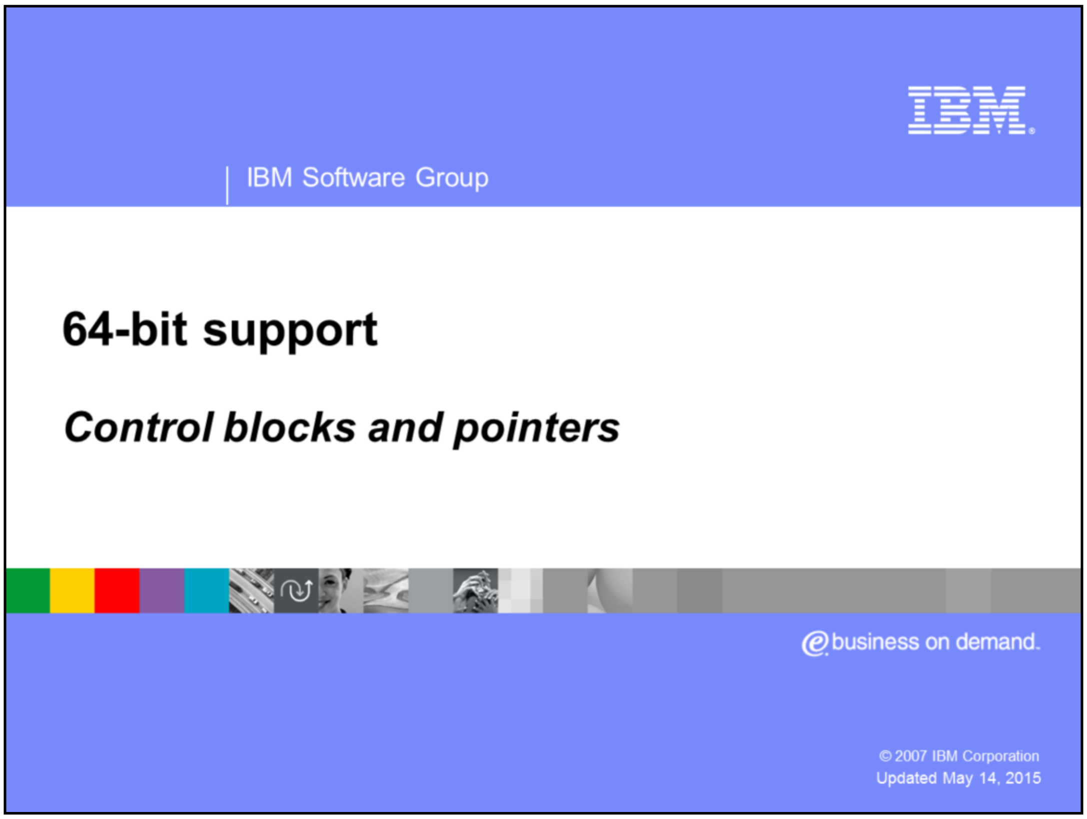


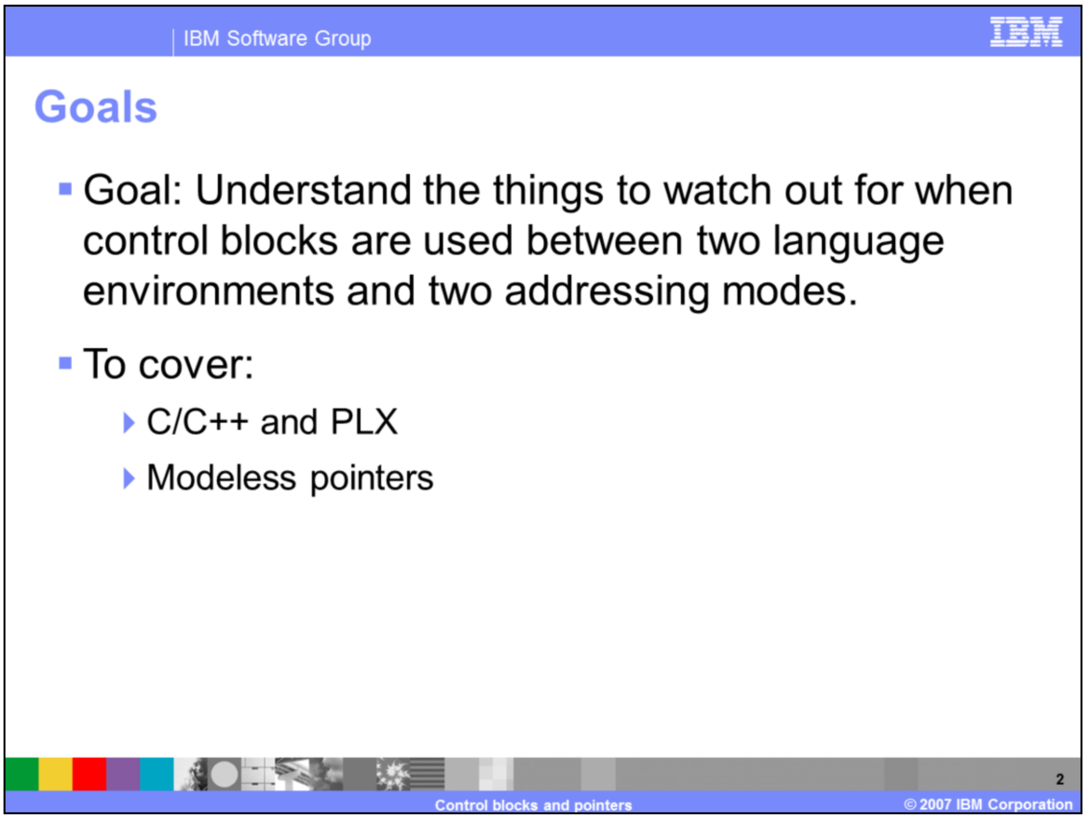
64-bit support

Control blocks and pointers

business on demand.

Updated May 14, 2015

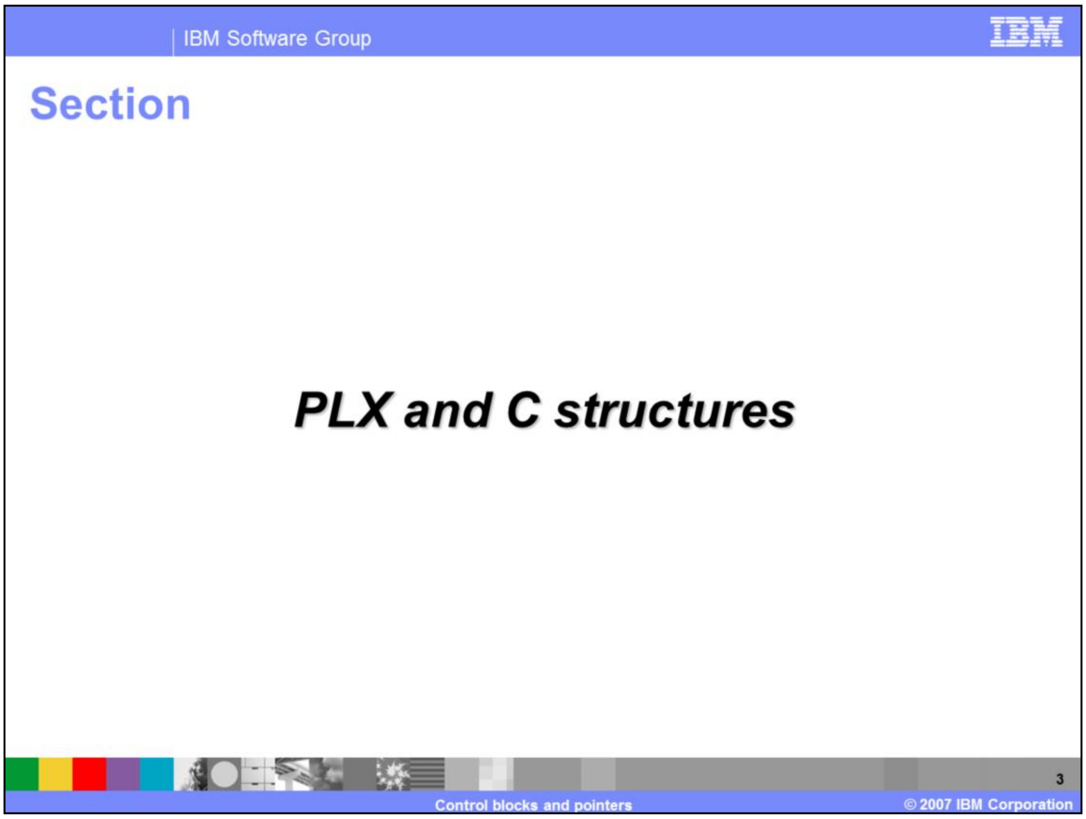
This presentation will discuss control blocks and pointers.



Goals

- Goal: Understand the things to watch out for when control blocks are used between two language environments and two addressing modes.
- To cover:
  - C/C++ and PLX
  - Modeless pointers

