string is not found. If `string2` points to a string with zero length, `wcswcs()` returns `string1`.

**Example that uses wcswcs()**

This example finds the first occurrence of the wide character string `pr` in `buffer1`.

```c
#include <stdio.h>
#include <wcstr.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wchar_t * wch = L"pr";

    ptr = wcswcs( buffer1, wch );
    printf( "The first occurrence of %ls in '%ls' is '%ls'\n", wch, buffer1, ptr );
}
```

The first occurrence of `pr` in 'computer program' is 'program'

**Related Information**

- "strchr() — Search for Character" on page 342
- "strcspn() — Find Offset of First Character Match" on page 348
- "strpbrk() — Find Characters in String" on page 367
- "strchr() — Locate Last Occurrence of Character in String" on page 373
- "strspn() — Find Offset of First Non-matching Character" on page 374
- "strstr() — Locate Substring" on page 375
- "wcschr() — Search for Wide Character" on page 431
- "wcsncmp() — Compare Wide-Character Strings" on page 433
- "wcsncmp() — Find Offset of First Wide-Character Match" on page 436
- "wcspbrk() — Locate Wide Characters in String" on page 447
- "wcsrchr() — Locate Last Occurrence of Wide Character in String" on page 450
- "wcsspn() — Find Offset of First Non-matching Wide Character" on page 453
- "<wcsstr.h>" on page 17

---

**wcswidth() — Determine the Display Width of a Wide Character String**

**Format**

```c
#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);
```

**Language Level**: XPG4

**Threatsafe**: Yes.

**Description**
The `wcswidth()` function determines the number of printing positions that a
graphic representation of `n` wide characters (or fewer that `n` wide characters if a
null wide character is encountered before `n` wide characters have been exhausted)
in the wide string pointed to by `wcs` occupies on a display device. The number is
independent of its location on the device.

The value of `errno` may be set to `EINVAL` (non-printing wide character).

**Return Value**

The `wcswidth()` function either returns:
- 0, if `wcs` points to a null wide character; or
- the number of printing positions occupied by the wide string pointed to by `wcs`; or
- -1, if any wide character in the wide string pointed to by `wcs` is not a printing wide character.

The behavior of the `wcswidth()` function is affected by the LC_CTYPE category. If the program is compiled with a UNICODE LOCALETYPE, the wide character properties are those defined by the LC_UNI_CTYPE category of the current locale.

**Example that uses wcswidth()**

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"ABC";
    printf("wcs has a width of: %d\n", wcswidth(wcs,3));
}
```

The output is as follows:
```
wcs has a width of: 3
```

**Related Information**

- "wcswidth() — Determine the Display Width of a Wide Character String" on page 466
- "<wchar.h>" on page 17

---

**wcsxfrm() — Transform a Wide-Character String**

**Format**

```c
#include <wchar.h>
size_t wcsxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

**Language Level:** XPG4

**Threading and Security:**

**Description**
The wcsxfrm() function transforms the wide-character string pointed to by wcs2 to values which represent character collating weights and places the resulting wide-character string into the array pointed to by wcs1.

Note: This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. The behavior of this wide-character function is affected by the LC_COLLATE category of the current locale, unless LOCALETYPE(*LOCALEUTF) was specified on the compilation command. If LOCALETYPE(*LOCALEUTF) was specified this function is affected by the LC_UNI_COLLATE category of the current locale.

Return Value

The wcsxfrm() function returns the length of the transformed wide-character string (not including the ending null wide character code). If the value returned is n or more, the contents of the array pointed to by wcs1 are indeterminate.

If wcsxfrm() is unsuccessful, errno is changed. The value of errno may be set to EINVAL (the wcs1 or wcs2 arguments contain characters which are not available in the current locale).

Example that uses wcsxfrm()

#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t *wcs;
  wchar_t buffer[80];
  int length;

  printf("Type in a string of characters.\n ");
  wcs = fgetws(buffer, 80, stdin);
  length = wcsxfrm(NULL, wcs, 0);
  printf("You would need a %d element array to hold the wide string\n", length);
  printf("\n\n transformed according to this program's locale. \n");
}

Related Information

* strxfrm() — Transform String” on page 384
* “<wchar.h>” on page 17

wctob() — Convert Wide Character to Byte

Format

#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);

Language Level: ANSI

Threadsafe: Yes.

Description
The wctob() function determines whether wc corresponds to a member of the
extended character set, whose multibyte character has a length of 1 byte when in
the initial shift state. The behavior of the wctob() function is affected by the
LC_CTYPE category of the current locale.
|
|

Note: This function is not available when LOCALETYPE(*CLD) is specified on the
compilation command.
Return Value
If c corresponds to a multibyte character with a length of 1 byte, the wctob()
function returns the single-byte representation. Otherwise, it returns EOF.

|

If a conversion error occurs, errno may be set to ECONVERT.
Example that uses wctob()
This example uses the wctob() function to test if the wide character A is a valid
single-byte character.
#include <stdio.h>
#include <wchar.h>
int main(void)
{
wint_t wc = L’A’;
if (wctob(wc) == wc)
printf("%lc is a valid single byte character\n", wc);
else
printf("%lc is not a valid single byte character\n", wc);
return 0;
/************************************************************
The output should be similar to:
A is a valid single byte character
************************************************************/
}

Related Information
v “mbtowc() — Convert Multibyte Character to a Wide Character” on page 196
v “wctomb() — Convert Wide Character to Multibyte Character”
v “wcstombs() — Convert Wide-Character String to Multibyte String” on page 460
v “<wchar.h>” on page 17

wctomb() — Convert Wide Character to Multibyte Character
Format
#include <stdlib.h>
int wctomb(char *string, wchar_t character);

Language Level: ANSI
Threadsafe: No. Use wcrtomb() instead.
Description

Chapter 2. Library Functions

469


The \texttt{wctomb()} function converts the \texttt{wchar_t} value of \texttt{character} into a multibyte array pointed to by \texttt{string}. If the value of \texttt{character} is 0, the function is left in the initial shift state. At most, the \texttt{wctomb()} function stores MB_CUR_MAX characters in \texttt{string}.

The conversion of the wide character is the same as described in \texttt{wcstombs()}. See this function for a Unicode example.

\textbf{Return Value}

The \texttt{wctomb()} function returns the length in bytes of the multibyte character. The value -1 is returned if \texttt{character} is not a valid multibyte character. If \texttt{string} is a NULL pointer, the \texttt{wctomb()} function returns nonzero if shift-dependent encoding is used, or 0 otherwise.

If a conversion error occurs, \texttt{errno} may be set to \texttt{ECONVERT}.

\textbf{Example that uses \texttt{wctomb()}}

This example converts the wide character \texttt{c} to a multibyte character.

\begin{verbatim}
#include <stdio.h>
#include <stdlib.h>
#include <wcstr.h>
#define SIZE 40
int main(void)
{
    static char buffer[ SIZE ];
    wchar_t wch = L'c';
    int length;

    length = wctomb( buffer, wch );
    printf( "The number of bytes that comprise the multibyte "
            "character is \%d\n", length );
    printf( "And the converted string is \"%s\"
", buffer );
}

/****************  Output should be similar to:  ******************
The number of bytes that comprise the multibyte character is 1
And the converted string is "c"
*/
\end{verbatim}

\textbf{Related Information}
\begin{itemize}
    \item \texttt{mbtowc() — Convert Multibyte Character to a Wide Character} on page 196
    \item \texttt{wcslen() — Calculate Length of Wide-Character String} on page 440
    \item \texttt{wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)} on page 425
    \item \texttt{wcstombs() — Convert Wide-Character String to Multibyte String} on page 460
    \item \texttt{wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)} on page 451
    \item \texttt{<stdlib.h>" on page 15}
\end{itemize}
#include <wctype.h>
wctrans_t wctrans(const char *property);

Language Level: ANSI

Threading: Yes.

Description

The `wctrans()` function constructs a value with type `wctrans_t`. This value describes a mapping between wide characters identified by the string argument `property`. The two strings listed under the `towctrans()` function description shall be valid in all locales as `property` arguments to the `wctrans()` function.

Return Value

If `property` identifies a valid mapping of wide characters according to the LC_CTYPE category of the current locale, the `wctrans()` function returns a nonzero value that is valid as the second argument to the `towctrans()` function. Otherwise, it returns 0.

Example that uses `wctrans()`

This example translates the lowercase alphabet to uppercase, and back to lowercase.
**Related Information**

- "towctrans() — Translate Wide Character" on page 396
- "<wctype.h>" on page 17

---

**wctype() — Get Handle for Character Property Classification**

**Format**

```c
#include <wctype.h>
wctype_t wctype(const char *property);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Description**

The `wctype()` function is defined for valid property names as defined in the current locale. The property is a string that identifies a generic character class for which locale specific type information is required. The function returns a value of type `wctype_t`, which can be used as the second argument to a call of the `iswctype()` function.

The `wctype()` function determines values of `wctype_t` according to rules of the coded character set that are defined by character type information in the program’s
locale (category LC_CTYPE). Values that are returned by the `wctype()` function are valid until a call to `setlocale()` that changes the category LC_CTYPE.

**Return Value**

The `wctype()` function returns zero if the given property name is not valid for the current locale (category LC_CTYPE). Otherwise it returns a value of type `wctype_t` that can be used in calls to `iswctype()`.

**Example that uses `wctype()`**
#include <wchar.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
    int wc;
    for (wc = 0; wc <= UPPER_LIMIT; wc++) {
        printf("%#4x\t", wc);
        printf("%c", iswctype(wc, wctype("print")) ? wc : "");
        printf("%c", iswctype(wc, wctype("alnum")) ? "AN" : "");
        printf("%c", iswctype(wc, wctype("alpha")) ? "A" : "");
        printf("%c", iswctype(wc, wctype("blank")) ? "B" : "");
        printf("%c", iswctype(wc, wctype("cntrl")) ? "C" : "");
        printf("%c", iswctype(wc, wctype("digit")) ? "D" : "");
        printf("%c", iswctype(wc, wctype("graph")) ? "G" : "");
        printf("%c", iswctype(wc, wctype("lower")) ? "L" : "");
        printf("%c", iswctype(wc, wctype("punct")) ? "PU" : "");
        printf("%c", iswctype(wc, wctype("space")) ? "S" : "");
        printf("%c", iswctype(wc, wctype("print")) ? "PR" : "");
        printf("%c", iswctype(wc, wctype("upper")) ? "U" : "");
        printf("%c", iswctype(wc, wctype("xdigit")) ? "X" : "");
    }
    putchar('
');
    return 0;
    /**************************************************************************************
    The output should be similar to :
    :
    0x1f              C
    0x20           B                   S    PR
    0x21   !                  G       PU      PR
    0x22   "                  G       PU      PR
    0x23   #                  G       PU      PR
    0x24   $                  G       PU      PR
    0x25   %                  G       PU      PR
    0x26   &                  G       PU      PR
    0x27   '                  G       PU      PR
    0x28   (                  G       PU      PR
    0x29   )                  G       PU      PR
    0x2a   *                  G       PU      PR
    0x2b   +                  G       PU      PR
    0x2c   ,                  G       PU      PR
    0x2d   .                  G       PU      PR
    0x2e   /                  G       PU      PR
    0x2f   0                   AN           D   G     PR      X
    0x30   1                   AN           D   G     PR      X
    0x31   2                   AN           D   G     PR      X
    0x32   3                   AN           D   G     PR      X
    0x33   4                   AN           D   G     PR      X
    0x34   5                   AN           D   G     PR      X
    :
    **************************************************************************************
}
wcwidth() — Determine the Display Width of a Wide Character

Format

```c
#include <wchar.h>
int wcwidth (const wint_t wc);
```

Language Level: XPG4

Threadsafe: Yes.

Description

The `wcwidth()` function determines the number of printing positions that a graphic representation of `wc` occupies on a display device. Each of the printing wide characters occupies its own number of printing positions on a display device. The number is independent of its location on the device.

The value of `errno` may be set to `EINVAL` (non-printing wide character).

**Note:** This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

Return Value

The `wcwidth()` function either returns:

- 0, if `wc` is a null wide character; or
- the number of printing position occupied by `wc`; or
- -1, if `wc` is not a printing wide character.

The behavior of the `wcwidth()` function is affected by the LC_CTYPE category of the current locale. If the program is compiled with a UNICODE LOCALETYPE, the wide character properties are those defined by the LC_UNI_CTYPE category of the current locale.

Example that uses `wcwidth()`

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';
    printf("%lc has a width of %d\n", wc, wcwidth(wc));
    return 0;
}
```

The output should be similar to:

```
A has a width of 1
```

***************
Related Information

- "wcswidth() — Determine the Display Width of a Wide Character String" on page 466
- "<wchar.h>" on page 17

**wfopen() — Open Files**

**Format**
```c
#include <ifs.h>
FILE * wfopen(const wchar_t *filename, const wchar_t *mode);
```

**Language Level:** ILE C Extension

**Threading:** Yes

**Description**

The `wfopen()` function works like the `fopen()` function, except:

- `wfopen()` accepts file name and mode as wide characters.
- `wfopen()` assumes CCSID 13488 if neither CCSID nor codepage keyword is specified.

**Note:** This function is available only when SYSIFCOPT(*IFSIO) and LOCALETYPE(*LOCALE) or LOCALETYPE(*LOCALEUCS2) are specified on the compilation command.

**wmemchr() — Locate Wide Character in Wide-Character Buffer**

**Format**
```c
#include <wchar.h>
wchar_t * wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

**Language Level:** ANSI

**Threading:** Yes.

**Description**

The `wmemchr()` function locates the first occurrence of `c` in the initial `n` wide characters of the object pointed to by `s`. If `n` has the value 0, the `wmemchr()` function finds no occurrence of `c`, and returns a NULL pointer.

**Return Value**

The `wmemchr()` function returns a pointer to the located wide character, or a NULL pointer if the wide character does not occur in the object.

**Example that uses wmemchr()**

This example finds the first occurrence of ‘A’ in the wide-character string.
```c
#include <stdio.h>
#include <wchar.h>

main()
{
  wchar_t *in = L"1234ABCD";
  wchar_t *ptr;
  wchar_t fnd = L'A';

  printf("\nEXPECTED: ABCD");
  ptr = wmemchr(in, L'A', 6);
  if (ptr == NULL)
    printf("\n** ERROR ** ptr is NULL, char L'A' not found\n");
  else
    printf("\nRECEIVED: %ls \n", ptr);
}
```

### Related Information
- "memchr() — Search Buffer“ on page 197
- "strchr() — Search for Character“ on page 342
- "wcschr() — Search for Wide Character“ on page 431
- “wmemcmp() — Compare Wide-Character Buffers”
- "wmemcpy() — Copy Wide-Character Buffer“ on page 478
- "wmemmove() — Copy Wide-Character Buffer“ on page 479
- "wmemset() — Set Wide Character Buffer to a Value“ on page 480
- "<wchar.h>” on page 17

---

**wmemcmp() — Compare Wide-Character Buffers**

**Format**

```c
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `wmemcmp()` function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`. If `n` has the value 0, the `wmemcmp()` function returns 0.

**Return Value**

The `wmemcmp()` function returns a value according to the relationship between the two strings, `s1` and `s2`:

<table>
<thead>
<tr>
<th>Integer Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>s1</code> less than <code>s2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>s1</code> equal to <code>s2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>s1</code> greater than <code>s2</code></td>
</tr>
</tbody>
</table>

**Example that uses `wmemcmp()`**
This example compares the wide-character string in to out using the wmemcmp() function.

```c
#include <wchar.h>
#include <stdio.h>
#include <locale.h>

main()
{
    int rc;
    wchar_t *in = L"12345678";
    wchar_t *out = L"12AAAAAB";
    setlocale(LC_ALL, "POSIX");

    printf("\nGREATER is the expected result\n");
    rc = wmemcmp(in, out, 3);
    if (rc == 0)
        printf("\nArrays are EQUAL %ls %ls \n", in, out);
    else
    {
        if (rc > 0)
            printf("\nArray %ls GREATER than %ls \n", in, out);
        else
            printf("\nArray %ls LESS than %ls \n", in, out);
    }

    /*******************************************
    The output should be:
    GREATER is the expected result
    Array 12345678 GREATER than 12AAAAAB
    *******************************************/
}
```

Related Information

- "memcmp() — Compare Buffers" on page 198
- "strcmp() — Compare Strings" on page 343
- "wcsncmp() — Compare Wide-Character Strings" on page 443
- "<wchar.h>" on page 17

**wmemcmp() — Copy Wide-Character Buffer**

Format

```c
#include <wchar.h>
wchar_t *wmemcmp(wchar_t *s1, const wchar_t *s2, size_t n);
```

Language Level: ANSI

Threadsafe: Yes.

Description
The `wmemcpy()` function copies \( n \) wide characters from the object pointed to by `s2` to the object pointed to by `s1`. If `s1` and `s2` overlap, the result of the copy is unpredictable. If \( n \) has the value 0, the `wmemcpy()` function copies 0 wide characters.

**Return Value**

The `wmemcpy()` function returns the value of `s1`.

**Example that uses `wmemcpy()`**

This example copies the first four characters from `out` to `in`. In the expected output, the first four characters in both strings will be "ABCD".

```c
#include <wchar.h>
#include <stdio.h>

main()
{
    wchar_t *in = L"12345678";
    wchar_t *out = L"ABCDEFGH";
    wchar_t *ptr;

    printf("Expected result: First 4 chars of in change\n" and are the same as first 4 chars of out");
    ptr = wmemcpy(in, out, 4);
    if (ptr == in)
        printf("Array in %ls array out %ls %n", in, out);
    else
    {
        printf("*** ERROR ***\n" returned pointer wrong");
    }
}
```

**Related Information**

- "memcpy() — Copy Bytes" on page 199
- "strcpy() — Copy Strings" on page 347
- "strncpy() — Copy Strings" on page 364
- "wcsncpy() — Copy Wide-Character Strings" on page 435
- "wcsncmp() — Compare Wide-Character Buffers" on page 477
- "wmemchr() — Locate Wide Character in Wide-Character Buffer" on page 476
- "wmemcpy() — Copy Wide-Character Buffer" on page 479
- "<wchar.h>" on page 17

**wmemmove() — Copy Wide-Character Buffer**

**Format**

```c
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.
**Description**

The `wmemmove()` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`. Copying takes place as if the `n` wide characters from the object pointed to by `s2` are first copied into a temporary array, of `n` wide characters, that does not overlap the objects pointed to by `s1` or `s2`. Then, the `wmemmove()` function copies the `n` wide characters from the temporary array into the object pointed to by `s1`. If `n` has the value 0, the `wmemmove()` function copies 0 wide characters.

**Return Value**

The `wmemmove()` function returns the value of `s1`.

**Example that uses `wmemmove()`**

This example copies the first five characters in a string to overlay the last five characters in the same string. Since the string is only nine characters long, the source and target overlap.

```c
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *theString = L"ABCDEFGHI";
    printf("\nThe original string: %ls \n", theString);
    wmemmove(theString+4, theString, 5);
    printf("\nThe string after wmemmove: %ls \n", theString);
    return;
}/*******************************************************************/
    The output should be:
    The original string: ABCDEFGHI
    The string after wmemmove: ABCDABCDE
 /*******************************************************************/
}
```

**Related Information**

- "`memmove()` — Copy Bytes“ on page 202
- "`wmemchr()` — Locate Wide Character in Wide-Character Buffer“ on page 476
- "`wmemcpy()` — Copy Wide-Character Buffer” on page 478
- "`wmemcmp()` — Compare Wide-Character Buffers“ on page 477
- "`wmemset()` — Set Wide Character Buffer to a Value“
- "`<wchar.h>“ on page 17

---

**wmemset() — Set Wide Character Buffer to a Value**

**Format**

```c
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

**Language Level**: ANSI

**Threadsafe**: Yes.
Description

The wmemset() function copies the value of c into each of the first n wide characters of the object pointed to by s. If n has the value 0, the wmemset() function copies 0 wide characters.

Return Value

The wmemset() function returns the value of s.

Example that uses wmemset()

This example sets the first 6 wide characters to the wide character ‘A’.

```c
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;

    printf("\nEXPECTED: AAAAAACD\n");
    ptr = wmemset(in, L'A', 6);
    if (ptr == in)
        printf("\nResults returned - %ls %n", ptr);
    else
    {
        printf("\n** ERROR ** wrong pointer returned\n");
    }
}
```

Related Information

- "memset() — Set Bytes to Value” on page 203
- “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 476
- “wmemcpuy() — Copy Wide-Character Buffer” on page 478
- “wmemcmp() — Compare Wide-Character Buffers” on page 477
- “wmemmove() — Copy Wide-Character Buffer” on page 479
- “<wchar.h>” on page 17

wprintf() — Format Data as Wide Characters and Print

Format

```
#include <stdio.h>
int wprintf(const wchar_t *format,...);
```

Language Level: ANSI

Threadsafe: Yes.

Description

A wprintf(format, ...) is equivalent to fwprintf(stdout, format, ...).

Note: This function is available when SYSIFCOPT("IFSIO") is specified on the compilation command, and LOCALETYPEN("CLD") is not specified on the compilation command.
Return Value

The wprintf() function returns the number of wide characters transmitted. If an output error occurred, the wprintf() function returns a negative value.

Example that uses wprintf()

This example prints the wide character a. Date and time may be formatted according to your locale's representation. The output goes to stdout.

```c
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "POSIX");
    wprintf(L "%c\n", L'a');
    return(0);
}
/* A long 'a' is written to stdout */
```

Related Information

- "printf() — Print Formatted Characters” on page 211
- "btowc() — Convert Single Byte to Wide Character” on page 55
- "mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 186
- "vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 407
- "fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 133
- "vswprintf() — Format and Write Wide Characters to Buffer” on page 418
- "<wchar.h>” on page 17

wscanf() — Read Data Using Wide-Character Format String

**Format**

```
#include <stdio.h>
int wscanf(const wchar_t *format,...);
```

**Language Level:** ANSI  
**Threadsafe:** Yes.

**Description**

The wscanf() function is equivalent to the fwscanf() function with the argument stdin interposed before the arguments of the wscanf() function.

