Application Note

AN2213/D
11/2001

Using Cosmic Software's M68HC12 Compiler for MC9S12DP256 Software Development

By: Gordon Doughman
Freescale Field Applications Engineer

Introduction

Cosmic Software's C compiler suite provides a competent set of tools and utilities for MC9S12DP256 software development. Although Cosmic's documentation adequately covers the installation and general use of their tools, it doesn't cover the specifics of compilation and linking of code modules for execution in the paged memory environment of the MC9S12DP256. While this application note is not a substitute for the documentation provided with the Cosmic tool set, it provides the additional details necessary to compile, link, and generate an S-record object code file that can be executed on the MC9S12DP256 using Cosmic's compiler tools for the M68HC12 Family of Motorola microcontrollers (MCU). Be sure to read the Cosmic documentation to gain an understanding of the compiler, linker, and assembler features.

NOTE: *M68HC12 Family is used in this document to represent both the M68HC12 and HCS12 Families of products.*

This document makes specific reference to the MC9S12DP256, but its concepts on paging and compiler directives can be applied to all HCS12 Family members with paged memory.

The MC9S12DP256 Memory Map

Because an M68HC12 Family device is a 16-bit microcontroller with a 16-bit program counter, it cannot directly address a total of more than 64K bytes of memory. To enable the M68HC12 Family to address more than 64K bytes of program memory, a paging mechanism was designed into the architecture. Access to program memory beyond the 64K limit is provided through a 16K byte window located from $8000 through $BFFF. An 8-bit paging register, called the PPAGE register, provides access to a maximum of 256 16K byte pages, or 4M bytes of program memory.

In addition to the hardware paging mechanism, the instruction set has two instructions that allow inter-page function (subroutine) calls:

- The CALL instruction is similar in function to the JSR instruction, however, in addition to placing the PPAGE window return address on the stack, it also places the value of the PPAGE register on the stack before writing the 8-bit value supplied by the CALL instruction to the PPAGE register.
- The RTC instruction is similar to the RTS instruction except that it is used to terminate functions called by the CALL instruction.

Both the PPAGE register value and the PPAGE window address are restored from the stack, continuing execution at the next instruction after the call.

The MC9S12DP256 implements 6 bits of the PPAGE register which gives it a 1 Mbyte program memory address space that is accessed through the PPAGE window. The lower 768K portion of the address space, accessed with PPAGE values $00 through $2F, is reserved for external memory when the part is operated in expanded mode. The upper 256K of the address space, accessed with PPAGE values $30 through $3F, is occupied by the on-chip FLASH memory as shown in **Figure 1**.

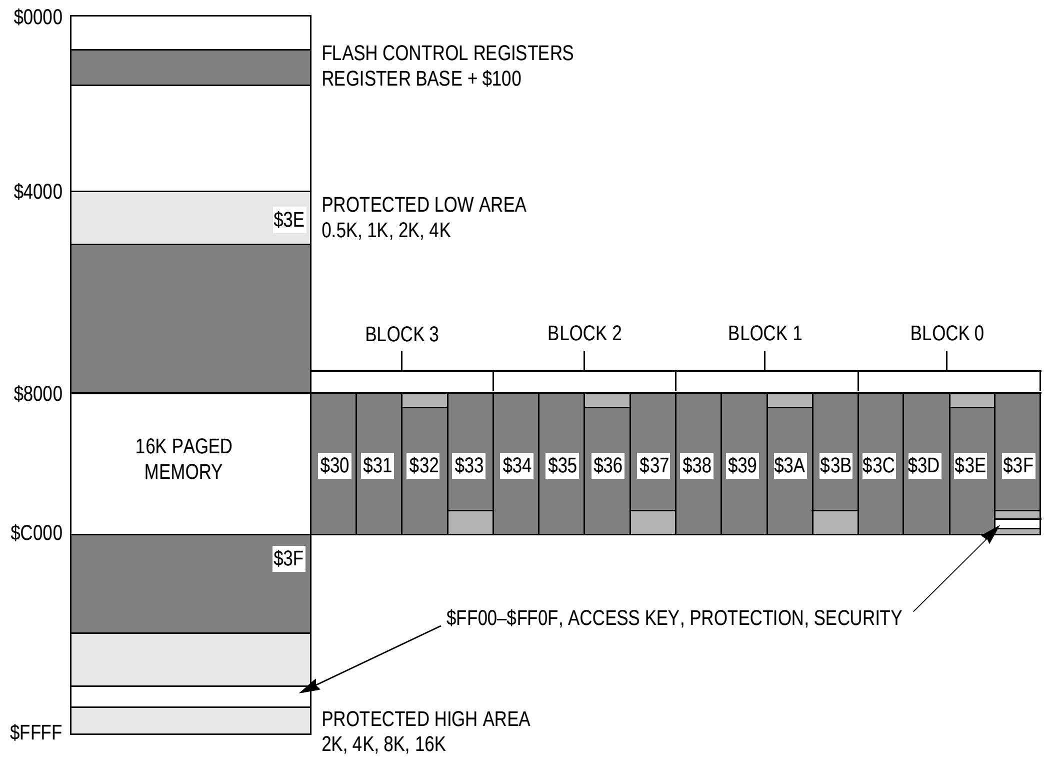

Figure 1. MC9S12DP256 Memory Map

While all 256K of FLASH memory can be accessed through the 16K PPAGE window, two of the 16K byte pages can also be accessed at fixed address locations as shown in **Figure 1**. The fixed page at $4000–$7FFF is the same block of memory that can be accessed through the PPAGE window when the PPAGE register contains $3E. The fixed page from $C000–$FFFF is the same block of memory that can be accessed through the PPAGE window when the PPAGE register contains $3F. These two fixed page areas are provided to overcome some of the restrictions of the M68HC12 memory paging design.