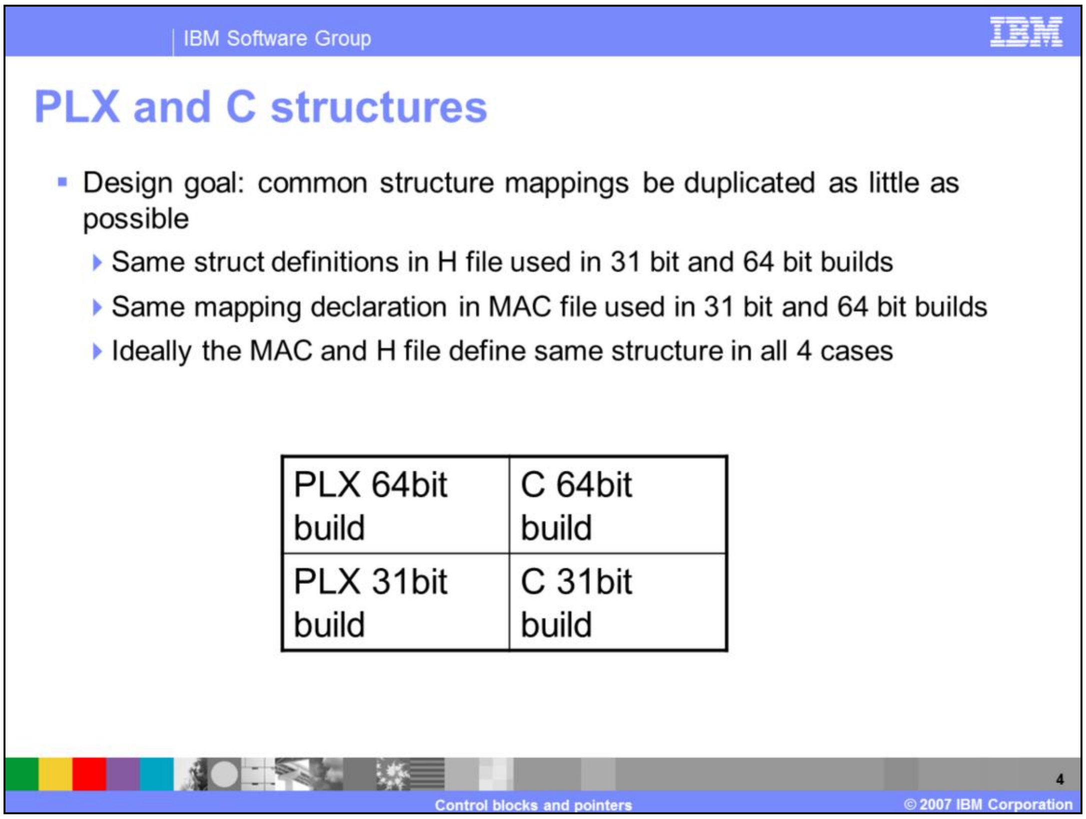
The goal of this presentation is to expose the hazards and solutions to maintaining consistent control block mappings without spending a lot of time managing duplicate code. This presentation will cover language considerations, some of which should be familiar and then introduce the concept of 'modeless pointers' or "mptrs" for short.



Section

PLX and C structures

This section will discuss PLX and C structures.


IBM Software Group

PLX and C structures

- Design goal: common structure mappings be duplicated as little as possible
  - Same struct definitions in H file used in 31 bit and 64 bit builds
  - Same mapping declaration in MAC file used in 31 bit and 64 bit builds
  - Ideally the MAC and H file define same structure in all 4 cases

PLX 64bit build	C 64bit build
PLX 31bit build	C 31bit build

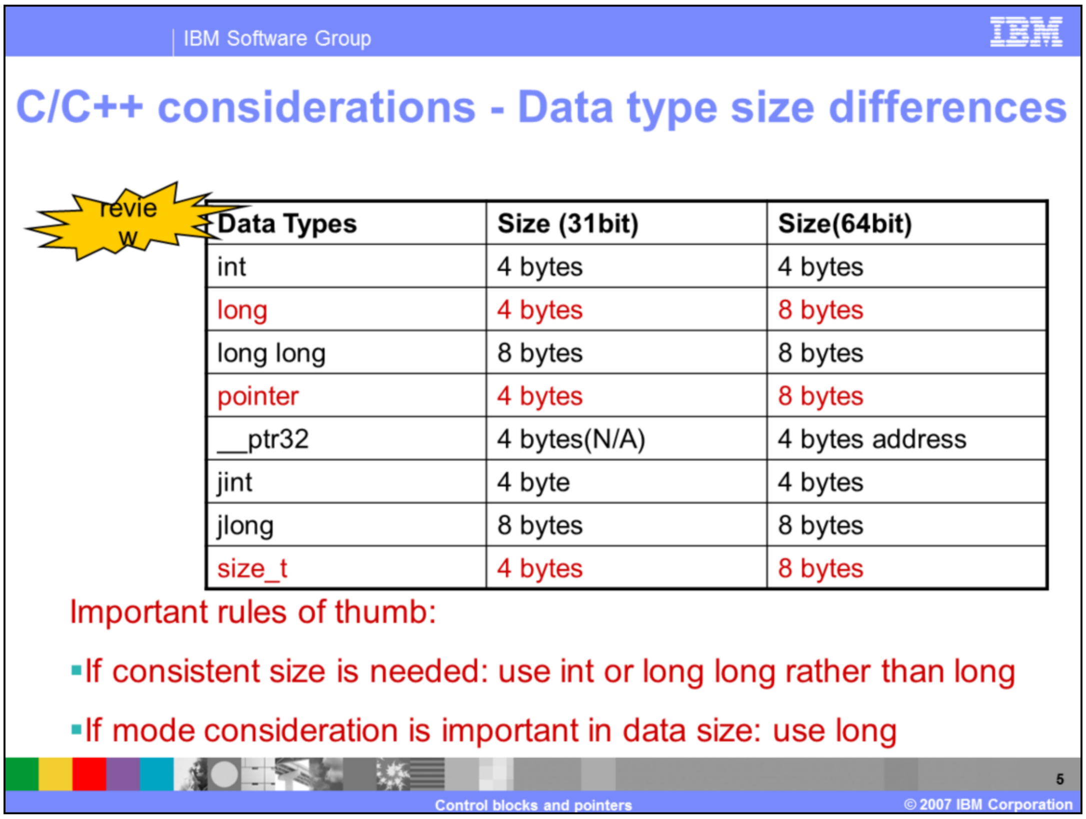
4

Control blocks and pointers

© 2007 IBM Corporation

64-bit support poses certain problems when dealing with mapping macros for control blocks. Ideally you should be able to map these control blocks without having several versions of code and these mappings would be consistent. The above table shows the area of support that has been addressed. There are control blocks shared between C and PLX that need to be consistent between 64-bit and 31-bit builds and be consistent across the two languages.

