Redbooks Paper

Vamsi Kulukuru

z/OS 64-bit C/C++ and Java Programming Environment

Introduction

One of the key enhancements available in z/OS V1R6 is 64-bit application development support for C/C++. With 64-bit support, applications can take advantage of the expanded 1 million TB address space. In this IBM Redpaper, we introduce C/C++ programming in a 64-bit environment, examine changes to the z/OS Language Environment, discuss migrating to 64 bit, and consider Java 64-bit programming. The topics include:

- The z/OS 64-bit programming model
- Compiling and linking 64-bit programs
- The 64-bit memory model
- Changes to the Language Environment
- LP64 and ILP32 coexistence
- General C/C++ migration tips
- Porting Java applications

The z/OS 64-bit programming model

The z/OS 64-bit implementation uses the Long-Pointer 64 (LP64) programming model proposed by the Open Group consortium. In this model, long integers and pointers are 64 bits (or eight bytes) in length. This contrasts to the 32-bit Integer Long Pointer (ILP32) implementation where integers, long integers, and pointers are four bytes in length.

Note: Although pointers in the ILP32 data model are 32 bits long, only 31 bits are used to access memory locations. The high-order bit is used to indicate the addressing mode in effect (AMODE31 or AMODE 24).

Differences between the ILP32 and LP64 implementations can be summarized as:

- **Address space size**

  With 64-bit pointers, the LP64 address space expands to 16 million TB from the 2 GB address space available for ILP32.

  Note: The actual address space available to an application is limited to the amount of physical memory installed on the system.

- **Execution mode**

  LP64 programs execute in 64-bit addressing mode (AMODE64); ILP32 programs execute in 31-bit addressing mode (AMODE31).

  Note: In both AMODE31 and AMODE64, code resides and executes in memory addresses below 2 GB.

- **Linkage**

  Only high-performance linkage (XPLINK) is supported with LP64. In ILP32, both XPLINK and non-XPLINK linkages are supported.

- **Dynamic Link Libraries (DLLs)**

  64-bit applications can only run with 64-bit DLLs, and 32-bit applications can only run with 32-bit DLLs.

- **C data type sizes and alignments**

  In addition to longer pointer sizes, the LP64 model uses larger sizes for some C data types. The alignment for these data types also changes from the ILP32 model. We discuss these changes in “Data type sizes and alignments” on page 3.

When to port programs to 64 bit

The LP64 model can offer some significant benefits to 32-bit applications, such as:

- **Expanded memory addressing**

  The larger LP64 virtual address space allows applications to process large amounts of data in memory without the use of temporary files on secondary storage or dataspaces. This can reduce I/O and application complexity.

- **Large integer arithmetic**

  The LP64 model can perform signed integer operations where operands and the result are in the range $-2^{64} < x < 2^{64} - 1$.

- **Backward compatibility**

  In both the LP64 and ILP32 data models, the **int** data type is four bytes in length. This can minimize the effort to port applications that require larger virtual memory but require only 32-bit arithmetic.

However, 32-bit applications should not be migrated to 64 bit as a matter of course. Consider migration if the application:

- Requires 64-bit pointers for larger virtual memory addressing.
- Accesses 64-bit DLLs or utilities. This may be the case if vendor products were ported to 64 bit.

Note: By default, the z/OS compiler generates 32-bit code. In order to generate 64-bit code, you must specify the LP64 option as described in “Compiling and linking 64-bit programs” on page 5.

There can be costs associated with 64-bit migration:

- **Source code analysis and modification**

  The migration effort can be minimal in cases where standard C programming practices were followed. However, nonstandard C constructs can greatly increase the cost to migrate an application.

- **Performance considerations**

  There is no inherent performance advantage to executing in 64-bit mode. A 64-bit application can impact performance due to:

  – Increase in module size
  Longer 64-bit instructions result in a larger module size.

  – Increase in the Writable Static Area (WSA) and stack sizes
  Pointers and **long** data types require larger WSA and stack sizes.

  Note: The WSA is a memory area which may be modified during program execution. It typically contains global variables and control blocks to access DLL functions and variables. In LP64, the WSA is allocated above the 2 GB bar.

  – Reentrancy
  The RENT and DLL options are required for 64-bit compilation. This can reduce performance for applications previously compiled with the NORENT and NODLL options.

Data type sizes and alignments

Table 1 on page 4 compares data type sizes and alignments between ILP32 and LP64.

