WebSphere Application Server for z/OS

C/C++ Code Considerations With 64-bit WebSphere Application Server for z/OS

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP101095** under the category of "White Papers"

Version Date: August 24, 2007
See Document Change History on page 11 for a description of the changes in this version of the document

IBM Software Group

Frank Golazeski
IBM WebSphere Application Server for
z/OS Development
267-391-1015
fgol@us.ibm.com

© 2007, IBM Corporation, Poughkeepsie, NY

Many thanks for input, mentoring, and reviewing go to ...

Don Bagwell
of the IBM Washington Systems Center

David Follis,
Colette Manoni,
Kalyan Roy, and
Dave Wright
of the of the WebSphere for z/OS development organization.

Table of Contents

Introduction 1
The Trouble with Types 1
C/C++ macro assists for single sourced mix address mode code 3
Solution for structure mapping in a mixed address mode environment 5
Run time coexistence in a mixed mode environment by using glue code 8
Concluding Points 10
Document Change History 11

Introduction

In version 6.1.0.4, WebSphere Application Server for z/OS first supported 64-bit addressing mode. WebSphere administrators can make a quick configuration change to a server, restart it, and then very large amounts of virtual storage become immediately available in the server's address space. Applications running on the server can take advantage of this additional space if they are prepared to access addresses beyond what a 31-bit pointer can hold. This paper will highlight things to consider when migrating C/C++ server applications to run in a 64-bit server environment.

Java applications are for the most part ready to go because the application byte code can handle 64-bit pointers if the Java Virtual Machine (JVM) can handle them. WebSphere Application Server for z/OS version 6.1 comes packaged with two JVMs, one for each address mode. So either way, Java applications can run on servers in both modes as long as they are not calling across the interface to native code, that is, the JNI.

C/C++ applications require a bit more thought in order to exploit the benefits of 64-bit mode. A C/C++ program gets set in stone when it is compiled because the resulting object program has built in assumptions regarding pointer lengths and other architecture attributes. It cannot necessarily change during execution in order to adapt to a new addressing scheme of the machine. Once a pointer is compiled as four bytes, there's not much chance to cram five or more bytes into it. Any address beyond the 2 GB address “bar” cannot be seen with a 31-bit pointer.

Application migration can be less of a chore and done in a way that allows for easier code maintenance in the future. There are some pitfalls you can avoid so that the inventory of every C/C++ application you have is not suddenly doubled because you need to maintain a 31-bit version and a 64-bit version. Extraneous copies of source parts make for twice the necessary work every time a fix is needed. One fix becomes two. Beside needing to do duplicate coding for fixes and enhancements, another danger is that the two similar modules, one 31-bit and the other as 64-bit, will begin to grow apart from each other if changes done in one are not done in the other.

Here is a worthwhile goal in migrating to 64-bit mode: Even though a C/C++ application has to run in two address modes, code it in such a way that it can 1) live as **one** source file with as little duplicate code as possible, and 2) can compile into the **two** executable modules.

The Trouble with Types

As outlined earlier, a primary compatibility problem between different address modes is in the size of the address itself. In software terms, this is the number of bytes that are needed to hold the largest possible address the program might encounter. Prior to WebSphere server for z/OS version 6.1.0.4, C/C++ applications only needed four byte pointers as 64-bit mode was not supported. The programmer usually writes C code without wondering how large a pointer is; the compiler hides that detail. Enabling 64-bit addressing can be as simple as re-compiling the source code with a _LP64 flag which directs compilation to build all pointer and long data types as 64-bit capable, that is eight bytes in length, rather than the four bytes that pre-ESA/ME architecture zServer machines handled.

© 2007, IBM Corporation, Poughkeepsie, NY

Note: _LPI64 is a variation of the _LP64 option that creates eight byte integers in addition to pointers and longs. The paper does not discuss the implications of eight byte verse four byte integers.