Because of the manner in which the memory paging mechanism is implemented, functions residing in paged memory mapped between $8000 and $BFFF cannot access constant data residing in a different memory page. This restriction is necessary because the PPAGE register would have to be written with a different value in order to access the data. Clearly, writing the PPAGE register with a new value would result in a CPU runaway situation because the code it was executing would disappear as soon as the new value was written. Any constant data such as lookup tables, string or numeric constants that are shared by functions residing on different pages **must** be placed in one of the two fixed FLASH memory pages. In addition, if a pointer to an entry in a table of constant data is returned by a function, the data table must reside in one of the fixed pages if the calling function could reside on a page other than the data table. Also, Cosmic's library routines (Libd.h12, Libf.h12, Libi.h12 and Libm.h12) are written such that they cannot be executed from paged memory, therefore, the library routines must be placed in one of the fixed pages.

Finally, because the reset and interrupt vectors are only 16-bits, all interrupt service routines and the initial reset routine must begin in one of the fixed page memory areas. This does not mean that the entire initialization or interrupt service routines must reside in the fixed memory areas, however, they must begin there. If it is desired to place the bulk of the interrupt service routine or initialization code in paged memory, the portion of the interrupt service routine in the fixed page area could consist of a CALL to the paged functions followed by an RTI instruction.

Declaring Functions For Paged Memory

As mentioned previously, compiled functions are normally called with a JSR or BSR instruction and are terminated with an RTS instruction; however, inter-page functions must be invoked using the CALL instruction and must end with the RTC instruction. To support the inter-page function calling mechanism of the M68HC12 family, Cosmic has provided an extension to the ANSI C standard for function declarations. This extension is used to inform the compiler that a function located in paged memory can be called from a page other than the page in which the function resides. The @**far** type qualifier must be used in both the declaration (prototype) and definition of any functions residing in paged memory and called from a different page. Figure 2 shows both the declaration and definition of a function using the @far type qualifier. Notice that instead of using the @far qualifier directly in the declaration and definition of the function, the word **far** is used instead. With the preprocessor definitions included in the figure, it allows the source code to easily be used in either a paged or non-paged environment.

NOTE: *Intra-page function calls (those functions that are only called within the page in which they reside) do not require the type qualifier. As such, those functions will be called with a JSR or BSR instruction and will end with and RTS making the code smaller and faster.*

```
#ifdef PagedMem
#define far @far
#else
#define far
#endif

ErrorNum far TargetInit (void);      /* This is a prototype */

ErrorNum far TargetInit (void)       /* This is a definition /*

{
/* Code goes here */
}                                    /* end TargetInit */
```

Figure 2. Using the @far Type Qualifier

Placement of Constant Data During Compilation

As mentioned previously, functions residing in paged memory cannot access constant data residing in a different memory page. Any constant data such as lookup tables, string or numeric constants that are shared by functions residing on different pages must be placed in one of the two fixed FLASH memory pages. During the compilation process, by default, the Cosmic compiler places **all** constant data and strings in a section named **.const**. At link time all the constant data can be placed in one of the two fixed pages. For large programs, it is conceivable that constant data will exceed the space provided by the two fixed memory pages. In this case, it is recommended that any constant data within a compile module not referenced from another page be placed in the .**text** section of a compile module. This can be accomplished by using the **+nocst** compiler command line option as shown in Figure 3.

```
cx6812 -1 -e +nocst +debug MonErrors.c
```

Figure 3. Use of the +nocst Compiler Directive

Linking Compiled Code Modules

Before presenting an example linker command file for the MC9S12DP256, a short discussion on memory addresses used by the linker is necessary. Many development tools view the MC9S12DP256's program memory expansion space as a single linear address space. As mentioned previously, of this 1M byte memory space, the lower 768K is reserved for external memory when the part is operated in expanded mode. The upper 256K of the address space is occupied by the on-chip FLASH memory. Linear addresses corresponding to PPAGE window address ranges and various PPAGE values are shown in **Table 1**.

Table 1. Linear Address to PPAGE/Window Address Correspondence

Linear Address Range	PPAGE Value	Window Address Range	Memory Type
\$00000–\$BFFFF	\$00–\$2F	\$8000–\$BFFF	Off-chip memory
\$C0000–\$C3FFF	\$30	\$8000–\$BFFF	On-chip FLASH
\$C4000–\$C7FFF	\$31	\$8000–\$BFFF	On-chip FLASH
\$C8000–\$CBFFF	\$32	\$8000–\$BFFF	On-chip FLASH
\$CC000–\$CFFFF	\$33	\$8000–\$BFFF	On-chip FLASH
\$D0000–\$D3FFF	\$34	\$8000–\$BFFF	On-chip FLASH
\$D4000–\$D7FFF	\$35	\$8000–\$BFFF	On-chip FLASH
\$D8000–\$DBFFF	\$36	\$8000–\$BFFF	On-chip FLASH
\$DC000–\$DFFFF	\$37	\$8000–\$BFFF	On-chip FLASH
\$E0000–\$E3FFF	\$38	\$8000–\$BFFF	On-chip FLASH
\$E4000–\$E7FFF	\$39	\$8000–\$BFFF	On-chip FLASH
\$E8000–\$EBFFF	\$3A	\$8000–\$BFFF	On-chip FLASH
\$EC000–\$EFFFF	\$3B	\$8000–\$BFFF	On-chip FLASH
\$F0000–\$F3FFF	\$3C	\$8000–\$BFFF	On-chip FLASH
\$F4000–\$F7FFF	\$3D	\$8000–\$BFFF	On-chip FLASH
\$F8000–\$FBFFF	\$3E	\$8000–\$BFFF	On-chip FLASH
\$FC000–\$FFFFF	\$3F	\$8000–\$BFFF	On-chip FLASH
\$F8000–\$FBFFF	N/A	\$4000–\$7FFF	On-chip FLASH
\$FC000–\$FFFFF	N/A	\$C000–\$FFFF	On-chip FLASH