An interesting angle is that the PLX code is built almost exclusively in 31-bit mode. So the mapping it has for any given common control block has to match the C header file that maps it in 64-bit mode in the same driver build.



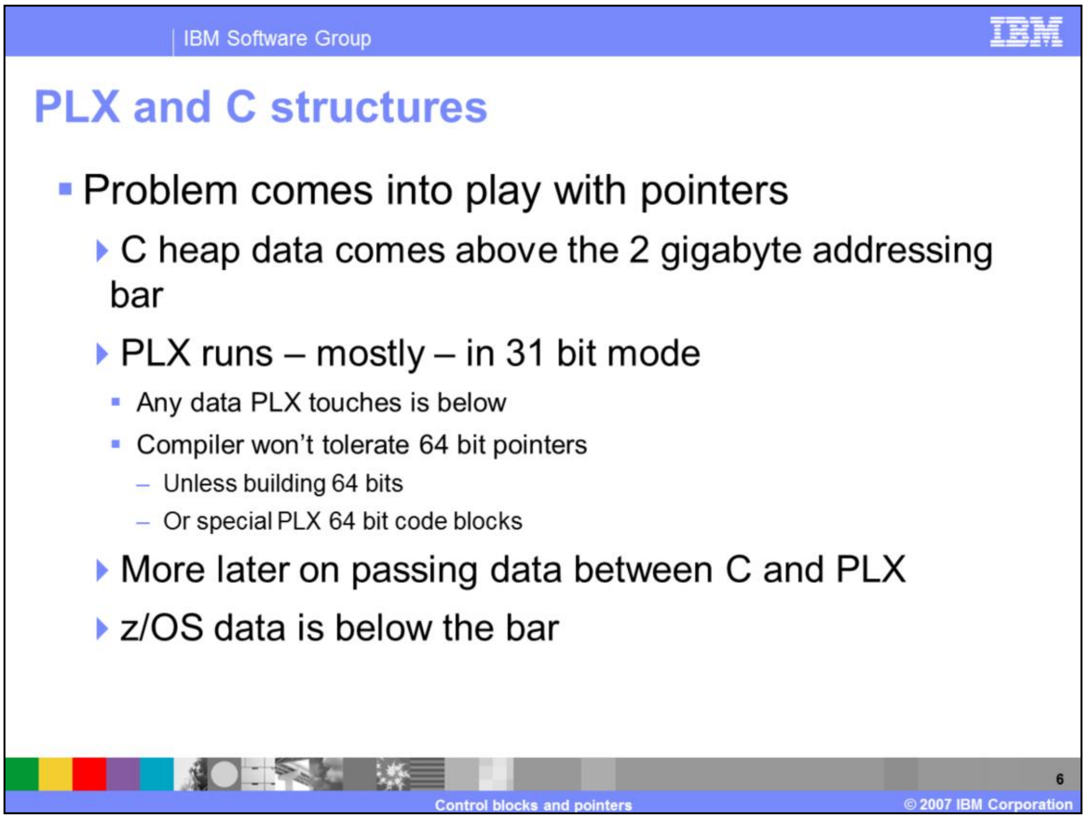
C/C++ considerations - Data type size differences

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

Important rules of thumb:

- If consistent size is needed: use int or long long rather than long
- If mode consideration is important in data size: use long

A quick review – remember that certain things in C change size between 31 and 64bit mode. These differences become problematic in control blocks.

