OS/390

UNIX System Services
tcsh (C Shell) Kit Support Guide

OS/390

UNIX System Services
tcsh (C Shell) Kit Support Guide

Note

Before using this information and the product it supports, be sure to read the general information under Appendix A, “Notices” on page 229.

December 1999 Edition

Contents

Tables

About This Book

This document supports the OS/390 Version 2 Release 8 tcsh (C Shell) Kit, SK3T-4243–00. This kit provides the tsch shell for OS/390 Version 2 Release 8 UNIX System Services. This document is available only on the OS/390 UNIX System Services web site at:

http://www.ibm.com/s390/unix/tcsh/

The tcsh shell provides users with a means to run tcsh scripts and also offers features such as programmable word completion and spelling correction that were not available in the previous versions of OS/390.

Information to support the addition of tcsh has been added to the following OS/390 manuals. For easy reference, all tcsh information from these books is excerpted here:

- *OS/390 UNIX System Services Planning*, SC28-1890
- *OS/390 UNIX System Services User's Guide*, SC28-1891
- *OS/390 UNIX System Services Command Reference*, SC28-1892
- *OS/390 UNIX System Services Messages and Codes*, SC28-1908

To view the original OS/390 Version 2 Release 8 UNIX System Services library, go to http://www.ibm.com/s390/os390 and select "The Library".

Part 1. OS/390 UNIX System Services Planning

Chapter 1. tcsh in Version 2 Release 9

tcsh Shell

Description

A new C shell, tcsh, is available for OS/390 UNIX.

What This Change Affects

This support may affect the following areas of OS/390 UNIX processing.

Area	Considerations
Administration	The system administrator can modify the **/etc/csh.cshrc**, **/etc/csh.login**, and **/etc/csh.logout** files to contain settings for all users.
Application Development	The **pk, make, lex,** and **yacc** utilities may need to be enabled. Procedures regarding national code pages may have to be updated. See Chapter 3.
Auditing	None
Customization	The startup files found in the user's home directory can be changed to suit individual preferences.
General User	Users should be aware that the tcsh shell is available.
Operations	None
Interfaces	None
Maintenance	None

Migration Tasks

The following migration tasks are associated with this enhancement. A **required** task must be performed regardless of whether you implement this function at your installation. An **optional** task need only be performed if your installation uses the specified functions.

Task	Condition	Reference Information
Copy **/samples/csh.cshrc** into **/etc/csh.cshrc** and merge any system customizations from **/etc/profile**, changing the syntax to C shell syntax. This task is recommended.	Optional	Chapter 2
Copy **/samples/csh.login** into **/etc/csh.login**. This task is recommended.	Optional	Chapter 2
Copy **/samples/.tcshrc** into user's **$HOME/.tcshrc**. This task is recommended.	Optional	Chapter 2
Copy **/samples/.login** into user's **$HOME/.login**. This task is recommended.	Optional	Chapter 2

Task	Condition	Reference Information
Copy **/samples/complete.tcsh** into **/etc/complete.tcsh**. This task is recommended.	Optional	Chapter 2

Chapter 2. Customizing the tcsh Shell

This chapter explains how to customize the tcsh shell.

To work interactively, the shell user connects to the system in one of the following ways:

- Logs on to TSO/E and enters the TSO/E command OMVS, which invokes a shell. The OMVS command provides a 3270 terminal interface to the shell, and you can use the options to customize the interface—for example, function key settings.
- Issues the **rlogin** command, which invokes the shell. It provides an asynchronous terminal interface to the shell, familiar to UNIX users.
- Issues the **telnet** command. It provides an asynchronous terminal interface to the shell, familiar to UNIX users.
- Performs a Communications Server login from a serially attached terminal. This provides an asynchronous terminal interface, familiar to UNIX users.

For a description of these interfaces to the shell, see *OS/390 UNIX System Services User's Guide*.

After the user logs in to the tcsh shell, the system initializes the shell for that user. During the initialization, the system does the following:

1. Determines whether the user is authorized to use the shell by checking for a UID value in the user's RACF user profile. It also checks that the user's RACF group has a GID assigned to it.
2. Sets the LOGNAME, HOME, and SHELL environment variables from data in the RACF user profile, which was specified in the RACF ADDUSER and ALTUSER commands. See “Customizing the RACF User Profile” on page 6.
3. Connects the user to the initial working directory that was identified in the HOME environment variable in the RACF user profile. If the RACF user profile does not identify a working directory, the system uses the root as the user's working directory and issues a message.
4. Invokes the shell named in the SHELL environment variable. For the tcsh shell, this will be **/bin/tcsh**.
5. Runs commands that were specified in the **/etc/csh.login** file, if one was provided.
6. Runs commands that were specified in the **$HOME/.login** file, if one was provided.
7. Runs commands that were specified in the **/etc/csh.cshrc** file, if one was provided.

8. Runs commands that were specified in the **$HOME.tcshrc** file, if one was provided.
9. Runs commands that were specified in the **$HOME/.cshrc** file, if one was provided.

Customizing the Shell and Environment Variables

If the shell or environment variables are not set in one of the files in the following list, or in a shell command or shell script, then they are not set and have no value. Setting the variables is optional. The places to set shell or environment variables, in the order that the system sets them, are:

1. The RACF user profile
2. **/etc/csh.login**, the system-wide file that sets environment variables, if it is a login shell.
3. **$HOME/.login**, which sets environment variables for individual users, if it is a login shell.
4. **/etc/csh.cshrc**, the system-wide file that sets shell variables, some environment variables (like PATH), and umask. It also defines command aliases. It is used by subshells.
5. **$HOME.tcshrc**, which sets shell variables for individual users. It is used by subshells.
6. **$HOME/.cshrc**, if it is provided for compatibility with the C shell.

Later settings take precedence. For example, the values set in **$HOME/.login** override those in **/etc/csh.login**.

Similar systems usually have an **/etc/passwd** file, which contains the HOME and PROGRAM environment variables, plus the users' passwords. To provide better security, the OS/390 shell does not use the **/etc/passwd** file; instead, it uses the initial values assigned to these variables in the RACF user profiles. RACF maintains the passwords.

Customizing the RACF User Profile

The security administrator defines a user by creating a RACF user profile with an ADDUSER command or alters the user profile with an ALTUSER command. The RACF user profile contains the settings for the following environment variables:

LOGNAME Specifies the TSO/E user ID

HOME Specifies the pathname of the user's home directory as specified in the HOME parameter of the RACF command. If the HOME parameter was not specified, HOME is the root directory.

SHELL Specifies the pathname of the file containing the shell program as specified in the PROGRAM parameter on the RACF command. If PROGRAM was not specified, SHELL is **/bin/sh**.

The PROGRAM parameter can specify a special-purpose shell or another kind of program.

Customizing /etc/csh.login

The **/etc/csh.login** file is used for setting environment variables such as TERM and is only read by tcsh when it is a login shell.

Important

Because **/etc/csh.login** is the tcsh equivalent to **/etc/profile** for **sh**, you need to keep system-wide information for both sets of users in synch. Any customization that you have done for **/etc/profile** (such as setting environment variables) needs to be duplicated in C shell syntax in **/etc/csh.login**. Future changes to **/etc/profile** also need to be made to **/etc/csh.login**. If you maintain a non-OS/390 UNIX system, you could consider porting **/etc/csh.cshrc** and **/etc/csh.login** from that system to OS/390 and merging them with the OS/390 samples.

Figure 1 on page 8 shows a sample **/samples/csh.login** file:

```
# ============================================================================
#                     STEPLIB environment variable
#                     ----------------------------
tty -s

set tty_rc=$status
if (($?STEPLIB == 0 ) && ($tty_rc == 0)) then
    setenv STEPLIB none
    exec tcsh -l
endif
unset tty_rc

# ============================================================================
#                      TZ environment variable
#                      -----------------------
setenv TZ EST5EDT

# ============================================================================
#                      LANG environment variable
#                      -------------------------
setenv LANG C

# ============================================================================
#                     LIBPATH environment variable
#                     ----------------------------
# ============================================================================
setenv LIBPATH /lib:/usr/lib:.

# ============================================================================

# ============================================================================
#                      MAIL environment variable
#                      -------------------------
setenv MAIL /usr/mail/$LOGNAME

# ============================================================================
# Start of c89/cc/c++ customization section
# ============================================================================
# foreach _CMP(_C89_CC_CXX)
#   setenv ${_CMP}_CLIB_PREFIX "CBC"
#   setenv ${_CMP}_PLIB_PREFIX "CEE"
#   setenv ${_CMP}_SLIB_PREFIX "SYS1"
#   setenv ${_CMP}_INCDIRS "/usr/include /usr/lpp/ioclib/include"
#   setenv ${_CMP}_LIBDIRS "/lib /usr/lib"
#
# Esoteric unit for data sets:
#   setenv ${_CMP}_WORK_UNIT "SYSDA"
# end
# unset _CMP
#
# ============================================================================
# End of c89/cc/c++ customization section
# ============================================================================
```

Figure 1. Partial Contents of IBM-Supplied /samples/csh.login

Use the **cp** command to copy **/samples/csh.login** to **/etc/csh.login**. Edit **/etc/csh.login** to change or add environment variables.

Customizing $HOME/.login

To change or add environment variables such as TERM that are customized for individual users, first use the **cp** command to copy **/samples/.login** to **$HOME/.login**. Then edit the file to change or add environment variables. The **$HOME/.login** file is only read by tcsh when it is a login shell.

Customizing /etc/csh.cshrc

The **/etc/csh.cshrc** is the system-wide profile for tcsh shell users and is read by subshells.

Figure 2 shows suggested settings for **/etc/csh.cshrc** provided in the IBM-supplied **/samples/.csh.cshrc**:

```
# ======================================================================
# path shell variable
# ======================================================================
#
# Specifies the list of directories that the system searches for an
# executable command.
set path = ( /bin)
# ======================================================================
#
#  umask variable
#
umask 022
# ======================================================================
```

Figure 2. Partial Contents of IBM-Supplied /samples/csh.cshrc

Use the **cp** command to copy the **/samples/csh.cshrc** file to **/etc/csh.cshrc**. Then edit **/etc/csh.cshrc** to change or add shell variables.

Customizing $HOME/.tcshrc

The **$HOME/.tcshrc** file contains commands that set or change the values of shell variables for individual users and is read by subshells. HOME is a variable for the pathname for a user's home directory. The values set in **$HOME/.tcshrc** overrides those in **/etc/csh.cshrc**.

Use the **cp** command to copy **/samples/.tcshrc** to your **$HOME** directory. Then edit the new file to change or add shell variables.

Customizing c89, cc, and c++ (cxx)

The **c89** utility is customized by setting environment variables. The ones that most commonly require setting are specified in the **c89** customization section in **/etc/csh.login**. *OS/390 UNIX System Services Command Reference* lists the rest of the variables that might require setting for typical **c89** usage.

OS/390 UNIX System Services Command Reference, in its **c89** section, assumes that the current level of OS/390 C/C++ compiler and Language Environment

run-time library will be used. If you must use a previous level of the compiler, or target the executables produced by **c89** to run on a previous level of the run-time library, then you must customize other environment variables, which are only documented here.

The environment variables used by the **cc** utility have the same names as the ones used by **c89** except that the prefix is _CC instead of _C89. Likewise, for the **c++** (**cxx**) utility, the prefix is _CXX instead of _C89. Normally, you do not need to explicitly assign the environment variables for all three utilities. These **eval** commands set the variables for the other utilities, based on those set for **c89**. .

By placing any customization statements for **c89** into **/etc/csh.login**, the environment variables are automatically be assigned for **cc** and **c++** as well. After you customize **/etc/csh.login**, it is unlikely that it will need to be changed again. However, you can change the variables at any time; the next time a user logs into the shell, they will get the new settings

Using Non-Default High-Level Qualifiers

If any of the following installed products did not use the installation default for the high-level qualifier, then the appropriate environment variable must be assigned to make **c89** aware of this. The environment variables in this table are set to the default values for the current level of OS/390, but you will need to set them to your high-level qualifiers.

Note: These high-level qualifiers are used to construct the names of data sets used by **c89**. All named data sets used by **c89** must be cataloged.

Element	c89 Environment Variable
C/C++ Compiler	_C89_CLIB_PREFIX=CBC
Language Environment	_C89_PLIB_PREFIX=CEE
BCP	_C89_SLIB_PREFIX=SYS1

Using a System That Does Not Have UNIT=SYSDA

If the system is not configured with an esoteric unit SYSDA, or some other esoteric unit is to be used for VIO temporary unnamed work data sets set by **c89**, the following environment variable needs to be set. Specifying a null value for this variable ("") results in **c89** using an installation-defined default for the UNIT. The environment variable is shown being set to the default value:

Temporary Unnamed Data Set Unit	c89 Environment Variable
All c89-allocated work data sets	_C89_WORK_UNIT=SYSDA

Selecting Previous C/C++ Compilers

For each OS/390 release, there is a default compiler for **c89**, **cc**, or **c++** (**cxx**) to use. Optionally, you can choose to use a non-default compiler. This section lists the compiler choices for each release, including the default compilers; the environment variable settings for each compiler are identified.

The **c89/cc/c++** utilities use a number of environment variables. The default values are specified as comments in the **/samples/csh.login** file that is shipped with each release. The environment variables for:

- **c89** begin with the prefix **_C89**
- **cc** begin with the prefix **_CC**
- **c++** begin with the prefix **_CXX**

If the C/C++ Class Library DLLS are used in building your executables (the default for the **c++** utility), then this will also target your executable for the same level of C/C++ Class Library

Using the Same Compiler for the Entire System

If you are using the same compiler for the entire system, then put the compiler data set name in the linklist. By default, the linklist contains the name of the default compiler

If you are using a compiler that is not the system-wide default, then you must specify the compiler data set name in the STEPLIB environment variable and assign it. Note that this may affect performance somewhat.

These are the statements for each compiler version:

- For the OS/390 V1R2 compiler and onward:

  ```
  setenv STEPLIB "CBC.SCBCCMP"
  ```

- For the IBM C/C++ V3R2 compiler:

  ```
  setenv STEPLIB "CBC.V3R2M0.SCBC3CMP"
  ```

- For the AD/Cycle C/370 V1R2 compiler:

  ```
  setenv STEPLIB "EDC.V1R2M0.SEDCDCMP"
  ```

Invoking Earlier Levels of the Compiler

If you are invoking an earlier level of the compiler and Language Environment headers and stubs, in addition to the normal steps, you must:

- Ensure that the older compiler and Language Environment headers and stubs are available on your system. The older versions of the compiler and run-time library are not included with each new version of OS/390. You must save them from an earlier installation.
- Set _xxx_PVERSION, and typically also _xxx_CLIB_PREFIX and _xxx_PLIB_PREFIX. You also need to put the _xxx_CVERSION level of compiler first in the MVS search order (typically via STEPLIB).
- Assign STEPLIB for the version of the compiler you want to use.
- Set {_INCDIRS} to point to the location of the older version of the header files instead of **/usr/include**.
- Set _xxx_CVERSION and _xxx_CLIB_PREFIX. For details on these environment variables, look at the description of the **c89** utility in *OS/390 UNIX System Services Command Reference*.

OS/390 V2R6, V2R7, and V2R8

For OS/390 V2R6, V2R7, and V2R8, the OS/390 V2R6 compiler is the default for **c89**, **cc**, and **c++**. You do not need to set the _xxx_CVERSION and _xxx_CLIB_PREFIX environment variables to use the default. The default settings are:

```
_C89_CVERSION="0x22080000"
_CC_CVERSION="0x22080000"
_CXX_CVERSION="0x22080000"

_C89_CLIB_PREFIX="CBC"
_CC_CLIB_PREFIX="CBC"
_CXX_CLIB_PREFIX="CBC"
```

To compile with an earlier level of the compiler, you need to set the environment variables to point to one of those:

- OS/390 V2R4 compiler
- OS/390 V1R3 compiler
- OS/390 V1R2 compiler
- IBM C/C++ V3R2 compiler
- AD/Cycle C/370 V1R2 compiler

OS/390 V2R6 Run-Time Library

The default run-time library is OS/390 V2R6 Language Environment. The default environment settings are:

```
_C89_PVERSION="0x22060000"
_CC_PVERSION="0x22060000"
_CXX_PVERSION="0x22060000"


_C89_PLIB_PREFIX="CEE"
_CC_PLIB_PREFIX="CEE"
_CXX_PLIB_PREFIX="CEE"
```

To build an executable targeted for an earlier Language Environment release, you can optionally specify:

- OS/390 V2R4 Language Environment. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V2R4 Run-Time Library” on page 13.
- OS/390 V1R3 Language Environment. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R3 Run-Time Library” on page 15.
- OS/390 V1R2 Language Environment. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R2 Run-Time Library” on page 15.
- Language Environment for MVS and VM V1R5. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R1 Run-Time Library” on page 16.

Targeting an Earlier Release

When targeting an earlier release, you must pass the 'compat' option to the binder:

```
-WI,compat=pm2
```

A convenient way to do this as part of the setup is to use _xxx_OPTIONS(along with the other environment variables like _xxx_VERSION). For example, to do this for **c89**, you can issue:

```
setenv _C89_OPTIONS "-WI,compat=pm2"
```

If you target a Language Environment release that is not the default, remember to change the setting of the {_INCDIRS} environment variable so that it does not point to **/usr/include** (the default setting). You can set {_INCDIRS} to a null string. For example:

```
setenv _C89_INCDIRS=""
```

Or, if you have other directories that you want to be automatically searched, they can be added to {_INCDIRS}, as long as **/usr/include** is deleted.

OS/390 V2R5 and V2R4

For OS/390 V2R5 and V2R4, the OS/390 V2R4 compiler is the default for **c89**, **cc**, and **c++**. You do not need to set the _xxx_CVERSION and _xxx_CLIB_PREFIX environment variables to use the default. The default settings are:

```
_C89_CVERSION="0x22040000"
_CC_CVERSION="0x22040000"
_CXX_CVERSION="0x22040000"

_C89_CLIB_PREFIX="CBC"
_CC_CLIB_PREFIX="CBC"
_CXX_CLIB_PREFIX="CBC"
```

To compile with an earlier level of the compiler, you would need to set the environment variables to point to one of these:

- OS/390 V1R3 compiler
- OS/390 V1R2 compiler
- IBM C/C++ V3R2 compiler
- AD/Cycle C/370 V1R2 compiler

OS/390 V2R4 Run-Time Library

The default run-time library is OS/390 V2R4 Language Environment. The default environment settings are:

```
_C89_PVERSION="0x22040000"
_CC_PVERSION="0x22040000"
_CXX_PVERSION="0x22040000"


_C89_PLIB_PREFIX="CEE"
_CC_PLIB_PREFIX="CEE"
_CXX_PLIB_PREFIX="CEE"
```

To build an executable targeted for an earlier Language Environment release, you can optionally specify:

- OS/390 V1R3 Language Environment. To do this, assign the _xxx_PRELIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R3 Run-Time Library” on page 15.
- OS/390 V1R2 Language Environment. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R2 Run-Time Library” on page 15.
- Language Environment for MVS and VM V1R5. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in “OS/390 V1R1 Run-Time Library” on page 16.

Targeting an Earlier Release

When targeting an earlier release:

1. You must pass the 'compat' option to the binder:

   ```
   -WI,compat=pm2
   ```

   A convenient way to do this as part of the setup is to use _xxx_OPTIONS (along with the other environment variables like _xxx_VERSION). For example, to do this for **c89**, you can issue:

   ```
   setenv _C89_OPTIONS="-WI,compat=pm2"
   ```

2. If you target a Language Environment release that is not the default, remember to change the setting of the {_INCDIRS} environment variable so that it does not point to **/usr/include** (the default setting). You can set {_INCDIRS} to a null string. For example:

   ```
   setenv _C89_INCDIRS=""
   ```

   Or, if you have other directories that you want to be automatically searched, they can be added to {_INCDIRS}, as long as **/usr/include** is deleted.

OS/390 V1R3

The OS/390 V1R3 compiler is the default for **c89**, **cc**, and **c++**. You do not need to set the _xxx_CVERSION and _xxx_CLIB_PREFIX environment variables to use the default. The default settings are:

```
_C89_CVERSION="0x21030000"
_CC_CVERSION="0x21030000"
_CXX_CVERSION="0x12030000"

_C89_CLIB_PREFIX="CBC"
_CC_CLIB_PREFIX="CBC"
_CXX_CLIB_PREFIX="CBC"
```

To compile with an earlier level of the compiler, you would need to set the environment variables to point to one of these:

- OS/390 V1R2 compiler
- IBM C/C++ V3R2 compiler
- AD/Cycle C/370 V1R2 compiler

OS/390 V1R3 Run-Time Library

The default run-time library is OS/390 V2R3 Language Environment. The default environment settings are:

```
_C89_PVERSION="0x21030000"
_CC_PVERSION="0x21030000"
_CXX_PVERSION="0x12030000"


_C89_PLIB_PREFIX="CEE"
_CC_PLIB_PREFIX="CEE"
_CXX_PLIB_PREFIX="CEE"
```

To build an executable targeted for an earlier Language Environment release, you can optionally specify:

- OS/390 V1R2 Language Environment. To do this, assign the _xxx_PRELIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in "OS/390 V1R2 Run-Time Library."
- Language Environment for MVS and VM V1R5. To do this, assign the _xxx_PLIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in "OS/390 V1R2 Run-Time Library."

If you target a Language Environment release that is not the default, remember to change the setting of the {_INCDIRS} environment variable so that it does not point to **/usr/include** (the default setting). You can set {_INCDIRS} to a null string. For example:

```
setenv _C89_INCDIRS=""
```

Or, if you have other directories that you want to be automatically searched, they can be added to {_INCDIRS}, as long as **/usr/include** is deleted.

OS/390 V1R2

For OS/390 V1R2, you have a choice of using these compilers:

- **OS/390 V1R2 C/C++ Compiler.** You can specify this compiler to use with **c89**, **cc**, and **c++**. To use it, you need to set the environment variables to point to this compiler (see "OS/390 V1R2 C/C++ Compiler" on page 16).
- **IBM C/C++ V3R2 Compiler.** For OS/390 Release 2, This is the default compiler for **c++**. You can specify this compiler to use with **c89** or **cc**. See "OS/390 C/C++ V3R2 Compiler" on page 17.
- **AD/Cycle C/390 V1R2 Compiler.** For OS/390 Release 2, this is the default compiler for **c89** and **cc**. It cannot be used with **c++**. See "AD/Cycle C/370 V1R2 Compiler" on page 17.

OS/390 V1R2 Run-Time Library

The default run-time library is OS/390 V1R2 Language Environment. The default environment settings are:

```
_C89_PVERSION="0x21020000"
_CC_PVERSION="0x21020000"
_CXX_PVERSION="0x12020000"
```

```
_C89_PLIB_PREFIX="CEE"
_CC_PLIB_PREFIX="CEE"
_CXX_PLIB_PREFIX="CEE"
```

To build an executable targeted for an earlier Language Environment release, you can optionally specify Language Environment for MVS and VMV1R5. To do this, assign the _xxx_PRELIB_PREFIX and _xxx_PVERSION environment variables with the settings shown in "OS/390 V1R1 Run-Time Library."

If you target a Language Environment release that is not the default, remember to change the setting of the {_INCDIRS} environment variable so that it does not point to **/usr/include** (the default setting). You can set {_INCDIRS} to a null string. For example:

```
setenv _C89_INCDIRS=""
```

Or, if you have other directories that you want to be automatically searched, they can be added to {_INCDIRS}, as long as **/usr/include** is deleted.

OS/390 V1R1

For OS/390 V1R2, you have a choice of using these compilers:

- **IBM C/C++ V3R2 Compiler.** For OS/390 Release 1, this is the default compiler for **c++**. You can specify this compiler to use with **c89** or **cc** (see "OS/390 C/C++ V3R2 Compiler" on page 17).
- **AD/Cycle C/390 V1R2 Compiler.** For OS/390 Release 1, this is the default compiler for **c89** and **cc**. It cannot be used with **c++**. See "OS/390 V1R2 C/C++ Compiler"

OS/390 V1R1 Run-Time Library

The default run-time library is Language Environment for MVS and VM V1R5. The default environment variable settings are:

```
setenv _C89_PVERSION="0x11050000"
setenv _CC_PVERSION="0x11050000"
setenv _CXX_PVERSION="0x11050000"
```

```
setenv _C89_PLIB_PREFIX="CEE.V1R5M0"
setenv _CC_PLIB_PREFIX="CEE.V1R5M0"
setenv _CXX_PLIB_PREFIX="CEE.V1R5M0"
```

OS/390 V1R2 C/C++ Compiler

For **c89**, **cc** or **c++** to use this compiler, you must specify one of these:

```
setenv _C89_CVERSION "0x21020000"
setenv _CC_CVERSION "0x21020000"
setenv _CXX_CVERSION "0x21020000"
```

Then this corresponding variable is set by default:

```
setenv _C89_CLIB=_PREFIX "CBC"
setenv _CC_CLIB_PREFIX "CBC"
setenv _CXX_CLIB_PREFIX "CBC"
```

OS/390 C/C++ V3R2 Compiler

For OS/390 V1R1 and V1R2, this is the default compiler for **c++**. To use this compiler when it is not the default, you must specify one of these:

```
setenv _C89_CVERSION "0x13020000"
setenv _CC_CVERSION "0x13020000"
setenv _CXX_CVERSION "0x13020000"
```

Then this corresponding variable is set by default:

```
setenv _C89_CLIB=_PREFIX "CBC.V3R2M0"
setenv _CC_CLIB_PREFIX "CBC.V3R2M0"
setenv _CXX_CLIB_PREFIX "CBC.V3R2M0"
```

AD/Cycle C/370 V1R2 Compiler

For OS/390 V1R1 and V1R2, this is the default compiler for **c89** and **cc**. You cannot use it for **c++**. To use this compiler when it is not the default, you must specify one of these:

```
setenv _C89_CVERSION "0x11020000"
setenv _CC_CVERSION "0x11020000
```

Then this corresponding variable is set by default:

```
setenv _C89_CLIB=_PREFIX "EDC.V1R2M0"
setenv _CC_CLIB_PREFIX "EDC.V1R2M0"
```

Customizing the terminfo Database

Full-screen application programs such as the **vi** editor and the **more** utility require a terminfo database. The terminfo database contains the characteristics of different terminal types that are used to run these full-screen applications.

The terminfo database is shipped as part of OS/390 UNIX System Services Application Services. The database is populated with the terminal types defined by **ibm.ti**, **dec.ti**, **wyse.ti**, and **dtterm.ti**. The database is in the directory **/usr/share/lib/terminfo** and the source files are in **/samples**.

If you have been using Release 6 or earlier, you will need to comment out the **tic** commands from your customized copy of **/etc/rc**.

To define any other terminal or workstation for a terminfo database, do the following steps:

1. Create a subdirectory in your home directory for the terminfo database terminal definition. For example: **mkdir /u/myhome/terminfo** where *myhome* is the name of your home directory.
2. Copy the **.ti** file for the terminal that you are building the terminfo database for, into the directory that you just created. You can obtain the terminal file from another UNIX operating system, if necessary. For example, you can copy the file **pc.ti** into the directory:

   ```
   /u/myhome/terminfo/pc.ti
   ```

3. Set the TERMINFO environment variable to the directory that the terminal definitions are in:

```
setenv TERMINFO /u/myhome/terminfo
```

4. Run the **tic** command, specifying the terminal file. For example,

   ```
   tic /u/myhome/terminfo/pc.ti
   ```

5. Set the TERM environment variable to the name of the terminal you wish to use:

   ```
   setenv TERM sun
   ```

Customizing Electronic Mail

The **mailx** shell command sends electronic mail between shell users. For **mailx** processing, do the following:

- Set up a system startup file, **/etc/mailx.rc**, which contains variable values and definitions common to all shell users. The IBM-supplied sample is in **/samples/mailx.rc**. Copy this file to **/etc/mailx.rc**.
- If you use a system mailbox directory other than **/usr/mail**, identify it in the $MAIL environment variable in **/etc/csh.login**.

Users can give names to mail files using variables in **$HOME/.login** or they can use files with the default names. See “Customizing /etc/csh.login” on page 7.

Chapter 3. Customizing for Your National Code Page in the Shell

This chapter explains how to customize for your national code page in the your tcsh shell. It also explains how to customize your system so that OS/390 UNIX messages are displayed in Japanese or Simplified Chinese. (OS/390 UNIX messages are available in English, Japanese, or Simplified Chinese.)

See the appendix in *OS/390 UNIX System Services User's Guide* for information on the locale objects, source files, and charmaps that the OS/390 UNIX System Services Application Services support.

For the tcsh shell, if you want to set the language for yourself, or for just one user, you can make these changes in the **$HOME/.login**, or log on to the OS/390 shell and set the LANG and NLSPATH environment variables.

"Setting Up Your National Code Page" explains how to set up a default language for all users of the OS/390 shell. "Customizing for Japanese and Simplified Chinese" on page 21 details what you need to do when customizing for a language other than English.

Setting Up Your National Code Page

This section provides the general setup information for setting up your national code page for shell users. If you will be using Japanese or Simplified Chinese, you still need to do these steps first before going on to "Customizing for Japanese and Simplified Chinese" on page 21.

1. Copy the login file for your shell, if necessary. Copy **/samples/profile** to **/samples/csh.login**. You may have already done this, as described in "Customizing /etc/csh.login" on page 7.
2. Customize the login file for your shell. Customize **/etc/csh.login** so that your selected national page is enabled when the tcsh shell is first invoked. Be careful that the shell, with the updated **/etc/csh.login** does not keep restarting itself after you restart the shell. To make sure that **exec tcsh -l** is executed only once, you can copy the code shown in the sample **/etc/csh.login**, updated with your national code page.
3. You must convert from ASCII to your national code page. Change the data conversion for rlogin and Communications Server terminal sessions using the **chcp** command. The sample **/etc/csh.login** in Figure 3 on page 20 shows examples of statements to convert the terminal session data using ASCII code page ISO8859-1 and EBCDIC code page IBM-277.

```
tty -s
set tty_rc=$status
if (($?LOCALE_SWITCH == 0 && tty_rc == 0)) then
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
  echo " - Logon shell will now be invoked to reflect        - "
  echo " - code page IBM-277                                 - "
  echo " - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - "
  setenv LOCALE_SWITCH EXECUTED
  setenv LANG C
  setenv LC_ALL Da_DK.IBM-277
  # Issue chcp if not using OMVS command
  if ($?_BPX_TERMPATH != "OMVS" ) then
     chcp -a ISO8859-1 -e IBM-277
     endif
     exec tcsh -l
  endif
  unset tty_rc
```

Figure 3. Sample /etc/csh.login for Customizing National Code Pages

4. Customize certain utilities. You need to customize **lex**, **mailx**, **make**, and **yacc**. These utilities expect all input files, both system files and user files, to be in the same code page. Use the **iconv** command to convert the following system files to your selected locale:

   /etc/yylex.c
   /etc/mailx.c
   /etc/startup.c
   /etc/yyparse.c

   For example,

   ```
   mv /etc/mailx.c.277 /etc/mailx.c
   iconv -f IBM-1047 -t IBM-277 /etc/mailx.c >/etc/mailx.c.277
   ```

5. Update BPXBATCH or OSHELL, if necessary. If you use BPXBATCH or OSHELL (which uses BPXBATCH), you must do this step in order to get the code page working immediately under BPXBATCH and OSHELL. Use the STDENV ddname to point to a file or data set that contains the environment variable definitions for the code page. The code page you specify will not affect the shell because ddname is read before the first shell is invoked, (Because the STDENV DD statement does not affect the OMVS command, you need to put the environment variables in **/etc/csh.login,** also.)

   For more information about BPXBATCH and STDENV, see *OS/390 UNIX System Services User's Guide*.

6. Customize for Japanese or Simplified Chinese, if needed. If you are customizing for Japanese or Simplified Chinese, there are more steps you need to follow. Go to “Customizing for Japanese and Simplified Chinese” on page 21.

7. Save **/etc/csh.login**.

8. Verify your code page. To find out what code page was set up for your shell, issue:

   ```
   echo $HOME
   ```

If you entered the shell before the code page was set up, you will see $HOME. Otherwise, the shell displays the pathname of your home directory. The $ should be read as your code page's dollar sign.

In that case, for the tcsh shell, you need to modify **/etc/csh.login** as described above to enable the national code page support .

Customizing for Japanese and Simplified Chinese

If you are customizing for Japanese or Simplified Chinese, you need to make more changes to your login file after completing the steps in “Setting Up Your National Code Page” on page 19. You also need to customize **/etc/init**.

These changes take effect the next time OMVS is started.

In addition, this section explains how to customize using MMS (MVS Message Service), TSO messages and help panels, and ISPF.

The examples are for Japanese.

Customize the login File

While editing **/etc/csh.login**:

1. Change the line **LANG=C** to **setenv LANG Ja_JP**.
2. Change to add the **LC_ALL** line:

   ```
   setenv LC_ALL Ja_JP.IBM-939
   ```

   This enables you to run in the Japanese locale.
3. Save **/etc/csh.login**.

Customize /etc/init

The next series of steps help you set the **/etc/init** process to display messages in Japanese. While editing **/etc/init.options**:

1. Locate the line

   ```
   *e  LANG=En_US.IBM-1047
   ```

   Replace it with:

   ```
   -e  LANG=Ja_JP
   ```
2. Locate the line

   ```
   *e  NLSPATH=/usr/lib/nls/msg/%L/%N
   ```

   Replace it with:

   ```
   -e  NLSPATH=/usr/lib/nls/msg/%L/%N
   ```
3. Save **/etc/init.options**.

Displaying Translated Messages Using MVS Message Service (MMS)

To set the system default to display translated messages, do the following:

1. Compile the English and translated message skeletons.
2. Create or update the following SYS1.PARMLIB members to initialize values for MMS:
   - MMSLSTxx
   - CNLcccxx
   - CONSOLxx to define the MMSLSTxx member in effect for the system
3. Activate MMS. One way to do this is to issue `SET MMS=xx` from the MVS operator console, where *xx* refers to the MMSLSTxx member of SYS1.PARMLIB.

MMS does not support translating messages to the MVS operator console. To see translated messages, you must set up a TSO/E console that mirrors the operator's console. TSO/E displays Japanese and Simplified Chinese messages to DBCS terminals only.

TSO/E Messages and Help Panels

TSO/E messages are issued through MMS. For more information, see the section "Providing Translated Messages" in the chapter "Customizing TSO/E for Different Languages" in *OS/390 TSO/E Customization*.

If you do not want Japanese or Simplified Chinese to be the default language, but want to see translated messages on your terminal, follow these instructions:

- For Japanese, issue **PROFILE PLANGUAGE(JPN)** at the TSO/E `READY` prompt on your DBCS terminal. This TSO/E command sets the primary language. The code JPN must match the LANGCODE statement in SYS1.PARMLIB(MMSLSTxx).
- For Simplified Chinese, issue **PROFILE PLANGUAGE(CHS)** at the TSO/E `READY` prompt on your DBCS terminal. The code CHS must match the LANGCODE statement in SYS1.PARMLIB(MMSLSTxx).

The TSO/E help panels must be set up separately.

Edit your SYS1.PARMLIB(IJKTSOxx) member in effect and ensure that the HELP statement refers to where the TSO/E help files are.

If you allocate a SYSHELP DDNAME in SYS1.PARMLIB, TSO/E searches there, rather than in the data sets pointed to by the HELP statement. For the format of the HELP statement, see *OS/390 TSO/E Command Reference*.

See the section "Specifying Help Data Sets" in the chapter "Customizing TSO/E for Different Languages" in *OS/390 TSO/E Customization* for more information on setting up help data sets.

Concatenating Target Libraries to ISPF

To use the Japanese translation of the panels, messages, and tables, you must concatenate the following target libraries to the appropriate ISPF data definition names (ddnames):

- SYS1.SBPXPJPN to ISPPLIB
- SYS1.SBPXMJPN to ISPMLIB
- SYS1.SBPXTJPN to ISPTLIB
- SYS1.KHELP to SYSHELP

To use the Simplified Chinese translation, concatenate the following target libraries to the appropriate ISPF ddnames:

- SYS1.SBPXPCHS to ISPPLIB
- SYS1.SBPXMCHS to ISPMLIB
- SYS1.SBPXTCHS to ISPTLIB
- SYS1.PHELP to SYSHELP

Recommendations for Running the OMVS Command

The PROFILE PLANGUAGE setting in effect when the OMVS TSO/E command is first issued determines the language for all OMVS command messages not from TSO/E, until you exit OMVS and return to TSO/E.

If PROFILE PLANGUAGE(JPN) is specified, and later you go to TSO/E and enter PROFILE PLANGUAGE(ENU), most TSO/E messages appear in English—including TSO/E messages about the OMVS command.

However, any OMVS command message not from TSO/E (such as the help panels invoked from <PF1> or the FSUM23-prefix messages) appear in Japanese. In particular, the TSO/E prompt message "`OMVS - enter a TSO/E command`" still appears in Japanese but all other messages appear in English while you are in TSO/E.

Part 2. OS/390 UNIX System Services User's Guide

Chapter 4. An Introduction to the OS/390 Shells

There are two shells available for use on OS/390 UNIX System Services:

- The OS/390 shell
- The tcsh shell

The OS/390 shell is modeled after the UNIX System V shell with some of the features found in the KornShell. As implemented for OS/390 UNIX services, this shell conforms to POSIX standard 1003.2, which has been adopted as ISO/IEC International Standard 9945-2: 1992.

The tcsh shell is an enhanced but compatible version of csh, the Berkeley UNIX C shell. It is a command language interpreter usable as a login shell and as a shell script command processor.

Figure 4 shows how these shells fit into MVS.

Figure 4. How the shells fit into MVS

About Shells

A shell is a command interpreter that you use to:

- Invoke shell commands or utilities that request services from the system.
- Write shell scripts using the shell programming language.
- Run shell scripts and C-language programs interactively (in the foreground), in the background, or in batch.

Shell Commands and Utilities

Both the OS/390 shell and the tcsh shell provide commands and utilities that give the user an efficient way to request a range of services. In this book, the term *command* is used to include both a *command* (a directive to a shell to perform a specific task) and a *utility* (the name of a program callable by name from a shell).

Shell commands often have *options* (also known as *flags*) that you can specify, and they usually take an *argument*—such as the name of a file or directory. The format for specifying the command begins with the command name, then the option or options, and finally the argument, if any. For example:

```
ls -a myfiles
```

ls is the command name, **-a** is the option, and *myfiles* is the argument.

This book describes various commands you can use to perform certain tasks; most of these are shell commands, and some are TSO/E commands. Typically, this discussion highlights only certain functions of the command. For complete information about each command and all its options, always refer to *OS/390 UNIX System Services Command Reference* .

The Locale in the Shell

A *locale* specifies cultural and language characteristics of the OS/390 UNIX System Services environment for an application program. Locale affects collation, date and time conventions, numeric and monetary formats, program messages, yes and no prompts, and the hexadecimal encoding for the 13 “variant” characters whose encoding varies on different EBCDIC code pages.

The shell and utilities support a variety of locales.

Daemon Support

OS/390 UNIX System Services provides daemons, such as **cron**, a batch scheduler, and **inetd**, which handles **rlogin** requests.

- For information about each daemon that OS/390 UNIX System Services provides, see *OS/390 UNIX System Services Command Reference* .
- For information about the Outboard Communications Server (OCS) login monitor daemon, see *OS/390 UNIX System Services Communications Server Guide*.

Running an X-Window Application

If you are accessing the a shell from a workstation or X-terminal running an X-Window server, you can run an X-Window application from the shell. An X-Window application needs the TCP/IP address and display identifier for your workstation.

For more information on X-Window interfaces, see *TCP/IP for MVS: Programmer's Reference*.

The Shell User

There are two categories of shell user: *superuser* and *user*. The superuser can do anything a user can, but has special authority to perform certain additional tasks (such as mounting and unmounting a file system), and can access all OS/390 UNIX services and the files in the hierarchical file system.

Security

This book assumes that your system includes the RACF security product. Instead of RACF, your system could have an equivalent security product.

The system programmer defines a shell user by assigning the user an *OMVS user ID (UID)* and *group ID (GID)*. These are numeric values associated with a TSO/E user ID; they are set in the RACF user profile and group profile when a user is authorized to use OS/390 UNIX services. The system uses the UID and GID to identify the files that a user owns and the processes that a user runs. The UID identifies a user of OS/390 UNIX services. The GID is a unique number assigned to a group of related users.

As a user, you can control read, write, and execute access to your files by other users in your group or outside of your group, by setting the permission bits associated with the files.

Accessing the Shells — The Choices

User's settings are initially configured with the OS/390 shell as the default login shell. To display these settings, from TSO type:

```
LISTUSER USERNAME OMVS
```

This will display the user's RACF settings as follows:

```
 UID= 0000000012
 HOME= /shut/home/billyjc
 PROGRAM= /bin/sh
 CPUTIMEMAX= NONE
 ASSIZEMAX= NONE
 FILEPROCMAX= NONE
 PROCUSERMAX= NONE
 THREADSMAX= NONE
 MMAPAREAMAX= NONE
 READY
```

The PROGRAM line refers to the User's login shell. If it is /bin/sh, the login shell is set to the OS/390 shell. If it is /bin/tcsh, the login shell is the tcsh shell. To change a user's default login shell from the OS/390 shell to the tcsh shell, issue the following command:

```
ALTUSER USERNAME OMVS(PROGRAM('/bin/tcsh'))
```

To change a user's default login shell from the tcsh shell to the OS/390 shell, type:

```
ALTUSER USERNAME OMVS(PROGRAM('/bin/sh'))
```

Terminal Emulators

OS/390 provides several terminal emulators that you can use to access the shells:

- The TSO/E OMVS command, a 3270 terminal interface
- The **rlogin** command, an asynchronous terminal interface
- The **telnet** command, an asynchronous terminal interface

Additionally, with OS/390 Communication Servers support, you have the asynchronous terminal interface if you directly login to the OS/390 shells from a terminal attached to a serial port on a RISC System/6000 running AIX V4.1.

When selecting a terminal emulator, there are several key points to consider:

- **Code Page Conversion:** By default, OS/390 UNIX System Services operates in the POSIX locale (also known as the C locale) using code page IBM-1047, but it can operate in other locales, including doublebyte locales. Unless you change the locale in the shell so that the code page used by the shell matches the code page used by the workstation for the OS/390 UNIX session, a terminal emulator must perform some code page conversion. Mechanisms are provided to specify the conversion required for your situation:
  - The OMVS command has the CONVERT parameter to specify the conversion between the code page used at your workstation and the code page used in the shell.
  - **rlogin** and **telnet** convert from ASCII ISO8859-1 to EBCDIC IBM-1047 by default. Once you are logged in to the shell, you can use the **chcp** to select other code pages to convert between for the session.

- **Number of Sessions:** Some terminal emulators allow multiple interactive sessions for the same user. This can be accomplished by multiple logins or by using an emulator that allows multiple sessions with one login.
- **File Editing:** With the OMVS emulator, you can use the ISPF editor. For the other terminal emulators, **vi** is the editor of choice.
- **Shell Mode: rlogin** and **telnet** provide both line mode (also known as canonical mode) and raw mode, while OMVS operates in line mode only. Line mode is sufficient for most shell utilities. However, the full function of certain useful utilities, such as **vi** and the command line editing feature of the shell, are available only in raw mode.