Table 1 Comparison between 32-bit and 64-bit data types

		32 bit		64 bit	
Category	**Data type**	**Size**	**Alignment**	**Size**	**Alignment**
Character	char	1 byte	1 byte	1 byte	1 byte
	wchar_t [a]	**2 bytes**	**2 byte**	**4 bytes**	**4 byte**
Integer	short	2 bytes	2 byte	2 bytes	2 byte
	int	4 bytes	4 byte	4 bytes	4 byte
	long	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	long long [b]	8 bytes	8 byte	8 bytes	8 byte
Floating point	float	4 bytes	4 byte	4 bytes	4 byte
	double	8 bytes	8 byte	8 bytes	8 byte
	long double	16 bytes	8 byte	16 bytes	8 byte
Pointer	**pointer types**	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	ptrdiff_t	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
Other	**size_t**	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	ssize_t	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	time_t	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	off_t	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**
	rlim_t	**4 bytes**	**4 byte**	**8 bytes**	**8 byte**

a. Typically 4 byte length for both 32 bit and 64 bit on other UNIX platforms
b. Code must be compiled with “langlvl(longlong)” or “langlvl(extended)” option

Note: Data type size and alignment changes from 32 bit to 64 bit can increase the size of structures.

Pointer arithmetic

When porting to LP64, closely examine arithmetic operations involving pointers. In ILP32, pointers can be safely assigned to both **int** and **long** data types using explicit casts (each being four bytes long). This is *not* the case in a 64-bit environment. With LP64, pointers are eight bytes long while the **int** data type remains four bytes in length.

When a pointer is incremented, the size of the data type to which it points is added to the pointer’s value. This can lead to unexpected results when porting to LP64. We consider the following code example:

```
int  *p1;
long *p2
p1++;
p2++
```

For ILP32, the increment operator adds four bytes to both p1 and p2. In LP64, p1 is incremented by four bytes (the size of the **int** data type) while p2 is incremented by eight bytes (the size of the **long** data type).

Compiling and linking 64-bit programs

As previously noted, the z/OS compiler generates ILP32 (AMODE31) code by default. To generate 64-bit code (AMODE64), use the LP64 compiler option. Supply the **-Wc,lp64** option to the **c89** command to compile 64-bit code under z/OS UNIX System Services.

Tip: To pass the LP64 option to the **c89** command when linking pre-compiled object code, use the **-Wl,lp64** option.

By default, the LP64 enables the following options:

- **XPLINK**
  LP64 requires XPLINK.
- **GOFF**
  This option produces Generalized Object File Format output. The maximum GOFF file output is 1 GB.
- **ARCH(5)**
  Level 5 is the minimum required for 64-bit code.

If an incompatible option is specified (NOXPLINK, NOGOFF, or ARCH level less than 5), the compiler ignores the option, issues a warning message, and sets the ARCH(5) option.

Tip: When the LP64 option is specified, the _LP64 macro is defined to the compiler. This allows conditional compilation based on the desired object code output (IL32 or LP64).

The _DEBUG_FORMAT environment variable controls the debug format (DWARF or ISD) that is utilized when the -g option is used with LP64.

LP64 restrictions

Several restrictions exist for usage of the LP64 option:

- **Coexistence of the ILP32 and LP64 options**

  The ILP32 and LP64 options are mutually exclusive. If both are specified, the last specified option is accepted.
- **Prelinker**

  The prelinker cannot be used with LP64 code.
- **Use of the TARGET option**

  The TARGET(IMS) option is not supported (a warning message is generated). The LP64 option is ignored when a target prior to z/OS V1R6 is specified.
- **Use of the FLOAT option**

  By default, LP64 generates FLOAT(IEEE) code. However FLOAT(HEX) may be specified.
- **Mixed addressing modes**

  LP64 and ILP32 object code cannot be intermixed. If the binder encounters mixed addressing modes, an error message is generated.
- **Correct function prototype files**

  If no function prototype is provided, a function returns a four byte **int** value by default. In LP64, we highly recommend that you prototype all functions. As an example, in order to

use the C malloc() function the stdlib.h file *must* be included. This is because the returned pointer is 64 bits in length, not 32 bits:

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

The WARN64 compiler option

The new WARN64 compiler option can be useful when porting to 64-bit. This option is designed to flag code fragments that may not port as expected from ILP32 to LP64. When used with the FLAG(I) option, WARN(64) flags conditions such as:

- Constants assigned to **unsigned long** data types. In LP64, these values can be assigned to a **long** data type with overflow.
- Constants larger than UINT_MAX and smaller than ULONGLONGMAX. Although these may overflow in ILP32, values are acceptable in LP64.
- Loss of digits when assigning a **long** to **int** data type.
- Possible change in result when assigning an **int** to **long**.
- Truncation of high-order bytes when assigning a pointer to **int**.
- Incorrect results when assigning **int** to a pointer.
- Warning when constant is assigned to **long**.