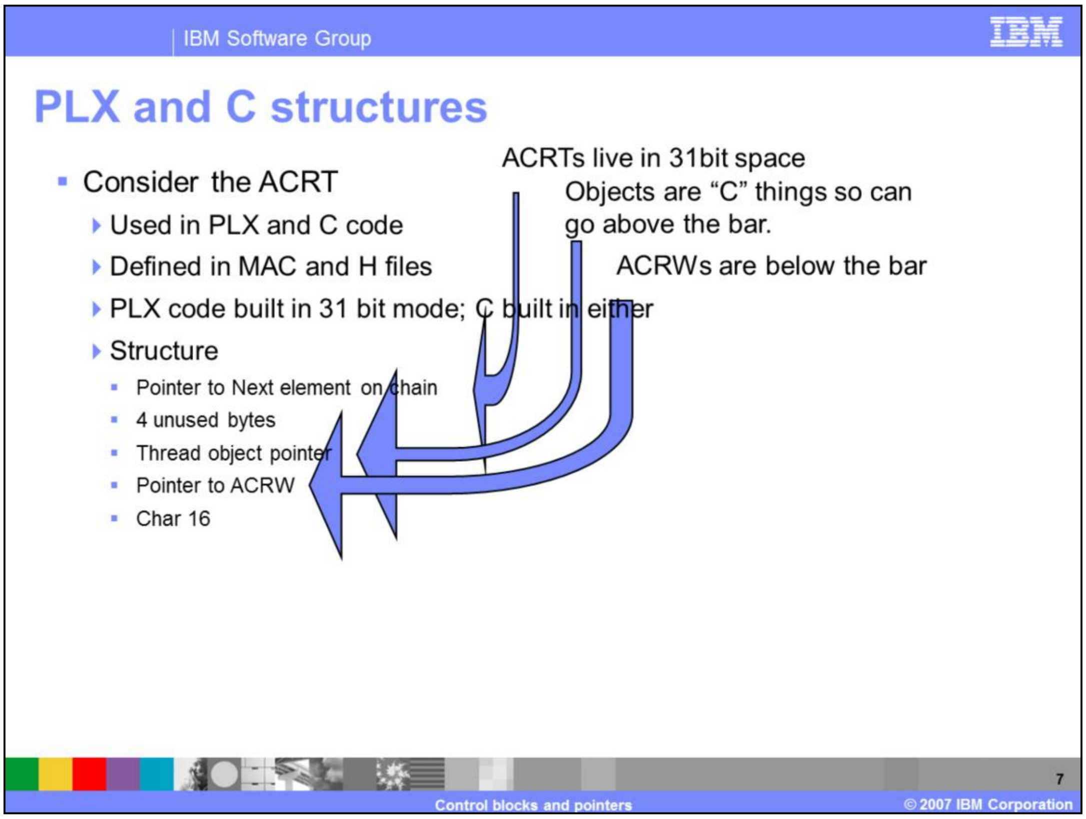


PLX and C structures

- Problem comes into play with pointers
  - C heap data comes above the 2 gigabyte addressing bar
  - PLX runs – mostly – in 31 bit mode
    - Any data PLX touches is below
    - Compiler won't tolerate 64 bit pointers
      - Unless building 64 bits
      - Or special PLX 64 bit code blocks
  - More later on passing data between C and PLX
  - z/OS data is below the bar

The problem surrounds the use of pointers. Having a control block without pointers the mapping is easier because the control block is the same between the two compile modes. An easier scenario would be to have all 64bit mode systems because there would be no question: all pointers would have to be 8 bytes whether the address is above the bar or below. However, since not all environments are 64-bit, this is impractical. Most, practically all, of the PLX code runs in 31-bit mode. It is fine to make every pointer 8 bytes but the PLX compiler is a bit strict in how it handles the code. It will not compile a ptr(64) in 31-bit mode.

Most of the system data on z/OS is also below the bar, which means more often than not four byte pointers are what get passed back and forth. It is impossible to dictate that all pointers be 8 bytes.



IBM Software Group

PLX and C structures

- Consider the ACRT
  - Used in PLX and C code
  - Defined in MAC and H files
  - PLX code built in 31 bit mode; C built in either
  - Structure
    - Pointer to Next element on chain
    - 4 unused bytes
    - Thread object pointer
    - Pointer to ACRW
    - Char 16

ACRTs live in 31bit space
Objects are "C" things so can go above the bar.
ACRWs are below the bar