  When you first login to the shell, you are in line mode. Depending on your means of access, you may then be able to use utilities that require raw mode or run an X-Window application.

line mode	Your input is processed after you press <Enter>.
raw mode	Each character is processed as you type it.
graphical mode	A graphical user interface for X-Window applications

Figure 5. The OMVS Interface to the Shell

Figure 6. The Asynchronous Terminal Interface to the Shell

Interoperability between the Shells and MVS

Figure 7. OS/390 UNIX System Services Provides the User Interfaces of Both MVS and UNIX

There is a high degree of interoperability between MVS and the OS/390 shells:

- You can move data between MVS data sets and the *hierarchical file system (HFS)*: You can copy or move MVS data sets into the file system; likewise, you can copy or move HFS files into MVS data sets.
- To work with the HFS, you can use TSO/E commands or shell commands. If you have access to ISPF, you can use the panel interface of the ISPF shell. For example, you can create a directory with the TSO/E MKDIR command, or the shell **mkdir** command, or the ISPF shell.
- You can issue TSO/E commands from the shell command line, from a shell script, or from a program.
- You can write MVS job control language (JCL) that includes shell commands.
- To edit HFS files, you can use the ISPF/PDF full-screen editor or one of the editors available in the shell.
- OS/390 UNIX extensions to the Restructured Extended Executor (REXX) language enable REXX programs to access kernel callable services. You can run REXX programs using these extensions from TSO/E, batch, the shell, or a C program.

- You can use the OSHELL REXX exec to run a non-interactive shell command or shell script from the TSO/E `READY` prompt and display the output to your terminal. This exec is shipped with OS/390 UNIX services.

Parallels between the MVS Environment and the Shell Environment

Figure 8 indicates how basic programming tasks are performed in the MVS environment and in the shell environment.

An interactive user who uses the OMVS command to access the shell can switch back and forth between the shell and TSO/E, the interactive interface to MVS.

- Programmers whose primary interactive computing environment is a UNIX or AIX workstation find the shell programming environment familiar.
- Programmers whose primary interactive computing environment is TSO/E and ISPF can do much of their work in that environment.

Figure 8. Working Interactively in the MVS and Shell Environments

Programming for Everyday Tasks

The shell programming environment with its shell scripts provides function similar to the TSO/E environment with its command lists (CLISTs) and the *REstructured eXtended eXecutor (REXX)* execs.

The *CLIST language* is a high-level interpreter language that lets you work efficiently with TSO/E. A *CLIST* is a program, or command procedure, that performs a given task or group of tasks. CLISTs can handle any number of tasks, from running multiple TSO/E commands to running programs written in other languages. CLISTs can run only in a TSO/E environment. For a discussion of CLISTs, see *OS/390 TSO/E CLISTs*.

The *REXX language* is a high-level interpreter language that enables you to write programs in a clear and structured way. You can use the REXX language to write programs called *REXX programs,* or *REXX execs,* that perform given tasks or groups of tasks. REXX programs can run in any MVS address space. You can run REXX programs that call OS/390 UNIX services in TSO/E, batch, in the shell environment, or from a C program. For more information about writing REXX programs, see *OS/390 TSO/E REXX User's Guide*, *OS/390 TSO/E REXX Reference*, and *OS/390 Using REXX and OS/390 UNIX System Services*.

In the shells, command processing is similar to command processing for CLISTs. You can write executable *shell scripts* (a sequence of shell commands stored in a text file) to perform many programming tasks. They can run in any dubbed MVS address space. They can be run interactively, using **cron**, or using BPXBATCH. With its commands and utilities, the shell provides a rich programming environment.

Editing

In MVS, you edit the hierarchical file system (HFS) files by using the TSO/E OEDIT command to invoke ISPF File Edit or by selecting File Edit on the ISPF menu, if installed.

In a shell, you can use the **ed** and **sed** editors for editing HFS files. You can use the **oedit** shell command to invoke ISPF File Edit. If you use **rlogin** or **telnet** to login to the shell, you can also use the **vi** editor.

Job Control

In MVS, you can use the System Display and Search Facility (SDSF) to monitor and control a job. You can also use the TSO/E CANCEL, STATUS, and OUTPUT commands.

In the shell, you use the **ps** command or the **jobs** command to check the status of a job, and the **kill** command to end a job before it completes.

Additionally, in the shell you can stop, or suspend, a foreground job, and then enter the **bg** command to run it in the background or the **fg** command to start it back up in the foreground.

Background Jobs

In MVS, you write a background job in job control language (JCL) and start it with the TSO/E SUBMIT command.

In the shell, you start a background job by typing an ampersand (**&**) at the end of the command line.

Programming

In MVS, you use the OS/390 c/c++ compiler and the linkage editor to create a traditional OS/390 c/c++ application program as a load module or to create an OS/390 c/c++ application program as an executable file or a load module.

In the shell, you can use the **c89** or **cc** or **c++** command to compile and link-edit an OS/390 UNIX program, creating an executable file. The **make** command is available for building applications, and **lex** and **yacc** are available for developing applications.

Debugging

Under TSO/E, for traditional OS/390 c/c++ application programs, TSO/E Test and Inspect facilities are available for debugging. You can use TSO/E TEST for OS/390 UNIX application programs that do not use **fork()** or **exec()**.

In the shell, **dbx** is the debugging facility for OS/390 c/c++ programs. With **dbx**, you can debug multithreaded applications at the C-source level or at the machine level. Support for multithreaded applications gives you the ability to:

- Debug or display information about the following objects related to multithreaded applications: threads, mutexes, and condition variables.
- Control program execution by holding and releasing individual threads

The **dbx** debugger provides support for recognizing, displaying, and modifying program variables and constants that include doublebyte character set (DBCS) characters.

Data Management

In MVS, the storage administrator uses Data Facility System-Managed Storage Hierarchical Storage Manager (DFSMShsm) to automatically back up and archive hierarchical file systems.

In the shell, you can use **tar**, **cpio**, and **pax** to read or write an archive file in the file system.

You can copy archive files to an MVS data set, and then to tape. You can retrieve archive files from a tape into an MVS data set and then copy them into the file system.

Chapter 5. The Asynchronous Terminal Interface to the Shells

For people who have worked with UNIX systems, the asynchronous terminal interface is quite familiar. You use the asynchronous terminal interface if you access the OS/390 shells with one of these methods:

- **rlogin**
- **telnet**
- **rlogin** or **telnet** via the Communications Server
- Communications Server login from a serially attached terminal

ASCII-EBCDIC Translation

When you use **rlogin**, **telnet**, or Communications Server to access the shell, the data you enter is translated from ASCII (ISO8859-1) to EBCDIC (IBM-1047) before the shell processes it. To change code pages for the current session, use the **chcp** command. To automatically change code pages after you login, see “Changing the Locale in the Shell” on page 45.

For a complete list of the singlebyte and doublebyte ASCII and EBCDIC code pages that you can specify, see the *OS/390 C/C++ Programming Guide*.

Using rlogin to Access the Shell

When the **inetd** daemon is set up and active, you can **rlogin** to a shell from a workstation that has **rlogin** client support and is connected via TCP/IP or Communications Server to the MVS system. To login, use the **rlogin** command syntax supported at your site.

To improve performance when you **rlogin** into a shell, you can use shared address space.

Note: If you are writing or porting an **rlogin** command to rlogin into a shell, the shell interface to **rlogin** consists of the FOMTLINP and FOMTLOUT modules, documented in *OS/390 UNIX System Services Planning*.

Using telnet to Access the Shell

You can **telnet** to the shell from a workstation that is connected via TCP/IP or Communications Server to the MVS system. Use the **telnet** command syntax supported at your site.

Using Communications Server login to Access the Shell

If you are working at a terminal that is serially attached to the Communications Server, you can login directly to the shell.

1. Specify the host you want to login to. You receive a message confirming that you are connecting to the host.
2. At the prompts, enter your user ID and password.

The Shell Session

Once your login completes, you see your normal shell prompt (for example, $ or >). This is a UNIX interface, not the 3270-type interface that is displayed by the OMVS command. By default, the terminal interface is in line mode (also known as canonical mode), which means that each time you type a command at the prompt, you need to press Enter to process the command. Some utilities switch the terminal interface to raw mode. When you use a raw mode utility (such as **vi** or **talk**), or when command line editing is enabled in the shell, each keystroke is transmitted; you do not need to press <Enter>.

When you are in a shell session, you can:

- Run all shell commands and utilities.
- Run any application from the hierarchical file system (HFS).
- Use the **vi editor** and other full-screen applications such as **talk** and **more**.

In the OS/390 UNIX environment, the asynchronous terminal interface session has some differences from an OMVS session:

1. You cannot switch to TSO/E. However, you can use the **tso** shell command to run a TSO/E command from your session.
2. You cannot use the ISPF editor. (This includes the **oedit** and TSO/E OEDIT commands, which invoke ISPF File Edit.)

Entering a Shell Command

You type shell commands and press <Enter> to pass them to the shell.

If you are typing a long command that will not fit on one line, you can use the \ (backslash) continuation character at the end of the first line. When you then press <Enter>, the line is cleared so that you can continue typing. The line you typed prior to the backslash is displayed in the output area, and beneath it the shell prompt changes to > (? in tcsh) to indicate that you are continuing a command.

Interrupting a Shell Command

If you want to interrupt a command and stop it from completing, type <Ctrl-C>. The command stops executing and the system displays the shell prompt. You can now enter another command.

Using Multiple Sessions

With **rlogin**, **telnet**, or Communications Server, you can login to a shell more than once, using the same user ID and password. You can also be logged in to a shell using the OMVS 3270 interface and the asynchronous terminal interface at the same time, using the same user ID and password.

Using a Doublebyte Character Set (DBCS)

If you want to display or enter doublebyte data:

- You must work at a terminal that is configured to generate data in code page IBM-939 and follow the procedures for the terminal emulator being used, if any.
- Customize your locale and use the **chcp** command to specify the ASCII and EBCDIC code pages you are using.
  - For information on how to customize your locale and configure your setup files, see “Changing the Locale in the Shell” on page 45.

When you are working with a doublebyte character set, there are some restrictions.

telnet from the Shell

There are no **telnet** commands shipped with OS/390.

rlogin is also not shipped with OS/390.

Standard Shell Escape Characters

The following are some of the standard shell escape characters:

- <Ctrl-C> — Program interruption
- <Ctrl-D> — End of file
- <Ctrl-V> — Quit Program
- <Ctrl-Z> — Suspend Program

Chapter 6. Customizing the tcsh Shell

If you are interested in using the tcsh shell, read this chapter and Chapter 8, “Writing tcsh Shell Scripts” on page 79.

You can personalize your use of the tcsh shell. This chapter covers these topics:

- Understanding and modifying your startup files
- Understanding shell variables
- Customizing the search path for commands with the **PATH** variable
- Improving the performance of shell scripts
- Changing the locale
- Customizing the language of messages
- Setting the time zone
- Building a STEPLIB environment
- Setting options for a shell session

Understanding the Startup Files

When you start the tcsh shell, it uses information in several files to determine your particular needs or preferences as a user. The files are accessed in the following order:

1. **/etc/csh.cshrc**
2. **/etc/csh.login**
3. **$HOME/.tcshrc**
4. **$HOME/.cshrc**
5. **$HOME/.history**
6. **$HOME/.login**
7. **$HOME/.cshdirs**

Settings established in a file accessed earlier can be overwritten by the settings in a file accessed later.

The **/etc/csh.cshrc** file contains systemwide settings that are common to all shell users. It is used for setting shell variables and defining command aliases. Usually, it will set environment variables such as PATH.

The **/etc/csh.login** file is a systemwide file that is only executed by tcsh login shells, and is used for setting environment variables such as TERM. Opening messages are typically placed here.

The **/$HOME/.tcshrc** file contains settings that may be customized for an individual shell user. It is used for setting shell variables and defining command aliases. Here, users can set variables that are different than the system defaults set in the systemwide profiles.

The **/$HOME/.cshrc** file is included for compatiblity with C Shell users, and is read only if **/$HOME/.tcshrc** does not exist. It contains the same type of settings as **/$HOME/.tcshrc**.

The /$HOME/.history file is read by login shells to initalize the history list. It is created by the shell, based on the setting of certain shell variables.

The **/$HOME/.login** file is only executed by tcsh login shells, and is used for setting environment variables that have been customized for an individual user. It usually contains commands that affect a user's terminal settings.

Typically, your **.login** file might contain the following:

```
# set TERM environment variable
setenv TERM vt220

# set DISPLAY environment variable
setenv DISPLAY mymachine.mydomain.com:0
```

Figure 9. A Sample .login

The **$HOME/.cshdirs** file is read by login shells to initalize the directory stack. It is created by the shell, based on the setting of certain shell variables.

The systemwide startup files (located in /etc) are modified by system administrators to contain settings that should pertain to all users. The startup files in a user's home directory ($HOME/. . .) can be altered to suit specific user preferences, with the exception of **$HOME/.history** and **$HOME/.cshdirs**, which are created by the shell. A user can "unset" or "unalias" anything that was defined in a systemwide startup file.

Quoting Variable Values

When you have blanks in a variable value, you need to enclose it in quotes. The quotes tell the shell to treat blanks as literals and not delimiters. Single quotes are more “serious” about this than are double quotes:

- Single quotes preserve the meaning of (that is, treat literally) all characters.
- Double quotes still allow certain characters ($, ` (backquote), and \ (backslash)) to be expanded. This is important if you want variable expansion. For example, see how the $ is handled here:

  ```
  setenv HOMEMSG "Using $HOME as Home Directory"
  ```

  If your home directory were set to **/u/user**, the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using /u/user as home directory
  ```

  If, instead, you enclosed the variable value in single quotes, like this:

  ```
  setenv HOMEMSG 'Using $HOME as home directory'
  ```

  the following:

  ```
  echo $HOMEMSG
  ```

  would display:

  ```
  Using $HOME as home directory
  ```

  As you can see, the $ is not expanded.

Changing Variable Values Dynamically

You can also change any of these values for the duration of your session (or until you change them again). You enter the name of the environment or shell variable and equate it to a new value. For example:

```
set prompt='+>'
```

changes the command prompt string to +>.

Understanding Shell Variables

You can display the shell's variables and their values by entering this command:

```
set
```

or

```
set -r
```

set –r displays readonly shell variables.

You may see many variables that you don't recognize. These are *built-in*, or *predefined*, variables that are set up with default values when you start the shell.

You can customize the built-in variables by setting their value in your **.tcshrc** file.

Only the shell variables that are defined in the **.tcshrc** file are available to shell scripts and commands invoked from the shell. Environment variables are inherited by subshells, and can be displayed by entering either of these commands:

```
setenv
printenv
```

You can display the value of a single variable with the **echo** command or the **printenv** command. For example, either of these commands

```
echo $HOME
```

```
printenv $HOME
```

displays the current value of the **HOME** variable.

In general, **echo** displays the current values of all its arguments, after any shell processing has taken place. For example, consider:

```
echo *.doc
```

The shell first expands the wildcard character *****. This produces the names of every file in the working directory that has the suffix **.doc**. So the output of **echo** is a list of all such files. And if there are no filenames ending in **.doc**, the command output is just `*.doc`.

For more information about shell variables,

- Built-in variables are listed in a table in the **tcsh** command description in *OS/390 UNIX System Services Command Reference*.
- There is an appendix that lists shell variables in *OS/390 UNIX System Services Command Reference*.

Customizing Your Shell Environment: The .tcshrc File

So far, we have discussed customization that is set up inside your **.login** file. However, the shell reads this file only when you log into the shell or when you enter the **tcsh** command with the **–l** option. Note that the option is a lowercase "L".

To always have a customized shell session, you need to have a special shell script that customizes your shell variables each time you start the shell; this is the purpose of the **.tcshrc** file (also known as a startup script).

For example, you might put all your alias definitions and other setup instructions into this file. You want these instructions run when your shell starts after you login and whenever you explicitly create the shell during a session (for example, as a child shell to run a shell script).

Below is a sample **.tcshrc** file:

```
# ================================================================
#                       path shell variable
#                       -------------------
# Lists directories in which to look for executable commands.
# ================================================================
#set path = ( /bin /usr/local/bin /usr/bin )

# test if we are an interactive shell
if ($?prompt) then
# ================================================================
#                       prompt shell variable
#                       ---------------------
# The string which is printed before reading each command from the
# terminal.  Currently set to display hostname, and current working
# directory.
# ================================================================
set prompt = "%m:% > "

# ================================================================
#                       rmstar shell variable
#                       ---------------------
# If set, the user is prompted before 'rm *' is executed.
# ================================================================
set rmstar

# ================================================================
#                       noclobber shell variable
#                       ------------------------
# If set, output redirection will not overwrite existing files.
# ================================================================
#set noclobber

# ================================================================
# source complete.tcsh
# ================================================================
if ( filetest -e /etc/complete.tcsh ) then
        source /etc/complete.tcsh
endif
endif  # interactive shell

# ================================================================
# set up useful aliases
# ================================================================
alias m more
```

Figure 10. A Sample .tcshrc

Customizing the Search Path for Commands: The PATH Variable

Command interpreters usually have to *search* for a file that contains the command you want to run. When using the shell, you tell the shell where to search for a command. Essentially, the shell uses a list of directories in which commands may be found. This list is specified in your **PATH** variable in your **etc/csh.cshrc** file. The list could be called your *search path,* because it tells the shell where you want to search.

You can set up a search path with a command of the form:

```
setenv path 'dir:dir:...'
```

or,

```
set path=(dir1 dir2)
```

For example, you might enter:

```
setenv path '/bin:/usr/bin:/usr/macneil/bin:/usr/games:/usr'
```

The shell then searches the directories in the following order, when looking for commands or shell scripts:

1. **/bin**
2. **/usr/bin**
3. **/usr/macneil/bin**
4. **/usr/games**
5. **/usr**

As soon as the shell finds a file with an appropriate name, it runs that file.

Because the shell runs a command as soon as it finds a file with an appropriate name, pay close attention to the order in which you list directory names in your search path. For example, the previous search path specifies the **/bin** directory (where OS/390 shell commands are stored) before the **/usr/bin** directory.

If you set up your **PATH** incorrectly, you could get the wrong command. You should generally search the shell commands directory first: **/bin**.

Adding Your Working Directory to the Search Path

You can have the shell search your working directory for commands (in addition to the standard directories that contain commands). As an example, suppose you have different directories containing the source code for different programs. In each directory, you create a shell script named **compile** that compiles all the source modules of the program in that directory. To compile a particular program, enter **cd** to change to the appropriate directory and then enter:

```
compile
```

The shell searches the working directory, finds the **compile** shell script, and runs it.

You can add your working directory to your search path by one of these methods:

- Putting in an entry without a name
- Using a period (**.**) for the working directory.

For example, both of these specify that the working directory should be searched after **/bin** but before **/usr/local**:

```
setenv path '/bin::/usr/local'    #no name
setenv path '/bin:.:/usr/local'   #using a period
```

Both of these say that your working directory should be searched before anything else:

```
setenv path ':/bin:/usr/local'    #no name
setenv path '.:/bin:/usr/local'   #using a period
```

Both of these say that your working directory should be searched after everything else:

```
setenv path '/bin:/usr/local:'    #no name, ends in a colon
setenv path '/bin:/usr/local:.'   #using a period
```

The best way to specify search paths is to put them into your **.tcshrc** file. That way, they are set up every time you log into the shell.

Checking the Search Path Used for a Command

With aliases and search paths, it can be easy to lose track of what is actually executed when you enter a command. The **which** command can tell you which file is executed if you enter a command line that begins with a specific command. The **where** command can tell you where versions of the command are located. For example:

```
which kill
```

tells you:

```
kill: shell built-in command.
```

and the command:

```
where kill
```

tells you:

```
kill is a shell built-in
/bin/kill
```

Customizing the DLL Search Path: The LIBPATH Variable

If you use a utility that uses a dynamic link library (DLL) —for example, **dbx**— you can set up the search path for the DLL with the LIBPATH variable. If this variable is not set, your working directory is searched for the DLL. The default setting shipped in **/samples/login** is:

```
setenv LIBPATH "/lib:/usr/lib:."
```

Changing the Locale in the Shell

The default locale for the shell and utilities is C. If you want to change the locale, read the topics below.

For additional information on locale and **LC_SYNTAX**, see *OS/390 Language Environment Programming Guide*.

Advantages of a Locale Compatible with the MVS Code Page

Running the shell and utilities in a locale whose code page matches the code page you are using in MVS (which may not be compatible with code page IBM-1047 with respect to the EBCDIC variant characters) has several advantages:

- Converting data from a given country's native code page to IBM-1047 is no longer required. This may enhance interoperability with other non-OS/390 UNIX components of MVS.
- Remapping your keyboard is unnecessary.

Customizing for a Locale Not Based on Code Page IBM-1047

If you select a locale that is not based on code page IBM-1047 and you use the utilities **lex**, **mailx**, **make**, and **yacc**, there is a further customizing step. These utilities expect all their input files, both system files and user-created files, to be in the same code page. So, for example, if you select the German locale De_DE.IBM-273, these utilities expect the files they process to be in code page IBM-273. Because system files are in code page IBM-1047, you need to use **iconv** to convert the following system files to the code page used by your selected locale:

Utility	*File*
lex	**/etc/yylex.c**
mailx	**/etc/mailx.rc**
make	**/etc/startup.mk**
yacc	**/etc/yyparse.c**

Advantages of a Locale Generated with Code Page IBM-1047

On the other hand, you may prefer using one of the locales compatible with IBM-1047, but not compatible with the MVS code page if:

- You already use one of the IBM-1047 locales and have made an investment in data conversion and keyboard remapping.
- You have a requirement to run, in your shell environment, strictly standards-compliant applications or other applications that do *not* use **LC_SYNTAX**. If you want to use a single compiled and link-edited instance of a program in multiple locales, such a program is guaranteed to work in multiple locales only if IBM-1047 locales are used.
- You have shell scripts that are used in multiple locales. Having different users operating in various locales that are not generated from code page IBM-1047 requires multiple copies of a shell script, one for each different locale's code page.

There are other important code page conversion considerations when the shell uses code page 1047 and MVS does not.

Changing the Locale Setting in Your Profile

To change the locale, you set the value for the **LC_ALL** variable. This variable overrides any values for locale specified for the **LC_** variables such as **LC_COLLATE**, **LC_MESSAGES**, and **LC_SYNTAX**, but it does not override **LC_CTYPE**.

If you change **LC_ALL** to a new locale, and OS/390 UNIX messages are provided in that language, change the **LANG** variable setting to match the **LC_ALL** setting. Currently, OS/390 UNIX messages are shipped in English, Kanji, and Simplified Chinese. If you do not change **LANG**, the messages will be in English.

If OS/390 UNIX messages are not provided in your language then changing **LANG** by itself will have no effect. However, although messages are not supplied in your language, the OS/390 UNIX messages that are displayed in English will use your national language characters and should display correctly on your terminals.

When you change the locale, the shell and utilities run in the new locale, but the shell locale category **LC_CTYPE** stays in the POSIX locale. This can affect parsing and shell expansion, and cause unpredictable behavior. In order to avoid this problem, after you change locale you must overwrite the current shell by issuing the **exec tcsh -l** command. The new shell will correctly interpret the proper character set for the new locale.

If you place an `setenv LC_ALL` *`localename`* statement in your login profile, or if one has been placed in **/etc/csh.login**, make sure it is followed with **exec tcsh -l** and protect that with **tty -s** as shown in the example below. If you don't protect it with the **tty -s** test, then BPXBATCH SH *command* will not run the command.

If you use **exec tcsh -l**, there are two situations that you must take into account:

1. Loop control; you only want the **exec tcsh -l** executed the first time.
2. If you plan to use BPXBATCH or OSHELL (which calls BPXBATCH) with national language support, you need to define the LANG and LC_ALL variables in a file for BPXBATCH to use.

If your **/etc/csh.login** has been set up for the proper locale, you only need to change your .login if you want a different locale than already set up as the default. For more information on setting up locale and messages, see “Customizing for Your National Code Page” in *OS/390 UNIX System Services Planning*.

Examples: Changing Locale

For example, say you are using OMVS, the 3270 terminal interface. If your **/etc/csh.login** is not set up for your locale and LANG, then in order to work in a locale such as Danish, you should add this to your **.login** file:

```
tty -s
set tty_rc=$status
if (($?LOCALE_SWITCH == 0 ) && ($tty_rc == 0)) then
        echo "-------------------------------------------------"
        echo "- Logon shell will now be invoked to reflect   -"
        echo "- code page IBM-277                            -"
        echo "-------------------------------------------------"
        setenv LOCALE_SWITCH EXECUTED
        setenv LANG C
        setenv LC_ALL Da_DK.IBM-277
        # Issue chcp if not using OMVS command
        if ($?_BPX_TERMPATH != "OMVS"  ) then
          chcp -a ISO8859-1 -e IBM-277
        endif
        exec tcsh -l
endif
unset tty_rc
```

If you want your messages displayed in a different language than that specified in the system-wide **/etc/csh.login**, you have to modify your **.login** accordingly.

The LC_SYNTAX Environment Variable

There are 13 “variant” characters in the POSIX portable character set whose encoding may vary on various EBCDIC code pages:

Right brace (})
Left brace ({)
Backslash (\)
Right square bracket (])
Left square bracket ([)
Circumflex (^)
Tilde (~)
Exclamation point (!)
Pound sign (#)
Vertical bar (|)
Dollar sign ($)
Commercial at-sign (@)
Accent grave (`)

When you specify a locale with the **LC_ALL** variable, the **LC_SYNTAX** environment variable is set. The shell uses the **LC_SYNTAX** environment variable to determine the code points to use for the 13 variant characters. This means the shell can dynamically adapt to the code page of the current locale.

Applications that use **LC_SYNTAX** will work in multiple locales using multiple code pages. To be sensitive to the 13 variant characters, an application must be enabled to use **LC_SYNTAX**. For information on how to do this, see *OS/390 C/C++ Programming Guide*.

LC_SYNTAX—An Example

For example, consider the **echo** command and its use of the backslash (\) character. The backslash is one of the 13 variant characters. When the echo style is **all** or **sysv**, the following command:

```
echo 'this is\nreal handy'
```

produces the following output at the terminal:

```
this is
real handy
```

echo finds and converts the `\n` in the input to a <newline> character in the output. To do this, **echo** must know the encoding for the backslash character in the current user's environment—in this case, the character generated by the user's terminal when the backslash key is pressed.

A 3270 terminal operating in the USA locale En_US.IBM-037 (code page IBM-037) generates X'E0' for the backslash, while a 3270 terminal operating in the German locale De_DE.IBM-273 (code page IBM-273) generates X'EC'. The **LC_SYNTAX** locale category provides this locale-specific hexadecimal encoding information to **echo** and the other utilities.

When the USA user runs in locale En_US.IBM-037, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'E0'. Likewise, when the German user runs in locale De_DE.IBM-273, **echo** determines from the LC_SYNTAX information in this locale that the expected encoding for backslash is X'EC'.

Limitations

The **LC_SYNTAX** setting does not affect:

- REXX execs.
- The ISPF shell (ISHELL). ISHELL runs in the locale that MVS is using, and therefore this could be different from the shell locale.
- Shell scripts: The code page in which a shell script is encoded must match the code page of the locale in which it is run. For a shell script to be shared by multiple users, they must all be in a locale that uses the same code page as the code page in which the shell script is encoded.

  If you have different users operating in various locales, you need multiple copies of a shell script, one for each different locale code page. You can use the **iconv** command to convert a shell script from one code page to another.

The LOCPATH Environment Variable

LOCPATH is an environment variable that tells the **setlocale()** function the name of the directory from which to load locale object files. If LOCPATH is not defined, the default directory **/usr/lib/nls/locale** is searched. LOCPATH is similar to the PATH environment variable; it contains a list of HFS directories separated by colons. For detailed information on how **setlocale()** searches for locale object files, see the description of **setlocale()** in *OS/390 C/C++ Run-Time Library Reference*.

Customizing the Language of Your Messages

If you want your messages displayed in a different language than that specified in the system-wide **/etc/.login**, add this line to your **.login**:

```
setenv LANG your_language
```

your_language is the first part of the locale name—for example, Ja_JP in the locale name JA_JP.IBM-939. Currently, OS/390 UNIX ships messages in English, Kanji and Simplified Chinese.

Setting Your Local Time Zone

The shell and utilities assume that the times stored in the file system and returned by the operating system are stored using the Greenwich Mean Time (GMT) or Universal Time Coordinated (UTC) as a universal reference. In the system-wide **/etc/csh.login**, the **TZ** environment variable maps that reference time to the local time specified with the variable. You can use a different time zone by setting the **TZ** variable in your **.login**.

The three primary fields in the time zone specification are:

1. The local standard time, abbreviated—for example, EST or MSEZ.
2. The time offset west from the universal reference time, typically specified in hours (minutes and seconds are optional). A minus sign (–) indicates an offset east of the universal reference time.
3. The daylight savings time zone, abbreviated—for example, EDT. If this and the first field are identical or this value is missing, daylight savings time conversion is disabled. Optionally, you can specify an additional rule that indicates when Daylight Savings Time starts and ends.

For example, if you want to set your time zone to Eastern Standard Time (EST)and export it, you would specify:

```
setenv TZ "EST5EDT"
```

- EST is Eastern Standard Time, the local time zone.
- The standard time zone is 5 hours west of the universal reference time.
- EDT is Eastern Daylight Savings time zone.

For complete information on how to specify the local time zone, see *OS/390 UNIX System Services Command Reference*.

Building a STEPLIB Environment: The STEPLIB Environment Variable

Traditionally, some MVS users have preferred to alter the search order for MVS executable files when they are running a new or test version of an application program, such as a runtime library. To do this, they code a STEPLIB DD statement on the JCL used to run the application. Accessed ahead of LINKLIB or LPALIB, a STEPLIB is a set of private libraries where the new or test version of the application is stored.

The STEPLIB environment variable provides the ability to use a STEPLIB when running an HFS executable file. This variable is used to determine how to set up the STEPLIB environment for an executable file.

You can set the variable in one of three ways:

`setenv STEPLIB CURRENT` Passes on any currently active TASKLIB, STEPLIB, or JOBLIB allocations from the invoker's MVS program search order environment to the environment created for the executable file to run in. Any STEPLIB environment in the invoker's process image is re-created in the new process image for the executable file when the file is invoked. This is the default value that is set if no STEPLIB variable is specified.

If an application uses **fork()**, **spawn()**, or **exec()**, then the STEPLIB data sets must be cataloged.

`setenv STEPLIB NONE` Specifies that no STEPLIB environment should be set up for executable files.

`setenv STEPLIB DSN1:DSN2:DSN3` Sets up a library search order for the STEPLIB, in the order that the data sets are specified. You can specify up to 255 fully qualified data set names, separated by colons—for example:

```
setenv STEPLIB SMITH.C.LOADLIB:SMITH.PL1.LOADLIB
```

The specified data sets must be cataloged MVS load libraries that you have security access to. The data sets specified here are built into a STEPLIB environment for the executable file.

Restrictions on STEPLIB Data Sets

For executable files that have the set-user-ID or set-group-ID bit set, there are restrictions on the data sets that can be built into the STEPLIB environment for the file to run in. The system programmer maintains a STEPLIB sanction list of data sets that can be included in the STEPLIB environment for such executable files. Only data sets on that list are built into the STEPLIB environment for such files. If you need a data set added to the list, contact your system programmer. For more information on the STEPLIB sanction list, see *OS/390 UNIX System Services Planning*.

Setting Variables for a Shell Session

The **set** and **unset** commands let you set and unset variables for your shell session. These variables control the way the shell handles certain situations. To display the shell variables that are currently set, type `set`. To turn an option on, enter:

```
set name
```

where *name* is the name of the option you want to turn on. If you want an option turned on for every shell session, put the **set** command in your **.tschrc** file.

To turn an option off, enter:

```
unset name
```

The following discussion highlights some of the options you may find useful. For all the options, see *set in the tcsh shell* under **set** in *OS/390 UNIX System Services Command Reference*.

Displaying Current Option Settings

The command:

```
set
```

displays all current option settings.

Controlling Redirection

The command:

```
set noclobber
```

indicates that you do not want the > redirection operator to overwrite existing files. When this option is on and you specify the construct >*file*, the redirection works only if *file* does not already exist. If you have this option on and you really do want to redirect output into an existing file, you must use >**|***file* (with an "or" bar after the >) to indicate output redirection.

Preventing Wildcard Character Expansion

The command:

```
set noglob
```

tells the shell not to expand wildcard characters in filenames. This command is occasionally useful if you are entering command lines that contain a number of characters that would normally be expanded.

Displaying Input from a File

The command:

```
set xtrace
```

tells the shell to display its input on the screen as the input is read. This command lets you keep track of material that comes from a file.

Displaying Deletion Verification

The command:

```
set rmstar
```

prompts you for deletion verification when you enter the **rm** command in conjunction with the * character.

Files Accessed at Termination

When you terminate the tcsh shell, the following files are read at logout in this order:

1. **/etc/csh.logout**
2. **$HOME/.logout**

Chapter 7. Working with tcsh Shell Commands

The shell is, above all, a *programmer's* interface. As a result, the shell commands are strongly slanted towards the needs of a programmer. The tcsh shell has many *general* tools that can help any programmer, and is specifically designed to have syntax similar to the C programming language. In addition, there are a number of commands designed especially for the C programmer.

Specifying Shell Command Options

Most of the commands discussed in this chapter accept options. Shell command options are usually specified by a minus sign (–) followed by a single character. For example, the **ls** command simply lists a directory's contents in multiple columns on your screen. However:

```
ls –F
```

distinguishes between various file types when listing the contents of a directory.

```
ls –1
```

lists directory names in a single column.

Options consisting of a minus sign followed by a character are called *simple options*. You specify simple options after the name of the command and before any other arguments for the command (that is, arguments that are not options). For example, you would enter:

```
ls –1 dir1
```

to list the contents of **dir1** in a single column.

Command options and arguments must be typed as singlebyte characters. Additionally, delimiters such as a slash, curly brackets, and parentheses must be typed as singlebyte characters.

The order of options and arguments is important. If you enter:

```
ls dir1 –F
```

ls lists the contents of **dir1** and then tries to list the contents of the directory, or attributes of the file, called **–F**.

As a special notation, most tcsh shell commands let you specify a double minus sign (--) to separate the options from the nonoption arguments; `--` means that there are no more options. Thus, if you really have a directory named **–F**, you could enter:

```
ls -- –F
```

to list the contents of that directory or the file attributes.

The tcsh shell gives you a shorthand way to specify more than one simple option to a command. For example, **–t** and **–v** are both simple options that you can specify with the **cat** command. (To find out what these options do, read the description of **cat** in *OS/390 UNIX System Services Command Reference*.) You could enter:

```
cat –t –v file
```

or you could combine the two options into:

```
cat –tv file
```

The order of the options is not important:

```
cat –vt file
```

is equivalent to the previous version of the command.

Specifying Options with Accompanying Arguments

In addition to simple options, some commands accept options that have accompanying arguments. Such options look like simple options followed by additional information. The argument may be a number, a string, the name of a file, or something else.

For example, if you read the description of **ps** in *OS/390 UNIX System Services Command Reference*, you will see that **ps** accepts an argument of the form:

```
–u userlist
```

When *OS/390 UNIX System Services Command Reference* shows part of a command line in *italics*, the italicized material is just a placeholder; when you actually use the command, you should fill in something else in its place. In this case, the *userlist* should be a string of one or more UID numbers or login names separated by commas and enclosed in single quotes. In the command:

```
ps –u 'macneil,wellie1'
```

the *userlist* string is macneil,wellie1. (If the string does not contain spaces, tabs, or other special characters, you can actually omit the enclosing single quotes, but the command is often easier to read if you use quotes anyway.) When executed, **ps** displays information for the specified users.

Help for Shell Command Usage

If you incorrectly specify a command, a usage note for the command is displayed. The usage note displays the proper format for the command. Often you can display a usage note deliberately if you specify the command with a *-?* option.

For online help information about a command, see “Online Help” on page 74.

Understanding Standard Input, Standard Output, and Standard Error

Once a command begins running, it has access to three files:

1. It reads from its *standard input* file. By default, standard input is the keyboard.
2. It writes to its *standard output* file.
   - If you invoke a shell command from the shell, a C program, or a REXX program invoked from TSO READY, standard output is directed to your terminal screen by default.
   - If you invoke a shell command, REXX program, or C program from the ISPF shell, standard output cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.
3. It writes error messages to its *standard error* file.

- If you invoke a shell command from the shell or from a C program or from a REXX program invoked from TSO READY, standard error is directed to your terminal screen by default.
- If you invoke a shell command, REXX program, or C program from the ISPF shell, standard error cannot be directed to your terminal screen. You can specify an HFS file or use the default, a temporary file.

  If the standard output or standard error file contains any data when the command completes, the file is displayed for you to browse.

Using the Shell:

In the shell, the names for these files are:

- **stdin** for the *standard input* file.
- **stdout** for the *standard output* file.
- **stderr** for the *standard error* file.

Using TSO/E:

When you are invoking the BPXBATCH utility, you can specify these standard files in MVS DD statements, TSO/E ALLOCATE commands, or DYNALLOC macros using the ddnames:

- STDIN for standard input
- STDOUT for standard output
- STDERR for standard error

Using ISPF:

When you run shell commands, REXX programs, and C programs from the ISPF shell, **stdout**, and **stderr** cannot be directed to your terminal. You can specify an HFS file, or use the default—a temporary file. If it has any contents, the file is displayed for you to browse when the command or program completes.

Redirecting Command Output to a File

Commands entered at the command line typically use the three standard files described in the previous section, but you can redirect the output for a command to a file you name. If you redirect output to a file that does not already exist, the system creates the file automatically.

Most shell commands display information on your workstation screen, *standard output*. If you redirect the output, you can save the output from a command in a file instead. The output is sent to the file rather than to the screen. At the end of any command, enter:

```
>filename
```

For example:

```
cat file1 file2 file3 >outfile
```

writes the contents of the three files into another file called **outfile**. All the information in the original three files is concatenated into a single file, **outfile**.

When you redirect output with >*filename* and it is an existing file, the output writes over any information that the file already contains. To *append* command output at the end of the file, use:

```
>>filename
```

instead.

Another example:

```
(sort -u file1 >output) >&outerr
```

redirects the result of the sort to the file named **output** (instead of standard output) and redirects any error messages to the file **outerr**, which is a record of errors encountered during various sorts.

Suppose you entered:

```
sort -u filea >output
```

In this command, you see two redirections:

- Error output from the sort is redirected to standard output, the display screen.
- The result of the sort is redirected to the file named **output**.

Here is another example of redirection, sending both standard error and standard output to a file. This command produces the program **hello** and a listing with error messages in a file called **hello.list**:

```
c89 -o hello -V hello.c >&hello.list
```

Redirecting Input from a File

You can redirect input in much the same way that you redirect output. A command that normally takes input from standard input can be redirected to take input from a file instead. For example, with this **mailx** command, you can send the file **lessons** to another user.

```
mailx JAYD <lessons
```

The file **lessons** becomes input to **mailx**, rather than your input from the keyboard.

Redirecting Error Output to a File

You can redirect error output from the workstation screen to a file. For example:

```
(sort -u filea >dev/tty) >& outerr
```

sorts **filea**, checking for unique output records. Any messages regarding duplicate records are redirected to a file named **outerr**.

And if you do not care about seeing the error output, you can just redirect it to **/dev/null**, also known as the “bit bucket.” This is equivalent to discarding the error messages.

```
(sort -u filea >/dev/tty) >& /dev/null
```

Dumping Nontext Files to Standard Output

The **od** command can dump the contents of a file to *standard output,* your workstation screen, in several different formats.

```
od file
```

dumps a file in octal.

