Keywords: MAXQ, IAR, Linker File, .xcl, IAR EWMAXQ

APPLICATION NOTE 4128

IAR EWMAXQ 4.0 Linker File—Simplified

By: Faisal Tariq
Dec 10, 2007

Abstract: This application note discusses the IAR Embedded Workbench™ (EWMAXQ) 4.0 linker file used when programming a MAXQ® microcontroller for an application. The article explains how to modify the IAR EWMAXQ linker configuration (.xcl) template file to work with different data RAM and program flash sizes. At the end of this application note the MAXQ2000 microcontroller is used as an example to show how simple it is to change the template for an application.

References to general information for the IAR Embedded Workbench, the MAXQ User's Guide, and relevant application notes are given.

Introduction

This application note provides general background on the IAR Embedded Workbench (EWMAXQ) linker configuration (.xcl) file. The IAR EWMAXQ integrated development environment (IDE) modifies this linker file to accommodate the different amounts of data RAM and program flash in a MAXQ microcontroller and when an application has special requirements.

Overview

The IAR EWMAXQ IDE requires a set of "personality files" to determine how to compile an application written in C. A set of these personality files contains information about a specific MAXQ microcontroller, and most files require no intervention from the user. There is, however, an exception. Within the EWMAXQ IDE 4.0 is a linker file that might require modification in situations such as: if development is done on a MAXQ processor that has a derivative with different amounts of RAM in data space or flash in program space; or if an application has special requirements, such as changing the location of constants in code space.

This application note does not discuss all the details of the linker file. Instead, the article gives the reader enough information to modify a linker file to manage his application needs. To modify the linker file the user will need to know the size of the RAM in data space and flash in code space on their MAXQ microcontroller.

For general details on the linker file, please refer to the *IAR Linker and Library Tools Reference Guide* and *MAXQ IAR C Compiler*, found in the IAR EWMAXQ help menu. This article assumes that the reader is familiar with the MAXQ architecture and the IAR EWMAXQ IDE. Additional useful references are cited at the end of this application note.

The Linker Configuration (.xcl) File Structure

Figures 1 and **2** show a typical IAR linker configuration file for a MAXQ microcontroller. Linker configuration files can be recognized by the .xcl extension and are found in the config directory of the IAR EWMAXQ installation directory.

For ease of discussion, the linker configuration example file here is divided into eight blocks, Block 1 through Block 8. The size of RAM in data space is 512 bytes (200h); the flash in program space is 64kB (10000h). **Note: the values entered (memory size) in the linker file are in bytes and hexadecimal. The user should remember this when modifying the linker file.**