By default, WARN64 is disabled (NOWARN64).

The 64-bit memory model

Next, we consider stack and heap memory allocations in 64-bit mode.

Stack memory allocation

Stack storage is obtained by the Language Environment at initialization. Stack memory is allocated above the 2 GB bar and the STACK64 runtime option determines the size.

To illustrate the data type size, alignment, and stack location, we consider the example datasize.c code shown in Figure 1 on page 7.

```
#include<stdio.h>
int main(int argc,char *argv[])
{
        int i;                          /* integer data type                                          */
        long l;                         /* long int data type                                         */
#ifdef _LONG_LONG                       /* _LONG_LONG is defined when compiled with langlvl(longlong) */
        long long ll;
#endif
        float f;                        /* floating point data type                                   */
        size_t st;                      /* data type returned by sizeof operator                      */
        int *ip = &i;                   /* pointer data type                                          */
        long *lp = &l;                  /* pointer to a long data type                                */

        struct T {
                char c;
                short s;
                #if !defined(_LP64)     /* _LP64 is defined when compiled with lp64 option            */
                        char pad1[4];
                #endif
                int *p;
                #if !defined(_LP64)
                        char pad2[4];
                #endif
        } t;
        printf(" size of int = %d memory located at %#lx\n",sizeof(i),&i);
        printf(" size of long = %d memory located at %#lx\n",sizeof(l),&l);
#ifdef _LONG_LONG
        printf(" size of long long = %d memory located at %#lx\n",sizeof(ll),&ll);
#endif
        printf(" size of float = %d memory located at %#lx\n",sizeof(f),&f);
        printf(" size of size_t = %d memory located at %#lx\n",sizeof(st),&st);
        printf(" size of integerpointer = %d memory located at %#lx\n",sizeof(ip),&ip);
        printf(" size of longpointer = %d memory located at%#lx\n",sizeof(lp),&lp);
        printf(" size of struct T = %d memory located at%#lx\n",sizeof(t),&t);

        return 0;
}
```

Figure 1 The datasize.c sample code

In this example, we note use of the _LONG_LONG and _LP64 macros:

- **_LONG_LONG**

  The _LONG_LONG macro is defined when the **langlvl(longlong)** option is supplied during compilation. We use this macro to conditionally compile code that defines and accesses the l**ong long** data type.

- **_LP64**

  The _LP64 macro is defined when the **LP64** option is supplied during compilation. We use this macro to conditionally add padding characters to the **struct T** data type when compiling in ILP32 mode.

We first compile the program in 32-bit using the command:

```
c89 datasize.c
```

Figure 2 on page 8 displays how the output appears when the command is executed.

```
size of int = 4 at memory location 0x 2093 f2b8
size of long = 4 at memory location 0x 2093 f2bc
size of float = 4 at memory location 0x 2093 f2c0
size of size_t = 4 at memory location 0x 2093 f2c4
size of integerpointer = 4 at memory location 0x 2093 f2c8
size of longpointer = 4 at memory location 0x 2093 f2cc
size of struct T = 16 at memory location 0x 2093 f2d0
```

Figure 2 Output of datasize in 32-bit mode

The reported data type sizes match the expected sizes as discussed in “Data type sizes and alignments” on page 3. Each memory location is below the 2 GB bar, as expected with AMODE31. The reported size of the **struct T** data type is 16 bytes, which is consistent with the conditional addition of two-character alignment arrays.

We next compile the program in 64-bit using the command:

```
c89 -W"c,lp64,langlvl(longlong)" -Wl,lp64 datasize.c
```

Figure 3 displays how the output appears when the command is executed.

```
size of int = 4 at memory location 0x1 082f fa00
size of long = 8 at memory location 0x1 082f fa08
size of float = 4 at memory location 0x1 082f fa10
size of size_t = 8 at memory location 0x1 082f fa18
size of integerpointer = 8 at memory location 0x1 082f fa20
size of longpointer = 8 at memory location 0x1 082f fa28
size of struct T = 16 at memory location 0x1 082f fa30
```

Figure 3 Output of datasize in 64-bit mode

In this case, we see that each memory location is above the 2 GB bar, as expected with AMODE64. We also note the LP64 size of the **long**, **size_t**, and pointer data types. When compiled in LP46 mode, the **struct T** data type is 16 bytes.