```
od -h file
```

dumps the file in hexadecimal. Either of these may be useful if you want to check the actual contents of a nontext file. Other dump formats are available.

Setting Up an Alias for a Command

After you have used the shell for a while, you will probably find that there are some commands that you use frequently. Rather than typing them over and over, you can set up an *alias* for these commands. An alias is a personalized name that stands for all or part of a command. You can create an alias by entering:

```
alias name "string"
```

in response to the shell's usual prompt for input. This is not a normal command; it is an instruction to the shell itself.

For example, suppose you have a hard time remembering that the **mv** command actually renames files. To make life easier for yourself, you could set up a simple alias by entering this on your command line:

```
alias renam "mv"
```

From this point onward in your session, whenever the shell sees the command **renam**, the **renam** is replaced with **mv**. The alias facility lets you create more usable commands.

Clearly, you could use an alias to save yourself some typing too. You could define **c** as an alias for **cat**. Then you would enter:

```
c file
```

to get the effect of:

```
cat file
```

Defining an Alias

If you will be using an alias frequently, put the **alias** command in your profile file (**$HOME/.tcshrc**). That way, you do not have to type them in every time you start using the shell. See “Understanding the Startup Files” on page 39 for more information about customizing your startup files.

To display all the currently defined aliases, you just enter:

```
alias
```

and the shell displays them.

Arguments in Aliases

Any arguments that follow an alias are treated just as if they had been following the command that the alias stands for. For example, if you define the alias **f** as follows:

```
alias f "ls"
```

the shell replaces **f** with **ls**, which is the command to list files in a directory.

You can refer to arguments in an alias by simply adding them at the end of the alias as you would with a command. For example:

```
f -la
```

would perform the **ls** command with the arguments **la**, which will list all the files in the directory in a long directory listing format. And,

```
f /bin
```

will list the contents of the **/bin** directory.

Redefining an Alias for a Session

You can redefine an alias during a session, even if it is defined in your profile file. If you enter the command:

```
alias name "string"
```

during a session and *name* is already an alias, the shell forgets the old meaning and uses the new meaning from then on.

Setting Up an Alias for a Particular Version of a Command

If you tend to use a command with the same options every time, you may want to set up an alias for the command with those particular options. Let's take an example. The **grep** command searches through files and prints out lines that contain a requested string. For example:

```
grep hello file
```

displays all the lines of *file* that contain the string `hello`. Normally, **grep** distinguishes between uppercase and lowercase letters; this means, for example, that the search in the previous example does *not* display lines that contained `HELLO`, `Hello`, and so forth. If you want **grep** to ignore the case of letters as it searches, you must specify the **–i** option, as in:

```
grep -i hello file
```

This finds `hello`, `HELLO`, `Hello`, and so on.

If you think you prefer to use the **–i** version of **grep** most of the time, you can define the alias:

```
alias grep "grep -i"
```

From this point on, if you use the command:

```
grep string file
```

it is automatically converted to:

```
grep -i string file
```

and you get the case-insensitive version of the command **grep**.

As another example, the **rm** command to delete (remove) a file has an **–i** option that prompts you to confirm the deletion. The filename and a question mark are

displayed. For example, if you entered `rm -i file1` and **file1** is in your working directory, you would see the prompt:

```
file1: ?
```

before the system actually removes the file. You then enter `y` (yes) or `n` (no) in response. If you like this extra bit of safety, you might define:

```
alias rm "rm -i"
```

After this, when you call **rm**, it automatically checks with you before deleting a file, just to make sure that you really want to delete it.

It may seem odd to define an alias that has the same name as a command that is used in the alias, but this is so common that the shell checks specially for an alias of the same name, and does the correct thing.

If you find yourself using the same option every time you call a command, you might consider creating an appropriate alias so that the shell automatically adds the option. Of course, the best place to define this alias is in your **.tcshrc** file; then the alias is set up every time you invoke the shell.

Turning Off an Alias

If you have set up an alias like the one previously described for **rm**, you may find that you *do not* want the alias to apply in some situations. For example, when you delete a huge number of files, you probably do not want **rm** to ask if it is okay to delete each one. In this situation, you have several options:

- Get rid of the alias entirely. The command:

  ```
  unalias rm
  ```

  gets rid of the **rm** alias for the session. After this, when you enter **rm**, you get the real **rm** command.

- Escape the alias. If you put a backslash in front of an alias, the shell uses the real command rather than the alias. For example:

  \rm *file*

- Specify the full pathname. For example:

  /bin/rm *file*

  tells the shell to run the program in **/bin/rm**. The shell does not perform alias substitution when you specify a command as a pathname.

These alternatives should help you get around options that you have automatically associated with a command.

Combining Commands

There are several simple ways you can combine several commands on a single command line.

- You can run a series of commands, one after the other:

  Using a semicolon (**;**)
  Using **&&** and ||

- You can run more than one command concurrently:

Using a pipe (**|**) or a filter with a pipe

The output from the first command is piped to the next command as the first command is running.

Using a Semicolon (;)

The shell lets you enter several commands on the same command line. To do this, just use the semicolon character to separate the commands; for example:

```
cd mydir ; ls
```

Also, if you have defined the alias:

```
alias l "ls –l"
```

you can enter:

```
cd mydir ; l
```

since you can use aliases such as **l** after a semicolon.

Using && and ||

When stringing together more than two commands, you may want to control the running of the second command based on the outcome of the first command. You can use:

&& If the command that precedes && completes successfully, the command following && is run. Leave a space on either side of the && operator: `command && command`.

|| If the command that precedes **||** fails, the command following **||** is run. Leave a space on either side of the **||** operator: `command || command`.

Using a Pipe

The output from one command can be *piped in* as input to the next command. Two or more commands linked by a pipe (|) are called a *pipeline*. A pipeline is written as:

command | *command* | **...**

You enter the commands on the same line and separate them by the “or-bar” character |.

Many commands are well suited to being used in a pipeline. For example, the **grep** command searches for a particular string in input from a file or standard input (the keyboard). A command such as:

```
history | grep "cp"
```

displays all the **cp** commands recorded among the 16 most recently recorded commands in your history file. The command:

```
ls –l | grep "Jan"
```

uses **ls** to obtain information on the contents of the working directory and uses **grep** to search through this information and display only the lines that contain the string `Jan`. The pipeline displays the files that were last changed in January.

A *filter* is a command that can read from standard input and write to standard output. A filter is often used within a pipeline. In the following example, **grep** is the filter:

```
ps -e | grep cc | wc -l
```

lists all your processes currently active in the system, pipes the output to **grep**, which searches for every instance of the string *cc*. The output from **grep** is then piped to **wc**, which counts every line in which the string *cc* occurs and sends the number of lines to standard output.

Using Substitution in Commands

Another shell feature that is useful for programmers is *command substitution*. When encountering a construct of the form:

```
`command`
```

in an input command line, the shell runs the given *command*. It then puts the output of the command, after converting newlines into spaces, back into the command line, replacing *command*, and runs the new command line. This is called *command substitution*.

As an example of how a programmer could use command substitution, consider a file called **srclist**, containing the following list of source code filenames: **alpha.c**, **beta.c**, and **gamma.c**. If you enter the command:

```
grep printf `cat srclist`
```

the shell runs **cat** against the contents of **srclist**, and rewrites the original command line, so that this line appears as:

```
grep printf alpha.c beta.c gamma.c
```

This line is then run, with **grep** searching through the given files, displaying lines that contain the string `printf`. This type of construct quickly locates all references to a particular variable or function in the source code for a program.

Using the find Command in Command Substitution Constructs

The **find** command is useful in command substitution constructs. **find** displays the names of files that have specified characteristics. For example:

```
find dir1 –name "*.c"
```

finds all files in the directory **dir1** whose names match the wildcard pattern `*.c`. In other words, it finds all files in that directory with names having the **.c** suffix.

The command:

```
ls -l `find dir1 –name "*.c"`
```

finds all the **.c** files and then uses **ls** to display information about these files.

Complicating things further, you could enter

```
ls -l `find dir1 –name "*.c"` | grep -F "Nov"
```

This sets up a pipeline that displays **ls** information only for files that were last changed in November. (To be perfectly accurate, it also displays information on files that have the string `Nov` in their names, too.)

Another useful **find** option has the form:

```
find path –ctime number
```

This says that you want to find files that have changed in the last *number* of days. For example:

```
ls -l `find dir –ctime 1`
```

displays **ls** information on all files that changed either yesterday or today.

On many UNIX and AIX systems, the **find** command prints out the filenames only if you specify the **–print** option. Thus, you would have to enter:

```
find dir –name "*.c" –print
```

to get the results just described. The OS/390 UNIX **find** command automatically prints its results without **–print**. However, if you have an existing shell script or compatibility with UNIX systems is important to you, you can use **–print**.

For more information on the **find** command, see *OS/390 UNIX System Services Command Reference*.

Characters That Have Special Meaning to the Shell

Certain characters have special meaning to the shell; these are often called *metacharacters*. If you enter a command that contains any of these characters, the shell often assumes that you are using the character in its special sense.

Characters Used with Commands

Character	Usage
|	Pipes the output from one command to a second command; separates commands in a *pipeline*.
||	Separates two commands. If the command preceding || fails, it runs the following command (Boolean OR operator).
>	Redirects stdout.
<	Redirects stdin.
&	Runs a command in the background, if placed at the end of a command line.
>&	Used for redirecting stdout and stderr.
&&	Separates two commands. If the command preceding && succeeds, it runs the following command (Boolean AND operator).
;	Separates sequential commands; allows you to enter more than one command on the same line.
( )	Around a sequence of commands, groups those commands that are to run as a separate process in a subshell environment. The commands run in a separate execution environment: changes to variables, the working directory, open files, and so on, will not remain in effect after the last command finishes. ( ) is also used to group mathematical operations.
{ }	Around a sequence of commands, groups those commands that are run in the current shell environment. Changes to variables, etc., will affect the current shell.

Both { and } are reserved words to the shell. To make it possible for the shell to recognize these symbols, you must enter a blank or <newline> after the {, and a semicolon or <newline> before the }.

Following a command in a shell script, indicates the beginning of a comment.

$ At the beginning of a string, indicates it is a variable name.

**** In general, the backslash character turns off the special meaning of the character that follows it. For more information, see “Using a Special Character without Its Special Meaning” on page 64.

' ' A pair of single quotes turns off the special meaning of all characters within the quotes. For more information, see “Using a Special Character without Its Special Meaning” on page 64.

" " A pair of double quotes turns off the special meaning of the characters within the quotes, except that `!event`, `$var`, and `` `cmd` `` will show history, variable, and command substitution. See “Using a Special Character without Its Special Meaning” on page 64 for more information.

Characters Used in Filenames

Character	Usage
/	Separates the components of a file's pathname.
~	(Tilde) symbolizes your home directory when used by itself. When used together with a user ID, ~ symbolizes that user's home directory. For example: `~valerie/.tcshrc` refers to user VALERIE's **.tcshrc** file.
.	When used as a component of a pathname, indicates the working directory.
..	When used as a component of a pathname, indicates the parent directory.
?	Used as a wildcard character that can match any one character, except a leading dot (**.**).
*****	Used as a wildcard character that can match a sequence of zero or more characters, except a leading dot (**.**).

Redirecting Input and Output

Character	Usage	Example
<	Redirects input to a specified file.	“Redirecting Input from a File” on page 56.
>	Redirects output to a specified file.	“Redirecting Command Output to a File” on page 55.
>>	Redirects output to be appended to the end of the specified file.	“Redirecting Command Output to a File” on page 55.
>&	Redirects stdout and stderr.	“Redirecting Error Output to a File” on page 56.

Character	Usage	Example

<<text	Reads standard input until it encounters *text*.	This is used in what is called a “here document.” Input is usually typed on the screen or in a shell script. For example, this script creates a file called **hello.c**, compiles it into **hello**, and then executes it:
		`# create program`
		`cat > hello.c << EOF`
		`main() {`
		`  puts("Hello, World!\n");`
		`}`
		`EOF`
		`# compile program`
		`c89 -o hello hello.c`
		`#execute program`
		`hello`
		
		When you run the shell script, it runs the **cat** > **hello.c** command using the input between the two End_of_File strings.

Using a Special Character without Its Special Meaning

If you do not want to use the special sense of the metacharacters, instruct the shell to ignore them by escaping them or quoting them. To do this, you use:

```
\
' '
" "
```

The Backslash (\)

The backslash character (\) turns off the special meaning of the character that follows it. For example:

```
echo it\'s me
```

prints:

```
it's me
```

If you just try:

```
echo it's me
```

without the backslash, the shell prints a > prompt after you press <Enter>instead of the usual $. The > prompt is a *continuation prompt*. An apostrophe ' without a backslash is taken to be the start of a string and the shell assumes that the string keeps going until you type another apostrophe, even if that goes on for several lines. The shell does not process the string until you type the closing apostrophe.

So remember to put a backslash in front of any special character, unless you know its special meaning and you want that meaning. Because a backslash itself is a special character, you must type two of them whenever you want a single backslash.

A Pair of Single Quotes (' ')

A pair of single quotes (**' '**) turns off the special meaning of *all characters* within the quotes.

A Pair of Double Quotes (" ")

A pair of double quotes turns off the special meaning of the characters within the quotes, except that `!event`, `$var`, and `` `cmd` `` will show history, variable, and command substitution.

Using a Wildcard Character to Specify Filenames

If you have used other operating systems, you are probably familiar with the concept of *wildcard characters*. (In an MVS context, the wildcard character is referred to as a *global character,* or *pattern-matching character*.) A wildcard character is a special character that may be used to save typing in filenames in shell commands. The tcsh shell recognizes several different wildcard characters:

```
*
?
[]
```

The * Character

The asterisk (*****) stands for any sequence of zero or more characters, except a leading dot. You can use the asterisk in filenames. For example:

```
ls aa*
```

lists all files in the working directory with names that begin with `aa`.

The command:

```
mv *.c dir1/dir2
```

moves every file with the **.c** suffix from your working directory to the directory **dir1/dir2**.

You can use the ***** wildcard character in directory names as well as in filenames. For example:

```
cat */*.c
```

displays the contents of all files that have the **.c** suffix, in directories under your working directory.

The ? Character

In a pathname, the question mark **?** can stand for any single character, except a leading dot. For example:

```
file.?
```

refers to any and all files with names that consist of **file.** followed by any single character. This can mean **file.a**, **file.b**, **file.c**, and so on ... whichever of the files currently exist.

You can combine ***** and **?**.

```
ls *.?
```

displays the names of all files under the working directory that have one-character filename suffixes.

Again, you can use the **?** in directory names as well as filenames. For example:

```
ls ???/*
```

shows all files in every directory under your working directory that have a three-character name.

The Square Brackets [ ]

Square brackets containing one or more characters stand for any one of the contained characters. For example:

```
[bch]at
```

matches **bat**, **cat**, or **hat**.

```
ls [abc]*
```

lists all files in the working directory the names of which start with `a`, `b`, or `c`, followed by any other sequence of zero or more characters. In other words, it lists all files whose names start with `a`, `b`, or `c`.

You can specify ranges of characters inside the square brackets by specifying the first character in the sequence, a hyphen (**-**), and the last character. For example:

```
[a–m]
```

This matches any character from `a` through `m`.

Suppose, for example, that you want to copy the contents of the working directory into two separate directories. You might enter:

```
cp [a–m]* dira
```

to copy all files with names beginning with the letters `a` through `m` to the directory **dira**, and then issue the second command:

```
cp [n–z]* dirb
```

to copy the rest of the files to the directory **dirb**. A command such as:

```
rm *.[a-z]
```

removes every file with a suffix consisting of a single lowercase letter.

If the first character inside a bracket construct is an exclamation mark **!**, the construct matches any character that *is not* inside the brackets. For example:

```
ls [!a–m]*
```

lists any file that *does not* begin with one of the letters in the range `a` through `m`.

In the same way:

```
rm [!0-9]*
```

removes any file with a name that does not start with a digit.

Retrieving Previously Entered Commands

In the tcsh shell, you can retrieve previously issued commands using:

- The **history** command, combined with the **!** command
- The two retrieve function keys that are part of the TSO/E OMVS command interface to the shell
- Command-line editing, when you are using an asynchronous terminal interface

Retrieving Commands from the History File

The shell records each command that you enter in a file under your *home directory*. This file is called the *history file*; its name is **.history**. If you enter the command:

```
history
```

the shell displays the current contents of your history file. Each command is numbered.

You can rerun any of the commands in your history file by typing **!**, followed by a space, followed by the number of the command you want to use.

For example, suppose that you are a programmer and you enter a complicated command to compile part of a program. The program contains a syntax error, so you call a text editor to edit the source code and correct the problem. Now you want to run the same compile command on the corrected program. You may save yourself a good deal of typing by using:

```
history
```

to find out the number of the previous compile command and then running the command with **!**. For example, if the history file shows you that the command you want to run is number 44, you would type:

```
! 44
```

to run the previous compile command.

Another time-saver is to specify your shell prompt as:

```
set prompt="\!>
```

in your **.tcshrc** file. The shell prompt is then preceded by the number assigned to the command in the command history file.

If you type **!** followed by a space, followed by a string of characters (not beginning with a digit), the shell checks backward through the history file and runs the most recent command that begins with the given string. For instance, look at the compilation example. Suppose you are using the **c++** command to compile your program. Then:

```
! c++
```

looks back through the history and runs the most recent **c++** command. You do not even have to check on the number of the command you want to enter. The shell displays the selected command in the output area of the screen and then runs it.

This *backward-search* feature of **!** can search for aliases as well as normal commands. **!** searches for the beginning of the command line as you typed it, not the way that the line looked after the alias was replaced.

If you enter **!!** without a number after it, the shell repeats the most recent command.

Editing Commands from the History File

Suppose that you have a sequence of source files named **file1.c**, **file2.c**, **file3.c**, and so on that you want to compile with similar **c89** commands. This situation is a little different from the one discussed in the previous section. You do not want to rerun the *same* command for each file; the command has the same form each time, but you have to specify in a new filename each time.

You can still do this using the history file. The command:

```
^old_string^new_string
```

runs a previous *command* but replaces the first occurrence of the *old* string with the *new* string. For example, suppose you compile **file1.c** with:

```
c89 options file1.c
```

Then the command:

```
^file1^file2
```

tells the shell to look at the previous command and to change **file1** to **file2**. The shell makes this change, and then displays and runs the modified command.

```
^file2^file3
```

performs the same kind of operation, changing **file2** in the previous command to **file3** and then going ahead with the compilation. This saves you the trouble of retyping all the options for the command.

Using the Retrieve Function Keys

If you are using the OMVS interface, there are two function key settings for retrieving commands:

Retrieve This key performs a “backward retrieve” function. It retrieves a saved command from a stack of saved input lines, starting with the most recent and moving down to the oldest available line.

FwdRetr This key is used with the Retrieve key to retrieve commands from the stack of saved input lines. If you press the Retrieve key one too many times and go past the line you want, you can press the FwdRetr key to display the line that was previously retrieved by the Retrieve key.

Press the Retrieve key repeatedly until the command you want to use is displayed on the command line. Once the command is displayed, you can modify the command or use it as it is displayed. Press <Enter> to run the command.

Command-Line Editing

When you use **rlogin** or **telnet** to login to the shell, you can use command-line editing. Command-line editing lets you access commands from your history file, edit them, and run the result. You have already seen this process before, when reading about some of the features of the **!** command.

Command editing is useful at those times when you are running the same sequence of commands, or slight variations on the same sequence of commands. The point of command editing is to save yourself the trouble of typing the same

thing over and over again—look especially for long commands that normally require a lot of typing. Command editing is also useful when you have made a mistake in typing a command line and wish to correct it.

Using the vi Command Editor

If you run the command:

```
bindkey -v
```

it tells the shell that you want the ability to edit commands the way that you normally edit text with **vi**; you are set up for **vi** command editing. Whenever the shell prompts you for input, it is as if the shell puts you into **vi** insert mode on a new line at the end of the history file. You can type in a new command just as you normally would.

You can also press <Esc> to enter a **vi**-like command mode. When you enter command mode, you can use the usual cursor movement commands to move around on the command line, or to move up and down in the history file. For example:

- Press the **k** key to move back to the previous line in the history file (the last command line you entered). Press the **k** key again, and you move to the line before that.
- Press **j** and you move forward in the history file.

In this way it is simple to retrieve recent commands from the history file. You can then edit them using standard **vi** commands. For example, you can use **$** to move to the end of the line, and **A** to begin appending text to the end of the line. When you have edited the line to produce the command that you want to run, simply press <Enter> to run that line.

As you might expect, you can use these search commands:

/string
?string

to search backwards and forwards through the history file. You can edit the command line with these **vi** commands:

w	Move to next word
b	Move to previous word
d	delete
c	change
a	append
i	insert
u	undo

and many of the other **vi** commands. For a complete list of available commands, see the description of **tcsh** in *OS/390 UNIX System Services Command Reference*.

Using the emacs Command Editor

To set up for **emacs** command editing, enter:

```
bindkey -e
```

This lets you use commands identical to **emacs** commands to edit your shell command line. For a more information, see the description of **tcsh** in *OS/390 UNIX System Services Command Reference*.

Using Filename Completion

Note: Filename Completion requires the use of the TAB key. This key must be mapped correctly for the feature to work. Most connections through **telnet** and **rlogin** will transmit the TAB information correctly. If you are connected in any other manner, this feature may not work correctly.

The tcsh shell provides a time saving feature for completing filenames. Rather than having to type out the entire string to access a file or execute a program, you can type just the first letter or letters and let the shell help you with the rest.

For example, if you have a file called *phonebook*, and you want to list the contents of this file on the screen with the **more** command, you can do so by typing the command, the first letter or letters of the file, and then pressing the TAB key. For example, if you type:

```
more ph
```

and then press the TAB key, the shell will provide you with:

```
more phonebook
```

you can then press ENTER and execute the command.

If you have more than one filename that matches the letter or letters you have typed, the shell will alert you with a beep. For example, if you have three files, called *list1*, *list2*, and *list3*, and you type:

```
more li
```

and press TAB, the beep will sound, and the shell will complete the filename as far as it can:

```
more list
```

you must then type *1*, *2*, or *3* and press ENTER.

If you are unsure of how many files there are, or which one you want, you can type <CRTL-D> when the shell beeps, and you will be provided with matching names. For example:

```
> more list
list1   list2  list3
> more list
```

Underneath the matching names the command prompt is displayed again. Now you can enter the number that you wish and then press ENTER.

If there are no matches for the letter or letters you have typed, the shell will beep, but when you press <CRTL-D>, nothing will be displayed.

You can also use filename completion to aid in changing between directories with long paths. If you keep files in the directory *stuff/data/graphics*, it is easier to use filename completion to access the directory than to type the entire path by hand. For example, if you are in your home directory, and *stuff* is a subdirectory containing *data/graphics*, and you want to change into that directory, you can do the following:

```
cd s    [TAB]
cd stuff/
cd stuff/d    [TAB]
cd stuff/data
cd stuff/data/g    [TAB]
cd stuff/data/graphics
```

then press ENTER, and the directory change command will execute.

More information on filename completion and the **complete** command can be found in the *OS/390 UNIX System Services Command Reference*.

Using Record-Keeping Commands

Record-keeping commands can be very helpful for programmers. For example, suppose you have a program that is split into several source files. For the sake of simplicity, assume that the source files all have the extension **.c** and are all stored in a subdirectory called **src**.

It is often the case that you want to find out which source files in the subdirectory refer to a particular variable or function. You can do this very simply with the command:

```
grep 'name' src/*.c
```

The command checks all the appropriate files in the subdirectory **src** and displays the lines that contain the given *name*. Each line is labeled with the name of the file that contains the line. You can quickly find the use of a function or data object in source files.

As another example of using record-keeping commands, suppose that you are working on a large program and every few days you back up the source code for the program by copying it to a directory in a different file system (as a precaution). You would like to compare the current versions of your source files with one of the saved versions, to find out what changes have been made between the two. The command:

```
diff oldfile newfile
```

prints out all the differences between two versions of a file, making comparisons possible.

The **cksum** command gives a checksum for each file. If applied to two versions of what was at one time the same file, **cksum** gives a convenient way to tell if the files are still the same. It does not, however, indicate what the differences are.

The **find** command also has applications to programming. For example, suppose you are looking for a particular C source program but cannot remember where it is stored.

```
find / –name '*.c'
```

searches all the files and file systems, starting at the root, and displays the names of all files with the **.c** extension.

Finding Elements in a File and Presenting Them in a Specific Format

awk is a powerful command that can perform many different operations on files. The general purpose of **awk** is to read the contents of one or more files, obtain selected pieces of information from the files, and present the information in a specified format.

One simple way to use **awk** is with a command line with the form:

```
awk '/regexp/ {action}' file
```

This asks **awk** to obtain information from the specified file. **awk** obtains the information by performing the specified *action* on every line in the file that contains a string matching the given regular expression, *regexp*. (For further information, see the appendix on regular expressions in *OS/390 UNIX System Services Command Reference*.) For example:

```
awk '/abc/ {print}' file
```

displays every record in the file that contains the string `abc`.

Timing Programs

The **time** command lets you time programs to find out how much processor time they actually require. You might use this to compare two versions of a program to see if one runs faster than the other. You can run a program with:

```
time command-line
```

where *command-line* is a command line that invokes the program you want to time. **time** runs the program and displays:

- The total time the program took to execute, labeled `real`
- The total time spent in the user program, labeled `user`
- The central processor time spent performing system services for the user, labeled `sys`

Using the passwd Command

You can change user's passwords by using the **passwd** command:

```
passwd [-u userid]
```

The **passwd** command changes the login password for the user ID specified. If *userid* is omitted, the login name associated with the current terminal is used. You are prompted for the new password, which may be truncated to the length defined as the maximum length for the passwords.

For example:

```
passwd
```

changes the password for the invoker. The invoker is prompted for the old password and the new password values.

Non-superusers can change the password for another user if they know the user ID and current password. Another example changes the password for user ID Jerry:

```
passwd -u Jerry
```

For more information about the **passwd** command, see *OS/390 UNIX System Services Command Reference*.

Switching to Superuser or Another ID

With the **su** command, you can switch to any user ID, including the superuser. A user can switch to superuser authority (with an effective UID of `0`), if the user is permitted to the BPX.SUPERUSER FACILITY class profile within the Resource Access Control Facility (RACF). Either the ISPF shell or the **su** shell command can be used for switching to superuser authority.

If you do not specify a user ID, the **su** command changes your authorization to that of the superuser. If you specify a user ID, **su** changes your authorization to that of the specified user ID.

When you switch to superuser (UID `0`) without specifying a userid, you keep your MVS identity (TSO/E ID). You keep your access authority to MVS data sets, while gaining authority to access any HFS files.

When you change user ID by specifying a user ID and password, you assume the MVS identity of the new userid even if the userid has UID `0`.

If you use the –s option on the **su** command you will not be prompted for a password. Use this option if you have access to the SURROGATE facility class profile BPX.SRV.*userid*. The *userid* is the MVS userid associated with the target UID.

To return to your own user ID, type:

```
exit
```

This returns you to the shell in which you entered the **su** command.

Using the whoami Command

The **whoami** command displays a username associated with the effective user ID, unlike the **who am i** command which displays the login name.

For example, if you login as 'user1' but then you use the **su** command to change to 'user2':

command	**returned**
`who am I`	`user1`
`whoami`	`user2`

For more information on the **whoami** command, see *OS/390 UNIX System Services Command Reference*.

Using the tso Command

To run a TSO/E command from the shell or in a shell script, simply preface the TSO/E command with the **tso** shell command; for example:

```
tso -t tso_command
```

There are two options you can use:

- Specify the *-t* option to run a command through the TSO/E service routine. The command output is written to **stdout**. If you specify a relative pathname, the command looks for the file in your current directory.

  Note: TSO/E has some restrictions on the type of commands that can be run using the TSO/E service routine (mini-TSO environment). In summary, you cannot run the following commands in this environment:

  - Commands that run authorized
  - FIB (foreground initiated background) commands
  - Other commands that require the TSO/E task structure, i.e., interactive commands such as **oedit**, where interactive means that the user can interact with the command processing while issuing additional terminal input (subcommands, function keys). For example, once the **oedit** command is entered, the user can enter additional subcommands to add more lines and then quit or exit the command.

  For a full description of the restrictions, see the section on IKJTSOEV in *OS/390 TSO/E Programming Guide*.

- Specify the *-o* option to run a TSO command as if it had been entered on the OMVS command line and run using the TSO subcommand or function key. If you use a relative pathname, the command looks for the file in the working directory of your TSO/E session, which is typically your home directory.

If no option is specified, the following rules are applied in this order:

1. If **stdout** is not a tty, the TSO service routine is used since it is possible that the command output is redirected to a file or piped to another command. Otherwise,
2. If the controlling tty supports 3270 passthrough mode, OMVS is used. Otherwise,
3. The TSO service routine is used.

The **tso** command supports several environment variables. For more information about the **tso** command and the environment variables associated with it, see *OS/390 UNIX System Services Command Reference*.

Online Help

Two help facilities are available with the shell:

- The **man** command, which you can use to display help information about a shell command. The man page is displayed in your shell session, and you can work in the shell while viewing the help information.
- The TSO/E OHELP command, which displays online reference information about shell commands, TSO/E commands, C functions, callable services, and messages issued by the shell and **dbx**.

  The IBM BookManager READ product is a requirement for OHELP. The help information is displayed in a BookManager session; while viewing the help information, you cannot work in the shell.

Using the man Command

You can use the **man** command to get help information about a shell command. The **man** syntax is:

man *command_name*

- To scroll the information in a man page, press <Enter>.
- To end the display of a man page, type **q** and press <Enter>.

To search for a particular string in a system that has a list of one-line command descriptions, use the *-k* option:

man *-k string*

For example, to produce a list of all the shell commands for editing, you could type:

```
man -k edit
```

You can use the **man** command to view manual descriptions of TSO/E commands. To do this, you must prefix all commands with **tso**. For example, to view a description of the MOUNT command, you would enter:

```
man tsomount
```

You can also use the **man** command to view manual descriptions of **dbx** subcommands. To do this, you must prefix all subcommands with **dbx**. For example, to view a description of the **dbx alias** subcommand, you would enter:

```
man dbxalias
```

For complete information about the **man** command, see *OS/390 UNIX System Services Command Reference*.

Using the OHELP Command

The TSO/E OHELP command provides a similar capability to the **man** shell command. OHELP displays online reference information about commands, C functions, callable services, and messages issued by the shell and **dbx**.

Your system must have the BookManager READ product installed for you to use OHELP.

The OHELP syntax is:

```
OHELP ref_id search_item
```

ref_id A number that specifies the online book to be searched. The default is 1 for the *OS/390 UNIX System Services Command Reference*. Each installation can define which number is associated with each book. To see the list of available books and the number associated with each book, type `ohelp`.

search_item This can specify a:

- Command name
- C function name
- Callable service name
- Message number
- Text string (enclosed in double quotes)

If you omit this operand, OHELP displays the table of contents of the book specified by the `ref_id`.

Example: Getting Help for a Command

For example, if you want information on the **cp** shell command, you would enter:

```
OHELP cp
```

(You do not need to enter the value 1 because 1 is the default.)

```
                        List All Topics with Matches

Fuzzy matches for: CP
                                                Search matches 1 to 13 of 13
2.33     cp -- Copy a file
A.0      Appendix A.  OS/390 Shell Command Summary
2.21     chcp -- Set or query ASCII/EBCDIC code pages for the terminal
FRONT_1  Permuted Index
2.17     cat -- Concatenate or display a text file
2.89     ln -- Create a link to a file
2.107    mv -- Rename or move a file or directory
2.133    rm -- Remove a directory entry
2.159    touch -- Change the file access and modification times
2.162    trap -- Intercept abnormal conditions and interrupts
2.34     cpio -- Copy in/out file archives
2.42     dd -- Convert and copy a file
2.184    vi -- Use the display-oriented interactive text editor


Command ===> _____________________________________________ SCROLL ===> PAGE
 F1=Help     F4=Text     F5=No Text  F6=Review   F7=Bkwd     F8=Fwd
F10=Explain F12=Cancel
```

Figure 11. Sample Output from the Command OHELP cp

When you look at the output, you can see a boxed display overlaying another display. The boxed display, titled “List All Topics with Matches” lists all references to the **cp** command in the online *OS/390 UNIX System Services Command Reference*.

- `Fuzzy matches for: CP` is the heading for the list of references to **cp** that were found. BookManager converts the shell command name to uppercase.
- `Search Matches 1 to 13 of 13` indicates that this boxed display contains all of the search matches. If there were a very long list of search matches, you would need to scroll to the next screen to get to the end of the list.
- If you press PF4 (PF4=TEXT) while viewing the list, an explanation of the reason for the match is displayed.
- Your cursor is under the first item in the boxed display: `2.33 cp`. This is the **cp** command description from *OS/390 UNIX System Services Command Reference* . The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire command description.
- To redisplay the boxed display of the search results, type `search cp`. Press <Enter>. Alternatively, you can position your cursor under the selection `Search`

at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.

- After you select a match, you can use type `find cp` to move to the next match

If you press the Cancel function key, the boxed display disappears and you see the underlying information: the table of contents for the online *OS/390 UNIX System Services Command Reference*.

To exit the online help, use the Cancel and Exit function keys, as appropriate, from each panel.

Example: Searching Help for All Instances of a Language Element Name

If you want to look at the reference information for all types of language element with the name **chmod**, you enter the command:

```
ohelp * chmod
```

The output displayed would look similar to this:

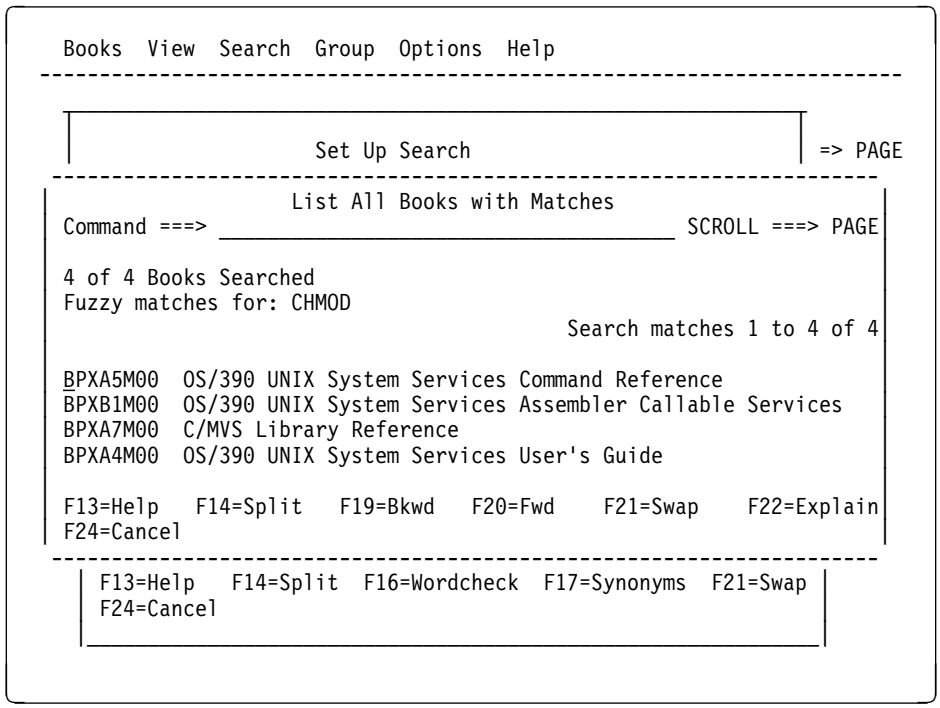

*Figure 12. Sample Output from the Command OHELP * chmod*

When you look at the output, you see a boxed display overlaying another display. The boxed display, titled “List All Books with Matches” lists all the reference books that document a language element named **chmod** command.

- `4 of 4 Books Searched` indicates that four books were searched.
- `Fuzzy matches for: CHMOD` is the heading for the list of references to **chmod** that were found. BookManager converts the shell command name to uppercase.

- `Search matches 1 to 4 of 4` indicates that this boxed display contains all of the search matches.
- Your cursor is under the first book in the list: `BPXA5M00`. If you press <Enter>, you see a boxed display showing all search matches for **chmod** in the online *OS/390 UNIX System Services Command Reference*. The first item in the list is usually the reference information for the language element you specified. Press <Enter>. You can read through the entire function description.
- To return to the boxed display from the reference information, position your cursor under the selection `Search` at the top of the screen. Press <Enter>. On the pulldown menu, select `List all topics with matches` and press <Enter>.
- The remaining items listed are cross-references to the **chmod** function throughout the online *OS/390 UNIX System Services Command Reference*.

If you press <F12>, the boxed display disappears and you see the "Set Up a Search" panel, which allows you to search for a different name.

To exit the online help, use <F12> and <F3> as appropriate.

Searching for a Text String

To search for a text string, enclose the text in double quotes and specify the `ref_id` for the specific book you want to search. For example, the command

```
ohelp 4 "improper type"
```

will search the book *OS/390 UNIX System Services Messages and Codes* for messages that contain the text *improper type*.

If you are searching for a text string and you use an `*` for `ref_id`, OHELP will search all the books on the shelf and locate every instance of that string.

Shell Messages

Messages issued by the tcsh shell and utilities are prefixed with the letters FSUC. To display online reference information about any shell message, use the OHELP command. The shell messages are documented in *OS/390 UNIX System Services Messages and Codes*.

Chapter 8. Writing tcsh Shell Scripts

Most people find themselves using some sequences of commands over and over again.

- A programmer may always use the same commands to compile source code, and link the resulting object code.
- A bookkeeper may have to go through the same sequence of shell commands each week to update the books and produce a report.

To simplify such jobs, the shell lets you run a sequence of commands that have been stored in a text file. For example, the programmer could store all the appropriate compiling and linking commands in a file. A file containing commands in this way is called a *shell script*. After such a file is completed and it is made "executable," the programmer can run all the commands in the file by entering the filename on the command line.

Putting commands in a shell script has several advantages over typing the commands individually. Using a shell script:

- Reduces the amount of typing you have to do. You have to type in the shell script only once. Then you can run all the commands in the script by entering the name of the file as a single shell command. A shell script can save you a lot of time and effort if you are working with many files, or if some command lines have several options.
- Reduces the number of errors. If you are typing in ten commands, you have ten chances to make a mistake. With a shell script, however, you can take your time, edit the file carefully, and get it right before you try to run it.
- Makes it easy for other people to do what you do. For example, consider the bookkeeper mentioned earlier. When the bookkeeper goes on vacation, someone else has to fill in. It is much easier for the substitute bookkeeper to type a single command that does everything correctly than to try to type in the full sequence of commands.

For all these reasons, you will probably find that the use of shell scripts makes your work easier and more productive. This chapter can provide only a brief overview, but it should give you an idea of how to write and use shell scripts.

Running a Shell Script

You can run a shell script by typing the name of the file that contains the script. For example, suppose you have a script named **totals.scp** that has three shell commands in it. If you enter:

```
totals.scp
```

the shell runs the three commands.

Before you can run a shell script, you must have read and execute permission to the file. Use the **chmod** and **umask** commands to set the permissions.

For another example, suppose you want to compile a collection of files written in the C programming language. You could use the **c89**, **cc**, or **c++** command. The

c89 command, for example, compiles any file **file.c**, link-edits the object module, and produces an executable file. The shell script:

```
c89 -c file1.c file2.c                          # compile only
c89 -o outfile file1.o file2.o file3.c          # outfile for executable
```

compiles and link-edits the files and produces an executable file, **outfile**. Notice that in a shell script you precede a comment with a `#`.

If you store this script in an executable file named **compile**, it could be run with the single command **compile**. A new process is created for the script to run in.

To run a shell script in your current environment, without creating a new process, use the **source** command. You could run the **calculate** shell script this way:

```
source calculate
```

Should you want to use a shell script that updates a variable in the current environment, run it with the **source** command.

Using the Magic Number

All tcsh scripts must have # as the first character of the script. When a script file starts with #*, the kernel's spawn and exec services recognize the file name after the #* as the program to be run. It is recommended that the first line of all tcsh scripts look like:

```
#!/bin/tcsh
```

with /bin/tcsh being the location of tcsh on the OS/390 UNIX system. The kernel recognizes the magic value (#*) and runs **/bin/tcsh**.

Using TSO/E Commands in Shell Scripts

A shell script can include TSO/E commands as well as shell commands, and it can process TSO/E command output. You use the **tso** shell command to run the TSO/E command.

Using Variables

You can think of shell scripts as *programs* made up of shell commands. To allow more versatile shell scripts, the shell supports many of the features of normal programming languages.

In a conventional programming language, a *variable* is a name that has an associated value. When you want to use the value, you can use the name instead.

Creating a Shell Variable

The shell also lets you create variables. A shell variable name can consist of uppercase or lowercase letters, plus digits and the underscore character _. The name can have any length, but the first character cannot be a digit. Uppercase letters are distinguished from lowercase ones, so **NAME**, **name**, and **Name** are all *different* names.

To create a shell variable, just enter:

```
set name='string'
```

as a command to the shell. For example:

```
set home='/usr/adams'
```

sets up a variable with the name **home** and the value **/usr/adams**.

After you set a variable, you refer to it by prefixing its name with a dollar sign ($). Any command can use the value of a variable by referring to it this way. For example, if **home** is set to **/usr/adams**:

```
cd $home
```

is equivalent to:

```
cd /usr/adams
```

Similarly:

```
cp $home/* /newdir
```

is equivalent to:

```
cp /usr/adams/* /newdir
```

To change the value of an existing variable, you use a command with the same form as the existing variable. For example:

```
set home='/usr/benjk'
```

changes the value of **home** from **/usr/adams** to **/usr/benjk**.

If the value on the right-hand side of the **=** sign does not contain spaces, tab characters, or other special characters, you can leave out the single quotes. For example, you can enter:

```
home=/usr/benjk
```

Calculating with Variables

Suppose you run the following commands either in a shell script or by typing in one command after another:

```
set i=1
set j=$i+1
echo $j
```

The output of **echo** is `1+1` because a normal variable assignment assigns a *string* to a variable. Thus `j` gets the string `1+1`.

To *evaluate* an arithmetic expression, you can enter:

```
@ variable=expression
```

This command line assigns the value of an expression to the given variable. For example:

```
i=1
@ j=$i + 1
echo $j
```

Here `j` is assigned the value of the expression and the **echo** command displays the value `2`.

You can also use **@** to change the value of a variable. If you enter:

```
i=1
@ i=$i + 1
echo $i
```

the **@** command *changes* the value of `i`. The new value of `i` is the old value plus 1.

An **@** command can have any of the standard arithmetic expressions:

`-A`	Negative `A`
`A * B`	`A` times `B`
`A / B`	`A` divided by `B`
`A % B`	Remainder of `A` divided by `B`
`A + B`	`A` plus `B`
`A - B`	`A` minus `B`

The standard mathematical order of operations is used, as shown in the way that operations are grouped:

- All unary minus operations are carried out;
- Then any `*`, `/`, or `%` operations (from left to right in the order they appear);
- Then any additions or subtractions (from left to right in the order they appear).

Many operators use special shell characters, so you usually need to put double quotes around the expression. Thus:

```
@ i =  5 + 2 * 3
```

assigns 11 to `i`, since the multiplication is done first. You can use parentheses in the usual way to change the order of operations. For example:

```
@ i = ((5 + 2) * 3 )
```

assigns 21 to `i`.

Note: **@** does not work with numbers that have fractional parts. It works only with integers.

Setting Environment Variables

Up to this point, we have talked about defining shell variables and then using them in later command lines. You can also define a shell variable and then call a shell script that makes use of that variable. But you have to do a certain amount of preparation first.

A shell script is run as a child process to the parent shell. By default, the child process does not share any variables with the parent. If you define a variable **var** in the parent shell, it is *local* to the current session; any shell script, or child process, that you call will not inherit **var**.

To deal with this situation, you can enter the following:

```
setenv var [value]
```

The **setenv** command says that you want the variable **var** passed on to all the child processes that you execute in this session. After you do this, **var** becomes inherited and the variable is known to all the commands and shell scripts that you use.

As an example, suppose you enter the commands:

```
setenv myname "Friar Tuck"
```

Now all your child processes can use the **myname** variable to obtain the associated name. You may, for example, have shell scripts that write form letters that contain your name, `Friar Tuck`, obtained from the **myname** variable.

Note: You could use single or double quotes to enclose the variable value. See "Quoting Variable Values" on page 40 for more information.

When a script or child process begins running, it automatically inherits all the environment variables passed on to it. However, if the script changes the value of one of those variables, that change is *not* passed back to the parent process —unless you run the script with the **source** utility.

By default, any variables created within a shell script are *local* to that script. This means that when another program is run, those variables do not apply in its environment. However, the script can use the **setenv** command to turn shell variables into global environment ones. Inside a shell script:

```
setenv name [value]
```

indicates that the variable with the given *name* should be defined as an environment variable. When other programs are run from that script, they inherit the value of all environment variables. However, when the script ends, all its environment variables are lost to the calling shell.

Some variables are automatically inherited by the software that creates them. For example, if you invoke the shell, the initialization procedure automatically marks the **home** variables for environment variables so that other commands and shell scripts can use it. In Chapter 6, you saw that in a typical **.tcshrc** file for an individual user, the **PATH** variable is an environmental variable. Making the **PATH** variable an environmental variable ensures that search rules and changes to search rules are automatically shared by all shell sessions and scripts.

Using Positional Parameters — the $N Construct

The sample shell script discussed earlier in this chapter compiled and link-edited a program stored in a collection of source modules. This section discusses a shell script that can compile and link-edit a C program stored in any file.

To create such a script, you need to be familiar with the idea of *positional parameters*. When the shell encounters a `$N` construct formed by a `$` followed by a single digit, it replaces the construct with a value taken from the command line that started the shell script.

- `$1` refers to the first string after the name of the script file on the command line
- `$2` refers to the second string, and so on.

As a simple example, consider a shell script named **echoit** consisting only of the command:

```
#
echo $1
```

Suppose we run the command:

```
echoit hello
```

The shell reads the shell script from **echoit** and tries to run the command it contains. When the shell sees the `$1` construct in the **echo** command, it goes back to the command line and obtains the first string following the name of the shell script on the command line. The shell replaces the `$1` with this string, so the **echo** command becomes:

```
echo hello
```

The shell then runs this command.

A construct like `$1` is called a *positional parameter*. Parameters in a shell script are replaced with strings from the command line when the script is run. The strings on the command line are called *positional parameter values* or *command-line arguments*.

If you enter:

```
echoit Hello there
```

the string `Hello` is considered parameter value `$1` and `there` is `$2`. Of course, the shell script is only:

```
echo $1
```

so the **echo** command displays only the `Hello`.

Positional parameters that include a blank can be enclosed in quotes (single or double). For example:

```
echoit "Hello there"
```

echoes the two words instead of just one, because the two words are handled as one parameter.

Returning to a compile and link example, a programmer could write a more general shell script as:

```
c89 -c $1.c
c89 -o $1 $1.o
```

If this shell script were named **clink**, the command:

```
clink prog
```

would compile and link **prog.c**, producing an executable file named **prog** in the working directory. In the same way, the command:

```
clink dir/prog2
```

would compile and link **dir/prog2.c**. The shell script compiles and links a C program stored in a single file.

As another example of a shell script containing a positional parameter, suppose that the file **lookup** contains:

```
grep $1 address
```

(where **address** is a file containing names, addresses, and other useful information). The command:

```
lookup Smith
```

displays address information on anyone in the file named Smith.

Using Quotes to Enclose a Construct in a Shell Script

A `$N` construct in a shell script can be enclosed in double or single quotes.

- When *double* quotes are used, the parameter is replaced by the appropriate value from the command line. For example, suppose the file **search** contains:

  ```
  grep "$1" *
  ```

  If you enter the command:

  ```
  search 'two words'
  ```

  the parameter value `'two words'` replaces the construct `$1` in the **grep** command:

  ```
  grep "two words" *
  ```

  If the **grep** command does not contain the double quotes, the parameter replacement would result in:

  ```
  grep two words *
  ```

  which has an entirely different meaning.

- When you use *single* quotes to enclose a `$N` construct in a shell script, the `$N` is *not* replaced by the corresponding parameter value. For example, if the file **search** contains:

  ```
  grep '$1' *
  ```

  grep searches for the string `$1`. The `$1` is not replaced by a value from the command line. In general, single quotes are “stronger” than double quotes. Less is more!

Using Parameter and Variable Expansion

As we just discussed, a $ followed by a number stands for a positional parameter passed to the script or function. A positional parameter is represented with either a single digit (except 0) or two or more digits in curly braces; for example, 7 and {15} are both valid representations of positional parameters. For example, if the command:

```
echo $1
```

appeared in a shell script, it would **echo** the first positional parameter.

Similarly, a $ followed by the name of a shell variable (such as **$HOME**) stands for the value of the variable.

These constructs are called *parameter expansions*. In this sense, the term *parameter* can mean either a positional parameter or a shell variable.

The tcsh shell also supports more complicated forms of parameter expansions, letting you obtain only part of a parameter value or a modified form of the value.

Modifier	Description
r	Root of value
e	Extension of value
h	Head of value

Modifier	Description
t	Tail of value

For example, to extract only part of a filename, you can add one of the above modifiers as follows:

Filename	r	e	h	t
/usr/bin/vi.txt	/usr/bin/vi	txt	/usr/bin	vi.txt
/u/bobby/mail	/u/bobby/mail	empty	/u/bobby	mail
storybook.pdf	storybook	pdf	empty	storybook.pdf
INSTALL	INSTALL	empty	empty	INSTALL

Using Special Parameters in Commands and Shell Scripts

The tcsh shell has a variety of special parameters that may be used in command lines and shell scripts. These parameters are listed in *OS/390 UNIX System Services Command Reference* under **tcsh** in the "Variable Substitution" section.

Using Control Structures

The shell provides facilities similar to those found in programming languages. It offers these *control structures,* which are related to programming control structures:

- The **if** conditional
- The **while** loop
- The **for** loop

The if Conditional

An **if** conditional runs a sequence of commands if a particular condition is met. It has the form:

```
if (expr) command
```

The end of the commands is indicated by **endif**. For example, you could have:

```
if ( -d $1 ) then
     ls $1
endif
```

This tests to see if the string associated with the first positional parameter, `$1`, is the name of a directory. If so, it runs an **ls** command to display the contents of the directory.

Any number of commands may come between the **then** and the **endif** that ends the control structure. For example, you might have written:

```
if ( -d $1 ) then
     echo "$1 is a directory"
     ls $1
 endif
```

This example also shows that the commands do not have to begin on the same line as **then**, and the condition being tested does not have to begin on the same

line as **if**. The condition and the commands are indented to make them stand out more clearly. This is a good way to make your shell scripts easier to read.

Another form of the **if** conditional is:

```
if (expr) then
commands
else
commands
endif
```

If the condition is true, the commands after the **then** are run; otherwise, the commands after the **else** are run. For example, suppose you know that the string associated with the variable *pathname* is the name of either a directory or a file. Then you could write:

```
if (  -d $pathname ) then
     echo "$pathname is a directory"
     ls $pathname
 else
     echo "$pathname is a file"
     cat  $pathname
 endif
```

If the value of *pathname* is the name of a file, this shell script uses **echo** to display an appropriate message, and then uses **cat** to display the contents of the file.

The final form of the **if** control structure is:

```
if (expr1) then
commands1
else if (expr2) then
commands2
else if (expr3) then
commands3
else
commands
endif
```

In this example, if *expr1* is true, *commands1* are run; otherwise, the shell goes on to check *expr2*. If that is true, *commands2* are run; otherwise, the shell goes on to check *expr3* and so on. If none of the test conditions are true, the *commands* after the **else** are run. Here is an example of how this can be used:

```
if ( !  $?argv ) then
     echo "no positional parameters"
 else if ( -d $1 ) then
     echo "$1 is a directory"
     ls $1
 else if ( -f $1 ) then
     echo "$1 is a file"
     cat  $1
 else
     echo "$1 is just a string"
 endif
```

The test after the **if** determines if the value of the first positional parameter, `$1`, is an empty string. If so, there are no positional parameters, so the shell script uses **echo** to display an appropriate message; otherwise, the script checks to see if the parameter is a directory name; if so, the contents of the directory are listed with **ls** (after an appropriate message). If that does not work, the script checks to see if the

parameter is a filename; if so, the contents of the file are listed with **cat** (after an appropriate message). Finally, if none of the previous tests work, the parameter is assumed to be an arbitrary string, and the script displays a message to this effect.

You could put that script into a file named **listit** and run commands of the form:

```
listit name
```

to list the contents of *name* in a useful form.

The while Loop

The **while** loop repeats one or more commands while a particular condition is true. The loop has the form:

```
while (expr)
commands
end
```

The shell first tests to see if *condition* (expr) is true. If it is, the shell runs the *commands*. The shell then goes back to check the *condition*. If it is still true, the shell runs the *commands* again, and so on, until the *condition* is found to be false.

As an example of how this can be used, suppose you want to run a program named **prog** 100 times to get an idea of the program's average running speed. The following shell script does the job:

```
@ i=100
date
while ( $i > 0)
    prog
    @ i--
end
date
```

The script begins by setting a variable *i* to 100. It then uses the **date** command to get the current date and time.

Next the script runs a **while** loop. The condition says that the loop should keep on going as long as the value of *i* is greater than zero. The commands of the loop run **prog** and then subtract 1 from the *i* variable, similiar to C programming language syntax. In this way, *i* goes down by 1 each time through the loop, until it is no longer greater than 0. At this point, the loop stops and the final instruction of the script prints out the date and time at the end of the loop. The difference between the starting time and the ending time should give some idea of how long it took to run the program 100 times.

(Of course, the shell itself takes some time to perform the condition and to do the calculations with *i*. If **prog** takes a long time to run, the time spent by the shell is relatively unimportant; if **prog** is a quick program, the extra time that the shell takes may be large enough to make the timing incorrect.)

The foreach Loop

The final control structure to be examined is the **foreach** loop. It has the form:

```
foreach name (wordlist)
commands
end
```

The parameter *name* should be a variable name; if this variable doesn't exist, it is created. The parameter *list* is a list of strings separated by spaces. The shell begins by assigning the first string in *list* to the variable *name*. It then runs the *commands* once. Then the shell assigns the next string in *list* to *name*, and repeats the *commands*. The shell runs the *commands* once for each string in *list*.

As a simple example of a shell script that uses **foreach**, consider:

```
foreach file ( *.c )
  c89 $file
end
```

When the shell looks at the **foreach** line, it expands the expression `*.c` to produce a *list* containing the names of all files (in the working directory) that have the suffix **.c**. The variable *file* is assigned each of the names in this list, in turn. The result of the **foreach** loop is to use the **c89** command to compile all **.c** files in the working directory. You could also write:

```
foreach file ( *.c )
  echo $file
  c89 $file
end
```

so that the shell script displayed each filename before compiling it. This would let you keep track of what the script was doing.

As you can see, the **foreach** loop is a powerful control structure. The *list* can also be created with command substitution, as in:

```
foreach file ( `find . -name "*.c" -print` )
     echo $file
     c89 $file
end
```

Here the **find** command finds all **.c** files in the working directory, and then compiles these files. This is similar to the previous shell script, but also looks at subdirectories of the working directory.

Combining Control Structures

You can combine control structures by nesting (that is, putting one inside another). For example:

```
foreach file ( `find . -name "*.c" -print` )
        if ( -M $file > -M $1 ) then
                echo $file
                c89 -c $file
        endif
end
```

This shell script takes one positional parameter, giving the name of a file. The script looks in the working directory and finds the names of all **.c** files. The **if** control structure inside the **foreach** loop tests each file to see if it is older than the file named on the command line. If the **.c** file is older, **echo** displays the name, and the file is compiled. You can think of this as making a set of files up to date with the filename specified on the command line.

Chapter 9. tcsh Shell Command Summary

The following list presents the built-in tcsh shell commands, grouped by the task a user might want to perform, and their functions. Similar tasks are organized together.

General Use

alloc	Show the amount of dynamic memory acquired
builtins	Print the names of all built-in commands
bye	Terminate the login shell
echo	Write arguments to standard output
echotc	Exercise the terminal capabilities in args
exec	Run a command and open, close, or copy the file descriptors
glob	Write each word to standard output
hashstat	Print a statistic line on hash table effectiveness
login	Terminate a login shell
logout	Terminate a login shell
nice	Run a command at a different priority
notify	Notify user of job status changes
repeat	Execute command count times
source	Read and execute commands from name
time	Display processor and elapsed times for a command
where	Report all instances of command
which	Display next executed command

Controlling Your Environment

@ (at)	Print the value of tcsh shell variables, or assign a value
alias	Display or create a command alias
bindkey	List all bound keys, or change key bindings
complete	List completions
history	Display a command history list
hup	Run command so it exits on a hang-up signal
newgrp	Change to a new group
onintr	Control the action of the tcsh shell on interrupts
printenv	Display the values of environment variables
rehash	Recompute internal hash table
sched	Print scheduled event list
set	Set or unset command options and positional parameters
setenv	Set environment variable name to value
settc	Tell tcsh shell the terminal capability cap value
setty	Control tty mode changes
shift	Shift positional parameters
telltc	List terminal capability values
unalias	Remove alias definitions
uncomplete	Remove completions whose names match pattern
unhash	Disable use of internal hash table
unlimit	Remove resource limitations
unset	Unset values and attributes of variables and functions
unsetenv	Remove environment variables that match pattern
watchlog	Report on users who are logged in.

Managing Directories

cd	Change the working directory
chdir	Change the working directory
dirs	Print the directory stack
popd	Pop the directory stack
pushd	Make exchanges within directory stack

Computing and Managing Logic

break	Exit from a loop in a shell script
breaksw	Cause a break from a switch
continue	Skip to the next iteration of a loop in a shell script
default	Label default case in a switch statement
eval	Construct a command by concatenating arguments
exec	Run a command and open, close, or copy the file descriptors
exit	Return to the shell's parent process or to TSO/E
filetest	Apply a file inquiry operator to a file

Managing Files

ls-F	List files

Controlling Processes

bg	Move a job to the background
fg	Bring a job into the foreground
jobs	Return the status of jobs in the current session
kill	End a process or job, or send it a signal
limit	Limit consumption of processes
nohup	Start a process that is immune to hangups
stop	Suspend a process or job
suspend	Send a SIGSTOP to the current shell
time	Display processor and elapsed times for a command
wait	Wait for a child process to end

Part 3. OS/390 UNIX System Services Command Reference

Chapter 10. tcsh Commands

alias — Display or create a command alias

Format

alias [**–tx**] [*name*[=*value*] ...]
alias –r

tcsh shell: **alias** [*name* [*wordlist*]]

Description

When the first word of a shell command line is not a shell keyword, **alias** causes the shell to check for the word in the list of currently defined *aliases*. If it finds a match, the shell replaces the alias with its associated string value. The result is a new command line that might begin with a shell function name, a built-in command, an external command, or another alias.

When the shell performs alias substitution, it checks to see if *value* ends with a blank. If so, the shell also checks the next word of the command line for aliases. The shell then checks the new command line for aliases and expands them, following these same rules. This process continues until there are no aliases left on the command line, or recursion occurs in the expansion of aliases.

Calling **alias** without parameters displays all the currently defined aliases and their associated values. Values appear with appropriate quoting so that they are suitable for reinput to the shell.

Calling **alias** with parameters of the form *name*=*value* creates an alias for each *name* with the given string *value*.

If you are defining an alias where *value* contains a backslash character, you must precede it with another backslash. The shell interprets the backslash as the escape character when it performs the expansion. If you use double quotes to enclose *value*, you must precede each of the two backslashes with an additional backslash, because the shell escapes characters—that is, the shell does not interpret the character as it normally does—both when assigning the alias and again when expanding it.

To avoid using four backslashes to represent a single backslash, use single quotes rather than double quotes to enclose *value*, because the shell does not escape characters enclosed in single quotes during assignment. As a result, the shell escapes characters in single quotes only when expanding the alias.

Calling **alias** with *name* without any value assignment displays the function name (*name*) and its associated string value (*value*) with appropriate quoting.

DBCS Recommendation: We recommend that you use singlebyte characters when specifying an alias name, because the POSIX standard states that alias names must contain only characters in the POSIX portable character set.

alias in the tcsh shell

Without arguments, **alias** in the tcsh shell prints all aliases. With *name*, **alias** prints the alias for name. With *name* and *wordlist*, **alias** assigns *wordlist* as the alias of *name*. *wordlist* is command and filename substituted. *name* may not be *alias* or *unalias*.

See also “unalias in the tcsh shell” on page 197.

Options

–r Removes all tracked aliases.

–t Makes each *name* on the command line a tracked alias. Each tracked alias resolves to its full pathname; the shell thus avoids searching the **PATH** directories whenever you run the command. The shell assigns the full pathname of a tracked alias to the alias the first time you invoke it; the shell reassigns a pathname the first time you use the alias after changing the **PATH** variable.

When you enter the command:

```
set –h
```

each subsequent command you use in the shell automatically becomes a tracked alias. Running **alias** with the **–t** option, but without any specified names, displays all currently defined tracked aliases with appropriate quoting.

–x Marks each alias *name* on the command line for export. If you specify **–x** without any names on the command line, **alias** displays all exported aliases. Only exported aliases are passed to a shell that runs a shell script.

Several aliases are built into the shell. Some of them are:

```
alias autoload="typeset –fu"
alias functions="typeset –f"
alias hash="alias –t"
alias history="fc –l"
alias integer="typeset –i"
alias nohup="nohup "
alias r="fc –s"
alias stop="kill –STOP"
alias suspend="stop \$\$"
```

You can change or remove any of these aliases, and the changes will remain in effect for the current shell and any shell scripts or child shells invoked implicitly from the command. These aliases are reset to their default built-in values each time a new shell is invoked from the command line.

Example

The command:

```
alias ls="ls –C"
```

defines **ls** as an alias. From this point onward, when you issue an **ls** command, it produces multicolumn output by default.

alias in the tcsh shell: examples

To alias the **!!** history command, use **\!-1** instead of **\!\!**. For example:

```
alias mf 'more \!-1$'
```

creates an alias for looking at the file named by the final argument of the previously entered command. Example output would be the following:

```
alias mf 'more \!-1$'
echo "We love tcsh." > file1
mf


We love tcsh.
"file1" (EOF)
```

where **mf** pulls the last argument of the previous command (**file1**), and then displays that file using the **more** command.

Localization

alias uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Notes

1. **alias** is a built-in shell command.
2. Because exported aliases are only available in the current shell environment and to the child processes of this environment, they are not available to any new shell environments that are started (via the **exec sh** command, for example). To make an alias available to all shell environments, define it as a nonexported alias in the **ENV** file, which is executed whenever a new shell is run.

Exit Values

0 Successful completion
1 Failure because an alias could not be set
2 Failure because of an incorrect command-line option

If you define **alias** to determine the values of a set of names, the exit value is the number of those names that are not currently defined as aliases.

Portability

POSIX.2 User Portability Extension, OS/390 UNIX kornshell.

The **–t** and **–x** options are extensions to the POSIX standard.

Related Information

fc, **hash**, **nohup**,**set**, **sh**, **typeset**, **unalias**, **tcsh**

---

bg — Move a job to the background

Format

bg [*job...*]

tcsh shell: **bg** [*%job ...*]

Description

bg runs one or more jobs in the background. The *job* IDs given on the command line identify these jobs, which should all be ones that are currently stopped. If you do not specify any *job* IDs, **bg** uses the most recently stopped job.

bg works only if job control is enabled; see the **–m** option of **set** for more information. Job control is enabled by default in the OS/390 shell.

bg in the tcsh shell

In the tcsh shell, **bg** puts the specified jobs (or, without arguments, the current job) into the background, continuing each if it is stopped. *job* may be a number, a string, ", %, + or - .

In the tcsh shell, **%job &** is a synonym of the **bg** command.

Localization

bg uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Note

bg is a built-in shell command.

Exit Values

`0` Successful completion

`>0`
Failure because a *job* argument is incorrect or there is no current job

If an error occurs, **bg** exits and does not place the job in the background.

Portability

POSIX.2 User Portability Extension, UNIX systems.

Related Information

at, **batch**, **fg**, **jobs**, **set**, **tcsh**

---

break — Exit from a loop in a shell script

Format

break [*number*]

tcsh shell: **break**

Description

break exits from a **for**, **select**, **while**, or **until** loop in a shell script. If *number* is given, **break** exits from the given number of enclosing loops. The default value of *number* is 1.

break in the tcsh shell

In the tcsh shell, **break** causes execution to resume after the end of the nearest enclosing **foreach** or **while**. The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

Localization

break uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Note

break is a special built-in shell command.

Exit Value

break always exits with an exit status of 0.

Portability

POSIX.2, X/Open Portability Guide.

Related Information

continue, **sh**, **tcsh**

cd — Change the working directory

Format

cd [*directory*]
cd *old new*
cd –

tcsh shell: **cd** [**-p**] [**-l**] [**-nl-v**] [*name*]

Description

The command **cd** *directory* changes the working directory of the current shell execution environment (see **sh**) to *directory*. If you specify *directory* as an absolute pathname, beginning with /, this is the target directory. **cd** assumes the target directory to be the name just as you specified it. If you specify *directory* as a relative pathname, **cd** assumes it to be relative to the current working directory.

If the variable **CDPATH** is defined in the shell, the built-in **cd** command searches for a relative pathname in each of the directories defined in **CDPATH**. If **cd** finds the directory outside the working directory, it displays the new working directory.

Use colons to separate directories in **CDPATH**. In **CDPATH**, a null string represents the working directory. For example, if the value of **CDPATH** begins with a separator character, **cd** searches the working directory first; if it ends with a separator character, **cd** searches the working directory last.

In the shell, the command **cd -** is a special case that changes the current working directory to the previous working directory by exchanging the values of the variables **PWD** and **OLDPWD**.

Note: Repeating this command toggles the current working directory between the current and the previous working directory.

Calling **cd** without arguments sets the working directory to the value of the **HOME** environment variable, if the variable exists. If there is no **HOME** variable, **cd** does not change the working directory.

The form **cd** *old new* is an extension to the POSIX standard and optionally to the Korn shell. The shell keeps the name of the working directory in the variable **PWD**. The **cd** command scans the current value of **PWD** and replaces the first occurrence of the string *old* with the string *new*. The shell displays the resulting value of **PWD**, and it becomes the new working directory.

If either directory is a symbolic link to another directory, the behavior depends on the setting of the shell's **–o** logical option. See the **set** command for more information.

cd in the tcsh shell

If a directory name is given, **cd** changes the tcsh shell's working directory to *name*. If not, it changes the directory to home. If *name* is '-' it is interpreted as the previous working directory. If *name* is not a subdirectory of the current directory (and does not begin with /, ./ or ../), each component of the tcsh variable **cdpath** is checked to see if it has a subdirectory name. Finally, if all else fails but *name* is a

tcsh shell variable whose value begins with /, then this is tried to see if it is a directory (see also the **implicitcd** tcsh shell variable).

Options for the **cd** tcsh built-in command are:

-l Output is expanded explicitly to home or the pathname of the home directory for the user.

-n Entries are wrapped before they reach the edge of the screen.

-p Prints the final directory stack.

-v Entries are printed one per line, preceded by their stack positions.

If more than one of **-n** or **-v** is given, **-v** takes precedence. **-p** is accepted but does nothing.

Environment Variables

cd uses the following environment variables:

CDPATH
Contains a list of directories for **cd** to search in when *directory* is a relative pathname.

HOME Contains the name of your home directory. This is used when you do not specify *directory* on the command line.

OLDPWD
Contains the pathname of the previous working directory. This is used by **cd –**.

PWD Contains the pathname of the current working directory. This is set by **cd** after changing to that directory.

Localization

cd uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Note

cd is a built-in shell command.

Exit Values

0 Successful completion

1 Failure due to any of the following:

- No **HOME** directory
- No previous directory
- A search for *directory* failed
- An *old*-to-*new* substitution failed

2 An incorrect command-line option

Messages

Possible error messages include:

dir **bad directory**
cd could not locate the target directory. This does not change the working directory.

Restricted
You are using the restricted version of the shell (for example, by specifying the **–r** option for **sh**). The restricted shell does not allow the **cd** command.

No HOME directory
You have not assigned a value to the **HOME** environment variable. Thus, when you run **cd** in order to return to your home directory, **cd** cannot determine what your home directory is.

No previous directory
You tried the command **cd –** to return to your previous directory; but there is no record of your previous directory.

Pattern *old* **not found in** *dir*
You tried a command of the form **cd** *old new*. However, the name of the working directory *dir* does not contain any string matching the regular expression *old*.

Portability

POSIX.2, X/Open Portability Guide.

All OS/390 UNIX systems feature the first form of the command.

The **cd** *old new* form of the command is an extension of the POSIX standard.

Related Information

dirs, **popd**, **pushd**, **set**, **sh**, **tcsh**

---

continue — Skip to the next iteration of a loop in a shell script

Format

continue [*n*]

Description

continue skips to the next iteration of an enclosing **for**, **select**, **until**, or **while** loop in a shell script. If a number *n* is given, execution continues at the loop control of the *n*th enclosing loop. The default value of *n* is 1.

Usage Note

continue is a special built-in shell command.

Localization

continue uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

0 Successful completion
1 The value of *n* given was not an unsigned decimal greater than 0.

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

break, **sh**, **tcsh**

---

echo — Write arguments to standard output

Format

echo *argument ...*

tcsh shell: **echo** [**-n**] *word...*

Description

echo writes its arguments, specified with the *argument* argument, to standard output. **echo** accepts these C-style escape sequences:

\a	Bell
\b	Backspace
\c	Removes any following characters, including **\n** and **\r**.
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\0*num*	The byte with the numeric value specified by the zero to three-digit octal *num*.
\–	Backslash

echo follows the final argument with a newline unless it finds \c in the arguments. Arguments are subject to standard argument manipulation.

echo in the tcsh shell

In the tcsh shell, **echo** writes each word to the shell's standard output, separated by spaces and terminated with a newline.

tcsh **echo** accepts these C-style escape sequences:

\a	Bell

\b	Backspace
\e	Escape
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
nnn	The EBCDIC character corresponding to the octal number *nnn*

See “tcsh — Invoke a C shell” on page 127.

Examples

1. One important use of **echo** is to expand filenames on the command line, as in:

   ```
   echo *.[ch]
   ```

   This displays the names of all files with names ending in **.c** or **.h**—typically C source and include (header) files. **echo** displays the names on a single line. If there are no filenames in the working directory that end in **.c** or **.h**, **echo** simply displays the string ***.[ch]**.

2. **echo** is also convenient for passing small amounts of input to a filter or a file:

   ```
   echo 'this is\nreal handy' > testfile
   ```

Usage Note

echo is a built-in shell command.

Localization

echo uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **LC_SYNTAX**
- **NLSPATH**

Exit Value

echo always returns the following exit status value:

`0` Successful completion

Portability

POSIX.2, X/Open Portability Guide, UNIX System V.

The POSIX.2 standard does not include escape sequences, so a strictly conforming application cannot use them. **printf** is suggested as a replacement.

Related Information

sh, **tcsh**

eval — Construct a command by concatenating arguments

Format

eval [*argument ...*]

tcsh shell: **eval** *argument ...*

Description

The shell evaluates each argument as it would for any command. **eval** then concatenates the resulting strings, separated by spaces, and evaluates and executes this string in the current shell environment.

eval in the tcsh shell

In the tcsh shell, **eval** treats the arguments as input to the shell and executes the resulting command(s) in the context of the current shell. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions. See “tcsh — Invoke a C shell” on page 127.

Examples

The command:

```
for a in 1 2 3
do
        eval x$a=fred
done
```

sets variables *x1*, *x2*, and *x3* to `fred`. Then:

```
echo $x1 $x2 $x3
```

produces: fred fred fred

Usage Note

eval is a special built-in shell command.

Localization

eval uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Value

The only possible exit status value is:

`0` You specified no arguments or the specified arguments were empty strings

Otherwise, the exit status of **eval** is the exit status of the command that **eval** runs.

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

exec, **sh**, **tcsh**

---

exec — Run a command and open, close, or copy the file descriptors

Format

exec [*command_line*]

tcsh shell: **exec** *command*

Description

The *command_line* argument for **exec** specifies a command line for another command. **exec** runs this command without creating a new process. Some people picture this action as *overlaying* the command on top of the currently running shell. Thus, when the command exits, control returns to the parent of the shell.

Input and output redirections are valid in *command_line*. You can change the input and output descriptors of the shell by giving only input and output redirections in the command. For example:

```
exec 2>errors
```

redirects the standard error stream to **errors** in all subsequent commands ran by the shell.

If you do not specify *command_line*, **exec** returns a successful exit status.

exec in the tcsh shell

In the tcsh shell, **exec** executes the specified command in place of the currrent shell. See “tcsh — Invoke a C shell” on page 127.

Usage Note

exec is a special built-in shell command.

Localization

exec uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

If you specify *command_line*, **exec** does not return to the shell. Instead, the shell exits with the exit status of *command_line* or one of the following exit status values:

1–125 A redirection error occurred.

126 The command in *command_line* was found, but it was not an executable utility.

127 The given *command_line* could not be run because the command could not be found in the current **PATH** environment.

If you did not specify *command_line*, **exec** returns with an exit value of zero.

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

sh, **tcsh**

---

exit — Return to the shell's parent process or to TSO/E

Format

exit [*expression*]

tcsh shell: **exit** [*expr*]

Description

exit ends the shell. If there is an *expression*, the value of the *expression* is the exit status of the shell.

The value of expression should be between 0 and 255. The **EXIT** trap is raised by the **exit** command, unless **exit** is being called from inside an **EXIT** trap.

If you have a shell background job running, you cannot exit from the shell until it completes. However, you can switch to subcommand mode and exit.

exit in the tcsh shell

The shell exits either with the value of the specified expression or, without expression, with the value of the **status** variable. The value of expression should be between 0 and 255. See “tcsh — Invoke a C shell” on page 127.

Usage Note

exit is a special built-in shell command.

Localization

exit uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

exit returns the value of the arithmetic expression specified by the *expression* argument to the parent process as the exit status of the shell. If you omit *expression*, **exit** returns the exit status of the last command run.

Related Information

return, **sh**, **tcsh**

The **exit()** ANSI C function, the **_exit** callable service, and the **_exit()** POSIX C function are unrelated to the **exit** shell command.

---

fg — Bring a job into the foreground

Format

fg [**%***job-identifier*]

tcsh shell: **fg** [*%job ...*]

Description

fg restarts a suspended job or moves a job from the background to the foreground. To identify the job, you give a *job-identifier* (preceded by **%**) as given by the **jobs** command.

If you do not specify *job-identifier*, **fg** uses the most recent job to be suspended (with the **kill** command) or placed in the background (with the **bg** command). **fg** is available only if you have enabled job control. See the **–m** option of **set** for more information.

fg in the tcsh shell

In the tcsh shell, **fg** brings the specified jobs (or, without arguments, the current job) into the foreground, continuing each if it is stopped. *job* may be ", %, +, –, a number, or a string. See also the **run-fg-editor** editor command described in “tcsh — Invoke a C shell” on page 127.

Localization

fg uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

0 Successful completion

>0
No current job

Messages

Possible error messages include:

Not a stopped job Job was not stopped.

Portability

POSIX.2 User Portability Extension.

Related Information

bg, **jobs**, **kill**, **ps**, **tcsh**

---

history — Display a command history list

Format

history [*first*[*last*]]

tcsh shell:

history [**-hTr**] [*n*]

history -Sl-Ll-M [*filename*]

history -c

Description

history is an alias for **fc –l**. Like **fc –l**, **history** displays the list of commands that have been input to an interactive shell. This command does not edit or reenter the commands. If you omit *last*, **history** displays all commands from the one indicated by *first* through to the previous command entered. If you omit both *first* and *last* with this command, the default command range is the 16 most recently entered commands.

history in the tcsh shell

In the tcsh shell, **history**, used alone, prints the history event list. If *n* is given only the *n* most recent events are printed or saved.

Note: See “tcsh — Invoke a C shell” on page 127 for descriptions of the tcsh shell variables and commands indicated below.

The tcsh shell **history** built-in command uses the following options:

- With **-h**, the history list is printed without leading numbers.
- With **-T**, timestamps are printed also in comment form. (This can be used to produce files suitable for loading with **history -L** or **source -h**.)
- With **-r**, the order of printing is most recent first rather than oldest first.

- With **-S**, **history** saves the history list to *filename*. If the first word of the **savehist** shell variable is set to a number, at most that many lines are saved. If the second word of **savehist** is set to merge, the history list is merged with the existing history file instead of replacing it (if there is one) and sorted by time stamp. Merging is intended for an environment like the X Window System with several shells in simultaneous use. Currently it only succeeds when the shells quit one after another.
- With **-L**, the shell appends *filename*, which is presumably a history list saved by the **-S** option or the **savehist** mechanism, to the history list. **-M** is like **-L**, but the contents of *filename* are merged into the history list and sorted by timestamp. In either case, **histfile** is used if filename is not given and **~/.history** is used if histfile is unset. **history -L** is exactly like **source -h** except that it does not require a filename.
- With **-c**, clears the history list.

tcsh login shells do the equivalent of **history -L** on startup and, if **savehist** is set, **history -S** before exiting. Because only **~/.tcshrc** is normally sourced before **~/.history**, **histfile** should be set in **~/.tcshrc** rather than **~/.login**. If **histlit** is set, the first form (**history** [**-hTr**] [*n*]) and second form (**history -S|-L|-M** [*filename*]) print and save the literal (unexpanded) form of the history list.

Related Information

fc, **sh**, **tcsh**

---

jobs — Return the status of jobs in the current session

Format

jobs [**–l**|**–p**] [*job-identifier...*]

tcsh shell: **jobs** [**-l**]

Description

jobs produces a list of the processes in the current session. Each such process is numbered for easy identification by **fg** or **kill**, and is described by a line of information:

[job-identifier] default state shell_command

job-identifier
: Is a decimal number that identifies the process for such commands as **fg** and **kill** (preface *job-identifier* with **%** when used with these commands).

default
: Identifies the process that would be the default for the **fg** and **bg** commands (that is, the most recently suspended process). If *default* is a **+**, this process is the default job. If *default* is a **–**, this job becomes the default when the current default job exits. There is at most one **+** job and one **–** job.

state Shows a job as:

`Running` If it is not suspended and has not exited

`Done`	If it exited successfully
`Done(exit status)`	If it exited with a nonzero exit status
`Stopped` **(***signal***)**	If it is suspended; *signal* is the signal that suspended the job

shell_command
Is the associated shell command that created the process.

jobs in the tcsh shell

In the tcsh shell, **jobs** lists the active jobs. With **-l**, lists process IDs in addition to the normal information. See "tcsh — Invoke a C shell" on page 127.

Options

–l Displays the process group ID of a job (before *state*).
–p Displays the process IDs of all processes.

The **–l** and **–p** options are mutually exclusive.

Localization

jobs uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Note

jobs is a built-in shell command.

Exit Values

`0` Successful completion
`2` Failure due to an incorrect command-line argument

Portability

POSIX.2 User Portability Extension.

Related Information

bg, **fg**, **kill**, **ps**, **wait**, **tcsh**

---

kill — End a process or job, or send it a signal

Format

kill –l [*exit_status*]
kill [**–s** *signal_name*] [*pid ...*] [*job-identifier ...*]
kill [**–***signal_name*] [*pid ...*] [*job-identifier ...*]
kill [**–***signal_number*] [*pid ...*] [*job-identifier ...*]

tcsh shell:

kill [*-signal*] *%job|pid ...*

kill -l

Description

kill ends a process by sending it a signal. The default signal is **SIGTERM**.

kill in the tcsh shell

In the tcsh shell, **kill** [*-signal*] *%job|pid ...* sends the specified *signal* (or if none is given, the TERM (terminate) signal) to the specified jobs or processes. *job* may be a number, a string, '', %, + or - . Signals are either given by number or by name. When using the tcsh **kill** command, do not use the first three characters (*SIG*) of the *signal_name*. Enter the *signal_name* with uppercase characters. For example, if you want to send the **SIGTERM** signal, you would enter **kill -TERM** *pid* not **kill -SIGTERM** *pid*.

There is no default *job*. Specifying **kill** alone does not send a signal to the current job. If the signal being sent is TERM or HUP (hangup), then the job or process is sent a CONT (continue) signal as well.

kill -l lists the signal names. See “tcsh — Invoke a C shell” on page 127.

The *signal_numbers* and *signal_names* described in “Options” are also used with the tcsh **kill** command.

Options

–l Displays the names of all supported signals. If you specify *exit_status*, and it is the exit code of a ended process, **kill** displays the ending signal of that process.

–s *signal_name*
Sends the signal *signal_name* to the process instead of the **SIGTERM** signal. When using the **kill** command, do not use the first three characters (*SIG*) of the *signal_name*. Enter the *signal_name* with uppercase characters. For example, if you want to send the **SIGABRT** signal, enter:

```
kill –s ABRT pid
```

–*signal_name*
(Obsolete.) Same as **–s** *signal_name*.

–*signal_number*
(Obsolete.) A non-negative integer representing the signal to be sent to the process, instead of **SIGTERM**.

The *signal_number* represents the *signal_name* shown below:

signal_number	***signal_name***
0	**SIGNULL**
1	**SIGHUP**
2	**SIGINT**
3	**SIGQUIT**
4	**SIGILL**
5	**SIGPOLL**
6	**SIGABRT**
7	**SIGSTOP**

8	**SIGFPE**
9	**SIGKILL**
10	**SIGBUS**
11	**SIGSEGV**
12	**SIGSYS**
13	**SIGPIPE**
14	**SIGALRM**
15	**SIGTERM**
16	**SIGUSR1**
17	**SIGUSR2**
18	**SIGABND**
19	**SIGCONT**
20	**SIGCHLD**
21	**SIGTTIN**
22	**SIGTTOU**
23	**SIGIO**
24	**SIGQUIT**
25	**SIGTSTP**
26	**SIGTRAP**
27	**SIGIOERR**
28	**SIGWINCH**
29	**SIGXCPU**
30	**SIGXFSZ**
31	**SIGVTALRM**
32	**SIGPROF**
38	**SIGDCE**
39	**SIGDUMP**

Note: The *signal_number*s (3 and 6) associated with **SIGQUIT** and **SIGABRT**, respectively, differ from the values of **SIGQUIT** and **SIGABRT** used by the OS/390 kernel, but they are supported for compatibility with other UNIX platforms. (The **kill** command will send the OS/390 **SIGQUIT** or **SIGABRT** to the process.) (This note is also true for **kill** in the tcsh shell) .

Options

job-identifier

Is the job identifier reported by the shell when a process is started with **&**. It is one way to identify a process. It is also reported by the **jobs** command. When using the job identifier with the **kill** command, the job identifier must be prefaced with a percent (%) sign. For example, if the job identifier is 2, the **kill** command would be entered as follows:

```
kill -s KILL %2
```

pid Is the process ID that the shell reports when a process is started with **&**. You can also find it using the **ps** command. The *pid* argument is a number that may be specified as octal, decimal, or hex. Process IDs are reported in decimal. **kill** supports negative values for *pid*.

If *pid* is negative but not -1, the signal is sent to all processes whose process group ID is equal to the absolute value of *pid*. The negative *pid* is specified in this way:

```
kill -KILL -- -nn
```

where *nn* is the process group ID and may have a range of 2 to 7 digits (*nn* to *nnnnnnn*).