NOTE: *The last two entries in the table do not associate linear address ranges with PPAGE window addresses, instead, these entries correspond to the two fixed page memory address ranges. Observe that while the addresses in the window* *address range correspond to the fixed page memory addresses, the addresses in the linear address range column correspond to the same addresses as those for PPAGE $3E and $3F.*

The segment definitions used in a linker command file to control the placement of code and data must use both a linear address and a PPAGE window address to properly locate object code and resolve address references for the paged memory scheme. As shown in the example segment definition in **Figure 4**, the linear or **physical** address for a segment is specified using the -b segment option while the PPAGE window or **logical** address is specified using the -o segment option. The linker uses these addresses to resolve inter-page and intra-page function call and data references.

```
+seg .text -b 0xf8000 -0 0x4000 -n FixPage3e
```

Figure 4. Segment Definition Example

Placement of Variable Data

The segment directive used in the linker command file provides the basic mechanism for the placement of code and constant data in FLASH memory. The numerous options supported by the segment directive supports a wide range of possibilities relating to memory map code and data assignments. However, this section will only explore the basic set of linker directives necessary for the MC9S12DP256. **Figure 5** shows the segment definitions necessary for the placement of initialized global data, uninitialized global data and EEPROM data. The `.data` segment is used to contain initialized global data and begins at the default start address of the MC9S12DP256's on-chip RAM. The -n option is used to assign the output name, iRAM, to this segment. Assigning an output name to the segment allows other linker control directives to reference this particular segment definition. If an application does not contain initialized global data, this segment directive may be omitted from the linker command file.

```
# data segment for initialized data
+seg .data -b 0x1000 -n iRAM -m 0x3000

# data segment for uninitialized data
+seg .bss -a iRAM
+def __sbss=@.bss

# data segment for eeprom data
+seg .eeprom -b 0x0400 -m 0x0c00
```

Figure 5. .data, .bss, and .eeprom Segment Definitions

The `.bss` segment is used to contain uninitialized global data. Rather than assign an absolute start address to the `.bss` segment using the -b option, the -a option instructs the linker to place all uninitialized global data immediately after the output segment named iRAM, which contains the initialized global data. The directive immediately following the .bss segment definition is used to define a symbol and assign a value to it. In this case the symbol `__sbss` is assigned a value equal to the address of the next byte to be placed in the `.bss` segment. In this case because it appears in the linker command file before any object file names, `__bss` will be assigned the beginning address of the `.bss` section. This symbol is used by the crts.s and crtsi.s startup routines to initialize all locations in the `.bss` section to zero.

The third segment definition is only required if global variables have been declared using the @eeprom type qualifier. The -b option is used to assign an address of $0400 to this segment. Because the I/O register block overlaps the lower 1024 bytes of the EEPROM, this address is the first accessible location. If the application moves the EEPROM block to a different base address, making the entire 4096 bytes available, the segment start address and its size (-m option) would need to be changed. If none of the variables in an application declared with the @eeprom type qualifier were initialized when defined (i.e., int @eeprom Velocity = 500;) the -c option should be used to suppress the output of data from this section. This will prevent superfluous data from appearing in S-record files that are created from the resulting linked object file.

Placement of Code and/or Constant Data in the Lower Fixed Page

Segment directives following those for the variable and EEPROM data generally consist of segment directives for various areas of the FLASH memory. The number and type of segment directives and object file name placement will depend on the organization of the firmware and the options used when various files are compiled. If the lower fixed page is used by the application, a segment directive must be used to locate code and/or constant data in that memory area.

Figure 6 shows three different examples of segment directives that can be used to place code and constant data in the lower fixed page.

```
+seg .text -b 0xf8000 -o 0x4000 -m 0x4000
File1.o
File2.o
File3.o
```

```
+seg .text -b 0xf8000 -o 0x4000 -m 0x4000 -n FixPage3e
+seg .const -a FixPage3e
File1.o
File2.o
File3.o
```

```
+seg .const -b 0xf8000 -o 0x4000 -m 0x4000
```

Figure 6. Segment Definition Examples for the Lower Fixed Page

The first example in **Figure 6** defines a single segment that will contain any code and data residing in the `.text` section of object files following the segment definition. This particular segment definition is useful when files have been compiled with the +nocst option which places constant data in the text section with code. **Figure 7(a)** shows a graphical layout of how text and constant data would be placed in memory. This segment definition may also be useful for files compiled without the +nocst option if constant data for the files is placed in a `.const` section located in the upper fixed page. As **Figure 7(b)** shows, only `.text` section code is placed in the lower fixed page segment.

The second example in **Figure 6** uses two segment directives for the placement of code and constant data. The `.text` segment directive is similar to the first example except that it uses the -n option to associate a name with the output of the segment. The .const segment directive defines a segment for all constant data appearing in the three files following the directive. The -a option used in this directive causes the .const sections to start at the end of the named output segment. In this case, as shown in **Figure 7(c)**, the constant data immediately follows the text section code.

The last example in **Figure 6** defines only a constant segment for the lower fixed page. This definition is useful if the lower fixed page is being used exclusively for constant data such as lookup tables, string or numeric constants shared by functions residing on different pages within the PPAGE window. Notice that no file names immediately follow the segment directive. Instead, constant data contained in the .const section of files appearing later in the link command file will be placed in the lower fixed page. The example in **Figure 7(d)** shows the situation where the constant data for File1.o and File2.o is placed in the lower fixed page, but the code in the text section is placed elsewhere.