Control blocks and pointers

7

© 2007 IBM Corporation

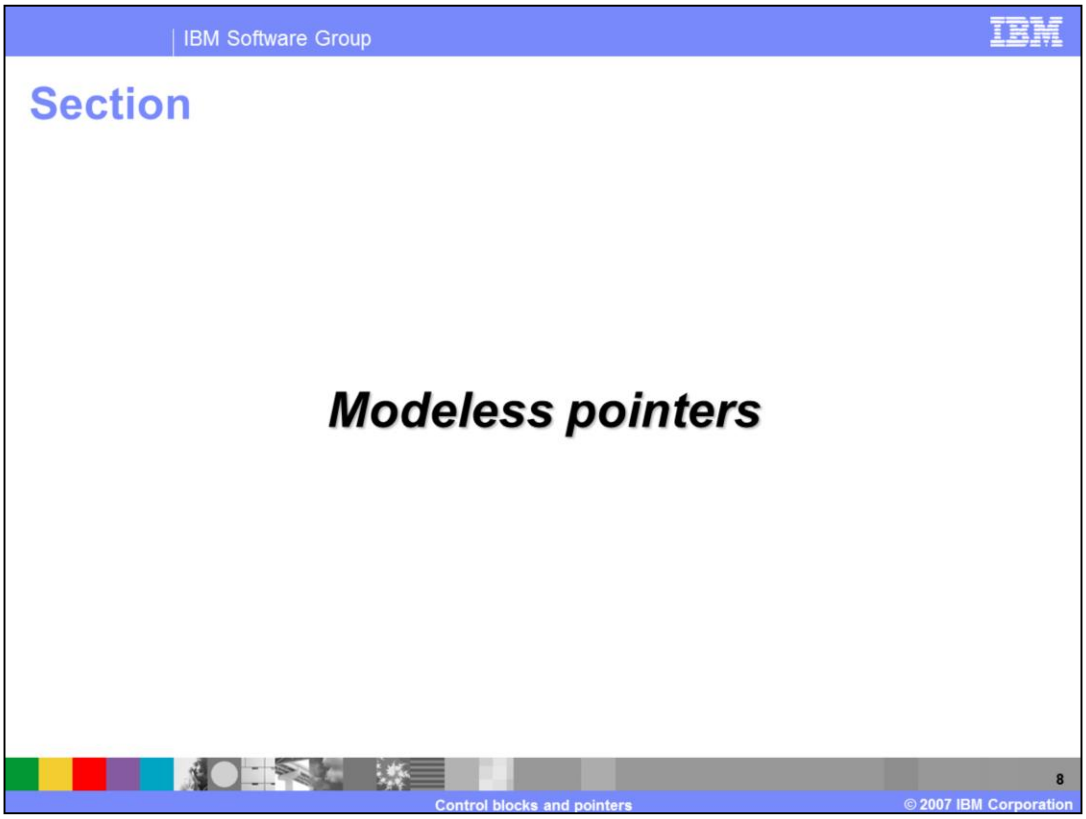
Here is an example of a control block that illustrates the compatibility situation. The ACRT is used in both C and PLX code. It is defined in both an H file and a MAC file. The PLX code is built in 31-bit mode, which is fine, but the C code is built in both. Left to it's own desires, the C compiler will end up generating two versions of offsets for fields in the same control block. One, because of 4 byte pointers and the other because the pointers are 8 bytes.

Let's look at some of the features of the ACRT.

It is used by PLX code which is mostly 31-bit mode so the ACRT has to live below the bar. That means the chaining pointer has to be a 4 byte pointer. 8 byte pointers will not work because the 31-bit code will look at offset zero for four bytes and find a zero. The C mapping is interesting because it will make an 8 byte pointer in 64-bit compile unless ones intentions are asserted. Another important point is that the ACRT has to be allocated below the bar.

The ACRT Points to C objects. They come from the C heap, which is above the bar. Things that could go above the bar have been allocated above the bar whenever possible. The ACRT pointer needs to be 8 bytes and the PLX code can not try to access it. Since it is a pointer to a C object, it is unlikely the PLX code will access it. Notice that problems may arise if this field is a four byte field in 31-bit mode and 8 bytes in the other.

There is a pointer to ACRWs. The pointer lives below the bar and likewise this pointer needs to be four bytes and stay four bytes.


IBM Software Group

Section

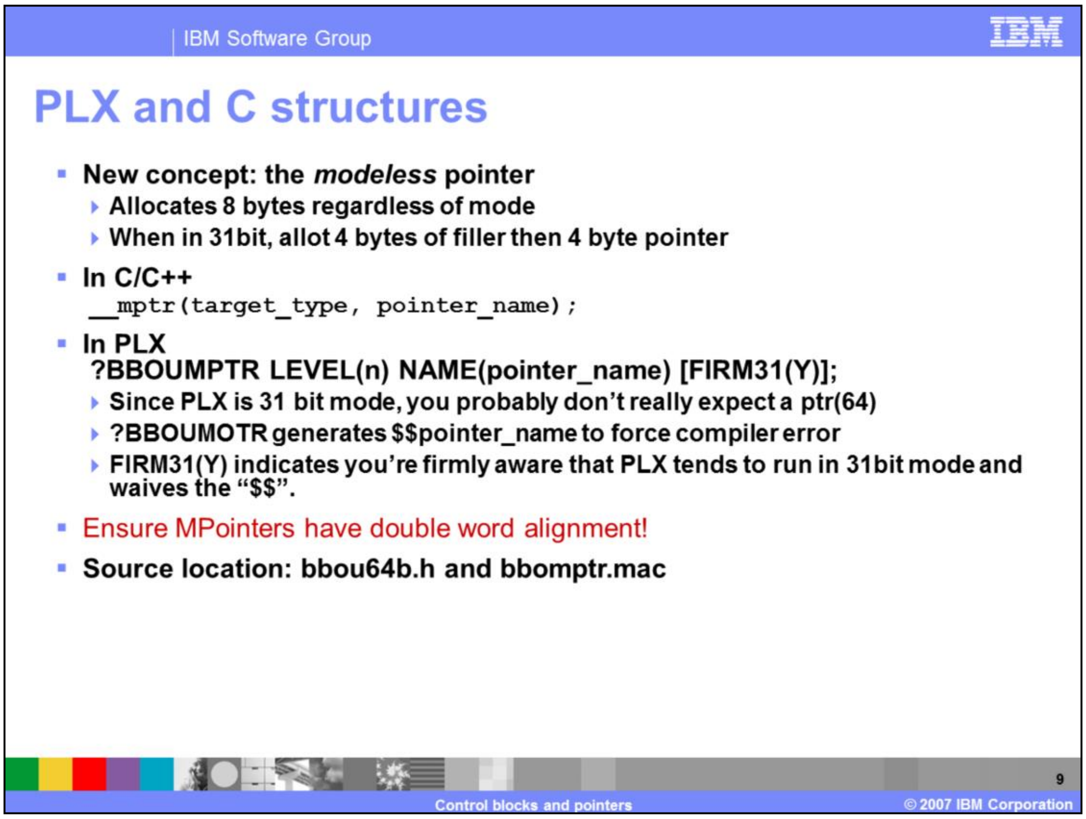
Modeless pointers

8

Control blocks and pointers

© 2007 IBM Corporation

This section will discuss a new time saving construct: the modeless pointer.


