Freescale Semiconductor
Application Note

Document Number: AN3730
Rev. 0, 05/2008

Understanding Memory Paging in 9S08 Devices

by: Eduardo Viramontes / Rick Li
Freescale Technical Support

1 Fundamentals of Memory Paging in 9S08 Devices

Memory paging provides digital systems with greater memory addressing capabilities without expanding basic architecture resources, such as bus size or address pointers. The memory page is a set of memory addresses that comprise a "view" by a digital system under specific conditions, such as changes to a combination of flags resulting from setting a variable or register.

The viewable address ranges in a memory page is the "paging window." Memory paging is an accepted practice and a common way to access large memory ranges in computer systems.

9S08 devices with a flash memory greater than 60 KB include the Memory Management Unit (MMU) and the PPAGE register to access larger memory pages with the same 9S08 architecture (16-bit address pointer with a maximum of 64 KB addresses).

Contents

1 Fundamentals of Memory Paging in 9S08 Devices 1
1.1 Standard 9S08 Memory Map 2
1.2 Paged 9S08 Memory Map 2
2 The Memory Management Unit 3
2.1 The Program Page (PPAGE) 4
2.2 Linear Address Pointer and Linear Address Space 5
3 Window Paging Implementation 5
4 The Full Memory Map 7
5 Using Paged Memory and Linear Memory in CodeWarrior 8
5.1 Selecting the 9S08 Device Memory Mode 8
5.2 Modifying the PRM file. 8
5.3 Lab 1: Placing Code in a Specific Memory Section 9
5.4 Lab 2: Placing Data in the Extended Memory and the Usage of LAP 11
6 Conclusions 13
7 Reference Material 13

© Freescale Semiconductor, Inc., 2008. All rights reserved.

1.1 Standard 9S08 Memory Map

The standard 9S08 memory is represented as follows:

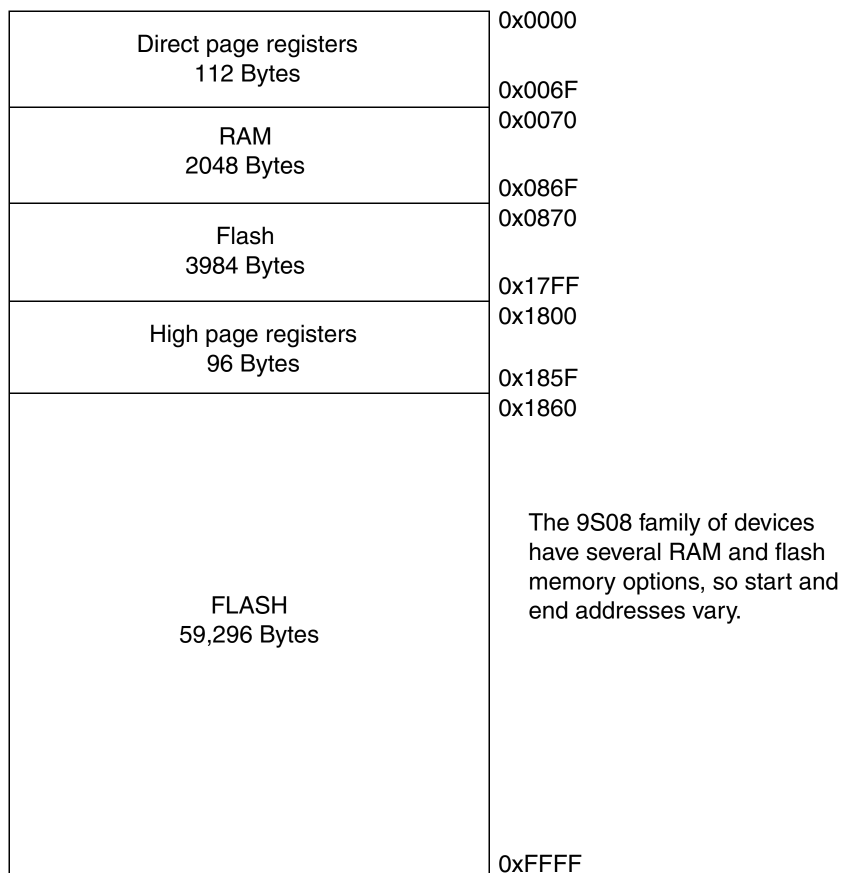

Figure 1. Standard 9S08 Memory Map

Figure 1 shows the 9S08AW60 memory map that has the standard linear memory used in 9S08 devices.