Heap memory allocation

When compiled in ILP32 mode, heap memory is allocated below the 2 GB bar. In LP64 mode, heap memory is allocated above the 2 GB bar. We use the code shown in Figure 4 to illustrate operation of the malloc function for both ILP32 and LP64 modes.

```
#include<stdlib.h>
int main(int argc,char*argv[])
{
        int * p;
        p = (int*)malloc(sizeof(int));
        printf(" size of integer pointer = %d,address of pointer=%#lx\n",sizeof(p),(p));
        return 0;
}
```

Figure 4 Heap allocation using malloc

When compiled in ILP32 mode using the **c89 heap64.c** command, output appears as:

```
size of integer pointer = 4,address of pointer = 0x 208d ec10
```

In this case, the pointer is four bytes in length and points to memory below 2 GB.

When compiled and linked in LP64 using the **c89 -Wc,lp64 -Wl,lp64 heap64.c** command, output appears as:

```
size of integer pointer = 8,address of pointer = 0x1 0830 1010
```

In this case, the pointer is eight bytes in length and points to memory above 2 GB.

Note: The stdlib.h include is required when calling malloc in 64-bit mode. The malloc prototype defined in stdlib.h returns a pointer that is eight bytes long. If stdlib.h is not included, no function prototype is defined for malloc, and it returns a four-byte integer. In this case, the actual eight-byte pointer value is truncated to four bytes, which results in a runtime error.

Dynamic memory allocation below the 2 GB bar

In LP64 mode, it is possible to allocate memory below the 2 GB bar using the new __malloc31 and __malloc24 functions. The function prototype for __malloc31 is:

```
void *__malloc31(size_t sz);
```

The return value from __malloc31 is a pointer to memory of the requested size that resides *below* the 2 GB bar. If not enough free memory exists to fulfill the request or the requested size is zero, __malloc31 returns NULL. If insufficient memory is available, the errno global variable is set to ENOMEM.

The function prototype for __malloc24 is:

```
void *__malloc24(size_t sz);
```

The return value from __malloc24 is a pointer to memory of the requested size that resides *below* the 16 MB line.

Note: Both __malloc31 and __malloc24 return a pointer. In 64-bit mode, pointers are eight bytes in length. Although the allocated memory resides below the 2 GB bar, the pointer to that memory is eight bytes in length (the high-order bits are set to zero).

The __ptr32 data type

Normally, pointers in LP64 are eight bytes in length. To support pointers to memory locations below 2 GB and to provide compatibility with 31-bit interfaces, the new __ptr32 keyword was defined. The keyword appears as part of a pointer declaration:

```
int *__ptr32 p32;
```

In this example, the p32 variable is defined to be a four byte pointer to an **int** data type that resides below 2 GB. When defined with the __ptr32 qualifier, a 64-bit address is formed by filling the high-order 33 bits with zeros. In Figure 5, we illustrate __ptr32 usage.

```
int *p;
int *__ptr32 q;
int *__ptr32 const r = q;
int *__ptr32 *s;
```

Figure 5 The __ptr32 keyword usage

The example in Figure 5 defines:

- Variable p is a 64-bit pointer to an **int** value.
  This is the normal situation for pointers in 64-bit mode.
- Variable q is a 32-bit pointer to an **int** value.
  The __ptr32 qualifier indicates that q is a four byte pointer.
- Variable r is a 32-bit constant pointer to an **int** value.
  The const qualifier indicates that r always points to memory addressed by variable q.
- Variable s is a 64-bit pointer to a 32-bit pointer to an **int** value.
  In this case, the __ptr32 qualifier is applied to *s, indicating that s is a four byte pointer.

Note: The __ptr32 qualifier can be useful when porting applications from ILP32 to LP64 (particularly when dealing with pointers defined in **struct** data types). When using __ptr32, be certain it *always* applies to a pointer addressing memory below the 2 GB bar.

Be aware that __ptr64 is a reserved word. If encountered during compilation, the compiler issues a warning message.

External linkage with __ptr32

When used with external linkage, the __ptr32 qualifier must be uniformly applied in all compilation units. If a pointer is qualified with the __ptr32 attribute in one compilation unit, the pointer must be defined with __ptr32 in *all* compilation units; otherwise, the pointer behavior is undefined.