IBM Software Group

PLX and C structures

- **New concept: the *modeless* pointer**
  - **Allocates 8 bytes regardless of mode**
  - **When in 31bit, allot 4 bytes of filler then 4 byte pointer**
- **In C/C++**

```
__mptr(target_type, pointer_name);
```

- **In PLX**
  ?BBOUMPTR LEVEL(n) NAME(pointer_name) [FIRM31(Y)];
  - **Since PLX is 31 bit mode, you probably don't really expect a ptr(64)**
  - **?BBOUMOTR generates $$pointer_name to force compiler error**
  - **FIRM31(Y) indicates you're firmly aware that PLX tends to run in 31bit mode and waives the "$$".**
- Ensure MPointers have double word alignment!
- **Source location: bbou64b.h and bbomptr.mac**

9

Control blocks and pointers

© 2007 IBM Corporation

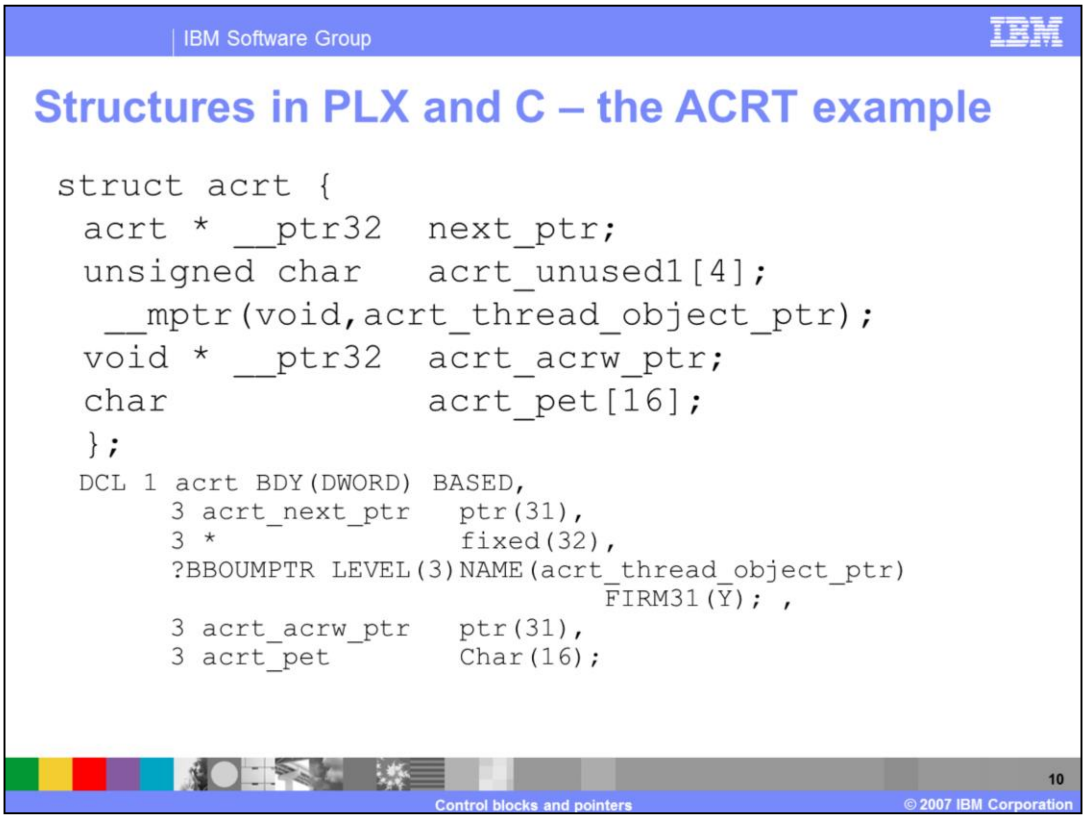
The problem with pointers is not that they change sizes, it is that the offset of everything in the structure that follows will get thrown about depending on the length of the pointer. So the mptr was created to maintain steady field offsets.

An mptr always takes up 8 bytes even though the pointer itself might be only 4 bytes. The advantage is that the pointer itself accommodates the amode of the code that is accessing it but within the structure mapping, the overall field width is consistently 8 bytes.

The chart shows how mptrs are declared in both C and PLX. These code macros expand into 8 byte pointers for 64-bit compiles. In 31-bit compiles, a mptr expands into a four byte filler and a four byte pointer. That is how the overall 8 byte width stays consistent in the structure. An important note is that 64-bit pointers must be on a doubleword boundary so be careful not to let the PLX compiler stick in slack bytes to ensure a proper alignment.