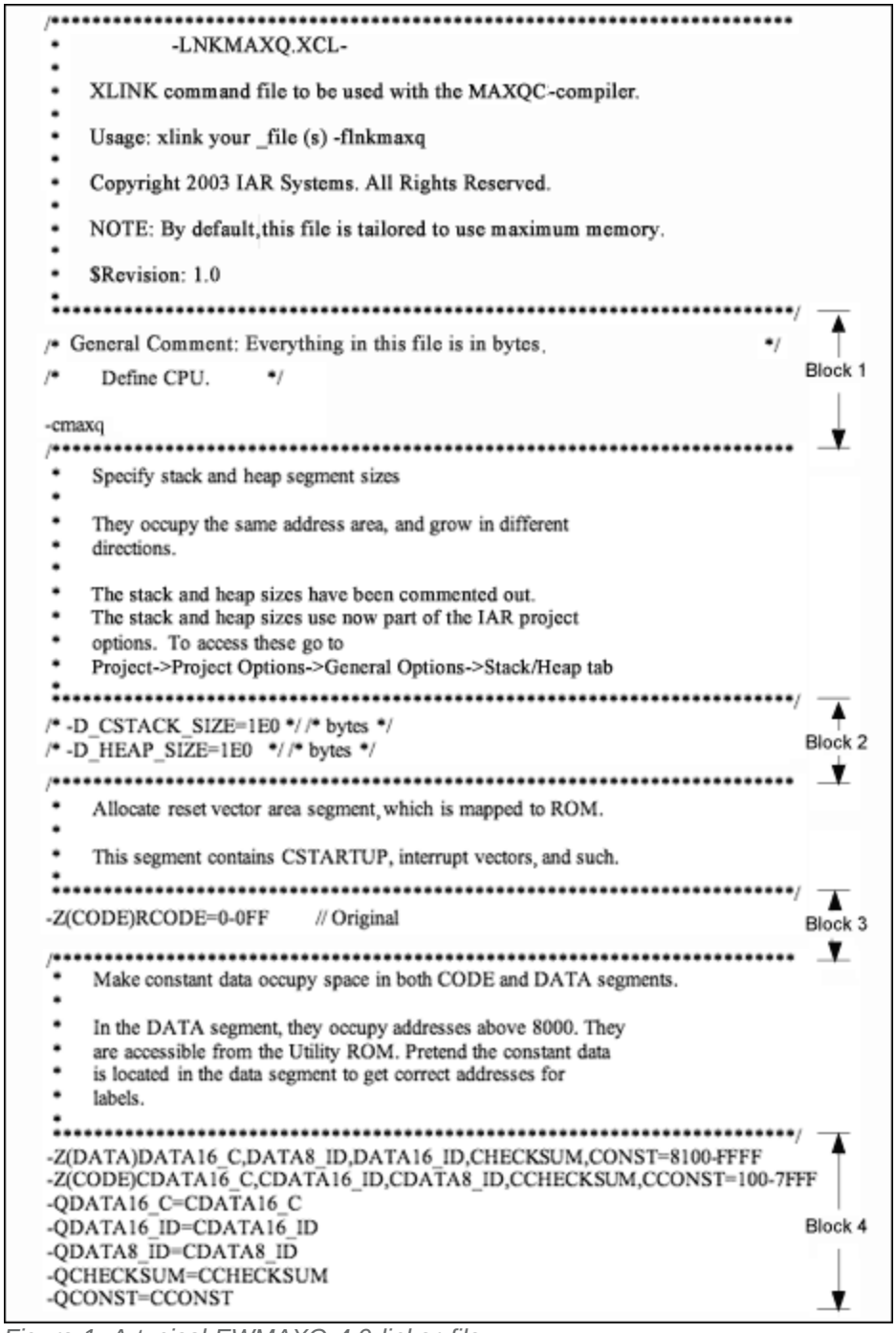

Figure 1. A typical EWMAXQ 4.0 linker file.

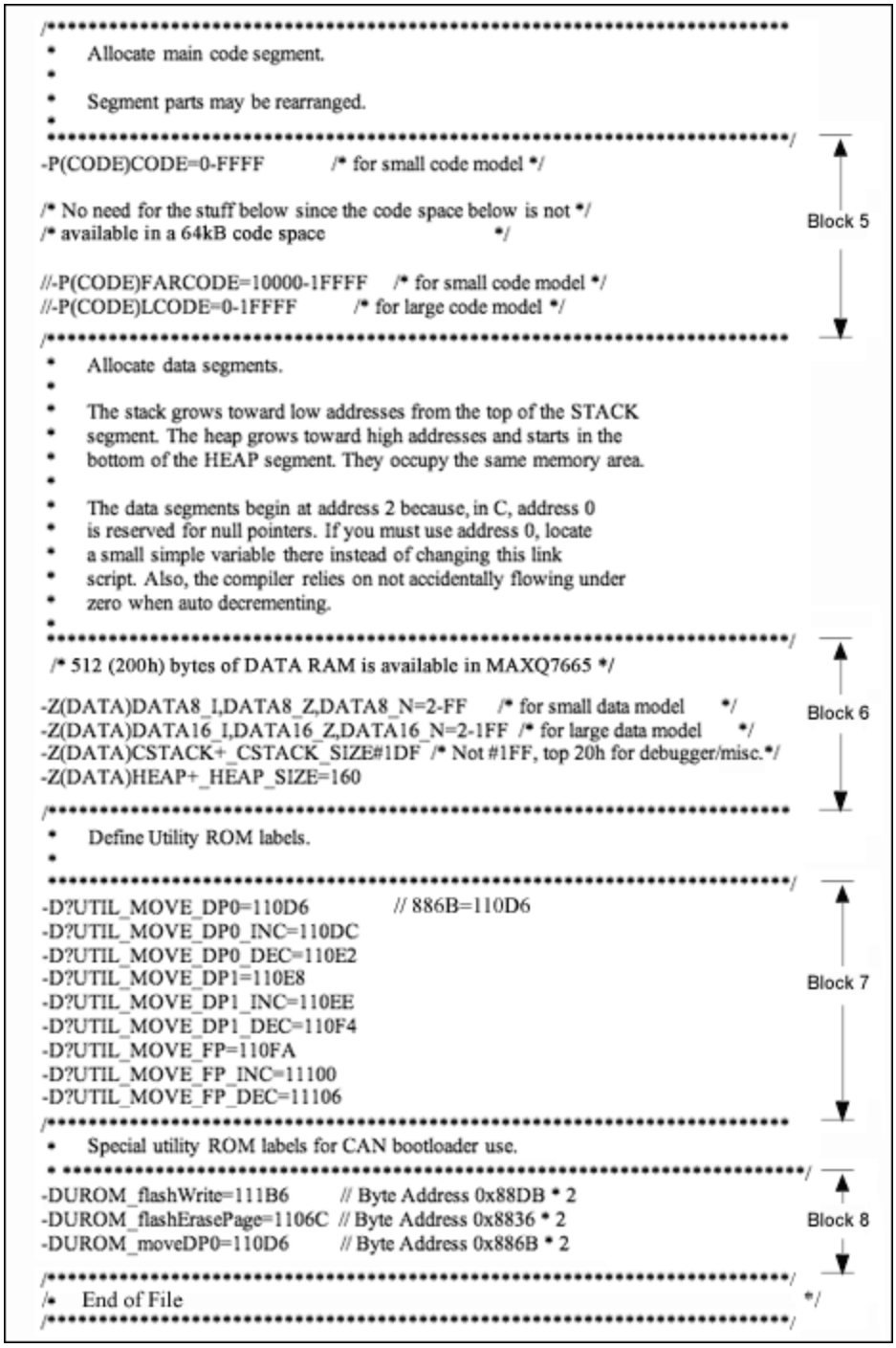

Figure 2. Another typical linker file.