```
kill –s KILL —— –9812753
```

The format must include the — before the *–nn* in order to specify the process group ID.

If *pid* is `0`, the signal is sent to all processes in the process group of the invoker.

The process to be killed must belong to the current user, unless he or she is the superuser.

Localization

kill uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Notes

kill is a built-in shell command.

Exit Values

`0` Successful completion

`1` Failure due to one of the following:

- The job or process did not exist
- There was an error in command-line syntax

`2` Failure due to one of the following:

- Two jobs or processes did not exist
- Incorrect command-line argument
- Incorrect signal

`>2`
Tells the number of processes that could not be killed

Messages

Possible error messages include:

job-identifier **is not a job**
You specified an incorrect ID.

signal_name **is not a valid signal**
You specified a noninteger signal for **kill**, or you specified a signal that is outside the range of valid signal numbers.

Portability

POSIX.2, X/Open Portability Guide.

Related Information

jobs, **ps**, **sh**, **tcsh**

---

newgrp — Change to a new group

Format

newgrp [**–l**] [*group*]
newgrp [**–**] [*group*]

tcsh shell: **newgrp** [-] *group*

Description

newgrp lets you change to a new group. You stay logged in and your working directory does not change, but access permissions are calculated according to your new real and effective group IDs. If an error occurs, your session may be ended, and you must log in again.

After the group IDs are changed, a new shell is initialized within the existing process, effectively overlaying the current shell from which **newgrp** was invoked. The new shell is determined from the initial program value of the OMVS segment of your user profile.

newgrp does not change the value of exported shell variables, and all others are either set to their default or are unset.

If you did not specify any arguments on the command line, **newgrp** changes to the default group specified for your user ID in the system user database. It also sets the list of supplementary groups to that set in the systems group database.

If you specify a group, **newgrp** changes your real and effective group ID to that group. You are permitted to change to that group only if you are a member of that group, as specified in the system group database.

group can be a group name from the security facility group database, or it can be a numeric group ID. If a numeric group exists as a group name in the group data base, the group ID number associated with that group is used.

On systems where the supplementary group list also contains the new effective group ID or where the previous effective group ID was actually in the supplementary group list:

- If the supplementary group list also contains the new effective group ID, **newgrp** changes the effective group ID.
- If the supplementary group list does not contain the new effective group ID, **newgrp** adds it to the list (if there is room).

On systems where the supplementary group list does not normally contain the effective group ID or where the old effective group ID was not in the supplementary group list:

- If the supplementary group list contains the new effective group ID, **newgrp** removes it from the list.

- If the supplementary group list does not contain the old effective group ID, **newgrp** adds it to the list (if there is room).

newgrp in the tcsh shell

newgrp in the tcsh shell, as in the OS/390 shell, allows you to change to a new group.

Options

–l Starts the new shell session as a login session. This implies that it can run any shell profile code.

– Is the obsolescent version of **–l**.

Localization

newgrp uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Notes

newgrp is not supported from an address space running multiple processes because it would cause all processes in the address space to have their security environment changed unexpectedly. If you are using the OMVS interface, you must be using the NOSHAREAS parameter before you issue the **newgrp** command. Also, if you are running in an environment with the **_BPX_SHAREAS** environment variable set to YES, you must unset it and start a new shell before issuing **newgrp**. For example:

```
unset _BPX_SHAREAS; sh
```

Exit Values

If **newgrp** succeeds, its exit status is that of the shell. Otherwise, the exit status is:

`>0`
Failure because **newgrp** could not obtain the proper user or group information or because it could not run the shell, and it ends the current shell.

Portability

POSIX.2 User Portability Extension, UNIX systems.

Related Information

export, **fc**, **sh**, **tcsh**

nice — Run a command at a different priority

Format

nice [**–n** *number*] *command-line* **nice** [**–***number*] *command-line*

tcsh shell: **nice** [*+number*] [*command*]

Description

nice runs a command at a different priority than usual. Normally, **nice** lowers the current priority by 10.

The *command-line* must invoke a single utility command, without using compound commands, pipelines, command substitution, and other special structures.

nice in the tcsh shell

In the tcsh shell, **nice** sets the scheduling priority for the tcsh shell to *number*, or, without *number*, to `4`. With *command*, **nice** runs *command* at the appropriate priority. The greater the number, the less cpu the process gets. The super-user may specify negative priority by using:

```
nice -number ...
```

command is always executed in a sub-shell, and the restrictions placed on commands in simple if statements apply. See “tcsh — Invoke a C shell” on page 127.

Options

–n *number*
: Lowers the current priority by *number*. On systems supporting higher priorities, a user with appropriate privileges can use **nice** to increase priority by specifying a negative value for *number*. For example,

  ```
  nice –n –3 command
  ```

  runs the command with an increased priority of 3.

–*number*
: Is an obsolescent version of **–n** *number*.

Localization

nice uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

If **nice** invokes the *command-line*, it exits with the exit status returned by *command-line;* otherwise its exit status is one of the following:

1-125 An error occurred in the **nice** utility.
126 **nice** could not invoke *command-line*.
127 **nice** could not find the utility specified in *command-line*.

Portability

POSIX.2 User Portability Extension, X/Open Portability Guide, UNIX systems.

Related Information

nohup, **renice**, **tcsh**

---

nohup — Start a process that is immune to hangups

Format

nohup *command-line*

tcsh shell: **nohup** *command*

Description

nohup invokes a utility program using the given *command-line*. The utility runs normally; however, it ignores the **SIGHUP** signal.

If the standard output is a terminal, **nohup** appends the utility's output to a file named `nohup.out` in the working directory. This file is created if it doesn't already exist; if it can't be created in the working directory, it is created in your home directory.

If the standard error stream is a terminal, **nohup** redirects the utility's error output to the same file as the standard output.

nohup simply runs a program from an executable file. *command-line* cannot contain such special shell constructs as compound commands or pipelines; however, you can use **nohup** to invoke a version of the shell to run such a command line, as in:

`nohup sh –c '`*command*`*ssq.`

where *command* can contain such constructs.

nohup in the tcsh shell

With *command*, **nohup** runs command such that it will ignore hangup signals. Commands may set their own response to hangups, overriding **nohup**. Without an argument (allowed only in a shell script), **nohup** causes the tcsh shell to ignore hangups for the remainder of the script. See “tcsh — Invoke a C shell” on page 127.

Localization

nohup uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

126 **nohup** found the utility program but could not invoke it.

127 An error occurred before **nohup** invoked the utility, or **nohup** could not find the utility program.

Otherwise, the exit status is the exit status of the utility program that is invoked.

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

exec, **hup**, **nice**, **sh**, **tcsh**

---

printenv — Display the values of environment variables

Format

printenv [**name**]

tcsh shell: **printenv** [**name**]

Description

The **printenv** command displays the values of environment variables. If the *name* argument is specified, only the value associated with *name* is printed. If it is not specified, printenv displays the current environment variables, one *name=value* pair per line.

If a *name* argument is specified but is not defined in the environment variable, **printenv** returns exit status `1`; otherwise it returns status `0`.

printenv in the tcsh shell

In the tcsh shell, **printenv** prints the names and values of all environment variables or, with *name*, the value of the environment variable named. See “tcsh — Invoke a C shell” on page 127.

Options

There are no options.

Example

To find the current setting of the **HOME** environment variable, enter:

```
printenv HOME
```

Usage Notes

1. Only one *name* argument can be specified.
2. **printenv SOMENAME** is equivalent to **echo $SOMENAME** for exported variables.
3. **printenv** without any arguments is functionally equivalent to **env** without any arguments.

Exit Values

0 Successful completion
1 Failure due to one of the following:
- More than one environment variable was specified
- An option was specified (**printenv** has no options)

Portability

printenv is compatible with the AIX **printenv** utility.

Related Information

env, **tcsh**

---

set — Set or unset command options and positional parameters

Format

set [±**abCefhiKkLmnpstuvx–**] [±**o**[*flag*]] [±**A***name*][*parameter ...*]

tcsh shell:

1. **set** [**-r**]
2. **set** [**-r**] *name ...*
3. **set** [**-r**] *name=word ...*
4. **set** [**-r**] [**-fl-l**] *name=(wordlist*) *...*
5. **set** *name*[*index*]=*word ...*

Description

Calling **set** without arguments displays the names and values of all shell variables, sorted by name, in the following format:

```
Variable="value"
```

The quoting allows the output to be reinput to the shell using the built-in command **eval**. Arguments of the form **–***option* set each shell flag specified as an option. Similarly, arguments of the form **+***option* turn off each of the shell flags specified as an option. (Contrary to what you might expect, **–** means *on,* and + means *off.*)

Note: All of the **set** options except ±**A**, **–s**, **–**, and **—** are shell flags. Shell flags can also be set on the **sh** command line at invocation.

set in the tcsh shell

tcsh shell: See format section above to view the forms described below.

1. The first form of the command prints the value of all shell variables. Variables which contain more than a single word print as a parenthesized word list.

   Variables which are read-only will only be displayed by using the **-r** option. For forms 2, 3 and 4, if **-r** is specified, the value is set to read-only.

2. The second form sets *name* to the null string.
3. The third form sets *name* to the single *word*.
4. The fourth form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded. If **-f** or **-l** is specified, **set** only unique words keeping their order. **-f** prefers the first occurrence of a word, and **-l** the last.
5. The fifth form sets the *index*'th component of *name* to *word*; this component must already exist.

These arguments can be repeated to set and/or make read-only multiple variables in a single set command. However, variable expansion happens for all arguments before any setting occurs. Also, '=' can be adjacent to both name and word or separated from both by whitespace, but cannot be adjacent to only one or the other. For example:

```
set -r name=word and set -r name = word
```

are allowed, but

```
set -r name= word and set -r name =word
```

are not allowed.

See “tcsh — Invoke a C shell” on page 127.

Options

–a Sets all subsequently defined variables for export.

–b Notifies you when background jobs finish running.

–C Prevents the output redirection operator > from overwriting an existing file. Use the alternate operator >| to force an overwrite.

–e Tells a noninteractive shell to execute the ERR trap and then exit. This flag is disabled when reading profiles.

–f Disables pathname generation.

–h Makes all commands use tracked aliases.

–i Makes the shell interactive.

–K Tells the shell to use KornShell-compatible behavior in any case where the POSIX.2 behavior is different from the behavior specified by the KornShell. For more details, see the **let** and **trap** command descriptions.

–k Allows assignment parameters anywhere on the command line and still includes them in the environment of the command.

–L Makes the shell a login shell. Setting this flag is effective only at shell invocation.

–m Runs each background job in a separate process group and reports on each as they complete.

–n Tells a noninteractive shell to read commands but not run them.

–o *flag*
Sets a shell *flag*. If you do not specify *flag*, this option lists all shell flags that are currently set. *flag* can be one of the following:

allexport Is the same as the **–a** option.

errexit Is the same as the **–e** option.

bgnice Runs background jobs at a lower priority.

emacs Specifies **emacs**- style inline editor for command entry. See **shedit** for information about the **emacs**editing mode.

gmacs Specifies **gmacs**- style inline editor for command entry. See **shedit** for information about the **gmacs** editing mode.

ignoreeof Tells the shell not to exit when an end-of-file character is entered.

interactive Is the same as the **–i** option.

keyword Is the same as the **–k** option.

korn Is the same as the **–K** option.

logical Specifies that **cd, pwd** and the **PWD** variable use logical pathnames in directories with symbolic links. If this flag is not set, these built-ins and **PWD** use physical directory pathnames. For example, assume **/usr/spool** is a symbolic link to **/var/spool**, and that it is your current directory. If **logical** is not set, **PWD** has the value **/var/spool**, and **cd** changes the current directory to **/var**. If **logical** is set, **PWD** has the value **/usr/spool** and **cd** changes the current directory to **/usr**.

login Is the same as the **–L** option of **sh**.

markdirs Adds a trailing slash (/) to filename-generated directories.

monitor Is the same as the **–m** option.

noclobber Is the same as the **–C** option.

noexec Is the same as the **–n** option.

noglob Is the same as the **–f** option.

nolog Does not record function definitions in the history file.

notify Is the same as the **–b** option.

nounset Is the same as the **–u** option.

privileged Is the same as the **–p** option.

trackall Is the same as the **–h** option.

verbose Is the same as the **–v** option.

xtrace Is the same as the **–x** option.

vi Specifies **vi**- style inline editor. See **shedit** for information about the **vi**editing mode.

warnstopped
Tells the shell to issue a warning, but not to exit, when there are stopped jobs.

–p Resets the **PATH** variable to the default value, disables processing of **$HOME/.profile**, and ignores the value of the **ENV** variable.

–s Sorts the positional parameters.

–t Exits after reading and running one command.

–u Tells the shell to issue an error message if an unset parameter is used in a substitution.

–v Prints shell input lines as they are read.

–x Prints commands and their arguments as they run.

Other options:

– Turns off the **–v** and **–x** options. Also, parameters that follow this option do not set shell flags, but are assigned to positional parameters (see **sh**).

— Specifies that parameters following this option do not set shell flags, but are assigned to positional parameters.

+A *name*
Assigns the parameter list to the elements of *name*, starting at *name*[0].

–A *name*
Unsets *name* and then assigns the parameter list to the elements of *name* starting at *name*[0].

Usage Notes

set is a special built-in shell command.

Localization

set uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

0 Successful completion
1 Failure due to an incorrect command-line argument
2 Failure resulting in a usage message, usually due to a missing argument

Portability

POSIX.2, X/Open Portability Guide.

Several shell flags are extensions of the POSIX standard: **bgnice**, **ignoreeof**, **keyword**, **markdirs**, **monitor**, **noglob**, **nolog**, **privileged**, and **trackall** are extensions of the POSIX standard, along with the shell flags ±**A**, ±**h**, ±**k**, ±**p**, ±**s**, and ±**t**.

Related Information

alias, **eval**, **export**, **sh**, **trap**, **typeset**, **shedit**, **tcsh**

---

shift — Shift positional parameters

Format

shift [*expression*]

tcsh shell: **shift** [*variable*]

Description

Note: **shift** can be used in all OS/390 shells (**/bin/sh** and tcsh).

shift renames the positional parameters so that *i+n*th positional parameter becomes the *i*th positional parameter, where *n* is the value of the given arithmetic *expression*. If you omit *expression*, the default value is 1. The value of *expression* must be between zero and the number of positional parameters (**$#**), inclusive. The value of **$#** is updated.

shift in the tcsh shell

Without arguments, **shift** discards **argv[1]** and shifts the members of **argv** to the left. It is an error for **argv** not to be set or to have less than one word as value. With *variable*, **shift** performs the same function on *variable*. See “tcsh — Invoke a C shell” on page 127.

Examples

The commands:

```
set a b c d
shift 2
echo $*
```

produce:

```
c d
```

Usage Note

shift is a special built-in shell command.

Localization

shift uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

0 Successful completion

1 Failure because the *expression* had a negative value or was greater than the number of positional parameters.

Messages

Possible error messages include:

bad shift count *expr*
You specified an expression that did not evaluate to a number in the range from 0 to the number of remaining positional parameters.

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Allowing an expression, rather than just a number, is an extension found in the OS/390 UNIX System Services shell (a KornShell).

Related Information

set, **sh**, **tcsh**

---

stop — Suspend a process or job

Format

stop [*pid ...*] [*job—identifier ...*]

tcsh shell: **stop** *%job|pid ...*

Description

Note: **stop** can be used in all OS/390 shells (**/bin/sh** and tcsh).

stop is an alias for **kill –STOP**. Like **kill –STOP**, **stop** sends a **SIGSTOP** to the process you specify.

See “kill — End a process or job, or send it a signal” on page 111 for more information.

stop in the tcsh shell

In the tcsh shell, **stop** stops the specified jobs or processes which are executing in the background. *job* may be a number, a string, ", %, + or - . There is no default *job*. Specifying stop alone does not stop the current job. See “tcsh — Invoke a C shell” on page 127.

Options

job-identifier
Is the job identifier reported by the shell when a process is started with **&**. It is one way to identify a process. It is also reported by the **jobs** command. When using the job identifier with the **stop** command, the job identifier must be

prefaced with a percent (%) sign. For example, if the job identifier is 2, the **stop** command would be entered as follows:

```
stop %2
```

pid Is the process ID that the shell reports when a process is started with **&**. You can also find it using the **ps** command. The *pid* argument is a number that may be specified as octal, decimal, or hex. Process IDs are reported in decimal. **stop** supports negative values for *pid*.

If *pid* is negative but not –1, the signal is sent to all processes whose process group ID is equal to the absolute value of *pid*. The negative *pid* is specified in this way:

```
stop — –nn
```

where *nn* is the process group ID and may have a range of 2 to 7 digits (*nn* to *nnnnnnn*).

```
stop — –9812753
```

The format must include the — before the *–nn* in order to specify the process group ID.

If *pid* is `0`, the signal is sent to all processes in the process group of the invoker.

The process to be killed must belong to the current user, unless he or she is the superuser.

Related Information

kill, **jobs**, **sh**, **suspend**, **tcsh**

---

suspend — Send a SIGSTOP to the current shell

Format

suspend

tcsh shell: same as above

Description

suspend is an alias for **stop $$**, where **stop** is an alias of **kill –STOP** and **$$** expands to the current process of the shell. **suspend** sends a **SIGSTOP** to the current shell.

See “kill — End a process or job, or send it a signal” on page 111 for more information.

suspend in the tcsh shell

suspend causes the tcsh shell to stop in its tracks, much as if it had been sent a stop signal with ^Z. See “tcsh — Invoke a C shell” on page 127.

Related Information

kill, **sh**, **tcsh**

---

tcsh — Invoke a C shell

Format

tcsh [**–bcdeFfimnqstvVxX**]
tcsh –l

Note: **–l** is a lowercase L, not an uppercase i.

Description

tcsh contains the following sections and subsections:

- Options and invocation
- Options
- Editing
- Command syntax
- Substitutions
- Command Execution
- Features
- Jobs
- Status Reporting
- Automatic, Periodic, and Time Events
- Native Language System Report
- Signal Handling
- Built-in Commands
- Shell and Environment Variables
- Files
- Problems and Limitations

Options and Invocation

The tcsh shell is an enhanced but completely compatible version of the Berkeley UNIX C shell, csh. It is a command language interpreter usable both as an interactive login shell and a shell script command processor. It includes a command-line editor, programmable word completion, spelling correction, a history mechanism, job control, and a C-like syntax.

You can invoke the shell by typing an explicit **tcsh** command. A login shell can also be specified by invoking the shell with the **–l** option as the only argument.

A login shell begins by executing commands from the system files **/etc/csh.cshrc** and **/etc/csh.login**. It then executes commands from files in the user's home directory: first **~/.tcshrc**, then **~/.history** (or the value of the **histfile** shell variable), then **~/.login**, and finally **~/.cshdirs** (or the value of the **dirsfile** shell variable). The shell reads **/etc/csh.login** after **/etc/csh.cshrc**.

Non-login shells read only **/etc/csh.cshrc** and **~/.tcshrc** or **~/.cshrc** on invocation.

Commands like **stty**, which need be run only once per login, usually go in the user's **~/.login** file.

In the normal case, the shell begins reading commands from the terminal, prompting with >. The shell repeatedly reads a line of command input, breaks it into words, places it on the command history list, and then parses and executes each command in the line. See “Command Execution” on page 146.

A user can log out of a tcsh shell session by typing ^D, **logout**, or **login** on an empty line (see **ignoreeof** shell variable), or via the shell's autologout mechanism. When a login shell terminates, it sets the **logout** shell variable to *normal* or *automatic* as appropriate, then executes commands from the files **/etc/csh.logout** and **~/.logout**.

Note: The names of the system login and logout files vary from system to system for compatibility with different csh variants; see “tcsh Files” on page 170.

Options

If the first argument (argument 0) to the tcsh shell is - (hyphen), then it is a login shell. You can also specify the login shell by invoking the tcsh shell with the **–l** as the only argument.

The OS/390 UNIX System Services tcsh shell accepts the following options on the command line:

–b Forces a break from option processing, causing any further shell arguments to be treated as non-option arguments. The remaining arguments will not be interpreted as shell options. This may be used to pass options to a shell script without confusion or possible subterfuge.

–c Reads and executes commands stored in the command shell (this option must be present and must be a single arugment). Any remaining arguments are placed in the **argv** shell variable.

–d Loads the directory stack from **~/.cshdirs** as described under “Options and Invocation” on page 127, whether or not it is a login shell.

–e Terminates shell if any invoked command terminates abnormally or yields a non-zero exit status.

–i Invokes an interactive shell and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

–l Invokes a login shell. Only applicable if **–l** is the only option specified.

Note: **–l** is a lower-case L not an upper-case i.

–m Loads **~/.tcshrc** even if it does not belong to the effective user.

–n Parses commands but does not execute them. This aids in debugging shell scripts.

–q Accepts SIGQUIT and behaves when it is used under a debugger. Job control is disabled. (u)

–s Take command input from the standard input.

–t Reads and executes a single line of input. A \ (backslash) may be used to escape the newline at the end of this line and continue onto another line.

–v Sets the **verbose** shell variable so command input is echoed after history substitution.

–x Sets the **echo** shell variable so commands are echoed immediately before execution.

–V Sets the **verbose** shell variable even before executing **~/.tcshrc**.

–X Is to **–x** as **–V** is to **–v**.

After processing of option arguments, if arguments remain but none of the **–c**, **–i**, **–s**, or **–t** were given, the first argument is taken as the name of a file of commands, or script , to be executed. The shell opens this file and saves its name for possible resubstitution by $0. Since many systems use shells whose shell scripts are not compatible with this shell, the tcsh shell uses such a **standard** shell to execute a script whose character is not a #, that is, which does not start with a comment.

Remaining arguments are placed in the **argv** shell variable.

tcsh shell Editing

In this section, we first describe the Command-Line Editor. We then discuss Completion and Listing and Spelling Correction which describe two sets of functionality that are implemented as editor commands but which deserve their own treatment. Finally, the Editor Commands section lists and describes the editor commands specific to the tcsh shell and their default bindings.

tcsh shell Command-Line Editor

Command-line input can be edited using key sequences much like those used in GNU Emacs or vi. The editor is active only when the edit shell variable is set, which it is by default in interactive shells. The **bindkey** built-in command can display and change key bindings. Emacs-style key bindings are used by default, but **bindkey** can change the key bindings to vi-style bindings.

The shell always binds the arrow keys to:

down down-history

up up-history

left backward-char

right forward-char

unless doing so would alter another single-character binding. One can set the arrow key escape sequences to the empty string with **settc** to prevent these bindings.

Other key bindings are, for the most part, what Emacs and vi users would expect and can easily be displayed by **bindkey**, so there is no need to list them here. Likewise, **bindkey** can list the editor commands with a short description of each.

Note: Editor commands do not have the same notion of a *word* as does the tcsh shell. The editor delimits words with any non-alphanumeric characters not in the shell variable wordchars, while the tcsh shell recognizes only whitespace and some of the characters with special meanings to it, listed under “Command Syntax” on page 137.

Completion and Listing

The tcsh shell is often able to complete words when given a unique abbreviation. Type part of a word (for example **ls** */usr/lost*) and press the tab key to run the **complete-word** editor command. The shell completes the filename **/usr/lost** to **/usr/lost+found/**, replacing the incomplete word with the complete word in the input buffer. (Note the terminal / (forward slash); completion adds a / to the end of completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion. The **addsuffix** shell variable can be unset to prevent this.) If no match is found (for example, **/usr/lost+found** doesn't exist), the terminal bell rings. If the word is already complete (for example, there is a **/usr/lost** on your system, or you were thinking too far ahead and typed the whole thing) a / or space is added to the end if it isn't already there.

Completion works anywhere in the line, not just at the end; completed text pushes the rest of the line to the right. Completion in the middle of a word often results in leftover characters to the right of the cursor which need to be deleted.

Commands and variables can be completed in much the same way. For example, typing **em** *[tab]* would complete 'em' to 'emacs' if **emacs** were the only command on your system beginning with 'em'. Completion can find a command in any directory in the path or if given a full pathname. Typing **echo** *$ar[tab]* would complete '$ar' to '$argv' if no other variable began with 'ar'.

The shell parses the input buffer to determine whether the word you want to complete should be completed as a filename, command or variable. The first word in the buffer and the first word following ';', '|', '|&', '&&' or '||' is considered to be a command. A word beginning with '$' is considered to be a variable. Anything else is a filename. An empty line is **completed** as a filename.

You can list the possible completions of a word at any time by typing ^D to run the **delete-char-or-list-or-eof** editor command. The tcsh shell lists the possible completions using the **ls-F** built-in and reprints the prompt and unfinished command line, for example:

```
> ls /usr/l['^D]
lbin/ lib/ local/ lost+found/
> ls /usr/l
```

If the **autolist** shell variable is set, the tcsh shell lists the remaining choices (if any) whenever completion fails:

```
> set autolist
> nm /usr/lib/libt[tab]
libtermcap.a@ libtermlib.a@
> nm /usr/lib/libterm
```

If **autolist** is set to *ambiguous*, choices are listed only if multiple matches are possible, and if the completion adds no new characters to the name to be matched.

A filename to be completed can contain variables, your own or others' home directories abbreviated with ~ (tilde; see “Filename Substitution” on page 144) and directory stack entries abbreviated with = (equal; see “Directory Stack Substitution” on page 145). For example:

```
> ls ~k[^D]
kahn kas kellogg
> ls ~ke[tab]
> ls ~kellogg/

or

> set local = /usr/local
> ls $lo[tab]
> ls $local/[^D]
bin/ etc/ lib/ man/ src/
> ls $local/
```

Variables can also be expanded explicitly with the **expand-variables** editor command.

delete-char-or-list-or-eof only lists at the end of the line; in the middle of a line it deletes the character under the cursor and on an empty line it logs one out or, if **ignoreeof** is set, does nothing. M-^D, bound to the editor command **list-choices**, lists completion possibilities anywhere on a line, and **list-choices** (or any one of the related editor commands which do or don't delete, list and/or log out, listed under **delete-char-or-list-or-eof**) can be bound to ^D with the **bindkey** built-in command if so desired.

The **complete-word-fwd** and **complete-word-back** editor commands (not bound to any keys by default) can be used to cycle up and down through the list of possible completions, replacing the current word with the next or previous word in the list.

The tcsh shell variable **fignore** can be set to a list of suffixes to be ignored by completion. Consider the following:

```
> ls
Makefile condiments.h~ main.o side.c
README main.c meal side.o
condiments.h main.c~
> set fignore = (.o \~)
> emacs ma[^D]
main.c main.c~ main.o
> emacs ma[tab]
> emacs main.c
```

'main.c~' and 'main.o' are ignored by completion (but not listing), because they end in suffixes in **fignore**. \ is needed in front of ~ to prevent it from being expanded to **home** as described under “Filename Substitution” on page 144. **fignore** is ignored if only one completion is possible.

If the **complete** shell variable is set to *enhance*, completion: 1.) ignores case and 2.) considers periods, hyphens and underscores ('.', '-' and '_') to be word separators and hyphens and underscores to be equivalent.

If you had the following files:

```
comp.lang.c comp.lang.perl comp.std.c++
comp.lang.c++ comp.std.c
```

and typed **mail -f** *c.l.c[tab]*, it would be completed to **mail -f** *comp.lang.c*, and ^D would list comp.lang.c and comp.lang.c++. **mail -f** *c..c++[^D]* would list comp.lang.c++ and comp.std.c++. Typing **rm** *a--file[^D]* in the following directory

```
A_silly_file a-hyphenated-file another_silly_file
```

would list all three files, because case is ignored and hyphens and underscores are equivalent. Periods, however, are not equivalent to hyphens or underscores.

Completion and listing are affected by several other tcsh shell variables: **recexact** can be set to complete on the shortest possible unique match, even if more typing might result in a longer match. For example:

```
> ls
fodder foo food foonly
> set recexact
> rm fo[tab]
```

just beeps, because 'fo' could expand to 'fod' or 'foo', but if we type another 'o',

```
> rm foo[tab]
> rm foo
```

the completion completes on 'foo', even though 'food' and 'foonly' also match. **autoexpand** can be set to run the **expand-history** editor command before each completion attempt, and **correct** can be set to complete commands automatically after one hits 'return'. **matchbeep** can be set to make completion beep or not beep in a variety of situations, and **nobeep** can be set to never beep at all. **nostat** can be set to a list of directories and/or patterns which match directories to prevent the completion mechanism from stat(2)ing those directories.

Note: The completion operation succeeds, but faster. The setting of **nostat** is evident when using the **listflags** variable. For example:

```
>set listflags=x
>ls-F /u/pluto
Dir1/exe1*
>set nostat=(/u/pluto/)
>ls-F /u/pluto
Dir1exe1
>
```

Although, you must be careful when setting **nostat** to keep the trailing / (forward slash).

listmax and **listmaxrows** can be set to limit the number of items and rows (respectively) that are listed without asking first. **recognize_only_executables** can be set to make the shell list only executables when listing commands, but it is quite slow.

Finally, the **complete** built-in command can be used to tell the shell how to complete words other than filenames, commands and variables. Completion and listing do not work on glob-patterns (see “Filename Substitution” on page 144), but the **list-glob** and **expand-glob** editor commands perform equivalent functions for glob-patterns.

Spelling Correction

The tcsh shell can sometimes correct the spelling of filenames, commands and variable names as well as completing and listing them.

Individual words can be spelling-corrected with the **spell-word** editor command (usually bound to M-s and M-S where M=Meta Key or escape (ESC) key) and the entire input buffer with **spell-line** (usually bound to M-$). The **correct** shell variable

can be set to 'cmd' to correct the command name or 'all' to correct the entire line each time return is typed.

When spelling correction is invoked in any of these ways and the shell thinks that any part of the command line is misspelled, it prompts with the corrected line:

```
> set correct = cmd
> lz /usr/bin
CORRECT>ls /usr/bin (y|n|e|a)?
```

where one can answer 'y' or space to execute the corrected line, 'e' to leave the uncorrected command in the input buffer, 'a' to abort the command as if ^C had been hit, and anything else to execute the original line unchanged.

Spelling correction recognizes user-defined completions (see the **complete** built-in command). If an input word in a position for which a completion is defined resembles a word in the completion list, spelling correction registers a misspelling and suggests the latter word as a correction. However, if the input word does not match any of the possible completions for that position, spelling correction does not register a misspelling.

Like completion, spelling correction works anywhere in the line, pushing the rest of the line to the right and possibly leaving extra characters to the right of the cursor.

Attention: Spelling correction is not guaranteed to work the way one intends, and is provided mostly as an experimental feature.

Editor Commands

bindkey lists key bindings and **bindkey -l** lists and briefly describes editor commands. Only new or especially interesting editor commands are described here. See **emacs** and **vi** for descriptions of each editor's key bindings.

The character or characters to which each command is bound by default is given in parentheses. ^*character* means a control character and M-*character* a meta character, typed as escape-character on terminals without a meta key. Case counts, but commands which are bound to letters by default are bound to both lower- and uppercase letters for convenience.

complete-word
Completes a word as described under “Completion and Listing” on page 130.

complete-word-back
Like **complete-word-fwd**, but steps up from the end of the list.

complete-word-fwd
Replaces the current word with the first word in the list of possible completions. May be repeated to step down through the list. At the end of the list, beeps and reverts to the incomplete word.

complete-word-raw
Like **complete-word**, but ignores user-defined completions.

copy-prev-word
Copies the previous word in the current line into the input buffer. See also **insert-last-word**.

dabbrev-expand
Expands the current word to the most recent preceding one for which the current is a leading substring, wrapping around the history list (once) if

necessary. Repeating **dabbrev-expand** without any intervening typing changes to the next previous word etc., skipping identical matches much like **history-search-backward** does.

delete-char (not bound)
Deletes the character under the cursor. See also **delete-char-or-list-or-eof**.

delete-char-or-eof (not bound)
Does **delete-char** if there is a character under the cursor or **end-of-file** on an empty file. See also **delete-char-or-list-or-eof**.

delete-char-or-list (not bound)
Does **delete-char** if there is a character under the cursor or list-choices at the end of the line. See also **delete-char-or-list-or-eof**.

delete-char-or-list-or-eof (^D)
Does **delete-char** if there is a character under the cursor, **list-choices** at the end of the line or **end-of-file** on an empty line. See also **delete-char-or-eof**, **delete-char-or-list** and **list-or-eof**.

down-history
Like **up-history**, but steps down, stopping at the original input line.

end-of-file
Signals an end of file, causing the tcsh shell to exit unless the **ignoreeof** shell variable is set to prevent this. See also **delete-char-or-list-or-eof**.

expand-history (M-space)
Expands history substitutions in the current word. See “History Substitution” on page 138. See also **magic-space**, **toggle-literal-history**, and the **autoexpand** shell variable.

expand-glob(^X-*)
Expands the glob-pattern to the left of the cursor. For example:

```
>ls test*[^X-*]
```

would expand to

```
>ls test1.c test2.c
```

if those were the only two files in your directory that begin with 'test'. See “Filename Substitution” on page 144.

expand-line (not bound)
Like **expand-history**, but expands history substitutions in each word in the input buffer.

expand-variables (^X-$)
Expands the variable to the left of the cursor. See “Variable Substitution” on page 142.

history-search-backward (M-p, M-P)
Searches backwards through the history list for a command beginning with the current contents of the input buffer up to the cursor and copies it into the input buffer. The search string may be a glob-pattern (see “Filename Substitution” on page 144) containing '*', '?', '[]' or '{}'. **up-history** and **down-history** will proceed from the appropriate point in the history list. Emacs mode only. See also **history-search-forward** and **i-search-back**.

history-search-forward(M-n, M-N)
Like **history-search-backward**, but searches forward.

i-search-back (not bound)
Searches backward like **history-search-backward**, copies the first match into the input buffer with the cursor positioned at the end of the pattern, and prompts with 'bck: ' and the first match. Additional characters may be typed to extend the search. **i-search-back** may be typed to continue searching with the same pattern, wrapping around the history list if necessary, (**i-search-back** must be bound to a single character for this to work) or one of the following special characters may be typed:

^W Appends the rest of the word under the cursor to the search pattern.

delete (or any character bound to backward-delete-char)
Undoes the effect of the last character and deletes a character from the search pattern if appropriate.

^G If the previous search was successful, aborts the entire search. If not, goes back to the last successful search.

escape
Ends the search, leaving the current line in the input buffer.

Any other character not bound to **self-insert-command** terminates the search, leaving the current line in the input buffer, and is then interpreted as normal input. In particular, a carriage return causes the current line to be executed. Emacs mode only. See also **i-search-fwd** and **history-search-backward**.

i-search-fwd
Like i-search-back, but searches forward.

insert-last-word (M-_)
Inserts the last word of the previous line (!$) into the input buffer. See also **copy-prev-word**.

list-choices (M-D)
Lists completion possibilities as described under “Completion and Listing” on page 130. See also **delete-char-or-list-or-eof**.

list-choices-raw (^X-^D)
Like **list-choices**, but ignores user-defined completions.

list-glob (^X-g, ^X-G)
Lists (via the **ls-F**) matches to the glob-pattern (see “Filename Substitution” on page 144) to the left of the cursor.

list-or-eof (not bound)
Does **list-choices** or **end-of-file** on an empty line. See also **delete-char-or-list-or-eof**.

magic-space (not bound)
Expands history substitutions in the current line, like **expand-history**, and appends a space. **magic-space** is designed to be bound to the spacebar, but is not bound by default.

normalize-command (^X-?)
Searches for the current word in PATH and, if it is found, replaces it with the full path to the executable. Special characters are quoted. Aliases are expanded and quoted but commands within aliases are not. This command is useful with commands which take commands as arguments, for example, **dbx** and **sh -x**.

normalize-path (^X-n, ^X-N)
Expands the current word as described under the *expand* setting of the **symlinks** shell variable.

overwrite-mode (unbound)
Toggles between input and overwrite modes.

run-fg-editor (M-^Z)
Saves the current input line and looks for a stopped job with a name equal to the last component of the file name part of the EDITOR or VISUAL environment variables, or, if neither is set, ed or vi. If such a job is found, it is restarted as if **fg %job** had been typed. This is used to toggle back and forth between an editor and the shell easily. Some people bind this command to ^Z so they can do this even more easily.

run-help (M-h, M-H)
Searches for documentation on the current command, using the same notion of **current command** as the completion routines, and prints it. There is no way to use a pager; **run-help** is designed for short help files. Documentation should be in a file named *command*.help, *command*.1, *command*.6, *command*.8 or *command*, which should be in one of the directories listed in the HPATH enviroment variable. If there is more than one help file only the first is printed.

self-insert-command (text characters)
In insert mode (the default), inserts the typed character into the input line after the character under the cursor. In overwrite mode, replaces the character under the cursor with the typed character. The input mode is normally preserved between lines, but the **inputmode** shell variable can be set to *insert* or *overwrite* to put the editor in that mode at the beginning of each line. See also **overwrite-mode**.

sequence-lead-in (arrow prefix, meta prefix, ^X)
Indicates that the following characters are part of a multi-key sequence. Binding a command to a multi-key sequence really creates two bindings: the first character to **sequence-lead-in** and the whole sequence to the command. All sequences beginning with a character bound to **sequence-lead-in** are effectively bound to **undefined-key** unless bound to another command.

spell-line (M-$)
Attempts to correct the spelling of each word in the input buffer, like **spell-word**, but ignores words whose first character is one of '-', '!', '^' or '%', or which contain '\', '*' or '?', to avoid problems with switches, substitutions and the like. See “Spelling Correction” on page 132.

spell-word (M-s, M-S)
Attempts to correct the spelling of the current word as described under “Spelling Correction” on page 132. Checks each component of a word which appears to be a pathname.

toggle-literal-history (M-r, M-R)
Expands or unexpands history substitutions in the input buffer. See also **expand-history** and the **autoexpand** shell variable.

undefined-key (any unbound key)
Beeps.

up-history (up-arrow, ^P)
Copies the previous entry in the history list into the input buffer. If **histlit** is set, uses the literal form of the entry. May be repeated to step up through the history list, stopping at the top.

vi-search-back (?)
Prompts with ? for a search string (which may be a glob-pattern, as with **history-search-backward**), searches for it and copies it into the input buffer. The bell rings if no match is found. Hitting return ends the search and leaves the last match in the input buffer. Hitting escape ends the search and executes the match. vi mode only.

vi-search-fwd (/)
Like **vi-search-back**, but searches forward.

which-command (M-?)
Does a **which** (built-in command) on the first word of the input buffer.

Command Syntax

The tcsh shell splits input lines into words at blanks and tabs. The special characters '&', '|', ';', '<', '>', '(', and ')' and the doubled characters '&&', '||', '<<' and '>>' are always separate words, whether or not they are surrounded by whitespace.

When the tcsh shell's input is not a terminal, the character '#' is taken to begin a comment. Each # and the rest of the input line on which it appears is discarded before further parsing.

A special character (including a blank or tab) may be prevented from having its special meaning, and possibly made part of another word, by preceding it with a backslash (\) or enclosing it in single ('), double (") or backward (' ` ') quotes. When not otherwise quoted a newline preceded by a \ is equivalent to a blank, but inside quotes this sequence results in a newline.

Furthermore, all substitutions (see “Substitutions” on page 138) except history substitution can be prevented by enclosing the strings (or parts of strings) in which they appear with single quotes or by quoting the crucial character(s) (e.g. '$' or '`' for variable substitution or command substitution respectively) with \. (alias substitution is no exception: quoting in any way any character of a word for which an alias has been defined prevents substitution of the alias. The usual way of quoting an alias is to precede it with a backslash.) History substitution is prevented by backslashes but not by single quotes. Strings quoted with double or backward quotes undergo Variable substitution and Command substitution, but other substitutions are prevented.

Text inside single or double quotes becomes a single word (or part of one). Metacharacters in these strings, including blanks and tabs, do not form separate words. Only in one special case (see “Command Substitution” on page 144) can a double-quoted string yield parts of more than one word; single-quoted strings never do. Backward quotes are special: they signal command substitution, which may result in more than one word.

Quoting complex strings, particularly strings which themselves contain quoting characters, can be confusing. Remember that quotes need not be used as they are in human writing! It may be easier to quote not an entire string, but only those parts

of the string which need quoting, using different types of quoting to do so if appropriate.

The **backslash_quote** shell variable can be set to make backslashes always quote \, ', and ". This may make complex quoting tasks easier, but it can cause syntax errors in csh (or tcsh) scripts.

Substitutions

This section describes the various transformations the tcsh shell performs on input in the order in which they occur. The section will cover data structures involved and the commands and variables which affect them. Remember that substitutions can be prevented by quoting as described under “Command Syntax” on page 137.

History Substitution

Each command, or **event**, input from the terminal is saved in the history list. The previous command is always saved, and the **history** shell variable can be set to a number to save that many commands. The **histdup** shell variable can be set to not save duplicate events or consecutive duplicate events.

Saved commands are numbered sequentially from 1 and stamped with the time. It is not usually necessary to use event numbers, but the current event number can be made part of the prompt by placing an exclamation point (!) in the **prompt** shell variable.

The shell actually saves history in expanded and literal (unexpanded) forms. If the **histlit** shell variable is set, commands that display and store history use the literal form.

The **history** built-in command can print, store in a file, restore and clear the history list at any time, and the **savehist** and **histfile** shell variables can be set to store the history list automatically on logout and restore it on login.

History substitutions introduce words from the history list into the input stream, making it easy to repeat commands, repeat arguments of a previous command in the current command, or fix spelling mistakes in the previous command with little typing and a high degree of confidence.

History substitutions begin with the character !. They may begin anywhere in the input stream, but they do not nest. The ! may be preceded by a \ to prevent its special meaning; for convenience, a ! is passed unchanged when it is followed by a blank, tab, newline, = or (. History substitutions also occur when an input line begins with ^. This special abbreviation will be described later. The characters used to signal history substitution (! and ^) can be changed by setting the **histchars** shell variable. Any input line which contains a history substitution is printed before it is executed.

A history substitution may have an **event specification**, which indicates the event from which words are to be taken, a **word designator**, which selects particular words from the chosen event, and/or a **modifier**, which manipulates the selected words.

An event specification can be

n A number, referring to a particular event

–n An offset, referring to the even *n* before the current event

The current event. This should be used carefully in csf, where there is no check for recursion. tcsh allows 10 levels of recursion.

! The previous event (equivalent to -1)

s The most recent event whose first word begins with the string *s*

?*s*? The most recent event which contains the string *s*. The second ? can be omitted if it is immediately followed by a newline.

For example, consider this bit of someone's history list:

```
9 8:30 nroff -man wumpus.man
10 8:31 cp wumpus.man wumpus.man old
11 8:36 vi wumpus.man
12 8:37 diff wumpus.man.old wumpus.man
```

The commands are shown with their event numbers and time stamps. The current event, which we haven't typed in yet, is event 13. !11 and !-2 refer to event 11. !! refers to the previous event, 12. !! can be abbreviated ! if it is followed by : (colon; described below). !n refers to event 9, which begins with *n*. !?old? also refers to event 12, which contains *old*. Without word designators or modifiers history references simply expand to the entire event, so we might type !cp to redo the copy command or !!|more if the **diff** output scrolled off the top of the screen.

History references may be insulated from the surrounding text with braces if necessary. For example, !vdoc would look for a command beginning with *vdoc*, and, in this example, not find one, but !{v}doc would expand unambiguously to vi wumpus.mandoc. Even in braces, history substitutions do not nest.

While csh expands, for example, !3d to event 3 with the letter d appended to it, tcsh expands it to the last event beginning with 3d; only completely numeric arguments are treated as event numbers. This makes it possible to recall events beginning with numbers. To expand !3d as in csh say !\3d.

To select words from an event we can follow the event specification by a : (colon) and a designator for the desired words. The words of an input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0 The first command word

n The *n*th argument

^ The first argument, equivalent to 1

$ The last argument

% The word matched by an ?*s*? search

x-y A range of words

–y Equivalent to 0–y

***** Equivalent to ^–$, but returns nothing if the event contains only 1 word

x* Equivalent to *x-$*

x- Equivalent to *x**, but omitting the last word ($)

Selected words are inserted into the command line separated by single blanks. For example, the **diff** command in the previous example might have been typed as **diff**

!!:1.old !!:1(using :1 to select the first argument from the previous event) or **diff** *!-2:2 !-2:1*to select and swap the arguments from the **cp** command. If we didn't care about the order of the **diff** we might have said **diff** *!-2:1-2*or simply **diff** *!-2:**. The **cp** command might have been written **cp** *wumpus.man !#:1.old*, using # to refer to the current event. !n:- hurkle.man would reuse the first two words from the **nroff** command to say **nroff -man** *hurkle.man*.

The : separating the event specification from the word designator can be omitted if the argument selector begins with a '^', '$', '*', '%' or '-'. For example, our **diff** command might have been **diff** *!!^.old !!^* or, equivalently, **diff** *!!$.old !!$*. However, if !! is abbreviated !, an argument selector beginning with - (hypen) will be interpreted as an event specification.

A history reference may have a word designator but no event specification. It then references the previous command. Continuing our **diff** example, we could have said simply **diff** *!^.old !^*or, to get the arguments in the opposite order, just **diff** *!**.

The word or words in a history reference can be edited, or **modified**, by following it with one or more modifiers, each preceded by a : (colon):

h Remove a trailing pathname component, leaving the head.

t Remove all leading pathname components, leaving the tail.

r Remove a filename extension .xxx, leaving the root name.

e Remove all but the extenstion

u Uppercase the first lowercase letter.

l Lowercase the first uppercase letter.

s/l/r Substitute *l* for *r*. *l* is simply a string like *r*, not a regular expression as in the eponymous **ed** command. Any character may be used as the delimiter in place of /; a \ can be used to quote the delimiter inside *l* and *r*. The character & in the *r* is replaced by *l*; \ also quotes &. If *l* is empty (""), the *l* from a previous substitution or the *s* from a previous ?*s*? event specification is used. The trailing delimiter may be omitted if it is immediately followed by a newline.

& Repeat the previous substitution

g Apply the following modifier once to each word.

a Apply the following modifier as many times as possible to a single word. 'a' and 'g' can be used together to apply a modifier globally. In the current implementation, using the 'a' and 's' modifiers together can lead to an infinite loop. For example, :as/f/ff/ will never terminate. This behavior might change in the future.

p Print the new command line but do not execute it.

q Quote the substituted words, preventing further substitutions.

x Like **q**, but break into words at blanks, tabs and newlines.

Modifiers are applied only to the first modifiable word (unless 'g' is used). It is an error for no word to be modifiable.

For example, the **diff** command might have been written as **diff** *wumpus.man.old !#^:r*, using :r to remove *.old* from the first argument on the same line (!#^). We could say **echo** *hello out there*, then **echo** *!*:u* to capitalize 'hello', **echo** *!*:au* to say it out loud, or **echo** *!*:agu* to really shout. We might follow **mail -s** *"I forgot my*

password" rot with *!:s/rot/root* to correct the spelling of 'root' (but see “Spelling Correction” on page 132 for a different approach).

There is a special abbreviation for substitutions. ^, when it is the first character on an input line, is equivalent to *!:s^*. Thus, we might have said *^rot^root* to make the spelling correction in the previous example. This is the only history substitution which does not explicitly begin with !.

In csh as such, only one modifier may be applied to each history or variable expansion. In tcsh, more than one may be used, for example

```
% mv wumpus.man /usr/man/man1/wumpus.1
% man !$:t:r
man wumpus
```

In csh, the result would be *wumpus.1:r*. A substitution followed by a colon may need to be insulated from it with braces:

```
> mv a.out /usr/games/wumpus
> setenv PATH !$:h:$PATH
Bad ! modifier: $.
> setenv PATH !{-2$:h}:$PATH
setenv PATH /usr/games:/bin:/usr/bin:.
```

The first attempt would succeed in csh but fails in tcsh, because tcsh expects another modifier after the second colon rather than $.

Finally, history can be accessed through the editor as well as through the substitutions just described. The following commands search for events in the history list and compile them into the input buffer:

- **up-history**
- **down-history**
- **history-search-backward**
- **history-search-forward**
- **i-search-back**
- **i-search-fwd**
- **vi-search-back**
- **vi-search-fwd**
- **copy-prev-word**
- **insert-last-word**

The **toggle-literal-history** editor command switches between the expanded and literal forms of history lines in the input buffer. **expand-history** and **expand-line** expand history substitutions in the current word and in the entire input buffer respectively.

Alias Substitution

The shell maintains a list of aliases which can be set, unset and printed by the **alias** and **unalias** commands. After a command line is parsed into simple commands (see “Command Execution” on page 146) the first word of each command, left-to-right, is checked to see if it has an alias. If so, the first word is replaced by the alias. If the alias contains a history reference, it undergoes history substitution as though the original command were the previous input line. If the alias does not contain a history reference, the argument list is left untouched.

Thus if the alias for **ls** were **ls -l** the command **ls** */usr*would become **ls -l** */usr*, the argument list here being undisturbed. If the alias for **lookup** were **grep** *!^*

/etc/passwd then **lookup bill** would become **grep** *bill /etc/passwd*. Aliases can be used to introduce parser metasyntax. For example, **alias print** *'pr \!* | lpr'* defines a **command** (**print**) which prints its arguments to the line printer.

Alias substitution is repeated until the first word of the command has no alias. If an alias substitution does not change the first word (as in the previous example) it is flagged to prevent a loop. Other loops are detected and cause an error.

Some aliases are referred to by the shell; see “tcsh Built-in Commands” on page 154.

Variable Substitution

The tcsh shell maintains a list of variables, each of which has as value a list of zero or more words. The values of tcsh shell variables can be displayed and changed with the **set** and **unset** commands. The system maintains its own list of "environment" variables. These can be displayed and changed with **printenv**, **setenv** and **unsetenv**.

Variables may be made read-only with **set -r**. Read-only variables may not be modified or unset; attempting to do so will cause an error. Once made read-only, a variable cannot be made writable, so **set -r** should be used with caution. Environment variables cannot be made read-only.

Some variables are set by the tcsh shell or referred to by it. For instance, the **argv** variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways. Some of the variables referred to by the tcsh shell are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the **verbose** variable is a toggle which causes command input to be echoed. The **-v** command line option sets this variable. Special shell variables lists all variables which are referred to by the shell.

Other operations treat variables numerically. The **@** (at) command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by $ characters. This expansion can be prevented by preceding the $ with a \ except within double quotes (") where it always occurs, and within single quotes (') where it never occurs. Strings quoted by backward quotes or accents (`) are interpreted later (see “Command Substitution” on page 144) so $ substitution does not occur there until later, if at all. A $ is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together. It is thus possible for the first (command) word (to this point) to generate more than one word, the first of which becomes the command name, and the rest of which become arguments.

Unless enclosed in double quotes (") or given the :q modifier the results of variable substitution may eventually be command and filename substituted. Within ", a variable whose value consists of multiple words expands to a (portion of a) single

word, with the words of the variable's value separated by blanks. When the :q modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable which is not set.

$*name***[***selector***]**

${*name***[***selector***]}**
Substitutes only the selected words from the value of name. The selector is subjected to $ substitution and may consist of a single number or two numbers separated by a - (hyphen). The first word of a variable's value is numbered 1. If the first number of a range is omitted it defaults to 1. If the last member of a range is omitted it defaults to *$#name*. The selector * selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$0 Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$*number*

${*number***}**
Equivalent to $argv[*number*].

$* Equivalent to $argv, which is equivalent to $argv[*].

The : (colon) modifiers described under “History Substitution” on page 138, except for :p, can be applied to the substitutions above. More than one may be used. Braces may be needed to insulate a variable substitution from a literal colon just as with history substitution; any modifiers must appear within the braces. The following substitutions can not be modified with : modifiers.

$?*name*

${?*name***}**
Substitutes the string `1` if *name* is set, `0` if it is not.

$0 Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$?0 Substitutes `1` if the current input filename is known, `0` if it is not. Always `0` in interactive shells.

$#*name* **or ${#***name***}**
Substitutes the number of words in *name*.

**$#** Equivalent to '$#argv'.

$*%name*

${*%name***}**
Substitutes the number of characters in name.

$*%number*

${*%number***}**
Substitutes the number of characters in $argv[number].

$? Equivalent to $status.

$$ Substitutes the (decimal) process number of the (parent) shell.

$! Substitutes the (decimal) process number of the last background process started by this shell.

$< Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a shell script. While csh always quotes $<, as if it were equivalent to $<:q, tcsh does not. Furthermore, when tcsh is waiting for a line to be typed the user may type an interrupt to interrupt the sequence into which the line is to be substituted, but csh does not allow this.

The editor command **expand-variables**, normally bound to ^X-$, can be used to interactively expand individual variables.

Command, Filename and Directory Stack Substitiution

The remaining substitutions are applied selectively to the arguments of tcsh built-in commands. This means that portions of expressions which are not evaluated are not subjected to these expansions. For commands which are not internal to the tcsh shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command Substitution: Command substitution is indicated by a command enclosed in ' ' '. The output from such a command is broken into separate words at blanks, tabs and newlines, and null words are discarded. The output is variable and command substituted and put in place of the original string.

Command substitutions inside double quotes (") retain blanks and tabs; only newlines force new words. The single final newline does not force a new word in any case. It is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

Filename Substitution: If a word contains any of the characters '*', '?', '[' or '{' or begins with the character '~' it is a candidate for filename substitution, also known as **globbing**. This word is then regarded as a pattern (glob-pattern), and replaced with an alphabetically sorted list of file names which match the pattern.

In matching filenames, the character **.** (period) at the beginning of a filename or immediately following a / (forward slash), as well as the character / must be matched explicitly. The character * matches any string of characters, including the null string. The character ? matches any single character. The sequence [...] matches any one of the characters enclosed. Within [...], a pair of characters separated by - matches any character lexically between the two.

Some glob-patterns can be negated: The sequence [^...] matches any single character not specified by the characters and/or ranges of characters in the braces.

An entire glob-pattern can also be negated with ^:

```
> echo *
bang crash crunch ouch
> echo ^cr*
bang ouch
```

Glob-patterns which do not use '?', '*', or'[]' or which use '{}' or '^' (below) are not negated correctly.

The metanotation **a{b,c,d}e** is a shorthand for abe ace ade. Left-to-right order is preserved: **/usr/source/s1/{oldls,ls}.c** expands to /usr/source/s1/oldls.c **/usr/source/s1/ls.c**. The results of matches are sorted separately at a low level to preserve this order: **../{memo,*box}** might expand to **../memo ../box ../mbox**. (Note that 'memo' was not sorted with the results of matching '*box'.) It is not an error when this construct expands to files which do not exist, but it is possible to get an error from a command to which the expanded list is passed. This construct may be nested. As a special case the words {, } and {} are passed undisturbed. The character ~ at the beginning of a filename refers to home directories. Standing alone, i.e. ~, it expands to the invoker's home directory as reflected in the value of the home shell variable. When followed by a name consisting of letters, digits and - (hyphen) characters the shell searches for a user with that name and substitutes their home directory; thus **~ken** might expand to **/usr/ken** and **~ken/chmach** to **/usr/ken/chmach**. If the character ~ is followed by a character other than a letter or / or appears elsewhere than at the beginning of a word, it is left undisturbed. A command like **setenv** *MANPATH /usr/man:/usr/local/man:~/lib/man* does not, therefore, do home directory substitution as one might hope. It is an error for a glob-pattern containing '*', '?', '[' or '~', with or without '^', not to match any files. However, only one pattern in a list of glob-patterns must match a file (so that, for example, rm *.a *.c *.o would fail only if there were no files in the current directory ending in '.a', '.c', or '.o'), and if the nonomatch shell variable is set a pattern (or list of patterns) which matches nothing is left unchanged rather than causing an error.

The **noglob** shell variable can be set to prevent filename substitution, and the **expand-glob** editor command, normally bound to ^X-*, can be used to interactively expand individual filename substitutions.

Directory Stack Substitution: The directory stack is a list of directories, numbered from zero, used by the **pushd**, **popd** and **dirs** built-in commands for tcsh. **dirs** can print, store in a file, restore and clear the directory stack at any time, and the **savedirs** and **dirsfile** shell variables can be set to store the directory stack automatically on logout and restore it on login. The **dirstack** shell variable can be examined to see the directory stack and set to put arbitrary directories into the directory stack.

The character = (equal) followed by one or more digits expands to an entry in the directory stack. The special case **=-** expands to the last directory in the stack. For example,

```
> dirs -v
0 /usr/bin
1 /usr/spool/uucp
2 /usr/accts/sys
> echo =1
/usr/spool/uucp
> echo =0/calendar
/usr/bin/calendar
> echo =-
/usr/accts/sys
```

The **noglob** and **nonomatch** shell variables and the **expand-glob** editor command apply to directory stack as well as filename substitutions.

Other Substitutions: There are several more transformations involving filenames, not strictly related to the above but mentioned here for completeness. Any filename may be expanded to a full path when the **symlinks** variable is set to *expand*. Quoting prevents this expansion, and the **normalize-path** editor command does it on demand. The **normalize-command** editor command expands commands in PATH into full paths on demand. Finally, **cd** and **pushd** interpret - (hyphen) as the old working directory (equivalent to the tcsh shell variable **owd**). This is not a substitution at all, but an abbreviation recognized only by those commands. Nonetheless, it too can be prevented by quoting.

Command Execution

The next three sections describe how the shell executes commands and deals with their input and output.

Simple Commands, Pipelines, and Sequences

A simple command is a sequence of words, the first of which specifies the command to be executed. A series of simple commands joined by '|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next.

Simple commands and pipelines may be joined into sequences with ';', and will be executed sequentially. Commands and pipelines can also be joined into sequences with '||' or '&&', indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively.

A simple command, pipeline or sequence may be placed in parentheses, '()', to form a simple command, which may in turn be a component of a pipeline or sequence. A command, pipeline or sequence can be executed without waiting for it to terminate by following it with an '&'.

Built-in and Non-Built-in Command Execution

tcsh Built-in commands are executed within the shell. If any component of a pipeline except the last is a built-in command, the pipeline is executed in a subshell.

Parenthesized commands are always executed in a subshell:

```
(cd; pwd); pwd
```

which prints the home directory, leaving you where you were (printing this after the home directory), while

```
cd; pwd
```

leaves you in the home directory. Parenthesized commands are most often used to prevent **cd** from affecting the current shell.

When a command to be executed is found not to be a built-in command the tcsh shell attempts to execute the command via **execve**. Each word in the variable pathnames a directory in which the tcsh shell will look for the command. If it is given neither a **-c** nor a **-t** option, the shell hashes the names in these directories into an internal table so that it will only try an **execve** in a directory if there is a possibility that the command resides there. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via **unhash**), if the shell was given a **-c** or **-t** argument or in any case for each directory component of path which does not begin with a /, the shell concatenates the current working directory with the given command name to form a pathname of a file which it then attempts to execute.

If the file has execute permissions but is not an executable to the system (that is, it is neither an executable binary nor a script which specifies its interpreter), then it is assumed to be a file containing shell commands and a new shell is spawned to read it. The **shell** special alias may be set to specify an interpreter other than the shell itself.

Input or Output

The standard input and standard output of a command may be redirected with the following syntax:

Table 1 (Page 1 of 2). Standard Input/Output Syntax for tcsh Shell

Syntax	Description
< *name*	Open file name (which is first variable, command and filename expanded) as the standard input.
<< *word*	Read the shell input up to a line which is identical to *word*. *word* is not subjected to variable, filename or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting \, " , ' ' or ' ' ' appears in *word* variable and command substitution is performed on the intervening lines, allowing \ to quote $, \ and ' (single quote). Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.
> *name* >! *name* >& *name* >&! *name*	The file *name* is used as standard output. If the file does not exist then it is created; if the file exists, its is overwritten and, therefore, the previous contents are lost. If the shell variable **noclobber** is set, then the file must not exist or be a character special file (for example, a terminal or **/dev/null**) or an error results. This helps prevent accidental destruction of files. In this case the ! forms can be used to suppress this check. The forms involving & (ampersand) route the diagnostic output into the specified file as well as the standard output. *name* is expanded in the same way as < input filenames are.

Table 1 (Page 2 of 2). Standard Input/Output Syntax for tcsh Shell	
Syntax	**Description**
>> *name* >>& *name* >>! *name* >>&! *name*	Like >, but appends output to the end of *name*. If the shell variable *noclobber* is set, then it is an error for the file *not* to exist, unless one of the ! forms is given.

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands run from a file of shell commands have no access to the text of the commands by default; rather they receive the original standard input of the shell. The << mechanism should be used to present inline data. This permits shell command scripts to function as components of pipelines and allows the shell to block read its input. The default standard input for a command run detached is *not* the empty file **/dev/null**, but the original standard input of the shell. If this is a terminal and if the process attempts to read from the terminal, then the process will block and the user will be notified (see “Jobs” on page 151).

Diagnostic output may be directed through a pipe with the standard output. Simply use the form |& rather than just |.

The shell cannot presently redirect diagnostic output without also redirecting standard output, but (command > output-file) >& error-file is often an acceptable workaround. Either output-file or error-file may be **/dev/tty** to send output to the terminal.

Features

Having described how the tcsh shell accepts, parses and executes command lines, we now turn to a variety of its useful features.

Control Flow

The tcsh shell contains a number of commands which can be used to regulate the flow of control in command files (shell scripts) and (in limited by useful ways) from terminal output. These commands all operate by forcing the shell to reread or skip in its input and, due to the implementation, restrict the placement of some of the commands.

The foreach, switch, and while statements, as well as the if-then-else form of the if statement, require that the major keywords appear in a single simple command on an input line.

If the shell's input is not seekable, the shell buffers up input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop . (To the extent that this allows, backward gotos will succeed on non-seekable inputs.)

Expressions

The **if**, **while**, and **exit** built-in commands use expressions with a common syntax. The expressions can include any of the operators described in the next three sections. Note that the **@** built-in command has its own separate syntax.

Logical, Arithmetical and Comparison Operators: These operators are similar to those of C and have the same precendence. They include:

```
|| && | ^ & == != =~ !~ <= >=
< > << >> + - * / % ! ~ ( )
```

Here the precedence increases to the right, '==' '!=' '=~' and '!~', '<=' '>=' '<' and '>', '<<' and '>>', '+' and '-', '*' / and '%' being in groups, at the same level. The '==' '!=' '=~' and '!~' operators compare their arguments as strings; all others operate on numbers. The operators '=~' and '!~' are like '!=' and '==' except that the right hand side is a glob-pattern (see “Filename Substitution” on page 144) against which the left hand operand is matched. This reduces the need for use of the **switch** built-in command in shell scripts when all that is really needed is pattern matching.

Strings which begins with `0` are considered octal numbers. Null or missing arguments are considered `0`. The results of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser ('$' '|' '<' '>' '(' ')') they should be surrounded by spaces.

Command Exit Status: Commands can be executed in expressions and their exit status returned by enclosing them in braces ({}). Remember that the braces should be separated from the words of the command by spaces. Command executions succeed, returning true, that is, `1`, if the command exits with status `0`, otherwise they fail, returning false (`0`). If more detailed status information is required then the command should be executed outside of an expression and the status shell variable examined.

File Inquiry Operators: Some of these operators perform true/false tests on files and related objects. They are of the form **-op** file, where op is one of:

r Read access

w Write access

x Execute access

X Executable in the path or shell built-in. For example, **–X ls** and **–X ls-F** are generally true, but **–X /bin/ls** is not.

e Existence

o Ownership

x Zero size

s Non-zero size

f Plain file

d Directory

l Symbolic link

b Block special file

c Character special file

p Named pipe (fifo)

S Socket special file

u Set-user ID bit is set

g Set-group-ID bit is set

k Sticky bit is set

t t *file_descriptor* (which must be a digit) is an open file descriptor for a terminal device

L Applies subsequent operators in a multiple-operator test to a symbolic link rather that to the file to which the link points

file is command and filename expanded and then tested to see if it has the specified relationship to the real user. If file does not exist or is inaccessible or, for the operators indicated by *, if the specified file type does not exist on the current system, then all inquiries return false (`0`).

These operators may be combined for conciseness: -xy file is equivalent to -x file && -y file. For example, -fx is true (returns `1`) for plain executable files, but not for directories.

L may be used in a multiple-operator test to apply subsequent operators to a symbolic link rather that to the file to which the link points. For example, -lLo is true for links owned by the invoking user. Lr, Lw, and Lx are always ture for links and false for non-links. L has a different meaning when it is the last operator in a multiple-operator test.

It is possible but not useful, and sometimes misleading, to combine operators which expect file to be a file with operators which do not (for example, X and t). Following L witha non-file operator can lead to particularly strange results.

Other operators return other information, i.e. not just `0` or `1`. They have the same format as before where op may be one of:

A Last file access time, as the number of seconds since epoch

A: Like **A**, but in timestamp format, that is, 'Fri May 14 16:36:10 1993'

M Last file modification time

M: Like **M**, but in timestamp format

C Last inode modification time

C: Like **C**, but in timestamp format

D Device number

I Inode number

F Composite file identifier, in the form **device : inode**

L The name of the file pointed to by a symbolic link

N Number of (hard) links

P Permissions, in octal, without leading zero

P: Like **P**, with leading zero

P mode
Equivalent to -P mode & file, that is, -P22 file returns 22 if file is writable by group and other, 20 if by group only, and 0 if by neither.

P mode:
Like **P mode**, with leading zero

U Numeric userid

U: Username, or the numeric userid if the username is unknown

G Numeric groupid

G: Groupname, or the numeric groupid if the groupname is unknown

Z Size in bytes

Only one of these operators may appear in a multiple-operator test, and it must be the last. **L** has a different meaning at the end of and elsewhere in a multiple-operator test. Because 0 is a valid return value for many of these operators, they do not return 0 when they fail: most return -1, and **F** returns : (colon).

File inquiry operators can also be evaluated with the **filetest** built-in command.

Jobs

The shell associates a job with each pipeline. It keeps a table of current jobs, printed by the jobs command, and assigns them small integer numbers. When a job is started asynchronously with & (ampersand), the shell prints a line which looks like

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may press the suspend key (usually ^Z), which sends a STOP signal to the current job. The shell will then normally indicate that the job has been 'Suspended' and print another prompt. If the **listjobs** shell variable is set, all jobs will be listed like the **jobs** built-in command; if it is set to 'long' the listing will be in long format, like **jobs -l**. You can then manipulate the state of the suspended job. You can put it in the background with the **bg** command or run some other commands and eventually bring the job back into the foreground with **fg**. (See also the **run-fg-editor** editor command.) A ^Z takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. The **wait** built-in command causes the shell to wait for all background jobs to complete.

The ^] key sends a delayed suspend signal, which does not generate a STOP signal until a program attempts to read it, to the current job. This can usefully be typed ahead when you have prepared some commands for a job which you wish to stop after it has read them. The ^Y key performs this function in csh; in tcsh , ^Y is an editing command.

A job being run in the background stops if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command stty tostop. If you set the stty option, then background jobs will stop when they try to produce output like they do when they try to read input.

There are several ways to refer to jobs in the shell. The character % introduces a job name. If you wish to refer to job number 1, you can name it as %1. Just naming a job brings it to the foreground; thus %' is a synonym for **fg** *%1*, bringing job 1 back into the foreground. Similarly, saying %1 & resumes job 1 in the background, just like **bg** *%1*. A job can also be named by an unambigous prefix of the string typed in to start it: %ex would normally restart a suspended 'ex' job, if there were only one suspended job whose name began with the string 'ex'. It is also possible to say %? string to specify a job whose text contains string , if there is only one such job.

The shell maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a + (plus) and the previous job with a - (hyphen). The abbreviations %+, %, and (by analogy with the syntax of the history mechanism) %% all refer to the current job, and %- refers to the previous job.

The job control mechanism requires that the stty option *new* be set on some systems. It is an artifact from a *new* implementation of the tty driver which allows generation of interrupt characters from the keyboard to tell jobs to stop. See stty and the **setty** tcsh built-in command for details on setting options in the new tty driver.

Status Reporting

The tcsh shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before it prints a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable **notify**, the shell will notify you immediately of changes of status in background jobs. There is also a shell command **notify** which marks a single process so that its status changes will be immediately reported. By default notify marks the current process; simply say 'notify' after starting a background job to mark it.

When you try to leave the shell while jobs are stopped, you will be warned that 'You have stopped jobs.' You may use the **jobs** command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time, and the suspended jobs will be terminated.

Automatic, Periodic and Timed Events

There are various ways to run commands and take other actions automatically at various times in the **life cycle** of the shell.

- The **sched** built-in command puts commands in a scheduled-event list, to be executed by the shell at a given time.
- The **beepcmd**, **cwdcmd**, **periodic** and **precmd** special aliases can be set, respectively, to execute commands when the shell wants to ring the bell, when the working directory changes, every tperiod minutes and before each prompt.
- The **autologout** shell variable can be set to log out of the shell after a given number of minutes of inactivity.
- The **mail** shell variable can be set to check for new mail periodically.
- The **printexitvalue** shell variable can be set to print the exit status of commands which exit with a status other than zero.

- The **rmstar** shell variable can be set to ask the user, when **rm *** is typed, if that is really what was meant.
- The **time** shell variable can be set to execute the **time** built-in command after the completion of any process that takes more than a given number of CPU seconds.
- The **watch** and **who** shell variables can be set to report when selected users log in or out, and the **log** built-in command reports on those users at any time.

National Language System Report

When using the system's NLS, the setlocale function is called to determine appropriate character classification and sorting. This function typically examines the LANG and LC_CTYPE environment variables; refer to the system documentation for further details.

Unknown characters (those that are neither printable nor control characters) are printed in the format *\nnn*.

The version shell variable indicates what options were chosen when the shell was compiled. Note also the **newgrp** built-in and **echo_style** shell variables and the locations of the shell's input files (see “tcsh Files” on page 170).

The tcsh shell currently does not support 3 locales. They are IBM-1388 (Chinese), IBM-933 (Korean) and IBM-937 (Traditional Chinese).

Signal Handling

Login shells ignore interrupts when reading the file ~/.logout. The shell ignores quit signals unless started with **-q**. Login shells catch the terminate signal, but non-login shells inherit the terminate behavior from their parents. Other signals have the values which the shell inherited from its parent.

In shell scripts, the shell's handling of interrupt and terminate signals can be controlled with **onintr**, and its handling of hangups can be controlled with **hup** and **nohup**.

The shell exits on a hangup (see also the **logout** shell variable). By default, the shell's children do too, but the shell does not send them a hangup when it exits. **hup** arranges for the shell to send a hangup to a child when it exits, and **nohup** sets a child to ignore hangups.

Terminal Management

The shell uses three different sets of terminal (tty) modes: **edit**, used when editing, **quote**, used when quoting literal characters, and **execute**, used when executing commands. The shell holds some settings in each mode constant, so commands which leave the tty in a confused state do not interfere with the shell. The shell also matches changes in the speed and padding of the tty. The list of tty modes that are kept constant can be examined and modified with the **setty** built-in. Although the editor uses CBREAK mode (or its equivalent), it takes typed-ahead characters anyway.

The **echotc**, **settc** and **telltc** commands can be used to manipulate and debug terminal capabilities from the command line.

The tcsh shell adapts to window resizing automatically and adjusts the environment variables LINES and COLUMNS if set.

tcsh Built-in Commands

The list below contains **tcsh** built-in commands which are not **/bin/sh** built-ins. Descriptions for these commands are found at the end of this chapter.

%	**filetest**	**popd**	**uncomplete**
alloc	**glob**	**pushd**	**unhash**
bindkey	**hashstat**	**rehash**	**unlimit**
breaksw	**hup**	**repeat**	**unsetenv**
builtins	**limit**	**sched**	**watchlog**
bye	**login**	**setenv**	**where**
chdir	**logout**	**settc**	**which**
complete	**ls-F**	**setty**	
dirs	**notify**	**source**	
echotc	**onintr**	**telltc**	

Other tcsh built-in commands are also found in the OS/390 shell. In some cases, they may differ in function; see the specific command description for a discussion of the tcsh version of the command.

: (colon)	**cd**	**fg**	**nice**	**stop**	**unset**
@ (at)	**echo**	**history**	**nohup**	**suspend**	**wait**
alias	**eval**	**jobs**	**printenv**	**time**	
bg	**exec**	**kill**	**set**	**umask**	
break	**exit**	**newgrp**	**shift**	**unalias**	

As well as built-in commands, the tcsh shell has a set of special aliases:

beepcmd	**periodic**	**shell**
cwdcmd	**precmd**	

If set, each of these aliases executes automatically at the indicated time. They are initially undefined. For more information about aliases, see “Alias Substitution” on page 141.

Descriptions of these aliases are as follows:

beepcmd
Runs when the shell wants to ring the terminal bell.

cwdcmd
Runs after every change of working directory. For example, if the user is working on an X window system using xterm and a re-parenting window manager that supports title bars such as twm and does

```
> alias cwdcmd 'echo -n "^[]2;${HOST}:$cwd ^G"'
```

then the shell will change the title of the running xterm to be the name of the host, a colon, and the full current working directory. A fancier way to do that is

```
> alias cwdcmd 'echo -n "^[]2;${HOST}:$cwd^G^[]1;${HOST}^G"'
```

This will put the hostname and working directory on the title bar but only the hostname in the icon manager menu. Putting a **cd, pushd** or **popd** in **cwdcmd** may cause an infinite loop.

periodic
Runs every **tperiod** minutes. This provides a convenient means for checking on common but infrequent changes such as new mail. For example, if one does

```
> set tperiod = 30
> alias periodic checknews
```

then the **checknews** program runs every 30 minutes. If **periodic** is set but **tperiod** is unset or set to 0, **periodic** behaves like **precmd**.

precmd
Runs just before each prompt is printed. For example, if one does

```
> alias precmd date
```

then **date** runs just before the shell prompts for each command. There are no limits on what **precmd** can be set to do, but discretion should be used.

shell
Specifies the interpreter for executable scripts which do not themselves specify an interpreter. The first word should be a full pathname to the desired interpreter. For example: **/bin/tcsh** or **/usr/local/bin/tcsh** (by default, this is set to **/bin/tcsh**).

tcsh Programming Constructs

1. **breaksw**

   Causes a break from a **switch**, resuming after the **endsw**.

2. **case** *label*

   A label in a switch. See the **switch** built-in description.

3. **continue**

   Continues execution of the nearest enclosing **while** or **foreach**. The rest of the commands on the current line are executed.

4. **default**

   Labels the default **case** in a **switch** statement. It should come after all **case** labels.

5. **else**
   end
   endif
   endsw

   See the description of the **foreach, if, switch**, and **while** statements that follow.

6. **goto** *word*

   With **goto**, *word* is filename and command substituted to yield a string of the form label. The tcsh shell rewinds its input as much as possible, searches for a line of the form *label*, possible preceded by blanks or tabs, and continues exectution after that line.

7.

foreach
...
end

Successively sets the variable name to each member of wordlist and executes the sequence of commands between this command and the matching end. (Both foreach and end must appear alone on separate lines.) The built-in command **continue** may be used to continue the loop prematurely and the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read once prompting with **foreach?** (or **prompt2**) before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

8.

if (*expr*) **then**
...
else if (*expr2*) **then**
...
else
...
endif

If the specified *expr* is true then the commands to the first **else** are executed; otherwise if *expr2* is true then the commands to the second **else** are executed, etc.. Any number of **else-if** pairs are possible; only one **endif** is needed. The **else** part is optional. (The words **else** and **endif** must appear at the beginning of input lines; the **if** must appear alone on its input line or after an **else**.)

9.

switch (*string*)
case *str1*:
...
breaksw
...
default
...
breaksw
endsw

Each case label is successively matched, against the specified *string* which is first command and filename expanded. The file metacharacters *, ? and [...] may be used in the **case** labels, which are variable expanded. If none of the labels match before a **default** label is found, then the execution begins after the **default** label. Each **case** label and the **default** label must appear at the beginning of a line. The command **breaksw** causes execution to continue after the **endsw**. Otherwise control may fall through **case** labels and default labels as in C. If no label matches and there is no default, execution continues after the **endsw**.

10.

while (*expr*)
...
end

Executes the commands between the **while** and the matching **end** while *expr* (expression) evaluates non-zero. **while** and **end** must appear alone on their input lines. **break** and **continue** may be used to terminate or continue the loop prematurely. If the input is a terminal, the user is prompted the first time through the loop as with **foreach**.

tcsh Shell and Environment Variables

The variables described in this section have special meaning to the tcsh shell. The tcsh shell sets **addsuffix**, **argv**, **autologout**, **command**, **echo_style**, **edit**, **gid**, **group**, **home**, **loginsh**, **path**, **prompt**, **prompt2**, **prompt3**, **shell**, **shlvl**, **tcsh**, **term**, **tty**, **uid**, **user**, and **version** at startup. They do not change thereafter, unless changed by the user. The tcsh shell updates **cwd**, **dirstack**, **owd**, and **status** when necessary, and sets **logout** on logout.

The shell synchronizes **group**, **home**, **path**, **shlvl**, **term**, and **user** with the environment variables of the same names: whenever the environment variable changes the shell changes the correpsonding shell variable to match (unless the shell variable is read-only) and vice versa. Although **cwd** and PWD have identical meanings, they are not synchronized in this manner.

The shell automatically interconverts the different formats of path and PATH.

Table 2 (Page 1 of 9). tcsh Built-in Shell Variables

Variable	Purpose
addsuffix	If set, filename completion adds / to the end of directories and a space to the end of normal files.
ampm	This variable gives a user the ability to alter the time format in their tcsh prompt. Specifically, **ampm** will override the %T and %P formatting sequences in a user's prompt. If set, all times are shown in 12hour AM/PM format.
argv	The arguments to the shell. Positional parameters are taken from argv, that is, $1 is replaced by $argv, etc.. Set by default, but usually empty in interactive shells.
autocorrect	If set, the **spell-word** editor command is invoked automatically before each completion. (This variable is not implemented.)
autoexpand	If set, the **expand-history** editor command is invoked automatically before each completion attempt.
autolist	If set, possibilities are listed after an ambiguous completion. If set to *ambiguous*, possibilites are listed only when no new characters are added by completion.
autologout	Set to the number of minutes of inactivity before automatic logout. Automatic locking is an unsupported feature on the OS/390 platform. If you specify a second parameter on the **autologout** statement (intending it to be for autolock), this parameter will be assigned to **autologout**. When the shell automatically logs out, it prints 'autologout', sets the variable logout to *automatic* and exits. Set to `60` (automatic logout after 60 minutes) by default in login and superuser shells, but not if the shell thinks it is running under a window system (the DISPLAY environment variable is set), or the tty is a pseudo-tty (pty). See also the **logout** shell variable.

Table 2 (Page 2 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**
backslash_ quote	If set, backslashes (\) always quote \, ' (single quote) and " (double quote). This may make complex quoting tasks easier, but it can cause syntax errors in csh scripts.
cdpath	A list of directories in which **cd** should search for subdirectories if they aren't found in the current directory.
command	If set, the command which was passed to the shell with the **-c** flag.
complete	If set to **enhance**, completion first ignores case and then considers periods, hyphens and underscores ('.', '-' and '_') to be word separators and hyphens and underscores to be equivalent.
correct	If set to *cmd*, commands are automatically spelling-corrected. If set to *complete*, commands are automatically completed. If set to *all*, the entire command line is corrected.
cwd	The full pathname of the current directory. See also the **dirstack** and **owd** shell variables.
dextract	If set, **pushd** *+n* extracts the *n*th directory from the directory stack rather than rotating it to the top.
dirsfile	The default location in which **dirs -S** and **dirs -L** look for a history file. If unset, **~/.cshdirs** is used. Because only **~/.tcshrc** is normally sourced before **~/.cshdirs**, dirsfile should be set in **~/.tcshrc** rather than **~/.login**. For example: `set dirsfile = ~/.cshdirs`
dirstack	An array of all the directories on the directory stack. $dirstack[1] is the current working directory, $dirstack[2] the first directory on the stack, etc. Note that the current working directory is $dirstack[1] but =0 in directory stack substitutions, etc. One can change the stack arbitrarily by setting **dirstack**, but the first element (the current working directory) is always correct. See also the **cwd** and **owd** shell variables.
dunique	If set, **pushd** removes any instances of *name* from the stack before pushing it onto the stack.
echo	If set, each command with its arguments is echoed just before it is executed. For non-built-in commands all expansions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively. Set by the **-x** command line option.

Table 2 (Page 3 of 9). tcsh Built-in Shell Variables

Variable	Purpose

echo_style	The style of the **echo** built-in. May be set to:
	bsd Don't echo a newline if the first argument is **-n**.
	sysv Recognize backslashed escape sequences in echo strings.
	both Recognizes both the **-n** flag and backslashed escape sequences; the default.
	none Recognize neither.
	Set to *both* by default to the local system default.
	The following is an example of this variable's use:
	`> echo $echo_style`
	`bsd`
	`> echo "\n"`
	`\n`
	`> echo -n "test"`
	`test>`
	`> set echo_style=sysv`
	`> echo $echo_style`
	`sysv`
	`> echo "\n"`
	
	`> echo -n "test"`
	`-n test`
	`> set echo_style=both`
	`> echo $echo_style`
	`both`
	`> echo -n "test"`
	`test> echo "\n"`
	
	`>set echo_style=none`
	`> echo $echo_style`
	`none`
	`> echo -n "test"`
	`-n test`
	`> echo "\n"`
	`\n`
	`>`

edit	If set, the command-line editor is used. Set by default in interactive shells.
ellipsis	If set, the %c'/'%. and %C prompt sequences (see the **prompt** shell variable) indicate skipped directories with an ellipsis (...) instead of /.
fignore	Lists file name suffixes to be ignored by completion.
filec	In the tcsh shell, completion is always used and this variable is ignored.
gid	The user's real group ID.
group	The user's group name.
histchars	A string value determining the characters used in history substitution. The first character of its value is used as the history substitution character, replacing the default character ! (exclamation point). The second character of its value replaces the character ^ (caret) in quick substitutions.

Table 2 (Page 4 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**
histdup	Controls handling of duplicate entries in the history list. If set to *all* only unique history events are entered in the history list. If set to *prev* and the last history event is the same as the current command, then the current command is not entered in the history. If set to *erase* and the same event is found in the history list, that old event gets erased and the current one gets inserted. The *prev* and *all* options renumber history events so there are no gaps.
histfile	The default location in which **history -S** and **history -L** look for a history file. If unset, *~/.history* is used. **histfile** is useful when sharing the same home directory between different machines, or when saving separate histories on different terminals. Because only *~/.tcshrc* is normally sourced before *~/.history*, **histfile** should be set in *~/.tcshrc* rather than *~/.login*. An example: `set histfile = ~/.history`
histlit	If set, built-in and editor commands and the savehist mechanism use the literal (unexpanded) form of lines in the history list. See also the **toggle-literal-history** editor command.
history	The first word indicates the number of history events to save. The optional second word indicates the format in which history is printed; if not given, %h\t%T\t%R\n is used. The format sequences are described below under **prompt**. (Note that %R has a variable meaning). Set to `100` by default.
home	Initialized to the home directory of the invoker. The filename expansion of ~ refers to this variable.
ignoreeof	If set to the empty string or 0 and the input device is a terminal, the **end-of-file** command (usually generated by the user by typing ^D on an empty line) causes the shell to print 'Use "logout" to leave tcsh.' instead of exiting. This prevents the shell from accidentally being killed. If set to a number *n*, the shell ignores n - 1 consecutive end-of-files and exits on the *n*th. If unset, 1 is used. That is, the shell exits on a single ^D.
implicitcd	If set, the shell treats a directory name typed as a command as though it were a request to change to that directory. If set to verbose, the change of directory is echoed to the standard output. This behavior is inhibited in non-interactive shell scripts, or for command strings with more than one word. Changing directory takes precedence over executing a like-named command, but it is done after alias substitutions. Tilde and variable expansions work as expected.
inputmode	If set to *insert* or *overwrite*, puts the editor into that input mode at the beginning of each line.
listflags	If set to *x, a* or *A*, or any combination thereof (for example, *xA*), they are used as flags to **ls-F**, making it act like **ls -xF**, **ls -Fa**, **ls -FA** or a combination (for example, **ls -FxA**): *a* shows all files (even if they start with a '.'), *A* shows all files but '.' and '..', and *x* sorts across instead of down. If the second word of listflags is set, it is used as the path to ls(1).
listjobs	If set, all jobs are listed when a job is suspended. If set to *long*, the listing is in long format.

Table 2 (Page 5 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**
listlinks	If set, the **ls-F** built-in command shows the type of file to which each symbolic link points. For an example of its use, see “ls-F built-in command for tcsh: List files” on page 184.
listmax	The maximum number of items which the **list-choices** editor ocmmand will list without asking first.
listmaxrows	The maximum number of rows of items which the list-choices editor command will list without asking first.
loginsh	Set by the shell if is a login shell. Setting or unsetting it within a shell has no effect. See also **shlvl**.
logout	Set by the shell to *normal* before a normal logout, *automatic* before an automatic logout, and *hangup* if the shell was killed by a hangup signal (see “Signal Handling” on page 153). See also the **autologout** shell variable.
mail	The names of the files or directories to check for incoming mail, separated by whitespace, and optionally preceeded by a numeric word. Before each prompt, if 10 minutes have passed since the last check, the shell checks each file and says 'You have new mail.' (or, if mail contains multiple files, 'You have new mail in name.') if the filesize is greater than zero in size and has a modification time greater than its access time. If you are in a login shell, then no mail file is reported unless it has been modified after the time the shell has started up, in order to prevent redundant notifications. Most login programs will tell you whether or not you have mail when you log in. If a file specified in mail is a directory, the shell will count each file within that directory as a separate message, and will report 'You have n mails.' or 'You have n mails in name.' as appropriate. This functionality is provided primarily for those systems which store mail in this manner, such as the Andrew Mail System. If the first word of mail is numeric it is taken as a different mail checking interval, in seconds. Under very rare circumstances, the shell may report 'You have mail.' instead of 'You have new mail.'
matchbeep	If set to *never*, completion never beeps. If set to *nomatch*, it beeps only when there is no match. If set to *ambiguous*, it beeps when there are multiple matches. If set to *notunique*, it beeps when there is one exact and other longer matches. If unset, *ambiguous* is used.
nobeep	If set, beeping is completely disabled.
noclobber	If set, restrictions are placed on output redirection to insure that files are not accidentally destroyed and that >> redirections refer to existing files, as described in “Input or Output” on page 147.
noglob	If set, filename substitution and directory stack substitution are inhibited. This is most useful in shell scripts which do not deal with filenames, or after a list of filenames has been obtained and further expansions are not desirable.
nokanji	If set and the shell supports Kanji (see the **version** shell variable), it is disabled so that the meta key can be used.
nonomatch	If set, a filename substitution or directory stack substitution which does not match any existing files is left untouched rather than causing an error. It is still an error for the substitution to be malformed, that is, **echo [** still gives an error.

Table 2 (Page 6 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**
nostat	A list of directories (or glob-patterns which match directories; see "Filename Substitution" on page 144) that should not be stat(2)ed during a completion operation. This is usually used to exclude directories which take too much time to stat(2), for example */afs*.
notify	If set, the shell announces job completions asynchronously. The default is to present job completions just before printing a prompt.
owd	The old working directory, equivalent to the - (hyphen) used by **cd** and **pushd**. See also the **cwd** and **dirstack** shell variables.
path	A list of directories in which to look for executable commands. A null word specifies the current directory. If there is no path variable then only full pathnames will execute. **path** is set by the shell at startup from the **PATH** environment variable or, if **PATH** does not exist, to a system-dependent default something like (/usr/local/bin /usr/bsd /bin /usr/bin .). The shell may put '.' first or last in path or omit it entirely depending on how it was compiled; see the **version** shell variable. A shell which is given neither the **-c** nor the **-t** option hashes the contents of the directories in path after reading *~/.tcshrc* and each time path is reset. If one adds a new command to a directory in path while the shell is active, one may need to do a rehash for the shell to find it.
printexitvalue	If set and an interactive program exits with a non-zero status, the shell prints 'Exit status'.
prompt2	The string with which to prompt in while and foreach loops and after lines ending in \. The same format sequences may be used as in prompt (note the variable meaning of %R). Set by default to *%R?* in interactive shells.
prompt3	The string with which to prompt when confirming automatic spelling correction. The same format sequences may be used as in **prompt** (note the variable meaning of *%R*). Set by default to *CORRECT>%R (y|n|e|a)?* in interactive shells.
promptchars	If set (to a two-character string), the %# formatting sequence in the prompt shell variable is replaced with the first character for normal users and the second character for the superuser.
pushdtohome	If set, **pushd** without arguments does **pushd ^**, like **cd**.
pushdsilent	If set, **pushd** and **popd** do not print the directory stack.
recexact	If set, completion completes on an exact match even if a longer match is possible.
recognize_ only_ executables	If set, command listing displays only files in the path that are executable.
rmstar	If set, the user is prompted before **rm *** is executed.
rprompt	The string to print on the right-hand side of the screen (after the command input) when the prompt is being displayed on the left. It recognises the same formatting characters as **prompt**. It will automatically disappear and reappear as necessary, to ensure that command input isn't obscured, and will only appear if the prompt, command input, and itself will fit together on the first line. If **edit** isn't set, then **rprompt** will be printed after the prompt and before the command input.
savedirs	If set, the shell does **dirs -S** before exiting.

Table 2 (Page 7 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**
savehist	If set, the shell does **history -S** before exiting. If the first word is set to a number, at most that many lines are saved. (The number must be less than or equal to history.) If the second word is set to **merge**, the history list is merged with the existing history file instead of replacing it (if there is one) and sorted by time stamp and the most recent events are retained. An example: `set savehist = ( 15 merge )`
sched	The format in which the sched built-in command prints scheduled events. If not given, %h\t%T\t%R\n is used. The format sequences are described above under **prompt**; note the variable meaning of *%R*.
shell	The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system (see "Built-in and Non-Built-in Command Execution" on page 146. Initialized to the (system-dependent) home of the shell.
shlvl	The number of nested shells. Reset to `1` in login shells. See also **loginsh**.
status	The status returned by the last command. If it terminated abnormally, then `0200` is added to the status. tcsh built-in commands which fail return exit status `1`, all other built-in commands return status `0`.
tcsh	The version number of the shell in the format R.VV.PP, where R is the major release number, VV the current version and PP the patchlevel.
term	The terminal type. Usually set in *~/.login* as described under "Options and Invocation" on page 127.
tperiod	The period, in minutes, between executions of the periodic special alias.
tty	The name of the tty, or empty if not attached to one.
uid	The user's login name.
user	The user's login name.
verbose	If set, causes the words of each command to be printed, after history substitution (if any). Set by the **–v** command line option.

Table 2 (Page 8 of 9). tcsh Built-in Shell Variables

Variable	Purpose
version	The version ID stamp. It contains the shell's version number (see tcsh), origin, release date, vendor, operating system and machine (see VENDOR, OSTYPE, and MACHTYPE environment variables) and a comma-separated list of options which were set at compile time. Options which are set by default in the distribution are noted. **8b** The shell is eight bit clean; default. **7b** The shell is not eight bit clean. **nls** The system's NLS is used; default for systems with NLS. **lf** Login shells execute **/etc/csh.login** before instead of after **/etc/csh.cshrc** and *~/.login* before instead of after **~/.tcshrc** and **~/.history**. **dl** '.' is put last in path for security; default. **nd** '.' is omitted from path for security. **vi** vi-style editing is the default rather than emacs. **dtr** Login shells drop DTR when exiting. **bye** **bye** is a synomym for logout and log is an alternate name for watchlog. **al** **autologout** is enabled; default. **kan** Kanji is used and the ISO character set is ignored, unless the nokanji shell variable is set. **sm** The system's malloc is used. **hb** The #!<program> <args> convention is emulated when executing shell scripts. **ng** The **newgrp** built-in is available. **rh** The shell attempts to set the REMOTEHOST environment variable. **afs** The shell verifies your password with kerberos server if local authentication fails. The **afsuser** shell variable or the AFSUSER environment variable override your local username if set. An administrator may enter additional strings to indicate differences in the local version.
visiblebell	If set, a screen flash is used rather than the audible bell. See **nobeep**.(`Currently not implemented`)

Table 2 (Page 9 of 9). tcsh Built-in Shell Variables	
Variable	**Purpose**

watch	A list of user/terminal pairs to watch for logins and logouts. If either the user is *any* all terminals are watched for the given user and vice versa. Setting watch to (*any any*) watches all users and terminals. For example,
	`set watch = (george ttyd 1 any console $user any)`
	reports activity of the user george on ttyd1, any user on the console, and oneself (or a trespasser) on any terminal.
	Logins and logouts are checked every 10 minutes by default, but the first word of watch can be set to a number to check every so many minutes. For example,
	`set watch = (1 any any)`
	reports any login/logout once every minute. For the impatient, the **log** built-in command triggers a watch report at any time. All current logins are reported (as with the **log** built-in) when watch is first set.
	The **who** shell variable controls the format of watch reports.

who	The format string for watch messages. The following sequences are replaced by the given information: **%n** The name of the user who logged in/out. **%a** The observed action, i.e., 'logged on'. 'logged off', or 'replaced olduser on'. **%l** The terminal (tty) on which the user logged in/out. **%M** The full hostname of the remote host, or 'local' if the login/logout was from the local host. **%m** The hostname of the remote host up to the first '.' (period). The full name is printed if it is an IP address or an X Window System display. %M and %m are available only on systems which store the remote hostname in /etc/utmp. If unset, %n has %a %l from %m. is used, or %n has %a %l. on systems which don't store the remote hostname.
wordchars	A list of non-alphanumeric characters to be considered part of a word by the forward-word, back-ward word, etc. editor commands. If unset, *?_-.[]~= is used.

tcsh shell variables not described in the above table are described below:

prompt
The string which is printed before reading each command from the terminal. prompt may include any of the following formatting sequences, which are replaced by the given information:

%/ The current working directory.

%~ The current working directory, but with one's home directory represented by ~ and other users' home directories represented by ~user as per filename substitution. ~user substitution happens only if the shell has already used ~user in a pathname in the current session.

%c[[0]*n*], %.[[0]*n*]
The trailing component of the current working directory, or *n* trailing components if a digit *n* is given. If *n* begins with `0`, the number of

skipped components precede the trailing component(s) in the format **/trailing**. If the ellipsis shell variable is set, skipped components are represented by an ellipsis so the whole becomes **...trailing**. ~ substitution is done as in %~~ above, but the ~ component is ignored when counting trailing components.

%C Like **%c**, but without ^ substitution.

%h, %!, !
The current history event number.

%M The full hostname.

%m The hostname up to the first '.' (period).

%S (%s)
Start (stop) standout mode.

%B (%b)
Start (stop) boldfacing mode.

%U (%u)
Start (stop) underline mode.

%t, %@
The time of day in 12–hour AM/PM format.

%T Like %t, but in 24–hour format (but see the **ampm** shell variable).

%p The precise time of day in 12–hour AM/PM format, with seconds.

%P Like **%p**, but in 24–hour format (but see the **ampm** shell variable).

\c *c* is parsed as in **bindkey**.

^c *c* is parsed as in **bindkey**.

%% A single %.

%n The user name.

%d The weekday in 'Day' format.

%D The day in 'dd' format.

%w The month in 'Mon' format.

%W The month in 'mm' format.

%y The year in 'yy' format.

%Y The year in 'yyyy' format.

%l The tcsh shell's tty.

%L Clears from the end of the prompt to end of the display or the end of the line.

%$ Expands the shell or environment variable name immediately after the $.

%# > (or the first character of the **promptchars** shell variable) for normal users, # (or the second character of **promptchars**) for the superuser.

%{*string%***}**
Includes *string* as a literal escape sequence. It should be used only to change terminal attributes and should not move the cursor location. This cannot be the last sequence in **prompt**.

%? The return code of the command executed just before the prompt.

%R In **prompt2**, the status of the parser. In **prompt3**, the corrected string. In **history**, the history string.

The bold, standout and underline sequences are often used to distinguish a superuser shell. For example,

```
>set prompt = "%m [%h] %B[%@%b [%/] you rang?"
tut [37] [2:54] [/usr/accts/sys] you rang? _
```

Set by default to %# in interactive shells.

symlinks

Can be set to several different values to control symbolic link ('symlink') resolution:

- If set to *chase*, whenever the current directory changes to a directory containing a symbolic link, it is expanded to the real name of the directory to which the link points. This does not work for the user's home directory.
- If set to *ignore*, the shell tries to construct a current directory relative to the current directory before the link was crossed. This means that cding through a symbolic link and then **cd..'ing** returns one to the original directory. This only affects built-in commands and filename completion.
- If set to *expand*, the shell tries to fix symbolic links by actually expanding arguments which look like pathnames. This affects any command, not just built-ins. Unfortunately, this does not work for hard-to-recognize filenames, such as those embedded in command options. Expansion may be prevented by quoting. While this setting is usually the most convenient, it is sometimes misleading and sometimes confusing when it fails to recognize an argument which should be expanded. A compromise is to use *ignore* and use the editor command normalize-path (bound by default to ^X-n) when necessary.

Some examples are in order. First, let's set up some play directories:

```
> cd /tmp
> mkdir from from/src to
> ln -s from/src to/dist
```

Here's the behavior with **symlinks** unset,

```
> cd /tmp/to/dist; echo $cwd
/tmp/to/dist
> cd ..; echo $cwd
/tmp/from
```

here's the behavior with symlinks set to *chase*,

```
> cd /tmp/to/dst; echo $cwd
/tmp/from/src
> cd ..; echo $cwd
/tmp/from
```

here's the behavior with symlinks set to *ignore*,

```
> cd /tmp/to/dist; echo $cwd
/tmp/to/dst
> cd ..; echo $cwd
/tmp/to
```

and here's the behavior with symlinks set to *expand*.