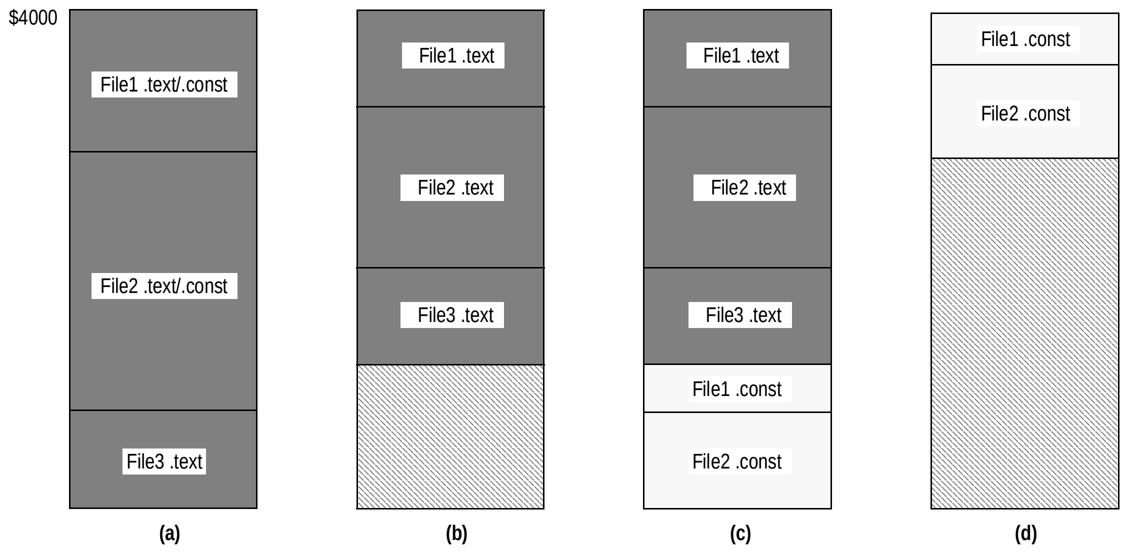

Figure 7. Load Maps for Segment Definition Examples for the Lower Fixed Page

Placement of Code and/or Constant Data Within the PPAGE Window

There are several different methods that can be used for the placement of code and constant data in the FLASH memory visible within the PPAGE window address range. The most straight forward method is to simply use separate segment definitions for each 16K page. Assuming that page $3E is used as the lower fixed page, this method will require 14 separate segment definitions if all of the remaining FLASH pages are utilized. Figure 8 shows an example of `.text` segment definitions for the first and last two PPAGE memory blocks.

```
+seg .text -b 0xc0000 -o 0x8000 -m 0x4000
#
# put files for PPAGE 0x30 here
#
+seg .text -b 0xc4000 -o 0x8000 -m 0x4000
#
# put files for PPAGE 0x31 here
#
+seg .text -b 0xf0000 -o 0x8000 -m 0x4000
#
# put files for PPAGE 0x3c here
#
+seg .text -b 0xf4000 -o 0x8000 -m 0x4000
#
# put files for PPAGE 0x3d here
#
```

Figure 8. PPAGE Memory Window Segment Definitions

Notice that the example does not include any constant segment definitions. As recommended earlier, all compile modules containing constant data not shared by functions on other pages should be contained in the `.text` section by using the +nocst command line option when a file is compiled. Any file containing constant data that must be shared by functions on other pages should be compiled without the +nocst option, creating a `.const` section in the resulting object file. Using a `.const` section definition such as the last example in **Figure 6** would cause all shared constant data to be placed in the lower fixed page.

Using separate segment definitions for each 16K page, object file names are listed after each segment definition until a particular segment becomes full. When a segment becomes full, object file names are listed under the next segment definition until it becomes full. While this method does provide precise control over the placement of code modules in the available memory pages, it forces the developer to manage the arrangement of the code modules to best utilize the space available on each 16K page.

To relieve the developer of the difficulty involved in manually managing the arrangement of the code modules, the linker provides a segment control option that automatically creates a new segment when one becomes full. **Figure 9** shows a segment definition used for automatic bank creation. The -w segment control option is used to set the PPAGE window size and to activate the automatic bank creation mechanism. As shown, the -m option can be used to set the maximum segment size; however, when the automatic bank creation mechanism is activated, it is used to specify the maximum amount of space available for all consecutive banks. In this example, the value following the -m option is obtained by multiplying the PPAGE window size ($4000) by the number of available PPAGE window banks (14).

```
+seg .text -b 0xc0000 -o 0x8000 -w 0x4000 -m 0x38000
#
# put files for PPAGE 0x30–0x3d here
#
```

Figure 9. Automatic PPAGE Segment Creation

As new segments are created, the new segment's physical address (-b option) is obtained by adding the PPAGE window size to the prior bank's physical starting address. In this case, when the bank for PPAGE $30 is filled, the segment physical starting address for PPAGE $31 would be $C4000. The logical starting address for new segments (-o option) is always equal to the address specified in the original segment definition. If the -m option is used as shown in the example, the maximum segment size for newly created segments is obtained by subtracting the PPAGE window size from the prior segments maximum segment size.