If a server environment requires some applications to run in either 31-bit mode or 64-bit mode, the _LP64 flag is a handy way to create the two versions of the executable code from the single source program file. Caution is required though if certain assumptions in your code depend on knowing how big a field is. This can happen in structures because the byte length of a pointer or a long affects the offset of subsequent fields. If those fields need to be at a specific offset or on a particular boundary then data in structures that gets passed across applications of different address modes might have fields at unexpected offsets.

Similarly, if longs are used to hold pointers we can feel glad that they change length as the pointers do, depending on compile mode. But since longs and ints are often used together in calculations and assignments, care must be taken to make sure there are no problems if they are different length fields in 64-bit mode (longs as eight bytes and ints as four) but the same length in 31-bit mode (both are four bytes.)

The following table compares the data type lengths in standard 31-bit and in _LP64 compilations.

Data Types	Size (31bit)	Size(64bit)
int	4 bytes	4 bytes
long	4 bytes	8 bytes
long long	8 bytes	8 bytes
Pointer	4 bytes	8 bytes
__ptr32	4 bytes(N/A)	4 bytes address
jint	4 byte	4 bytes
jlong	8 bytes	8 bytes
size_t	4 bytes	8 bytes

As an example of the impact of type lengths between the two modes, suppose you had the following data declaration.

```
struct {
      int sequence;
      int * count_ptr;
      long bobs_your_uncle;
} purchase_rec_typ;
```

A simple structure like this composed of an integer, pointer, and long is a mere twelve bytes in 31-bit mode compiles. That makes sense as the three data items are each four bytes long. When the compilation is done with the _LP64 flag set, we might expect the structure length to grow by eight bytes since the long and pointer are each four bytes longer than before. A quick look at the listing

from the 64-bit compilation though shows the structure is actually *24 bytes* in size, or four bytes longer than we thought it would be. This might seem surprising at first until we realize that both of the two newly eight byte data items, the pointer and the long, each want to be on a double word boundary. Since the four byte int throws that off, there is a four byte padding inserted before the count_ptr field.

Important Rules of Thumb:

1) Use a pointer or long data type when you need a field length to be adjusted depending on the address mode.

2) Use long long and int data types when the field length needs to be consistent across the two modes.

   Corollary to #2: Use the __ptr32 modifier on pointer declarations when you want them to be a consistent four bytes, as in: `int * __ptr32 count_ptr;` in both compile modes.

3) Word and double word data declarations will tend to want corresponding boundary alignments in either compilation mode.

If you want a single source file to produce the 31-bit mode and 64-bit mode load modules, you will need to take the above rules into consideration.

C/C++ macro assists for single sourced mix address mode code

With forewarned knowledge of the data type related issues with running a C/C++ module through 64-bit mode and 31-bit mode compilations, let us look a bit at how to implement some code that can straddle the two addressing worlds.

The first bit of good news is that your code can interrogate the address mode compiler flag during macro processing. If you test for the presence, i.e. definition, or absence of the _LP64 indicator you can adjust the code that actually gets poured into the compiler. The following C/C++ macro code segment illustrates the idea:

```
#ifdef _LP64
   …64bit version of code segment
#else
   …31bit version of code segment
#endif
```

So… you might code something like the following:

```
purchase_rec_typ purchase_rec
  #ifdef _LP64
    __attribute__ ((aligned(8)))
  #endif
  ;
```

When compiled in 31-bit mode, the above results in this being seen by the compiler:

```
purchase_rec_typ purchase_rec;
```

But the compiler sees this in 64-bit mode:

```
purchase_rec_typ purchase_rec __attribute__ ((aligned(8)));
```

This code could be useful, for example, in avoiding the problem of getting an unexpected four byte padding in 64-bit mode that was illustrated earlier in **The Trouble with Types**. Thanks to this `#ifdef` logic, structures of type purchase_rec_typ will not have slack bytes added by the compiler.