Block 1

```
/*      Define CPU.        */

-cmaxq
```

This block defines the type of the target processor. The `-cmaxq` specifies that it is a MAXQ processor. **Note: the user should not change this code for a MAXQ processor.**

Block 2

```
/*****************************************************************************
 *    Specify stack and heap segment sizes
 *
 *    They occupy the same address area, and grow in different
 *    directions.
 *
 *    The stack and heap sizes have been commented out.
 *    The stack and heap sizes are now part of the IAR project
 *    options.  To access these go to
 *    Project->Project Options->General Options->Stack/Heap tab
 *
 *****************************************************************************/
/* -D_CSTACK_SIZE=1E0 */ /* bytes */
/* -D_HEAP_SIZE=1E0   */ /* bytes */
```

This block is relevant for IAR EWMAXQ version 3.0 and lower. For version 4.0 (and higher) this block is commented out, as shown above. In version 4.0 (and higher) the stack and heap sizes are declared in the IAR EWMAXQ Project Options menu, as shown in **Figure 3** below. **Note: the values shown in the dialog box in Figure 3 are in decimal format.**

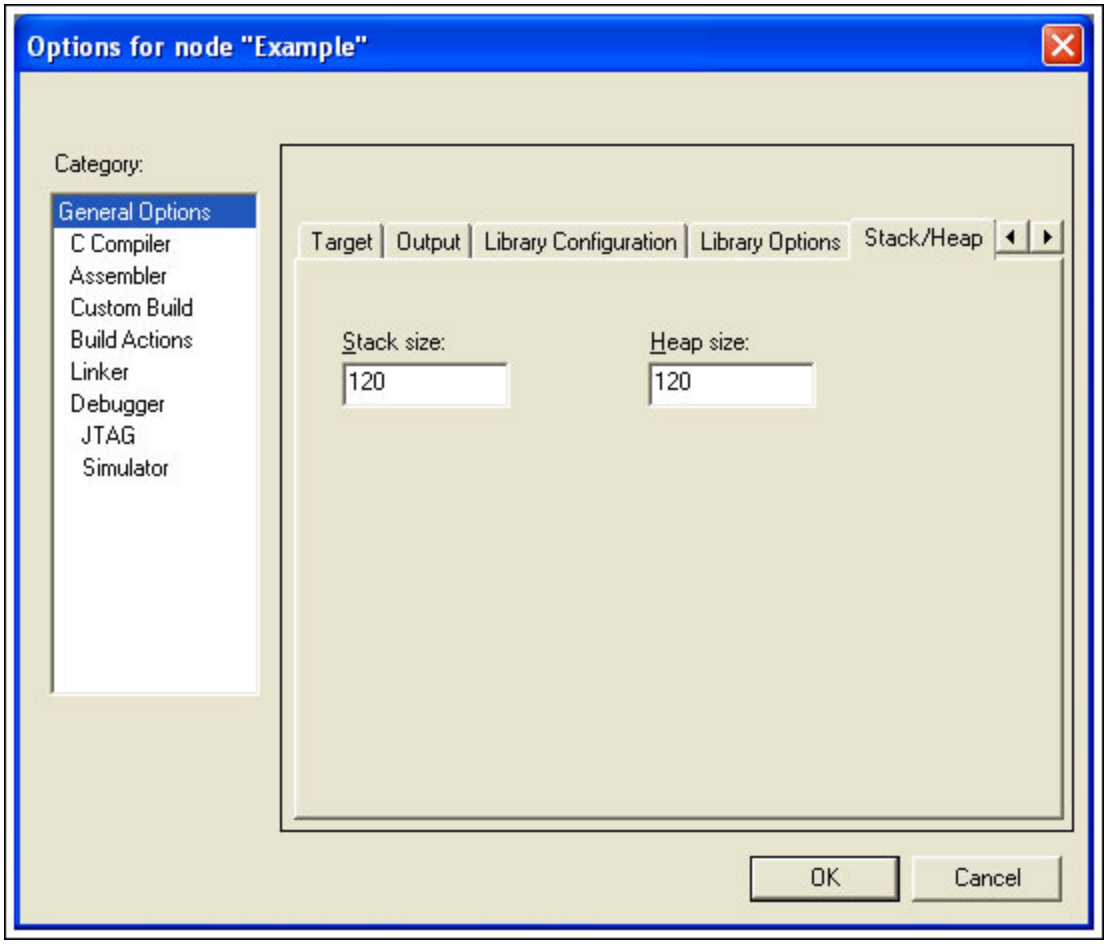

Figure 3. The selection of stack and heap size is made from the IAR EWMAXQ Options menu.

Block 3

```
/*****************************************************************************
*   Allocate reset vector area segment which is mapped to ROM.
*
*   This segment contains CSTARTUP, interrupt vectors and such.
*
*****************************************************************************/
-Z(CODE)RCODE=0-0FF        // Original
```

This block defines the space allocated for `cstartup.s66`, which contains code used during system setup, runtime initialization (cstartup), and system termination (cexit). The `cstartup.s66` is part of the IAR EWMAXQ C compiler, and most users will *not* need to modify this block.

The value 0-0FF indicates the starting and the ending address of the `cstartup.s66` location. This value should *not* be changed for the default `cstartup.s66`.

Block 4

```
/*****************************************************************************
 *    Make constant data occupy space in both CODE and DATA segments.
 *
 *    In the DATA segment, they occupy addresses above 8000. They
 *    are accessible from the Utility ROM. Pretend the constant data
 *    is located in the data segment to get correct addresses for
 *    labels.
 *
 *****************************************************************************/
-Z(DATA)DATA16_C,DATA8_ID,DATA16_ID,CHECKSUM,CONST=8100-FFFF
-Z(CODE)CDATA16_C,CDATA16_ID,CDATA8_ID,CCHECKSUM,CCONST=100-7FFF
-QDATA16_C=CDATA16_C
-QDATA16_ID=CDATA16_ID
-QDATA8_ID=CDATA8_ID
-QCHECKSUM=CCHECKSUM
-QCONST=CCONST
```

This block manages both the location of the constant data in the program space and from where that data is read in the data space. The constants in program space are read with the help of the Utility ROM code, which is transparent to the user. (If the reader is not familiar with the Utility ROM, refer to the *MAXQ User's Guide* and application notes cited at the end of this article).

The IAR EWMAXQ only allows constants in the program segment at 0x0000-0x7FFF bytes (also known as P0) to be read from the data space at 0x8000-0xFFFF. See **Figure 4**.

It is important to note that the flash space shown in Figure 4 is mapped from word mode (16 bits) in the program space to byte mode in the data space (8 bits). The flash is accessed through the Utility ROM functions. In all cases, the values entered in the linker file are in bytes.

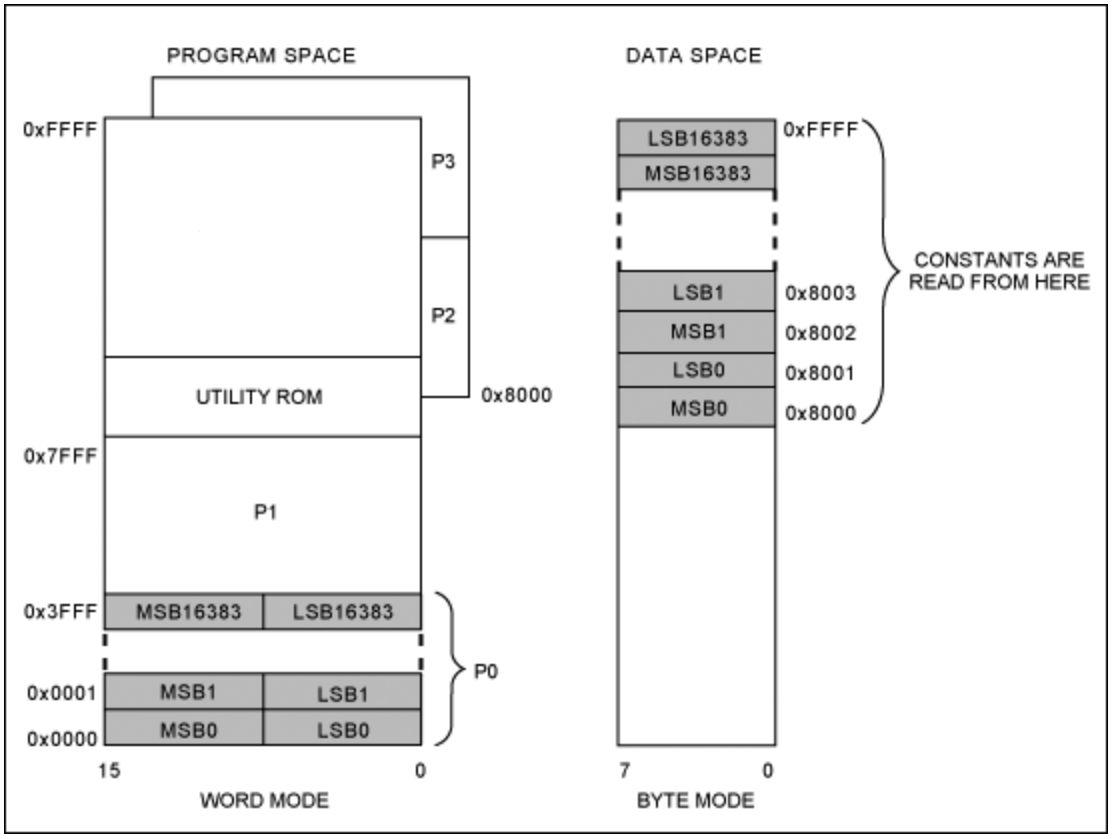

Figure 4. The flash memory is in word mode when in the program space, but is mapped to read in byte mode (as set by the IAR EWMAXQ) when in data space.

Refer now back to the first line in Block 4:

```
-Z(DATA)DATA16_C,DATA8_ID,DATA16_ID,CHECKSUM,CONST=8100-FFFF      /* in bytes
*/
```

This line indicates where the constants stored in program space are accessed from the data space. The value `8100-FFFF` indicates the start and the end of the allowed space. The space starts at 8100h and not 8000h because 0-FF hex is required for `cstartup.s66`, as presented in the Block 1 discussion.

Turn now to the second line in Block 4:

```
-Z(CODE)CDATA16_C,CDATA16_ID,CDATA8_ID,CCHECKSUM,CCONST=100-7FFF    /* in
bytes */
```

This line indicates where the constants actually reside in the program space. (The value being entered is in bytes.) Also remember that 0-FF has been used by `cstartup.s66` as mentioned above and presented in the Block 1 discussion.