While the automatic segment creation feature of the linker relieves the developer from the task of arranging object file names after separate segment definitions, it will not make the most efficient use of the segments it creates without careful ordering of the object file names by the developer. For example, suppose that four object files, File1.o, File2.o, File3.o, and File4.o, contained text sections of approximately 8K, 4k, 7k, and 10k respectively were placed, in order, after the segment definition in **Figure 9**. As the linker reads and processes the listed object files, it simply attempts to combine the text sections of the files in the order in which they appear. Because the combined size of File1.o, File2.o, and File3.o exceeds the PPAGE window size, only File1.o and File2.o are placed in the first segment. Because the combined size of these two files is only 12k, 4k of the FLASH memory at the end of PPAGE $30 would be unused. As the linker continued processing the listed files, it would attempt to combine File3.o and File4.o into a single segment. Because the combined size of these two files exceeds the PPAGE window size, two segments would be created, one for File3.o and one for File4.o. With File3.o placed in PPAGE $31, 9k of the FLASH memory at the end of PPAGE $31 would remain unused. This situation is graphically illustrated in **Figure 10(a)**. If the file names were rearranged, File1.o and File3.o could be combined and placed in the first segment and File4.o and File2.o could be combined and placed in the second segment. This arrangement, illustrated in **Figure 10(b)**, provides for much better utilization of the paged FLASH memory. While the file name rearrangement in this example is rather obvious, ordered file name management of projects containing hundreds of files would become extremely difficult.

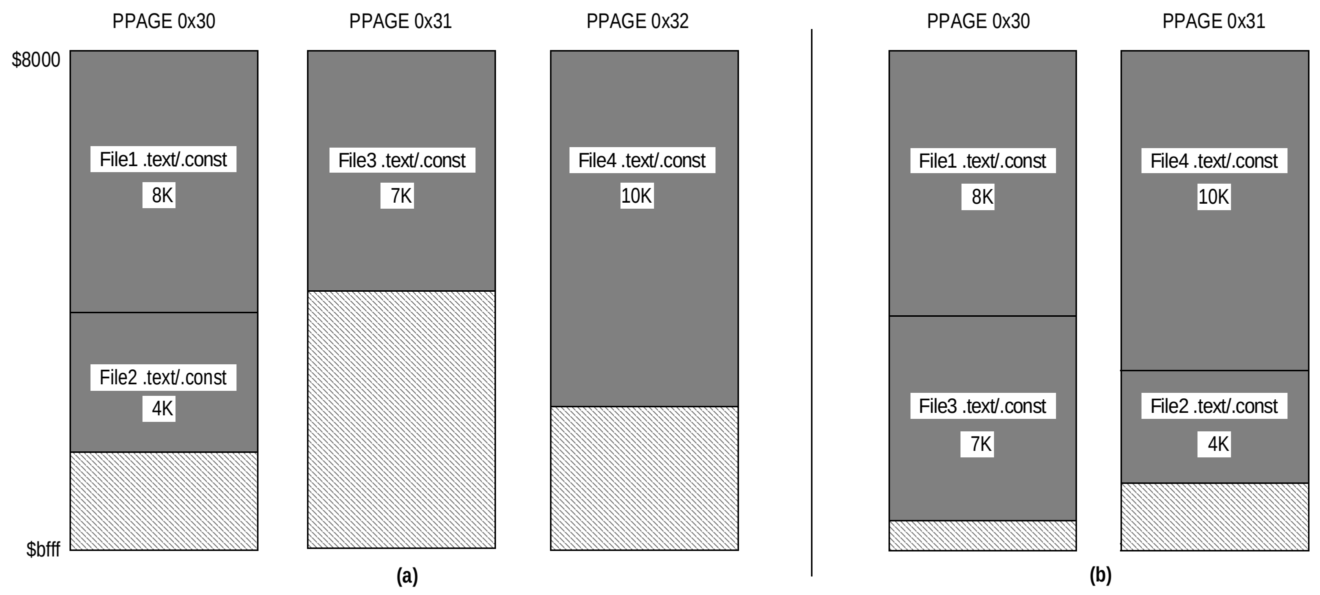

Figure 10. File Order Effect of Automatic PPAGE Segment Filling

Looking at the potential for underutilization of the paged FLASH memory using the automatic segment creation capability of the linker, it does not appear that this feature presents any great advantage over manual segment creation. Fortunately, Cosmic includes a utility, **cbank**, as part of their compiler package that will order a list of object file names so as to fill each segment as efficiently as possible. Using the cbank utility in conjunction with the automatic segment creation capability of the linker provides a mechanism to efficiently utilize the paged FLASH memory with very little effort on the part of the developer.

When using the cbank utility, the names of the object files to be positioned in paged memory are placed in a separate text file rather than in the linker command file. As shown in the command line example in **Figure 11**, cbank will read the object file names in the input file, ObjList.txt, reorder the list of names and write the result to an output file named BankList.txt. The -w command line option sets the page size and the -m option sets the maximum number of pages or banks. Once cbank has created its output file, the resulting list of filenames can be included in the link file using the **+inc** linker directive. **Figure 12** shows an example of automatic segment creation using the +inc directive to include the file name list created by the cbank utility.

```
cbank -m 14 -w 0x4000 -o BankList.txt ObjList.txt
```

Figure 11. cbank Command Line

```
+seg .text -b 0xc0000 -o 0x8000 -w 0x4000 -m 0x38000
#
+inc BankList.txt
#
```

Figure 12. Using Automatic PPAGE Segment Creation with the +inc Directive

Even with the automatic bank packing capabilities of the cbank utility, efficient utilization of paged memory space requires that compile modules be kept as small as possible. Architecting a firmware design to utilize a large number of small compile modules rather than a small number of large compile modules will help the cbank utility to arrange object files to most efficiently utilize the paged memory space.

The combination of automatic PPAGE segment creation, the cbank utility and the linker’s +inc directive will solve the problem of efficiently utilizing paged FLASH memory in most cases. Yet, there may be occasions where a combination of manual segment creation and automatic segment creation can be used to efficiently group files in paged memory. As an example, consider the situation where a number of compile modules share tables of constant data.