Suppose you were interested in taking advantage of much larger data buffers that are possible when the server runs in 64-bit mode. You might use a set of code such as this next example. Note that there is some duplicate code but at least it is a small amount and it is also contained within a localized area.

```
#ifdef _LP64
   #define MAXIMUM_BUFFER_SIZE        2*1024*1024*1024
   #define MAXIMUM_MESSAGE_LENGTH     (1024*1024-1)
#else
   #define MAXIMUM_BUFFER_SIZE        1024*1024
   #define MAXIMUM_MESSAGE_LENGTH     (1024-1)
#endif
```

The macro time test for the _LP64 flag is useful to help direct run time processing as well. Assignments involving data types of different lengths can be handled by casting. What if you wanted to protect against losing data when you assign a long value to an integer? In 31-bit mode that is not a concern; this could be problem in 64-bit mode when longs and ints are suddenly different sizes. Assignments like this could become problematic when migrating to 64-bit code:

```
long larger;
int smaller;
.
.
.
smaller = (int) larger;
```

With some macro sleight of hand. we can create the proper execution time code that can guard against accidental data loss from assigning an eight byte value into a four byte field. #ifdef can create a simple assignment in 31-bit mode and a more thorough one for 64-bit compiles. Take a look at this code example:

© 2007, IBM Corporation, Poughkeepsie, NY

```
typedef union{
     long long really_long;
     struct{
            int high_int;
            int low_int;
           };
}long_to_int;


#ifndef _LP64  //31-bit version
  inline int assignLongToInt( long from_long )  {
    return (int)from_long;
  }
#else  // function to be used under 64-bit mode
  inline int assignLongToInt( long from_long )  {
    long_to_int local;
    local.really_long = from_long;
    if (local.high_int!=0)  {   // Is the value too large to fit?
       // Yes: Throw an exception, put out a message, whatever.
    }
    return local.low_int;
  }
#endif
```

Note that the above #if is actually different than ones we have used so far in the code examples. It is #if**n**def and it checks that the _LP64 flag is ***not*** defined, which is true in 31-bit compilations. When this code is built for 64-bit mode execution, before the assignment is done the high order word gets tested to make sure it does not contain data that will get lost. If there is data there, that means that an incorrect assumption was made when the application 64-bit migration was designed. The code detects the error at run time and can then wave a big flag so that the loss of data can be trapped. So the above assignment can be replaced with this:

```
smaller = assignLongToInt(larger);
```

Solution for structure mapping in a mixed address mode environment

Consider a situation in which a C/C++ application runs on a WebSphere Server for z/OS that sometimes executes in 64-bit mode and sometimes in 31-bit mode. Suppose the application uses a structure that maps data which is passed between two modules in your application system; one runs in 64-bit mode and the other in 31-bit mode and both access the same data structure. Problems can arise if the shared data structure contains longs or pointers because field sizes will change so at least one of the program's will have an inaccurate understanding of the structure's layout. Field widths and field offsets in the data declaration will differ from the actual data structure in storage. See the following example:

	31-bit mode	64-bit mode
`struct {`		
`element_typ * next_ptr;`	4 bytes at offset 0	8 bytes at offset 0
`long item_key;`	4 bytes at offset 4	8 bytes at offset 8
`char [30] desc;`	30 bytes at offset 8	30 bytes at offset 16
`} rec_typ;`	Length 38 bytes	Length 46 bytes

Here we cannot tolerate the impact of the field differences in the structure because it affects how the code will look at real data in the machine at run time. We might not know where to find the desc field. Depending on who created the actual instance of the data, it might be at offset 8 or 16. Depending on what mode in which the referencing program was built it might be looking at the correct offset or not. The structure mapping needs to be done in a way that's friendly to both worlds of address mode.

The easier problem is the numeric data item named item_key. Changing it to either a long long or an int seems like a reasonable plan because it would ensure a non-changing field width between the two modes. The question is whether the code that uses item_key can handle eight bytes of length or whether it has to be restricted to four bytes. Some additional investigation with the help of a source library cross reference tool will let you know how the data is used in order to spot places that tell you which field length choice to make. Casting might be needed to ease concerns by the compiler or you simply might need to change declarations of local copies of item keys to be in sync with the field in the structure.

The size difference with the pointer is a bit trickier because it is often harder to force the code into a specific pointer size in a *one size fits all* approach.