Therefore, to summarize for the first and second lines in Block 4, the second line declares where the constant should be placed in the program space and the first line is the location where the data would be read from in the program space. If the constant in code space is in the range of 4100-7FFF, it would get mapped to C100-FFFF.

Turn now to the remaining lines in Block 4:

```
-QDATA16_C=CDATA16_C
-QDATA16_ID=CDATA16_ID
-QDATA8_ID=CDATA8_ID
-QCHECKSUM=CCHECKSUM
-QCONST=CCONST
```

Each of these lines maps one memory segment from data space to its equivalent location in code space. The first line, for example, maps the `DATA16_C` segment (declared in the first line of the block as occupying data space at addresses 8100-FFFF) to the equivalent segment in code space `CDATA16_C` (declared in the second line of the block as occupying code space at addresses 0100-7FFF.)

Block 5

```
/*****************************************************************************
 *    Allocate main code segment.
 *
 *    Segment parts may be rearranged.
 *
 *****************************************************************************/

-P(CODE)CODE=0-FFFF          /* for small code model */

/* No need for the stuff below since the code space below is not */
/* available in a 64kB code space                                */

//-P(CODE)FARCODE=10000-1FFFF    /* for small code model */
//-P(CODE)LCODE=0-1FFFF          /* for large code model */
```

This block declares the space available for code in the program space. In this example, the available space is 10000h bytes. The range of available memory is thus 0-FFFFh bytes.

There are two types of data and code space models: small and large. (Detailed discussion follows.) These spaces are selected in the options window of the IAR EWMAXQ as shown in **Figure 5**.

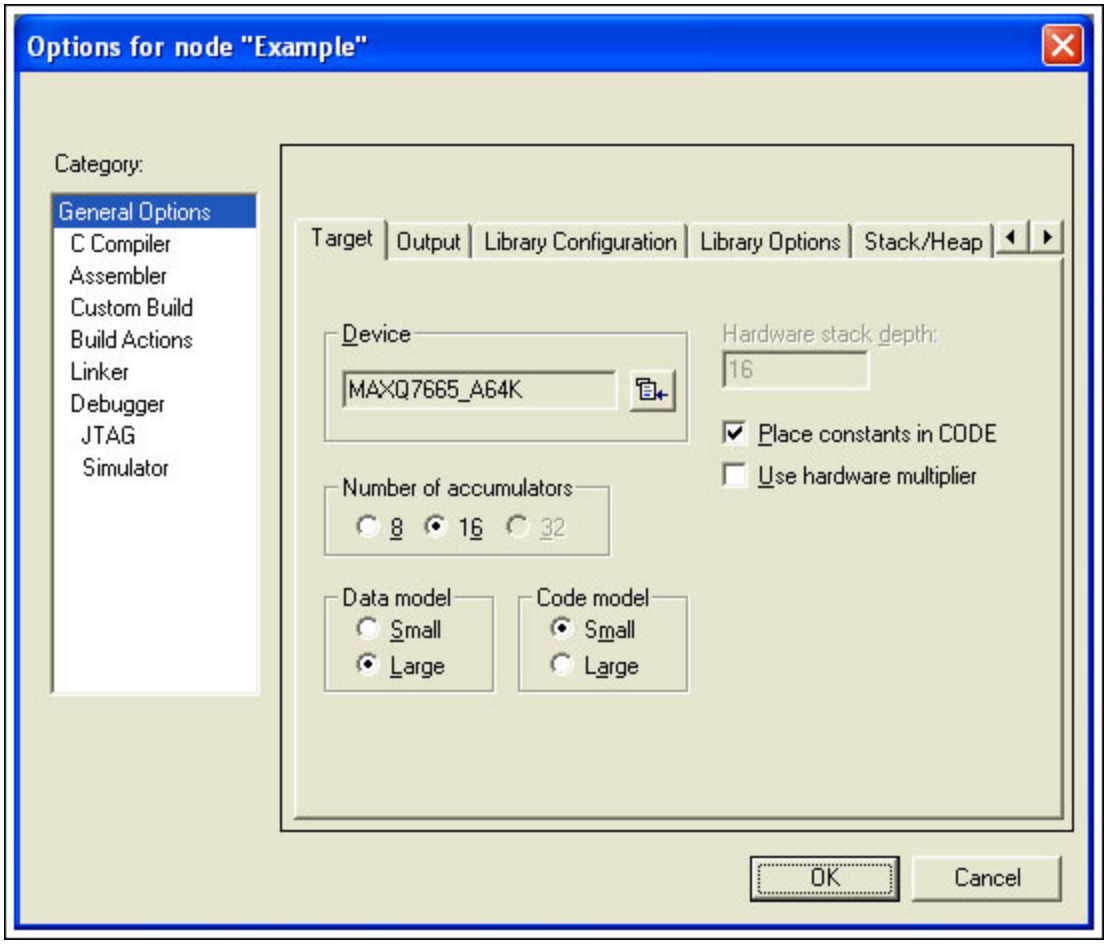

Figure 5. The Code model and Data model are selected from the IAR EWMAXQ Options menu.

Small Code Model

If the Small Code model is selected, then the following two lines in the linker file are relevant:

```
-P(CODE)CODE=0-FFFF              /* for small code model */
-P(CODE)FARCODE=10000-1FFFF      /* for small code model */
```

The code space for the Small model is divided into two spaces: CODE and FARCODE, as shown in the lines above. The 0-FFFFh space is called the CODE space and the 10000h-1FFFFh space is called the FARCODE space.