One solution, as previously explained, would be to place the constant data on one of the two fixed pages where it could be accessed by all functions no matter where in the paged memory the functions resided. However, because of the limited amount of fixed page memory, it could easily become filled with constant data. As an alternative to placing the constant data used by a group of functions in one of the fixed pages, the functions and the constant data could be placed in one of the paged memory banks using an explicit segment definition. An example is shown in **Figure 13**.

NOTE: *The linear or physical address (-b) definition for the automatic PPAGE segment creation and the total paged memory size (-m) were changed to compensate for the explicitly declared segment definition for PPAGE $30.*

```
+seg .text -b 0xc0000 -o 0x8000 -m 0x4000 -n PPAGE0x30
+seg .const -a PPAGE0x30
File1.o
File2.0
File3.o
+seg .text -b 0xc4000 -o 0x8000 -w 0x4000 -m 0x34000
#
+inc BankList.txt
#
```

Figure 13. Using Manual and Automatic PPAGE Segment Creation

Placement of Code and/or Constant Data in the Upper Fixed Page

Like the lower fixed page of FLASH memory, the upper fixed page of FLASH can be used to contain code, constant data, Cosmic's library routines, or interrupt service routines. In addition to these elements, the upper fixed page of FLASH contains two areas that must have data placed at specific addresses. The first piece of data, beginning at $FF00, is a 16 byte memory area related to the microcontroller's memory security and protection mechanism. The second area is the interrupt and reset vector table. Before discussing the placement of these two pieces of data, it is important to understand the function of the memory security and protection features.

FLASH Memory Protection

The protected areas of each FLASH block are controlled by four bytes of FLASH memory residing in the fixed page memory area from $FF0A–$FF0D During the microcontroller reset sequence, each of the four banked FLASH protection registers (FPROT) is loaded from values programmed into these memory locations. As shown in **Table 2**, location $FF0A controls protection for block three, $FF0B controls protection for block two, $FF0C controls protection for block one and $FF0D controls protection for block zero. The values loaded into each FPROT register determine whether the entire block or just subsections are protected from being accidentally erased or programmed.

Table 2. FLASH Protection and Security Memory Locations

Address	Description
$FF00–$FF07	Security backdoor comparison key
$FF08–$FF09	Reserved
$FF0A	Protection byte for FLASH block 3
$FF0B	Protection byte for FLASH block 2
$FF0C	Protection byte for FLASH block 1
$FF0D	Protection byte for FLASH block 0
$FF0E	Reserved
$FF0F	Security byte

As mentioned previously, each 64K block can have two protected areas. One of these areas, known as the lower protected block, grows from the middle of the 64K block upward. The other, known as the upper protected block, grows from the top of the 64K block downward. In general, the upper protected area of FLASH block zero is used to hold bootloader code since it contains the reset and interrupt vectors. The lower protected area of block zero and the protected areas of the other FLASH blocks can be used for critical parameters that would not change when program firmware was updated.

The FPOPEN bit in each FPROT register determines whether the the entire FLASH block or subsections of it can be programmed or erased. When the FPOPEN bit is erased (1) the remainder of the bits in the register determine the state of protection and the size of each protected block. In its programmed state (0) the entire FLASH block is protected and the state of the remaining bits within the FPROT register is irrelevant.

The FPHDIS and FPLDIS bits determine the protection state of the upper and lower areas within each 64K block respectively. The erased state of these bits allows erasure and programming of the two protected areas and renders the state of the FPHS[1:0] and FPLS[1:0] bits immaterial. When either of these bits is programmed, the FPHS[1:0] and FPLS[1:0] bits determine the size of the upper and lower protected areas. Table 3 summarizes the combinations of the FPHS[1:0] and FPLS[1:0] bits and the size of the protected area selected by each.

Table 3. FLASH Protection Select Bits

FPHS[1:0]	Protected Size
0:0	2K
0:1	4K
1:0	8K
1:1	16K

FPLS[1:0]	Protected Size
0:0	512 bytes
0:1	1K
1:0	2K
1:1	4K

The FLASH protection registers are loaded during the reset sequence from address $FF0D for FLASH block 0, $FF0C for FLASH block 1, $FF0B for FLASH block 2 and $FF0A for FLASH block 3. This is indicated by the “F” in the reset row of the register diagram in the MC9S12DP256 data book. This register determines whether a whole block or subsections of a block are protected against accidental program or erase. Each FLASH block can have two protected areas, one starting from relative address $8000 (called lower) towards higher addresses and the other growing downwards from $FFFF (called higher). While the later is mainly targeted to hold the boot loader code since it covers the vector space (FLASH 0), the other area may be used to keep critical parameters. Trying to alter any of the protected areas will result in a protect violation error and bit PVIOL will be set in the FLASH status register (FSTAT).

NOTE: *A mass or bulk erase of the full 64K byte block is only possible when the FPLDIS and FPHDIS bits are in the erased state (= ‘1’).*

FLASH Security

The security of a microcontroller’s program and data memories has long been a concern of companies for one main reason. Because of the considerable time and money that is invested in the development of proprietary algorithms and firmware, it is extremely desirable to keep the firmware and associated data from prying eyes. This was an especially difficult problem for earlier M68HC12 Family members as the background debug module (BDM) interface provided easy, uninhibited access to the FLASH and EEPROM contents using a two wire connection. Later revisions of the original ‘D’ Family parts provided a method that allowed a customer’s firmware to disable the BDM interface (BDM lockout) once the part had been placed in the circuit and programmed. While this prevents the FLASH and EEPROM from being easily accessed in-circuit, it does not prevent a ‘D’ Family part from being removed from the circuit and placed in expanded mode so the FLASH and EEPROM can be read.