First consideration: when data gets created by 64-bit code, by default it is in the C run time heap which is located above the 2GB address bar. Pointers to it must be wider than four bytes. If that data then needs to be passed to an application that is running in 31-bit mode, there are interface issues to resolve, like how does the called application "see" above the bar resident data? See "Run time coexistence in a mixed mode environment by using glue code" on page 8 for tips on passing information above and below the bar.

Second, even if the data structure is used only by applications running in one mode at a time **if** that mode could vary then the way in which the data structure is defined is important. It can affect how easily you can create the two executable modules from single source files. The trick with pointers in structures is to make them independent of the mode in which they are compiled so that in both cases the pointer size adapts to the necessary width yet the offsets of it and the data following it remain consistent. So called *modeless pointers* get the job done. Let's revisit the structure definition for rec_typ to see how modeless pointers can help.

To eliminate one question for the sake of simplifying this example, we will assume we can change the item_key to an int. As far as the pointer is concerned, we would like it to be eight bytes in 64-bit mode and four bytes in 31-bit mode. We would also like anything following it to be found at the same offset in both modes. Since we cannot force an eight byte pointer to occupy four bytes,

© 2007, IBM Corporation, Poughkeepsie, NY

maybe we can allow a four byte pointer to spread itself out over eight bytes, thus keeping everything at pretty much the same offset. Step one is to use the macro test of the _LP64 flag as shown here:

	31-bit mode	64-bit mode
`struct {`		
`  #ifndef _LP64`		
`    int :32;`	4 bytes at offset 0	0 bytes at offset 0
`  #endif`		
`  element_typ * next_ptr;`	4 bytes at offset 4	8 bytes at offset 0
`  int item_key;`	4 bytes at offset 8	4 bytes at offset 8
`  char [30] desc;`	30 bytes at offset 12	30 bytes at offset 12
`} rec_typ;`	Length 42 bytes	Length 42 bytes

Some things worth noticing.

1. The data structure is now the same length in both compile modes.
2. The space in the structure for the pointers is consistently eight bytes. In one case it is made up of a four byte filler space followed by a four byte pointer. In the other case it is one eight byte pointer. In both cases eight bytes are used.
3. The four byte filler is defined as a set of 32 bits. Why that instead of int or char[4]? Many languages allow defining an unnamed data item; C and C++ are not among them – unless you are declaring a set of bits. So to avoid being required to come up with needless data names like padding1, padding2, … padding-infinity, it seems easier to sneak the four bytes through without creating a name, so we define four bytes of bits.
4. Overall every field is at a consistent offset though an argument could be made that the pointer is really at two different offsets – plus four and plus zero. That is actually advantageous though. A look how the pointer would appear in storage shows why. Assume the next_ptr pointer contains an address of 0x12345678, an instance of a rec_typ might look like this:

   ```
   00000000 12345678 00C0FFEE C3D6D3C4  |............COLD|
   40D6D940 C8D6E340 C4D9C9D5 D240C3E4  | OR HOT DRINK CU|
   D7000000 00000000 0000               |P.........      |
   ```

   The trick question is, does this represent the 31-bit structure or the 64-bit structure? Fortunately the answer is as we would hope: both. The pointer data does not give any information away because a word of zeros followed by 0x12345678 could be a doubleword pointer or it could simply be a word of zeros followed by a pointer. The important aspect of this is that it works correctly in both sides of the address mode machine state.

A bothersome question in the above example point might be that if the execution mode of the application were 64-bit addressing, would not the next_ptr really be pointing to something beyond the 2GB address bar? Normally yes it would because dynamic (via malloc or new) and local data will generally come out of the heap, which would be in 64-bit space. You can define that the heap be below the 2GB address bar but that defeats some of the purpose of running in 64-bit mode. If there are specific instances where you need some data to stay below the bar no matter what address mode your application will run in, you can allocate the storage with the __malloc31 method.

© 2007, IBM Corporation, Poughkeepsie, NY

This will get space from the below-the-bar heap and can allow data to be passed between 64-bit and 31-bit programs.