```
> cd /tmp/to/dist; echo $cwd
/tmp/to/dst
> cd ..; echo $cwd
/tmp/to
> cd /tmp/to/dist; echo $cwd
/tmp/to/dst
> cd ".."; echo $cwd
/tmp/from
> /bin/echo ..
/tmp/to
> /bin/echo ".."
..
```

expand expansion:

1. works just like *ignore* for built-ins like **cd**,
2. is prevented by quoting, and
3. happens before filenames are passed to non-built-in commands.

time

If set to a number, then the **time** built-in command executes automatically after each command which takes more than that many CPU seconds. If there is a second word, it is used as a format string for the output of the time built-in. The following sequences may be used in the format string:

%U The time the process spent in user mode in cpu seconds.

%S The time the process spent in kernel mode in cpu seconds.

%E The elapsed (wall clock) time in seconds.

%P The CPU percentage computed as (%U + %S) / %E.

%W The number of times the process was swapped.

%X The average amount in (shared) text space used in Kbytes.

%D The average amount in (unshared) data/stack space used in Kbytes.

%K The total space used (%X + %D) in Kbytes.

%M The maximum memory the process had in use at any time in Kbytes.

%F The number of major page faults (page needed to be brought from disk).

%R The number of minor page faults.

%I The number of input operations.

%O The number of output operations.

%r The number of socket messages received.

%s The number of socket messages sent.

%k The number of signals received.

%w The number of voluntary context switches (waits).

%c The number of involuntary context switches.

Only the first four sequences are supported on systems without BSD resource limit functions. The default time format is

```
Uu   %Ss %E %P %X+%Dk %I+%Oio   %Fpf+%Ww
```

for systems that support resource usage reporting.

The following table contains a list of tcsh environment variables.

Table 3. tcsh Environment Variables

Environment Variable	Purpose
COLUMNS	A list of directories in which **cd** should search for subdirectories if they aren't found in the current directory.
DISPLAY	Used by X Window System. If set, the shell does not set **autologout**.
EDITOR	The pathname to a default editor. See also the VISUAL environment variable and the **run-fg-editor** editor command.
GROUP	Equivalent to the **group** shell variable.
HOME	Equivalent to the **home** shell variable..
HOST	Initialized to the name of the machine on which the shell is running, as determined by the gethostname system call.
HOSTTYPE	Initialized to the type of the machine on which the shell is running, as determined at compile time. This variable is obsolete and will be removed in a future version.
HPATH	A colon-separated list of directories in which the run-help editor command looks for command documentation.
LANG	Gives the preferred character environment. See “National Language System Report” on page 153.
LC_CTYPE	If set, only ctype character handling is changed. See “National Language System Report” on page 153.
LINES	The number of lines in the terminal. See “Terminal Management” on page 153.
MACHTYPE	The machine type (microprocessor class or machine model), as determined at compile time.
NOREBIND	If set, printable characters are not rebound to **self-insert-command**. After a user sets NOREBIND, a new shell must be started. See “National Language System Report” on page 153.
OSTYPE	The operating system, as determined at compile time.
PATH	A colon-separated list of directories in which to look for executables. Equivalent to the **path** shell variable, but in a different format..
PWD	Equivalent to the **cwd** shell variable, but not synchronized to it; updated only after an acutal directory change.
REMOTE-HOST	The host from which the user has logged in remotely, if this is the case and the shell is able to determine it. (The OS/390 tcsh shell is not currently compiled with REMOTEHOST defined; see the **version** shell variable).
SHLVL	Equivalent to the **shlvl** shell variable.
TERM	Equivalent to the **term** shell variable.
USER	Equivalent to the **user** shell variable.
VENDOR	The vendor, as determined at compile time.
VISUAL	The pathname to a default full-screen editor. See the EDITOR environment variable and the **run-fg-editor** editor command.

tcsh Files

/etc/csh.cshrc
Read first by every shell.

/etc/csh.login
Read by login shells after /etc/csh.cshrc..

~/.tcshrc
Read by every shell after **/etc/csh.cshrc** or its equivalent.

~/.history
Read by login shells after **~/.tcshrc** if **savehist** is set. See also **histfile**.

~/.login
The shell reads ~/.login after **~/.tcshrc** and **~/.history**. See the **version** shell variable.

~/.cshdirs
Read by login shells after **~/.login** if **savedirs** is set. See also **dirsfile**.

~/.logout
Read by login shells at logout.

/bin/sh
Used to interpret shell scripts not starting with a #.

/tmp/sh*
Temporary file for < <.

tcsh shell: Problems and Limitations

When a suspended command is restarted, the tcsh shell prints the directory it started in if this is different from the current directory. This can be misleading (that is, wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/restartable. Command sequences of the form 'a ; b ; c' are also not handled gracefully when stopping is attempted. If you suspend 'b', the tcsh shell will then immediately execute 'c'. This is especially noticeable if this expansion results from an alias. It suffices to place the sequence of commands in ()'s to force it to a subshell, for example, (a ; b ; c).

Control over tty output after processes are started is primitive. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops are not placed in the history list. Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with |, and to be used with & and ; (semi-colon) metasyntax.

foreach doesn't ignore here documents when looking for its end.

It should be possible to use the : (colon) modifiers on the output of command substitutions.

The screen update for lines longer than the screen width is very poor if the terminal cannot move the cursor up (terminal type 'dumb').

It is not necessary for HPATH and NOREBIND to be environment variables.

Glob-patterns which do not use '?', '*' or '[]' or which use '{}' or '~' are not negated correctly.

The single-command form of **if** does output redirection even if the expression is false and the command is not executed.

ls-F includes file identification characters when sorting filenames and does not handle control characters in filenames well. It cannot be interrupted.

visiblebell shell variable is currently not implemented.

In filename and programmed completion, the 'C' completion rule word list type does not correctly select completion from the given directory.

There are three locales (codepages) which the tcsh shell will not correctly support: IBM-1388 (Chinese), IBM-933 (Korean) and IBM-937 (Traditional Chinese).

If you want to help maintain and test tcsh, send mail to listserv@mx.gw.com with the text 'subscribe tcsh '.

Limitations

Some limitations of the tcsh shell are:

- Words can be no longer than 1024 characters.
- The system limits argument lists to 10240 characters.
- The number of arguments to a command which involves filename expansion is limited to 1/6th the number of characters allowed in an argument list.
- Command substitutions may substitute no more characters than are allowed in an argument list.
- To detect looping, the shell restricts the number of alias substitutions on a single line to 20.

Related Information

: (colon), **@ (at)**, **alias**, **bg**, **break**, **cd**, **continue**, **echo**, **eval**, **exec**, **exit**, **fg**, **history**, **jobs**, **kill**, **newgrp**, **nice**, **nohup**, **printenv**, **set**, **shift**, **stop**, **suspend**, **time**, **umask**, **unalias**, **unset**, **wait**

% (percent) built-in command for tcsh: Move jobs to the foreground or background

Format

% [*job*] [*&*]

Description

%, is a synonym for the **fg** built-in command.

- **%** (percent) without arguments will bring the current job to the foreground.
- **%** specified with a job number attempts to bring that particular job to the foreground.
- **%** *job &* will move the specified job to the background. This syntax works the same as the **bg** built-in command. If no job is specified, the current job is moved to the background.

  Note: Current jobs will have a **+** next to the status column in jobs command output. See “Jobs” on page 151.

Related Information

jobs, **tcsh**

---

alloc built-in command for tcsh: Show the amount of dynamic memory acquired

Format

alloc *argument*

Description

Shows the amount of dynamic memory acquired, broken down into used and free memory. The argument shows the number of free and used blocks in each size category. The categories start at size 8 and double at each step.

Note: **alloc** is supported, but the output is not meaningful on OS/390.

Related Information

tcsh

---

bindkey built-in command for tcsh: List all bound keys

Format

bindkey [-l|-d|-e|-v|-u]
bindkey [-a] [-b] [-k] [-r] [– –] *key*
bindkey [-a] [-b] [-k] [-c|-s] [– –] *key command*

Description

bindkey specified alone (without options, *key*, or *key command*) lists all bound keys and the editor command to which each is bound.

bindkey specified with *key* (with or without options) lists the editor command to which key is bound.

bindkey specified with *key command* (with or without options) binds the editor *command* to *key*.

Options

–l Lists all commands and a short description of each.

–d Binds all keys to the standard bindings for the default editor.

–e Binds all keys to the standard GNU Emacs-like bindings.

–v Binds all keys to the standard vi-like bindings.

–a Lists or changes key-bindings in the alternative key map. This is the key map used in vi command mode.

–b *key* is interpreted as a control character written ^*character* ('^A') or C-*character* ('C-A'), a meta character written M-character ('M-A'), or an extended prefix key written X-character ('X-A').

–k *key* is interpreted as a symbolic arrow key name, which may be one of 'down', 'up', 'left' or 'right'.

–r Removes key's binding. Be careful: **bindkey -r** does not bind key to **self-insert-command**, it unbinds *key* completely.

–c *command* is interpreted as a built-in or external command instead of an editor command.

–s *command* is taken as a literal string and treated as terminal input when *key* is typed. Bound keys in *command* are themselves reinterpreted, and this continues for ten levels of interpretation.

– – Forces a break from option processing, so the next word is taken as *key* even if it begins with '-'.

Usage Notes

1. *key* may be a single character or a string. If a command is bound to a string, the first character of the string is bound to **sequence-lead-in** and the entire string is bound to the command.
2. Control characters in *key* can be literal (they can be typed by preceding them with the editor command **quoted-insert**, normally bound to '^V') or written caret-character style, for example, '^A'. Delete is written '^?' (caret-question mark). *key* and *command* can contain backslashed escape sequences (in the style of System V echo) as follows:

\a Bell

\b Backspace

\e Escape

\f Form feed

\n Newline

\r Carriage return

\t Horizontal tab

\v Vertical tab

*****nnn* The EBCDIC character corresponding to the octal number nnn

'\' nullifies the special meaning of the following characters, notably '\' and '^'.

Related Information

tcsh

---

builtins built-in command for tcsh: Prints the names of all built-in commands

Format

builtins

Description

Prints the names of all built-in commands.

Related Information

tcsh

---

bye built-in command for tcsh: Terminate the login shell

Format

bye

Description

A synonym for the **logout** built-in command. (See the **version** shell variable.)

Related Information

logout

---

chdir built-in shell command for tcsh: Change the working directory

Format

chdir

Description

A synonym for the **cd** built-in command.

Related Information

cd, **tcsh**

---

complete built-in command for tcsh: List completions

Format

complete[*command* [*word*/*pattern*/*list*[*:select*]/[[*suffix*]/] ...]]

Description

complete, without arguments, lists all completions. With *command*, **complete** lists completions for command. With *command* and *word* etc., **complete** defines completions.

Arguments

command

command may be a full command name or a glob-pattern. See “Filename Substitution” on page 144. It can begin with – to indicate that completion should be used only when *command* is ambiguous.

word

word specifies which word relative to the current word is to be completed, and may be one of the following:

c Current-word completion. *pattern* is a glob-pattern which must match the beginning of the current word on the command line. *pattern* is ignored when completing the current word.

C Like **c**, but includes *pattern* when completing the current word.

n Next-word completion. *pattern* is a glob-pattern which must match the beginning of the previous word on the command line.

N Like **n**, but must match the beginning of the word two before the current word.

p Position-dependent completion. *pattern* is a numeric range, with the same syntax used to index shell variables, which must include the current word.

list The list of possible completions, which may be one of the following:

a Aliases

b Bindings (editor commands)

d Directories

D Directories which begin with the supplied path prefix

e Environment variables

f Filenames

F Filenames which begin with the supplied path prefix

g Groupnames

j Jobs

l Limits

n Nothing

s Shell variables

S Signals

t Plain (text) files

T	Plain (text) files which begin with the supplied path prefix
v	Any variables
u	Usernames
x	Like **n**, but prints select when **list-choices** is used
X	Completions
$var	Words from the variable **var**
(...)	Words from the given list
...	Words from the output of command

select

select is an optional glob-pattern. If given, only words from *list* which match select are considered and the **fignor**e shell variable is ignored. The last three types of completion may not have a select pattern, and **x** uses select as an explanatory message when the **list-choices** editor command is used.

suffix

suffix is a single character to be appended to a successful completion. If null, no character is appended. If omitted (in which case the fourth delimiter can also be omitted), a slash is appended to directories and a space to other words.

Examples

1. Some commands take only directories as arguments, so there's no point in completing plain files. For example:

   ```
   > complete cd 'p/1/d/'
   ```

   completes only the first word following **cd** (*p/1*) with a directory.

2. **p**-type completion can be used to narrow down command completion. For example:

   ```
   > co[^D]
   complete compress
   > complete -co* 'p/0/(compress)/'
   > co[^D]
   > compress
   ```

   This completion completes commands (words in position *0*, *p/0*) which begin with *co* (thus matching *co**) to *compress* (the only word in the list). The leading - indicates that this completion is to be used only with ambiguous commands.

3.

   ```
   > complete find 'n/-user/u/'
   ```

   This is an example of **n**-type completion. Any word following *find* and immediately following *-user* is completed from the list of users.

4.

   ```
   > complete cc 'c/-I/d/'
   ```

   This demonstrates **c**-type completion. Any word following *cc* and beginning with *-I* is completed as a directory. *-I* is not taken as part of the directory because we used lowercase **c**.

5. Different *lists* are useful with different commands:

```
> complete alias 'p/1/a/'
> complete man 'p/*/c/'
> complete set 'p/1/s/'
> complete true 'p/1/x:Truth has no options./'
```

These complete words following **alias** with aliases, **man** with commands, and **set** with shell variables. **true** doesn't have any options, so **x** does nothing when completion is attempted and prints 'Truth has no options.' when completion choices are listed.

The **man** example, and several other examples below, could just as well have used *c/** or *n/** as *p/**.

6. Words can be completed from a variable evaluated at completion time,

```
> complete ftp 'p/1/$hostnames/'
> set hostnames = (rtfm.mit.edu tesla.ee.cornell.edu)
> ftp [^D]
rtfm.mit.edu tesla.ee.cornell.edu
> ftp [^C]
> set hostnames = (rtfm.mit.edu tesla.ee.cornell.edu uunet.uu.net)
> ftp [^D]
rtfm.mit.edu tesla.ee.cornell.edu uunet.uu.net
```

or from a command run at completion time:

```
> complete kill 'p/*/'ps | awk \{print\ \$1\}'/'
> kill -9 [^D]
23113 23377 23380 23406 23429 23529 23530 PID
```

The **complete** command does not itself quote its arguments, so the braces, space and $ in {print $1} must be quoted explicitly.

7. One command can have multiple completions:

```
> complete dbx 'p/2/(core)/' 'p/*/c/'
```

This example completes the second argument to **dbx** with the word *core* and all other arguments with commands. The positional completion is specified before the next-word completion. Since completions are evaluated from left to right, if the next-word completion were specified first it would always match and the positional completion would never be executed. This is a common mistake when defining a completion.

8. The select pattern is useful when a command takes only files with particular forms as arguments. For example,

```
> complete cc 'p/*/f:*.[cao]/'
```

completes *cc* arguments only to files ending in **.c**, **.a**, or **.o**. *select* can also exclude files, using negation of a glob-pattern as described under “Filename Substitution” on page 144.

9. One might use

```
> complete rm 'p/*/f:^*.{c,h,cc,C,tex,1,man,l,y}/'
```

to exclude precious source code from **rm** completion. Of course, one could still type excluded names manually or override the completion mechanism using the **complete-word-raw** or **list-choices-raw** editor command.

10. The **D, F** and **T** lists are like **d, f** and **t** respectively, but they use the select argument in a different way: to restrict completion to files beginning with a

particular path prefix. For example, the Elm mail program uses **=** as an abbreviation for one's mail directory. One might use

```
> complete elm c@=@F:$HOME/Mail/@
```

to complete *elm -f =* as if it were *elm -f ~/Mail/*. We used @ instead of / to avoid confusion with the select argument, and we used $HOME instead of ~ because home directory substitution only works at the beginning of a word.

11. *suffix* is used to add a nonstandard suffix (not space or '/' for directories) to completed words. For example,

    ```
    > complete finger 'c/*@/$hostnames/' 'p/1/u/@'
    ```

    completes arguments to *finger* from the list of users, appends an @, and then completes after the @ from the **hostnames** variable. Note the order in which the completions are specified.

12. A more complex example:

    ```
    complete find \
    'n/-name/f/' 'n/-newer/f/' 'n/-{,n}cpio/f/' \
    'n/-exec/c/' 'n/-ok/c/' 'n/-user/u/' \
    'n/-group/g/' 'n/-fstype/(nfs 4.2)/' \
    'n/-type/(b c d f l p s)/' \
    'c/-/(name newer cpio ncpio exec ok user \
    group fstype type atime ctime depth inum \
    ls mtime nogroup nouser perm print prune \
    size xdev)/' \
    'p/*/d/'
    ```

    This completes words following -name, -newer, -cpio or ncpio (note the pattern which matches both) to files, words following -exec or -ok to commands, words following user and group to users and groups respectively and words following -fstype or -type to members of the given lists. It also completes the switches themselves from the given list (note the use of c-type completion) and completes anything not otherwise completed to a directory.

    Programmed completions are ignored if the word being completed is a tilde substitution (beginning with ~) or a variable (beginning with $). **complete** is an experimental feature, and the syntax may change in future versions of the shell. See also the **uncomplete** built-in command.

Related Information

tcsh, **uncomplete**

---

dirs built-in command for tcsh: Print the directory stack

Format

dirs [**-l**] [**-n**|**-v**]
dirs -S|**-L** [*filename*]
dirs -c

Description

dirs used alone prints the directory stack in the following format: The top of the stack is at the left and the first directory in the stack is the current directory. For example:

```
 > cd <========== # Change to home dir
 > pushd /bin <== # Change dir to /bin and add /bin to dir stack
 /bin ~
 > pushd /tmp <== # Change dir to /tmp and add /tmp to dir stack
 /tmp /bin ~
 > dirs <======== # Display current dir stack
 /tmp /bin ~
 > dirs -l <===== # Display in expanded (long) format
 /tmp /bin /u/erinf
 > dirs -v <===== # Display in verbose format
 0 /tmp
 1 /bin
 2 ~
 > popd <======== # Change dir back to /bin and remove /tmp from dir stack
 /bin ~
 >pwd
 /bin
Note:  dir=directory
```

Options

–l Output is expanded explicitly to home or the pathname of the home directory for the user.

-n Entries are wrapped before they reach the edge of the screen.

-v Entries are printed one per line, preceded by their stack postions.

If more than one of **-n** or **-v** is given, **-v** takes precedence.

-S Saves the directory stack to filename as a series of **cd** and **pushd** commands.

-L The tcsh shell sources filename, which is presumably a directory stack file saved by the **-S** option or the **savedirs** mechanism. In either case, **dirsfile** is used if filename is not given and **~/.cshdirs** is used if dirsfile is unset.

Login shells do the equivalent of **dirs -L** on startup and, if **savedirs** is set, you should issue **dirs -S** before exiting. Because only **~/.tcshrc** is normally sourced before **~/.cshdirs**, **dirsfile** should be set in **~/.tcshrc** rather than **~/.login**.

–c Clear the directory stack.

Related Information

tcsh

echotc built-in command for tcsh: Exercise the terminal capabilities in args

Format

echotc [**-sv**] **arg** ...

Description

echotc uses the terminal capabilities in args. For example, **echotc** cm 3 10 sends it to column 3 and row 10.

If arg is *baud*, *cols*, *lines*, *meta* or *tabs*, **echotc** prints the value of that capability (either yes or no, which indicates that the terminal does or does not have that capability). You might use this to make the output from a shell script less verbose on slow terminals, or limit command output to the number of lines on the screen:

```
> set history=`echotc lines`
> @ history--
```

Termcap strings may contain wildcards which will not echo correctly. One should use double quotes when setting a shell variable to a terminal capability string, as in the following example that places the date in the status line:

```
> set standout=`echotc sõ
> set end_standout=`echotc se`
> echo -n "$standout"; date; echo -n "$end_standout"
Mon Oct 25 10:06:48 EDT 1999
>
```

Note: The date, as indicated above, is printed out in standout mode.

The **infocmp** command can be used to print the current terminal description in termcap format (instead of terminfo format).

Options

-s Nonexistent capabilities return the empty string rather than causing an error.

-v Messages are verbose.

Related Information

tcsh

filetest built-in command for tcsh: Apply the op file inquiry operator to a file

Format

filetest *-op file –*

Description

filetest applies *op* (which is a file inquiry operator) to each file and returns the results as a space-separated list. For more information on file inquiry operators, see "File Inquiry Operators" on page 149.

Related Information

tcsh

---

glob built-in command for tcsh: Write each word to standard output

Format

glob *wordlist*

Description

glob is like **echo**, but no \ (backslash) escapes are recognized and words are delimited by null characters in the output. This command is useful for programs which wish to use the shell to filename expand a list of words.

Related Information

echo, **tcsh**

---

hashstat built-in command for tcsh: Print a statistic line on hash table effectiveness

Format

hashstat

Description

hashstat prints a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding exec's). An exec is attempted for each component of the path where the hash function indicates a possible hit, and in each component which does not begin with a / (forward slash).

OS/390 systems have a vfork() command, however, tcsh is not compiled to use it. Typically on machines without vfork, **hashstat** prints only the number and size of hash buckets, but on OS/390 systems, a **hashstat** print out would contain this:

```
> hashstat
> hashstat 512 hash buckets of 8 bits each
>
```

Related Information

tcsh

hup built-in command for tcsh: Run command so it exits on a hang-up signal

Format

hup [*command*]

Description

With *command*, **hup** runs the command such that it will exit on a hangup signal and arranges for the shell to send it a hangup signal when the shell exits. Commands may set their own response to hangups, overriding **hup**. Without an argument (allowed only in a shell script), **hup** causes the shell to exit on a hangup for the remainder of the script. See “Signal Handling” on page 153.

Related Information

nohup, **tcsh**

limit built-in command for tcsh: Limit consumption of processes

Format

limit [**–h**] [*resource* [*maximum-use*]]

Description

limit limits the consumption by the current process and each process it creates to not individually exceed maximum-use on the specified resource. If no *maximum-use* is given, then the current limit is printed; if no *resource* is given, then all limitations are given. If the **-h** flag is given, the hard limits are used instead of the current limits. The hard limits impose a ceiling on the values of the current limits. All hard limits can be raised only by a process which has superuser authority (except for **coredumpsize, vmemoryuse**, and **descriptors**), but a user may lower or raise the current limits within the legal range.

Controllable resources currently include:

cputime
: The maximum number of cpu-seconds to be used by each process.

filesize
: The largest single file which can be created.

datasize
: The maximum growth of the data+stack region via **sbrk** beyond the end of the program text.

stacksize
: The maximum size of the automatically-extended stack region.

coredumpsize
: The size of the largest core dump that will be created.

memoryuse
: The maximum amount of physical memory a process may have allocated to it at a given time.

maximum-use may be given as a (floating point or integer) number followed by a scale factor. For all limits other than *cputime* the default scale is k or kilobytes (1024 bytes); a scale factor of m or megabytes may also be used. For *cputime* the default scaling is seconds, while m for minutes or h for hours, or a time of the form mm:ss giving minutes and seconds may be used.

For both *resource* names and scale factors, unambiguous prefixes of the names suffice.

Related Information

tcsh, **ulimit**, **unlimit**

Also see **setrlimit()** in *OS/390 C/C++ Run-Time Library Reference*.

---

log built-in command for tcsh: Print the watch tcsh shell variable

Format

log

Description

Prints the **watch** shell variable and reports on every user indicated in **watch** that is logged in, regardless of when they last logged in.

Note: The OS/390 tcsh shell is compiled to use **watchlog**. If you attempt to use **log** on an OS/390 system, you will get an error that says "Command not found".

Related Information

tcsh, **watchlog**

---

login built-in command for tcsh: Terminate a login shell

Format

login

Description

login terminates a login shell, replacing it with an instance of **/bin/login**. This is one way to log off (included for compatibility with **sh**).

Related Information

logout, **tcsh**

logout built-in command for tcsh: Terminate a login shell

Format

logout

Description

logout terminates a login shell. Especially useful if **ignoreeof** is set.

Related Information

login, **tcsh**

ls-F built-in command for tcsh: List files

Format

ls-F [*-switch ...*] [*file ...*]

Description

In the tcsh shell, **ls-F** lists files like **ls -F**, but is much faster than **ls-F**. It identifies each type of special file in the listing with a special character:

/ Directory

***** Executable

Block device

% Character device

| Named pipe

= Socket

@ Symbolic link

If the **listlinks** shell variable is set, symbolic links are identified in more detail (only, of course, on systems which have them):

@ Symbolic link to a non-directory

> Symbolic link to a directory

& Symbolic link to nowhere

listlinks also slows down **ls-F**.

If you use files which are set-up as follows:

```
#creating a file
   touch file1
#creating a symbolic link to the file
   ln -s file1 link1
#creating a directory
   mkdir dir1
#creating a symbolic link to the directory
   ln -s dir1 linkdir1
#creating a symbolic link to a file that doesn't exist
   ln -s noexist linktonowhere
```

when you issue an **ls-F** with **listlinks** unset, you will get the following output:

```
> ls-F
dir1/ file1 link1@ linkdir1@ linktonowhere@
>
```

with **listlinks** set:

```
> set listlinks
> ls-F
dir1/ file1 link1@ linkdir1> linktonowhere&
>
```

If the **listflags** shell variable is set to x, a or A, or any combination thereof (for example, xA), they are used as flags to **ls-F**, making it act like **ls -xF, ls -Fa, ls -FA** or a combination **ls -Fx**A. On OS/390, **ls -C** is the default. However, on machines where **ls -C** is not the default, **ls-F** acts like **ls -CF**, unless **listflags** contains an x, in which case it acts like **ls -xF**.

See “tcsh — Invoke a C shell” on page 127.

Usage Note

To view an online manual description for the **ls-F** command, you must type **ls-F** without the dash. So, to see the man page you would issue:

```
man lsF
```

Related Information

ls, **tcsh**

---

notify built-in command for tcsh: Notify user of job status changes

Format

notify [*%job ...*]

Description

notify causes the shell to notify the user asynchronously when the status of any of the specified jobs (or, without *%job*, the current job) changes, instead of waiting until the next prompt. *job* may be a number, a string, ", %, + or '-' as described under “Jobs” on page 151. See also the **notify** shell variable.

Related Information

tcsh

---

onintr built-in command for tcsh: Control the action of the tcsh shell on interrupts

Format

onintr [-|*label*]

Description

onintr controls the action of the shell on interrupts. Without arguments, **onintr** restores the default action of the shell on interrupts, which is to terminate shell scripts or to return to the terminal command input level. With '-', causes all interrupts to be ignored. With *label*, causes the shell to execute a **goto** *label* when an interrupt is received or a child process terminates because it was interrupted.

onintr is ignored if the shell is running detached and in system startup files, where interrupts are disabled anyway.

Related Information

goto, **tcsh**

---

popd built-in command for tcsh: Pop the directory stack

Format

popd [**-p**] [**-l**] [**-n**|**-v**] [*+n*]

Description

popd without options, pops the directory stack and returns to the new top directory. With a number *+n*, discards the *n*'th entry in the stack. All forms of **popd** print the final directory stack, just like **dirs**. The **pushdsilent** shell variable can be set to prevent this.

Options

–l Output is expanded explicitly to home or the pathname of the home directory for the user.

-n Entries are wrapped before they reach the edge of the screen.

-p Overrides **pushdsilent**.

-v Entries are printed one per line, preceded by their stack postions.

If more than one of **-n** or **-v** is given, **-v** takes precedence.

Related Information

tcsh

---

pushd built-in command for tcsh: Make exchanges within directory stack

Format

pushd [**-p**] [**-l**] [**-n**|**-v**] [*name*| *+n*]

Description

pushd with options, exchanges the top two elements of the directory stack. If **pushdtohome** is set, **pushd** without arguments does *pushd ~*, like **cd**. With *name*, **pushd** pushes the current working directory onto the directory stack and changes to name. If name is '-', it is interpreted as the previous working directory (see “Filename Substitution” on page 144). If **dunique** is set, **pushd** removes any instances of *name* from the stack before pushing it onto the stack. With a number *+n*, **pushd** rotates the *n*'th element of the directory stack around to be the top element and changes to it. If **dextract** is set, however, **pushd** *+n* extracts the *n*'th directory, pushes it onto the top of the stack and changes to it. So, instead of just rotating the entire stack around, **dextract** lets the user have the *n*'th directory extracted from its current position, and pushes it onto the top. For example:

```
> pushd /tmp
/tmp ~
> pushd /bin
/bin /tmp ~
> pushd /u
/u /bin /tmp ~
> pushd /usr
/usr /u /bin /tmp ~
> pushd +2
/bin /tmp ~ /usr /u
> set dextract
> dirs
/bin /tmp ~ /usr /u
> pushd +2
~ /bin /tmp /usr /u
>
```

Finally, all forms of **pushd** print the final directory stack, just like **dirs**. The **pushdsilent** tcsh shell variable can be set to prevent this.

Options

–l Output is expanded explicitly to home or the pathname of the home directory for the user.

-n Entries are wrapped before they reach the edge of the screen.

-p Overrides **pushdsilent**.

-v Entries are printed one per line, preceded by their stack postions.

If more than one of **-n** or **-v** is given, **-v** takes precedence.

Related Information

cd, **tcsh**

---

rehash built-in command for tcsh: Recompute internal hash table

Format

rehash

Description

rehash causes the internal hash table of the contents of the directories in the **path** variable to be recomputed. This is needed if new commands are added to directories in path while you are logged in. This should only be necessary if you add commands to one of your own directories, or if a systems programmer changes the contents of one of the system directories. Also flushes the cache of home directories built by tilde (~) expansion.

Related Information

hashstat, **tcsh**

---

repeat built-in command for tcsh: Execute command count times

Format

repeat *count command*

Description

The specified *command* is executed *count* times. **repeat** is subject to the same restrictions as the command in the one line **if** statement. I/O redirections occur exactly once, even if *count* is 0.

Related Information

tcsh

---

sched built-in command for tcsh: Print scheduled event list

Format

sched
sched *hh:mm command*
sched *n*

Description

sched used alone prints the scheduled-event list. The sched shell variable may be set to define the format in which the scheduled-event list is printed. **sched** *hh:mm command* adds command to the scheduled-event list. For example:

```
 >sched 11:00 echo It\'s eleven o\'clock.
```

causes the shell to echo 'It's eleven o'clock.' at 11 AM. The time may be in 12-hour AM/PM format

```
>sched 5pm set prompt='[%h] It\'s after 5; go home: >'
```

or may be relative to the current time:

```
>sched +2:15 /usr/lib/uucp/uucico -r1 -sother
```

A relative time specification may not use AM/PM format. The third form removes item *n* from the event list:

```
> sched
1 Wed Apr 4 15:42 /usr/lib/uucp/uucico -r1 -sother
2 Wed Apr 4 17:00 set prompt=[%h] It's after 5; go home: >
> sched -2
> sched
1 Wed Apr 4 15:42 /usr/lib/uucp/uucico -r1 -sother
```

A command in the scheduled-event list is executed just before the first prompt is printed after the time when the command is scheduled. It is possible to miss the exact time when the command is to be run, but an overdue command will execute at the next prompt. A command which comes due while the shell is waiting for user input is executed immediately. However, normal operation of an already-running command will not be interrupted so that a scheduled-event list element may be run.

This mechanism is similar to, but not the same as, the **at** command on some UNIX systems. Its major disadvantage is that it may not run a command at exactly the specified time. Its major advantage is that because **sched** runs directly from the shell, it has access to shell variables and other structures. This provides a mechanism for changing one's working environment based on the time of day.

Related Information

tcsh

---

setenv built-in command for tcsh: Set environment variable name to value

Format

setenv [*name* [*value*]]

Description

setenv without arguments, prints the names and values of all environment variables. Given *name*, sets the environment variable name to *value* or, without *value*, to the null string.

Related Information

tcsh

---

settc built-in command for tcsh: Tell tcsh shell the terminal capability cap value

Format

settc *cap value*

Description

settc tells the tcsh shell to believe that the terminal capability cap (as defined in **termcap**) has the value *value*. No sanity checking is done. Concept terminal users may have to **settc** *xn no* to get proper wrapping at the rightmost column.

Related Information

tcsh

---

setty built-in command for tcsh: Control tty mode changes

Format

setty [**-d|-q|-x**] [**-a**] [+|-]*mode*]

Description

setty controls which tty modes the shell does not allow to change. Without arguments, **setty** lists the modes in the chosen set which are fixed on (+mode) or off (-mode). The available modes, and thus the display, vary from system to system. With +mode, -mode or mode, fixes mode on or off or removes control from mode in the chosen set. For example, **setty** *+echok echoe* fixes echok mode on and allows commands to turn echoe mode on or off, both when the shell is executing commands.

Options

–a List all tty modes in the chosen set whether or not they are fixed.

[-d|-q|-x]
Tells **setty** to act on the edit, quote or execute set of tty modes respectively; without **-d, -q or -x**, execute is used.

Related Information

tcsh

source built-in command for tcsh: Read and execute commands from name

Format

source [**-h**] *name* [*args ...*]

Description

Using **source**, the shell reads and executes commands from *name*. The commands are not placed on the history list. If any arguments are given, they are placed in **argv**. **source** commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a source at any level terminates all nested source commands.

Options

–h Commands are placed on the history list instead of being executed, much like **history -L**.

Related Information

history, **tcsh**

telltc built-in command for tcsh: List terminal capability values

Format

telltc

Description

telltc lists the values of all terminal capabilities.

Related Information

tcsh

uncomplete built-in command for tcsh: Remove completions whose names match pattern

Format

uncomplete *pattern*

Description

uncomplete removes all completions whose names match pattern. For example, **uncomplete** * removes all completions. It is not an error for nothing to be **uncomplete**d.

Related Information

complete, **tcsh**

---

unhash built-in command for tcsh: Disable use of internal hash table

Format

unhash

Description

unhash disables use of the internal hash table to speed location of executed programs.

Related Information

tcsh

---

unlimit built-in command for tcsh: Remove resource limitations

Format

unlimit [**-h**] [*resource*]

Description

unlimit removes the limitation on *resource* or, if no resource is specified, all resource limitations.

The hard limit may be lowered to any value that is greater than or equal to the soft limit. All hard limits can be raised only by a process which has superuser authority except for **coredumpsize**, **vmemoryuse**, and **descriptors**. This behavior is identical to **ulimit** in the OS/390 shell. Both the soft limit and hard limit can be changed by a single call to **setrlimit()**.

Options

–h Corresponding hard limits are removed. Only the superuser may do this.

Related Information

limit, **tcsh**, **ulimit**

Also see **setrlimit()** in *OS/390 C/C++ Run-Time Library Reference*.

---

unsetenv built-in command for tcsh: Remove environmental variables that match pattern

Format

unsetenv *pattern*

Description

unsetenv removes all environment variables whose names match pattern. For example, **unsetenv** * removes all environment variables; we **strongly** recommend against this. It is not an error for nothing to be **unsetenv**ed.

Related Information

setenv, **tcsh**

---

watchlog built-in command for tcsh: Print the watch shell variable

Format

watchlog

Description

watch is an alternate name for the **log** built-in command. It prints the **watch** shell variable and reports on every user indicated in **watch** that is logged in, regardless of when they last logged in.

See the **version** shell variable.

Related Information

log, **tcsh**

---

where built-in command for tcsh: Report all instances of command

Format

where *command*

Description

where reports all known instances of *command*, including aliases, built-ins and executables in path.

Related Information

tcsh, **which**

---

which built-in command for tcsh: Display next executed command

Format

which *command*

Description

which displays the command that will be executed by the shell after substitutions, path searching, and so on. This command correctly reports tcsh aliases and built-ins. See also the **which-command** editor command.

Related Information

tcsh, **where**

---

time — Display processor and elapsed times for a command

Format

time [**–p**] *command-line*

tcsh shell: **time** [*command*]

Description

time runs the command given as its argument and produces a breakdown of total time to run (`real`), total time spent in the user program (`user`), and total time spent in system processor overhead (`sys`).

Times given are statistical, based on where execution is at a clock tick. Output is written to standard error (**stderr**).

time in the tcsh shell

time executes *command* (which must be a simple command, not an alias, a pipeline, a command list, or a parenthesized command list) and prints a time summary as described under the tcsh **time** variable (see "tcsh — Invoke a C shell" on page 127). If necessary, an extra shell is created to print the time statistic when the *command* completes. Without *command*, **time** prints a time summary for the current shell and its children.

Option

–p Guarantees that the historical format of the **time** command is output.

Usage Note

time is a built-in shell command.

Localization

time uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **LC_NUMERIC**
- **NLSPATH**

Exit Values

If **time** successfully invokes *command-line*, it returns the exit status of *command-line*. Otherwise, possible exit status values are:

0 Successful completion
1 An error occurred in the **time** utility
2 Failure due to an invalid command-line option
2 Invalid command-line argument
126 **time** found *command* but could not invoke it
127 **time** could not find *command*

Portability

POSIX.2 User Portability Extension, X/Open Portability Guide, UNIX systems.

Related Information

sh, **tcsh**

---

umask — Set or return the file mode creation mask

Format

umask [**–S**] [*mode*]

tcsh shell: **umask** [*value*]

Description

umask sets the file-creation permission-code mask of the invoking process to the given *mode*. You can specify the *mode* in any of the formats recognized by **chmod**; see **chmod** for more information.

The *mode* may be specified in symbolic (rwx) or octal format. The symbolic form specifies what permissions are allowed. The octal form specifies what permissions are disallowed.

The file-creation permission-code mask (often called the *umask*) modifies the default (initial) permissions for any file created by the process. The umask specifies the permissions which are **not** to be allowed.

If the bit is turned off in the umask, a process can set it on when it creates a file. If you specify:

```
umask a=rx
```

you have allowed files to be created with read and execute access for all users. If you were to look at the mask, it would be 0222. The write bit is set, because write is not allowed. If you want to permit created files to have read, write, and execute access, then set umask to 0000. If you call **umask** without a *mode* argument, **umask** displays the current umask.

umask in the tcsh shell

In the tcsh shell, **umask** sets the file creation mask to *value*, which is given in octal. Common values for the mask are `002`, giving all access to the group and read and execute access to others, and `022`, giving read and execute access to the group and others. Without *value*, **umask** prints the current file creation mask. See "tcsh — Invoke a C shell" on page 127.

Options

–S Displays the umask in a symbolic form:

`u=perms,g=perms,o=perms`

giving owner, group and other permissions. Permissions are specified as combinations of the letters **r** (read), **w** (write), and **x** (execute).

Localization

umask uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

`0` Successful completion
`1` Failure due to an incorrect command-line argument, or incorrect *mode*

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

chmod, **tcsh**

---

unalias — Remove alias definitions

Format

unalias *name ...*
unalias –a

tcsh shell: **unalias** *pattern*

Description

unalias removes each alias *name* from the current shell execution environment.

unalias in the tcsh shell

In the tcsh shell, **unalias** removes all aliases whose names match *pattern*. For example,

```
unalias *
```

removes all aliases. It is not an error for nothing to be **unalias**ed. See “tcsh — Invoke a C shell” on page 127.

Options

–a Removes all aliases in the current shell execution environment.

Localization

unalias uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Notes

unalias is a built-in shell command.

Exit Values

0 Successful completion

1 There was an alias that could not be removed

2 Failure due to an incorrect command-line option or there were two aliases that could not be removed

>2
Tells the number of aliases that could not be removed

Portability

POSIX.2 User Portability Extension, X/Open Portability Guide.

Related Information

alias, **sh**, **tcsh**

---

unset — Unset values and attributes of variables and functions

Format

unset *name ...*
unset –fv *name ...*

tcsh shell: **unset** *pattern*

Description

Calling **unset** with no options removes the value and attributes of each variable or function name.

unset in the tcsh shell

unset removes all variables whose names match pattern, unless they are read-only. For example:

```
unset *
```

which we **strongly** recommend you do not do, will remove all variables unless they are read-only. It is not an error for nothing to be **unset**.

See “tcsh — Invoke a C shell” on page 127.

Options

–f Removes the value and attributes of each function *name*.
–v Removes the attribute and value of the variable *name*. This is the default if no options are specified.

unset cannot remove names that have been set read-only.

Usage Notes

unset is a special built-in shell command.

Localization

unset uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_MESSAGES**
- **NLSPATH**

Exit Values

0 Successful completion
1 Failure due to an incorrect command-line option
2 Failure due to an incorrect command-line argument

Otherwise, **unset** returns the number of specified *names* that are incorrect, not currently set, or read-only.

Messages

Possible error messages include:

name **readonly variable**
The given *name* cannot be deleted because it has been marked read-only.

Portability

POSIX.2, X/Open Portability Guide.

Related Information

sh, **readonly**, **tcsh**

---

wait — Wait for a child process to end

Format

wait [*pid*|*job-id* ...]

tcsh shell: **wait**

Description

wait waits for one or more jobs or child processes to complete in the background. If you specify one or more *job-id* arguments, **wait** waits for all processes in each job to end. If you specify *pid*, **wait** waits for the child process with that process ID (PID) to end. If no child process has that process ID, **wait** returns immediately.

If you specify neither a *pid* nor a *job-id*, **wait** waits for the process IDs known to the invoking shell to complete.

wait in the tcsh shell

The tcsh shell waits for all background jobs. If the shell is interactive, an interrupt will disrupt the **wait** and cause the shell to print the names and job numbers of all outstanding jobs. See “tcsh — Invoke a C shell” on page 127.

Localization

wait uses the following localization environment variables:

- **LANG**
- **LC_ALL**
- **LC_CTYPE**
- **LC_MESSAGES**
- **NLSPATH**

Usage Notes

wait is a built-in shell command.

Exit Values

If you specified a *job-id* that has terminated or is unknown by the invoking shell, an error message and a return code of 127 is returned. If you specified a *pid* that has terminated or is unknown to the shell, a return code of 127 is returned. If a signal ended the process abnormally, the exit status is a value greater than 128 unique to that signal. Otherwise, possible exit status values are:

`0` Successful completion

`1–126`
An error occurred

127
A specified *pid* or *job-id* has terminated or is unknown by the invoking shell

Portability

POSIX.2, X/Open Portability Guide, UNIX systems.

Related Information

sleep, **tcsh**

Chapter 11. Localization

Internationalization enables you to work in a cultural context that is comfortable for you through locales, character sets, and a number of special environment variables. The process of adapting an internationalized application or program, particular to a language or cultural milieu, is termed *localization*.

A *locale* is the subset of your environment that deals with language and cultural conventions. It is made up of a number of categories, each of which is associated with an environment variable and controls a specific aspect of the environment. The following list shows the categories and their spheres of influence:

LC_COLLATE
Collating (sorting) order.

LC_CTYPE
Character classification and case conversion.

LC_MESSAGES
Formats of informative and diagnostic messages and interactive responses.

LC_MONETARY
Monetary formatting.

LC_NUMERIC
Numeric, nonmonetary formatting.

LC_TIME
Date and time formats.

LC_SYNTAX
EBCDIC-variant character encodings used by some C functions and utilities.

To give a locale control over a category, set the corresponding variable to the name of the locale. In addition to the environment variables associated with the categories, there are two other variables which are used in conjunction with localization, **LANG** and **LC_ALL**. All of these variables affect the performance of the shell commands. The general effects apply to most commands, but certain commands such as **sort**, with its dependence on **LC_COLLATE**, require special attention to be paid to one or more of the variables; this manual discusses such cases in the *Localization* section of the command. The effects of each environment variable is as follows:

LANG
Determines the international language value. Utilities and applications can use the information from the given locale to provide error messages and instructions in that locale's language. If **LC_ALL** variable is not defined, any undefined variable is treated as though it contained the value of **LANG**.

LC_ALL
Overrides the value of **LANG** and the values of any of the other variables starting with **LC_**.

LC_COLLATE
Identifies the locale that controls the collating (sorting) order of characters and determines the behavior of ranges, equivalence classes, and multicharacter collating elements.

LC_CTYPE
Identifies the locale that defines character classes (for example, *alpha, digit, blank*) and their behavior (for example, the mapping of lowercase letters to uppercase letters). This locale also determines the interpretation of sequences of bytes as characters (such as singlebyte versus doublebyte characters).

LC_MESSAGES
Identifies the locale that controls the processing of affirmative and negative responses. This locale also defines the language and cultural conventions used when writing messages.

LC_MONETARY
Determines the locale that controls monetary-related numeric formatting (for example, currency symbol, decimal point character, and thousands separator).

LC_NUMERIC
Determines the locale that controls numeric formatting (for example, decimal point character and thousands separator).

LC_TIME
Identifies the locale that determines the format of time and date strings.

LC_SYNTAX
Identifies the locale that defines the encodings for the variant characters in the portable character set.

The **NLSPATH** localization variable specifies where the message catalogs are to be found.

For example,

```
NLSPATH="/system/nlslib/%N.cat"
```

specifies that the OS/390 shell is to look for all message catalogs in the directory **/system/nlslib**, where the catalog name is to be constructed from the *name* parameter passed to the OS/390 shell with the suffix **.cat**.

Substitution fields consist of a % symbol, followed by a single-letter keyword. These keywords are currently defined:

%N The value of the *name* parameter
%L The value of the **LC_MESSAGES** category, or **LANG**, depending on how the catopen() function that opens this catalog is coded. For more information, refer to catopen() in *OS/390 C/C++ Run-Time Library Reference*.
%l The *language* element from the **LC_MESSAGES** category
%t The *territory* element from the **LC_MESSAGES** category
%c The *codeset* element from the **LC_MESSAGES** category

Templates defined in **NLSPATH** are separated by colons (:). A leading colon or two adjacent colons (::) are equivalent to specifying **%N**. For example:

```
NLSPATH=":%N.cat:/nlslib/%L/%N.cat"
```

specifies that the OS/390 shell should look for the requested message catalog in *name*, *name***.cat**, and **/nlslib**/*category*/*name***.cat**, where *category* is the value of the **LC_MESSAGES** or **LANG** category of the current locale.

Do not set the **NLSPATH** variable unless you need to override the default system path. Otherwise the commands may behave unpredictably.

Part 4. OS/390 UNIX System Services Messages and Codes

FSUC0501 Load average unavailable

Explanation: The **load average** editing command could not be completed.

System Action: Command ends.

FSUC0606 No matching command

Explanation: Command completion was not successful because the command does not exist.

System Action: Command ends.

User Response: Respecify statement with a valid command.

FSUC0607 Ambiguous command

Explanation: Command completion as not successful because more than one command matched the specifications.

System Action: Command ends.

User Response: Respecify command in a more precise manner.

FSUC0721 *program-name*: **No entry for terminal type** *string*

Explanation: There was no entry for the specified terminal type in the terminfo database.

System Action: Processing continues.

FSUC0722 *program-name*: **using dumb terminal settings.**

Explanation: No terminfo could be found, so a dumb terminal is being used.

System Action: Processing continues.

FSUC0801 Unknown switch

Explanation: An incorrect option was passed to the **setty** command. Valid options are: a, q, d, x.

System Action: Command ends.

User Response: Correct the syntax, and reissue statement.

FSUC0802 Invalid argument

Explanation: An incorrect argument was passed to the **setty** command.

System Action: Command ends.

User Response: Check the syntax, and reissue command.

FSUC0901 AddXkey: Null extended-key not allowed.

Explanation: A null extended-key was issued on the **bindkey** command.

System Action: Command ends.

User Response: Reissue the command using a non-null extended-key.

FSUC0902 AddXkey: sequence-lead-in command not allowed

Explanation: A **sequence-lead-in** command cannot be bound to multicharacter key binding.

System Action: Command ends.

User Response: Reissue statement with a different command.

FSUC0903 DeleteXkey: Null extended-key not allowed.

Explanation: A null extended-key was issued on the **bindkey -r** command.

System Action: Command ends.

User Response: Reissue the command using a non-null extended-key.

FSUC0904 Unbound extended key *key*

Explanation: The specified key on the **bindkey** command was not bound to anything.

System Action: Command ends.

User Response: Respecify command with the proper syntax.

FSUC0905 Some extended keys too long for internal print buffer

Explanation: The extended key was longer than the 95 character buffer limit.

System Action: Command ends.

FSUC0907 no input

Explanation: There is no specified function associated with this key.

System Action: Command ends.

FSUC0908 Something must follow: *string*

Explanation: The syntax of your **bindkey** command is not correct.

System Action: Command ends.

User Response: Check syntax, and reissue statement.

FSUC0909 Octal constant does not fit in a char.

Explanation: An octal constant was entered which is greater than 400.

System Action: Command ends.

User Response: Respecify command with an octal value less than 400.

FSUC1101 Warning: no access to tty (*string***).**

Explanation: You do not have access to **tty** job control. The process specified does not belong to a process in the same session with the **tty**.

User Response: Contact the system programmer.

System Programmer Response: **setpgid()** or **tcsetpgrp()** system call failed. These calls succeed only if processed by a super-user, or if id is the real or effective user(group) id of the calling process.

FSUC1102 Thus no job control in this shell.

Explanation: You do not have access to **tty** job control. The process specified does not belong to a process in the same session with the **tty**.

User Response: Contact the system programmer.

System Programmer Response: **setpgid()** or **tcsetpgrp()** system call failed. These calls succeed only if processed by a super-user, or if id is the real or effective user(group) id of the calling process.

FSUC1305 *string***: shell built-in command.**

Explanation: The command specified is a shell built-in command. It is a registered command but not found in alias.

FSUC1306 *string***: Command not found.**

Explanation: The command specified was not found. It is not a registered command nor an alias.

User Response: Check the syntax on the command issued, including options and arguments, and try again.

FSUC1307 where: / in command makes no sense.

Explanation: The command specified is not a valid command. Cannot process / in command.

User Response: Check the syntax on the command issued, including options and arguments, and try again.

FSUC1308 *string* **is aliased to.**

Explanation: If the command specified is an alias, then display its alias path.

FSUC1309 *string* **is a shell built-in.**

Explanation: The command specified is a shell built-in command.

FSUC1501 *string***:** *string***: Can't** *string string* **limit.**

Explanation: Unable to set/remove file size limits. Write to stderror file.

FSUC1607 Bad seek type *number***.**

Explanation: Bad seek type. Valid seek types are 0, 1, and 2.

User Response: Respecify command with valid seek type.

FSUC1701 BUG: waiting for background job!.

Explanation: Now keep pausing as long as we are not interrupted (SIGINT), and the target process, or any of its friends, are still running.

System Programmer Response: Processing continues.

User Response: Please wait for process to return.

FSUC1703 BUG: process flushed twice.

Explanation: Process id is 0.

System Programmer Response: Process is ended.

FSUC1708 BUG: status=*status*

Explanation: Unrecognized process status message received.

FSUC1709 (core dumped).

Explanation: Process ends with core dump.

FSUC1712 ***string*****: Already suspended.**

Explanation: Current shell is suspended/stopped.

FSUC1801 Warning: ridiculously long PATH truncated.

Explanation: Incorrect PATH specified. Exported path exceeds maximum buffer size.

FSUC1802 Warning: unknown multibyte display; using default(euc(JP)).

Explanation: Incorrect multibyte display type. Using default multibyte display (euc(JP)).

FSUC1803 Warning: unknown multibyte code ***number*****; multibyte disabled.**

Explanation: Incorrect multibyte code received. Multibyte disabled.

FSUC1804 Warning: Invalid multibyte table length (*number***); multibyte disabled.**

Explanation: Incorrect multibyte table length. Multibyte disabled.

FSUC1805 Warning: bad multibyte code at offset +*number***; multibyte disabled.**

Explanation: Bad multibyte code at offset. Multibyte disabled.

FSUC2001 Invalid key name ***string*****.**

Explanation: The specified key name is not valid.

FSUC2002 Bad key name: ***string*****.**

Explanation: The specified key name is not valid.

FSUC2003 Bad command name: ***string*****.**

Explanation: The command name is not valid.

FSUC2004 Bad key spec ***string*****.**

Explanation: Bad key specified.

FSUC2005 Null string specification.

Explanation: String is empty.

FSUC2203 Faulty alias precmd removed.

Explanation: You cannot alias **precmd**.

FSUC2204 Faulty alias cwdcmd removed.

Explanation: You cannot alias **cwdcmd**.

FSUC2205 Faulty alias beepcmd removed.

Explanation: You cannot alias **beepcmd**.

FSUC2206 Faulty alias periodic removed.

Explanation: You cannot alias **periodic**.

FSUC2323 getwd: Cannot stat / (*string***).**

Explanation: Unable to get status of / directory. Write to stderror file.

FSUC2324 getwd: Cannot stat . (*string***).**

Explanation: Unable to get status of . directory. Write to stderror file.

FSUC2325 getwd: Cannot stat directory *string* **(***string***).**

Explanation: Unable to get status of working directory. Write to stderror file.

FSUC2326 getwd: Cannot open directory *string* **(***string***).**

Explanation: Unable to open working directory. Write to stderror file.

FSUC2327 getwd: Cannot find . in .. (*string***).**

Explanation: Unable to find . in .. directory. Write to stderror file.

FSUC2502 error: bsd_signal(*number***) signal out of range.**

Explanation: Bsd signal is out of range.

User Response: Contact your system programmer.

System Programmer Response: Determine why bsd_signal was out of range.

FSUC2503 error: bsd_signal(*number***) - sigaction failed, errno** *number***.**

Explanation: Bsd signal failed.

User Response: Contact your system programmer.

System Programmer Response: Determine why bsd signal failed.

FSUC2601 cannot stat *string***. Please unset watch.**

Explanation: Unable to get temporary file status.

User Response: Verify that temporary file exists and _PATH_UTMP temporary file environmental variable has been set.

FSUC2602 *string* **cannot be opened. Please unset watch.**

Explanation: Unable to open temporary file.

User Response: Verify that temporary file exists and _PATH_UTMP temporary file environmental variable has been set.

FSUC2607 *name* **has** *terminal date* **from** *host***.**

Explanation: Display current element data with host field.

FSUC3004 *string*: **Internal match error.**

Explanation: An internal editing command error has occured.

System Action: Command ends.

User Response: Contact your system administrator.

System Programmer Response: Follow local procedures for reporting a problem to IBM.

FSUC3009 tcsh internal error: I don't know what I'm looking for!

Explanation: An internal error has occurred for a completion command.

System Action: Command ends.

User Response: Contact your system programmer.

System Programmer Response: Follow local procedures for reporting a problem to IBM.

FSUC3110 not a directory

Explanation: Completion cannot process successfully because the specified name is not a valid directory.

System Action: Command ends.

User Response: Reissue the command with a valid directory name.

FSUC3111 not found

Explanation: Completion cannot process successfully because the specified file/directory name cannot be found.

System Action: Command ends.

User Response: Reissue the command with a valid file/directory name.

FSUC3112 unreadable

Explanation: Completion cannot process successfully because the specified file/directory name cannot be read.

System Action: Command ends.

User Response: Change permissions of file/directory, or reissue the command with a different, readable file/directory.

FSUC5001 Syntax Error

Explanation: A command or construct was issued with incorrect syntax.

System Action: Command ends.

User Response: Check the syntax on the command or construct and reissue.

FSUC5002 *string* **is not allowed**

Explanation: You are not allowed to have a < or a numerical digit after a $?, $#, or $%.

System Action: Command ends.

User Response: Correct the syntax and reissue the command.

FSUC5003 Word too long

Explanation: Word used in $ expansion, command substitution or history substitution is more than the buffer can hold.

System Action: Command ends.

User Response: Try to split the expansion to use multiple smaller expansions.

FSUC5004 $< line too long

Explanation: The input value for $< is longer than the buffer allows.

System Action: Command ends.

User Response: Try to shorten the input and/or split input between multiple reads.

FSUC5005 No file for $0

Explanation: $0 is the name for the current shell input file. If unknown, this var is unset, and any reference to it is an error.

System Action: Command ends.

User Response: Set $0 and reissue command.

FSUC5006 Incomplete [modifier

Explanation: A newline or EOF indicator was reached before the ending].

System Action: Command ends.

User Response: Respecify command with correct syntax.

FSUC5007 $ expansion must end before “

Explanation: The $ expansion was incomplete before reaching the “ character.

System Action: Command ends.

User Response: Respecify command, placing the “ character after variable expansion.

FSUC5008 Bad : modifier in $ (%c)

Explanation: Valid modifiers are limited to luhtrqxes.

System Action: Command ends.

User Response: Respecify command with valid modifiers.

FSUC5009 Subscript error

Explanation: The closing “ on the array subscript was not found after a numerical value.

System Action: Command ends.

User Response: Correct the syntax and reissue the command.

FSUC5010 Badly formed number

Explanation: Statement indicated requires numerical value.

System Action: Command ends.

User Response: Check the syntax and reissue the statement.

FSUC5011 No more words

Explanation: argv or variable specified on **shift** command is either not set or has less than one word as value.

System Action: Command ends.

User Response: set the **shift** argument to have enough words, or stop using **shift** command when all words are shifted.

FSUC5012 Missing file name

Explanation: Command specified is expecting a filename to be passed as an argument.

System Action: Command ends.

User Response: Respecify the command with the appropriate filename.

FSUC5013 Internal glob error

Explanation: An internal glob error has occurred.

System Action: Command ends.

User Response: Contact the system programmer or try and reissue the statement without glob characters.

System Programmer Response: Follow your local procedures for reporting a problem to IBM.

FSUC5014 Command not found

Explanation: The command specified was not found in your search path.

System Action: Command ends.

User Response: Check if the command exists, change search path as necessary.

FSUC5015 Too few arguments

Explanation: Function specified requires more arguments than you have listed.

System Action: Command ends.

User Response: Check command syntax and reissue the statement.

FSUC5016 Too many arguments

Explanation: Function specified requires fewer arguments than you have listed.

System Action: Command ends.

User Response: Check the command syntax and reissue the statement.

FSUC5017 Too dangerous to alias that

Explanation: It is not valid to alias the commands **alias** and **unalias**.

System Action: Command ends.

User Response: Do not try and alias these commands.

FSUC5018 Empty if

Explanation: The value of the **if** command cannot be NULL.

System Action: Command ends.

User Response: Issue **if** statement with non-null expression.

FSUC5019 Improper then

Explanation: **then** statement must be followed by a command.

System Action: Command ends.

User Response: Reissue **then** followed by a valid command.

FSUC5020 Words not parenthesized

Explanation: The wordlist within the **foreach** statement must be enclosed in parenthesis.

System Action: Command ends.

User Response: Enclose the wordlist in parenthesis and reissue the statement.

FSUC5021 *string* **not found**

Explanation: Either a then, endif, endsw, end or a case label statement was not found.

System Action: Command ends.

User Response: Check the syntax of conditional statement, adding appropriate tag.

FSUC5022 Improper mask

Explanation: Masking values for the **umask** command must be between 0 and 777.

System Action: Command ends.

User Response: Reissue the **umask** command with the appropriate masking values.

FSUC5023 No such limit

Explanation: The resource value specified for the **limit** command does not exist. Controllable resources are: cputime, filesize, datasize, stacksize, coredumpsize, and memoryuse.

System Action: Command ends.

User Response: Reissue **limit** command with one of the resources listed above.

FSUC5024 Argument too large

Explanation: You have exceeded the maximum or minimum value defined on your system.

System Action: Command ends.

User Response: If possible, respecify argument within appropriate boundaries.

FSUC5025 Improper or unknown scale factor

Explanation: The scale factor for the maximum use field of the **limit** command is not valid. Valid values are either k for kilobytes, or m for megabytes.

System Action: Command ends.

User Response: Reissue **limit** command with an appropriate scale factor.

FSUC5026 Undefined variable

Explanation: Variable used in specified command is undefined.

System Action: Command ends.

User Response: Define variable with the **set** command before using.

FSUC5027 Directory stack not that deep

Explanation: The numerical value following the = is greater than the size of the directory stack.

System Action: Command ends.

User Response: You can find out how deep the directory stack is with the **dirs -v** command. Reissue =n where n is no greater than the largest stack value.

FSUC5028 Bad signal number

Explanation: The user specified an unknown signal number on the **kill** command.

System Action: Command ends.

User Response: Valid signal names and numbers are listed in *OS/390 UNIX System Services Command Reference* under the **kill** command.

FSUC5029 Unknown signal; kill -l lists signals

Explanation: The user specified an unknown signal on the **kill** command.

System Action: Command ends.

User Response: The -l option will list valid signal names. Reissue the command with a valid signal name.

FSUC5030 Variable name must begin with a letter

Explanation: The variable being initialized after the **set** command must begin with a letter.

System Action: Command ends.

User Response: Change name of variable so that a character occupies the first position.

FSUC5031 Variable name too long

Explanation: The variable name after the **set** command cannot exceed 30 characters in length.

System Action: Command ends.

User Response: Shorten variable name to less than 30 characters.

FSUC5032 Variable name must contain alphanumeric characters

Explanation: Variable name after the **set** command is expected to consist only of alphabetic characters, or a combination of alphabetic and numeric characters where the first letter in the variable name is alphabetic.

System Action: Command ends.

User Response: Change variable name to meet syntax guidelines.

FSUC5033 No job control in this shell

Explanation: This shell does not have job control capabilities.

System Action: Command ends.

User Response: Do not issue any job control commands.

FSUC5034 Expression Syntax

Explanation: Syntax of specified command is not correct.

System Action: Command ends.

User Response: Check syntax and respecify command.

FSUC5035 No home directory

Explanation: The $home variable is not set, therefore you cannot issue the **cd** or **chdir** command without any arguments.

System Action: Command ends.

User Response: Either set $home or specify a directory on the **cd** or **chdir** command.

FSUC5036 Can't change to home directory

Explanation: The $home variable is not set so using the ‘ ’ character to reference your home directory is not valid.

System Action: Command ends.

User Response: Either set $home or explicitly specify directory.

FSUC5037 Invalid null command.

Explanation: An unexpected NULL string was encountered.

System Action: Command ends.

User Response: Check syntax and reissue command.

FSUC5038 Assignment missing expression

Explanation: The **@ name=expr** command is missing the expr argument.

System Action: Command ends.

User Response: Reissue statement specifying expr argument.

FSUC5039 Unknown operator

Explanation: The operator used in the **@** command is not valid.

System Action: Command ends.

User Response: Check syntax and reissue statement.

FSUC5040 Ambiguous

Explanation: Specified function is ambiguous.

System Action: Command ends.

User Response: Check syntax, and reissue the statement.

FSUC5041 *filename*: **File exists**

Explanation: The specified file already exists and cannot be appended to or overwritten.

System Action: Command ends.

User Response: Use a different filename, or rename existing file.

FSUC5042 Argument for -c ends in backslash

Explanation: The -c tcsh option cannot be used with a script file that ends in a backslash.

System Action: Command ends.

User Response: Change name of script so that it does not end in a backslash.

FSUC5043 Interrupted

Explanation: A SIGINT has been received. Specified process has been interrupted.

System Action: Specified process has been interrupted.

FSUC5044 Subscript out of range

Explanation: User tried to access a value outside the scope of the array.

System Action: Command ends.

User Response: The **$#variable** command will tell you how many elements are in the array. Your subscript value must be an integer no greater than this value, but no less than one.

FSUC5045 Line overflow

Explanation: A line within the here-document notation exceeded the 1020 character limit.

System Action: Command ends.

User Response: Use multiple here-documents, so that you can split the input such that it fits within this character limit.

FSUC5046 No such job

Explanation: There is no job with the corresponding name/number.

System Action: Command ends.

User Response: The **jobs -l** command will list all current jobs, along with their corresponding process id's. Any job specified must be listed in the **jobs -l** output.

FSUC5047 Can't from terminal

Explanation: The **onintr** command cannot be issued from a terminal. The **hup** and **nohup** commands cannot be issued from a terminal without a corresponding command.

System Action: Command ends.

User Response: The **onintr** command can be issued from a script. The **hup** and **nohup** commands must be issued with a corresponding command, or can be issued without commands from a script.

FSUC5048 Not in while/foreach

Explanation: A **break**, **end**, or **continue** statement can only be issued from inside a **while** or **foreach** loop.

System Action: Command ends.

User Response: Check syntax of statement. Make any necessary changes and reissue.

FSUC5049 No more processes

Explanation: There are insufficient resources to create another process, or you have already reached the maximum number of processes you can run.

System Action: Command ends.

User Response: Contact your system administrator.

System Programmer Response: Determine why **fork()** failed.

FSUC5050 No match

Explanation: The wildcard expansion issued in your statement does not expand to a valid argument.

System Action: Command ends.

User Response: Be more explicit when issuing this statement.

FSUC5051 Missing *character*

Explanation: Statement missing either -, }, “, or).

System Action: Command ends.

User Response: Check syntax and respecify.

FSUC5052 Unmatched *character*

Explanation: A closing ' or " is missing from your statement.

User Response: Check syntax and respecify.

FSUC5053 Out of memory

Explanation: There were not enough system resources to allocate the required memory.

System Action: Command ends.

User Response: Free up more system resources and try again, or contact your system administrator for additional help.

FSUC5054 Can't make pipe

Explanation: **Pipe** command cannot be processed.

System Action: Command ends.

User Response: Check syntax and reissue statement.

FSUC5055 *function***:** *return-code*

Explanation: A system error has occured for the specified function.

System Action: Command ends.

User Response: A correlating return code has been given. Contact your system administrator.

System Programmer Response: Follow local procedures for reporting a problem to IBM.

FSUC5058 Arguments should be jobs or process id's

Explanation: Arguments to the specified command need to be either jobs or process id's. These can be found using the **jobs -l** builtin command.

System Action: Command ends.

User Response: Respecify command with arguments that are found in the **jobs -l** command.

FSUC5059 No current job

Explanation: Specified command cannot process because there is no current job.

System Action: Command ends.

FSUC5060 No previous job

Explanation: Specified command cannot process because there is no previous job.

System Action: Command ends.

FSUC5061 No job matches pattern

Explanation: There is no job that matches string in the '%?string' reference.

System Action: Command ends.

User Response: You can get a list of all current jobs with the jobs command. Use a job from within that list.

FSUC5062 Fork nesting > *number*; maybe '...' loop

Explanation: There is a maximum nesting limit of 16 processes. This is done to avoid forking loops.

System Action: Command ends.

User Response: Try to minimize the use of subshells and nested calls to builtin functions.

FSUC5063 No job control in subshells

Explanation: **Job** commands can only be issued from the parent shell.

System Action: Command ends.

User Response: Return to parent shell and reissue command.

FSUC5065 *string* There are suspended jobs

Explanation: There are suspended jobs in the shell that prevent you from exiting.

System Action: Command ends, shell still remains active.

User Response: To find out what jobs are suspended, issue the **jobs** command and either resume or kill these jobs.

FSUC5067 No other directory

Explanation: The **pushd** command with no arguments will exchange the top two elements in the stack. In this case, it cannot process because there is only one directory entry in the stack.

System Action: Command ends.

User Response: Cannot issue command until there is more than one entry in the stack.

FSUC5068 Directory stack empty

Explanation: The directory stack is empty, so the **popd** command can neither print values, nor remove directories from it.

System Action: Command ends.

User Response: Cannot issue command until there are entries in the stack.

FSUC5069 Bad directory

Explanation: The directory specified on the **popd** command is not valid.

System Action: Command ends.

User Response: Respecify with a valid entry from the stack. This can be found using the **dirs** builtin command.

FSUC5071 No operand for -h flag

Explanation: When using the **source -h** command, no operand was given.

System Action: Command ends.

User Response: Reissue with an argument after -h.

FSUC5072 Not a login shell

Explanation: The **login** and **logout** commands both terminate the login shell. These commands cannot process if they are issued from a non-login shell.

System Action: Command ends, shell still remains active.

User Response: To exit, issue the **exit** command.

FSUC5073 Division by 0

Explanation: Divide by 0 is not allowed.

System Action: Command ends.

User Response: Respecify equation so that a divide by 0 does not occur.

FSUC5074 Mod by 0

Explanation: In the expression a%b, b was evaluated to be 0 which attempts a divide by 0.

System Action: Command ends.

User Response: Respecify statement so that b does not equate to 0

FSUC5075 Bad scaling; did you mean *string*?

Explanation: Scale factors for all resources besides cputime default to k or kilobytes. A scale factor of m or megabytes may also be used. For cputime, the default scaling is in seconds, but m for minutes, h for hours or a time form of mm:ss (where m=minutes and s=seconds) may also be used.

System Action: Command ends.

User Response: Respecify the **limit** command with syntax in the proper format.

FSUC5076 Can't suspend a login shell (yet)

Explanation: The **suspend** command cannot be issued when operating from a login shell.

System Action: Command ends, shell still remains active.

User Response: Try using the **logout** command instead.

FSUC5077 Unknown user: *user*

Explanation: The user specified in *user* does not exist.

System Action: Command ends.

User Response: Check that the user exists, check spelling.

FSUC5078 No $home variable set

Explanation: Cannot **cd** to the home directory as the $home variable is not set.

System Action: Command ends.

User Response: Set the $home variable, and the reissue command.

FSUC5080 $, ! or < not allowed with $# or $?

Explanation: An illegal $, ! or < was found in the name portion of $# name or $?name.

System Action: Command ends.

User Response: Reissue this shell variable without the illegal characters.

FSUC5081 Newline in variable name

Explanation: An illegal newline character was found in the variable name.

System Action: Command ends.

User Response: Respecify the variable name to exclude any newlines. Respecify the command.

FSUC5082 * not allowed with $# or $?

Explanation: A wildcard character was found in name portion of either $#name or $?name

System Action: Command ends.

User Response: Respecify the shell variable reference without a * in name.

FSUC5083 $?<digit> or $#<digit> not allowed

Explanation: $? or $# cannot be followed by a digit.

System Action: Command ends.

User Response: Respecify the shell variable reference with a variable name as an argument.

FSUC5084 Illegal variable name

Explanation: Variable name must consist only of alphanumeric characters.

System Action: Command ends.

User Response: Take any non-alphnumeric characters out of the variable name.

FSUC5085 Newline in variable index

Explanation: A newline character is not allowed in the index of an array.

System Action: Command ends.

User Response: Respecify array[index] without any newlines in index.

FSUC5086 Expansion buffer overflow

Explanation: While attempting to resolve a variable expansion (such as $expression), the 1020 character buffer limit was exceeded .

System Action: Command ends.

User Response: Try and minimize complex expressions.

FSUC5087 Variable syntax

Explanation: Variable modifiers cannot have a :g or :a at the end of the word selector.

System Action: Command ends.

User Response: Correct the syntax of modifiers, and the reissue command.

FSUC5088 Bad ! form

Explanation: No closing } was found on the ! history substitution character.

System Action: Command ends.

User Response: Correct the syntax of the statement and reissue.

FSUC5089 No previous substitute

Explanation: There is no previous s substitution for the '' modifier to repeat.

System Action: Command ends.

User Response: Cannot use this modifier until you issue a valid s substitution. Use another form and/or combination of modifiers to process desired history substitution.

FSUC5090 Bad substitute

Explanation: The :s/x/y/ modifier format is not of proper syntax.

System Action: Command ends.

User Response: Correct the syntax, and reissue the statement.

FSUC5091 No previous left hand side

Explanation: There is no previous left hand side for the :s/x/y/ modifier format.

System Action: Command ends.

User Response: Correct the syntax, and reissue the statement.

FSUC5092 Right hand side too long

Explanation: The right hand side of the :s/x/y/ modifier format is too long.

System Action: Command ends.

User Response: Try to shorten the substitution, try and use another form of history substitution, or manually type in command line.

FSUC5093 Bad ! modifier: *modifier*

Explanation: Valid modifiers are: p s & r e h t q x u l g and a.

System Action: Command ends.

User Response: Respecify command with valid modifiers.

FSUC5094 Modifier failed

Explanation: Specified modifier could not complete properly.

System Action: Command ends.

User Response: Check syntax and logic of the statement.

FSUC5095 Substitution buffer overflow

System Action: Command ends.

FSUC5096 Bad ! arg selector

Explanation: The % modifier must be used in conjunction with the !?*string*? reference (for example, !?*string*?:%) where % will match the entire word matching *string*.

System Action: Command ends.

User Response: Correct the syntax and reissue the statement.

FSUC5097 No prev search

Explanation: !?? will repeat the last **search** command. In this case, there is no previous **search** command, therefore, this form of history substitution cannot process.

System Action: Command ends.

User Response: Use another form of history substitution.

FSUC5098 *string***: Event not found**

Explanation: !?*string*? will be replaced with the most recent history line containing *string* in line. No match was found, hence, no history substitution can occur.

System Action: Command ends.

User Response: Use another form of history substitution, or explicitly type in the command.

FSUC5099 Too many)'s

Explanation: There are more closing parenthesis than opening parenthesis.

System Action: Command ends.

User Response: Correct the syntax and reissue the statement.

FSUC5100 Too many ('s

Explanation: There are more opening parenthesis than closing parenthesis.

System Action: Command ends.

User Response: Correct the syntax and reissue the statement.

FSUC5101 Badly placed (

Explanation: The syntax of your statement is not correct due to a misplaced (.

System Action: Command ends.

User Response: Correct the syntax and reissue the statement.

FSUC5102 Missing name for redirect

Explanation: The < or > redirection symbols were used without the appropriate source or target arguments.

System Action: Command ends.

User Response: Reissue the statement with valid arguments on redirection.

FSUC5103 Ambiguous output redirect

Explanation: Output redirection cannot process because the filename and/or pipe is ambiguous.

System Action: Command ends.

User Response: Correct the syntax, and reissue the statement.

FSUC5104 Can't << within ()'s

Explanation: The << redirection symbol cannot be used within a set of parenthesis.

System Action: Command ends.

User Response: Reissue the statement without this symbol inside the ()'s. You may want to try putting the << shell input lines inside a variable, or within a file.

FSUC5105 Ambiguous input redirect

Explanation: Input redirection cannot process because the filename and/or pipe is ambiguous.

System Action: Command ends.

User Response: Correct the syntax, and reissue the statement.

FSUC5106 Badly placed ()'s

Explanation: The syntax of your statement is not correct due to a misplaced parenthesis.

System Programmer Response: Command ends.

User Response: Correct the syntax and reissue the statement.

FSUC5107 Alias loop

Explanation: You have exceeded the maximum value of 50 nested alias expansions.

System Action: Command ends.

User Response: If possible, do not nest this alias.

FSUC5108 No $watch variable set

Explanation: The **log'/'watchlog** command cannot process because the $watch variable was not set.

System Action: Command ends.

User Response: You must set the $watch variable in order to use this command.

FSUC5109 No scheduled events

Explanation: The -n option on the **sched** command cannot process because there are no scheduled events to remove.

System Action: Command ends.

User Response: There are no scheduled events to remove, therefore you don't need to take further action.

FSUC5111 Not that many scheduled events

Explanation: The -n option on the **sched** command cannot process because there are not n number of scheduled events.

System Action: Command ends.

User Response: To see what the correct number of the event is, use the **sched** command with no arguments. Reissue **sched -n** with the correct n value.

FSUC5112 No command to run

Explanation: A corresponding command for the **sched** command was not given.

System Action: Command ends.

User Response: Reissue the command with the correct syntax.

FSUC5113 Invalid time for event

Explanation: The time for the **sched** command is not valid.

System Action: Command ends.

User Response: Correct time syntax and reissue statement.

FSUC5114 Relative time inconsistent with am/pm

Explanation: Relative time cannot have an AM/PM extension. Relative time is number of hours and minutes away from the current time.

System Action: Command ends.

User Response: Reissue the statement without AM/PM extension.

FSUC5117 Unknown capability *capability*

Explanation: The terminal capability passed into the **settc** command is unknown.

System Action: Command ends.

User Response: Reissue statement with a correct terminal capability.

FSUC5118 Unknown termcap parameter *parameter*

Explanation: Valid termcap parameters are: d,2,3,.,+,%,>,i,r,n,B,D

System Action: Command ends.

User Response: Reissue statement with a valid termcap parameter.

FSUC5119 Too many arguments for *command* **(***arguments-required***)**

Explanation: More arguments were given for the specified command than it's syntax allows.

System Action: Command ends.

User Response: Correct syntax and reissue statement.

FSUC5120 *command* **requires** *number* **arguments**

Explanation: The command specified is not in proper syntax.

System Action: Command ends.

User Response: Correct syntax and reissue command.

FSUC5122 *file*: *return-code*. **Binary file not executable**

Explanation: File failed execution with the specified return code. Even though the file has the proper permissions, it is not an executable file.

System Action: Command ends.

User Response: See the return code description for how to proceed. Check the spelling of the command entered.

FSUC5123 !# History loop

Explanation: The !# event specification for history substitution has reached its maximum of 10 levels of recursion.

System Action: Command ends.

User Response: Either use another form of history substitution or explicitly type in command.

FSUC5124 Malformed file inquiry

Explanation: The syntax of the **filetest** command is incorrect.

System Action: Command ends.

User Response: Correct the syntax, making sure to check the file inquiry operator is valid.

FSUC5125 Selector overflow

Explanation: Expansion of the selector expression exceeded the 2056 character limit.

System Action: Command ends.

User Response: Try and simplify the expression.

FSUC5129 Invalid completion: *argument*

Explanation: The specified list argument for the completion rule is not valid.

System Action: Command ends.

User Response: Correct syntax using a valid list specifier.

FSUC5130 Invalid *string*: *string*

Explanation: The specified command or separator field is not of the correct syntax.

System Action: Command ends.

User Response: Correct syntax and reissue completion rule.

FSUC5131 Missing separator *separator* **after** *string string*

Explanation: The syntax of the **completion** statement is not correct due to the specified missing separator.

System Action: Command ends.

User Response: Correct syntax and reissue statement.

FSUC5132 Incomplete *command*: *string*

Explanation: There is no specified range for the positional completion rule.

System Action: Command ends.

User Response: Respecify rule with correct syntax.

FSUC5133 No operand for -m flag

Explanation: The syntax for the -m option on the **source** command is incorrect.

System Action: Command ends.

User Response: Reissue statement with correct syntax.

FSUC5135 $*variable* **is read-only**

Explanation: The specified variable is read only. Any operations that may need to write, append or delete this variable cannot be processed.

System Action: Command ends.

User Response: Do not set this variable as read only, or use another variable.

FSUC5136 No such job

Explanation: The job specified on the command does not exist. You can get a list of jobs and their corresponding process ID's by issuing the **jobs -l** command.

System Action: Command ends.

User Response: Reissue command with a valid job.

FSUC5137 Unknown colorls variable *variable*

Explanation: The LS_COLORS shell variable could not be processed because the specified variable is not valid.

System Action: Command ends.

User Response: Correct syntax and reissue statement.

FSUC5138 The autolock feature is not implemented

Explanation: The command **set autologout=(x y)** was issued in which the y variable was intended to specify the number of minutes the shell can sit idle before it automatically locks.

System Action: The **autologout** command is still implemented, however a**utologout** takes on the value of the y variable, rather than the x.

User Response: If this is not the value you want to take effect for **autologout**, respecify the statement with only one parameter.

FSUC5140 *pid/job-number*: *string*

Explanation: The **kill()** run-time function failed with the specified pid/job number and returned the printed system message. Either the specified signal isn't supported, the caller does not have permission to send to the process specified, or there are no processes corresponding to the specified pid.

System Action: The **kill** command terminates without sending the signal to the process/job.

User Response: Double-check the value of the pid or job number you used when issuing the **kill** command.

FSUC5141 The afsuser special shell variable is not implemented

Explanation: Since the autolock feature is not implemented, setting this variable offers no benefit.

System Action: Processing continues.

User Response: None.

FSUC5142 The autocorrect special shell variable is not implemented

Explanation: Setting this variable will not automatically invoke the **spell-word editor** command before each completion attempt.

System Action: Processing continues.

User Response: To spell check a word, you can manually invoke the **spell-word editor** command. To find out what this command is mapped to, issue the **bindkey** command

Part 5. Appendixes

Appendix A. 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
North Castle Drive
Armonk, NY 10504-1785
USA

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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of OS/390 UNIX System Services (OS/390 UNIX).

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

ACF/VTAM
BookManager
C/370
IBM
IBMLink
Library Reader
MVS/ESA
OpenEdition
OS/390
RACF
SP
System/370
TalkLink

UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

Other company, product, and service names may be trademarks or service marks of others.

ANSI	American National Standards Institute
DFS	Transarc Corporation
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
Network File System	Sun Microsystems, Inc.
NFS	Sun Microsystems, Inc.
Notes	Lotus Development Corporation
POSIX	Institute of Electrical and Electronics Engineers

The information contained in the glossary section and tagged by the word [POSIX] is copyrighted information of the Institute of Electrical and Electronics Engineers, Inc., extracted from IEEE Std 1003.1-1990, IEEE P1003.0, and IEEE P1003.2. This information was written within the context of these documents in their entirety. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

Index

Special Characters

Numerics

A

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Communicating Your Comments to IBM

OS/390
UNIX System Services
tcsh (C Shell) Kit Support Guide

Publication No. b

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: http://www.ibm.com/s390/os390/

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

OS/390
UNIX System Services
tcsh (C Shell) Kit Support Guide

Publication No. b

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

[ ]	As an introduction	[ ]	As a text (student)
[ ]	As a reference manual	[ ]	As a text (instructor)
[ ]	For another purpose (explain)		

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Name	Address
Company or Organization	
Phone No.	

Reader's Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Program Number: 5647-A01

Printed in U.S.A.