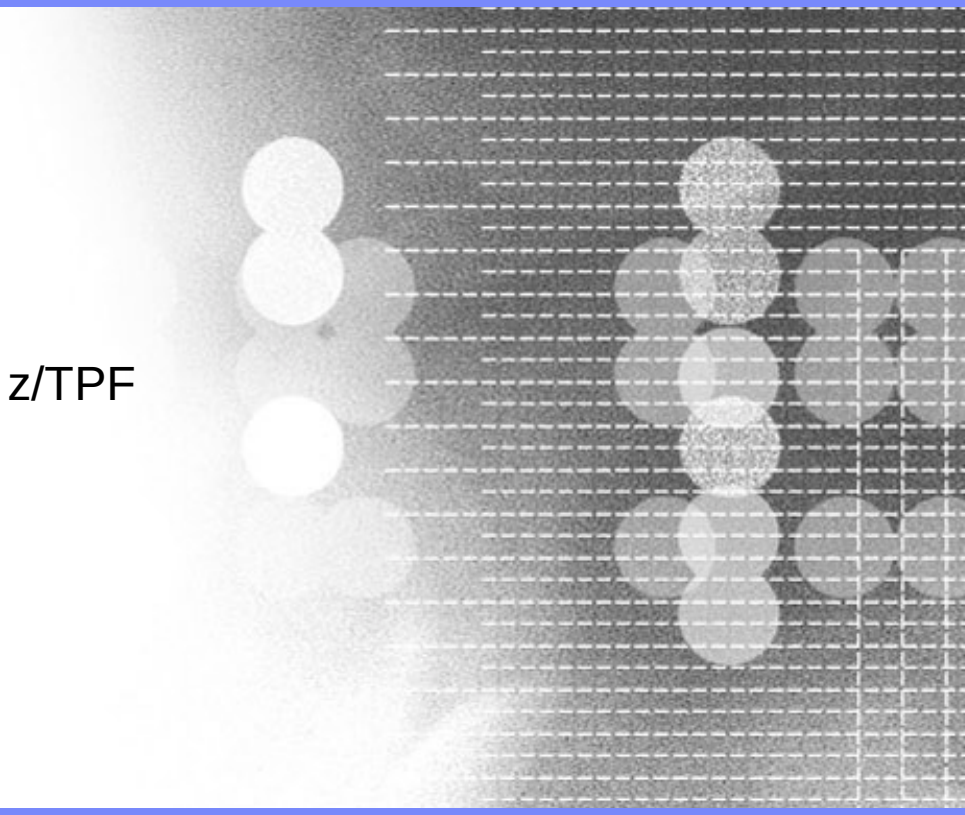
TPF Users Group
Grapevine, Texas

IBM Software Group



Coding Today for z/TPF Tomorrow

C/C++ Coding Changes for Ease of Migration to z/TPF

Languages Subcommittee

Chris Filachek

October 2004

AIM Core and Enterprise Solutions

IBM z/Transaction Processing Facility Enterprise Edition 1.1.0

Topics

- Data Type Changes
- Changes in Compiler Behavior
- Proper Programming Practices

Integer Data Types

- Unchanged
  - short (2 byte integer)
  - int (4 byte integer)
  - long long (8 byte integer)
- Changed
  - long changes from 4 bytes to 8 bytes
- Recommendations
  - Leave all short, int, and long long alone
  - Change all "long" to "int"
    - Ensures all 4 byte fields in TPF 4.1 are also 4 bytes in z/TPF
    - Changing "long" to "int"
      - Use tool to change all references at once
      - Change references manually
      - Change one package at a time, by hand or through tools
    - Exception: On z/TPF, casting pointers using (int) will truncate the address
      - Use (long) to cast pointers to integer data

Pointer Data Type

- * (pointer) changes from 4 bytes to 8 bytes
- 8 byte pointers
  - 8 byte pointers are preferred
  - Leave pointers as 8 bytes where possible
    - Function pointers must be 8 bytes
- 4 byte pointers
  - Only use 4 byte pointers where required
    - Structures mapped by assembler DSECTs or written to file
  - Pointers can be forced to 4 bytes
    - PTR32ATT macro
    - __ptr32_t, __uiptr32_t, or __chptr32_t data types
  - Examples:
    - `__ptr32_t ce1cr0;  /* 4 byte core block ptr */`
    - `struct mystruct * PTR32ATT ptr; /*4 byte struct ptr */`
  - See APAR PJ29575