Important note: If you use a modeless pointer in a data structure, the four byte padding that precedes the pointer in the 31-bit version might end up containing garbage. If that data were passed to a 64-bit piece of code, it would end up interpreting garbage word as part of the eight byte pointer. **Be sure to initialize your data structures to binary zeros.**

Another important note: Pointers and long data are doublewords when compiled in 64-bit (_LP64) mode. They will want to be on doubleword boundaries so be sure to define modeless pointers on doubleword boundaries within the structure. Some helpful news is that dynamic data in 64-bit C/C++ gets allocated on doubleword boundaries and likewise the static data declarations start on doublewords.

A handy note: Modeless pointers can be easily defined with macro logic to simplify your source code. If you use a standard header file in which you declare installation wide constants or macros, you can add the following modeless pointer definition:

```
#ifdef _LP64
  #define __mptr(typ,nam)    \
          typ *nam
#else
  #define __mptr(typ,nam)    \
          int :32;           \
          typ *nam
#endif
```

Then in the rec_typ data structure you can declare next_ptr as:

```
__mptr(element_typ, next_ptr);
```

and the pointer will occupy eight bytes in both modes without any further hand modification or duplicate code.

Run time coexistence in a mixed mode environment by using glue code

Until every byte of code in the application suite of an installation runs exclusively in 64-bit mode, there will be the requirement to wonder about any piece of data that gets passed between two programs. Can the data be instantiated above the 2GB address bar or does it have to remain below? That decision affects how it is defined, how it is allocated, and how other things point to it.

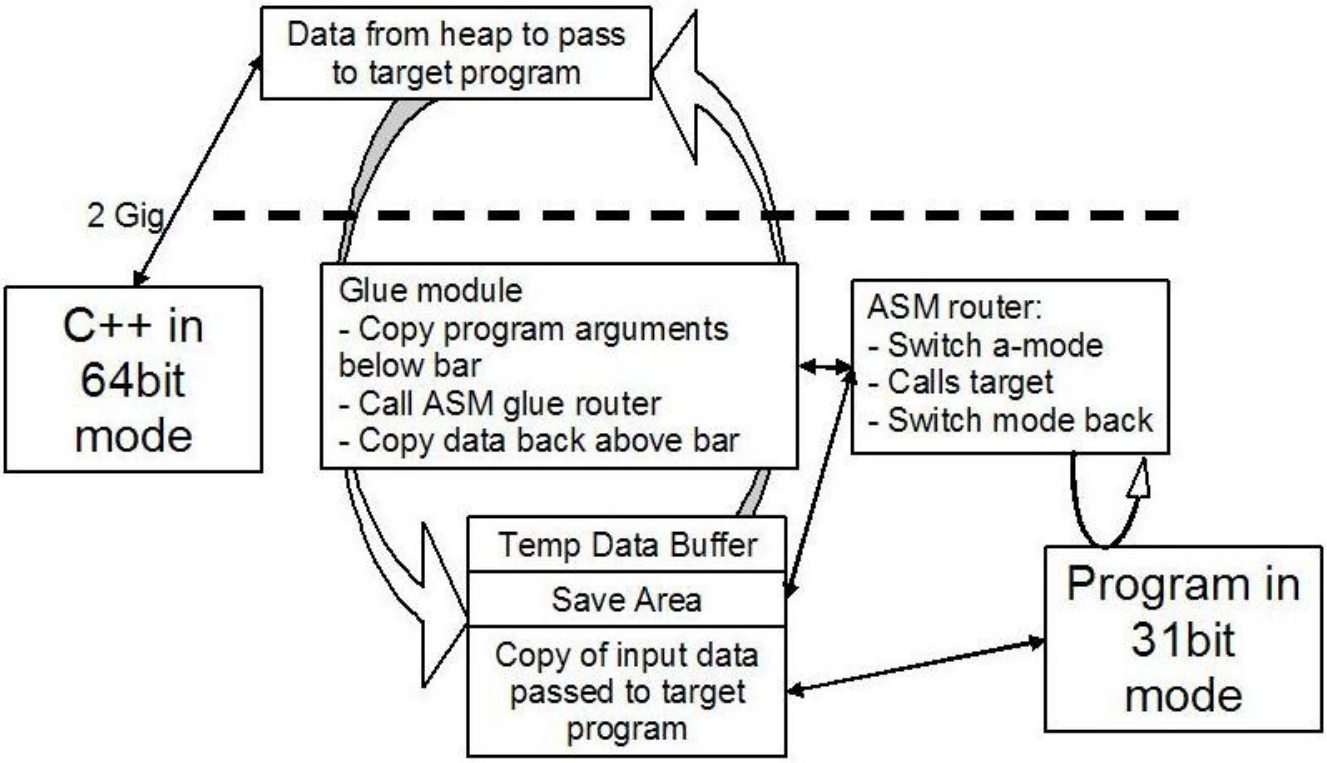
If it is unavoidable that you are going to have an interface between a 64-bit module and a 31-bit module though which you will pass data, you might need to have an intermediate program to manage the data passing between the caller and the service program. We will refer to this intermediate layer as a *glue module*. In this approach, if the machine is running in 64-bit mode, the caller invokes the glue module rather than calling the service program directly. The glue layer copies the data across the 2 GB address bar. switches machine mode, and then passes control to the service program. On the return path the glue module performs the same sort of steps but in the opposite manner. Consider the situation where a migrated 64-bit application invokes a legacy service that runs in 31-bit mode. Data is passed to the service and we expect some return information. For this mixed mode environment, a glue layer is required. It performs the following steps:

- ✓ It obtains a temporary buffer below the 2GB bar,
- ✓ copies the data passed from the caller into the temporary buffer,
- ✓ switches machine mode to 31-bit addressing, and
- ✓ calls the target service program passing the buffer data,

Upon return the glue module

- ✓ changes the machine mode back to 64-bit mode,
- ✓ copies the return data above the bar to wherever the caller expects it,
- ✓ frees the temporary buffer, and finally
- ✓ returns to the calling program.

This set of steps is more or less shown in the following diagram.



Notes:

- ✓ Since the machine mode switching is done by means of assembler instructions SAM31 and SAM64, we need a short assembler program to do the switch. The glue module actually calls that assembler code and passes to it the address of the target service.
- ✓ A register save area will be needed by the assembler code, the glue module should allocate it as part of the temporary buffer. This will help keep the linkage organized, allow register saving, and provide a means for passing the target program's address to the assembler router.

If your service is above the 2GB bar and the calling application runs in 31-bit mode, your glue layer can be much simpler. Any arguments passed up to the service can stay below the bar but the interface might be expecting eight byte addresses; the glue code might have to re-pass any

pointers. Any return data will need to be copied down to wherever the caller expects to find it. You might also have to manually switch the machine modes.

Within the calling program, the interface can be managed with macro time processing code assists. Something like the following code might be useful:

```
#ifdef _LP64
  TargetRtnViaGlue (int a, int b, void * Addr_TargetRtn);
#else
  TargetRtn (int a, int b);
#endif
```

Concluding Points

Running WebSphere Application Server for z/OS in 6-bit mode offers many advantages to application systems by enabling huge address spaces for data storage and the subsequence run time ease in which this “in memory” data can be accessed, shared, and managed. While J2EE compliant Java applications ought to generally ready to run in 64-bit mode without modification, native applications like ones written in C/C++ might need some tweaking. The way in which the 64-bit code changes are done will determine how much future work duplication will be necessary for feature enhancements or code maintenance. Taking advantage of assistance built into the C/C++ language can help ease the period of time when application code needs to be ready to run in either 31-bit or 64-bit mode.

For additional information on running WebSphere Application Server for z/OS in 64-biyt mode see:

WP100920 - 64 Bit Addressing Support by Don Bagwell and Nick Carlin.

© 2007, IBM Corporation, Poughkeepsie, NY

Document Change History

Check the date in the footer of the document for the version of the document.

August 24, 2007	Original document.

End of WP101095