The Utility ROM resides in the code space above 8000h, so there is little overhead to make the code in the FARCODE space accessible by setting the UPA bit. (Refer to the *MAXQ User's Guide* for an explanation of this bit.) If a 64kB space is used, there is no reason to declare the FARCODE space. If, however, the program space is bigger than 64kB, then both CODE and FARCODE spaces must be declared.

Declaring a space bigger than what is actually available will not cause errors during code build. Therefore, to avoid problems later the user must ensure that the generated code is not bigger than the available space.

In this example, the code model selected is small and the program memory size is 64kB. Consequently, the FARCODE line has been commented out, as it would serve no purpose, and just the CODE

declaration remains.

Large Code Model
If a Large Code model is used (when flash memory is greater than 64kB) then the following instruction applies:

```
-P(CODE)LCODE=0-1FFFF           /* for large code model */
```

The Large Code model is one continuous memory space declared as LCODE.

In the Large Code model, the UPA bit is always 1 and no specific keywords are required. However, the Utility ROM cannot be called directly. Instead, a special stub function carries out this task, which adds some overhead.

The Large Code model is good for application code where the Utility ROM code is rarely used. An example is, when data constants are kept in the data space (RAM) instead of the program space, the Utility ROM is not needed to access data from program memory, as presented in Block 4 discussion above.

In this example, the flash memory size is 64kB, so LCODE serves no purpose and has been commented out. For now, LCODE acts as a placeholder for other devices where this linker file can be used as a template.

Block 6

```
/*****************************************************************************
 *    Allocate data segments.
 *
 *    The stack grows toward low addresses from the top of the STACK
 *    segment. The heap grows toward high addresses and starts in the
 *    bottom of the HEAP segment. They occupy the same memory area.
 *
 *    The data segments begin at address 2 because, in C, address 0
 *    is reserved for null pointers. If you must use address 0, locate
 *    a small simple variable there instead of changing this link
 *    script. Also, the compiler relies on not accidentally flowing under
 *    zero when auto decrementing.
 *
 *****************************************************************************/
 /* 512 (200h) bytes of DATA RAM is available in MAXQ7665 */

-Z(DATA)DATA8_I,DATA8_Z,DATA8_N=2-FF     /* for small data model      */
-Z(DATA)DATA16_I,DATA16_Z,DATA16_N=2-1FF  /* for large data model      */
-Z(DATA)CSTACK+_CSTACK_SIZE#1DF  /* Not #1FF, top 20h for debugger/misc.*/
-Z(DATA)HEAP+_HEAP_SIZE=160
```

This block handles data memory allocation. For this example, the available memory space is 200h bytes occupying the range 0-1FFh.

There are two types of data memory model implementations here: the Small Data model and the Large Data model.

The Small Data Model
In the Small Data memory model, storage is located in the first 256 bytes of the data memory space.

This is the only memory type that can be accessed using 8-bit pointers. The advantages of this memory model are that only eight bits are needed for pointer storage and that accesses are smaller and faster.

```
-Z(DATA)DATA8_I,DATA8_Z,DATA8_N=2-FF      /* for small data model       */
```

Since the Small Data model employs an 8-bit pointer, the range of data addresses is 00-FF (256 locations). The data segments begin at address 2 because, in C, address 0 is reserved for null pointers. Therefore, it is best to leave the lower limit at 2.

The Large Data Model

In the Large Data memory model, storage is located in the entire 64kB of the data memory. This memory can be accessed using 16-bit pointers. More data is available here at the cost of reduced speed.

```
-Z(DATA)DATA16_I,DATA16_Z,DATA16_N=2-1FF  /* for large data model  */
```

Theoretically, like the code space, the upper limit of this Large Data model could be set at FFFFh bytes, since 16 bits of address space are available. It is best, however, to specify in the configuration file only what is available so those who might review the file are not mislead. The example device has 512 bytes of space, so the upper limit is set to 1FFh.

Stack and Heap

The following lines indicate the beginning address of the soft stack and the heap. (However, remember from the discussion of Block 2 that the stack and heap address is declared in the options menu of the IAR.)

```
-Z(DATA)CSTACK+_CSTACK_SIZE#1DF   /* Not #1FF, top 20h for debugger/misc.*/
-Z(DATA)HEAP+_HEAP_SIZE=160
```

The stack grows downwards and the heap grows upwards from their starting location.

The stack and the heap should be placed at the top of the RAM space, because the lower addresses are reserved by the compiler for declared variables. **Figure 6** illustrates how the stack and heap are allocated. The top 32 bytes of RAM are used by the debugger. Therefore, leave the top 32 bytes untouched, subtract 32 from the largest value of the data memory, and use that as the beginning value of the stack. Note that the size of both the stack and the heap are declared in the project options window of the EWMAXQ, as discussed earlier for BLOCK 2.

