

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 page 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 page 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 cannot directly 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 problem because the code that was being executed would disappear from the page window as soon as the new PPAGE value was written. Any constant data such as lookup tables, string or numeric constants that are shared by functions residing on different pages must either be placed in one of the two fixed Flash memory pages or must be accessed through a runtime routine that is located in the fixed Flash memory.

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.

CodeWarrior Compiler

To efficiently handle the paged memory architecture, the Codewarrior compiler introduces two non-ANSI keywords: **near** and **far**. A near function is called with a JSR or BSR instruction, whereas a far function is called with a CALL instruction. For example,

```
void far func1(void);
```

declares a far function which will be located in paged memory and which may be called by a function in a different page.

The near and far keywords may also be used with data pointers. For example,

segment until the next `#pragma CODE_SEG` statement is encountered. Thus assignment of the default segment is a convenient means of terminating a specific segment:

```
#pragma CODE_SEG DEFAULT
```

A function prototype must be declared in the same code segment as the function definition itself. Thus, if the function prototypes are declared in a separate header file, the `#pragma CODE_SEG` statement must be duplicated, as shown in [Figure 2](#) and [Figure 3](#).

```
#pragma CODE_SEG FUNCTIONS
void func1(void);
void func2(void);
#pragma CODE_SEG DEFAULT
```

Figure 2. Example header file functions.h

```
#include "functions.h"

#pragma CODE_SEG FUNCTIONS
void func1(void)
{
  /* code */
};

void func2(void)
{
  /* code */
};

#pragma CODE_SEG DEFAULT
```

Figure 3. Example Code with include file

It is recommended not to omit the `#pragma CODE_SEG DEFAULT` statement at the end of the `functions.h` file, as the resulting behaviour will very much depend on the use of `#pragma CODE_SEG` statements in other included files and on the order in which such files are included.

constant data to the segment ROM_VAR. All constant variables following the #pragma CONST_SEG statement are included in the segment until the next #pragma CONST_SEG statement. Thus assignment of the default segment is a convenient means of terminating a specific segment:

```
#pragma CONST_SEG DEFAULT
```

Any declarations of external data variables must be within the same data segment as the data variable definition itself. Thus if var2 is defined in a constant data segment B in a file, any other files which contain code which accesses var2 would contain the declaration:

```
#pragma CONST_SEG B
extern const int var2;
#pragma CONST_SEG DEFAULT
```

Interrupt Service Routines

If the application uses interrupts, each interrupt service routine must have the #pragma TRAP_PROC command immediately preceding the function, as shown in [Figure 4](#)

```
#pragma CODE_SEG __NEAR_SEG INTERRUPT_ROUTINES
#pragma TRAP_PROC
void interrupt_func1(void)
{
    /* code */
};
#pragma CODE_SEG DEFAULT
```

Figure 4. Example of ISR

The #pragma TRAP_PROC causes the interrupt routine to be terminated with a RTI instruction instead of a RTS instruction. Furthermore, if the compiler option -CpPpage=RUNTIME is specified due to paged data variable accesses, it is also necessary to specify the compiler option -CpPpage=0x30. This specifies the address of the PPAGE register which will be saved on the stack at the start of the interrupt routine and restored at the end of the interrupt routine.

The interrupt vectors themselves are only 16-bits wide and therefore can only point to non-paged memory. This means that the interrupt service routines must be located in non-paged memory. This is easily achieved by creating a specific segment for interrupt routines. In the example in [Figure 4](#), the code segment name INTERRUPT_ROUTINES is defined. Note the use of the attribute __NEAR_SEG in the #pragma statement; this makes it quite explicit that the code segment has to be placed into non-banked memory.

shown in **Figure 7** is changed to that shown in **Figure 8**. This change has been implemented in version 2.0 of the Codewarrior product.

```

/* Compile with option -DDG128 to activate this code */

#ifdef DG128 /* HC12 DG128 derivative has page register at 0xff and only P page */
#define PPAGE_ADDR      (0xFF+REGISTER_BASE)
#ifdef __PPAGE__ /* may be set already by option -CPPPAGE */
#define __PPAGE__
#endif
#elif defined (A4)
/* all setting default to A4 already */
#endif