Illustrating __malloc31 and __ptr32 usage

To illustrate use of the __malloc31 function, we consider the code shown in Figure 6.

```
#include <stdlib.h>
#include <errno.h>
int main(int argc,char *argv[])
{
      int * __ptr32 p = (int *)__malloc31(sizeof(int) * 4);
      if( p!= NULL)
        {
           printf(" Address of memory allocated for p = %#lx\n",p);
        }
      else
        {
          printf("__malloc31 failed with an error of %d\n",errno);
        }
      return 0;
}
```

Figure 6 Using the __malloc31 function

When compiled in LP64 mode and executed, the code produces the following output:

```
Address of memory allocated for p = 0x 2088 1048
```

In this case, we see that __malloc31 returns a pointer to memory below the 2 GB bar. The source code includes both stdlib.h and errno.h. The __malloc31 function is declared in stdlib.h, and the global errno variable is defined in errno.h.

Note: In the example, the memory allocated by __malloc31 is addressed by variable p, qualified with the __ptr32 attribute (declaring p to be four bytes in length). Be aware that __malloc31 returns an eight byte pointer. In this case, the high-order four bytes of the returned pointer are truncated during assignment to variable p. This is acceptable because the high-order four bytes of the pointer are zero (__malloc31 allocates storage below 2 GB).

The MEMLIMIT specification

Overall application memory allocation, stack and heap, above the 2 GB bar is limited by the MEMLIMIT specification. MEMLIMIT may be set for an application using:

- SMFPRM*xx* MEMLIMT parameter

  This sets the default MEMLIMIT for each address space. System programmers should set this value based on system requirements. By default, this system setting is zero.

- SETSMF command

  Use the SETSMF command to interactively change the system default MEMLIMIT. This affects new address spaces which have not specified a MEMLIMIT value.

- MEMLIMIT keyword

  The JOB and EXEC JCL statements allow the MEMLIMIT to be set on a per-job basis.

- Specify MEMLIMIT for Unix System Services users via the RACF OMVS segment

  The RACF commands ADDUSER and ALTUSER support setting MEMLIMIT using the following specification:

  ```
  MEMLIMIT(memory-limit-size) | NOMEMLIMIT
  ```

  This allows you to assign a specific MEMLIMIT value for users that logon through telnet or rlogin (where there is no LOGON PROC to specify a MEMLIMIT). The MEMLIMIT value from the OMVS segment takes affect after a setuid/exec or spawn with userid (which is how daemons typically create address spaces for users).

- Specify MEMLIMIT on spawn

  The MEMLIMIT can be set on the spawn system call using the inheritance structure. This maintains consistency with places that control region size (the region size can also be specified in the inheritance structure). This mechanism allows a program to create an AMODE64 child process which can acquire more storage above the 2GB bar than the system default would normally permit.

- IEFUSI system exit

  The IEFUSI system exit can increase or decrease the MEMLIMIT value that has been specified or defaulted for an address space.

Important: A 4093 system abend (abnormal end of task) with reason code 548 occurs if a 64-bit C/C++ application is executed when MEMLIMIT is set to zero.

Changes to the Language Environment

IBM Language Environment provides a single run-time environment for applications written in High Level Languages, such as COBOL, C/C++, PL/I, and Fortran.

Note: Support for 64-bit mode is provided only for C/C++ using the Language Environment run-time environment, and for Language Environment conforming assembler, which has no run-time environment.

With the addition of LP64 mode, Language Environment is supported in:

- Base
  This provides 31-bit support, EBCDIC support, Standard Language Environment Linkage, and the Hex and IEEE math libraries.
- XPLink
  This provides 31-bit support, EBCDIC and Enhanced ASCII support, XPLINK linkage, and the IEEE math libraries.
- 64-bit XPLink
  This provides 64-bit support, EBCDIC and Enhanced ASCII support, XPLINK linkage, and the IEEE math libraries.

Language Environment continues to be supported and shipped for both 31-bit and 24-bit addressing. The CELQLIB library was added to provide 64-bit application support. Parts associated with 64-bit support start with CELQ prefix. Language Environment resides in the data sets beginning with SCEE.

The Language Environment form is invoked from an application determined by the Language Environment bootstrap routine. For LP64, the C compiler emits the CELQSTRT csect name (the corresponding ILP32 name is CEESTART). In Table 2, we list Language Environment module names used in ILP32 and their respective LP64 equivalents.

Table 2 Load modules in ILP32 and LP64 modes

ILP32 module name	LP64 module name
CEESTART	CELQSTRT
CEEMAIN	CELQMAIN
CEEFMAIN	CELQFMAN
CEEROOTA/B (non-XPLINK)	CELQBST
CEEROOTD (XPLINK)	
CEEINT/CEEBPIRA	
CEEBETBL	CELQETBL
CEESG003	CELQSG03
CEEBLLST	CELQLLST
EDCINPL	CELQINPL
CEEBTRM	CELQTRM

New run-time options

Several 64-bit run-time options were added:

- **HEAPPOOLS64**

  This option controls heap pools used to improve performance of the 64-bit malloc function.
- **HEAP64**

This option controls allocation of user heap memory.

- **IOHEAP64**

  This controls allocation of I/O heap memory.

- **LIBHEAP64**

  This controls allocation of heap memory used by Language Environment itself.

- **STACK64**

  This controls allocation of stack memory.

- **THREADSTACK64**

  This controls allocation of thread stack memory.

Run-time options may be specified using the #pragma compiler directive:

```
#pragma runopts(opt1, opt2, ...)
```

Existing run-time options supported in AMODE64 include:

- ARGPARSE / NOARGPARSE
- ENVAR
- EXEOPTS / NOEXECOPS
- FILETAG
- HEAPCHK
- INFOMSGFILTER
- NATLANG
- POSIX
- PROFILE
- REDIR/NOREDIR
- RPTOPTS
- RPTSTG
- STORAGE
- TERMTHDACT
- TEST/NOTEST
- TRACE
- TRAP

All other runtime options are not supported.

Changed run-time options

The following runtime options were modified for LP64:

- STORAGE
  Reserve stack size ignored. It is always 1 MB (above the 2 GB bar).
- TERMTHDACT
  The CESE option is ignored.
- TRACE
  The tablesize is ignored. It is always set to 1 MB.

Deprecated features

Several Language Environment features are deprecated in LP64 mode:

- Additional heaps and HIPERSPACE are not supported in LP64 mode. Files of TYPE=MEMORY(HIPERSPACE) are treated as regular memory files.

- Global streams are not supported for LP64.
- The GONUMBER compiler option is not supported in LP64 mode.
- CEEINT, CEEBPIRA, CEEBXITA, IBMBXITA,CEEBTRM, CEEROOTA,CEEROOTB, and CEEBPUBT were eliminated.

LP64 and ILP32 coexistence

LP64 applications cannot directly link to ILP32 libraries. Likewise, ILP32 applications cannot directly link to LP64 libraries. An application may only link to DLLs compiled for its addressing mode. When migrating to LP64, all required DLLs must also be migrated.

Code compiled for LP64 may interact with ILP32 code using:

- **Inter-Process Communication (IPC)**

  Processes running in different addressing modes may interact using IPC techniques. For example, a 64-bit application can fork itself and exec a 31-bit application from the child process. These two processes may interact using an IPC mechanism (named pipes, message queues, and shared memory).

  Note: If using shared memory, ensure that the shared memory area resides below the 2 GB bar. The shmget performed by a 64-bit application must specify the __IPC_BELOWBAR flag. If shmget is performed by a 31-bit application, the acquired shared memory resides below the 2 GB bar. Any pointers defined within the shared memory should be defined using the __ptr32 macro.

- **Use of address mode switching glue code**

  An AMODE64 assembler glue routine is required to call an AMODE31 program or service from an AMODE64 C or C++ program. The glue routine is linked with the C program and called in AMODE64. It may use Language Environment linkage macros or linkage OS. The assembler glue routine operates as follows:

  – Any arguments from the AMODE64 caller must be copied below the 2GB bar (to allow the AMODE31 target to access them).
  – The glue routine switches to AMODE31 using the SAM31 instruction, then invokes the AMODE31 target.
  – On return from the AMODE31 target, the glue routine switches back to AMODE64 using the SAM64 instruction.
  – Any return parameters from the AMODE31 target must be copied to storage above the bar before control is returned to the AMODE64 caller.

  While running in the glue routine or AMODE31 target, no Language Environment routines or library functions may be called. The glue routine will usually require storage below the bar. This can be pre-allocated and passed in as a parameter by the AMODE64 caller, or may be obtained by the glue routine using GETMAIN or STORAGE OBTAIN.

Tip: If the glue routine uses storage below the bar and is called repeatedly, you can avoid the overhead of obtaining storage on each call:

- Use __malloc31 to obtain storage in the AMODE64 caller.
- Pass a pointer to the allocated storage to the glue routine (along with any other arguments).
- The glue routine can used the allocated storage for the AMODE31 target parameter list.

This allows the storage to be managed by C and to be cleaned up at program termination.

In general, it is not possible to share data structures containing long and pointer members between 31-bit and 64-bit processes due to size and alignment changes (as outlined in “Data type sizes and alignments” on page 3).

General C/C++ migration tips

Following are important tips to keep in mind when porting an application from 32 bit to 64 bit:

- **Prototype all functions**.

  By default, un-prototyped functions are assumed to return a four byte **int** value. In the ILP32 model, where **int**, **long**, and pointer sizes are all four bytes, this assumption may be valid. In the LP64 model, the assumption is often incorrect.

- **Include stdlib.h when using the malloc function.**

  When porting to LP64, be sure to include stdlib.h if using the malloc function. Consider the following example:

```
#include<stdio.h>
int main(int argc,char* argv[])
{
        int *p;
        p = (int*) malloc(20);
        printf(" the address of p is %ld\n",p);
        p[0]= 10;
        return 0;
}
```

  The code compiles cleanly in LP64 mode (the malloc function defined in stdio.h is suitable for ILP32 operation). However, execution in LP64 mode results in a segmentation error. To correct the problem, include the stdlib.h file.

- **Ensure sufficient region size.**
  - As noted in “The MEMLIMIT specification” on page 11, a system abend occurs if you execute 64-bit applications with MEMLIMIT set to zero. Set MEMLIMIT to allow memory to be allocated above 2 GB. If no JOB or EXEC MEMLIMIT value is specified:
    - If REGION is zero, MEMLIMIT is assigned no limit.
    - If REGION is non-zero, the MEMLIMIT value from SMFPRM*xx* is used.

    The MEMLIMIT for an address space comes from the following places (in order of precedence):
    - IEFUSI system exit has the last word on any MEMLIMIT specification.

- Explicit specification of MEMLIMIT on JCL, spawn, or the OMVS segment of a user profile in RACF.
- The system default from SMFPRM*xx*.

- **Verify the length of declared variables.**

  Four byte variables are usually **int** or **unsigned int** for LP64; eight byte variables are typically **long** or **unsigned long**. Portability problems typically arise when:

  – **int** and **long** data types are used interchangeably

  – **long** arguments are passed to functions expecting **int** data types

  – Pointers are cast to **int**

  Be aware that assignment of **long** data types to **float** can result in loss of accuracy in 64-bit mode.

- **Check for arithmetic data expansion, truncation, and overflow.**

  Closely examine pointer arithmetic operations. Assignment of a pointer to an **int** causes truncation. Look for sections of code where logic depends on data sizes. **Long** to **int** conversions are common causes of truncation.

- **Use the WARN64 compiler option.**

  Use the FLAG(I) and WARN64 compiler options, which we discussed in “The WARN64 compiler option” on page 6, to identify likely migration problems.

- **Be sure to migrate required DLLs to LP64.**

  As discussed in “LP64 and ILP32 coexistence” on page 14, LP64 applications cannot directly link to ILP32 DLLs. Be sure that any required DLL library functions are migrated to LP64.

Other things to look for when migrating to 64-bit mode include:

- The sizeof operator returns a **size_t** type (an eight-byte unsigned long value).
- The **ptrdiff_t** type is used to hold different values between two pointers. This is also an eight byte unsigned long value in 64-bit mode.
- When compiled in 64-bit mode, masks generally lead to different values than when they are compiled in 31-bit mode.
- Many included files define pointers and structures. These can change the size of the program data section (even if the pointers and structures are not explicitly accessed).
- The size of constants change in LP64. This can lead to problematic behavior of the left and right shift operators at areas such as:
  – constant >= UINT_MAX
  – constant < INT_MIN
  – constant > INT_MAX

Porting Java applications

Java is an interpreted, platform-independent programming language. The Java Virtual Machine (JVM) translates pure Java code to the instruction set of the underlying hardware platform. The Java 2 Standard Edition (J2SE) Application Programming Interface (API) remains the same for both 32-bit and 64-bit implementations.

In keeping with the Java spirit of “write once, run anywhere”, pure Java code runs in both ILP32 and LP64 mode *without* modification. To execute pure Java code in LP64 mode, you need only use a 64-bit enabled JVM; no modification to the Java application is required.

Java Native Interface considerations

The Java Native Interface (JNI) is an API that allows pure Java to execute code written in a native language such as C or C++. Native code *is* platform dependent and typically compiled into one or more libraries. When porting Java applications that contain JNI calls to 64-bit, it is important to ensure that the underlying native code is recompiled and linked in LP64 mode.

In Table 3, we list the native Java data types and describe their size and attributes. Use this when mapping Java native types to C data types in JNI code.

Table 3 Java native types descriptions

Native Java data type	Description
jboolean	unsigned 8 bits
jbyte	signed 8 bits
jchar	unsigned 16 bits
jshort	signed 16 bits
jint	signed 32 bits
jlong	signed 64 bits
jfloat	32 bits
jdouble	64 bits

JNI example

For an illustration of porting considerations for JNI code, examine the Java code in Figure 7.

```
class Sum {

        public native long add(int i,long j);

        static {
            System.loadLibrary("SumNative");
            }

        public static void main(String[] args) {
            int a = 224;
            long b = 31289121;
            System.out.println("Class running");
            long c =  0;
            c = new Sum().add(a,b);
            System.out.println(" the result is " + c + "\n");
         }
}
```

Figure 7 Java code using JNI (Sum,java)

To compile the code, use the **javac Sum.java** command. To generate the C header file (Sum.h), use the **javah -jni Sum** command. Figure 8 on page 18 shows the JNI C implementation for the add function (Sum.c).

```
#include <stdio.h>
#include <jni.h>
#include "Sum.h"

JNIEXPORT jlong JNICALL Java_Sum_add
(JNIEnv * env, jobject obj, jint ai, jlong al)
{
  int a = (int) ai;
  long l = (long ) al;
  return (jlong) (a + l);
}
```

Figure 8 C implementation of the add function (Sum.c)

To generate 64-bit object code from the native C code, use the following command:

```
c89 -c -o Sum.o -I/JDK14/J1.4/include -W"c,lp64,dll,expo,langlvl(longlong)" Sum.c
```

To generate the libSumNative library, which contains the 64-bit implementation of the add function, use the following command:

```
c89 -o libSumNative.so -Wl,lp64,dll Sum.o
```

We set the LIBPATH environment variable to point to the directory containing the 64-bit libSumNative.so DLL. We execute the application using the **java Sum** command. Output is shown in Figure 9.

```
Class running
the result is 31289345
```

Figure 9 Output of Sum JNI code

Tip: If you encounter an error such as:

```
Could not load dll : libSumNative.so
: EDC5254S An AMODE31 application is attempting to load an AMODE64 DLL load module.
```

Ensure that the 64-bit JVM is used.

You can find more information about porting Java applications to 64-bit on the developerWorks Web site at:

http://www.ibm.com/developerworks/java/jdk/64bitporting/64BitJavaPortingGuide.pdf

Related publications

ITSO publications

- *XPLink: OS/390 Extra Performance Linkage*, SG24-5991
- *C/C++ Applications on z/OS and OS/390 UNIX*, SG24-5992
- *Tuning Large C/C++ Applications on OS/390 UNIX System Services*, SG24-5606

Other resources

- *z/OS Language Environment Programming Guide for 64-bit Virtual Addressing Mode*, SA22-7569
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767

Referenced Web sites

- Open Systems 64-bit Programming Models

  http://www.opengroup.org/public/tech/aspen/lp64_wp.htm
- Porting Guide - Moving Java applications to 64-bit systems

  http://www.ibm.com/developerworks/java/jdk/64bitporting/64BitJavaPortingGuide.pdf

The team that wrote this Redpaper

This Redpaper was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Vamsi Kulukuru is a Software Engineer at IBM Global Services in India. He has three years of experience in application software development on C, C++, and Java. He holds a degree in Computer Science from Regional Engineering College of Warrangal in Andhra Pradesh, India.

Thanks to the following people for their contributions to this project:

Rich Conway
International Technical Support Organization, Poughkeepsie Center

Don Ault, Clarence Clark III, Anuja Deedwaniya, Magdalen Leung, Barry Lichtenstein, Tien Nguyen, John Rankin
IBM Poughkeepsie

Kelly Arrey, Edison Kwok, Michael Wong
IBM Toronto

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 U.S.A.

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.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Send us your comments in one of the following ways:

- Use the online **Contact us** review redbook form found at:
  ibm.com/redbooks
- Send your comments in an email to:
  redbook@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYJ Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

developerWorks®	IMS™	S/390®
@server®	Language Environment®	z/OS®
ibm.com®	OS/390®	
IBM®	Redbooks (logo) ™	

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.