The security features of the MC9S12DP256 has been greatly enhanced. While no security feature can be 100% guaranteed to prevent access to an MCU’s internal resources, the MC9S12DP256’s security mechanism makes it extremely difficult to access the FLASH or EEPROM contents. Once the security mechanism has been enabled, access to the FLASH and EEPROM either through the BDM or the expanded bus is inhibited. Gaining access to either of these resources may only be accomplished by erasing the contents of the FLASH and EEPROM or through a built in back door mechanism. While having a back door mechanism may seem to be a weakness of the security mechanism, the target application must specifically support this feature for it to operate.

Erasing the FLASH or EEPROM can be accomplished using one of two methods:

1. The first method requires resetting the target MCU in special single-chip mode and using the BDM interface. When a secured device is reset in special single-chip mode, a special BDM security ROM becomes active. The program in this small ROM performs a blank check of the FLASH and EEPROM memories. If both memory spaces are erased, the BDM firmware temporarily disables device security, allowing full BDM functionally. However, if the FLASH or EEPROM are not blank, security remains active and only the BDM hardware commands remain functional. In this mode the BDM commands are restricted to reading and writing the I/O register space. Because all other BDM commands and on-chip resources are disabled, the contents of the FLASH and EEPROM remain protected. This functionality is adequate to manipulate the FLASH and EEPROM control registers to erase their contents.

CAUTION: *Use of the BDM interface to erase the FLASH and EEPROM memories is not present in the initial mask set (0K36N) of the MC9S12DP256. Great care must be exercised to ensure that the microcontroller is not programmed in a secure state unless the back door mechanism is supported by the target firmware.*

2. The second method requires the microcontroller to be connected to external memory devices and reset in expanded mode where a program can be executed from the external memory to erase the FLASH and EEPROM. This method may be preferred before parts are placed in a target system.

As shown in **Table 4** the security mechanism is controlled by the two least significant bits in the security byte. Because the only unsecured combination is when SEC1 has a value of ‘1’ and SEC0 has a value of ‘0’, the microcontroller will remain secured even after the FLASH and EEPROM are erased since the erased state of the security byte is $FF. As previously explained, even though the device is secured after being erased, the part may be reset in special single-chip mode allowing manipulation of the microcontroller via the BDM interface. However, after erasing the FLASH and EEPROM, the microcontroller can be placed in the unsecured state by programming the security byte with a value of $FE.

NOTE: *Because the FLASH must be programmed an aligned word at a time and because the security byte resides at an odd address ($FF0F), the word at $FF0E must be programmed with a value of $FFFE.*

Table 4. Security Bits

SEC[1:0]	Security State
0:0	Secured
0:1	Secured
1:0	Unsecured
1:1	Secured

Placement of Memory Security and Protection Data

Even if the memory security and protection features are not being utilized during development, a file containing data for this 16 byte area should be created, compiled, and inserted into the linker file for compatibility with some FLASH programming tools. Because of the inability to erase the FLASH and EEPROM using the BDM interface in the first mask set (0K36N) of the MC9S12DP256, many programming tools automatically program the security byte with a value of $FE after successfully erasing the FLASH. This prevents the device from accidentally being placed in a secure state if a programming operation were to fail. Having this block of data included in the object file with a value of $FE for the security byte will ensure that a verify operation will be performed properly.

The contents for the 16 byte memory area is shown in the C source listing in **Figure 14**. The values for each of the constants will vary depending on the memory security and protection features used by an application. However, especially during development, the security byte, `Sec`, should be $FE. This is the only value of the lower two bits in the security byte in which the part remains unsecured. Figure 15 shows an example segment definition for the placement of the memory security and protection data.

NOTE: *If the source file containing the security and protection data is compiled with the +nocst option, the segment definition must be changed to a `.text` segment.*

```
typedef uint unsigned int;
typedef uchar unsigned char;

const uint  BDKey1  = 0xffff; /* Backdoor Key word 1 */
const uint  BDKey2  = 0xffff; /* Backdoor Key word 2 */
const uint  BDKey3  = 0xffff; /* Backdoor Key word 3 */
const uint  BDKey4  = 0xffff; /* Backdoor Key word 4 */
const uchar Res08   = 0xff;   /* reserved */
const uchar Res09   = 0xff;   /* reserved */
const uchar BlkPrt3 = 0xff;   /* Protection byte for Flash block 3 */
const uchar BlkPrt2 = 0xff;   /* Protection byte for Flash block 2 */
const uchar BlkPrt1 = 0xff;   /* Protection byte for Flash block 1 */
const uchar BlkPrt0 = 0xff;   /* Protection byte for Flash block 0 */
const uchar Res0e   = 0xff;   */ reserved */
const uchar Sec     = 0xfe;   /* Security byte */
```

Figure 14. Memory Security and Protection Constant Values

```
+seg .const -b 0xfff00 -o 0xff00 -m 0x10
#
Security.o
#
```

Figure 15. Segment Definition for Security and Protection Data

Placement of Reset and Interrupt Vector Data

The reset and interrupt vector table for all M68HC12 Family devices consists of a 128 byte memory area that begins at $FF80. Because each vector occupies two bytes, a total of 64 unique vectors are supported. The MC9S12DP256 implements 58 of the 64 vectors beginning at $FF8C. The address constants for the reset and interrupt vectors may be generated using either a C or assembly language source file as shown in Figure 16 and Figure 17. Both of these examples show only the first and last two entries in the interrupt vector table. The remaining 54 interrupt service routine and reset vectors names would need to be added to these examples.

```
extern void PWMShutDnISR(void);
extern void PortPISR(void);
extern void ClkMonReset(void);
extern void _stext(void);

void (* const vector[])(void) =
     {
      PWMShutDnIsr,            /* PWM Shutdown ISR Vector */
      PortPISR,                /* Port P ISR Vector */
      .
      .
      .
      ClkMonReset,             /* Clock Monitor Reset Vector */
      _stext                   /* beginning of startup code */
     };
```