```

Figure 7. Code snippet from unmodified datapage.c

```

/* Compile with option -DHCS12 to activate this code */
#ifdef HCS12 /* HCS12 family has PPAGE register only at 0x30 */
#define PPAGE_ADDR      (0x30+REGISTER_BASE)
#ifdef __PPAGE__ /* may be set already by option -CPPPAGE */
#define __PPAGE__
#endif

/* Compile with option -DDG128 to activate this code */
#elif defined DG128 /* HC912DG128 derivative has PPAGE register only at 0xFF */
#define PPAGE_ADDR      (0xFF+REGISTER_BASE)
#ifdef __PPAGE__ /* may be set already by option -CPPPAGE */
#define __PPAGE__
#endif
#elif defined (A4)
/* all setting default to A4 already */
#endif

```

Figure 8. Recommended Code Snippet for datapage.c

Datapage.c should be recompiled with the compiler option **-DHCS12** when paged data will be accessed on the MC9S12DP256. If the registers on the MC9S12DP256 will be remapped in the application, it will be necessary to change the defined value of REGISTER_BASE. This value is also defined in datapage.c.

Datapage.c has already been compiled (without **-DHCS12**) and is included in the library 'ansib.lib'. Either the entire ansib.lib library can be rebuilt using the modified datapage.c and compiler option, or the modified datapage.c can be included in the Codewarrior project. As long as the modified datapage.c is above ansib.lib in the project link order list view, the modified datapage.c will be linked into the project instead of the standard datapage.c in the unmodified ansib.lib library.

\$C000 to \$FFFF corresponds to the *same* physical memory as \$3F8000 to \$3FBFFF. For the reasons described above, code is linked to \$C000 to \$FFFF in preference to \$3F8000 to \$3FBFFF.

Linker Command File

The allocation of the defined code and data segments to memory addresses is controlled by the linker command file. This file may be recognised by the file extension '.prm'. Within this file, the `SECTIONS` command block is used to define physical regions of memory. An example of the `SECTIONS` command block for the BANKED memory model on the MC9S12DP256 is given in [Figure 10](#).

```
SECTIONS
  EEPROM = READ_WRITE 0x0400 TO 0x0FFF;
  RAM = READ_WRITE 0x1000 TO 0x3FFF;

  /* non-paged FLASH ROM */
  ROM_4000 = READ_ONLY 0x4000 TO 0x7FFF;
  ROM_C000 = READ_ONLY 0xC000 TO 0xFEFF;

  /* paged FLASH ROM */
  PAGE_30 = READ_ONLY 0x308000 TO 0x30BFFF;
  PAGE_31 = READ_ONLY 0x318000 TO 0x31BFFF;
  PAGE_32 = READ_ONLY 0x328000 TO 0x32BFFF;
  PAGE_33 = READ_ONLY 0x338000 TO 0x33BFFF;
  PAGE_34 = READ_ONLY 0x348000 TO 0x34BFFF;
  PAGE_35 = READ_ONLY 0x358000 TO 0x35BFFF;
  PAGE_36 = READ_ONLY 0x368000 TO 0x36BFFF;
  PAGE_37 = READ_ONLY 0x378000 TO 0x37BFFF;
  PAGE_38 = READ_ONLY 0x388000 TO 0x38BFFF;
  PAGE_39 = READ_ONLY 0x398000 TO 0x39BFFF;
  PAGE_3A = READ_ONLY 0x3A8000 TO 0x3ABFFF;
  PAGE_3B = READ_ONLY 0x3B8000 TO 0x3BBFFF;
  PAGE_3C = READ_ONLY 0x3C8000 TO 0x3CBFFF;
  PAGE_3D = READ_ONLY 0x3D8000 TO 0x3DBFFF;
END
```

Figure 10. Example of Linker Sections Command Block

Within the `SECTIONS` command block, each separate section of physical memory is described with a name, an attribute and an address range. Each page of flash memory is listed, except for PPAGE values \$3E and \$3F. Instead this memory is allocated through the equivalent non-paged address, described as `ROM_4000` and `ROM_C000`. Note that `ROM_C000` ends at \$FEFF. \$FF00 to \$FF0F cannot be used for code as the Flash protection and security registers are located here. Also \$FF8C to \$FFFF are occupied by the interrupt vectors and cannot be used for code.

Once the `SECTIONS` are defined, the code and data segments are allocated to memory using the `PLACEMENT` command block.

```
typedef unsigned char tUINT8;

typedef struct
{
    volatile tUINT8  porta; /* port A data register */
    volatile tUINT8  portb; /* port B data register */
    tUINT8          ddra; /* port A data direction register */
    tUINT8          ddrb; /* port B data direction register */
                    /* continue... */
}tREGISTER;

#pragma DATA_SEG S12_REG
extern      tREGISTER
#pragma DATA_SEG DEFAULT

Registers;
```

Figure 12. Example Code for Registers Variable

```
SECTIONS

                                /* flash, RAM, EEPROM etc */

    DP256_REG = NO_INIT 0x0000 TO 0x0003; /* exact address for Registers */
END

PLACEMENT

                                /* Placement of code */

    S12_REG INTO DP256_REG; /* Placement of Registers */
END
```

Figure 13. Additional Linker Commands for Register Placement

As an alternative to placement of the Registers variable in the linker file, the Registers variable may be placed at an absolute address within the code, by using the non-ANSI '@' symbol as follows:

```
tREGISTER Registers @0x00;
```

Flash Protection and Security Registers

A short summary of the Flash protection and security features is given in Appendix A of this document. Further details are given in Application Note AN2400/D.

Even if the memory security and protection features are not being utilized during development, a constant variable containing data for this 16 byte area should be created, compiled and linked into the absolute 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

lower two bits in the security byte in which the part remains unsecured. **Figure 15** shows an example linker command for the placement of the flash security and protection data.

```
SECTIONS
    /* flash, RAM, EEPROM etc */
    FLASH_PROT_AREA = READ_ONLY 0xFF00 TO 0xFF0F;
END
PLACEMENT
    /* Placement of code */
    FLASH_PROT INTO FLASH_PROT_AREA; /* Placement of Flash Protection Registers */
END
```

Figure 15. Flash Protection Register Placement

Interrupt Vectors

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. There are two methods of creating the address constants for the interrupt vector table. Either the interrupt vector table can be created manually or the linker can generate the interrupt vector table automatically.

An example of a manually created vector table for the MC9S12DP256 is shown in **Figure 16**. This shows an array of constant pointers to functions within a segment called VECTORS.

```
#ifndef NULL
#define NULL 0
#endif

extern void _Startup(); /* startup routine */
extern void CAN0_WakeupISR();
extern void CAN0_ReceiveISR();
extern void CAN0_TransmitISR();

#pragma CONST_SEG VECTORS

void (* const vector_table[])() = {
    NULL, /* $FF8C:8D PWM Emergency Shutdown */
    NULL, /* $FF8E:8F Port P Interrupt */
    NULL, /* $FF90:91 MSCAN 4 transmit */
    NULL, /* $FF92:93 MSCAN 4 receive */
    NULL, /* $FF94:95 MSCAN 4 errors */
    NULL, /* $FF96:97 MSCAN 4 wake- up */
    NULL, /* $FF98:99 MSCAN 3 transmit */
    NULL, /* $FF9A:9B MSCAN 3 receive */
}
```

The VECTORS segment must be allocated to the correct memory address in the linker command file, as shown in [Figure 17](#). In addition, the ENTRIES command must be used to ensure that the table is included. As there are no direct references to the vector table in the code, it would otherwise be 'optimised' and removed.

```

SECTIONS
    /* flash, RAM, EEPROM etc */

    VECTOR_TABLE = READ_ONLY 0xFF8C TO 0xFFFF; /* Vector Table address */
END

PLACEMENT
    /* Placement of code */

    VECTORS INTO VECTOR_TABLE; /* Placement of vector_table */
END

ENTRIES
    vector_table
END

```

Figure 17. Example Linker Command File for Manual Vector Table

Automatic Interrupt Vector Entry Generation

As a much easier alternative to the manual method, the linker can create interrupt vector addresses automatically. This is done using the VECTOR instruction in the linker command file. The array of constant function pointers used for the manual method is not required.

```

VECTOR 0 _Startup
VECTOR 38 CAN0_ReceiveISR

```

Figure 18. Example of Linker Vector Command Block

The number following VECTOR instruction is used by the linker to calculate the vector address. For vector number 'n', the address is given by $\$FFFE - 2*n$. Thus the address of vector 38 in the example in [Figure 18](#) is $\$FFFE - \$4C = \$FFB2$. Vector 0 is the reset vector at $\$FFFE$ and `_Startup` is the default Codewarrior start-up routine. The VECTOR instructions are placed at the end of the linker command file. The ENTRIES command block is no longer required for vector entries when this method is used.


```

PAGE_38           = READ_ONLY  IBCC_FAR 0x388000 TO 0x38BFFF;
PAGE_39           = READ_ONLY  IBCC_FAR 0x398000 TO 0x39BFFF;
PAGE_3A           = READ_ONLY  IBCC_FAR 0x3A8000 TO 0x3ABFFF;
PAGE_3B           = READ_ONLY  IBCC_FAR 0x3B8000 TO 0x3BBFFF;
PAGE_3C           = READ_ONLY  IBCC_FAR 0x3C8000 TO 0x3CBFFF;
PAGE_3D           = READ_ONLY  IBCC_FAR 0x3D8000 TO 0x3DBFFF;
END

PLACEMENT
  _PRESTART, STARTUP,
  ROM_VAR, STRINGS,
  NON_BANKED,
  COPY           INTO  ROM_C000, ROM_4000;

DISTRIBUTE  DISTRIBUTE_INTRO PAGE_30, PAGE_31, PAGE_32, PAGE_33, PAGE_34, PAGE_35, PAGE_36,
PAGE_37, PAGE_38, PAGE_39, PAGE_3A, PAGE_3B;

EEPROM           INTO  EEPROM_AREA;

DEFAULT_RAM     INTO  RAM;
END

```

Figure 19. Example Linker Commands for the Distribute feature

The linker is now run with the **-Dist** option and if required, the **-DistSeg** option. This option suppresses the generation of the normal absolute file, instead a special header file is generated. This header file assigns each function to a segment, along with the appropriate calling mechanism for each function. The name of this header file can be specified with the **-DistFile** option. If this option is not specified the default name 'distr.inc' is assigned. This file is created in the project /bin sub-directory by default. This file requires another include file called interseg.h, which can be found in the Metrowerks /Include sub-directory.

Now the second pass compilation and link can be performed. This time the linker generated header file must be included in every compilation unit. This can conveniently be done by defining 'PASS2' on the compiler command line using the option **-DPASS2**, with the following code in every source file:

```

#ifdef PASS2
#include "distr.inc"
#endif

```

This time the compiler uses the optimized calling convention specified for each function in the linker generated header file. The second pass linking is run *without* any of the distribute feature specifiers: **-DIST** should not be included in the command line. This ensures that the absolute file is generated.

This two pass process could easily be handled by a make file.

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

Figure 20. Flash Protection and Security Memory Locations

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. The tables in **Figure 21** summarize the combinations of the FPHS[1:0] and FPLS[1:0] bits and the size of the protected area selected by each.

FPHS[1:0]	Protected Size	FPLS[1:0]	Protected Size
0:0	2K	0:0	512 Bytes
0:1	4K	0:1	1K
1:0	8K	1:0	2K
1:1	16K	1:1	4K

Figure 21. Flash Protection Select Bits

Trying to program or erase any of the protected areas will result in a protection 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.*

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

Figure 22. Security Bits

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.

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document. Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