Among the oddities of the PLX declaration is the optional keyword FIRM31 which has one useful parameter: "Y". Consider that PLX code tends to run only in 31-bit mode so if you are declaring an mptr that is shared between C and PLX, it had better be a four byte pointer. But if the pointer exists in this control block but is not actually referenced in PLX you are fine. If the pointer is referenced in PLX it could be problematic for C to stick an 8 byte address into a field of which 31-bit mode PLX will use only the lower word. So to head off problems, the macro renames the PLX pointer filed and if it is referenced somewhere in PLX code you will get a compiler error. The FIRM31(Y) prevents the renaming of the pointer because you are telling the macro you are confident that the pointer will work fine in 31-bit PLX.

MPtrs are defined using macros in bbou64b.h in C and bboumptr.mac for PLX.


IBM Software Group

Structures in PLX and C – the ACRT example

```
struct acrt {
 acrt * __ptr32  next_ptr;
 unsigned char   acrt_unused1[4];
  __mptr(void,acrt_thread_object_ptr);
 void * __ptr32  acrt_acrw_ptr;
 char            acrt_pet[16];
 };
```

```
DCL 1 acrt BDY(DWORD) BASED,
      3 acrt_next_ptr   ptr(31),
      3 *               fixed(32),
      ?BBOUMPTR LEVEL(3)NAME(acrt_thread_object_ptr)
                                 FIRM31(Y); ,
      3 acrt_acrw_ptr   ptr(31),
      3 acrt_pet        Char(16);
```

10

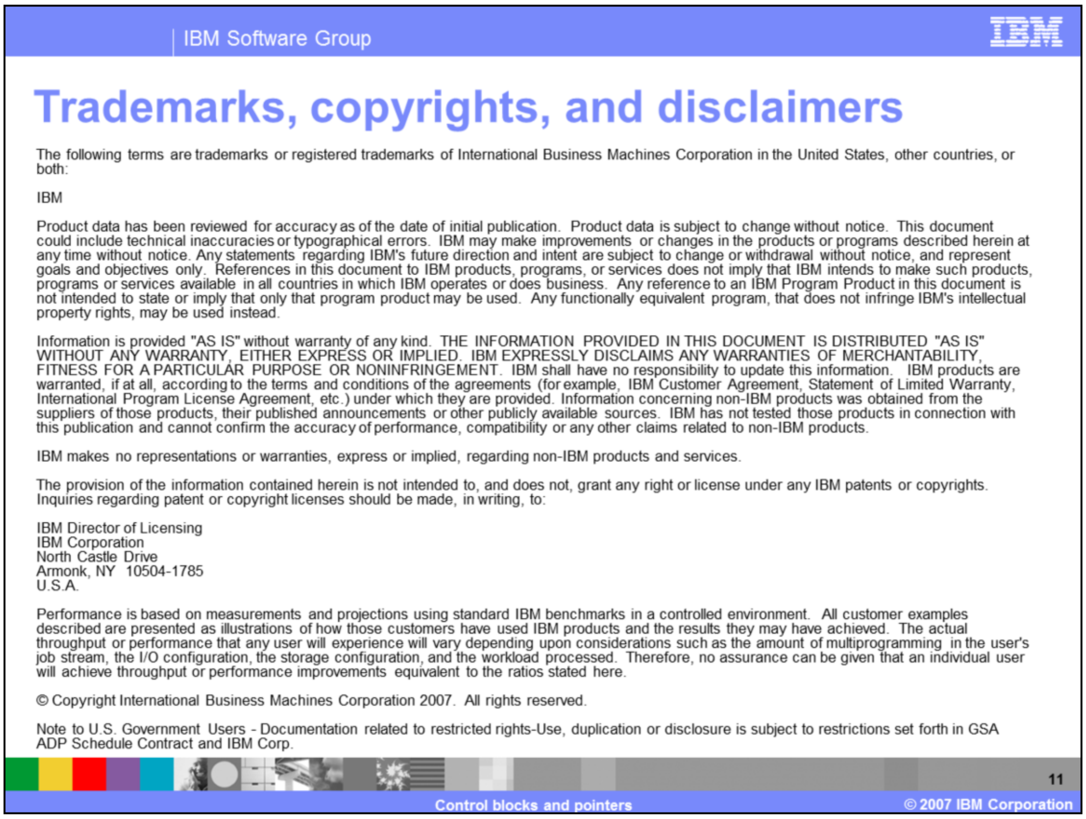
Control blocks and pointers © 2007 IBM Corporation

Here is the ACRT example shown in C and PLX declarations.

-The ACRT lives below the bar so its chain pointer needs to be four bytes in both compile modes. An important note is the __ptr32 in the C.

-The thread object pointer must be able to point to storage above the bar when the C code is running in 64bit mode so an mptr construct is used. The PLX code uses the FIRM31(Y) but that is probably unnecessary. Note that the mptr is on a DW boundary.

-Finally, the ACRW lives below the bar so the pointer to it gets a __ptr32 in C.



Trademarks, copyrights, and disclaimers