The complete memory map can be addressed with one 16-bit pointer in the 9S08 architecture. Expanding memory addressing to external memory for the 9S08 devices is not cost-efficient because 8-bit devices are commonly implemented in low cost systems.Therefore, adding a larger internal memory and a method to access memory beyond 64 KB addresses is required.

1.2 Paged 9S08 Memory Map

Having the 9S08 architecture limitation of a 16-bit pointer to memory, a paged memory scheme increases the addressing range to as many memory pages as available.

Figure 2 represents the basic memory structure that allows internal memory paging in 9S08 devices. By reorganizing the memory map, you can increase the number of accessible memory addresses.

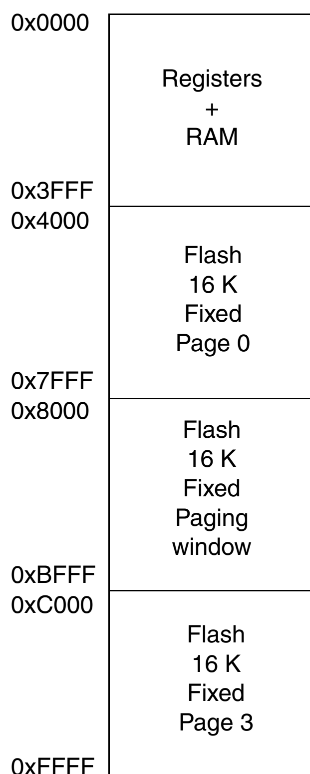

Figure 2. 9S08 Memory Paging Map

The reorganized memory map has several differences from the standard memory map:

— RAM and registers are grouped together at the beginning of the memory map
— Flash memory is divided into 16 KB sectors
— Each 16 KB sector is considered a page
— Special pages are considered “paging windows”

You can address the RAM and flash memory in this memory map with a 16-bit address pointer, but the remaining memory is not accessible.

2 The Memory Management Unit

Because the 9S08 architecture has limited addressing capacity, the MMU supports core accesses to the remaining internal memory that is not accessible with the 16-bit address pointer.

2.1 The Program Page (PPAGE)

The memory is divided into four sections: the first section is referred to as RAM even though it includes the registers, and the remaining three sections are referred to as flash.The flash sections are 16 KB each for a total of 48 KB flash. The rest of the flash memory is only accessible using the MMU.

The Program Page (PPAGE) register in the MMU allows you to view a 16 KB block in the paging window located between addresses 0x8000–0xBFFF. PPAGE is an 8-bit register containing a 3-bit field with valid values from 0 to 7.

The value in PPAGE is the currently addressable memory page in the paging window, which shows the address ranges of the accessible memory. Figure 3 shows PPAGE and its location within the memory pages:

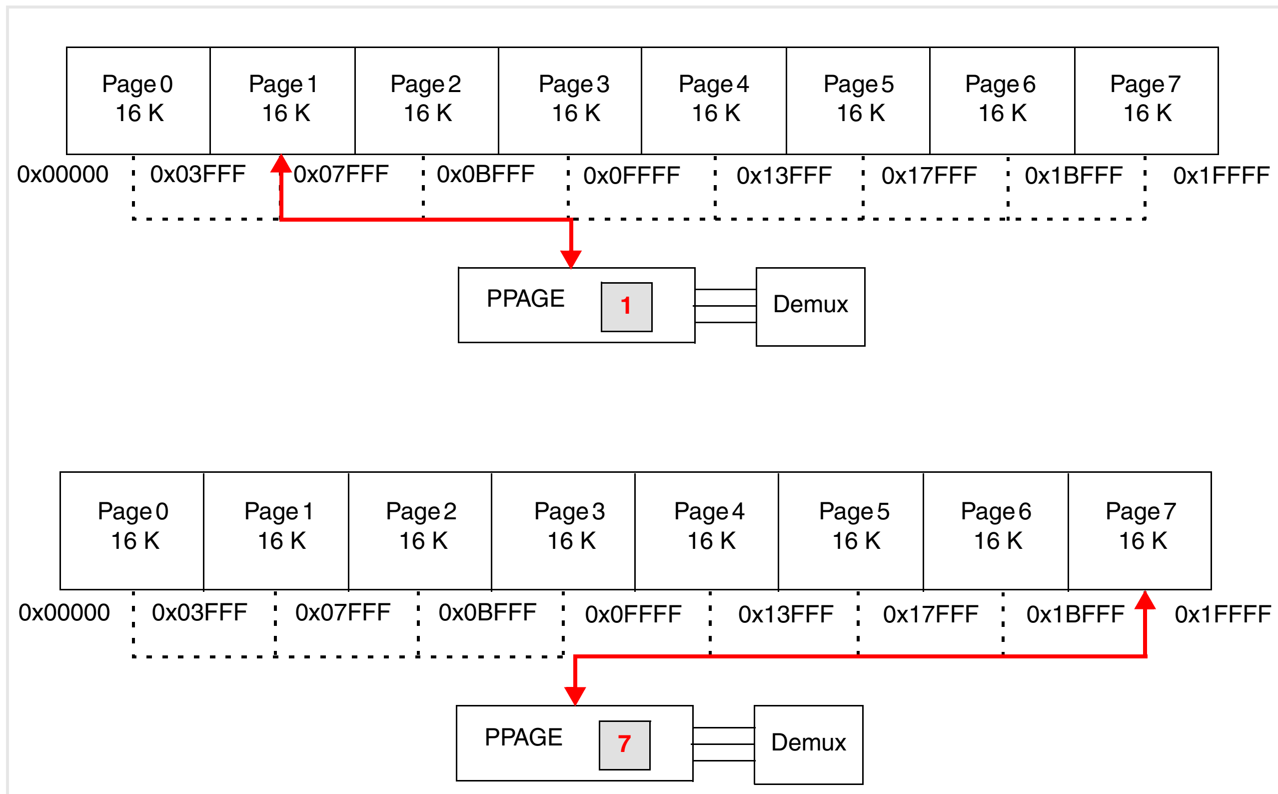

Figure 3. PPAGE and Memory Pages

The 16 KB memory page is the optimal page size for the 9S08 devices:

- A bigger memory page reduces the amount of unpaged memory used for system operations, such as interrupts and internal logic, which are required regardless of the PPAGE value.
- A smaller memory page requires more pages for the same amount of memory, which slows memory access and program performance.

To read and write data, set the page value in the PPAGE register and then access the memory between addresses 0x8000 and 0xBFFF. Every page is offset with the PPAGE value.

In code, the compiler and linker call functions according to the allocated memory location. The instructions used to execute the function depend on its location in the memory page:

- JSR jumps to code located in the standard 64 KB memory
- CALL accesses paged memory

Use the directives "__near" and "__far" to force the compiler and linker to specify the local or paged memory for a function. Jumping to functions in local memory takes less CPU cycles than jumping to paged memory. Given that local memory is smaller than paged memory, allocate higher priority functions or time critical functions to local memory; allocate functions with less priority to paged memory.

2.2 Linear Address Pointer and Linear Address Space

The Linear Address Pointer (LAP) is a 17-bit register that can directly access the full memory map. The LAP register contains three 8-bit registers; two complete 8-bit registers and one 8-bit register for the 17th bit. Use the LAP register and MMU module for direct access to the entire memory map.

The Linear Address Space (LAS) is the memory that the MMU can address linearly using the LAP register.

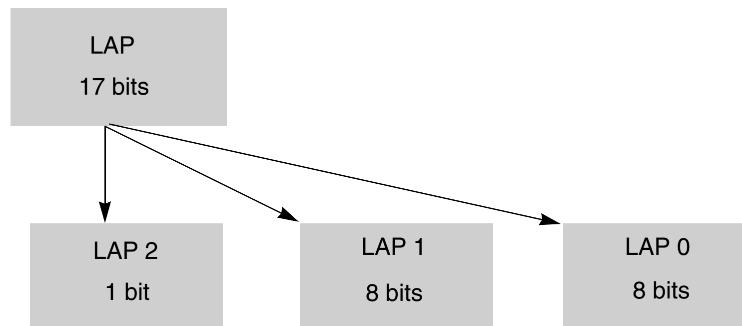

Figure 4. Linear Address Pointers

3 Window Paging Implementation

The LAP register addresses the full memory map. Since the MMU is a separate module and not part of the processor core, access to MMU registers is required to access the data to which the LAP points:

- Linear Byte register (LB)
  — 1 byte register
  — Read from or write to this byte to return the data to which the LAP register points
- Linear Byte Post Increment register (LBP)
  — 1 byte register
  — Read from or write to this byte to return the data to which the LAP register points; and increments the LAP register by one.

- Linear Word Post Increment register (LWP)
  - 2 byte register (data is overlapped by reading to the 16-bit HX).
  - Read from or write to this byte to return the data to which the LAP register points; and increments the LAP register by two.

The ability of LBP and LWP to increment the LAP register allows these registers to access large lists of data, as shown in the following examples:

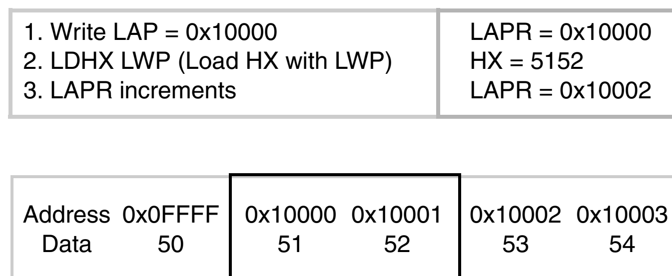

Figure 5. Using LWP to Access Data Tables

Working with data tables can require offsets to access the data. Write a two's complement byte value to the Linear Address Pointer Add Byte register (LAPAB) to increment or decrement the LAP by this value. This makes it easy to access from +127 to -128 addresses within the data range to which LAP points.

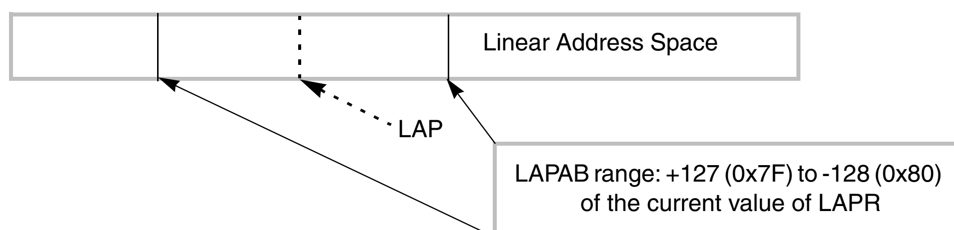

Figure 6. LAP Pointer Location in the LAPAB Address Range

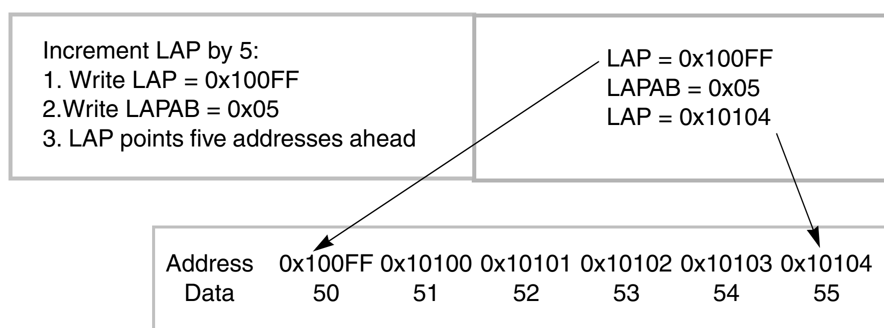

Figure 7. Using LAPAB to Access Data Tables

4 The Full Memory Map

Figure 8 represents the entire memory map for the 9S08 devices:

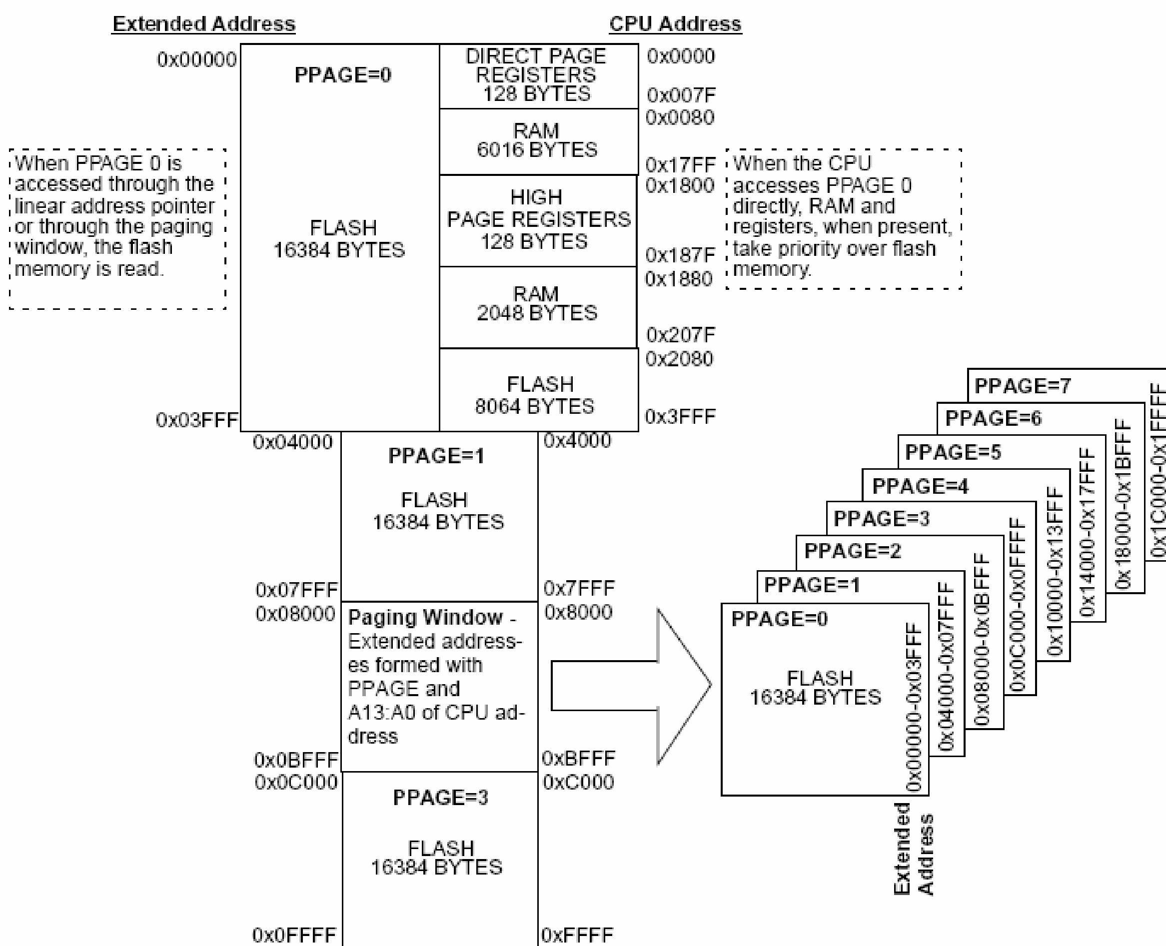

Figure 8. Complete 9S08 Memory Map

The extended address space, which is the actual physical address from 0x00000 to 0x1FFFF, is the complete 128 KB. 9S08 devices with smaller memories that implement paged memory are available, such as the 96 and 64 KB derivatives of the 9S08QE family and the 96 KB derivative of the 9S08AC family. Page 0 is the physical address 0x00000 and the CPU logical address 0x8000. The same principal applies to the remaining pages, according to the page number.

The following aspects are important to remember when developing code to access the full memory map:

- The RAM and flash memory (page 0) share CPU addresses 0x0000–0x3FFF. The CPU gives RAM priority access to this address range because it is accessible only by using the paging window or the linear addressing registers.
- Addresses 0x2080–0x3FFF is the final address range in page 0.
- The PPAGE default value is 2 at startup, which results in a contiguous memory block from page 1 to 3.

5 Using Paged Memory and Linear Memory in CodeWarrior

5.1 Selecting the 9S08 Device Memory Mode

When using the wizard in CodeWarrior to create a project, select from the following memory models:

- Tiny — By default, *all* variables reside in page 0 and require direct memory access. Use pragmas or the far keyword to access variables *not in* page 0.
- Small — By default, *no* variables reside in page 0 and require extended memory access. Use pragmas or the near keyword to access variables that reside in page 0.
- Banked — By default, functions require a special CALL instruction. No variables reside in page 0 and require extended memory access unless explicitly placed otherwise. Banked memory is only supported on devices with a MMU.

Banked memory is an option available on the HCS08 devices that uses the CALL and RTC instructions to the HCS08 core, in addition to the PPAGE memory mapping method.

Select banked memory option since the other two options are standard options for HCS08 devices with less than 64 K memory space.

5.2 Modifying the PRM file

The PRM file determines how the linker allocates the RAM and ROM memory. The standard procedure uses the default PRM file in the project window for the CPU derivative. However, you can modify the PRM file to change the memory allocation.

The following code shows the SEGMENTS section of a PRM file in a standard project for QE128.

```
Z_RAM     =  READ_WRITE   0x0080 TO 0x00FF;
RAM       =  READ_WRITE   0x0100 TO 0x17FF;
RAM1      =  READ_WRITE   0x1880 TO 0x207F;
/* unbanked FLASH ROM */
ROM       =  READ_ONLY    0x2080 TO 0x7FFF;
ROM1      =  READ_ONLY    0xC000 TO 0xFFAD;
/* INTVECTS  =  READ_ONLY    0xFFC0 TO 0xFFFF; */
/* Reserved for Interrupt Vectors */
/* banked FLASH ROM */
PPAGE_0   =  READ_ONLY    0x008000 TO 0x00A07F;
/* PAGE partially contained in ROM segment */
PPAGE_2   =  READ_ONLY    0x028000 TO 0x02BFFF;
PPAGE_4   =  READ_ONLY    0x048000 TO 0x04BFFF;
PPAGE_5   =  READ_ONLY    0x058000 TO 0x05BFFF;
PPAGE_6   =  READ_ONLY    0x068000 TO 0x06BFFF;
PPAGE_7   =  READ_ONLY    0x078000 TO 0x07BFFF;
/* PPAGE_1   =  READ_ONLY    0x018000 TO 0x01BFFF; */
/* PAGE already contained in segment at 0x4000-0x7FFF */
/* PPAGE_3   =  READ_ONLY    0x038000 TO 0x03BFFF; */
/* PAGE already contained in segment at 0xC000-0xFFFF */
```

The SEGMENTS section describes the memory blocks available to the MCU so the linker can allocate memory for different types of code. Each segment shows a contiguous block of memory locations with a qualifier and a range.

The ROM and ROM1 segments are not banked flash. The ROM segment includes addresses 0x2080–0x4000 of page 0 and all of page 1. Page 0 is also shown in the banked segment definition. This page is only defined at the lower addresses where the registers and RAM overlap. To access the overlapped part of Page 9 you must use the paging window (LAPR).

5.3 Lab 1: Placing Code in a Specific Memory Section

Project: MEM_banked.mcp

This lab shows you how to use the CALL and JSR call instructions and which RTC or RTS return instruction to use with the compiler, as well as how to point the linker to location for the code. The JSR and RTS instructions use a 16-bit address to access functions in memory less than 64 KB.

This lab creates a project with two simple functions:

- NearFunction() is located in non-banked ROM
- FarFunction() is located in the paged ROM

Use a CALL instruction to place the function in extended memory and the RTC instruction to return from the call. The CALL and RTC instructions use a 24-bit address. To place a function in banked memory, the keyword "__far" is required to qualify the function.

1. Launch CodeWarrior and use the wizard to create a project for MC9S08QE128
2. Select 'Banked' memory model
3. Create two functions, NearFunction() and FarFunction(), in the main.c file.
4. Add the "__near" keyword before function NearFunction() and its prototype
5. Add the "__far" keyword before function FarFunction() and its prototype
6. Call these two functions in the main function

   The following code shows the results of these steps:

```
Void __near NearFunction(void);
Void __far    FarFunction(void);
…
Void main(){
…
NearFunction();
FarFunction();
…
}
…
Void __near NearFunction(void){
}
Void __far FarFunction(void){
}
```

7. Disassemble the source file and study results of the compiler. NearFunction() is called by instruction CALL, and uses the return instruction is RTC. The function FarFunction() is called by instruction JSR, and uses the return instruction is RTS.

NOTE

Because functions in banked memory inherently require the '__far' keyword, you can remove it for the FarFunction().

TINY	SMALL	BANKED
• Functions are inherently __near if not specified otherwise • Functions in extended memory must be marked with __far (or #pragma CODE_SEG __FAR_SEG)	• Functions are inherently __near if not specified otherwise • Functions in extended memory must be marked with __far (or #pragma CODE_SEG __FAR_SEG)	• Functions are inherently __far if not specified otherwise • Functions that do not reside in extended memory (functions that use the standard calling convention) must be marked with __near (or #pragma CODE_SEG __NEAR_SEG)

8. Place function NearFunction() in the non-banked memory and FarFunction() in banked memory by using #pragma CODE_SEG convention. The following code shows the complete main.c file:

```
main.c

#include <hidef.h> /* for EnableInterrupts macro */
#include "derivative.h" /* include peripheral declarations */
#include <mmu_lda.h>

#pragma CODE_SEG NON_BANKED
void __near NearFunction(void);
#pragma CODE_SEG DEFAULT

#pragma CODE_SEG PAGED_ROM
void __far FarFunction(void);
#pragma CODE_SEG DEFAULT

void main(void) {

EnableInterrupts; /* enable interrupts */
/* include your code here */

NearFunction();
FarFunction();

for(;;) {
__RESET_WATCHDOG(); /* feeds the dog */
} /* loop forever */
/* please make sure that you never leave main */
}
#pragma CODE_SEG NON_BANKED
void __near NearFunction(void){
  }
#pragma CODE_SEG DEFAULT

#pragma CODE_SEG PAGED_ROM
void __far FarFunction(void){
  }
#pragma CODE_SEG DEFAULT
```

Make the project and review the .map file. The two functions are located at the correct memory address.

5.4 Lab 2: Placing Data in Extended Memory and Using LAP

With MMU support, the linear address pointer (LAP) register (LAP2:LAP0) and the linear data registers LB, LBP, LWP and LAPAB allow the CPU to read from or write to the entire memory space of HCS08 using a 17-bit linear address. Figure 9 shows the logical paged memory map and its linear address equivalent.

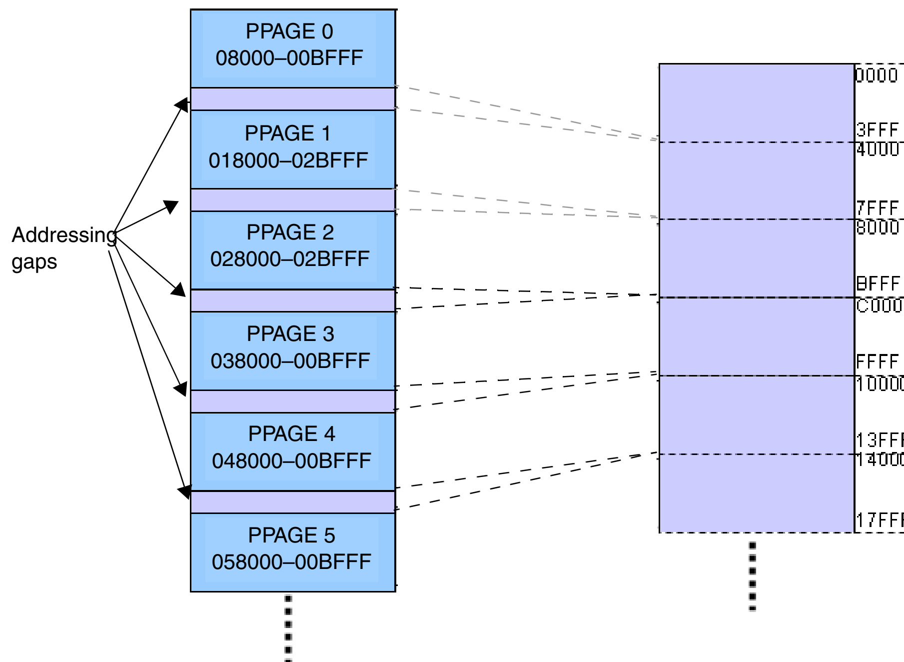

Figure 9. Paged Addresses and the Linear Address Equivalent

The equivalence formulas are:

- Linear_Address = (Logical_Address >> 16) * 0x4000 + (Logical_Address & 0xFFFF) – 0x8000
- Logical_Address = ((Linear_Address / 0x4000) << 16) | ((Linear_Address % 0x4000) + 0x8000)

This lab writes a constant string in the extended memory map (page 5, the equivalent linear memory range of 0x14000’F–0x17FFF’F), and then uses the LAP register to read the string. The following procedure puts a constant string and a constant character in extended memory:

1. Using CodeWarrior, create a project for MC9S08QE128 and select “banked” as the memory model.
2. In the .prm file, add the following text to the segments declaration:

```
ROM_LINEAR = READ_ONLY 0x014000'F TO 0x014FFF'F
```

3. Declare a placement within the ROM_LINEAR segment:

```
DATA_LINEAR INTO ROM_LINEAR
```

4. Add the following code to the main.c file to put the objects in section ROM_LINEAR

```
#pragma CONST_SEG __LINEAR_SEG DATA_LINEAR
const char MyString[255] = "Hello World !";
const char MyChar = 'A';
#pragma CONST_SEG DEFAULT
```

Use linear spacing for CONST_SEG and STRING_SEG only. Because two methods are available to specify the address locations for constants, a code overlap can occur. Code overlaps are reported in the debugger's command window but do not generate errors.

Command Window message: Code loading overlap detected in range 0x00058000 .. 0x0005BFFF

5. Add an ENTRIES field to the .prm file to force the linker to include the variable in the code.

```
ENTRIES
MyString
END
```

6. Make the project and review the .map file. The constant values MyString and MyChar are located in the linear space ROM_LINEAR. Use the LAP registers to read the data in extended memory.
7. Include the mmu_lda.h header file in the project. The macros are defined in this header file and are used to access and manage the LAP and MMU data registers.
8. Create a global pointer to a character value, and add the __linear keyword to it.

```
unsigned char * __linear LinearPointer;
```

The compiler uses the __linear keyword to declare 17-bit wide pointers. These pointers can store and manage the value of the LAP register, but they are *permanently* referenced, and the reference cannot be changed.

9. Create a local character variable LAP_Value.

The following code shows how to read data in extended memory:

```
*****************************************************************************
// load in LAP2:0 the linear address of string MyString
__asm  LDA  @MyString:LINEAR_HIGH     //load the high address of MyString to A
__asm  STA   LAP2                     //store high address of MyString to LAP2
__asm  LDA  @MyString:LINEAR_MID      //load the middle address of MyString to A
__asm  STA   LAP1                     //store middle address of MyString to LAP1
__asm  LDA  @MyString:LINEAR_LOW      //load the low address of MyString to A
__asm  STA   LAP0                     //store low address of MyString to LAP0
```

10. Read MyString using linear address data registers LB, LBP, LWP and LAPAB.

```
LAP_Value = LB;
```

This statement sets LAP_Value to the byte pointer in LAPR. After execution, the value of LAP_Value is 'H', the first character of string MyString. The value of LAPR does not change.

```
LAP_Value = LBP;
```

This statement sets LAP_Value to the byte pointer in LAPR. After execution, the value of LAP_Value is 'H', the first character of string MyString. the value of LAPR increases by 1.

Because the header file mmu_lda.h defined a set of macros to manage the LAPR and data registers, the following code can be used:

```
__LOAD_LAP_ADDRESS(MyString);   //load in LAP2:0 the linear address of MyString
__STORE_LAP(LinearPointer);
// Store the linear address in pointer LinearPointer __LOAD_LAP_ADDRESS(MyChar);
// load in LAP2:0 the linear address of MyChar __LOAD_BYTE_INC(LAP_Value);
// load a byte from the address in LAP2:0 and put it in
// LAP_Value
```

6 Conclusions

- The addressable memory a microcontroller can access can be expanded with a memory paging scheme.
- The Memory Management Unit (MMU) is a module that allows the 9S08 to access paged flash memory and extend the memory addressing capability of an S08 device.
- Pages are 16 KB.
- Memory can be accessed via a paging window. The page that can be accessed in the paging window is changed with the PPAGE register, a 3-bit register that can address up to 8 pages in the paging window. This is the preferred method to read and write code.
- The MMU includes registers that allow linear access to the full memory map as if it were not paged.
- The LAP (Linear Address Pointer) register is 17 bits and can address the full memory map.
- The Linear Byte (LB), Linear Byte Post Increment (LBP) and Linear Word Post Increment (LWP) registers can access the data to which LAP points.
- The Post Increment registers expand the capacity of the MMU by automatically increasing the address pointed to by LAP, making accessing data arrays easier.
- The Linear Address Pointer Add Byte Register (LAPAB) further improves the MMU addressing. A two's complement value written to this register increases or decreases the LAP register value by +127 to -128 addresses.

7 Reference Material

9S08QE128 data sheet ....Freescale Semiconductor

9S08AC128 data sheet ....Freescale Semiconductor

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

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)
www.freescale.com/support

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

Document Number: AN3730
Rev. 0
05/2008

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 that 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.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© Freescale Semiconductor, Inc. 2008. All rights reserved.

freescale™
semiconductor