**Note:** This function is available when SYSIFCOPT("IFSIO) is specified on the compilation command, and LOCALETYP("CLD) is not specified on the compilation command.

**Return Value**
If an input failure occurs before any conversion, the `wscanf()` function returns the value of the macro `EOF`.

Otherwise, the `wscanf()` function returns the number of input items assigned. It can be fewer than provided for, or even zero, in the event of an early matching failure.

**Example that uses `wscanf()`**

This example scans various types of data.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    int i;
    float fp;
    char c, s[81];

    printf("Enter an integer, a real number, a character and a string : \n");
    if (wscanf(L "%d %f %c %s", &i, &fp, &c, s) != 4)
        printf("Some fields were not assigned\n");
    else {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
    return 0;
}
```

The output should be similar to:

```
Enter an integer, a real number, a character and a string : 
12 2.5 a yes
integer = 12
real number = 2.500000
character = a
string = yes
```

**Related Information**

- "fscanf() — Read Formatted Data“ on page 123
- "fwprintf() — Format Data as Wide Characters and Write to a Stream“ on page 133
- "fwscanf() — Read Data from Stream Using Wide Character“ on page 137
- "scanf() — Read Data“ on page 313
- "sscanf() — Read Data“ on page 339
- "swprintf() — Format and Write Wide Characters to Buffer“ on page 385
- "swscanf() — Read Wide Character Data“ on page 387
- "vfscanf() — Read Formatted Data“ on page 406
- "vfwscanf() — Read Formatted Wide Character Data“ on page 409
- "vscanf() — Read Formatted Data“ on page 412
- "vsscanf() — Read Formatted Data“ on page 416
- "vswscanf() — Read Formatted Wide Character Data“ on page 420
- "vwscanf() — Read Formatted Wide Character Data“ on page 423
• "wprintf() — Format Data as Wide Characters and Print" on page 481
• "<wchar.h>" on page 17
Chapter 3. Run-Time Considerations

This chapter provides information on:

- Exception and condition management
- Interlanguage data type compatibility
- CCSID (Coded Character Set Identifier) source file conversion

errno Macros

The following table lists which error macros the ILE C library functions can set.

Table 12. errno Macros

<table>
<thead>
<tr>
<th>Error Macro</th>
<th>Description</th>
<th>Set by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADDATA</td>
<td>The message data is not valid.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>EBADDF</td>
<td>The catalog descriptor is not valid.</td>
<td>catclose, catgets, clearerr, fgetc, fgetpos, fgets, fileno, freopen, fseek, fsetpos, getc, rewind</td>
</tr>
<tr>
<td>EBADKEYLN</td>
<td>The key length specified is not valid.</td>
<td>_Rreadk, _Rlocate</td>
</tr>
<tr>
<td>EBADMODE</td>
<td>The file mode specified is not valid.</td>
<td>fopen, freopen, _Ropen</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>Bad file name specified.</td>
<td>fopen, freopen, _Ropen</td>
</tr>
<tr>
<td>EBADPOS</td>
<td>The position specified is not valid.</td>
<td>fsetpos</td>
</tr>
<tr>
<td>EBADSEEK</td>
<td>Bad offset for a seek operation.</td>
<td>fgetpos, fseek</td>
</tr>
<tr>
<td>EBUSY</td>
<td>The record or file is in use.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>Conversion error.</td>
<td>wcstomb, wcstol, wcswidth</td>
</tr>
<tr>
<td>EDOM</td>
<td>Domain error in math function.</td>
<td>acos, asin, atan2, cos, exp, fmod, gamma, hypot, j0, j1, jn, y0, y1, yn, log, log10, pow, sin, strtol, strtof, sqrt, tan</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An illegal read operation occurred after a write operation.</td>
<td>fgetc, fread, getc, getchar</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>The character sequence does not form a valid multibyte character.</td>
<td>fgetwc, fgetws, getwc, mblen,mbrlen, mbtowc, mbrtowcs, mbstowcs, mbtowc, printf, scanf, ungetwc, wctomb, wcstombs, wcstombs, wcstomb, wcswidth, wcscwidth</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The signal is not valid.</td>
<td>printf, scanf, signal, swprintf, swscanf, wctol, wctoul, wcstoul, wcstoul</td>
</tr>
<tr>
<td>Error Macro</td>
<td>Description</td>
<td>Set by Function</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>EIO</td>
<td>Consecutive calls of I/O occurred.</td>
<td>I/O</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
<td>All I/O functions</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
<td>All I/O functions</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation attempted on a wrong device.</td>
<td>fgetpos, fsetpos, fseek, ftell,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rewind</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or library is not found.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>ENOPOS</td>
<td>No record at specified position.</td>
<td>fssetpos</td>
</tr>
<tr>
<td>ENOREC</td>
<td>Record not found.</td>
<td>fread, perror, strerror</td>
</tr>
<tr>
<td>ENOTDLT</td>
<td>File is not opened for delete operations.</td>
<td>_Rdelete</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>File is not opened.</td>
<td>clearerr, fclose, fflush,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fgetpos, fopen, freopen,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fseek, ftell, setbuf, setvbuf,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Ropen, _Rclose</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>File is not opened for read operations.</td>
<td>fgetc, fread, ungetc, _Rreadd,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Rreadf, _Rreadindv,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Rreadk, _Rreadl, _Rreadn,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Rreadnc, _Rreadp, _Rreads,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Rlocate</td>
</tr>
<tr>
<td>ENOTUPD</td>
<td>File is not opened for update operations.</td>
<td>_Rrlslck, _Rupdate</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>File is not opened for write operations.</td>
<td>fprintf, fwrite, _Rwrite,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_Rwrited, _Rwriterd</td>
</tr>
<tr>
<td>ENUMMBRS</td>
<td>More than 1 member.</td>
<td>ftell</td>
</tr>
<tr>
<td>ENUMRECS</td>
<td>Too many records.</td>
<td>ftell</td>
</tr>
<tr>
<td>EPAD</td>
<td>Padding occurred on a write operation.</td>
<td>fwrite</td>
</tr>
<tr>
<td>EPERM</td>
<td>Insufficient authorization for access.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read</td>
<td>fprintf, fwrite, fprintf, putc,</td>
</tr>
<tr>
<td></td>
<td>operation.</td>
<td>putchar</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Range error in math function.</td>
<td>cos, cosh, gamma, exp, j0, j1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jn, y0, y1, yn, log, log10,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ldexp, pow, sin, sinh, strtod,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strtol, strtoull, tan, wcstol,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wcstoll, wcstoul, wcstoull,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wcstod</td>
</tr>
<tr>
<td>ERECIO</td>
<td>File is opened for record I/O, so character-at-a-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time processing functions cannot be used.</td>
<td>fgetc, fgetpos, fprintf, fread,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fseek, fsetpos, ftell</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
<td>feof, ferror, fgetpos, fprintf,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fseek, fsetpos, ftell, fwrite</td>
</tr>
</tbody>
</table>
### Table 12. errno Macros (continued)

<table>
<thead>
<tr>
<th>Error Macro</th>
<th>Description</th>
<th>Set by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
<td>fgetc, fgetpos, fread, fseek, fsetpos, ftell</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
<td>fgetpos, fputc, fseek, fsetpos, ftell, fwrite</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on I/O operation.</td>
<td>Any I/O function that reads or writes a record sets errno to ETRUNC.</td>
</tr>
</tbody>
</table>

### errno Values for Integrated File System Enabled C Stream I/O

The following table describes the possible settings when using integrated file system enabled stream I/O.

### Table 13. errno Values for IFS Enabled C Stream I/O

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr</td>
<td>EBADF</td>
</tr>
<tr>
<td>fclose</td>
<td>EAGAIN, EBADF, EIO, ESCANFAILURE, EUNKNOWN</td>
</tr>
<tr>
<td>feof</td>
<td>EBADF</td>
</tr>
<tr>
<td>ferror</td>
<td>EBADF</td>
</tr>
<tr>
<td>fflush</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EINAVAL, EIO, ENOMEM, EUNOWKN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fgetc</td>
<td>EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINAVAL, EIO, ENOMEM, EUNOWKN, EGETANDPUT, ENOTREAD, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>fgetpos</td>
<td>EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINAVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>fgets</td>
<td>EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINAVAL, EIO, ENOMEM, EUNOWKN, EGETANDPUT, ENOTREAD</td>
</tr>
<tr>
<td>fgetwc</td>
<td>EBADF, EILSEQ</td>
</tr>
<tr>
<td>fgetws</td>
<td>EBADF, EILSEQ</td>
</tr>
<tr>
<td>fopen</td>
<td>EAGAIN, EBADNAME, EBADF, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINAVAL, EIO, EISDIR, ELOOOP, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, ESCANFAILURE</td>
</tr>
<tr>
<td>fprintf</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINAVAL, EIO, ENOMEM, EUNOWKN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fputc</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINAVAL, EIO, ENOMEM, EUNOWKN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fputs</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINAVAL, EIO, ENOMEM, EUNOWKN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fread</td>
<td>EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINAVAL, EIO, ENOMEM, EUNOWKN, EGETANDPUT, ENOTREAD</td>
</tr>
</tbody>
</table>
Table 13. errno Values for IFS Enabled C Stream I/O (continued)

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>freopen</td>
<td>EACCES, EAGAIN, EBADNAME, EBADE, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR</td>
</tr>
<tr>
<td>fscanf</td>
<td>EBADE, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>fseek</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>fsetpos</td>
<td>EACCES, EAGAIN, ABADE, EBUSY, EDAMAGE, EINVAL, EIO, EPERM, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>ftell</td>
<td>EACCESS, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>fwrite</td>
<td>EACCESS, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>getc</td>
<td>EBADE, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>getchar</td>
<td>EBADE, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>gets</td>
<td>EBADE, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>getwc</td>
<td>EBADE, EILSEQ</td>
</tr>
<tr>
<td>perror</td>
<td>EBADE</td>
</tr>
<tr>
<td>printf</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EFBIG, EILSEQ, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>putc</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>putchar</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>puts</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>remove</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROOBJ, EUNKNOWN, EXDEV</td>
</tr>
<tr>
<td>rename</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, ECONVERT, EDAMAGE, EEXIST, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOTEMPTY, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EMLINK, EPERM, EUNKNOWN, EXDEV</td>
</tr>
<tr>
<td>rewind</td>
<td>EACCES, EAGAIN, EBADE, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN</td>
</tr>
</tbody>
</table>
Table 13. errno Values for IFS Enabled C Stream I/O (continued)

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanf</td>
<td>EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EILSEQ, EINVAL, EIO, ENOMEM, EUKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>setbuf</td>
<td>EBADF, EINVAL, EIO</td>
</tr>
<tr>
<td>setvbuf</td>
<td>EBADF, EINVAL, EIO</td>
</tr>
<tr>
<td>tmpfile</td>
<td>EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, EPERM, EROOBJ, EUNKNOWN, ENUNKNOWN, N, EXDEV</td>
</tr>
<tr>
<td>tmpnam</td>
<td>EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOENT, ENOSYSRSC, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>ungetc</td>
<td>EBADF, EIO</td>
</tr>
<tr>
<td>ungetwc</td>
<td>EBADF, EILSEQ</td>
</tr>
<tr>
<td>vfprintf</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>vprintf</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
</tbody>
</table>

---

**Record Input and Output Error Macro to Exception Mapping**

The following table describes what occurs if the signal SIGIO is raised. Only *ESCAPE, *NOTIFY, and *STATUS messages are monitored.

Table 14. Record Input and Output Error Macro to Exception Mapping

<table>
<thead>
<tr>
<th>Description</th>
<th>Messages (EXCP_MSGID)</th>
<th>errno setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>*STATUS and *NOTIFY</td>
<td>CPF4001 to CPF40FF, CPF4401 to CPF44FF, CPF4901 to CPF49FF, CPF5004</td>
<td>errno is not set, a default reply is returned to the operating system.</td>
</tr>
<tr>
<td>Recoverable I/O error</td>
<td>CPF4701 to CPF47FF, CPF4801 to CPF48FF, CPF5001 to CPF5003, CPF5005 to CPF50FF</td>
<td>EIORECERR</td>
</tr>
<tr>
<td>Non-recoverable I/O error(^2)</td>
<td>CPF4101 to CPF41FF, CPF4201 to CPF42FF, CPF4301 to CPF43FF, CPF4501 to CPF45FF, CPF4601 to CPF46FF, CPF5101 to CPF51FF, CPF5201 to CPF52FF, CPF5301 to CPF53FF, CPF5401 to CPF54FF, CPF5501 to CPF55FF, CPF5601 to CPF56FF</td>
<td>EIOERROR</td>
</tr>
<tr>
<td>Truncation occurred at I/O operation</td>
<td>C2M3003</td>
<td>ETRUNC</td>
</tr>
<tr>
<td>File is not opened</td>
<td>C2M3004</td>
<td>ENOTOPEN</td>
</tr>
</tbody>
</table>
### Signal Handling Action Definitions

The following table shows the initial state of the C signal values and their handling action definitions when `{SYSIFCOPT(*NOASYNCSIGNAL)}` is specified on the compilation command. SIG_DFL always percolates the condition to the handler. Resume indicates the exception is handled, and the application continues.

<table>
<thead>
<tr>
<th>Signal Value</th>
<th>Initial State</th>
<th>SIG_DFL</th>
<th>SIG_IGN</th>
<th>Return from Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT(^1)</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGALL(^2)</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore(^3)</td>
<td>Resume(^4)</td>
</tr>
<tr>
<td>SIGILL</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore(^3)</td>
<td>Resume(^4)</td>
</tr>
<tr>
<td>SIGINT</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGIO</td>
<td>SIG_IGN</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore(^3)</td>
<td>Resume(^4)</td>
</tr>
</tbody>
</table>

\(^1\) The error is percolated to the user, therefore the user’s direct monitor handlers, ILE C condition handlers and signal handler may get control. The initial setting for SIGIO is SIG_IGN.

\(^2\) The type of device determines whether the error is recoverable or not recoverable. The following IBM® publications contain information about recoverable and non-recoverable system exceptions for each specific file type:
- ICF Programming
- ADTS/400: Advanced Printer Function
- Application Display Programming
- Database Programming

---

Table 14. Record Input and Output Error Macro to Exception Mapping (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Messages (_EXCP_MSGID)</th>
<th>errno setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>File is not opened for read operations</td>
<td>C2M3005</td>
<td>ENOTREAD</td>
</tr>
<tr>
<td>File is not opened for write operations</td>
<td>C2M3009</td>
<td>ENOTWRITE</td>
</tr>
<tr>
<td>Bad file name specified</td>
<td>C2M3014</td>
<td>EBADNAME</td>
</tr>
<tr>
<td>The file mode specified is not valid</td>
<td>C2M3015</td>
<td>EBADMODE</td>
</tr>
<tr>
<td>File is not opened for update operations</td>
<td>C2M3041</td>
<td>ENOTUPD</td>
</tr>
<tr>
<td>File is not opened for delete operations</td>
<td>C2M3042</td>
<td>ENOTDLT</td>
</tr>
<tr>
<td>The key length specified is not valid</td>
<td>C2M3044</td>
<td>EBADKEYLN</td>
</tr>
<tr>
<td>A non-recoverable I/O error occurred</td>
<td>C2M3101</td>
<td>EIOERROR</td>
</tr>
<tr>
<td>A recoverable I/O error occurred</td>
<td>C2M3102</td>
<td>EIORECERR</td>
</tr>
</tbody>
</table>
Table 15. Handling Action Definitions for Signal Values (continued)

<table>
<thead>
<tr>
<th>Signal Value</th>
<th>Initial State</th>
<th>SIG_DFL</th>
<th>SIG_IGN</th>
<th>Return from Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGSEGV</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore(^3)</td>
<td>Resume(^4)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
</tbody>
</table>

Note:
- \(^1\) Can only be signaled by the \texttt{raise()} function or the \texttt{abort()} function
- \(^2\) SIGALL cannot be signaled by the \texttt{raise()} function.
- \(^3\) If the value of the signal is SIGFPE, SIGILL or SIGSEGV the behavior is undefined.
- \(^4\) If the signal is hardware-generated, then the behavior undefined.

The following table shows the initial state of the C signal values and their handling action definitions with SYSIFCOPT(*ASYNC SIGNAL) is specified on the compilation command.

Table 16. Default Actions for Signal Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Default Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>2</td>
<td>Abnormal termination.</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>2</td>
<td>Arithmetic exceptions that are not masked, such as overflow, division by zero, and incorrect operation.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>2</td>
<td>Detection of an incorrect function image.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Interactive attention.</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>2</td>
<td>Incorrect access to storage.</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>2</td>
<td>Termination request sent to the program.</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>2</td>
<td>Intended for use by user applications.</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>2</td>
<td>Intended for use by user applications.</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>2</td>
<td>A timeout signal that is sent by \texttt{alarm()}.</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>2</td>
<td>A controlling terminal is hung up, or the controlling process ended.</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>1</td>
<td>A termination signal that cannot be caught or ignored.</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>3</td>
<td>A write to a pipe that is not being read.</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>2</td>
<td>A quit signal for a terminal.</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>3</td>
<td>An ended or stopped child process. SIGCLD is an alias name for this signal.</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>5</td>
<td>If stopped, continue.</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>4</td>
<td>A stop signal that cannot be caught or ignored.</td>
</tr>
<tr>
<td>SIGTSTOP</td>
<td>4</td>
<td>A stop signal for a terminal.</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>4</td>
<td>A background process attempted to read from a controlling terminal.</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>4</td>