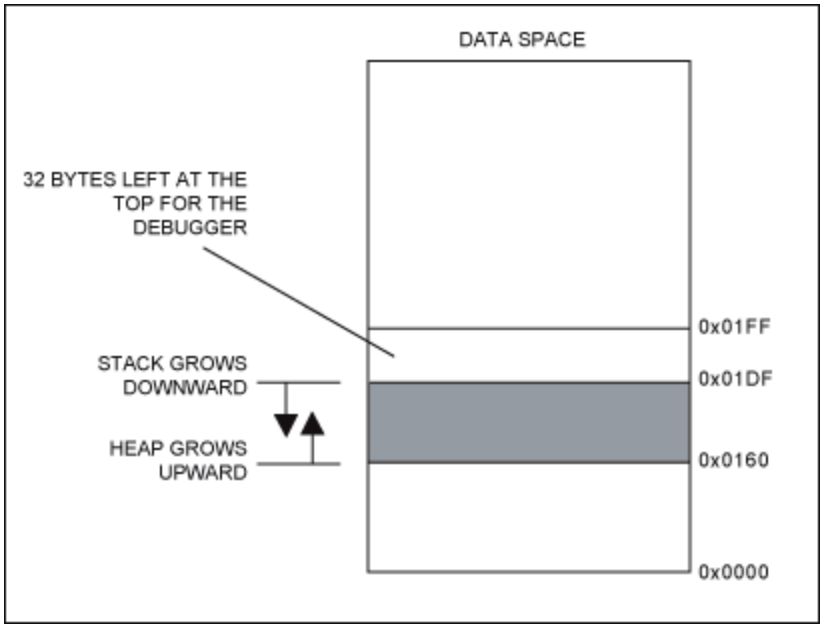

Figure 6. Illustration of the stack and heap placement in the data space.

In our example where the MAXQ has 512 bytes of RAM in data space, the stack starts at 1DF to account for the debugger space and the heap is declared at 160 (hex).

Block 7

```
/*****************************************************************************
 *    Define Utility ROM labels.
 *
 *****************************************************************************/

-D?UTIL_MOVE_DP0=110D6
-D?UTIL_MOVE_DP0_INC=110DC
-D?UTIL_MOVE_DP0_DEC=110E2
-D?UTIL_MOVE_DP1=110E8
-D?UTIL_MOVE_DP1_INC=110EE
-D?UTIL_MOVE_DP1_DEC=110F4
-D?UTIL_MOVE_FP=110FA
-D?UTIL_MOVE_FP_INC=11100
-D?UTIL_MOVE_FP_DEC=11106
```

This block contains the addresses of some of the Utility ROM functions that the compiler uses. These memory addresses can be found in the specific *User's Guide* of a MAXQ microcontroller and can be used by the programmer.

Block 8

```
/*************************************************************************
 *    Special Utility ROM labels for CAN bootloader use.
 * ************************************************************************/

-DUROM_flashWrite=111B6        // Byte Address 0x88DB * 2
-DUROM_flashErasePage=1106C  // Byte Address 0x8836 * 2
-DUROM_moveDP0=110D6          // Byte Address 0x886B * 2
```

This is a user-created block. If a specific Utility ROM function is not available in the default linker file (as shown in Block 7), then the user can declare that function here. The user should keep in mind that these values are in bytes.

An Application Example

The linker file discussed in this application note will be used as a template to create a new linker file for the MAXQ2000 microcontroller as shown in **Figures 7** and **8** below. The MAXQ2000 processor has 1K x 16 RAM (0-7FFh bytes) and 32K x 16Flash (0-FFFFh bytes). Every code item that is modified in the example linker file of Figure 1 is highlighted in blue.

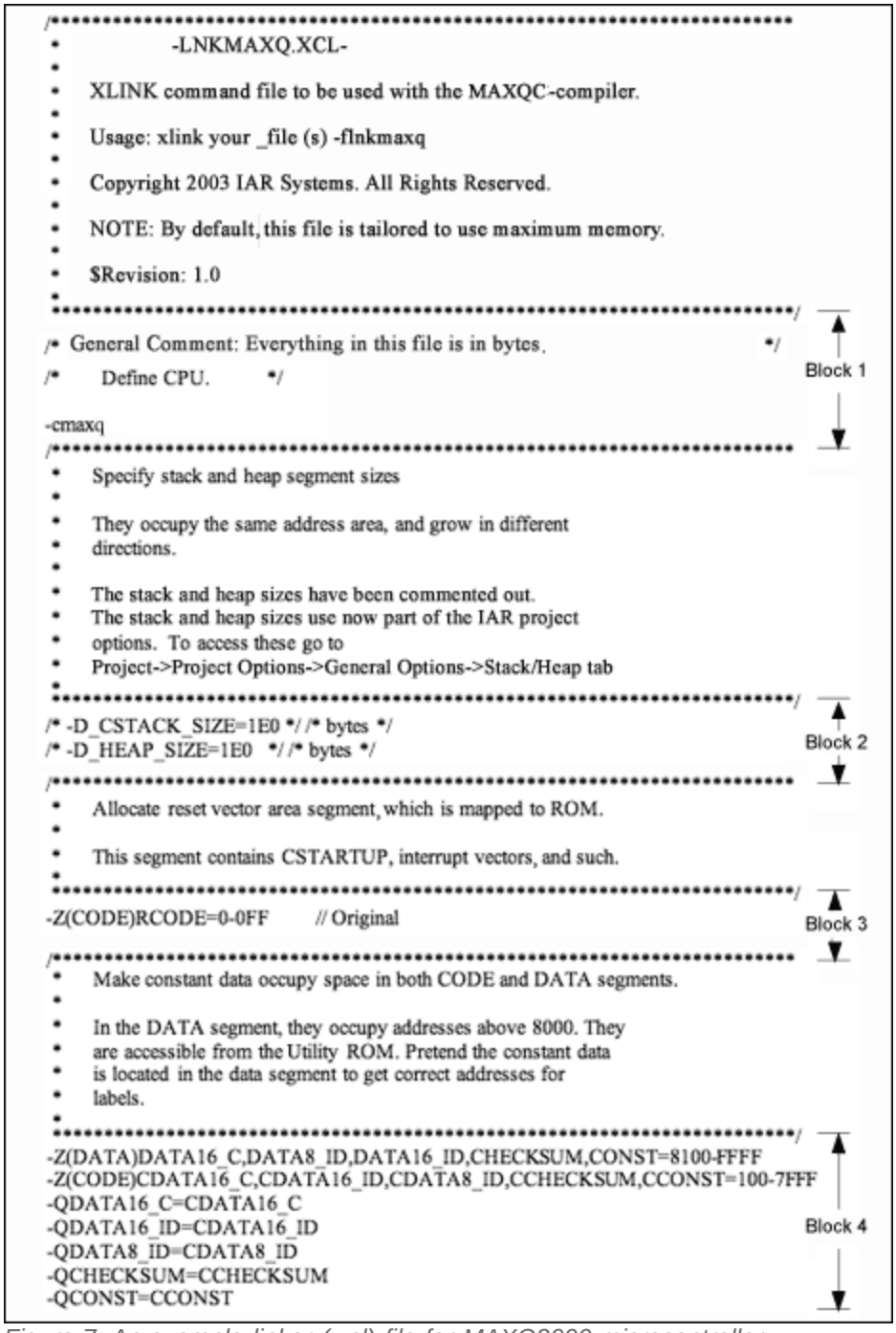

Figure 7. An example linker (.xcl) file for MAXQ2000 microcontroller.

```
/*****************************************************************************
*   Allocate main code segment.
*
*   Segment parts may be rearranged.
*
*****************************************************************************/
-P(CODE)CODE=0-FFFF          /* for small code model */

/* No need for the stuff below since the code space below is not */
/* available in a 64kB code space                                */
//-P(CODE)FARCODE=10000-1FFFF    /* for small code model */
//-P(CODE)LCODE=0-1FFFF          /* for large code model */
/*****************************************************************************
*   Allocate data segments.
*
*   The stack grows toward low addresses from the top of the STACK
*   segment. The heap grows towards high addresses and starts in the
*   bottom of the HEAP segment. They occupy the same memory area.
*
*   The data segments begin at address 2 because, in C, address 0
*   is reserved for null pointers. If you must use address 0, locate
*   a small simple variable there instead of changing this link
*   script. Also, the compiler relies on not accidentally flowing under
*   zero when auto decrementing.
*
*****************************************************************************/
 /* 2kB (800h) of DATA RAM is available in MAXQ2000 */

-Z(DATA)DATA8_I,DATA8_Z,DATA8_N=2-FF      /* for small data model        */
-Z(DATA)DATA16_I,DATA16_Z,DATA16_N=2-7FF  /* for large data model        */
-Z(DATA)CSTACK +_CSTACK_SIZE #7DF  /* Not #7FF, top 20h for debugger /misc */
-Z(DATA)HEAP +_HEAP_SIZE =760
/*****************************************************************************
*   Define utility ROM labels.
*
*****************************************************************************/
-D?UTIL_MOVE _DP0= 1090E
-D?UTIL_MOVE _DP0_INC =10914
-D?UTIL_MOVE _DP0_DEC =1091A
-D?UTIL_MOVE _DP1= 10920
-D?UTIL_MOVE _DP1_INC =10926
-D?UTIL_MOVE _DP1_DEC =1092C
-D?UTIL_MOVE _FP=10932
-D?UTIL_MOVE _FP_INC=10938
-D?UTIL_MOVE _FP_DEC=1093E
/*****************************************************************************
*   Special Utility ROM labels for miscellaneous use.
*
*****************************************************************************/
-DUROM_flashWrite =108C2    // Byte Address  0x8461 * 2
-DUROM_flashErasePage  =108CE // Byte Address  0x8467 * 2
-DUROM_moveDP 0=1090E       // Byte Address  0x8478 * 2
/*****************************************************************************
*   End of File
*****************************************************************************/
```

Block 5

Block 6

Block 7

Block 8

Figure 8. The continuation of the example linker (.xcl) file for MAXQ2000.

Conclusion

The linker file in the IAR EWMAXQ 4.0 IDE is the only personality file that the user will need to change for their application. A typical linker file exists and can be used as a template for simple modifications. A curious reader can always enhance his understanding of the linker file by reading the supplementary references listed in this article.

Relevant Links
MAXQ User Guide
Application Note 3222: *Introduction to MAXQ Architecture*
Application Note 3960: *Unlocking the secrets of the MAXQ*
Application Note 3384: *Accessing the functions provided in the MAXQ Utility ROM*

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

---

Application Note 4128: http://www.maximintegrated.com/an4128
APPLICATION NOTE 4128, AN4128, AN 4128, APP4128, Appnote4128, Appnote 4128