Storage Areas and Memory Addresses

- Storage areas guaranteed to be below 2GB (can use 4 byte or 8 byte pointers)
  - TPF core blocks (CE1CRx)
  - malloc()
  - 31-bit system heap
  - Application stack
    - alloca() and local variables
- Storage areas above 2GB (must use 8 byte pointers)
  - malloc64()
  - 64-bit system heap
  - static data and exported variables
    - Includes string literals and pre-initialized data
    - C/C++ programs can be forced below the 2 GB bar if necessary

Enumerated Data Types

- TPF 4.1: Enumerations are 1, 2, or 4 bytes
  - Size depends on values of the enumerations
- z/TPF: Enumerations are always 4 bytes
- For structures mapped by assembler DSECTs or written to file, change enumerations as follows
  - Change 1 byte enumerations to unsigned char
  - Change 2 byte enumerations to unsigned short
  - Leave 4 byte enumerations alone
- Example

```
enum options {OPTA = 1, OPTB = 2, OPTC = 3};
enum options optbyte;
```

Changes to

```
#define OPTA  1
#define OPTB  2
#define OPTC  3
unsigned char optbyte;
```

Floating Point Data Types

- float and double
  - Size is unchanged for float (4 bytes) and double (8 bytes)
  - Format is changed
    - TPF 4.1 uses Hexadecimal Floating Point (HFP)
    - z/TPF uses Binary Floating Point (BFP)
      - Conforms to IEEE 754 Standard
  - For z/TPF, all processing of floating point values is in BFP (calculations, printing, etc.)
  - If writing floating point to file, HFP/BFP conversion functions are available
    - See APARs PJ29849 & PJ29980*
- long double
  - TPF 4.1: long double is extended floating point (16 bytes)
  - z/TPF: Extended floating point (16 bytes) is not supported by GCC
    - long double interpreted same as double (8 bytes)
  - Use of long double should be avoided or removed

Fixed Point Decimal Data Type

- decimal(n,p)
  - An extension of the IBM compiler available on TPF 4.1
  - Not supported by GCC on z/TPF
- Recommendations
  - Use decNumber library in place of decimal data type
    - C code: decimal(,) is replaced with decNumber data type
    - C++ code: decimal(,) is replaced with decNumber or pDecimal class
  - See APAR PJ29576

Derived Data Types

- size_t, ssize_t, and time_t
  - Change from 4 byte integers to 8 byte integers
  - Only affects structures mapped by assembler DSECTs or written to file
    - Use new 4 byte data types
      - size_t32
      - ssize_t32
      - time_t32
    - Upper 4 bytes are truncated for the *_t32 data types
    - See APAR PJ29630
- clock_t
  - Size is 8 bytes on both systems
  - Format changes from double (floating point) to long (integer)
  - If stored on file or passed to assembler, consult with TPF CSR

Wide Characters

- Wide characters (wchar_t)
  - Changes from 2 byte EBCDIC to 4 byte Unicode (UCS-4)

```
wchar_t *wstr = L"ABC";          /* wide char string */
00C1 00C2 00C3                   /* 2 byte EBCDIC    */
0000000041 0000000042 0000000043 /* 4 byte Unicode   */
```

  - Multi-byte characters are unchanged
    - 1 and 2 byte multi-byte EBCDIC characters
    - Uses shift-in (0x0f) and shift-out (0x0e) characters to transition between 1 and 2 byte characters
  - Recommendations
    - Write data to file using multi-byte characters only
    - Only use wide characters for in-memory processing only
    - Investigate wide character literals
      - Wide characters coded using hex values breaks single source

Locales

- Categories supported on both TPF 4.1 and z/TPF
  - LC_ALL (all categories)
  - LC_COLLATE
  - LC_CTYPE
  - LC_MESSAGES
  - LC_MONETARY
  - LC_NUMERIC
  - LC_TIME
- Obsolete Categories from TPF 4.1
  - LC_TOD
    - Replaced by TZ environment variable support
    - See APAR PJ29957
  - LC_SYNTAX
    - Remove all usage of this category

Locale Dependent Functions

- Standard functions are unchanged from TPF 4.1 to z/TPF
  - setlocale()
  - strcoll() & strxfrm()
  - strfmon() & strftime()
  - isctype() family of functions (isalpha, isdigit, etc.)
  - toupper() & tolower()
- Nonstandard collation functions from TPF 4.1 are not supported on z/TPF (see collate.h on TPF 4.1)
  - ismccollel() strtocoll() colltostr() collequiv()
  - collrange() collorder() cclass() maxcoll()
  - getmccoll() getwmccoll()

Packing Structures

- IBM compiler and GCC both pack structures using #pragma pack()
- "packed" and "reset" parameters
  - Recognized by IBM compiler only

```
#pragma pack(packed) /* Pack structure */
   struct mystruct { ... }
#pragma pack(reset)  /* Stop packing   */
```

- "1" and empty parameters
  - Recognized by IBM compiler and GCC

```
#pragma pack(1)      /* Pack structure */
   struct mystruct { ... }
#pragma pack()       /* Stop packing   */
```

- Recommendations
  - Change "packed" to "1"
  - Change "reset" to empty parameter
  - Use #pragma pack() instead of __Packed keyword
  - Pack all C/C++ structures that are also mapped by Assembler DSECTs

Initializing Arrays of Structures

- Valid initializer syntax on TPF 4.1

```
struct mystruct array[] = {
      item1-a, item1-b, item1-c,      /* struct 1 */
      item2-a, item2-b, item2-c,      /* struct 2 */
      item3-a, item3-b, item3-c,      /* struct 3 */
      ......
    }
```

- For GCC, lack of brackets causes compiler errors. Add brackets around each structure

```
struct mystruct array[] = {
      { item1-a, item1-b, item1-c }, /* struct 1 */
      { item2-a, item2-b, item2-c }, /* struct 2 */
      { item3-a, item3-b, item3-c }, /* struct 3 */
      ......
    }
```

Order of Operations and Parenthesis

- GCC issues warnings when nested or multiple operators are coded without parenthesis
- Example: valid if statement

```
if a || b && c
```

- Which did the programmer intend?

```
if (a || b) && c
   or
if  a || (b && c)   /* equivalent to using no */
                    /* parenthesis            */
```

- Use parenthesis to explicitly group operations

Ambiguous "else" Statement

- GCC issues warnings when nested if-else statements are coded without brackets
- Example

```
if (a)
  if (b)
     QZZ1();
else
      QZZ2();
```

- Which did the programmer intend?
  - Indentation implies that else belongs to "if (a)"
  - Compiler associates the else with the innermost if statement - "if (b)"
- Use brackets around nested if statements

Proper Coding Practices

- Do not use hardcoded values for structure or data type sizes

```
malloc(100 * 4);
```

- Use sizeof() to determine the size of structures and data types

```
malloc(100 * sizeof (void *) );

malloc(100 * sizeof (int) );
```

Appropriate Storage Areas

- Some code may use the ECB workarea or core blocks to map a SIPCC request blocks, a TPF_regs structure, or other temporary storage areas
  - Usually found in converted TARGET(TPF) or assembler programming styles brought into C code
  - Sizes of these structures may have changed, exceeding these preallocated areas
- Use local variables (stack), alloca(), or malloc() for storage

Bad:
```
struct TPF_regs *treg;
treg = (struct TPF_regs *)&(ecbptr()->ebw000);
```

Good:
```
struct TPF_regs treg;   /* Use stack   */
```
or
```
struct TPF_regs *treg;  /* Use alloca() */
treg = (struct TPF_regs *)
          alloca(sizeof (struct TPF_regs));
```

Questions

Legal

- IBM is a registered trademark of International Business Machines, Inc.
- Other company, product, or service names may be trademarks or service marks of others.