Figure 16. C Interrupt Vector Example

```
        switch    .const
;
        xref      _PWMShutDnISR, _Port PISR, _ClkMonReset, __stext
;
        dc.w      _PWMShutDnISR     ; PWM Shutdown ISR Vector.
        dc.w      _Port PISR        ; Port P ISR Vector.
        .
        .
        .
        dc.w      _ClkMonReset      ; Clock Monitor Reset Vector.
        dc.w      __stext           ; beginning of startup code.
;
```

Figure 17. Assembly Language Interrupt Vector Example

In the assembly language example, **Figure 17**, notice that the interrupt service routines and the clock monitor reset vector names are prefixed with an underscore character. This is necessary when any C function is referenced in an assembly language file. The reset vector name, `__stext`, is defined in Cosmic's supplied run time startup code files, crts.s and crtsi.s.

Figure 18 presents the linker segment definition necessary for proper placement of the interrupt vector data. If the source file containing the interrupt and reset vector data is compiled with the +nocst option, the segment definition must be changed to a `.text` segment.

```
+seg .const -b 0xfff8c -o 0xff8c -m 0x74
#
Vectors.o
#
```

Figure 18. Segment Definition for Interrupt Vectors

Alternate Fixed Page Code and/or Constant Data Placement

In addition to the code and constant data placement methods described for the two fixed pages, the linker segment definition supports an option allowing two or more discontinuous memory areas to be automatically filled from a list of object file names. The -x option, shown in the first line of the example in **Figure 19**, operates in a manner much like the automatic segment creation method used for paged memory. When one of the listed segments becomes full, the linker automatically begins to fill the next listed segment.

NOTE: *All listed segments must be of the same type (i.e. `.text`, `.const`, etc.) and the segments must be declared before the occurrence of an object file that would cause a segment overflow. Also note that the -m option must be used to specify a size for each segment. If the -m option is not present, the -x option will be ignored.*

```
+seg .text -b 0xF8000 -o 0x4000 -m 0x4000 -n Fixed1 -x
+seg .text -b 0xFC000 -o 0xC000 -m 0x3F00 -n Fixed2
+seg .const -a Fixed2 -it
#
# put files for Fixed Page 0x3e & 0x3f here;
#
```

Figure 19. -x Option For Two Discontinuous Fixed Page Memory Areas

While the example in **Figure 19** indicates manual placement of object files following the segment definitions, the cbank utility could be used to order a list of object file names so each segment is filled as efficiently as possible. However, because these two segments are not of equal length and because cbank assumes equal length segments, it is possible that cbank might over flow the second segment. If this were to happen, the linker would issue an error message if the combination of code and data became too large to fit in the second segment.

In addition to the two `.text` segment definitions a `.const` segment definition is included in the example to accommodate the placement of any constant sections that may be contained in the fixed page object files.

NOTE: *The `.const` segment definition includes a -it option. This option instructs the linker to use this segment to contain the descriptor and image copies of initialized data used for automatic data initialization.*

S-Record Generation

While the Cosmic ZAP debugger can directly read linked absolute binary object files for programming a target microcontroller's FLASH memory, some development and programming tools require object files in the Motorola S-record format. S-record files can be directly generated from linked absolute binary object files using Cosmic's **chex** converter. The chex program has the ability to produce S-records with either linear or banked (paged) load addresses. By default, chex produces S-records with linear load addresses.

Summary

The compiler and linker tools provide a powerful, flexible code development environment for the MC9S12DP256. The flexibility provided by the linker command file does not allow a single example to be given that will accommodate all application needs. The linker command file shown in **Figure 20** shows one example that may work for many applications. This example simply combines many of the individual examples presented in this application note.

NOTE: *The example makes use of the +inc directive to include the list of file names in BankList.txt into the linker file for automatic segment creation. This list of file names in BankList.txt may be generated by the cbank utility.*

Two items included in this example, not shown in examples elsewhere, are the +def directive at the bottom of the example.

1. The first +def directive defines the symbol `__memory` and assigns the current address of the .bss section to it. This symbol is used by Cosmic's supplied startup code in conjunction with the `__sbss` symbol to clear (set to zero) all variable storage space in the .bss section.
2. The second +def directive defines the symbol `__stack` and assigns it a value of $4000. This symbol is used by the startup code to load an initial value into the CPU12's stack pointer. Because the CPU12 stack operates as a decrement then store stack, the value assigned to the symbol is one more than the last on-chip RAM location rather than the last RAM location.

```
# data segment for initialized data
+seg .data -b 0x1000 -n iRAM -m 0x3000

#data segment for uninitialized data
+seg .bss -a iRAM
+def __sbss=@.bss

# data segment for eeprom data
+seg .eeprom -b 0x0400 -m 0x0c00

+seg .const -b 0xf8000 -o 0x4000 -m 0x4000
#
# All data in .const sections will be placed in the lower fixed page
#

+seg .text -b 0xc0000 -o 0x8000 -w 0x4000 -m 0x38000
#
+inc BankList.txt
#

+seg .text -b 0xfc000 -o 0xc000 -m 0x3f00 -it
#
# put files for Fixed Page 0x3f here; Cosmic Libraries, ISRs, etc.
#

+seg .const -b 0xfff00 -o 0xff00 -m 0x10
#
Security.o
#

+seg .const -b 0xfff8c -o 0xff8c -m 0x74
#
Vectors.0
#

+def __memory=@.bss
+def __stack=0x4000
```

Figure 20. Example Link Command File

[redacted]

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor