Freescale Semiconductor
Application Note

Document Number: AN3784

Understanding the Memory Scheme in the S12(X) Architecture

by Christian Michel Sendis

1 Introduction

In an S12 or S12X architecture, it is necessary to make the distinction between two types of memory locations: banked and non-banked. This document describes how to ensure a correct access to a given memory location and aims at describing how the CodeWarrior linker distributes your code between these two memory types. Understanding an application's usage of its memory aids in avoiding common pitfalls and helps detecting where there may be room for code optimizations.

A consequence of the size of the HCS12(X) address bus is that not all memory locations are equal. Since the HCS12(X) CPU address bus is 16 bits wide, it can directly access an address that can be encoded in 16 bits. The number of bytes addressable with a 16-bit address is: 2^16 = 65536 bytes, or 64 kB. When you have more than 64 kB of memory, addresses beyond the first 64 kB will not fit in a 16-bit encoding.

Non-banked memory refers to those locations that can be accessed directly with a 16-bit address.

Contents

Banked memory refers to those locations where extra action is needed to expand the addressing capabilities of the HCS12(X) CPU.

Banked and non-banked are synonym terms to paged and non-paged respectively. The terms paged and non-paged come from the idea of the memory page, which is a concept used by the mechanism to extend the memory addressing capabilities. These terms are often used interchangeably in Freescale's literature.

To understand how an application accesses banked memory, you need to understand following three concepts:

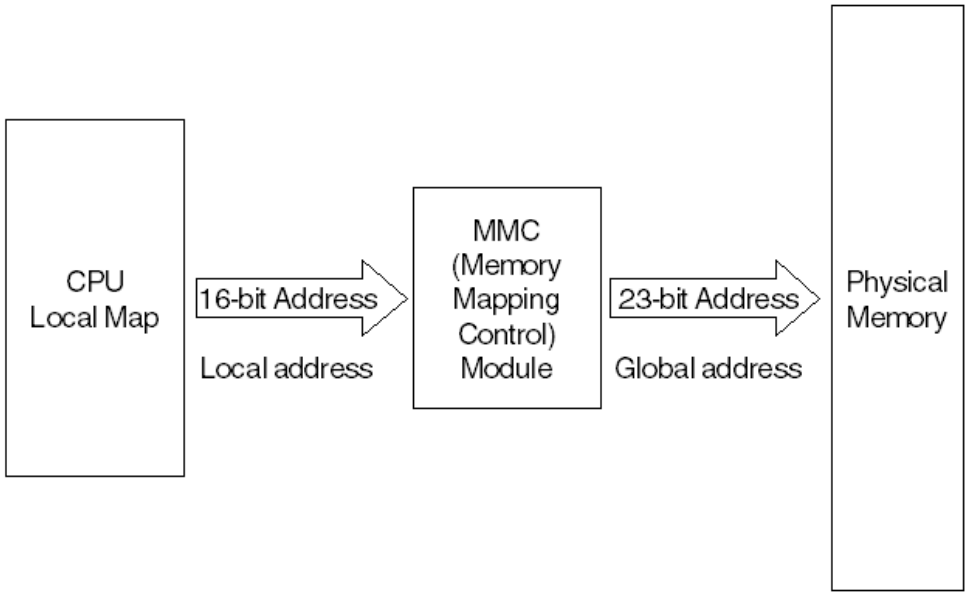
- CPU Local Map
- Page Window
- Memory Page

2 CPU Local Map

The term CPU local map refers to the 64 Kilobyte space that the CPU can directly access through its instruction set. This 64kB addressing space allows access to different kinds of memory resources: Register space, RAM, EEPROM and Flash. The CPU local map acts as a portal to these physical locations.

When reading from or writing to an address in the CPU local map, the memory mapping control MMC module and translates this local address to a different physical address. The MMC converts a local 16-bit address into a different, physical address, encoded in 23 bits. (see Figure 1.) We emphasize this because in the HCS12(X), this 23-bit global address space can also be directly accessed by special instructions. The MMC module is configured at chip integration to associate certain local addresses to certain on-chip memory resources.

Figure 1. MMC Module Translation Process



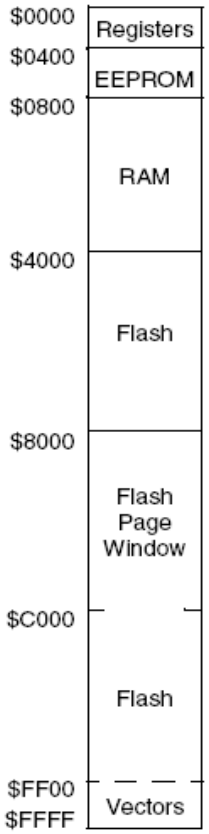
2.1 CPU Local Map for HCS12 Family

Figure 2. shows how addresses in the 64kB space are associated to a particular memory resource for an HCS12 device. The registers and other memory resources have dedicated address ranges. In this case the figure portrays an S12DP512 part.

In the case of the HCS12 family, local memory maps may change from device to device, however, they share two common characteristics:

- The first common characteristic is that RAM, EEPROM and Register space boundaries may change from device to device, but the default location to which MMC maps RAM, EEPROM, and Registers, after a power-on reset, is always in the first 16kB region of the local map (from addresses 0x0000 to 0x3FFF). In the HCS12 family only, you can modify the location of EEPROM, RAM and Register spaces by writing to special INIT registers inside the MMC module. Refer to the MMC section in your device documentation for more information.
- The second common characteristic is that the lower 48 kB hosts the Flash memory (from address 0x4000 to 0xFFFF). This Flash area is divided into three 16kB regions. The middle 16kB region, from addresses 0x8000 to 0xBFFF, is called the Flash page window. (see Page Window).

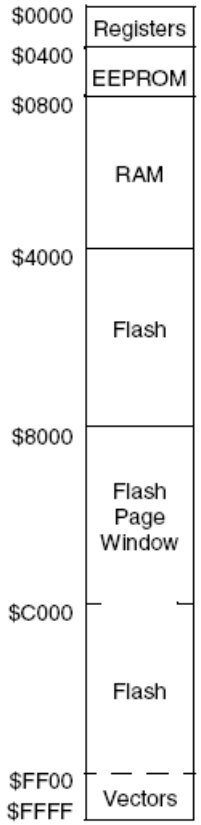
Figure 2. CPU Local Map for an HCS12 Device



2.2 CPU Local Map for the HCS12X Family

In the HCS12(X) family, CPU local maps have been homogenized for all devices. Figure 3. shows how addresses in the 64 kB space are associated to a particular memory resource. The addresses that define all these different parts of the CPU local map are identical across all HCS12(X) devices.

Figure 3. Default CPU Local Map for an HCS12X Device



3 Page Window

The page window concept is the same for the HCS12 and the HCS12X architecture. Most of the local addresses in the CPU local map always point to the well-defined fixed physical locations. Certain local addresses, however, do not always point to the same physical locations. These special address ranges are called `page windows`. Local addresses inside a page window range are addresses of 16 bits which do not contain enough information for the `MMC` module to determine a well-defined physical location.

For a local address inside a page window range, the `MMC` module requires additional information, stored inside a register, to be able to translate the given local address into a well-defined physical location. The register that contains this information is called a `page register`.

The HCS12 architecture contains only one page window used for Flash memory accesses. This page window is located from addresses `0x8000` to `0xBFFF`. The page register that is associated to this page

window is called the PPAGE register and is used to select the part of the physical memory to which the Flash page window points.

Changing the value of the PPAGE register will change the contents reflected inside the CPU local map page window.

The HCS12X CPU local map contains three page windows: one for EEPROM, one for RAM, and one for Flash. Special page registers select the part of the physical memory to which each page window points. The EPAGE page register selects the physical memory range for the EEPROM; RPAGE selects the physical memory range for the RAM; PPAGE selects the physical memory range for the Flash.

Changing the value of a page register will change the contents reflected inside the associated local map page window.

4 Memory Page

A memory page is a continuous section of physical memory with a fixed size. The page size depends on the memory resource considered: 1 Kilobyte for EEPROM, 4 kB for RAM and 16 kB for Flash The size of a memory page is the same as the size of that memory resource's page window in the local map.

The division of physical memory into pages is a conceptual division. Pages do not correspond to real physical divisions of memory. Each page is identified with a page number. This is the number that has to be written to the page register in order for that particular page to be displayed inside the page window. Page numbers are defined at chip integration. The numbering scheme is as follows:

- In the HCS12 Family, pages are chosen in sequential order in such a way that the last page of memory is given the number 0x3F.

  For example: If your S12 device has 32 kB of Flash, the Flash will be conceptually divided into two 16kB pages with numbers 0x3E and 0x3F. If it has 48 kB of Flash, 3 Flash pages will be defined, with numbers 0x3D, 0x3E and 0x3F,
- In the HCS12 Family, pages are chosen in sequential order in such a way that the last page of memory is given the number 0xFF.

  For example: If your S12 device has 8 kB of RAM, the RAM will be conceptually divided into two 4kB pages with numbers 0xFE and 0xFF. If it has 16 kB of RAM, 4 RAM pages will be defined, with numbers 0xFC, 0xFD, 0xFE and 0xFF.

NOTE Although the concepts of page and page numbers are only useful when accessing banked locations, it is important to understand that this conceptual division into numbered pages is done for the entire memory resource in question. Any memory location inside that resource will have an associated page number, regardless of whether it will later be used or not. That is, regardless of whether that memory location will be accessed through a page switching mechanism or directly.

4.1 Page Switching Mechanism

To view a particular physical page in the local map's page window, the page number needs to be written into the page register. For example, if writing in a high-level language, such as C, CodeWarrior compiler takes care of generating appropriate instructions and the handling of the page registers is transparent to the user. In this case, the user only needs to make sure that the compiler correctly understands when a variable or function is located in banked memory in order to generate the additional instructions. This is done by selecting an appropriate memory model during the project's creation, or by using special syntax when declaring a variable or a function.

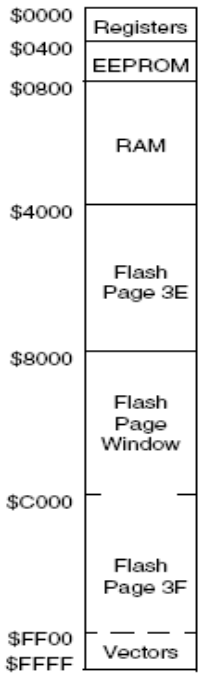
Once the appropriate page of physical memory is displayed inside the page window, the CPU can access data with a 16-bit address.

NOTE You can access the entire contents of a paged memory resource through the page window. However, writing to the page register every time you need to access a memory location introduces a certain amount of overhead. This is why certain locations are also mapped directly into the local map. Locations which are always mapped onto the CPU local map, regardless of any page-register value, are called non-banked, or non-paged locations. For these locations, paged access is not usually used and direct access is always preferred.

4.2 Page Switching for HCS12 Devices

Figure 4. shows the cpu local map, where the non-banked locations have been labeled with the corresponding page numbers that they always mirror. The examples which follow illustrate page switching for the HCS12 devices.

Figure 4. CPU Local Map



4.2.1 HCS12 Case Example for Non-banked Location

The local address value `0xC000` corresponds to a non-banked location. According to Figure 4. this address points to the first byte of Flash page number 0x3F. Reading or writing to address `0xC000` always results in an access to this same physical location, regardless of any page register value.

You can also access the first byte of the physical Flash page `0x3F` by writing the value `0x3F` into the `PPAGE` register. This displays all of the contents of Flash page `0x3F` inside the Flash page window range. The first byte of Flash page `0x3F` then appears at the local address `0x8000`.

Accessing the same physical location with these two procedures produces correct results in both cases; however, a direct access to `0xC000` has less overhead and is therefore preferred.

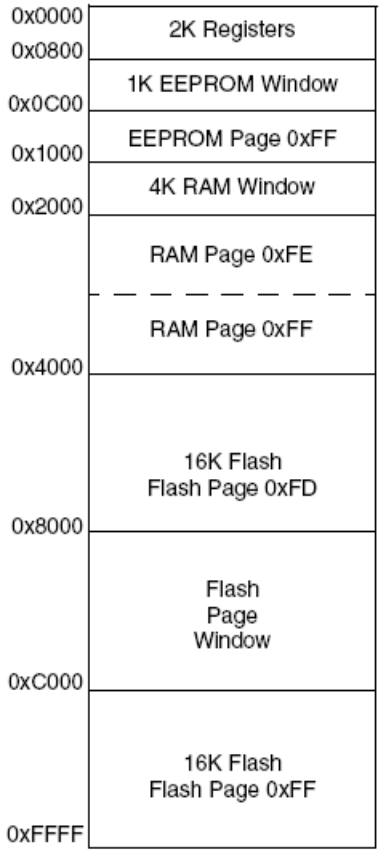
4.2.2 HCS12 Case Example for Banked Location

Suppose you wish to read the value of the first byte belonging to the Flash page `0x3C`. In this case, Flash page `0x3C` cannot be found as a non-banked location in the CPU local map (refer to Figure 4.). The only solution in this case is to use the Flash page window to make the access. The application must write value `0x3C` to the `PPAGE` register, then access the first byte of this page, located at local address `0x8000`.

4.3 Page Switching for HCS12X Devices

Figure 5. shows the non-banked locations with the corresponding numbers of physical memory to which they point, and the page numbers associated with the non-banked locations in the local map of the HCS12X. The examples which follow illustrate page switching for HCS12X devices.

Figure 5. HCS12X Local Map with Page Numbers



4.3.1 HCS12X Case Example for Non-banked Location

The local address value `0xC000` corresponds to a non-banked location and does not belong to any page window. According to Figure 5., this address points to the first byte of the Flash page number `0xFF`. Reading or writing to address `0xC000` always results in an access to this same physical location, regardless of any page register value.

You can access the first byte of the physical Flash page `0xFF` by writing the value `0xFF` into the `PPAGE` register. Then, all the contents of Flash page `0xFF` display inside the Flash page window range. The first byte of Flash page `0xFF` then appears at local address `0x8000`.

Accessing the same physical location with these two procedures produces correct results in both cases; however, a direct access to `0xC000` has less overhead and is therefore preferred.

4.3.2 HCS12X Case Example for Banked Location

Suppose we wish to read the value of the first byte belonging to Flash page 0xFC. In this case, Flash page 0xFC cannot be found as a non-banked location in the CPU local map (see Figure 5.). The only solution in this case is to use the Flash page window to make the access. The application must write value 0xFC to the PPAGE register, then access the first byte of this page, located at local address 0x8000.

In both the HCS12 and HCS12X cases, CodeWarrior's C compiler takes care of automatically inserting instructions that write the appropriate value to the page register before accessing a paged location. However, to ensure that this happens, the programmer needs to select the memory model that is most appropriate for the application, and eventually use special qualifiers like __near or __far keywords, or #pragma statements to locally modify compiler behavior where needed.

4.4 Global Access for HCS12X Devices

In the S12 architecture, the biggest size that an object can have is 16kB. This limitation is imposed by the CPU local map, where the biggest continuous memory space accessible at a time by the CPU is 16kB. In the S12 architecture attempting to allocate an object bigger than 16kB results in a linker error.

To reduce this limitation, in the S12X architecture, another method for accessing memory has been introduced: Global addressing.

Global addressing makes it possible to access continuous regions of memory of up to 64K in size, in a "new" address space, which is called the global address space.

From the point of view of the S12X CPU, there are two 64kB address spaces where it can access data:

- The 64kB CPU local map
- The 64kB Global map

This 64kB global map is completely independent from the 64 kB local map.

To instruct the CPU to access the Global map instead of the more commonly used local map, the programmer has to use special global instructions. These global instructions are: GLDAA, GLDAB, GLDD, GLDS, GLDX, GLDY, GSTAA, GSTAB, GSTD, GSTS, GSTX and GSTY (See *CPU Block Guide* for details).

4.4.1 Example

- GLDAA $100 loads the accumulator A with the value stored at the address 0x100 in the global map.
- LDAA $100 loads the accumulator A with the value stored at the address 0x100 from the local map.

Now that we have seen the notion of page windows and page registers, the 64 kB global map can be understood as a 64kB page window, where the contents displayed depend on a fourth page register called GPAGE.

Global addresses are 23 bit addresses that cover an 8 Mb address space, ranging from addresses 0x000000 to 0x7FFFFF. In this linear global address space, all memory resources are grouped, and the

GPAGE register can be used to access all RAM. EEPROM and FLASH locations, as well as external memory space.

Once the correct value for GPAGE has been written, the contents of the global map can only be accessed via global instructions.

The value for GPAGE is chosen in a similar way to that of the other page registers:

- Writing 0xFF to GPAGE will result in the global map displaying the last 64kB of the total memory space,
- Writing 0xFE will display the penultimate 64kB of physical memory and so on.

4.4.2 When to use Global Addressing

Global addressing is mostly intended to be used for two reasons:

1. When linking very big data objects that cannot be linked otherwise because of not having sufficient continuous space in the local map.

   In this case, using global addressing allows the programmer to have up to 64kB of continuous space that he can access through global instructions. The biggest size for a single data object that can be linked is now 64kB.
2. When trying to access a paged object (variable or constant) while running paged code from the same memory resource.

For example, when the application needs to access constants located in a given page of Flash while executing code running in a different page of Flash. Normally, when this situation is encountered in the S12 architecture, a non-banked runtime routine is used to access the paged object.

In the S12X, the new global access can be used to access constants anywhere in the Flash, while running from paged Flash, without disturbing the PPAGE register and without the need to jump to a non-banked routine.

In order to instruct the compiler to use global accesses for a given object, it is sufficient to declare it inside a #pragma DATA_SEG __GPAGE_SEG block, or #pragma CONST_SEG __GPAGE_SEG block, depending on the object's nature.

4.4.3 S12X Local Map Remapping Capabilities

In the newer S12X devices, the MMC module can be configured by the user so that the CPU local map address range 0x4000 to 0x7FFF. This portion of the local map is by default routed to display Flash, but it can be configured to display RAM or External space, therefore providing more flexibility to the user as to what locations are defined as non-banked. Please refre to the compiler option -Map of *S12X Compiler Manual* and your device's datasheet, MMC module, for more information on this feature.

You have now seen a general description of the different ways in which a memory location can be accessed. The next chapters describes how to instruct CodeWarrior linker to place our code and variables in desired memory locations. We will also see, when developing in C language, how to ensure that CodeWarrior's compiler is aware of which objects are placed in the banked or non-banked memory locations, in order to generate appropriate code.

5 Controlling Placement of Objects in Memory

This section describes how CodeWarrior linker behaves by default when placing objects in memory, and how to change this default behaviour to customize our application.

The term `objects` refers to entities that have a fixed address in memory. These can be:

- Functions (code)
- Variables (data and arrays placed in RAM)
- Constants (data placed in Flash and qualified as "const")
- Strings (string literals that are not previously defined as arrays.

  For example: `printf( "Hello World")` will generate the string "`Hello World`". On the contrary declaring a variable as unsigned char Message[] = "`Hello World`", will be considered by the linker as being an array and not a string).

The location of objects is controlled by the #pragma statements. What follows is not an exhaustive description of all #pragma statements available, but only of those more commonly used. For a more detailed description please refer to your compiler and build tools manuals located in your CodeWarrior installation directory.

The four kinds of #pragma statements that concerns us here are:

- `#pragma CODE_SEG`
- `#pragma DATA_SEG`
- `#pragma CONST_SEG`
- `#pragma STRING_SEG`

Each of these statements can be inserted inside a C source file and controls the location and attributes of the objects that follow the statement.

5.1 How to use pragma statements to control location of objects

Let's start with an example. Here, a variable, a constant and a function are placed in an explicit `placement section`. Placement sections are labels that point to specific areas in memory and that are defined in the `PLACEMENT` block of the project's linker parameter file `(*.prm file)`. A reminder of the linker parameter file structure is given in the next chapter, for reference.

```
unsigned char variable1;
const unsigned char constant1;
void function1(void)
{
/* code */
}
```

`#pragma` statements are not mandatory. In the example above, `#pragma` statement are absent. In this case, the linker assumes a default behavior and will place objects in their default locations.

- `DEFAULT_ROM` is the default location for code
- `DEFAULT_RAM` is the default location for variables and arrays

- ROM_VAR is the default location for constants (ROM variables)

DEFAULT_ROM, DEFAULT_RAM and ROM_VAR are labels defined inside the PLACEMENT block of the linker parameter file. They are special keywords recognized by CodeWarrior. The linker knows, for example, that ROM_VAR is the location where constants should be stored in the absence of the #pragma statements.

In this example:

- variable1 will be placed in DEFAULT_RAM
- constant1 will be placed in ROM_VAR
- function1 will be placed in DEFAULT_ROM

The linker parameter file defines the address ranges associated to each of the placement sections.

The next example illustrates the use of #pragma statements, in case the user wants to modify the default behavior of the linker.

```
#pragma DATA_SEG MYVARIABLES  (1)
unsigned char variable1;
#pragma DATA_SEG DEFAULT (2)

unsigned char variable2;

#pragma CONST_SEG MYCONSTANTS  (3)
const unsigned char constant1;
#pragma CONST_SEG DEFAULT      (4)

const unsigned char constant2;

#pragma CODE_SEG MYCODE    (5)
void function1(void)
{
/* code of function1 */
}
#pragma CODE_SEG DEFAULT   (6)

void function2(void)
{
/* code of function2 */
}
```

Here are some rules that define the behavior of #pragma statements :

- A #pragma statement has an effect over the objects that concern it only. For example, a #pragma DATA_SEG statement will not affect the location of const variables. Only #pragma CONST_SEG statements can affect the location of constant objects.
- A #pragma statement has an effect on objects that are declared after the #pragma statement only, and until another #pragma statement of the same nature is encountered, or until the end of the compilation unit (see comment below)

In the example above, six #pragma statements have been included, and are numbered to facilitate their reference.

In this example, we assume that the labels MYVARIABLES, MYCONSTANTS and MYCODE are names of placement sections defined by the user inside the linker parameter file of the project.

- #pragma statement (1) will cause variable1 to be allocated in the placement section MYVARIABLES.
- #pragma statement (2) will terminate the effect of #pragma statement (1). As a result of this, variable2 will be allocated in its default location, that is to say, DEFAULT_RAM.
- #pragma statement (3) will cause constant1 to be allocated in the placement section MYCONSTANTS.
- #pragma statement (4) will terminate the effect of #pragma statement (3). As a result of this, constant2 will be allocated in its default location, that is to say, ROM_VAR.
- #pragma statement (5) will cause function1 to be allocated in the placement section MYCODE.
- #pragma statement (6) will terminate the effect of #pragma statement (5). As a result of this, function2 will be allocated in its default location, DEFAULT_ROM.

NOTE We mentioned before that the effect of a #pragma statement is active until the next #pragma statement is encountered, or until the end of the compilation unit is reached. A compilation unit is equal to the source file plus all of its included header files. This means a #pragma statement inside a *.h header file can modify the location of objects declared inside the source file where this header file is included. And this can be difficult to keep track of, for the developer. This is why it is a good programming practice to always "close" the effect of a #pragma statement by explicitly writing a #pragma DEFAULT statement like we did in the example above.

NOTE In the example above, only the default behavior of the linker is being modified. The job of the linker finishes when all objects have been given an address. In the examples above the placement sections used can be banked or non-banked. The linker is to a certain extent not concerned about this. It is the compiler's job to generate instructions to access memory. And it is the responsibility of the developer to make sure that compiler is aware of what objects are placed in banked and non banked memory, for the compiler to be able to generate the correct access instructions. i.e. handle a page register or not.

Many developers take the approach of working with the default compiler assumptions. This removes the need to customize the compiler behavior for different objects in the application. A default compiler assumption concerning how to access memory is called a memory model. This is discussed in the next chapter.

5.2 Default behavior of the Compiler and Linker

When creating a new CodeWarrior project with the project Wizard, the user is asked to select a memory model among three possible: Small, banked and large memory models. The memory model chosen will determine where CodeWarrior's linker places your code and variables by default, and how CodeWarrior's compiler generates instructions to access your objects.

Here's a description of each of the memory model:

- **Small memory model**: Both your code and variables are put by default in non-banked locations.
- **Banked memory model**: Your code is put by default in a banked location, but your variables are put by default in non-banked locations.
- **Large memory model**: Both your code and your data are put by default in a banked location.

Choosing a memory model will affect three elements in your project:

a) compiler options
b) ANSI library
c) linker parameter file

5.2.1 Project's Compiler Options

The behavior of the compiler is affected by the memory model chosen. CodeWarrior project wizard inserts a `-M` compiler option in your project's compiler options. There are three options available, depending on the model: `-Ms`, `-Mb` or `-Ml`. This option instructs the compiler to behave accordingly to the assumptions made by the model.

The Small memory model will have the option `-Ms`. The compiler will not insert any instructions to handle any page register. Variables will be accessed directly in non-banked locations and your code will be executed using `JSR/RTS` instructions.

The Banked memory model will have the option `-Mb`. The compiler will use instructions that handle the `PPAGE` register when accessing your code. The `CALL` instruction will be used when calling a function. This `CALL` instruction takes care of writing your function's page number to the `PPAGE` register prior to executing it. By default, the variables continue to be accessed as if they were in non-banked locations.

The Large memory model will have the option `-Ml`. The compiler will use the `CALL` instruction to access your code and also, will insert page-handling instructions before each access to `ram` and `EEPROM` variables, which will be assumed to reside in paged locations. This memory model is therefore very demanding in code size and execution time and is most of the time not recommended.

If you need to access paged variables, most of the time it will be sufficient to select a banked memory model and use special `C-` qualifiers to notify the compiler whenever a variable is in a paged location. Accessing paged variables is discussed later in this document. This allows you to use the variable paged access only when needed and not as a default for all variables.

NOTE `JSR/RTS` and `CALL/RTC` instructions are discussed briefly in a subsequent section.

5.2.2 Project's ANSI Library

Choosing a memory model in the project wizard will also add the appropriate ANSI library. ANSI libraries come in precompiled `*.lib` files that need to be compatible to the compiler options chosen for your project.

5.2.3 Project's Linker Parameter File

This is the file that instructs the linker where to put your code. Depending on the memory model chosen, a different kind of parameter file will be included in your project.

Figure 6. show parameter files created by CodeWarrior project wizard for an `S12XEP100` device, for a project where the `XGATE` is not enabled.

We will look only at the `PLACEMENT` block of these files, for the small and banked memory models.

Figure 6. Small Memory Model

```
PLACEMENT /* here all predefined and user segments are placed into the SEGMENTS defined above. */
      _PRESTART,              /* Used in HIWARE format: jump to _Startup at the code start */
      STARTUP,                /* startup data structures */
      ROM_VAR,                /* constant variables */
      STRINGS,                /* string literals */
      VIRTUAL_TABLE_SEGMENT,  /* C++ virtual table segment */
    //.ostext,                /* eventually OSEK code  */
      DEFAULT_ROM, NON_BANKED,          /* runtime routines which must not be banked */
      COPY                    /* copy down information: how to initialize variables */
                              /* in case you want to use ROM_4000 here as well, make sure
                                 that all files (incl. library files) are compiled with the
                                 option: -OnB=b */
                      INTO  ROM_C000/*, ROM_4000*/;

      OTHER_ROM       INTO          PAGE_FE,          PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8,
                            PAGE_F7, PAGE_F6, PAGE_F5, PAGE_F4, PAGE_F3, PAGE_F2, PAGE_F1, PAGE_F0,
                            PAGE_EF, PAGE_EE, PAGE_ED, PAGE_EC, PAGE_EB, PAGE_EA, PAGE_E9, PAGE_E8,
                            PAGE_E7, PAGE_E6, PAGE_E5, PAGE_E4, PAGE_E3, PAGE_E2, PAGE_E1, PAGE_E0,
                            PAGE_DF, PAGE_DE, PAGE_DD, PAGE_DC, PAGE_DB, PAGE_DA, PAGE_D9, PAGE_D8,
                            PAGE_D7, PAGE_D6, PAGE_D5, PAGE_D4, PAGE_D3, PAGE_D2, PAGE_D1, PAGE_D0,
                            PAGE_CF, PAGE_CE, PAGE_CD, PAGE_CC, PAGE_CB, PAGE_CA, PAGE_C9, PAGE_C8,
                            PAGE_C7, PAGE_C6, PAGE_C5, PAGE_C4, PAGE_C3, PAGE_C2, PAGE_C1, PAGE_C0;

    //.stackstart,              /* eventually used for OSEK kernel awareness: Main-Stack Start */
      SSTACK,                   /* allocate stack first to avoid overwriting variables on overflow */
    //.stackend,                /* eventually used for OSEK kernel awareness: Main-Stack End */
      DEFAULT_RAM               /* all variables, the default RAM location */
                      INTO  RAM;

      PAGED_RAM       INTO  /* when using banked addressing for variable data, make sure to specify
                               the option -D__FAR_DATA on the compiler command line */
                            RAM_F0, RAM_F1, RAM_F2, RAM_F3, RAM_F4, RAM_F5, RAM_F6, RAM_F7,
                            RAM_F8, RAM_F9, RAM_FA, RAM_FB, RAM_FC, RAM_FD;

    //.vectors          INTO  OSVECTORS; /* OSEK vector table */
END
```

Figure 7. Banked Memory Model

```
PLACEMENT /* here all predefined and user segments are placed into the SEGMENTS defined above. */
      _PRESTART,              /* Used in HIWARE format: jump to _Startup at the code start */
      STARTUP,                /* startup data structures */
      ROM_VAR,                /* constant variables */
      STRINGS,                /* string literals */
      VIRTUAL_TABLE_SEGMENT,  /* C++ virtual table segment */
    //.ostext,                /* eventually OSEK code  */
      NON_BANKED,             /* runtime routines which must not be banked */
      COPY                    /* copy down information: how to initialize variables */
                              /* in case you want to use ROM_4000 here as well, make sure
                                 that all files (incl. library files) are compiled with the
                                 option: -OnB=b */
                      INTO  ROM_C000/*, ROM_4000*/;

      DEFAULT_ROM     INTO          PAGE_FE,          PAGE_FC, PAGE_FB, PAGE_FA, PAGE_F9, PAGE_F8,
                            PAGE_F7, PAGE_F6, PAGE_F5, PAGE_F4, PAGE_F3, PAGE_F2, PAGE_F1, PAGE_F0,
                            PAGE_EF, PAGE_EE, PAGE_ED, PAGE_EC, PAGE_EB, PAGE_EA, PAGE_E9, PAGE_E8,
                            PAGE_E7, PAGE_E6, PAGE_E5, PAGE_E4, PAGE_E3, PAGE_E2, PAGE_E1, PAGE_E0,
                            PAGE_DF, PAGE_DE, PAGE_DD, PAGE_DC, PAGE_DB, PAGE_DA, PAGE_D9, PAGE_D8,
                            PAGE_D7, PAGE_D6, PAGE_D5, PAGE_D4, PAGE_D3, PAGE_D2, PAGE_D1, PAGE_D0,
                            PAGE_CF, PAGE_CE, PAGE_CD, PAGE_CC, PAGE_CB, PAGE_CA, PAGE_C9, PAGE_C8,
                            PAGE_C7, PAGE_C6, PAGE_C5, PAGE_C4, PAGE_C3, PAGE_C2, PAGE_C1, PAGE_C0;

    //.stackstart,              /* eventually used for OSEK kernel awareness: Main-Stack Start */
      SSTACK,                   /* allocate stack first to avoid overwriting variables on overflow */
    //.stackend,                /* eventually used for OSEK kernel awareness: Main-Stack End */
      DEFAULT_RAM               /* all variables, the default RAM location */
                      INTO  RAM;

      PAGED_RAM       INTO  /* when using banked addressing for variable data, make sure to specify
                               the option -D__FAR_DATA on the compiler command line */
                            RAM_F0, RAM_F1, RAM_F2, RAM_F3, RAM_F4, RAM_F5, RAM_F6, RAM_F7,
                            RAM_F8, RAM_F9, RAM_FA, RAM_FB, RAM_FC, RAM_FD;

    //.vectors          INTO  OSVECTORS; /* OSEK vector table */
END
```

All the blue colored labels are special keywords recognized by CodeWarrior. Each of these labels fulfill a specific purpose:

- `DEFAULT_ROM` is where your code will be allocated by default, when there are no `#pragma CODE_SEG` statements that indicate otherwise. It is a mandatory field in the parameter file.
- `DEFAULT_RAM` is where your variables will be allocated by default, when there are no `#pragma DATA_SEG` statements indicating otherwise. It is a mandatory field in the parameter file.
- `__PRESTART` indicates where the Startup code will be placed.
- `STARTUP` is where the Startup data structure will be placed.
- `ROM_VAR` holds the default location of the constants (variables declared "const")
- `STRINGS` is the where your string literals will be allocated by default, when there is no `#pragma STRING_SEG` indicating otherwise. (A string literal is a literal value passed to a function, for example the string "`hello world`" used by `printf( "hello world");`
- `NON_BANKED` is a special label used by the libraries to store objects that must be non-banked. This label can also be used by the programmer in his code.
- `VIRTUAL_TABLE_SEGMENT` is used by C++ applications only.
- `COPY` is where the initialization values of the `ram` objects will be placed. For example, whenever declaring a variable :

  ```
  unsigned char variable=0xAA;
  ```

  Value `0xAA` is stored in Flash inside the `COPY` section. The startup code will copy this value into the location of "variable" after every reset.
- `SSTACK` is where your stack will be placed. The size of `SSTACK` is determined by the command `STACKSIZE`, also appearing in the prm file.

Any label colored black is not a special keyword. (see Figure 5.and Figure 6.) Here, the labels in black were defined for the sake of providing an example. This is the case of the labels PAGED_RAM or OTHER_ROM in the figures above. CodeWarrior does not use these labels for any particular purpose. Their use is intended to be specified by the programmer through the use of #pragma statements placed inside the application's code.

What changes basically between the small and banked memory models parameter files is the location of the DEFAULT_ROM label.

The small memory model parameter file does not use the paged Flash locations. The label OTHER_ROM is pointing to paged locations, but is not used by the wizard-created project. OTHER_ROM is only there for the programmer to use it if needed.

It is important to understand that you can work with paged or non-paged objects in any memory model. The memory model only affects the default location and the default compiler behavior.

Inside your code, you can always locally modify the default behavior by using special syntax.

For example, in order to work in a banked memory model with banked variables, some extra typing needs to be done. The following section describes coding precautions.

5.3 Deviating from Default Compiler Assumptions

5.3.1 Modifying Default Access to Code

The most efficient way to access a function placed in non-banked memory is via a JSR (jump to subroutine) / RTS (return from subroutine) instruction pair. The JSR/RTS instruction pair does not handle any page register and uses simple 16-bit addresses for the jumps.

On the other hand, to access functions placed in banked memory, the CALL/RTC (Return from call) instruction pair must be used. CALL/RTC instructions do handle the PPAGE register.

To force the use of an JSR/RTS pair when calling a function, that function has to be qualified with __near.

Example:

```
void __near myfunction(void);
```

To force the use of a CALL/RTC pair when calling a function, that function has to be qualified with "__far".

Example:

```
void __far myfunction(void);
```

5.3.2 Modifying Default Access to your Variables

If a variable is placed in a non-banked section of memory, no special keywords are needed to ensure a correct access.

In the S12 architecture only the internal Flash memory resource is paged so the only scenario where paged data would be accessed would concern data placed in paged Flash, that is to say, paged constants. See example below:

Example 1: Accessing paged constants in an S12 architecture

Listing 1. Accessing paged constants in an S12 architecture

```
#pragma CONST_SEG PAGEDCONSTANTS
volatile const unsigned int __far constant1=0xAAAA;
#pragma CONST_SEG DEFAULT
unsigned int variable1;
void main(void) {
   variable1=constant1;
for(;;) {}
}
```

Listing 1 shows the value of `constant1` is read into `variable1`. `constant1` is placed in a paged Flash location. Figure 8. shows the linker parameter file where this label is defined.

Figure 8. Linker Parameter File

```
DEFAULT_ROM     INTO  PAGE_20, PAGE_21, PAGE_22, PAGE_23,
                      PAGE_28, PAGE_29, PAGE_2A, PAGE_2B,
                      PAGE_30, PAGE_31, PAGE_32, PAGE_33,
                      PAGE_38, PAGE_39, PAGE_3A, PAGE_3B,
PAGEDCONSTANTS  INTO  PAGE_3C;
```

Notice the qualifier `__far` used here in the declaration of `constant1`. This `__far` qualifier instructs the compiler to handle the `PPAGE` register before accessing the address of constant1. Because only one page register exists in the S12 architecture, it is not necessary to specify more.

In the S12X architecture we do have to speficy which page register is associated to which variable. This is shown in the following example:

Example 2: Accessing Paged Variables in an S12X Architecture

When a variable is banked, the programmer needs to tell the compiler which is the page register associated with that variable's location. This is done throough a #pragma statement.

The keywords `__RPAGE_SEG`, `__EPAGE_SEG`, `__PPAGE_SEG` and `__GPAGE_SEG` are used to indicate to the compiler that `RPAGE`, `EPAGE`, `PPAGE` and `GPAGE` registers should be handled, respectively.

5.3.2.1 Variables in Banked RAM

Before accessing a banked `RAM` location, the compiler needs to insert instructions that write an appropriate value into the `RPAGE` register. The syntax to instruct the compiler to do this is:

```
#pragma  DATA_SEG  __RPAGE_SEG  PAGED_RAM
```

5.3.2.2 Variables in Banked EEPROM

Before accessing a banked EEPROM location, the compiler needs to insert instructions that write an appropriate value into the EPAGE register. The syntax to instruct the compiler to do this is:

```
#pragma DATA_SEG __EPAGE_SEG MY_EEPROM
```

5.3.2.3 Constants in Banked Flash

Before accessing a banked FLASH constant, the compiler needs to insert instructions that write an appropriate value into the PPAGE register. The syntax to instruct the compiler to do this is:

```
#pragma CONST_SEG   __PPAGE_SEG  OTHER_ROM
```

Let's see an example of how to access a paged RAM variable, and a paged ROM variable:

```
#pragma DATA_SEG __RPAGE_SEG PAGED_RAM      (1)
unsigned int variable1;
#pragma DATA_SEG DEFAULT

#pragma CONST_SEG __GPAGE_SEG PAGED_ROM   (2)
volatile const unsigned int constant1=0xAAAA;
#pragma CONST_SEG __GPAGE_SEG DEFAULT

void main(void) {
  variable1 = constant1;
  for(;;) {} /* wait forever */
  /* please make sure that you never leave this function */
}
```

Figure 9. displays the linker parameter file for this example.

Figure 9. Linker Parameter File

```
DEFAULT_ROM       INTO  PAGE_FC;
PAGED_ROM         INTO  PAGE_C0;
PAGED_RAM         INTO  RAM_F3;
```

In this example, `#pragma` statement (1) is used to indicate to the linker to place variable1 in the placement section called `PAGED_RAM`. At the same time, with the inclusion of the qualifier `__RPAGE_SEG` we are telling the compiler that the RPAGE register must be handled before any access to the variables that will be concerned by this `#pragma`.

`#pragma` statement (2) is used to instruct the linker to place the constant constant1 into the placement section called `PAGED_ROM`, and at the same time instruct the linker to use the `GPAGE` register for accesses to constants affected by this #pragma.

This is an interesting choice. We intentionally chose to use the `GPAGE` register to access paged Flash data, because in this example, the `DEFAULT_ROM` location, where our code resides, is also in a paged Flash location. Therefore while executing the main function, the `PPAGE` register needs to be set to the value `0xFC` (see Figure 9.), so that the CPU can read the code of the main function inside the Flash page window. While executing from the Flash page window, the `PPAGE` value must not be changed, otherwise the CPU

will get lost. This is why, in order to access another Flash page location we can no longer use the PPAGE register. We choose therefore to use the GPAGE register instead.

This is one typical example of the advantages of having a GPAGE register.

This concludes the present document. For more information on the linker and compiler behavior, refer to the build tool and compiler manuals located in the CodeWarrior installation directory. Code examples can be found in the (CodeWarrior_Examples) folder in the CodeWarrior installation directory.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com