<td>A background process attempted to write to a controlling terminal.</td>
</tr>
<tr>
<td>SIGIO</td>
<td>3</td>
<td>Completion of input or output.</td>
</tr>
</tbody>
</table>
Table 16. Default Actions for Signal Values (continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGURG</td>
<td>3  High bandwidth data is available at a socket.</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>2  Pollable event.</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>2  Specification exception.</td>
</tr>
<tr>
<td>SIGPRE</td>
<td>2  Programming exception.</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>2  Bad system call.</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>2  Trace or breakpoint trap.</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>2  Profiling timer expired.</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>2  Virtual timer expired.</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>2  Processor time limit exceeded.</td>
</tr>
<tr>
<td>SIGFSZ</td>
<td>2  File size limit exceeded.</td>
</tr>
<tr>
<td>SIGDANGER</td>
<td>2  System crash is imminent.</td>
</tr>
<tr>
<td>SIGPCANCEL</td>
<td>2  Thread termination signal that cannot be caught or ignored.</td>
</tr>
</tbody>
</table>

Default Actions:

1. End the process immediately
2. End the request.
3. Ignore the signal.
4. Stop the process.
5. Continue the process if it is currently stopped. Otherwise, ignore the signal.

Signal to iSeries Exception Mapping

The following table shows the system exception messages that are mapped to a signal. All *ESCAPE exception messages are mapped to signals. The *STATUS and *NOTIFY messages that map to SIGIO as defined in Table 14 on page 489 are mapped to signals.

Table 17. Signal to iSeries Exception Mapping

<table>
<thead>
<tr>
<th>Signal</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>C2M1601</td>
</tr>
<tr>
<td>SIGALL</td>
<td>C2M1610 (if explicitly raised)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>C2M1602, MCH1201 to MCH1204, MCH1206 to MCH1215, MCH1221 to MCH1224, MCH1838 to MCH1839</td>
</tr>
<tr>
<td>SIGILL</td>
<td>C2M1603, MCH10401, MCH1002, MCH1004, MCH1205, MCH1216 to MCH1219, MCH1801 to MCH1802, MCH1824 to MCH1825, MCH1832 to MCH1837, MCH1852 to MCH1854 to MCH1857, MCH1867, MCH2003 to MCH2004, MCH2202 to MCH2602, MCH2604, MCH2808, MCH2810 to MCH2811, MCH3201 to MCH3203, MCH4201 to MCH4211, MCH4213, MCH4296 to MCH4298, MCH4401 to MCH4403, MCH4406 to MCH4408, MCH4421, MCH4427 to MCH4428, MCH4801, MCH4804 to MCH4805, MCH5001 to MCH5003, MCH5401 to MCH5402, MCH5601, MCH6001 to MCH6002, MCH6201, MCH6208, MCH6216, MCH6220, MCH6403, MCH6601 to MCH6602, MCH6609 to MCH6612</td>
</tr>
<tr>
<td>SIGINT</td>
<td>C2M1604</td>
</tr>
</tbody>
</table>
Table 17. Signal to iSeries Exception Mapping (continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGIO</td>
<td>C2M1609, See Table 14 on page 489 for the exception mappings.</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>C2M1611 (if explicitly raised)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>C2M1605, MCH0201, MCH0601 to MCH0606, MCH0801 to MCH0803, MCH1001,</td>
</tr>
<tr>
<td></td>
<td>MCH1003, MCH1005 to MCH1006, MCH1220, MCH1401 to MCH1402, MCH1602,</td>
</tr>
<tr>
<td></td>
<td>MCH1604 to MCH1605, MCH1668, MCH1803 to MCH1806, MCH1809 to MCH1811,</td>
</tr>
<tr>
<td></td>
<td>MCH1813 to MCH1815, MCH1821 to MCH1823, MCH1826 to MCH1829, MCH1833,</td>
</tr>
<tr>
<td></td>
<td>MCH1836, MCH1848, MCH1850, MCH1851, MCH1864 to MCH1866, MCH1898,</td>
</tr>
<tr>
<td></td>
<td>MCH2001 to MCH2002, MCH2005 to MCH2006, MCH2201, MCH2203 to MCH2205,</td>
</tr>
<tr>
<td></td>
<td>MCH2401, MCH2601, MCH2603, MCH2605, MCH2801 to MCH2804, MCH2806 to</td>
</tr>
<tr>
<td></td>
<td>MCH2809, MCH3001, MCH3401 to MCH3408, MCH3410, MCH3601 to MCH3602,</td>
</tr>
<tr>
<td></td>
<td>MCH3603 to MCH3604, MCH3802, MCH4001 to MCH4002, MCH4010, MCH4212,</td>
</tr>
<tr>
<td></td>
<td>MCH4404 to MCH4405, MCH4416 to MCH4420, MCH4422 to MCH4426, MCH4429 to</td>
</tr>
<tr>
<td></td>
<td>MCH4437, MCH4601, MCH4802 to MCH4803, MCH4806 to MCH4812, MCH5201 to</td>
</tr>
<tr>
<td></td>
<td>MCH5204, MCH5602 to MCH5603, MCH5801 to MCH5804, MCH6203 to MCH6204,</td>
</tr>
<tr>
<td></td>
<td>MCH6206, MCH6217 to MCH6219, MCH6221 to MCH6222, MCH6401 to MCH6402,</td>
</tr>
<tr>
<td></td>
<td>MCH6404, MCH6603 to MCH6608, MCH6801</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>C2M1606</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>C2M1607</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>C2M1608</td>
</tr>
</tbody>
</table>

Cancel Handler Reason Codes

The following table lists the bits that are set in the reason code. If the activation group is to be stopped, then the activation group is stopped bit is also set in the reason code. These bits must be correlated to _CNL_MASK_T in _CNL_Hndlr_Parms_T in <except.h>. Column 2 contains the macro constant defined for the cancel reason mask in <except.h>.

Table 18. Determining Canceled Invocation Reason Codes

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits set in reason code</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library routines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>_EXIT_VERB</td>
<td>The definition of exit is normal end of processing, and therefore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invocations canceled by this function is done with a reason code of normal.</td>
</tr>
<tr>
<td>abort</td>
<td>_ABNORMAL_TERM _EXIT_VERB</td>
<td>The definition of abort is abnormal end of processing, and therefore</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invocations canceled by this function are done with a reason code of abnormal.</td>
</tr>
<tr>
<td>longjmp</td>
<td>_JUMP</td>
<td>The general use of the longjmp() function is to return from an exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>handler, although it may be used in non-exception situations as well. It</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is used as part of the “normal” path for a program, and therefore any</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invocations canceled because of it are cancelled with a reason code of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>normal.</td>
</tr>
<tr>
<td>Unhandled function</td>
<td><em>ABNORMAL_TERM</em> UNHANDLED_EXCP</td>
<td>Not handling an exception which is an abnormal situation.</td>
</tr>
<tr>
<td>check</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 18. Determining Canceled Invocation Reason Codes (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits set in reason code</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System APIs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEEMRCR</td>
<td>_ABNORMAL_TERM</td>
<td>This API is only used during exception processing. It is typically used to cancel invocations where a resume is not possible, or at least the behavior would be undefined if control was resumed in them. Also, these invocations have had a chance to handle the exception but did not do so. Invocations canceled by this API are done with reason code of abnormal.</td>
</tr>
<tr>
<td>QMHSNDPM</td>
<td>_ABNORMAL_TERM</td>
<td>All invocations down to the target invocation are canceled without any chance of handling the exception. The API topic contains information on these APIs.</td>
</tr>
<tr>
<td>QMHRSNEM</td>
<td>_ABNORMAL_TERM</td>
<td></td>
</tr>
<tr>
<td><strong>Message Handler APIs</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iSeries commands</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process end</td>
<td>_ABNORMAL_TERM</td>
<td>Any externally initiated shutdown of an activation group is considered abnormal.</td>
</tr>
<tr>
<td>RCLACTGRP</td>
<td>_ABNORMAL_TERM</td>
<td>The default is abnormal termination. The termination could be normal if a normal/abnormal flag is added to the command.</td>
</tr>
<tr>
<td>RCLRSC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 19. Common Reason Code for Cancelling Invocations

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Header File Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0</td>
<td>Reserved</td>
<td>_EXCP_SENT</td>
</tr>
<tr>
<td>Bits 1</td>
<td>Invocation canceled due to sending exception message</td>
<td>_ABNORMAL_TERM</td>
</tr>
<tr>
<td>Bits 2-15</td>
<td>Reserved</td>
<td>_AG_TERMINATING</td>
</tr>
<tr>
<td>Bit 16</td>
<td>0 - normal end of process 1 - abnormal end of process</td>
<td>_EXCP_SENT</td>
</tr>
<tr>
<td>Bit 17</td>
<td>Activation Group is ending.</td>
<td>_AG_TERMINATING</td>
</tr>
<tr>
<td>Bit 18</td>
<td>Initiated by Reclaim Activation Group (RCLACTGRP)</td>
<td>_RCLRSC</td>
</tr>
<tr>
<td>Bit 19</td>
<td>Initiated by the process end.</td>
<td>_PROCESS_TERM</td>
</tr>
<tr>
<td>Bit 20</td>
<td>Initiated by an exit() function.</td>
<td>_EXIT_VERB</td>
</tr>
<tr>
<td>Bit 21</td>
<td>Initiated by an unhandled function check.</td>
<td>_UNHANDLED_EXCP</td>
</tr>
<tr>
<td>Bit 22</td>
<td>Invocation canceled due to a longjmp() function.</td>
<td>_JUMP</td>
</tr>
<tr>
<td>Bit 23</td>
<td>Invocation canceled due to a jump because of exception processing.</td>
<td>_JUMP_EXCP</td>
</tr>
<tr>
<td>Bits 24-31</td>
<td>Reserved (0)</td>
<td></td>
</tr>
</tbody>
</table>
**Exception Classes**

In a CL program, you can monitor for a selected group of exceptions, or a single exception, based on the exception identifier. The only class values the exception handler will monitor for are _C2_MH_ESCAPE, _C2_MH_STATUS, _C2_MH_NOTIFY, and _C2_MH_FUNCTION_CHECK. For more information about using the #pragma exception handler directive, see the WebSphere Development Studio: ILE C/C++ Compiler Reference. This table defines all the exception classes you can specify.

*Table 20. Exception Classes*

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Header File Constant in &lt;except.h&gt;</th>
<th>Exception class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_C1_BINARY_OVERFLOW</td>
<td>Binary overflow or divide by zero</td>
</tr>
<tr>
<td>1</td>
<td>_C1_DECIMAL_OVERFLOW</td>
<td>Decimal overflow or divide by zero</td>
</tr>
<tr>
<td>2</td>
<td>_C1_DECIMAL_DATA_ERROR</td>
<td>Decimal data error</td>
</tr>
<tr>
<td>3</td>
<td>_C1_FLOAT_OVERFLOW</td>
<td>Floating-point overflow or divide by zero</td>
</tr>
<tr>
<td>4</td>
<td>_C1_FLOAT_UNDERFLOW</td>
<td>Floating-point underflow or inexact result</td>
</tr>
<tr>
<td>5</td>
<td>_C1_INVALID_FLOAT_OPERAND</td>
<td>Floating-point invalid operand or conversion error</td>
</tr>
<tr>
<td>6</td>
<td>_C1_OTHER_DATA_ERROR</td>
<td>Other data error, for example edit mask</td>
</tr>
<tr>
<td>7</td>
<td>_C1_SPECIFICATION_ERROR</td>
<td>Specification (operand alignment) error</td>
</tr>
<tr>
<td>8</td>
<td>_C1_POINTER_NOT_VALID</td>
<td>Pointer not set/pointer type invalid</td>
</tr>
<tr>
<td>9</td>
<td>_C1_OBJECT_NOT_FOUND</td>
<td>Object not found</td>
</tr>
<tr>
<td>10</td>
<td>_C1_OBJECT_DESTROYED</td>
<td>Object destroyed</td>
</tr>
<tr>
<td>11</td>
<td>_C1_ADDRESS_COMP_ERROR</td>
<td>Address computation underflow or overflow</td>
</tr>
<tr>
<td>12</td>
<td>_C1_SPACE_ALLOC_ERROR</td>
<td>Space not allocated at specified offset</td>
</tr>
<tr>
<td>13</td>
<td>_C1 DOMAIN_OR_STATE_VIOLATION</td>
<td>Domain/State protection violation</td>
</tr>
<tr>
<td>14</td>
<td>_C1 AUTHORIZATION_VIOLATION</td>
<td>Authorization violation</td>
</tr>
<tr>
<td>15</td>
<td>_C1 JAVA_THROWN_CLASS</td>
<td>Exception thrown for a Java™ class</td>
</tr>
<tr>
<td>16-28</td>
<td>_C1_VLIC_RESERVED</td>
<td>VLIC reserved</td>
</tr>
<tr>
<td>29</td>
<td>_C1 OTHER_MI_EXCEPTION</td>
<td>Remaining MI-generated exceptions (other than function check)</td>
</tr>
<tr>
<td>30</td>
<td>_C1_MI_GEN_FC_OR_MC</td>
<td>MI-generated function check or machine check</td>
</tr>
<tr>
<td>31</td>
<td>_C1_MI_SIGEXP_EXCEPTION</td>
<td>Message generated via Signal Exception instruction</td>
</tr>
</tbody>
</table>
Table 20. Exception Classes (continued)

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Header File Constant in &lt;except.h&gt;</th>
<th>Exception class</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-39</td>
<td>n/a</td>
<td>reserved</td>
</tr>
<tr>
<td>40</td>
<td>_C2_MH_ESCAPE</td>
<td>*ESCAPE</td>
</tr>
<tr>
<td>41</td>
<td>_C2_MH_NOTIFY</td>
<td>*NOTIFY</td>
</tr>
<tr>
<td>42</td>
<td>_C2_MH_STATUS</td>
<td>*STATUS</td>
</tr>
<tr>
<td>43</td>
<td>_C2_MH_FUNCTION_CHECK</td>
<td>function check</td>
</tr>
<tr>
<td>44-63</td>
<td>n/a</td>
<td>reserved</td>
</tr>
</tbody>
</table>

Data Type Compatibility

Each high-level language has different data types. When you want to pass data between programs that are written in different languages, you must be aware of these differences.

Some data types in the ILE C programming language have no direct equivalent in other languages. However, you can simulate data types in other languages that use ILE C data types.

The following table shows the ILE C data type compatibility with ILE RPG.

Table 21. ILE C Data Type Compatibility with ILE RPG

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE RPG D spec, columns 33 to 39</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>nA</td>
<td>n</td>
<td>An array of characters where n=1 to 32766.</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>1A</td>
<td>1</td>
<td>An Indicator that is a variable starting with *IN.</td>
</tr>
<tr>
<td>char[n]</td>
<td>nS 0</td>
<td>n</td>
<td>A zoned decimal.</td>
</tr>
<tr>
<td>char[2n+2]</td>
<td>Not supported.</td>
<td>2n+2</td>
<td>A graphic data type.</td>
</tr>
<tr>
<td>_Packed struct [short i; char[n]]</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>D</td>
<td>8, 10</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>T</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>Z</td>
<td>26</td>
<td>A timestamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>5I 0</td>
<td>2</td>
<td>An integer field.</td>
</tr>
<tr>
<td>short unsigned int</td>
<td>5U 0</td>
<td>2</td>
<td>An unsigned integer field.</td>
</tr>
<tr>
<td>int</td>
<td>10I 0</td>
<td>4</td>
<td>An integer field.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>10U 0</td>
<td>4</td>
<td>An unsigned integer field</td>
</tr>
<tr>
<td>long int</td>
<td>10I 0</td>
<td>4</td>
<td>An integer field.</td>
</tr>
<tr>
<td>long unsigned int</td>
<td>10U 0</td>
<td>4</td>
<td>An unsigned integer field.</td>
</tr>
</tbody>
</table>
Table 21. ILE C Data Type Compatibility with ILE RPG (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE RPG D spec, columns 33 to 39</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte unsigned integer, a bitfield.</td>
</tr>
<tr>
<td>float</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>n/p</td>
<td>n/2+1</td>
<td>A packed decimal. n must be less than or equal to 30.</td>
</tr>
<tr>
<td>union.element</td>
<td>&lt;type&gt; with keyword OVERLAY(longest field)</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>&lt;type&gt; with keyword DIM(n)</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>data structure</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>* with keyword PROCPTR</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with ILE COBOL.

Table 22. ILE C Data Type Compatibility with ILE COBOL

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n] char *</td>
<td>PIC X(n).</td>
<td>n</td>
<td>An array of characters where n=1 to 3,000,000</td>
</tr>
<tr>
<td>char</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>An indicator.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) DISPLAY</td>
<td>n</td>
<td>A zoned decimal.</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>PIC G(n)</td>
<td>2n</td>
<td>A graphic data type.</td>
</tr>
<tr>
<td>_Packed struct [short i; char[n]]</td>
<td>05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>5</td>
<td>A day field.</td>
</tr>
<tr>
<td>char</td>
<td>PIC X.</td>
<td>1</td>
<td>A day-of-week field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4.</td>
<td>2</td>
<td>A 2-byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) BINARY.</td>
<td>2</td>
<td>A 2-byte signed integer with a range of -9999 to +9999.</td>
</tr>
</tbody>
</table>
The following table shows the ILE C data type compatibility with ILE CL.

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) BINARY.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>int</td>
<td>USAGE IS INDEX</td>
<td>4</td>
<td>A 4-byte integer.</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) BINARY.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>PIC 9(9) COMP-4.</td>
<td>4</td>
<td>Bitfields can be manipulated using hex literals.</td>
</tr>
<tr>
<td>float</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p)v9(p) COMP-3</td>
<td>n/2+1</td>
<td>A packed decimal.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p) 9(p) PACKED-DECIMAL</td>
<td>n/2+1</td>
<td>A packed decimal.</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>OCCURS</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>01 record</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.</td>
</tr>
<tr>
<td></td>
<td>05 field1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>05 field2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointer to function</td>
<td>PROCEDURE-POINTER</td>
<td>16</td>
<td>A 16 byte pointer to a procedure.</td>
</tr>
<tr>
<td>Not supported.</td>
<td>PIC S9(18) COMP-4.</td>
<td>8</td>
<td>An 8 byte integer.</td>
</tr>
<tr>
<td>Not supported.</td>
<td>PIC S9(18) BINARY.</td>
<td>8</td>
<td>An 8 byte integer.</td>
</tr>
<tr>
<td>ILE C declaration in prototype</td>
<td>CL</td>
<td>Length</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>char[n]</td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>An array of characters where n=1 to 32766. A null-terminated string. For example, CHGVAR &amp;V1 VALUE (&amp;V *TCAT X'00') where &amp;V1 is one byte bigger than &amp;V.</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>*LGL</td>
<td>1</td>
<td>Holds ‘1’ or ‘0’.</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>integer types</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>A 1-, 2-, or 4- byte signed or unsigned integer.</td>
</tr>
<tr>
<td>float constants</td>
<td>CL constants only.</td>
<td>4</td>
<td>A 4- or 8-byte floating point.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>*DEC</td>
<td>n/2+1</td>
<td>A packed decimal. The limit of n is 15 and p is 9.</td>
</tr>
<tr>
<td>union.element</td>
<td>Not supported.</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>struct</td>
<td>Not supported.</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with OPM RPG/400:

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM RPG/400 I spec, DS subfield columns spec</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>1 10</td>
<td>n</td>
<td>An array of characters where n=1 to 32766.</td>
</tr>
<tr>
<td>char *</td>
<td>1 INxxxx</td>
<td>1</td>
<td>An Indicator that is a variable starting with *IN.</td>
</tr>
<tr>
<td>char[n]</td>
<td>1 nd (d&gt;=0)</td>
<td>n</td>
<td>A zoned decimal. The limit of n is 30.</td>
</tr>
<tr>
<td>char[2n+2]</td>
<td>Not supported.</td>
<td>2n+2</td>
<td>A graphic data type.</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>Not supported.</td>
<td>6, 8, 10</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>Not supported.</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>Not supported.</td>
<td>26</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>B 1 20</td>
<td>2</td>
<td>A 2-byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>int</td>
<td>B 1 40</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -9999999999 to +9999999999.</td>
</tr>
</tbody>
</table>
### Table 24. ILE C Data Type Compatibility with OPM RPG/400 (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM RPG/400 I spec, DS subfield columns spec</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>long int</td>
<td>B 1 40</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>struct [unsigned int : n]x;</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte unsigned integer, a bitfield.</td>
</tr>
<tr>
<td>float</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>Not supported.</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>P 1 n/2+1d</td>
<td>n/2+1</td>
<td>A packed decimal. n must be less than or equal to 30.</td>
</tr>
<tr>
<td>union.element</td>
<td>data structure subfield</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>E-SPEC array</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>data structure</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16 byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with OPM COBOL/400.

### Table 25. ILE C Data Type Compatibility with OPM COBOL/400

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>PIC X(n)</td>
<td>n</td>
<td>An array of characters where n=1 to 3,000,000</td>
</tr>
<tr>
<td>char *</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>An indicator.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) USAGE IS DISPLAY</td>
<td>n</td>
<td>A zoned decimal. The limit of n is 18.</td>
</tr>
<tr>
<td>_Packed struct [short i; char[n]]</td>
<td>05 VL-FIELD.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6, 8, 10</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4.</td>
<td>2</td>
<td>A 2 byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -9999999999 to +9999999999.</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999999999 to +999999999999999.</td>
</tr>
</tbody>
</table>
Table 25. ILE C Data Type Compatibility with OPM COBOL/400 (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>PIC 9(9) COMP-4. PIC X(4).</td>
<td>4</td>
<td>Bitfields can be manipulated using hex literals.</td>
</tr>
<tr>
<td>float</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p)V9(p) COMP-3</td>
<td>n/2+1</td>
<td>A packed decimal. The limits of n and p are 18.</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES element length</td>
<td>An element of a union.</td>
<td></td>
</tr>
<tr>
<td>data_type[n]</td>
<td>OCCURRS</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>01 record</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.</td>
</tr>
</tbody>
</table>

pointer to function Not supported. 16 A 16-byte pointer.

Not supported. PIC S9(18) COMP-4. 8 An 8 byte integer.

The following table shows the ILE C data type compatibility with CL.

Table 26. ILE C Data Type Compatibility with CL

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>CL</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n] char *</td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>An array of characters where n=1 to 32766. A null terminated string. For example, CHGVAR &amp;V1 VALUE (&amp;V *TCAT X'00') where &amp;V1 is one byte bigger than &amp;V. The limit of n is 9999.</td>
</tr>
<tr>
<td>char</td>
<td>*LGL</td>
<td>1</td>
<td>Holds ‘ ’ or ‘ ’.</td>
</tr>
<tr>
<td>_Packed struct [short int; char[n]]</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>integer types</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>A 1-, 2- or 4- byte signed or unsigned integer.</td>
</tr>
<tr>
<td>float constants</td>
<td>CL constants only.</td>
<td>4</td>
<td>A 4- or 8- byte floating point.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>*DEC</td>
<td>n/2+1</td>
<td>A packed decimal. The limit of n is 15 and p is 9.</td>
</tr>
<tr>
<td>union.element</td>
<td>Not supported.</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>struct</td>
<td>Not supported.</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
</tbody>
</table>
Table 26. ILE C Data Type Compatibility with CL (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>CL</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows how arguments are passed from a command line CL call to an ILE C program.

Table 27. Arguments Passed From a Command Line CL Call to an ILE C Program

<table>
<thead>
<tr>
<th>Command Line Argument</th>
<th>Argv Array</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td>argv[1..255]</td>
<td>normal parameters</td>
<td></td>
</tr>
<tr>
<td>’123.4’</td>
<td>argv[1] &quot;123.4&quot;</td>
<td></td>
</tr>
<tr>
<td>123.4</td>
<td>argv[2] 0000000123.40000D</td>
<td></td>
</tr>
<tr>
<td>’Hi’</td>
<td>argv[3] &quot;Hi&quot;</td>
<td></td>
</tr>
<tr>
<td>Lo</td>
<td>argv[4] &quot;LO&quot;</td>
<td></td>
</tr>
</tbody>
</table>

A CL character array (string) will not be NULL-ended when passed to an ILE C program. A C program that will receive such arguments from a CL program should not expect the strings to be NULL-ended. You can use the QCMDEXC to ensure that all the arguments will be NULL-ended.

The following table shows how CL constants are passed from a compiled CL program to an ILE C program.

Table 28. CL Constants Passed from a Compiled CL Program to an ILE C Program

<table>
<thead>
<tr>
<th>Compile CL Program Argument</th>
<th>Argv Array</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td>argv[1..255]</td>
<td>normal parameters</td>
<td></td>
</tr>
<tr>
<td>’123.4’</td>
<td>argv[1] &quot;123.4&quot;</td>
<td></td>
</tr>
<tr>
<td>123.4</td>
<td>argv[2] 0000000123.40000D</td>
<td></td>
</tr>
<tr>
<td>’Hi’</td>
<td>argv[3] &quot;Hi&quot;</td>
<td></td>
</tr>
<tr>
<td>Lo</td>
<td>argv[4] &quot;LO&quot;</td>
<td></td>
</tr>
</tbody>
</table>

A command processing program (CPP) passes CL constants as defined in Table 28. You define an ILE C program as a command processing program when you create your own CL command with the Create Command (CRTCMD) command to call the ILE C program.

The following table shows how CL variables are passed from a compiled CL program to an ILE C program. All arguments are passed by reference from CL to C.

Table 29. CL Variables Passed from a Compiled CL Program to an ILE C Program

<table>
<thead>
<tr>
<th>CL Variables</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL VAR(&amp;v) TYPE('CHAR) LEN(10) VALUE('123.4')</td>
<td>123.4</td>
</tr>
</tbody>
</table>
Table 29. CL Variables Passed from a Compiled CL Program to an ILE C Program (continued)

<table>
<thead>
<tr>
<th>CL Variables</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL VAR(&amp;d) TYPE(*DEC) LEN(10)</td>
<td>000000123.40000D</td>
</tr>
<tr>
<td>VALUE(123.4)</td>
<td></td>
</tr>
<tr>
<td>DCL VAR(&amp;h) TYPE(*CHAR) LEN(10)</td>
<td>Hi</td>
</tr>
<tr>
<td>VALUE('Hi')</td>
<td></td>
</tr>
<tr>
<td>DCL VAR(&amp;i) TYPE(*CHAR) LEN(10)</td>
<td>LO</td>
</tr>
<tr>
<td>VALUE(Lo)</td>
<td></td>
</tr>
<tr>
<td>DCL VAR(&amp;j) TYPE(*LGL) LEN(1)</td>
<td>1</td>
</tr>
<tr>
<td>VALUE('1')</td>
<td></td>
</tr>
</tbody>
</table>

CL variables and numeric constants are not passed to an ILE C program with null-ended strings. Character constants and logical literals are passed as null-ended strings, but are not padded with blanks. Numeric constraints such as packed decimals are passed as 15,5 (8 bytes).

Run-time Character Set

Each EBCDIC CCSID consists of two character types: invariant characters and variant characters.

The following table identifies the hexadecimal representation of the invariant characters in the C character set.

### Table 30. Invariant Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Hexadecimal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>0x4b</td>
</tr>
<tr>
<td>&lt;</td>
<td>0x4c</td>
</tr>
<tr>
<td>(</td>
<td>0x4d</td>
</tr>
<tr>
<td>+</td>
<td>0x4e</td>
</tr>
<tr>
<td>&amp;</td>
<td>0x50</td>
</tr>
<tr>
<td>*</td>
<td>0x5c</td>
</tr>
<tr>
<td>)</td>
<td>0x5d</td>
</tr>
<tr>
<td>;</td>
<td>0x5e</td>
</tr>
<tr>
<td>-</td>
<td>0x60</td>
</tr>
<tr>
<td>!</td>
<td>0x6a</td>
</tr>
<tr>
<td>%</td>
<td>0x6c</td>
</tr>
<tr>
<td>&gt;</td>
<td>0x6d</td>
</tr>
<tr>
<td>?</td>
<td>0x6e</td>
</tr>
<tr>
<td>:</td>
<td>0x6f</td>
</tr>
<tr>
<td>;</td>
<td>0x7a</td>
</tr>
<tr>
<td>@</td>
<td>0x7c</td>
</tr>
<tr>
<td>`</td>
<td>0x7d</td>
</tr>
<tr>
<td>`</td>
<td>0x7f</td>
</tr>
<tr>
<td>a-i</td>
<td>0x81 - 0x89</td>
</tr>
<tr>
<td>j-r</td>
<td>0x91 - 0x99</td>
</tr>
<tr>
<td>s-z</td>
<td>0xa2 - 0xa9</td>
</tr>
<tr>
<td>A-I</td>
<td>0xc1 - 0xc9</td>
</tr>
<tr>
<td>J-R</td>
<td>0xd1 - 0xd9</td>
</tr>
<tr>
<td>S-Z</td>
<td>0xe2 - 0xef</td>
</tr>
<tr>
<td>0-9</td>
<td>0xf0 - 0xf9</td>
</tr>
<tr>
<td>\a'</td>
<td>0x2f</td>
</tr>
<tr>
<td>\b'</td>
<td>0x16</td>
</tr>
<tr>
<td>\t'</td>
<td>0x05</td>
</tr>
<tr>
<td>\n'</td>
<td>0x0b</td>
</tr>
<tr>
<td>\f'</td>
<td>0x0c</td>
</tr>
</tbody>
</table>

The following table identifies the hexadecimal representation of the variant characters in the C character set for the most commonly used CCSIDs.

### Table 31. Variant Characters in Different CCSIDs

<table>
<thead>
<tr>
<th>CCSID</th>
<th>0x4f</th>
<th>0x5a</th>
<th>0x5f</th>
<th>0xe0</th>
<th>0x79</th>
<th>0x7b</th>
<th>0xba</th>
<th>0xbb</th>
<th>0xb0</th>
<th>0xce0</th>
<th>0xdd0</th>
<th>0x5b</th>
</tr>
</thead>
<tbody>
<tr>
<td>037</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>273</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>277</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>278</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 31. Variant Characters in Different CCSIDs (continued)

| CC-SID | | | | | | | | | | | | | | |
|--------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| 284    | 0x4f       | 0xbb       | 0x5f       | 0xe0       | 0x79       | 0x69       | 0xbd       | 0x4a       | 0x5a       | 0xba       | 0xc0       | 0xd0       | 0x61       | 0xb0       | 0x5b       |
| 285    | 0x4f       | 0x5a       | 0x5f       | 0xe0       | 0x79       | 0x7b       | 0xbc       | 0xb1       | 0xbb       | 0xba       | 0xc0       | 0xd0       | 0x61       | 0xb0       | 0x4a       |
| 297    | 0xbb       | 0x4f       | 0xba       | 0x48       | 0xa0       | 0xb1       | 0xbd       | 0x90       | 0x65       | 0x5f       | 0x51       | 0x54       | 0x61       | 0xb0       | 0x5b       |
| 500    | 0xbb       | 0x4f       | 0xba       | 0xe0       | 0x79       | 0x7b       | 0xa1       | 0x4a       | 0x5a       | 0x5f       | 0xc0       | 0xd0       | 0x61       | 0xb0       | 0x5b       |

See the Globalization topic for more information on coding variant characters in the other IBM CCSIDs.

Asynchronous Signal Model

The Asynchronous Signal Model (ASM) is used when the SYSIFCOPT(*ASYNCSIGNAL) option is specified on the compilation. It is intended for compatibility with applications ported from the UNIX operating system. In the ASM model, the `signal()` and `raise()` functions are implemented using the OS/400 Signal APIs described in the API topic under the Programming heading in the Information Center.

OS/400 exceptions sent to an ASM program are converted to asynchronous signals. The exceptions are processed by an asynchronous signal handler.

Modules compiled to use the ASM can be intermixed with modules using the Original Signal Model (OSM) in the same processes, programs, and service programs. This is true even when SYSIFCOPT(*ASYNCSIGNAL) is not specified on the compilation command for programs compiled with the OSM. There are several differences between the two signal models:

- The OSM is based on exceptions, while the ASM is based on asynchronous signals.
- Under the OSM, the signal vector and signal mask are scoped to the activation group. Under ASM, there is one signal vector per process and one signal mask per thread. Both types of signal vectors and signal masks are maintained during compilation.
- The same asynchronous signal vector and signal masks are operated on by all ASM modules in a thread, regardless of the activation group the signal vector and signal masks belong to. You should save and restore the state of the signal vector and signal masks to ensure that they are not changed by any ASM modules. The OSM does not use the asynchronous signal vector and signal masks.
- Signals raised by OSM programs will be sent as exceptions. Under the OSM, the exceptions are received and handled by the `_C_exception_router` function, which directly invokes the OSM signal handler of the the user. Asynchronous signals are not mapped to exceptions, and are not handled by signal handlers registered under the OSM. Under the ASM, the exceptions are received and handled by the `_C_async_exception_handler` function, which maps the exception to an asynchronous signal, and uses the SNDSIG MI instruction to raise an asynchronous signal. An ASM signal handler receives control from the OS/400 asynchronous signal component.

When an OSM program raises a signal, the generated exception percolates up the invocation stack until it finds an exception monitor. If the prior invocation is
an OSM function, the \texttt{\_C\_exception\_router} catches the exception and performs
the OSM signal action. The ASM signal handler does not receive the signal.
If the prior invocation is an ASM function, the \texttt{\_C\_async\_exception\_router}
handles the exception, maps to an asynchronous signal, and raises it using the
MI \texttt{SNDSIG} instruction. The handling of the asynchronous signal then depends
on the asynchronous signal vector and mask state of the thread, as defined in
the OS/400 Signal Management topic.
If the OSM program raising the signal is running in the same activation group
with the ASM program, the exception will be mapped to an asynchronous signal
using the mapping described previously. The signal ID is preserved when the
exception is mapped to a signal. So, signal handlers registered with the
asynchronous signal model will be able to receive signals generated under the
original signal model. Use this approach to integrate two programs using
different signal models.
If a program or service program using the OSM is running a different activation
group, any signal that is unmonitored in that activation group will cause the
termination of that program and activation group. The unmonitored signal is
then percolated to the calling program as a CEE9901 exception. The CEE9901
exception is mapped to a SIGSEGV asynchronous signal.

\textbf{Note:} This approach is not recommended.

\begin{itemize}
\item Under the ASM, the C functions \texttt{raise()} and \texttt{signal()} are integrated with the
system signal functions, such as \texttt{kill()} and \texttt{sigaction()}. These two sets of APIs
can be used interchangeably. This cannot be done under the OSM.
\item A user-specified exception monitor established with \#pragma exception\_handler
has precedent over the compiler-generated monitor, which invokes
\texttt{\_C\_async\_exception\_router}. This precedence enables you to bypass
the asynchronous signal generating exception monitor in some situations.
\item The \texttt{\_GetExcData()} function is not available under the ASM to retrieve the
exception ID associated with the signal. If an extended signal handler is
established using the sigaction() function, however, it can access the exception
information from the signal-specific data structure. For more information, see
\texttt{\_GetExcData()} — Get Exception Data” on page 145.
\end{itemize}

\section*{Unicode Support}

The Unicode Standard is a standardized character code designed to encode
international texts for display and storage. It uses a unique 16- or 32-bit value to
represent each individual character, regardless of platform, language, or program.
Using Unicode, you can develop a software product that will work with various
platforms, languages, and countries or regions. Unicode also allows data to be
transported through many different systems.

There are two different forms of Unicode support available from the compiler and
run-time. This section describes the two forms of Unicode support as well as some
of the features of and considerations for using that support. To obtain additional
information about Unicode, visit the \texttt{Unicode Home Page} at www.unicode.org.

The first type of Unicode support is UCS2 support. When the
\texttt{LOCALETYPE(*\texttt{LOCALEUCS2})} option is specified on the compilation command,
the compiler and run-time use wide characters (that is, characters of the \texttt{wchar\_t}
type) and wide character strings (that is, strings of the \texttt{wchar\_t\_t * type} that
represent two-byte Unicode characters. Narrow (non-wide) characters and narrow
character strings represent EBCDIC characters, just as they do when the UCS2 support is not enabled. The Unicode characters represent codepoints in CCSID 13488.

The second type of Unicode support is UTF-8/UTF-32 support, or UTF support. When the LOCALETYPE(*LOCALEUTF) option is specified on the compilation command, the compiler and run-time use wide characters and wide character strings that represent four-byte Unicode characters. Each four-byte character represents a single UTF-32 character. Narrow characters and narrow character strings represent UTF-8 characters. Each UTF-8 character is from one to four bytes in size. Most normal characters are a single byte in size, and, in fact, all 7-bit ASCII characters map directly to UTF-8 and are one byte in size. The UTF-32 characters represent codepoints in CCSID 1232. The UTF-8 characters represent codepoints in CCSID 1208.

When the UTF support is enabled, not only do the wide characters become UTF-32 Unicode, but the narrow characters become UTF-8 Unicode as well. As an example, consider the following HelloWorld program.

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

When this program is compiled with UTF support, the character string is stored within the program as UTF-8 characters and not EBCDIC characters. The printf() function knows this and is able to parse the UTF-8 characters and generate the output as expected. However, if this program called some other user-supplied routine that did not know how to handle UTF-8 characters, the other routine might yield incorrect results or behavior.

### Reasons to Use Unicode Support

There are two main situations in which you may want to use Unicode support for your application. The first situation is if your application is an international application and requires support for several different languages. The Unicode character set provides an easy way to allow a single application to easily handle any language or character set. The application can perform all input, processing, and output using Unicode characters. Another situation for using Unicode support is for porting a 7 bit ASCII application. Because the UTF-8 character set is a superset of 7 bit ASCII, an ASCII application can be ported more easily to a UTF-8 environment than to an EBCDIC environment.

### Pseudo CCSID Neutrality

When a program is compiled with UTF support, the run-time allows more than just UTF-8 characters, and it essentially becomes CCSID neutral. The run-time will handle whatever CCSID is contained within the current locale. By default, when a program is compiled with UTF support, the locale loaded is a UTF-8 (CCSID 1208) locale. This allows the run-time to handle CCSID 1208. If the setlocale() routine is called to set the locale to an EBCDIC locale (for example, a CCSID 37 locale), the run-time will handle CCSID 37. This, along with the #pragma convert support within the compiler, can be used to provide international application support. Here is an example:

```c
#include <stdio.h>
#include <locale.h>
```
int main() {
    /* This string is in CCSID 1208 */
    printf("Hello World\n");

    /* Change locale to a CCSID 37 locale */
    setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
    #pragma convert(37)

    /* This string is in CCSID 37 */
    printf("Hello World\n");

    return 0;
}

Standard Files

When using the UTF support, the default standard input and output files stdin, stdout, and stderr have some special processing done for them by the run-time. Since a program using UTF support contains data in UTF-8 and the standard files interact with the screen and spool files, there is a potential mismatch in data. The screen and spool file routines are provided by the operating system and thus expect EBCDIC. For stdout and stderr, the run-time will automatically convert UTF-8 data to EBCDIC. For stdin, the run-time will automatically convert the incoming EBCDIC to UTF-8 data.

Considerations

Because the default environment for OS/400 is primarily an EBCDIC environment, there are several things that you must be aware of when using UTF support in an application.

If a program or service program has some modules compiled with the UTF support and some modules compiled without the UTF support, care must be taken to ensure that unexpected mismatches do not occur. The wide characters and wide character strings are two bytes in size for a non-UTF module and four bytes in size for a UTF module, so sharing wide characters between the modules may not work correctly. The narrow (non-wide) characters and character strings are in job CCSID for a non-UTF module and in CCSID 1208 for a UTF module, so sharing narrow characters between the modules may not work correctly either.

Whenever a setlocale() is performed to set the locale to a different CCSID, the standard output files should be flushed to avoid buffering problems with character data containing multiple CCSIDs. Since stdout is line buffered by default, if each output line ends in a newline character, the problem will not occur. However, if this is not done, the output may not be shown as intended. The following example illustrates the problem.

#include <stdio>
#include <locale.h>

int main() {
    /* This string is in CCSID 1208 */
    printf("Hello World");

    /* Change locale to a CCSID 37 locale */
    setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
    #pragma convert(37)

    /* This string is in CCSID 37 */
printf("Hello World\n");
return 0;
}

In this case, the first printf() causes the CCSID 1208 string “Hello World” to be copied to the stdout buffer. Before the setlocale() is done, stdout should be flushed to copy that string to the screen. The second printf() causes the CCSID 37 string “Hello World\n” to be copied to the stdout buffer. Because of the trailing newline character, the buffer is flushed at that point and the whole buffer is copied to the screen. Because the CCSID of the current locale is 37 and the screen can handle CCSID 37 without problems, the whole buffer is copied without conversion. The CCSID 1208 characters are displayed as unreadable characters. If a flush had been done, the CCSID 1208 characters would have been converted to CCSID 37 and would have been displayed correctly.

Nearly all of the run-time routines have been modified to support UTF, but there are a handful of them that have not. Routines and structures that deal with exception handling, such as the _GetExcData() function, the _EXCP_MSGID variable, and the exception handler structure _INTRPT_Hndlr_Parms_T are provided by the operating system, not the run-time. They are strictly EBCDIC. The getenv() and putenv() functions handle only EBCDIC. The QXXCHGDA() and QXXRTVDA() functions handle only EBCDIC. The argv and envp parameters are also EBCDIC only.

Some of the record I/O routines (that is, functions beginning with _R) do not completely support UTF. The routines that do not support UTF are _Rformat(), _Rcommit(), _Racquire(), _Rrelease(), _Rpgmdev(), _Rindara(), and _Rdevatr(). They are available when compiling with the UTF option, but they accept and generate only EBCDIC. In addition, any character data within the structures returned by the _R functions will be in EBCDIC rather than UTF.

Other operating system routines have not been modified to support UTF. For example, the IFS routines such as open() still accept job CCSID. Other operating system APIs also still accept job CCSID. For UTF applications, the characters and character strings provided to these routines will need to be converted to job CCSID using QTQCVRT, iconv(), #pragma convert, or some other method.

**Default File CCSID**

When the fopen() function is used to open files, the default CCSID of the file is different depending on whether or not UTF support is used. If UTF support is not used (that is, if LOCALETYPE(\*CLD), LOCALETYPE(\*LOCALE), or LOCALETYPE(\*LOCALEUCS2) are specified on the compilation command), the file CCSID defaults to the current job CCSID. Usually this works well because the job CCSID is set correctly and the current locale is set to match the job CCSID.

With UTF support, the job CCSID cannot be set to UTF-8 because of system limitations. When LOCALETYPE(\*LOCALEUTF) is specified, the file CCSID defaults to the CCSID of the current locale. If the default locale is being used, the CCSID defaults to UTF-8 (CCSID 1208). If this default is not desired, the ccsid or o_ccsid keyword can be specified in the second parameter of the fopen() call. However, database files are an exception, because DB2 UDB for iSeries does not completely support UTF-8. When SYSIFCOPT(\*NOIFSIO) is specified, and the CCSID of the current locale is 1208, the CCSID of the file defaults to CCSID 65535
(no conversion) rather than CCSID 1208. This allows CCSID 1208 to be used with database files. For more information about file CCSIDs, see "fopen() — Open Files" on page 100.

Newline Character

When the UTF support is not used, the hex value generated by the compiler for the character \n and used by the run-time has two different values. The hex value 0x15 is used if SYSIFCOPT(*NOIFSIO) is specified on the compilation command. The hex value 0x25 is used if SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO) is specified on the compilation command. When the UTF support is used, the newline character in UTF-8 will be hex 0x0a regardless of what SYSIFCOPT value is used.

Conversion Errors

Some run-time routines perform a CCSID conversion from UTF-8 to an EBCDIC CCSID when required to interface with an operating system function that does not support UTF-8. When a conversion error occurs in these cases, a C2M1217 message is generated to the job log with the conversion information.
Part 2. Appendixes
Appendix A. Library Functions and Extensions

This chapter summarizes all the standard C library functions and the ILE C library extensions.

Standard C Library Functions Table, By Name

This table briefly describes the C library functions, listed in alphabetical order. This table provides the include file name and the function prototype for each function.

Table 32. Standard C Library Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>stdlib.h</td>
<td>void abort(void);</td>
<td>Stops a program abnormally.</td>
</tr>
<tr>
<td>abs</td>
<td>stdlib.h</td>
<td>int abs(int n);</td>
<td>Calculates the absolute value of an integer argument n.</td>
</tr>
<tr>
<td>acos</td>
<td>math.h</td>
<td>double acos(double x);</td>
<td>Calculates the arc cosine of x.</td>
</tr>
<tr>
<td>asctime</td>
<td>time.h</td>
<td>char *asctime(const struct tm *time);</td>
<td>Converts the time that is stored as a structure to a character string.</td>
</tr>
<tr>
<td>asctime_r</td>
<td>time.h</td>
<td>char *asctime_r (const struct tm *tm, char *buf);</td>
<td>Converts tm that is stored as a structure to a character string. (Restartable version of asctime.)</td>
</tr>
<tr>
<td>asin</td>
<td>math.h</td>
<td>double asin(double x);</td>
<td>Calculates the arc sine of x.</td>
</tr>
<tr>
<td>assert</td>
<td>assert.h</td>
<td>void assert(int expression);</td>
<td>Prints a diagnostic message and ends the program if the expression is false.</td>
</tr>
<tr>
<td>atan</td>
<td>math.h</td>
<td>double atan(double x);</td>
<td>Calculates the arc tangent of x.</td>
</tr>
<tr>
<td>atan2</td>
<td>math.h</td>
<td>double atan2(double y, double x);</td>
<td>Calculates the arc tangent of y/x.</td>
</tr>
<tr>
<td>atexit</td>
<td>stdlib.h</td>
<td>int atexit(void (*func)(void));</td>
<td>Registers a function to be called at normal termination.</td>
</tr>
<tr>
<td>atof</td>
<td>stdlib.h</td>
<td>double atof(const char *string);</td>
<td>Converts string to a double-precision floating-point value.</td>
</tr>
<tr>
<td>atoi</td>
<td>stdlib.h</td>
<td>int atoi(const char *string);</td>
<td>Converts string to an integer.</td>
</tr>
<tr>
<td>atol</td>
<td>stdlib.h</td>
<td>long int atol(const char *string);</td>
<td>Converts string to a long integer.</td>
</tr>
<tr>
<td>bsearch</td>
<td>stdlib.h</td>
<td>void *bsearch(const void *key, const void *base, size_t num, size_t size, int (*compare) (const void *element1, const void *element2));</td>
<td>Performs a binary search on an array of num elements, each of size bytes. The array must be sorted in ascending order by the function pointed to by compare.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>btowc</td>
<td>stdio.h wchar.h</td>
<td>wint_t btowc(int c)</td>
<td>Determines whether c constitutes a valid multibyte character in the initial shift state.</td>
</tr>
<tr>
<td>calloc</td>
<td>stdlib.h</td>
<td>void *calloc(size_t num, size_t size);</td>
<td>Reserves storage space for an array of num elements, each of size size, and initializes the values of all elements to 0.</td>
</tr>
<tr>
<td>catclose</td>
<td>nl_types.h</td>
<td>int catclose(nl_catd catd);</td>
<td>Closes a previously opened message catalog.</td>
</tr>
<tr>
<td>catgets</td>
<td>nl_types.h</td>
<td>char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);</td>
<td>Retrieves a message from an open message catalog.</td>
</tr>
<tr>
<td>catopen</td>
<td>nl_types.h</td>
<td>nl_catd catopen(const char *name, int oflag);</td>
<td>Opens a message catalog, which must be done before a message can be retrieved.</td>
</tr>
<tr>
<td>ceil</td>
<td>math.h</td>
<td>double ceil(double x);</td>
<td>Calculates the double value representing the smallest integer that is greater than or equal to x.</td>
</tr>
<tr>
<td>clearerr</td>
<td>stdio.h</td>
<td>void clearerr(FILE *stream);</td>
<td>Resets the error indicators and the end-of-file indicator for stream.</td>
</tr>
<tr>
<td>clock</td>
<td>time.h</td>
<td>clock_t clock(void);</td>
<td>Returns the processor time that has elapsed since the job was started.</td>
</tr>
<tr>
<td>cos</td>
<td>math.h</td>
<td>double cos(double x);</td>
<td>Calculates the cosine of x.</td>
</tr>
<tr>
<td>cosh</td>
<td>math.h</td>
<td>double cosh(double x);</td>
<td>Calculates the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>ctime</td>
<td>time.h</td>
<td>char *ctime(const time_t *time);</td>
<td>Converts time to a character string.</td>
</tr>
<tr>
<td>ctime_r</td>
<td>time.h</td>
<td>char *ctime_r(const time_t *timer, char *buf);</td>
<td>Converts timer to a character string. (Restartable version of ctime.)</td>
</tr>
<tr>
<td>difftime</td>
<td>time.h</td>
<td>double difftime(time_t time2, time_t time1);</td>
<td>Computes the difference between time2 and time1.</td>
</tr>
<tr>
<td>div</td>
<td>stdlib.h</td>
<td>div_t div(int numerator, int denominator);</td>
<td>Calculates the quotient and remainder of the division of numerator by denominator.</td>
</tr>
<tr>
<td>erf</td>
<td>math.h</td>
<td>double erf(double x);</td>
<td>Calculates the error function of x.</td>
</tr>
<tr>
<td>erfc</td>
<td>math.h</td>
<td>double erfc(double x);</td>
<td>Calculates the error function for large values of x.</td>
</tr>
<tr>
<td>exit</td>
<td>stdlib.h</td>
<td>void exit(int status);</td>
<td>Ends a program normally.</td>
</tr>
<tr>
<td>exp</td>
<td>math.h</td>
<td>double exp(double x);</td>
<td>Calculates the exponential function of a floating-point argument x.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>fabs</td>
<td>math.h</td>
<td>double fabs(double x);</td>
<td>Calculates the absolute value of a floating-point argument x.</td>
</tr>
<tr>
<td>fclose</td>
<td>stdio.h</td>
<td>int fclose(FILE *stream);</td>
<td>Closes the specified stream.</td>
</tr>
<tr>
<td>fdopen²</td>
<td>stdio.h</td>
<td>FILE *fdopen(int handle, const char *type);</td>
<td>Associates an input or output stream with the file identified by handle.</td>
</tr>
<tr>
<td>feof</td>
<td>stdio.h</td>
<td>int feof(FILE *stream);</td>
<td>Tests whether the end-of-file flag is set for a given stream.</td>
</tr>
<tr>
<td>ferror</td>
<td>stdio.h</td>
<td>int ferror(FILE *stream);</td>
<td>Tests for an error indicator in reading from or writing to stream.</td>
</tr>
<tr>
<td>fflush¹</td>
<td>stdio.h</td>
<td>int fflush(FILE *stream);</td>
<td>Writes the contents of the buffer associated with the output stream.</td>
</tr>
<tr>
<td>fgetc¹</td>
<td>stdio.h</td>
<td>int fgetc(FILE *stream);</td>
<td>Reads a single unsigned character from the input stream.</td>
</tr>
<tr>
<td>fgetpos²</td>
<td>stdio.h</td>
<td>int fgetpos(FILE *stream, fpos_t *pos);</td>
<td>Stores the current position of the file pointer associated with stream into the object pointed to by pos.</td>
</tr>
<tr>
<td>fgets¹</td>
<td>stdio.h</td>
<td>char *fgets(char *string, int n, FILE *stream);</td>
<td>Reads a string from the input stream.</td>
</tr>
<tr>
<td>fgetwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t fgetwc(FILE *stream);</td>
<td>Reads the next multibyte character from the input stream pointed to by stream.</td>
</tr>
<tr>
<td>fgetws⁶</td>
<td>stdio.h wchar.h</td>
<td>wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);</td>
<td>Reads wide characters from the stream into the array pointed to by wcs.</td>
</tr>
<tr>
<td>fileno⁵</td>
<td>stdio.h</td>
<td>int fileno(FILE *stream);</td>
<td>Determines the file handle currently associated with stream.</td>
</tr>
<tr>
<td>floor</td>
<td>math.h</td>
<td>double floor(double x);</td>
<td>Calculates the floating-point value representing the largest integer less than or equal to x.</td>
</tr>
<tr>
<td>fmod</td>
<td>math.h</td>
<td>double fmod(double x, double y);</td>
<td>Calculates the floating-point remainder of x/y.</td>
</tr>
<tr>
<td>fopen</td>
<td>stdio.h</td>
<td>FILE *fopen(const char *filename, const char *mode);</td>
<td>Opens the specified file.</td>
</tr>
<tr>
<td>fprintf</td>
<td>stdio.h</td>
<td>int fprintf(FILE *stream, const char *format-string, arg-list);</td>
<td>Formats and prints characters and values to the output stream.</td>
</tr>
<tr>
<td>fputc¹</td>
<td>stdio.h</td>
<td>int fputc(int c, FILE *stream);</td>
<td>Prints a character to the output stream.</td>
</tr>
<tr>
<td>fputs¹</td>
<td>stdio.h</td>
<td>int fputs(const char *string, FILE *stream);</td>
<td>Copies a string to the output stream.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>fputwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t fputwc(wchar_t *wc, FILE *stream);</td>
<td>Converts the wide character *wc to a multibyte character and writes it to the output stream pointed to by *stream at the current position.</td>
</tr>
<tr>
<td>fputws⁶</td>
<td>stdio.h wchar.h</td>
<td>int fputws(const wchar_t *wcs, FILE *stream);</td>
<td>Converts the wide-character string *wcs to a multibyte-character string and writes it to *stream as a multibyte character string.</td>
</tr>
<tr>
<td>fread</td>
<td>stdio.h</td>
<td>size_t fread(void *buffer, size_t size, size_t count, FILE *stream);</td>
<td>Reads up to count items of size length from the input *stream, and stores them in *buffer.</td>
</tr>
<tr>
<td>free</td>
<td>stdlib.h</td>
<td>void free(void *ptr);</td>
<td>Frees a block of storage.</td>
</tr>
<tr>
<td>freopen</td>
<td>stdio.h</td>
<td>FILE *freopen(const char *filename, const char *mode, FILE *stream);</td>
<td>Closes *stream, and reassigns it to the file specified.</td>
</tr>
<tr>
<td>frexp</td>
<td>math.h</td>
<td>double frexp(double x, int *expptr);</td>
<td>Separates a floating-point number into its mantissa and exponent.</td>
</tr>
<tr>
<td>fscanf</td>
<td>stdio.h</td>
<td>int fscanf(FILE *stream, const char *format-string, arg-list);</td>
<td>Reads data from *stream into locations given by arg-list.</td>
</tr>
<tr>
<td>fseek¹</td>
<td>stdio.h</td>
<td>int fseek(FILE *stream, long int offset, int origin);</td>
<td>Changes the current file position associated with *stream to a new location.</td>
</tr>
<tr>
<td>fsetpos¹</td>
<td>stdio.h</td>
<td>int fsetpos(FILE *stream, const fpos_t *pos);</td>
<td>Moves the current file position to a new location determined by *pos.</td>
</tr>
<tr>
<td>ftell¹</td>
<td>stdio.h</td>
<td>long int ftell(FILE *stream);</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>fwide⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwide(FILE *stream, int mode);</td>
<td>Determines the orientation of the stream pointed to by *stream.</td>
</tr>
<tr>
<td>fwprintf⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwprintf(FILE *stream, const wchar_t *format, arg-list);</td>
<td>Writes output to the stream pointed to by *stream.</td>
</tr>
<tr>
<td>fwrite</td>
<td>stdio.h</td>
<td>size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);</td>
<td>Writes up to count items of size length from *buffer to *stream.</td>
</tr>
<tr>
<td>fwscanf⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwscanf(FILE *stream, const wchar_t *format, arg-list)</td>
<td>Reads input from the stream pointed to by *stream.</td>
</tr>
<tr>
<td>gamma</td>
<td>math.h</td>
<td>double gamma(double x);</td>
<td>Computes the Gamma Function</td>
</tr>
<tr>
<td>getc¹</td>
<td>stdio.h</td>
<td>int getc(FILE *stream);</td>
<td>Reads a single character from the input *stream.</td>
</tr>
<tr>
<td>getchar¹</td>
<td>stdio.h</td>
<td>int getchar(void);</td>
<td>Reads a single character from stdin.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>getenv</td>
<td>stdlib.h</td>
<td>char *getenv(const char *varname);</td>
<td>Searches environment variables for varname.</td>
</tr>
<tr>
<td>gets</td>
<td>stdio.h</td>
<td>char *gets(char *buffer);</td>
<td>Reads a string from stdin, and stores it in buffer.</td>
</tr>
<tr>
<td>getwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t getwc(FILE *stream);</td>
<td>Reads the next multibyte character from stream, converts it to a wide character and advances the associated file position indicator for stream.</td>
</tr>
<tr>
<td>getwchar⁶</td>
<td>wchar.h</td>
<td>wint_t getwchar(void);</td>
<td>Reads the next multibyte character from stdin, converts it to a wide character, and advances the associated file position indicator for stdin.</td>
</tr>
<tr>
<td>gmtime</td>
<td>time.h</td>
<td>struct tm *gmtime(const time_t *time);</td>
<td>Converts a time value to a structure of type tm.</td>
</tr>
<tr>
<td>gmtime_r</td>
<td>time.h</td>
<td>struct tm *gmtime_r (const time_t *timer, struct tm *result);</td>
<td>Converts a timer value to a structure of type tm. (Restartable version of gmtime.)</td>
</tr>
<tr>
<td>hypot</td>
<td>math.h</td>
<td>double hypot(double side1, double side2);</td>
<td>Calculates the hypotenuse of a right-angled triangle with sides of length side1 and side2.</td>
</tr>
<tr>
<td>isalnum</td>
<td>ctype.h</td>
<td>int isalnum(int c);</td>
<td>Tests if c is alphanumeric.</td>
</tr>
<tr>
<td>isalpha</td>
<td>ctype.h</td>
<td>int isalpha(int c);</td>
<td>Tests if c is alphabetic.</td>
</tr>
<tr>
<td>isascii</td>
<td>ctype.h</td>
<td>int isascii(int c);</td>
<td>Tests if c is within the 7-bit US-ASCII range.</td>
</tr>
<tr>
<td>iscntrl</td>
<td>ctype.h</td>
<td>int iscntrl(int c);</td>
<td>Tests if c is a control character.</td>
</tr>
<tr>
<td>isdigit</td>
<td>ctype.h</td>
<td>int isdigit(int c);</td>
<td>Tests if c is a decimal digit.</td>
</tr>
<tr>
<td>isgraph</td>
<td>ctype.h</td>
<td>int isgraph(int c);</td>
<td>Tests if c is a printable character excluding the space.</td>
</tr>
<tr>
<td>islower</td>
<td>ctype.h</td>
<td>int islower(int c);</td>
<td>Tests if c is a lowercase letter.</td>
</tr>
<tr>
<td>isprint</td>
<td>ctype.h</td>
<td>int isprint(int c);</td>
<td>Tests if c is a printable character including the space.</td>
</tr>
<tr>
<td>ispunct</td>
<td>ctype.h</td>
<td>int ispunct(int c);</td>
<td>Tests if c is a punctuation character.</td>
</tr>
<tr>
<td>isspace</td>
<td>ctype.h</td>
<td>int isspace(int c);</td>
<td>Tests if c is a whitespace character.</td>
</tr>
<tr>
<td>isupper</td>
<td>ctype.h</td>
<td>int isupper(int c);</td>
<td>Tests if c is an uppercase letter.</td>
</tr>
<tr>
<td>iswalnum⁴</td>
<td>wctype.h</td>
<td>int iswalnum(wint_t wc);</td>
<td>Checks for any alphanumeric wide character.</td>
</tr>
<tr>
<td>iswalpha⁴</td>
<td>wctype.h</td>
<td>int iswalpha(wint_t wc);</td>
<td>Checks for any alphabetic wide character.</td>
</tr>
<tr>
<td>iswcntrl⁴</td>
<td>wctype.h</td>
<td>int iswcntrl(wint_t wc);</td>
<td>Tests for any control wide character.</td>
</tr>
</tbody>
</table>
**Table 32. Standard C Library Functions (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswcttype</td>
<td>wctype.h</td>
<td>int iswcttype(wint_t wc, wctype_t wc_prop);</td>
<td>Determines whether or not the wide character wc has the property wc_prop.</td>
</tr>
<tr>
<td>iswdigit</td>
<td>wctype.h</td>
<td>int iswdigit (wint_t wc);</td>
<td>Checks for any decimal-digit wide character.</td>
</tr>
<tr>
<td>iswgraph</td>
<td>wctype.h</td>
<td>int iswgraph (wint_t wc);</td>
<td>Checks for any printing wide character except for the wide-character space.</td>
</tr>
<tr>
<td>iswlower</td>
<td>wctype.h</td>
<td>int iswlower (wint_t wc);</td>
<td>Checks for any lowercase wide character.</td>
</tr>
<tr>
<td>iswprint</td>
<td>wctype.h</td>
<td>int iswprint (wint_t wc);</td>
<td>Checks for any printing wide character.</td>
</tr>
<tr>
<td>iswpunct</td>
<td>wctype.h</td>
<td>int iswpunct (wint_t wc);</td>
<td>Test for a wide non-alphanumeric, non-space character.</td>
</tr>
<tr>
<td>iswspace</td>
<td>wctype.h</td>
<td>int iswspace (wint_t wc);</td>
<td>Checks for any wide character that corresponds to an implementation-defined set of wide characters for which iswalnum is false.</td>
</tr>
<tr>
<td>iswupper</td>
<td>wctype.h</td>
<td>int iswupper (wint_t wc);</td>
<td>Checks for any uppercase wide character.</td>
</tr>
<tr>
<td>iswxdigit</td>
<td>wctype.h</td>
<td>int iswxdigit (wint_t wc);</td>
<td>Checks for any hexadecimal digit character.</td>
</tr>
<tr>
<td>isxdigit</td>
<td>wctype.h</td>
<td>int isxdigit(int c);</td>
<td>Tests if c is a hexadecimal digit.</td>
</tr>
<tr>
<td>j0</td>
<td>math.h</td>
<td>double j0(double x);</td>
<td>Calculates the Bessel function value of the first kind of order 0.</td>
</tr>
<tr>
<td>j1</td>
<td>math.h</td>
<td>double j1(double x);</td>
<td>Calculates the Bessel function value of the first kind of order 1.</td>
</tr>
<tr>
<td>jn</td>
<td>math.h</td>
<td>double jn(int n, double x);</td>
<td>Calculates the Bessel function value of the first kind of order n.</td>
</tr>
<tr>
<td>labs</td>
<td>stdlib.h</td>
<td>long int labs(long int n);</td>
<td>Calculates the absolute value of n.</td>
</tr>
<tr>
<td>ldexp</td>
<td>math.h</td>
<td>double ldexp(double x, int exp);</td>
<td>Returns the value of x multiplied by (2 to the power of exp).</td>
</tr>
<tr>
<td>ldiv</td>
<td>stdlib.h</td>
<td>ldiv_t ldiv(long int numerator, long int denominator);</td>
<td>Calculates the quotient and remainder of numerator/denominator.</td>
</tr>
<tr>
<td>localeconv</td>
<td>locale.h</td>
<td>struct lconv *localeconv(void);</td>
<td>Formats numeric quantities in struct lconv according to the current locale.</td>
</tr>
<tr>
<td>localtime</td>
<td>time.h</td>
<td>struct tm *localtime(const time_t *timeval);</td>
<td>Converts timeval to a structure of type tm.</td>
</tr>
</tbody>
</table>
Table 32. Standard C Library Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>localtime_r</td>
<td>time.h</td>
<td>struct tm *localtime_r (const time_t *timeval, struct tm *result);</td>
<td>Converts a timer value to a structure of type tm. (Restartable version of localtime.)</td>
</tr>
<tr>
<td>log</td>
<td>math.h</td>
<td>double log(double x);</td>
<td>Calculates the natural logarithm of x.</td>
</tr>
<tr>
<td>log10</td>
<td>math.h</td>
<td>double log10(double x);</td>
<td>Calculates the base 10 logarithm of x.</td>
</tr>
<tr>
<td>longjmp</td>
<td>setjmp.h</td>
<td>void longjmp(jmp_buf env, int value);</td>
<td>Restores a stack environment previously set in env by the setjmp function.</td>
</tr>
<tr>
<td>malloc</td>
<td>stdlib.h</td>
<td>void *malloc(size_t size);</td>
<td>Reserves a block of storage.</td>
</tr>
<tr>
<td>mblen</td>
<td>stdlib.h</td>
<td>int mblen(const char *string, size_t n);</td>
<td>Determines the length of a multibyte character string.</td>
</tr>
<tr>
<td>mbrlen</td>
<td>wchar.h</td>
<td>int mbrlen (const char *s, size_t n, mbstate_t *ps);</td>
<td>Determines the length of a multibyte character. (Restartable version of mblen.)</td>
</tr>
<tr>
<td>mbtowc</td>
<td>wchar.h</td>
<td>int mbtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);</td>
<td>Convert a multibyte character to a wide character. (Restartable version of mbtowc.)</td>
</tr>
<tr>
<td>mbsinit</td>
<td>wchar.h</td>
<td>int mbsinit (const mbstate_t *ps);</td>
<td>Test state object *ps for initial state.</td>
</tr>
<tr>
<td>mbsrtowcs</td>
<td>wchar.h</td>
<td>size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len, mbstate_t *ps);</td>
<td>Convert multibyte string to a wide character string. (Restartable version of mbsrtowcs.)</td>
</tr>
<tr>
<td>mbstowcs</td>
<td>stdlib.h</td>
<td>size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);</td>
<td>Converts the multibyte characters in string to their corresponding wchar_t codes, and stores not more than n codes in pwc.</td>
</tr>
<tr>
<td>mbtowc</td>
<td>stdlib.h</td>
<td>int mbtowc(wchar_t *pwc, const char *string, size_t n);</td>
<td>Stores the wchar_t code corresponding to the first n bytes of multibyte character string into the wchar_t character pwc.</td>
</tr>
<tr>
<td>memchr</td>
<td>string.h</td>
<td>void *memchr(const void *buf, int c, size_t count);</td>
<td>Searches the first count bytes of buf for the first occurrence of c converted to an unsigned character.</td>
</tr>
<tr>
<td>memcmp</td>
<td>string.h</td>
<td>int memcmp(const void *buf1, const void *buf2, size_t count);</td>
<td>Compares up to count bytes of buf1 and buf2.</td>
</tr>
<tr>
<td>memcpy</td>
<td>string.h</td>
<td>void *memcpy(const void *dest, const void *src, size_t count);</td>
<td>Copies count bytes of src to dest.</td>
</tr>
<tr>
<td>memmove</td>
<td>string.h</td>
<td>void *memmove(const void *dest, const void *src, size_t count);</td>
<td>Copies count bytes of src to dest. Allows copying between objects that overlap.</td>
</tr>
</tbody>
</table>
### Table 32. Standard C Library Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memset</td>
<td>string.h</td>
<td><code>void *memset(void *dest, int c, size_t count);</code></td>
<td>Sets count bytes of dest to a value c.</td>
</tr>
<tr>
<td>mktime</td>
<td>time.h</td>
<td><code>time_t mktime(struct tm *time);</code></td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>modf</td>
<td>math.h</td>
<td><code>double modf(double x, double *intptr);</code></td>
<td>Breaks down the floating-point value x into fractional and integral parts.</td>
</tr>
<tr>
<td>nl_langinfo</td>
<td>langinfo.h</td>
<td><code>char *nl_langinfo(nl_item item);</code></td>
<td>Retrieve from the current locale the string that describes the requested information specified by item.</td>
</tr>
<tr>
<td>perror</td>
<td>stdio.h</td>
<td><code>void perror(const char *string);</code></td>
<td>Prints an error message to stderr.</td>
</tr>
<tr>
<td>pow</td>
<td>math.h</td>
<td><code>double pow(double x, double y);</code></td>
<td>Calculates the value x to the power y.</td>
</tr>
<tr>
<td>printf</td>
<td>stdio.h</td>
<td><code>int printf(const char *format-string, arg-list);</code></td>
<td>Formats and prints characters and values to stdout.</td>
</tr>
<tr>
<td>putc</td>
<td>stdio.h</td>
<td><code>int putc(int c, FILE *stream);</code></td>
<td>Prints c to the output stream.</td>
</tr>
<tr>
<td>putchar</td>
<td>stdio.h</td>
<td><code>int putchar(int c);</code></td>
<td>Prints c to stdout.</td>
</tr>
<tr>
<td>putenv</td>
<td>stdlib.h</td>
<td><code>int *putenv(const char *varname);</code></td>
<td>Sets the value of an environment variable by altering an existing variable or creating a new one.</td>
</tr>
<tr>
<td>puts</td>
<td>stdio.h</td>
<td><code>int puts(const char *string);</code></td>
<td>Prints a string to stdout.</td>
</tr>
<tr>
<td>putwc</td>
<td>wchar.h</td>
<td><code>wint_t putwchar(wchar_t wc, FILE *stream);</code></td>
<td>Converts the wide character wc to a multibyte character, and writes it to the stream at the current position.</td>
</tr>
<tr>
<td>qsort</td>
<td>stdlib.h</td>
<td><code>void *qsort(void *base, size_t num, size_t width, int(*compare)(const void *element1, const void *element2));</code></td>
<td>Performs a quick sort of an array of num elements, each of width bytes in size.</td>
</tr>
<tr>
<td>raise</td>
<td>signal.h</td>
<td><code>int raise(int sig);</code></td>
<td>Sends the signal sig to the running program.</td>
</tr>
<tr>
<td>rand</td>
<td>stdlib.h</td>
<td><code>int rand(void);</code></td>
<td>Returns a pseudo-random integer.</td>
</tr>
<tr>
<td>rand_r</td>
<td>stdlib.h</td>
<td><code>int rand_r(void);</code></td>
<td>Returns a pseudo-random integer. (Restartable version)</td>
</tr>
<tr>
<td>realloc</td>
<td>stdlib.h</td>
<td><code>void *realloc(void *ptr, size_t size);</code></td>
<td>Changes the size of a previously reserved storage block.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>regcomp</td>
<td>regex.h</td>
<td>int regcomp(regex_t *preg, const char *pattern, int cflags);</td>
<td>Compiles the source regular expression pointed to by pattern into an executable version and stores it in the location pointed to by preg.</td>
</tr>
<tr>
<td>regerror</td>
<td>regex.h</td>
<td>size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size);</td>
<td>Finds the description for the error code errcode for the regular expression preg.</td>
</tr>
<tr>
<td>regexec</td>
<td>regex.h</td>
<td>int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t *pmatch, int eflags);</td>
<td>Compares the null-ended string string against the compiled regular expression preg to find a match between the two.</td>
</tr>
<tr>
<td>regfree</td>
<td>regex.h</td>
<td>void regfree(regex_t *preg);</td>
<td>Frees any memory that was allocated by regcomp to implement the regular expression preg.</td>
</tr>
<tr>
<td>remove</td>
<td>stdio.h</td>
<td>int remove(const char *filename);</td>
<td>Deletes the file specified by filename.</td>
</tr>
<tr>
<td>rename</td>
<td>stdio.h</td>
<td>int rename(const char *oldname, const char *newname);</td>
<td>Renames the specified file.</td>
</tr>
<tr>
<td>rewind1</td>
<td>stdio.h</td>
<td>void rewind(FILE *stream);</td>
<td>Repositions the file pointer associated with stream to the beginning of the file.</td>
</tr>
<tr>
<td>scanf</td>
<td>stdio.h</td>
<td>int scanf(const char *format-string, arg-list);</td>
<td>Reads data from stdin into locations given by arg-list.</td>
</tr>
<tr>
<td>setbuf</td>
<td>stdio.h</td>
<td>void setbuf(FILE *stream, char *buffer);</td>
<td>Controls buffering for stream.</td>
</tr>
<tr>
<td>setjmp</td>
<td>setjmp.h</td>
<td>int setjmp(jmp_buf env);</td>
<td>Saves a stack environment that can be subsequently restored by longjmp.</td>
</tr>
<tr>
<td>setlocale</td>
<td>locale.h</td>
<td>char *setlocale(int category, const char *locale);</td>
<td>Changes or queries variables defined in the locale.</td>
</tr>
<tr>
<td>setvbuf</td>
<td>stdio.h</td>
<td>int setvbuf(FILE *stream, char *buf, int type, size_t size);</td>
<td>Controls buffering and buffer size for stream.</td>
</tr>
<tr>
<td>signal</td>
<td>signal.h</td>
<td>void(*signal (int sig, void(*func)(int)) (int);</td>
<td>Registers func as a signal handler for the signal sig.</td>
</tr>
<tr>
<td>sin</td>
<td>math.h</td>
<td>double sin(double x);</td>
<td>Calculates the sine of x.</td>
</tr>
<tr>
<td>sinh</td>
<td>math.h</td>
<td>double sinh(double x);</td>
<td>Calculates the hyperbolic sine of x.</td>
</tr>
<tr>
<td>snprintf</td>
<td>stdio.h</td>
<td>int snprintf(char <em>outbuf, size_t n, const char</em>, ...);</td>
<td>Same as sprintf except that the function will stop after n characters have been written to outbuf.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>sprintf</td>
<td>stdio.h</td>
<td>int sprintf(char *buffer, const char *format-string, arg-list);</td>
<td>Formats and stores characters and values in buffer.</td>
</tr>
<tr>
<td>sqrt</td>
<td>math.h</td>
<td>double sqrt(double x);</td>
<td>Calculates the square root of x.</td>
</tr>
<tr>
<td>srand</td>
<td>stdlib.h</td>
<td>void srand(unsigned int seed);</td>
<td>Sets the seed for the pseudo-random number generator.</td>
</tr>
<tr>
<td>scanf</td>
<td>stdio.h</td>
<td>int sscanf(const char *buffer, const char *format, arg-list);</td>
<td>Reads data from buffer into the locations given by arg-list.</td>
</tr>
<tr>
<td>strcasecmp</td>
<td>strings.h</td>
<td>int strcasecmp(const char *string1, const char *string2);</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strcat</td>
<td>string.h</td>
<td>char *strcat(char *string1, const char *string2);</td>
<td>Concatenates string2 to string1.</td>
</tr>
<tr>
<td>strchr</td>
<td>string.h</td>
<td>char *strchr(const char *string, int c);</td>
<td>Locates the first occurrence of c in string.</td>
</tr>
<tr>
<td>strcmp</td>
<td>string.h</td>
<td>int strcmp(const char *string1, const char *string2);</td>
<td>Compares the value of string1 to string2.</td>
</tr>
<tr>
<td>strcoll</td>
<td>string.h</td>
<td>int strcoll(const char *string1, const char *string2);</td>
<td>Compares two strings using the collating sequence in the current locale.</td>
</tr>
<tr>
<td>strcpy</td>
<td>string.h</td>
<td>char *strcpy(char *string1, const char *string2);</td>
<td>Copies string2 into string1.</td>
</tr>
<tr>
<td>strcspn</td>
<td>string.h</td>
<td>size_t strcspn(const char *string1, const char *string2);</td>
<td>Returns the length of the initial substring of string1 consisting of characters not contained in string2.</td>
</tr>
<tr>
<td>strerror</td>
<td>string.h</td>
<td>char *strerror(int errnum);</td>
<td>Maps the error number in errnum to an error message string.</td>
</tr>
<tr>
<td>strfmon⁴</td>
<td>wchar.h</td>
<td>int strfmon (char *s, size_t maxsize, const char *format, ...);</td>
<td>Converts monetary value to string.</td>
</tr>
<tr>
<td>strftime</td>
<td>time.h</td>
<td>size_t strftime (char *dest, size_t maxsize, const char *format, const struct tm *timeptr);</td>
<td>Stores characters in an array pointed to by dest, according to the string determined by format.</td>
</tr>
<tr>
<td>strlen</td>
<td>string.h</td>
<td>size_t strlen(const char *string);</td>
<td>Calculates the length of string.</td>
</tr>
<tr>
<td>strncasecmp</td>
<td>strings.h</td>
<td>int strncasecmp(const char *string1, const char *string2, size_t count);</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>strncat</td>
<td>string.h</td>
<td>char *strncat(char *string1, const char *string2, size_t count);</td>
<td>Concatenates up to count characters of string2 to string1.</td>
</tr>
<tr>
<td>strncmp</td>
<td>string.h</td>
<td>int strncmp(const char *string1, const char *string2, size_t count);</td>
<td>Compares up to count characters of string1 and string2.</td>
</tr>
<tr>
<td>strncpy</td>
<td>string.h</td>
<td>char *strncpy(char *string1, const char *string2, size_t count);</td>
<td>Copies up to count characters of string2 to string1.</td>
</tr>
<tr>
<td>strpbrk</td>
<td>string.h</td>
<td>char *strpbrk(const char *string1, const char *string2);</td>
<td>Locates the first occurrence in string1 of any character in string2.</td>
</tr>
<tr>
<td>strptime</td>
<td>time.h</td>
<td>char *strptime (const char *buf, const char *format, struct tm *tm);</td>
<td>Date and time conversion</td>
</tr>
<tr>
<td>strchr</td>
<td>string.h</td>
<td>char *strchr(const char *string, int c);</td>
<td>Locates the last occurrence of c in string.</td>
</tr>
<tr>
<td>strspn</td>
<td>string.h</td>
<td>size_t strspn(const char *string1, const char *string2);</td>
<td>Returns the length of the initial substring of string1 consisting of characters contained in string2.</td>
</tr>
<tr>
<td>strstr</td>
<td>string.h</td>
<td>char *strstr(const char *string1, const char *string2);</td>
<td>Returns a pointer to the first occurrence of string2 in string1.</td>
</tr>
<tr>
<td>strtod</td>
<td>stdlib.h</td>
<td>double strtod(const char *nptr, char **endptr);</td>
<td>Converts nptr to a double precision value.</td>
</tr>
<tr>
<td>strtok</td>
<td>string.h</td>
<td>char *strtok(const char *string1, const char *string2);</td>
<td>Locates the next token in string1 delimited by the next character in string2.</td>
</tr>
<tr>
<td>strtok_r</td>
<td>string.h</td>
<td>char *strtok_r(const char *string, const char *seps, char **lasts);</td>
<td>Locates the next token in string delimited by the next character in seps. (Restartable version of strtok.)</td>
</tr>
<tr>
<td>strtol</td>
<td>stdlib.h</td>
<td>long int strtol(const char *nptr, char **endptr, int base);</td>
<td>Converts nptr to a signed long integer.</td>
</tr>
<tr>
<td>strtoul</td>
<td>stdlib.h</td>
<td>unsigned long int strtoul(const char *string1, char **endptr, int base);</td>
<td>Converts string1 to an unsigned long integer.</td>
</tr>
<tr>
<td>strxfrm</td>
<td>string.h</td>
<td>size_t strxfrm(char *string1, const char *string2, size_t count);</td>
<td>Converts string2 and places the result in string1. The conversion is determined by the program’s current locale.</td>
</tr>
<tr>
<td>swprintf</td>
<td>wchar.h</td>
<td>int swprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, arg-list);</td>
<td>Formats and stores a series of wide characters and values into the wide-character buffer wcsbuffer.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>swscanf</td>
<td>wchar.h</td>
<td>int swscanf (const wchar_t *buffer, const wchar_t *format, arg-list)</td>
<td>Reads data from buffer into the locations given by arg-list.</td>
</tr>
<tr>
<td>system</td>
<td>stdlib.h</td>
<td>int system(const char *string);</td>
<td>Passes string to the system command analyzer.</td>
</tr>
<tr>
<td>tan</td>
<td>math.h</td>
<td>double tan(double x);</td>
<td>Calculates the tangent of x.</td>
</tr>
<tr>
<td>tanh</td>
<td>math.h</td>
<td>double tanh(double x);</td>
<td>Calculates the hyperbolic tangent of x.</td>
</tr>
<tr>
<td>time</td>
<td>time.h</td>
<td>time_t time(time_t *timeptr);</td>
<td>Returns the current calendar time.</td>
</tr>
<tr>
<td>tmpfile</td>
<td>stdio.h</td>
<td>FILE *tmpfile(void);</td>
<td>Creates a temporary binary file and opens it.</td>
</tr>
<tr>
<td>tmpnam</td>
<td>stdio.h</td>
<td>char *tmpnam(char *string);</td>
<td>Generates a temporary file name.</td>
</tr>
<tr>
<td>toascii</td>
<td>ctype.h</td>
<td>int toascii(int c);</td>
<td>Converts c to a character in the 7-bit US-ASCII character set.</td>
</tr>
<tr>
<td>tolower</td>
<td>ctype.h</td>
<td>int tolower(int c);</td>
<td>Converts c to lowercase.</td>
</tr>
<tr>
<td>toupper</td>
<td>ctype.h</td>
<td>int toupper(int c);</td>
<td>Converts c to uppercase.</td>
</tr>
<tr>
<td>towctrans</td>
<td>wctype.h</td>
<td>wint_t towctrans(wint_t wc, wctrans_t desc);</td>
<td>Translates the wide character wc based on the mapping described by desc.</td>
</tr>
<tr>
<td>towlower4</td>
<td>wctype.h</td>
<td>wint_t towlower(wint_t wc);</td>
<td>Converts uppercase letter to lowercase letter.</td>
</tr>
<tr>
<td>towupper4</td>
<td>wctype.h</td>
<td>wint_t towupper(wint_t wc);</td>
<td>Converts lowercase letter to uppercase letter.</td>
</tr>
<tr>
<td>ungetc3</td>
<td>stdio.h</td>
<td>int ungetc(int c, FILE *stream);</td>
<td>Pushes c back onto the input stream.</td>
</tr>
<tr>
<td>ungetwc6</td>
<td>stdio.h</td>
<td>wint_t ungetwc(wint_t wc, FILE *stream);</td>
<td>Pushes the wide character wc back onto the input stream.</td>
</tr>
<tr>
<td>va_arg</td>
<td>stdarg.h</td>
<td>var_type va_arg(va_list arg_ptr, var_type);</td>
<td>Returns the value of one argument and modifies arg_ptr to point to the next argument.</td>
</tr>
<tr>
<td>va_end</td>
<td>stdarg.h</td>
<td>void va_end(va_list arg_ptr);</td>
<td>Facilitates normal return from variable argument list processing.</td>
</tr>
<tr>
<td>va_start</td>
<td>stdarg.h</td>
<td>void va_start(va_list arg_ptr, variable_name);</td>
<td>Initializes arg_ptr for subsequent use by va_arg and va_end.</td>
</tr>
<tr>
<td>vfprintf</td>
<td>stdio.h</td>
<td>int vfprintf(FILE *stream, const char *format, va_list arg_ptr);</td>
<td>Formats and prints characters to the output stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfscanf</td>
<td>stdio.h</td>
<td>int vfscanf(FILE *stream, const char *format, va_list arg_ptr);</td>
<td>Reads data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>vfwprintf⁶</td>
<td>stdarg.h, stdio.h, wchar.h</td>
<td>int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);</td>
<td>Equivalent to fwprintf, except that the variable argument list is replaced by arg.</td>
</tr>
<tr>
<td>vwscanf</td>
<td>stdio.h, stdarg.h, wchar.h</td>
<td>int vwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);</td>
<td>Reads wide data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vprintf</td>
<td>stdio.h, stdarg.h</td>
<td>int vprintf(const char *format, va_list_arg_ptr);</td>
<td>Formats and prints characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>vscanf</td>
<td>stdio.h, stdarg.h</td>
<td>int vscanf(const char *format, va_list_arg_ptr);</td>
<td>Reads data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vsprintf</td>
<td>stdio.h, stdarg.h</td>
<td>int vsprintf(char *target-string, const char *format, va_list_arg_ptr);</td>
<td>Formats and stores characters in a buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>stdio.h</td>
<td>int vsnprintf(char *outbuf, size_t n, const char *, va_list_arg_ptr);</td>
<td>Same as vsprintf except that the function will stop after n characters have been written to outbuf.</td>
</tr>
<tr>
<td>vsscanf</td>
<td>stdio.h, stdarg.h</td>
<td>int vsscanf(const char *buffer, const char *format, va_list_arg_ptr);</td>
<td>Reads data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vswprintf</td>
<td>stdarg.h, wchar.h</td>
<td>int vswprintf(const wchar_t *format, va_list_arg_ptr);</td>
<td>Formats and stores a series of wide characters and values in the buffer wcharbuffer.</td>
</tr>
<tr>
<td>vswscanf</td>
<td>stdio.h, wchar.h</td>
<td>int vswscanf(const wchar_t *format, va_list_arg_ptr);</td>
<td>Reads wide data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vwprintf⁶</td>
<td>stdarg.h, wchar.h</td>
<td>int vwprintf(const wchar_t *format, va_list_arg_ptr);</td>
<td>Equivalent to wprintf, except that the variable argument list is replaced by arg.</td>
</tr>
<tr>
<td>vwscanf</td>
<td>stdio.h, wchar.h</td>
<td>int vwscanf(const wchar_t *format, va_list_arg_ptr);</td>
<td>Reads wide data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>wcrtomb⁴</td>
<td>wchar.h</td>
<td>int wcrtomb (char *s, wchar_t wchar, mbstate_t *pss);</td>
<td>Converts a wide character to a multibyte character. (Restartable version of wcrtomb.)</td>
</tr>
<tr>
<td>wcscat</td>
<td>wchar.h, wcstr.h</td>
<td>wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);</td>
<td>Appends a copy of the string pointed to by string2 to the end of the string pointed to by string1.</td>
</tr>
<tr>
<td>wcschr</td>
<td>wchar.h, wcstr.h</td>
<td>wchar_t *wcschr(const wchar_t *string, wchar_t character);</td>
<td>Searches the wide-character string pointed to by string for the occurrence of character.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wcscmp</td>
<td>wcstr.h</td>
<td>int wcscmp(const wchar_t *string1, const wchar_t *string2);</td>
<td>Compares two wide-character strings, *string1 and *string2.</td>
</tr>
<tr>
<td>wcscoll*</td>
<td>wchar.h</td>
<td>int wcscoll (const wchar_t *wcs1, const wchar_t *wcs2);</td>
<td>Compares two wide-character strings using the collating sequence in the current locale.</td>
</tr>
<tr>
<td>wcscpy</td>
<td>wcstr.h</td>
<td>wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);</td>
<td>Copies the contents of *string2 (including the ending wchar_t null character) into *string1.</td>
</tr>
<tr>
<td>wcscspn</td>
<td>wcstr.h</td>
<td>size_t wcscspn(const wchar_t *string1, const wchar_t *string2);</td>
<td>Determines the number of wchar_t characters in the initial segment of the string pointed to by *string1 that do not appear in the string pointed to by *string2.</td>
</tr>
<tr>
<td>wcsftime</td>
<td>wchar.h</td>
<td>size_t wcsftime(wchar_t *wdest, size_t maxsize, const wchar_t *format, const struct tm *timeptr);</td>
<td>Converts the time and date specification in the timeptr structure into a wide-character string.</td>
</tr>
<tr>
<td>wcslen</td>
<td>wcstr.h</td>
<td>size_t wcslen(const wchar_t *string);</td>
<td>Computes the number of wide-characters in the string pointed to by string.</td>
</tr>
<tr>
<td>wcslocaleconv</td>
<td>locale.h</td>
<td>struct wcslocaleconv *wcslocaleconv(void);</td>
<td>Formats numeric quantities in struct wcslocaleconv according to the current locale.</td>
</tr>
<tr>
<td>wcscat</td>
<td>wcstr.h</td>
<td>wchar_t *wcscat(wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Appends up to count wide characters from string2 to the end of string1, and appends a wchar_t null character to the result.</td>
</tr>
<tr>
<td>wcscmp</td>
<td>wcstr.h</td>
<td>int wcscmp(const wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Compares up to count wide characters in string1 to string2.</td>
</tr>
<tr>
<td>wcscpy</td>
<td>wcstr.h</td>
<td>wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Copies up to count wide characters from string2 to string1.</td>
</tr>
<tr>
<td>wcspbrk</td>
<td>wcstr.h</td>
<td>wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);</td>
<td>Locates the first occurrence in the string pointed to by string1 of any wide characters from the string pointed to by string2.</td>
</tr>
<tr>
<td>wcsptime</td>
<td>wchar.h</td>
<td>wchar_t *wcsptime (const wchar_t *buf, const wchar_t *format, struct tm *tm );</td>
<td>Date and time conversion. Equivalent to strftime(), except that it uses wide characters.</td>
</tr>
<tr>
<td>wcsrchr</td>
<td>wcstr.h</td>
<td>wchar_t *wcsrchr(const wchar_t *string, wchar_t character);</td>
<td>Locates the last occurrence of character in the string pointed to by string.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wcsrtombs⁴</td>
<td>wchar.h</td>
<td>size_t wcsrtombs (char *dst, const wchar_t **src, size_t len, mbstate_t *ps);</td>
<td>Converts wide character string to multibyte string. (Restartable version of wcsombs.)</td>
</tr>
<tr>
<td>wcsspn</td>
<td>wchar.h</td>
<td>size_t wcsspn(const wchar_t *string1, const wchar_t *string2);</td>
<td>Computes the number of wide characters in the initial segment of the string pointed to by string1, which consists entirely of wide characters from the string pointed to by string2.</td>
</tr>
<tr>
<td>wcsstr</td>
<td>wchar.h</td>
<td>wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);</td>
<td>Locates the first occurrence of wcs2 in wcs1.</td>
</tr>
<tr>
<td>wcstod</td>
<td>wchar.h</td>
<td>double wcstod(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a double value.</td>
</tr>
<tr>
<td>wcstok</td>
<td>wchar.h</td>
<td>wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr)</td>
<td>Breaks wcs1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by wcs2.</td>
</tr>
<tr>
<td>wcstol</td>
<td>wchar.h</td>
<td>long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a long integer value.</td>
</tr>
<tr>
<td>wcstombs</td>
<td>stdlib.h</td>
<td>size_t wcstombs(char *dest, const wchar_t *string, size_t count);</td>
<td>Converts the wchar_t string into a multibyte string dest.</td>
</tr>
<tr>
<td>wcstoul</td>
<td>wchar.h</td>
<td>unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to an unsigned long integer value.</td>
</tr>
<tr>
<td>wcsxfrm⁴</td>
<td>wchar.h</td>
<td>size_t wcsxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);</td>
<td>Transforms a wide-character string to values which represent character collating weights and places the resulting wide-character string into an array.</td>
</tr>
<tr>
<td>wctob</td>
<td>stdarg.h wchar.h</td>
<td>int wctob(wint_t wc);</td>
<td>Determines whether wc corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.</td>
</tr>
<tr>
<td>wcctomb</td>
<td>stdlib.h</td>
<td>int wcctomb(char *string, wchar_t character);</td>
<td>Converts the wchar_t value of character into a multibyte string.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wctrans</td>
<td>wchar.h</td>
<td>wctrans_t wctrans(const char *property);</td>
<td>Constructs a value with type \texttt{wctrans_t} that describes a mapping between wide characters identified by the string argument \texttt{property}.</td>
</tr>
<tr>
<td>\textbf{wctype\textsuperscript{4}}</td>
<td>wchar.h</td>
<td>wctype_t wctype (const char *\texttt{property});</td>
<td>Obtains handle for character property classification.</td>
</tr>
<tr>
<td>wcwidth</td>
<td>wchar.h</td>
<td>int wcswidth(const wchar_t *\texttt{wcs}, size_t \texttt{n});</td>
<td>Determine the display width of a wide character string.</td>
</tr>
<tr>
<td>wmemchr</td>
<td>wchar.h</td>
<td>wchar_t wmemchr(const wchar_t *s, wchar_t \texttt{c}, size_t \texttt{n});</td>
<td>Locates the first occurrence of \texttt{c} in the initial \texttt{n} wide characters of the object pointed to by \texttt{s}.</td>
</tr>
<tr>
<td>wmemcmp</td>
<td>wchar.h</td>
<td>int wmemcmp(const wchar_t *\texttt{s1}, const wchar_t *\texttt{s2}, size_t \texttt{n});</td>
<td>Compares the first \texttt{n} wide characters of the object pointed to by \texttt{s1} to the first \texttt{n} characters of the object pointed to by \texttt{s2}.</td>
</tr>
<tr>
<td>wmemcpy</td>
<td>wchar.h</td>
<td>wchar_t wmemcpy(wchar_t *\texttt{s1}, const wchar_t *\texttt{s2}, size_t \texttt{n});</td>
<td>Copies \texttt{n} wide characters from the object pointed to by \texttt{s1} to the object pointed to by \texttt{s2}.</td>
</tr>
<tr>
<td>wmemmove</td>
<td>wchar.h</td>
<td>wchar_t wmemmove(wchar_t *\texttt{s1}, const wchar_t *\texttt{s2}, size_t \texttt{n});</td>
<td>Copies \texttt{n} wide characters from the object pointed to by \texttt{s1} to the object pointed to by \texttt{s2}.</td>
</tr>
<tr>
<td>wmemset</td>
<td>wchar.h</td>
<td>wchar_t wmemset(wchar_t *\texttt{s}, wchar_t \texttt{c}, size_t \texttt{n});</td>
<td>Copies the value of \texttt{c} into each of the first \texttt{n} wide characters of the object pointed to by \texttt{s}.</td>
</tr>
<tr>
<td>wprintf\textsuperscript{6}</td>
<td>wchar.h</td>
<td>int wprintf(const wchar_t *\texttt{format}, arg-list);</td>
<td>Equivalent to \texttt{fwprintf} with the argument \texttt{stdout} interposed before the arguments to \texttt{printf}.</td>
</tr>
<tr>
<td>wscanf\textsuperscript{6}</td>
<td>wchar.h</td>
<td>int wscanf(const wchar_t *\texttt{format}, arg-list);</td>
<td>Equivalent to \texttt{fscanf} with the argument \texttt{stdin} interposed before the arguments of \texttt{wscanf}.</td>
</tr>
<tr>
<td>y0</td>
<td>math.h</td>
<td>double y0(double \texttt{x});</td>
<td>Calculates the Bessel function value of the second kind of order 0.</td>
</tr>
<tr>
<td>y1</td>
<td>math.h</td>
<td>double y1(double \texttt{x});</td>
<td>Calculates the Bessel function value of the second kind of order 1.</td>
</tr>
<tr>
<td>yn</td>
<td>math.h</td>
<td>double yn(int \texttt{n}, double \texttt{x});</td>
<td>Calculates the Bessel function value of the second kind of order \texttt{n}.</td>
</tr>
</tbody>
</table>
## ILE C Library Extensions to C Library Functions Table

This table briefly describes all the ILE C library extensions, listed in alphabetical order. This table provides the include file name, and the function prototype for each function.

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include file</th>
<th>Function prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_Get_Ssn_Handle</td>
<td>stdio.h</td>
<td>_SSN_Handle_T _C_Get_Ssn_Handle(void);</td>
<td>Returns a handle to the C session for use with DSM APIs.</td>
</tr>
<tr>
<td>_C_TS_malloc64</td>
<td>stdlib.h</td>
<td>void *_C_TS_malloc64(unsigned long long int);</td>
<td>Same as _C_TS_malloc, but takes an unsigned long long int so the user can ask for more than 2 GB of storage on a single request.</td>
</tr>
<tr>
<td>_C_TS_malloc_info</td>
<td>mallocinfo.h</td>
<td>int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);</td>
<td>Returns current memory usage information.</td>
</tr>
<tr>
<td>_C_TS_malloc_debug</td>
<td>mallocinfo.h</td>
<td>int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level, struct _C_mallinfo_t *output_record, size_t sizeofoutput);</td>
<td>Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by _C_TS_malloc functions.</td>
</tr>
<tr>
<td>_GetExcData</td>
<td>signal.h</td>
<td>void _GetExcData(_INTRPT_Hndlr_Parms_T *parms);</td>
<td>Retrieves information about an exception from within a signal handler.</td>
</tr>
<tr>
<td>QXXCHGDA</td>
<td>xxdtaa.h</td>
<td>void QXXCHGDA(DTAA_NAME_T dname, short int offset, short int len, char *dtaptr);</td>
<td>Changes the OS/400 data area specified on dname using the data pointed to by dtaptr.</td>
</tr>
<tr>
<td>QXXDTOP</td>
<td>xxcvt.h</td>
<td>void QXXDTOP(unsigned char *pptr, int digs, int fraction, double value);</td>
<td>Converts a double value to a packed decimal value with digs total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>QXXDTOZ</td>
<td>xxcvt.h</td>
<td>void QXXDTOZ(unsigned char *zptr, int digits, int fraction, double value);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts a double value to a zoned decimal value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXITOP</td>
<td>xxcvt.h</td>
<td>void QXXITOP(unsigned char *pptr, int digits, int fraction, int value);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts an integer value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXITOZ</td>
<td>xxcvt.h</td>
<td>void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts an integer value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXPTOD</td>
<td>xxcvt.h</td>
<td>double QXXPTOD(unsigned char *pptr, int digits, int fraction );</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts a packed decimal number to a double value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXPTOI</td>
<td>xxcvt.h</td>
<td>int QXXPTOI(unsigned char *pptr, int digits, int fraction );</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts a packed decimal number to an integer value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXRTVDA</td>
<td>xxdtaa.h</td>
<td>void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset, short int len, char *dtastr);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Retrieves a copy of the OS/400 data area specified on dtaname.</td>
</tr>
<tr>
<td>QXXZTOD</td>
<td>xxcvt.h</td>
<td>double QXXZTOD(unsigned char *zptr, int digits, int fraction );</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts a zoned decimal number to a double value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXZTOI</td>
<td>xxcvt.h</td>
<td>int QXXZTOI(unsigned char *zptr, int digits, int fraction );</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Converts a zoned decimal value to an integer value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>_Racquire</td>
<td>recio.h</td>
<td>int _Racquire(_RFILE *fp, char *dev);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Prepares a device for record I/O operations.</td>
</tr>
<tr>
<td>_Rclose</td>
<td>recio.h</td>
<td>int _Rclose(_RFILE *fp);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Closes a file that is opened for record I/O operations.</td>
</tr>
<tr>
<td>_Rcommit</td>
<td>recio.h</td>
<td>int _Rcommit(char *cmtid);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Completes the current transaction, and establishes a new commitment boundary.</td>
</tr>
<tr>
<td>_Rdelete</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rdelete(_RFILE *fp);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Deletes the currently locked record.</td>
</tr>
<tr>
<td>_Rdevatr</td>
<td>xxfdbk.h</td>
<td>_XXDEV_ATR_T * _Rdevatr(_RFILE *fp, char *pgmdev);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recio.h</td>
<td></td>
<td>Returns a pointer to a copy of the device attributes feedback area for the file referenced by fp and the device pgmdev.</td>
</tr>
<tr>
<td>_Rfeod</td>
<td>recio.h</td>
<td>int _Rfeod(_RFILE *fp);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Forces an end-of-file condition for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rfeov</td>
<td>recio.h</td>
<td>int _Rfeov(_RFILE *fp);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Forces an end-of-volume condition for the tape file referenced by fp.</td>
</tr>
<tr>
<td>_Rformat</td>
<td>recio.h</td>
<td>void Rformat(_RFILE *fp, char *fmt);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sets the record format to fmt for the file referenced by fp.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>_Rindara</td>
<td>recio.h</td>
<td>void _Rindara (_RFILE *fp, char *indic_buf);</td>
<td>Sets up the separate indicator area to be used for subsequent record I/O operations.</td>
</tr>
<tr>
<td>_Riofbk</td>
<td>recio.h</td>
<td>_XXIOFB_T * _Riofbk(_RFILE *fp);</td>
<td>Returns a pointer to a copy of the I/O feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rlocate</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);</td>
<td>Positions to the record in the file associated with fp and specified by the key, klen_rrn and opt parameters.</td>
</tr>
<tr>
<td>_Ropen</td>
<td>recio.h</td>
<td>_RFILE * _Ropen(const char *filename, const char *mode ...);</td>
<td>Opens a file for record I/O operations.</td>
</tr>
<tr>
<td>_Ropnfbk</td>
<td>recio.h</td>
<td>_XXOPFB_T * _Ropnfbk(_RFILE *fp);</td>
<td>Returns a pointer to a copy of the open feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rpgmdev</td>
<td>recio.h</td>
<td>int _Rpgmdev(_RFILE *fp, char *dev);</td>
<td>Sets the default program device.</td>
</tr>
<tr>
<td>_Rreadd</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rreadd(_RFILE *fp, void *buf, size_t size, int opts, long rrn);</td>
<td>Reads a record by relative record number.</td>
</tr>
<tr>
<td>_Readf</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readf(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the first record.</td>
</tr>
<tr>
<td>_Readindv</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readindv(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads a record from an invited device.</td>
</tr>
<tr>
<td>_Readk</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readk(_RFILE *fp, void *buf, size_t size, int opts, void *key, int klen);</td>
<td>Reads a record by key.</td>
</tr>
<tr>
<td>_Readl</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readl(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the last record.</td>
</tr>
<tr>
<td>_Readn</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readn(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the next record.</td>
</tr>
<tr>
<td>_Readnc</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readnc(_RFILE *fp, void *buf, size_t size);</td>
<td>Reads the next changed record in the subfile.</td>
</tr>
<tr>
<td>_Readp</td>
<td>recio.h</td>
<td>_RIOFB_T * _Readp(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the previous record.</td>
</tr>
<tr>
<td>_Reads</td>
<td>recio.h</td>
<td>_RIOFB_T * _Reads(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the same record.</td>
</tr>
<tr>
<td>_Release</td>
<td>recio.h</td>
<td>int _Release(_RFILE *fp, char *dev);</td>
<td>Makes the specified device ineligible for record I/O operations.</td>
</tr>
<tr>
<td>_Rrlsck</td>
<td>recio.h</td>
<td>int _Rrlsck(_RFILE *fp);</td>
<td>Releases the currently locked record.</td>
</tr>
<tr>
<td>_Rrollbck</td>
<td>recio.h</td>
<td>int _Rrollbck(void);</td>
<td>Reestablishes the last commitment boundary as the current commitment boundary.</td>
</tr>
<tr>
<td>_Rupdate</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rupdate(_RFILE *fp, void *buf, size_t size);</td>
<td>Writes to the record that is currently locked for update.</td>
</tr>
<tr>
<td>_Rupfb</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rupfb(_RFILE *fp);</td>
<td>Updates the feedback structure with information about the last record I/O operation.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_Rwrite</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rwrite(_RFILE *fp, void *buf, size_t size);</td>
<td>Writes a record to the end of the file.</td>
</tr>
<tr>
<td>_Rwrited</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrn);</td>
<td>Writes a record by relative record number. It only writes over deleted records.</td>
</tr>
<tr>
<td>_Rwriterd</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rwriterd(_RFILE *fp, void *buf, size_t size);</td>
<td>Reads and writes a record.</td>
</tr>
<tr>
<td>_Rwrread</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rwrread(_RFILE *fp, void *inbuf, size_t in_buf_size, void *out_buf, size_t out_buf_size);</td>
<td>Functions as _Rwriterd, except separate buffers may be specified for input and output data.</td>
</tr>
<tr>
<td>__wcsicmp</td>
<td>wchar.h</td>
<td>int __wcsicmp(const wchar_t *string1, const wchar_t *string2);</td>
<td>Compares wide character strings without case sensitivity.</td>
</tr>
<tr>
<td>__wcsnicmp</td>
<td>wchar.h</td>
<td>int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Compares wide character strings without case sensitivity.</td>
</tr>
</tbody>
</table>
Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created...
programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication is intended to help you to write Integrated Language Environment C/C++ for OS/400 programs. It contains information necessary to use the Integrated Language Environment C/C++ for the OS/400 compiler. The ILE C/C++ Programmer’s Guide primarily documents general-use programming interfaces and associated guidance information provided by the Integrated Language Environment C/C++ for the OS/400 compiler.
Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM
Integrated Language Environment
iSeries
Operating System/400
OS/400
RPG/400

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.
Bibliography

For additional information about topics related to ILE C/C++ programming on the iSeries system, refer to the following IBM iSeries publications and iSeries Information Center topics (http://www.ibm.com/eserver/iseries/infocenter)

- The APIs topic in the Programming category of the iSeries Information Center provides information for experienced application and system programmers who want to use the OS/400 application programming interfaces (APIs).
- *Application Display Programming*, SC41-5715-00 provides information about using DDS to create and maintain displays, creating and working with display files, creating online help information, using UIM to define displays, and using panel groups, records, and documents.
- The Backup and Recovery topic in the Programming category of the iSeries Information Center provides information about how to plan a backup and recovery strategy, how to back up your server, how to manage tape libraries, and how to set up disk protection for your data. It also includes information about the Backup, Recovery and Media Services plug-in to iSeries Navigator, information about recovering your server, and answers to some frequently asked questions about backup and recovery.
- *Backup and Recovery*, SC41-5304-07 provides general information about recovery and availability options for the iSeries server. It describes the options available on the server, compares and contrasts them, and tells where to find more information about them.
- The CL (Control Language) topic in the Programming category of the iSeries Information Center provides a description of the OS/400 control language commands.
- *CL Programming*, SC41-5721-06 provides a wide-ranging discussion of iSeries programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- Standard C/C++ Library Reference, SC09-4949-01 contains reference information for the C/C++ languages.
- *Communications Management*, SC41-5406-02 provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- The Files and File Systems topic in the iSeries Information Center provides information about using files in application programs.
- The Globalization topic in the Programming category of the iSeries Information Center provides information for planning, installing, configuring, and using globalization and multilingual support of the iSeries server. It also provides an explanation of the database management of multilingual data and application considerations for a multilingual system.
- *ICF Programming*, SC41-5442-00, provides information needed to write application programs that use iSeries communications and the OS/400 intersystem communications function (OS/400-ICF). It also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *ILE Concepts*, SC41-5606-07 explains concepts and terminology pertaining to the Integrated Language Environment® architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00, provides information on Machine Interface instructions available in the ILE C compiler that provide system-level programming capabilities.
- The Printing category of information in the iSeries Information Center provides information...
about how to plan for and configure printing functions, as well as basic printing information.

- *Printer Device Programming, SC41-5713-06* This guide provides specific information on printing elements and concepts of the iSeries server, printer file and print spooling support, and printer connectivity.

- The *Security* topic in the iSeries Information Center provides information about how to set up and plan for your iSeries security, how to secure network and communications applications, and how to add highly secure cryptographic processing capability to your iSeries server. It also includes information about object signing and signature validation, identity mapping, and solutions to Internet security risks.

- *ILE C/C++ for AS/400 MI Library Reference, SC09-2418-00* contains reference information for the C/C++ languages.

- *iSeries Security Reference, SC41-5302-07* tells how system security support can be used to protect the system and data from being used by people who do not have the proper authorization, protect data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.

- The *Systems Management* topic in the iSeries Information Center provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.

- *WebSphere Development Studio: ILE C/C++ Language Reference, SC09-7852-00* contains reference information for the C/C++ languages.

- *WebSphere Development Studio: ILE C/C++ Compiler Reference, SC09-4816-02* contains reference information about using preprocessor statements, macros defined by and pragmas recognized by the ILE C/C++ compiler, command line options for both iSeries and QShell working environments, and I/O considerations for the iSeries environment.

- *WebSphere Development Studio: ILE C/C++ Programmer’s Guide, SC09-2712-04* provides information on how to develop applications using the ILE C language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also described. An appendix includes information on migrating source code from EPM C or System C to ILE C.

For more information about OS/400 programming utilities, see the following books:

- *ADTS/400: Programming Development Manager, SC09-1771-00*
- *ADTS for AS/400: Screen Design Aid, SC09-2604-00*
- *ADTS for AS/400: Source Entry Utility, SC09-2605-00*
Index

Special characters
__EXBDY built-in 6
__VBDY built-in 6
_C_Get_Ssn_Handle() function 57
_C_TS_malloc 181, 529
_C_TS_malloc_debug 529
_C_TS_malloc_debug() function 71
_C_TS_malloc_info 529
_C_TS_malloc_info() function 73
_C_TS_malloc64 181, 529
_EXBDY macro 6
_fputchar() function 111
_gcvt() function 141
_GetExcData() function 145
_INTRPT_Hndlr_Parms_T 4
_itoa() function 163
_ltoa() function 177
_Racquire() function 240
_Rclose() function 241
_Rcommit() function 243
_Rdelete() function 244
_Rdevatr() function 246
_Rfeod() function 260
_Rfeov() function 261
_Rformat() function 262
_Rindara() function 264
_Riofbk() function 266
_Rlocate() function 268
_Ropen() function 271
_Ropenfbk() function 275
_Rpgmdev() function 276
_Rread() function 277
_Rreadf() function 278
_Rreadindv() function 280
_Rreadk() function 281
_Rreadl() function 283
_Rreadn() function 284
_Rreadnc() function 285
_Rreadp() function 286
_Rreads() function 287
_Rreadx() function 288
_Rread0() function 289
_Rread() function 290
_Rrelease() function 292
_Rrelease() function 293
_Rreset() function 294
_Rreset() function 295
_Rrlslck() function 296
_Rrslck() function 297
_Rrslck() function 298
_Rrollbck() function 299
_Rrollbck() function 300
_Rruptate() function 301
_Rruptate() function 302
_Rruptate() function 303
_Rruptate() function 304
_Rruptions() function 305
_Rruptate() function 306
_Rruptate() function 307
_Rruptate() function 308
_Rruptate() function 309
_Rruptate() function 310
_Rruptate() function 311
_Ruart() function 312
_Ruart() function 313
_VBDY macro 6
_wcsicmp() function 439
_wcsnicmp() function 446

A
abnormal program end 38
abort() function 38
abs() function 39

B
bessel functions 23, 52
binary files 102
binary search 54
blksize 102
block size 102
bsearch() function 54
btowc() function 55
buffers
assigning 319
comparing 198
copying 199, 202
flushing 87
searching 197
setting characters 203

C
calculating
absolute value 39
absolute value of long integer 164
arccosine 40
arctangent 47
base 10 logarithm 176
cosine 66
error functions 79
calculating (continued)
exponential function 81
floating-point absolute value 81
floating-point remainder 99
hyperbolic cosine 67
hyperbolic sine 333
hypotenuse 155
logarithm 175
natural logarithm 175
quotient and remainder 77
sine 332
time difference 76
calloc() function 57
cancel handler reason codes 493
case mapping functions 36
catclose() function 59
catgets() function 60
catopen() function 61
ceil() function 63
celling function 63
character
converting
to floating-point 51
to integer 50
to long integer 51
reading 89, 142
setting 203
ungetting 399
writing 109, 221
character case mapping
tolower 395
toupper 395
towlower 397
towupper 397
character testing
ASCII value 158
character property 159, 162
isalnum 156
isalpha 156
iscntrl 156
isdigit 156
isgraph 156
islower 156
isprint 156
ispunct 156
isspace 156
isupper 156
isxdigit 156
wide alphabetic character 160
wide alphanumeric character 160
wide control character 160
wide decimal-digit character 160
wide hexadecimal digit 160
wide lowercase character 160
wide non-alphanumeric character 160

© Copyright IBM Corp. 1999, 2004
character testing (continued)
wide non-space character 160
wide printing character 160
wide uppercase character 160
wide whitespace character 160
character testing functions 35
clear error indicators 64
clearerr 64
clock() function 65
CLOCKS_PER_SEC 65
closing
file 241
message catalog 59
stream 82
comparing
buffers 198
strings 343, 346, 348, 362
comparing strings 340, 359
compile regular expression 250
concatenating strings 341, 361
conversion functions
QXXDTOP 231
QXXDTOZ 232
QXXITOP() 233
QXXITOZ 233
QXXPTOD 234
QXXPTOI 235
QXXZTOD 237
QXXZTOI 238
converting
character case 395
character string to double 376
character string to long integer 380
date 369, 448
double to zoned decimal 232
floating point to packed decimal 231
floating-point numbers to integers and fractions 205
from structure to string 41
from structure to string (restartable version) 43
integer to a character in the ASCII character set 394
integer to packed decimal 233
integer to zoned decimal 233
local time 204
monetary value to string 351
multibyte character to a wide character 186
multibyte character to wchar_t 196
multibyte string to a wide character string 191
packed decimal to double 234
packed decimal to integer 235
single byte to wide character 55
string to formatted date and time 437
string segment to unsigned integer 382
strings to floating-point values 49
strings to integer values 50
strings to long values 51
time 151, 153, 173, 174, 369, 448
time to character string 68, 69
wide character case 397
wide character string to multibyte string 451
converting (continued)
wide character to a multibyte character 425
wide character to byte 468
wide character to long integer 458
wide character to multibyte character 469
wide-character string to double 455
wide-character string to unsigned long 463
zoned decimal to double 237
zoned decimal to integer 238
copying
bytes 199, 202
strings 347, 364
cos() function 66
cosh() function 67
creating
a temporary file 392, 393
time_r() function 69
time() function 68
type functions 156
type.h include file 3
currency functions 24
D
data conversion
atol() function 49
atol() function 50
atol() function 51
data items 116
data type compatibility
CL 498, 501, 502
COBOL 500
ILE COBOL 497
RPG 496, 499
data type limits 7
date and time conversion 369, 448
decimal.h include file 3
deleting
file 257
record 244, 257
determine the display width of a wide character 475
determining
display width of a wide character string 466
display width of a wide-character string 467
length of a multibyte character 183
differential equations 23
diffftime() function 76
div() function 77
E
end-of-file indicator 64, 86
ending a program 38, 80
environment
functions 33
interaction 33
retrieving information 168
table 144
variables 144, 223
environment variables
adding 223
changing 223
searching 144
eofile
clearing 259
macro 14
resetting error indicator 64
erf() function 79
erfc() function 79
errno 4
errno macros 485, 486, 487
erro values for Integrated File System 487, 488, 489
errno variable 209
errno.h include file 4
error handling
assert 45
clearerr 64
error 87
functions 21
error 209
stream I/O 87
strerror 350
error indicator 87
error macros, mapping stream I/O exceptions 489, 490
error messages
printing 209
except.h include file 4
exception class
listing 495
mapping 492
EXIT_FAILURE 15, 80
EXIT_SUCCESS 15, 80
exit() function 80
exp() function 81
exponential functions
exp 81
frexp 122
ldexp 166
log 175
log10 176
pow 210
sqrt 337
F
fabs() function 81
fclose() function 82
fdopen() function 83
feof() function 84
ferror() function 85
fflush() function 87
fgetc() function 89
fgetpos() function 91
fgets() function 92
fgets() function 94
fgetws() function 96
file
appending to 83
handle 98
include 3
maximum opened 14
name length 14
positioning 259
renaming 257
file (continued)
  updating 83
file errors 64
file handling
  remove 257
  rename 257
  tmpnam 393
file name length 14
file names, temporary 14
file positioning 91, 124, 126, 128
FILE type 15
fileno() function 98
float.h include file 7
floor() function 99
flushing buffers 87
fmod() function 99
fopen, maximum simultaneous files 14
fopen() function 100
format data as wide characters 133
formatted I/O 108
fpos_t 15
fprintf() function 108
fputc() function 109
fputs() function 112
fputwc() function 113
fputws() function 115
fread() function 116
freopen() function 120
free() function 118
freeopen() function 120
frew() function 122
fscanf() function 123
fseek() function 124
fseeko() function 124
fsetpos() function 126
ftell() function 128
fwide() function 130
fwprintf() function 133
fwrite() function 136
fwscanf() function 137

G
  gamma() function 140
getc() function 142
getchar() function 142
getenv() function 144
gets() function 146
getting
  handle for character mapping 470
  handle for character property classification 472
  wide character from stdin 149
get(c) function 147
getwchar() function 149
gmtime() function 153
gmtime() function 151

H
  handling interrupt signals 330
  HUGE_VAL 8
  hypot() function 155
  hypotenuse 155

I
  1/O errors 64
  idate
    correcting for local time 173, 174
  include files
    assert.h 3
cctype.h 3
decimal.h 3
erro.h 4
except.h 4
float.h 7
limits.h 7
locale.h 7
math.h 7

  pointer.h 8

  recio.h 8

  regen.h 12
  setjmp.h 13
signal.h 13
stdlib.h 13
stddef.h 13
stdio.h 14
stdlib.h 15

  string.h 16
time.h 16
wctype.h 17

  wchar.h 18

  xcvt.h 18

  xdtaa.h 18

  xexch.h 18

  xfdbk.h 18

  indicators, error 64
initial strings 364

  integer
    pseudo-random 239

  Integrated File System errno values 487, 488, 489

  internationalization 7
interrupt signal 330

  invariant character
    hexadecimal representation 503

  isalnum() function 156

  isalpha() function 156

  isascii() function 158

  iscntrl() function 156

  isdigit() function 156

  isgraph() function 156

  islower() function 156

  isprint() function 156

  ispunct() function 156

  isspace() function 156

  isupper() function 156

  iswalpha() function 156

  iswctype() function 162

  iswdigit() function 160

  iswgraph() function 160

  iswlower() function 160

  iswprint() function 160

  iswpunct() function 160

  iswspace() function 160

  iswupper() function 160

  iswxdigit() function 160

  isxdigit() function 156

  conversion

L
  labs() function 164
  langinfo.h include file 7

  language collation string comparison 434

  ldexp() function 166

  ldiv() function 166

  length function 358

  length of variables 496

  library functions
    absolute value
      abs 39

      fabs 81

      labs 164

      hypot 81

      hypotenuse 155

      exp 81
<table>
<thead>
<tr>
<th>Library Functions (continued)</th>
<th>Library Functions (continued)</th>
<th>Library Functions (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponential (continued)</td>
<td>miscellaneous</td>
<td>stream input/output (continued)</td>
</tr>
<tr>
<td>frexp 122</td>
<td>assert 45</td>
<td>fgetpos 91</td>
</tr>
<tr>
<td>ldexp 166</td>
<td>getenv 144</td>
<td>fgets 92</td>
</tr>
<tr>
<td>log 175</td>
<td>longjmp 178</td>
<td>fgetwc 94</td>
</tr>
<tr>
<td>log10 176</td>
<td>perror 209</td>
<td>fgetws 96</td>
</tr>
<tr>
<td>pow 210</td>
<td>putenv 223</td>
<td>fprintf 108</td>
</tr>
<tr>
<td>file handling</td>
<td>rand 239</td>
<td>fputc 109</td>
</tr>
<tr>
<td>fileno 98</td>
<td>rand_r 239</td>
<td>fputs 112</td>
</tr>
<tr>
<td>remove 257</td>
<td>setjmp 321</td>
<td>fputwc 113</td>
</tr>
<tr>
<td>rename 257</td>
<td>srand 338</td>
<td>fputws 115</td>
</tr>
<tr>
<td>tmpfile 392</td>
<td></td>
<td>fread 116</td>
</tr>
<tr>
<td>tmpnam 393</td>
<td></td>
<td>freopen 120</td>
</tr>
<tr>
<td>locale</td>
<td></td>
<td>fscanf 123</td>
</tr>
<tr>
<td>localeconv 168</td>
<td></td>
<td>fseek 124</td>
</tr>
<tr>
<td>nl_langinfo 206</td>
<td></td>
<td>fsetpos 126</td>
</tr>
<tr>
<td>setlocale 322</td>
<td></td>
<td>ftell 128</td>
</tr>
<tr>
<td>strftime 384</td>
<td></td>
<td>fwrite 130</td>
</tr>
<tr>
<td>math</td>
<td></td>
<td>fscanf 133</td>
</tr>
<tr>
<td>acos 40</td>
<td></td>
<td>fwrite 136</td>
</tr>
<tr>
<td>asin 44</td>
<td></td>
<td>fwrite 137</td>
</tr>
<tr>
<td>atan 47</td>
<td></td>
<td>getc 142</td>
</tr>
<tr>
<td>atan2 47</td>
<td></td>
<td>getchar 142</td>
</tr>
<tr>
<td>bessel 52</td>
<td></td>
<td>gets 146</td>
</tr>
<tr>
<td>ceil 63</td>
<td></td>
<td>getwc 147</td>
</tr>
<tr>
<td>cos 66</td>
<td></td>
<td>getwchar 149</td>
</tr>
<tr>
<td>cosh 67</td>
<td></td>
<td>printf 211</td>
</tr>
<tr>
<td>div 77</td>
<td></td>
<td>puts 224</td>
</tr>
<tr>
<td>erf 79</td>
<td></td>
<td>putc 221</td>
</tr>
<tr>
<td>erfc 79</td>
<td></td>
<td>putchar 221</td>
</tr>
<tr>
<td>floor 99</td>
<td></td>
<td>puts 224</td>
</tr>
<tr>
<td>fmod 99</td>
<td></td>
<td>putwc 225</td>
</tr>
<tr>
<td>frexp 122</td>
<td></td>
<td>putwchar 227</td>
</tr>
<tr>
<td>gamma 140</td>
<td></td>
<td>scanf 313</td>
</tr>
<tr>
<td>hypot 155</td>
<td></td>
<td>setbuf 319</td>
</tr>
<tr>
<td>lddiv 166</td>
<td></td>
<td>setvbuf 328</td>
</tr>
<tr>
<td>log 175</td>
<td></td>
<td>sprintf 335</td>
</tr>
<tr>
<td>log10 176</td>
<td></td>
<td>sscanf 339</td>
</tr>
<tr>
<td>modf 205</td>
<td></td>
<td>swprintf 385</td>
</tr>
<tr>
<td>sin 332</td>
<td></td>
<td>swscanf 387</td>
</tr>
<tr>
<td>sinh 333</td>
<td></td>
<td>ungetc 399</td>
</tr>
<tr>
<td>sqrt 337</td>
<td></td>
<td>ungetwc 401</td>
</tr>
<tr>
<td>tan 389</td>
<td></td>
<td>vscanf 412</td>
</tr>
<tr>
<td>tanh 390</td>
<td></td>
<td>vsprintf 414</td>
</tr>
<tr>
<td>memory management</td>
<td></td>
<td>vsprintf 415</td>
</tr>
<tr>
<td>_C_T S malloc_debug 71</td>
<td></td>
<td>vsscanf 416</td>
</tr>
<tr>
<td>_C_T S malloc_info 73</td>
<td></td>
<td>vswprintf 418</td>
</tr>
<tr>
<td>calloc 57</td>
<td></td>
<td>vwprintf 420</td>
</tr>
<tr>
<td>free 118</td>
<td></td>
<td>vwscanf 422</td>
</tr>
<tr>
<td>malloc 180</td>
<td></td>
<td>wprintf 423</td>
</tr>
<tr>
<td>realloc 247</td>
<td></td>
<td>wscanf 482</td>
</tr>
<tr>
<td>memory operations</td>
<td></td>
<td>string manipulation</td>
</tr>
<tr>
<td>memchr 197</td>
<td></td>
<td>strcat 341</td>
</tr>
<tr>
<td>memcmp 198</td>
<td></td>
<td>strchr 342</td>
</tr>
<tr>
<td>memcpy 199</td>
<td></td>
<td>stricmp 343</td>
</tr>
<tr>
<td>memmove 202</td>
<td></td>
<td>strcoll 346</td>
</tr>
<tr>
<td>memset 203</td>
<td></td>
<td>strcpy 347</td>
</tr>
<tr>
<td>wmemchr 476</td>
<td></td>
<td>strlen 358</td>
</tr>
<tr>
<td>wmemcmp 477</td>
<td></td>
<td>strncmp 358</td>
</tr>
<tr>
<td>wmemcpy 478</td>
<td></td>
<td>swprintf 362</td>
</tr>
<tr>
<td>wmemmove 479</td>
<td></td>
<td>swscanf 364</td>
</tr>
<tr>
<td>wmemset 480</td>
<td></td>
<td>swscanf 367</td>
</tr>
<tr>
<td>message catalog</td>
<td></td>
<td>string manipulation</td>
</tr>
<tr>
<td>catchclos 59</td>
<td></td>
<td>strcspn 348</td>
</tr>
<tr>
<td>catgets 60</td>
<td></td>
<td>strlen 358</td>
</tr>
<tr>
<td>catopen 61</td>
<td></td>
<td>strftime 358</td>
</tr>
</tbody>
</table>

542 ILE C/C++ Run-Time Library Functions V5R3
library functions (continued)

string manipulation (continued)

llabs subroutine
absolute value of long long integer 164

lldiv subroutine
perform long long division 166

local time corrections 173
local time corrections (restartable version) 174

locale functions
localeconv 168
setlocale 322
strxfrm 384

locale.h include file 7

localeconv() function 168

locales
retrieve information 206
setting 322

localtime_r() function 174

localtime() function 173

locating storage 118

log() function 175

log10() function 176

logarithmic functions
log 175
log10 176

logic errors 45

logical record length 103

longjmp() function 178

lrecl 103

M

malloc() function 180

math functions
abs 39
acos 40
asin 44
atan 47
atan2 47
bessel 52
div 77
erf 79
erfc 79
exp 81
fabs 81
floor 99
fmod 99
frexp 122
gamma 140
hypot 155
labs 164
ldecp 166
ldiv 166
log 175
log10 176
modf 205
pow 210
sin 332
sinh 333
sqrt 337
tan 389
tanh 390

math.h include file 7

maximum
file name 14
opened files 14

memory allocation

_malloc_debug 71
_malloc_info 73
calloc 57
free 118
malloc 180
realloc 247

memory management

_malloc_debug 71
_malloc_info 73
calloc 57
free 118
malloc 180
realloc 247

memory object functions 33

memory operations

memchr 197
memcmp 198
memcpy 199
memmove 202
memset 203

wmemchr 476
wmemcmp 477
wmemcpy 478
wmemmove 479
wmemset 480

memset() function 203

miscellaneous functions

assert 45
getenv 144
longjmp 178
perror 209
putenv 223
rand 239
realloc 239
setjmp 321

stdlib 338

mktime() function 204

modf() function 205

monetary functions 24

monetary.h include file 8

multibyte functions

_wcscmp 439
_wcscmp 446
btowc 55
mblen 182

mbctowc() function 191

mbstowcs() function 191
mbstowcs() function 192
mbstowcs() function 196
memchr() function 197
memcmp() function 198
memcmp() function 199
memmove() function 202

Index 543
multibyte functions (continued)
towctrans 396 
wctomb 425 
wcschr 431 
wcsncpy 436 
wcsrtomb 451 
wcsspn 453 
wcsxfrm 467 
wctob 468 
wctomb 469 
wctrans 470 
wctype 472 
wctwidth 475

pseudorandom number functions (continued)
  srand 338 
  ptdiff_t 13 
  pushing characters 399 
  putc() function 221 
  putchar() function 221 
  putenv() function 223 
  puts() function 224 
  putwc() function 225 
  putwchar() function 227

Q
  qsort() function 228 
  quick sort 228 
  QXXCHGDA() function 230 
  QXXDTOP() function 231 
  QXXDTOZ() function 232 
  QXXITOP() function 233 
  QXXITOZ() function 233 
  QXXPTOD() function 234 
  QXXPTOI() function 235 
  QXXRTVDA() function 236 
  QXXZTOD() function 237 
  QXXZTOI() function 238

R
  raise() function 238 
  RAND_MAX 15 
  rand_r() function 239 
  rand() function 239 
  random access 124, 128 
  random number generator 239, 338 
  read operations 
    character from stdin 142 
    character from stream 142 
    data items from stream 116 
    formatted 123, 313, 339 
    line from stdin 146 
    line from stream 92 
    reading a character 89 
    scanning 123 
    reading 
      character 142 
      data 313 
      data from stream using wide 
        character 137 
        data using wide-character format 
        string 482 
        formatted data 123 
        items 116 
        line 146 
        messages 60 
        stream 92 
        wide character from stream 94, 147 
        wide-character string from stream 96 
  realloc() function 247 
  reallocation 247 
  realloc() function 247 
  reallocation 247 
  reftime 103 
  recio.h include file 8 
  record format 103 
  record input/output 
    _Racquire 240 
    _Rclose 241

S
  scanf() function 313 
  searching 
    bsearch function 54 
    environment variables 144 
    strings 342, 367, 374 
    strings for tokens 375, 379 
    searching and sorting functions 212 
    seed 338 
    send signal 238 
    separate floating-point value 122 
    setbuf() function 319 
    setjmp.h include file 13 
    setjmp() function 321 
    setlocale() function 322

N
  NDEBUG 3, 45 
  nl_langinfo() function 206 
  nltypes.h include file 8 
  nonlocal goto 178, 321 
  NULL pointer 14, 15

O
  offsetof macro 14 
  opening 
    message catalog 61

P
  passing 
    constants 502 
    variables 502 
  perror() function 209 
  prelude.h include file 8 
  printf() function 211 
  printing 
    error messages 209 
    process control 
      signal 330 
    program termination 
      abort 38 
      atexit 48 
      exit 80 
  pseudo-random integers 239 
  pseudorandom number functions 
    rand 239 
    rand_r 239

record input/output (continued)
  _Rcommit 243 
  _Rdelete 244 
  _Rdevatr 246 
  _Rfeed 260 
  _Rfreev 261 
  _Rformat 262 
  _Rindara 264 
  _Rlocate 268 
  _Ropnfbk 271 
  _Rread 277 
  _Rread 279 
  _Rreadindv 281 
  _Rreadk 284 
  _Rreadl 288 
  _Rreadn 289 
  _Rreadnc 292 
  _Rreadp 293 
  _Rreads 296 
  _Rrelease 297 
  _Rrlslck 299 
  _Rrollbck 300 
  _Rupfb 304 
  _Rupfb 305 
  _Rupfb 306 
  _Rwrted 307 
  _Rwrted 308 
  _Rwrted 309 
  _Rxyread 311 
  record program ending 48 
  redirection 120 
  regcomp() function 250 
  regerror() function 252 
  regex.h include file 12 
  regexec() function 253 
  regfree() function 257 
  remove() function 257 
  rename() function 257 
  reopening streams 120 
  reservation storage 
    _C_TS_malloc_debug 71 
    _C_TS_malloc_info 73 
    malloc 180 
    realloc 247 
  retrieve data area 236 
  retrieve locale information 206 
  rewind() function 259

544  ILE C/C++ Run-Time Library Functions V5R3
stream input/output (continued)
sprintf 335
sscanf 339
swprintf 385
swscanf 387
tmpfile 392
ungetc 399
ungetwc 401
va_arg 402
va_end 402
va_start 402
vfprintf 404
vscanf 406
vwprintf 407
vwscanf 409
vprintf 411
vscanf 412
vsnprintf 414
vsscanf 416
vswprintf 418
vswscanf 420
vprintf 422
vscanf 423
wprintf 481
wscanf 482
stream orientation 130
streams
access mode 120
appending 100, 120
binary mode 120
buffering 319
changing current file position 124, 128
changing file position 259
formated I/O 123, 211, 313, 335, 339
opening 100
reading characters 89, 142
reading data items 116
reading lines 92, 146
reopening 120
rewinding 259
text mode 120
translation mode 120
ungeting characters 399
updating 100, 120
writing characters 109, 221
writing data items 136
writing lines 224
writing strings 112
strerror() function 350
strerror() function 350
strfmon() function 351
strftime() function 354
stricmp() function 357
string manipulation
strcasemp 340
strcat 341
strchr 342
strjmp 343
strcoll() function 351
strtime() function 354
system() function 388

T
tan() function 389
tangent 389
tanh() function 390
testing
ASCII value 158
character property 159, 162
isalnum 156
trigonometric functions (continued)
  asin 44
  atan 47
  atan2 47
  cos 66
  cosh 67
  sin 332
  sinh 333
  tan 389
  tanh 390

type conversion
  atof 49
  atoi 50
  atol 51
  strtod 376
  strtold 380
  strxtoi 382
  toascii 394
  wchar 455
  wcharl 458
  wcstole 463

U
  unget() function 399
  ungetwc() function 401
  updating files 83

V
  va_arg() function 402
  va_end() function 402
  va_start() function 402
  variable argument functions 32
  verify condition 45
  vfprintf() function 404
  vfwprintf() function 407
  vfwscanf() function 409
  vfprintf() function 411
  vsprintf() function 412
  vsprintf() function 414
  vsprintf() function 415
  vsprintf() function 416
  vsprintf() function 418
  vsprintf() function 420
  vsprintf() function 422
  wcsprintf() function 423

W
  wchar.h include file 17
  wcrtomb() function 425
  wcsxstr() function 430
  wcserror() function 431
  wcsrchr() function 433
  wcstoll() function 434
  wcsxstr() function 435
  wcspn() function 436
  wcftime() function 437
  wmemset() function 440
  wstrerror() function 441
  wcsxstr() function 442
  wcsxstr() function 443
  wcslcspn() function 445
  wcspbrk() function 447
  wcftime() function 448
  wcsxstr() function 450
  wcsxstr() function 451
  wcspn() function 453
  wcss() function 455
  wcsm() function 455
  wcsw() function 457
  wcstol() function 458
  wcsxstr() subroutine
    wide character to long long
    integer 458
  wcstombs() function 460
  wcstoul() function 463
  wcsxstr() subroutine
    wide-character string to unsigned long
    long 463
  wcsxstr() include file 17
  wcsxstr() function 465
  wcsxstr() function 466
  wcsxstr() function 467
  wcst() function 468
  wcst() function 469
  wcstr() function 470
  wcsc() function 472
  wcst() function 475
  wide character string functions 36
  wcstr() function 476
  wcstr() function 477
  wcstr() function 478
  wcstr() function 479
  wcstr() function 480
  wprintf() function 481
  write operations
    character to stdout 109, 221
    character to stream 109, 221, 399
    data items from stream 136
    formatted 108, 211, 335
    line to stream 224
    printing 136
    string to stream 112
  writing
    character 109, 221
    data items from stream 136
    formatted data to a stream 108
    string 112, 224
    wide character 115, 225, 227
    wide characters to a stream 133
    wide-character string 115
    wscanf() function 482

X
  xxctv.h include file 18
  xxdtaa.h include file 18
  xxenv.h include file 18
  xxfdbk.h include file 18
Readers’ Comments — We’d Like to Hear from You

iSeries
ILE C/C++
Run-Time Library Functions
Version 5

Publication No. SC41-5607-02

Overall, how satisfied are you with the information in this book?

<table>
<thead>
<tr>
<th></th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall satisfaction</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

How satisfied are you that the information in this book is:

<table>
<thead>
<tr>
<th></th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accurate</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Complete</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Easy to find</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Easy to understand</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Well organized</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Applicable to your tasks</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name
__________________________

Address
__________________________

Company or Organization
__________________________

Phone No.
__________________________