Attachment 1

Partial Implementation of FreeRTOS on Freescale HC9S12C32 MCU Using M6811-ELF-GCC
# Table of Contents

Introduction......................................................................................................................... 3  
Implementation Approach .................................................................................................. 3  
HC9S12C32 MCU .............................................................................................................. 3  
  Overview......................................................................................................................... 3  
  Flash ................................................................................................................................ 4  
  Paging on Expanded-Mode S12 Devices ........................................................................ 6  
  RAM ............................................................................................................................... 7  
Hardware Development Environment ................................................................................ 8  
  Overview......................................................................................................................... 8  
  Hardware Setup............................................................................................................... 8  
Software Development Environment................................................................................ 10  
  Overview....................................................................................................................... 10  
  Compiler and Code Development Environment........................................................... 10  
  Setting up Eclipse ......................................................................................................... 10  
  Project Creation ............................................................................................................ 11  
    Introduction............................................................................................................... 11  
    Compiling for the HC9S12DP256 Target................................................................. 11  
    Project Compilation .................................................................................................. 13  
    HC12_Banked Demo Linker Files............................................................................ 16  
    Compiling for the HC9S12C32 Target ....................................................................... 22  
FreeRTOS Analysis on 9S12C32 ..................................................................................... 28
Introduction

This document describes partial implementation of the FreeRTOS on the Freescale HC9S12C32 MCU using the embedded GCC compiler M6811-elf-gcc. The HC9S12C32 MCU was mounted on a NanoCore12 board obtained from Technological Arts. This board (described shortly) provides the MCU with crystal, BDM and voltage regulation circuits while providing most of the pins from the MCU QFP package to a 32-pin DIP.

At the time of writing, the FreeRTOS code had been compiled to correctly fit into the tight memory spaces of the target platform. However, analysis of the FreeRTOS is pending the acquisition and implementation of debugging tools and will be added to this document at a later date.

Implementation Approach

FreeRTOS comes with a large suite of demo code that has been ported to many 8/16 bit targets through many compilers. This makes the RTOS attractive from a start-up and understanding point of view. Unfortunately, while demo code exists for the 9S12C32 (C32), it was compiled using a Code Warrior compiler. The thrust of this effort was to use embedded GCC. Therefore, there was no demo port of FreeRTOS immediately available. While this was certainly a disadvantage from the perspective of having a functioning RTOS immediately available for analysis, it did provide many opportunities to become familiar with the underlying hardware and compilation processes. This annex provides the results of those opportunities.

The implementation approach involved starting with a demo designed for a different HC12 target but compiled using GCC and modifying the code to be executable on the C32. The changes were not significant but did require significant understanding of the functional differences between various HC12 families as well as a detailed understanding of the embedded GCC compiler.

HC9S12C32 MCU

Overview

This section provides a brief overview of the memory configuration for the 9S12C32. An understanding of the paging mechanism within the Freescale HC12 family was key to configuring the compiler properly.

The 9S12C32 employs three types of memory: non-volatile Flash, EEPROM and RAM. Because there is relatively little RAM/Flash compared to other, larger devices, it is
critical to understand how it is located, organized, and managed at the hardware level in order to be able to configure the RTOS for most efficient use.

**Flash**

The flash memory-mapping scheme for the 9S12C32 depends on the specific device being used. The full identifier for the MCU is MC9S12C32MFA25. The appropriate entry from table E-1 in [12CREF06] is shown in the Figure A-1.

<table>
<thead>
<tr>
<th>MC9S12C32V8U52</th>
<th>xL45J / xLM84</th>
<th>-40°C, 105°C</th>
<th>80QFP</th>
<th>25MHz</th>
<th>CS2 die</th>
<th>32K</th>
<th>2K</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC9S12C32MA25</td>
<td>xL45J / xLM84</td>
<td>-40°C, 125°C</td>
<td>152QFP</td>
<td>30MHz</td>
<td>CS2 die</td>
<td>32K</td>
<td>2K</td>
<td>31</td>
</tr>
<tr>
<td>MC9S12C32MFA25</td>
<td>xL45J / xLM84</td>
<td>-40°C, 125°C</td>
<td>80QFP</td>
<td>25MHz</td>
<td>CS2 die</td>
<td>32K</td>
<td>2K</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure A-1: MC9S12C32MFA25 Characteristics

The figure confirms that the 9S12C32 being used is based on a 48 pin QFP, implements a 32K flash module, and has 2K of RAM. This information is significant. As shown in the Figure A-2 (figure 1-1 of [12CREF06]), the 48-pin QFP does not provide the data and address bus to external pins (these are identified in bold so they are only available in the 80 pin QFP).

Because the multiplexed data and address busses are not pinned-out, the C32 cannot operate in any of the expanded modes. The net result is that there is no possibility of
using memory paging (also referred to as banked memory) on this device in order to get more flash space than what is provided with the chip. Of course, this simplifies design requirements but also puts a limit on code size and requires the demo code to be edited to remove banking. It also has implications on the way in which the code is compiled. A brief overview of paging is provided in the next several paragraphs for comparison purposes.

The C32 implements the 32K flash module as described in Chapter 18 of [12CREF06] (S12FTS32KV1)). In general, the HS12 architecture places flash between 0x4000 and 0xFFFF which covers three 16K pages. The page space between 0x8000 to 0xBFFF is used in expanded mode to map to external flash memory pages. In the case of the C32, this page can be configured by the PPAGE register to be a duplicate of the lower flash block (0x3E) or the higher flash block (0x3F) as shown in the Figure A-3 below (Figure 18-2 from [12CREF06]).

![Figure A-3: Memory Map for the 9S12C32](image)

The PPAGE register determines whether pages 3E or 3F is mapped to the page window at reset. It turns out that, irrespective of what value is placed in the PPAGE register, if the value is ODD, then 0x3E will be duplicated in the page window while if the value is EVEN, then 0x3F will appear in the page window (see Table 1-11 in [12CREF06]).
Paging on Expanded-Mode S12 Devices

The HS12 core allows a maximum internal addressing space of up to 64K. Within this space, RAM, registers, EEPROM, and Flash blocks can be placed in a variable way (designated at boot time with some restrictions). However, the total FLASH can be increased to 1 MB on capable devices within this family by interfacing external memory components to the multiplexed data and address busses and making use of the PPAGE register. In order to "see" a particular 16K block of flash, the operating system code can "page it in" by setting a value in the PPAGE register and then reading within the page window ($8000 to $C000). The HS12 core performs an address translation and determines whether the location is on-chip or external. There are special addressing instructions to deal with memory accesses from one page to another. This scheme is shown in Figure A-4 below (Figure 4-1 from [MMC03]).

![Diagram: Paging Using PPAGE and 16K Block 0x800-0xBFFF](image)

Figure A-4: Paging Using PPAGE and 16K Block 0x800-0xBFFF

The use of off-chip Flash can only occur when the processor is not operating in single-chip mode because the address lines that would be used to communicate with the Flash will have been reassigned to other interfaces.
**RAM**

The 9S12C32 contains only 2K of RAM. At reset, this RAM is located at $0800-$FFFF but it can (and should) be re-mapped to a useful 2K boundary within the permissible address range for the MCU.

The overall memory map for the C32 is shown in the Figure A-5 below (Figure 1-5 of [FUG04]).

![Figure A-5](image)

The figure shows a useful map, which is not the map out of reset. After reset the map is:

- $0000 - $03FF: Register Space
- $0400 - $FFFF: 2K RAM

Flash Erase Sector Size is 512 Bytes.

**Figure 1-5 MC9S12C32 and MC9S12GC32 User Configurable Memory Map**

Figure A-5: RAM and Register Memory Map for HC9S12C32

Registers (I/O) are located starting at address 0x00.
Hardware Development Environment

Overview

This section describes the setup of the test instruments and target hardware environment. The software development environment is described separately.

Hardware Setup

The NanoCore12 device requires a few extraneous devices to enable communications and operate in the test environment. Figure A-6 shows this minimal set of components.

![Figure A-6: Circuit Diagram for C32 Additional Components](image)

It is not technically required to have an external voltage regulator since the device itself has a voltage regulator on board. However, with the current draw expected from external sensor components combined with the possibility of operating the device at 12V or more, it is anticipated that the on-board regulator will not have the heat dissipation capability required to operate safely. Therefore, a separate regulator has been added (low drop-out) to support future endeavors.

In order to reset the device, a reset push-button has been added. Communication with the device requires a serial port, a 9-Pin D-Shell connector wired to the device SPI as shown.

The overall test setup is shown in the Figure A-7.
Code was developed and compiled within the software development environment. After compiling and debugging, it was downloaded to the target using Karl Lunt's binary loader (binload) utility [BINLD]. Binload makes use of the serial monitor and debugger that ships with each MCU and which is located in a protected area of Flash. Binload can also be used interactively to perform some limited debugging on the target.

It should be noted that binload may perform a transform on the addresses of certain instructions as they are downloaded. According to the manual found at [BINLD], binload.exe will re-map any instruction address within the region $8000-$C000 to the region $4000-$8000. This re-mapping is performed because the serial monitor in the target is not capable of writing to the paged space ($8000-$C000). However, by default, the 9S12C32 resets with the least significant bit of PPAGE register 0. As discussed previously, this maps the lower page ($4000-$8000) into the paged space ($8000-$C000). The net result is a contiguous 32K flash memory space from $8000-$FFFF.

In order to use the serial debugger, the NanoCore must be put into the boot mode. This not only changes the clock frequency (boot mode enables the PLL bringing the device clock to 24MHz) but it also prevents the source code from running. In order to interrogate the device in operation, a MicroBDM12DX BDM Pod has been attached to the BDM port on the target.

In order to gather target performance metrics using signal outputs from the MCU, a logic analyzer was also connected (but not yet used).
Software Development Environment

Overview

This section describes the code development environment.

Compiler and Code Development Environment

The compiler being used is GCC 3.0.4 as part of release 2.2 of the GNU Development Chain for M6811/M6812 [GNU2.2]. The IDE is Eclipse Version 3.2 with the CDT.SDK-3.1.2 plug-in.

Initially, the Embedded GNU Development Environment written by Eric Engler was chosen for the target hardware platform. This IDE was attractive because it was simple, focused, and had certain built-in features to support the 9S12C32 (for example, binload.exe was integrated into the package, there was a default profile for the C32, etc). Unfortunately, it did carry a few disadvantages (e.g. all files had to be in a single folder). After some considerable testing and evaluation, Eclipse was adopted as the code development environment.

Downloading .S19 files to the 9S12C32 is still accomplished using Karl Lunt's binload utility.

Setting up Eclipse

Eclipse 3.2.2 was downloaded and installed to the root of C: (not surprisingly, in a directory called Eclipse). The default workspace is EclipseWS.

The C Development perspective was provided by the org.eclipse.cdt.sdk-3.1.2-win32.x86.zip plugin.
Project Creation

Introduction

As described earlier, no initial experience existed with the development environment (both Eclipse and the GNU development chain for the HC6812) so the implementation methodology was to begin with the code base from an existing demonstration and modify it to operate on the 9S12C32. The demonstration code chosen compiles using GCC for the HC9S12DP256.

The most utilitarian configuration for a FreeRTOS implementation was also chosen. This would be to have the FreeRTOS code initialize the hardware, start the scheduler, and initiate one task (besides the idle task).

Compiling for the HC9S12DP256 Target

As a first step in fully understanding the toolsets and compilation processes, no significant changes were initially made to the code base for the demonstration.

The first step was to create a project in Eclipse. A project called RTOSDEMO was created from the Eclipse menu. It was set up as a Standard Make C Project. An attempt was made to set the project up as a Managed Make C Project but the effort was eventually abandoned. A Standard Make C Project was deemed more suitable as a learning tool because it required the definition and control of a makefile.

Source code was then imported to the project using drag-and-drop. The entire FreeRTOS Source directory (including the portable and include subdirectories) was dragged into the RTOSDEMO project. In the Demo directory, all of the code within the HCS12_GCC_banked directory (and subdirectories) were then dragged into the RTOSDEMO project.

The resulting project file and folder tree is shown in the Figure A-8.
The Common folder (which is a part of all demonstrations) does not appear in this list. It is located at the same directory level as the HCS12_GCC_banked folder. An attempt was made to pull this folder into the RTOSDEMO project but significant difficulties were encountered. For reasons as yet undetermined, the linker consistently failed to find the Common directory even though the appropriate paths had been modified. The linker appeared to have a hardwired directory access that pointed it at ../..Common regardless of what was done within the project. By placing the Common directory at the same directory level as the RTOSDEMO project folder, the linker search error was resolved but not solved. In the final analysis, this error was deemed irrelevant anyway since the code in the Common directory was to be replaced eventually.

The Includes folder was populated by Eclipse as a result of having specified the discovery options as shown in the preferences tab in the Figure A-9.
Project Compilation

Getting the demo to compile required a thorough understanding of the build process and, to some degree, the files that were invoked to specify the memory layout for the device. This section provides some background information on compilation.

File Compilation

Figure A-10 represents the process of compiling a group of source files. Each source file is examined by m6811-elf-gcc and is converted into an object file. Each object file ends up defining a symbol table that contains a defined symbol for each defined function and variable. Each defined symbol consists of (at the very least) a name and an address. Each compiled object file consists of a number of sections, each having a name and a size. Some of the sections that can end up in an object file are shown in the Figure A-10.
File Linking [RHELDD]

Figure A-11 shows the process that the linker, (Ld) would use to "link" object files into an executable image. The linker examines each object file that has been identified as having something to contribute to the final executable image. The various sections of each object are then extracted and merged into the appropriate section of the executable (e.g. code found in .text1 and .text2 ends up in the .text section of the executable object file).

In addition to project files, the linker may also draw on other object files that have been compiled at an earlier time. In the figure, the linker extracts code from the crt0.o module.
Linking will occur if linker options are provided to GCC (options that follow the -Wl switch). When linking is specified, the linker will look for and use a linker script to assist it with the link process. If a linker script is not specified somewhere in the options sent to the linker executable, a default linker script is used. The default linker script can be tickled out of ld.exe using the following command:

```plaintext
m6811-elf-ld.exe -verbose >> deflinker.txt
```

The linker script is sometimes identified at compile time using the emulation switch -m such as in the following example:

```plaintext
>m6811-elf-gcc -g -Os -Wl,-m,m68hc11elfb -o hello.elf hello.c
```

The comma-delimited list following the -Wl switch are parameters to the linker. The linker script m68hc11elfb.x is identified by passing `-m,m68hc11elfb`. The other way to specify a non-default linker script is to use the `-T` option followed by the linker script name.

The various sections in the executable object must reside at a physical location on the target MPU. The linker script will look for a file called `memory.x` to provide this mapping. The memory mapping will be described shortly.

The linking process places the input sections into appropriate output sections. However, there is often a requirement to initialize certain sections of the MPU and these initializations will tend to be consistent with every application (e.g. setting up the stack pointer, performing operations that are required within the first 64 cycles, etc). This is taken care of by the default startup code `crt0.o` [GDEV05].

The `crt0.o` code creates four sections that are merged by the linker on the last pass and placed at the very start of the .text section [GDEV05]. These sections are:

- `.install[0]`: This is a reserved section that initializes the stack pointer.
- `.install[1]`: This is a section that provides a hook for application developers to install their own initialization code. Application initialization code can contain a section called `_premain()` and it will be inserted in `.install[1]`.
- `.install[2]`: This section is reserved for initializing the data and bss sections.
- `.install[3]`: This section is a placeholder for third party applications.
- `.install[4]`: This section is reserved to invokes main().

It is worth noting the actual code within the `crt0.o` file since the work done by `crt0.o` is not obvious to the un-initiated (the source for the object file normally does not show up in the code base anywhere). Figure A-12 provides the code for `crt0.o` found at [CRT0].
The symbols \_start, \_stack, and \_premain are referenced in this file. All of these should be defined before linking and we will see where this is done shortly.

If the code in crt0.o is objectionable, a private initialization script can be written entirely from scratch. In order to use it, the switch \texttt{-nostartfiles} must be provided to the linker (and crt0.o will not be used).

**HC12\_Banked Demo Linker Files**

The HC12\_Banked demo uses the linker script \texttt{ldscript-rtos.x}. It is invoked in place of the default link script by the following switch to the gcc command in the makefile:

```
LDFLAGS+=-$(CPU) -mshort -mlong-calls -WI,-T,ldscript-rtos.x
```
The following several lines of code have been extracted from the start of the `ldscript-rtos.x` file in the **HC12_Banked** demo:

```c
/* Linker script for MC689S12DP256 Flash rom banks.

Author Jefferson L Smith; Robotronics, Inc. 2006 */
OUTPUT_FORMAT("elf32-m68hc12", "elf32-m68hc12","elf32-m68hc12")
OUTPUT_ARCH(m68hc12)
ENTRY(_start)

/* Get memory banks definition from some user configuration file. This file must be located in some linker directory (search path with -L<dir>). See fixed memory banks emulation script. */

INCLUDE memory.x;
```

The **OUTPUT_FORMAT** command nominally provides three different BFD formats depending on command line options. In the present case, there is no difference in the format for any of the options. See [RHELLD] for details on BFD formats.

The **OUTPUT_ARCH** command identifies the m68hc12 as the architecture for the BFD library.

The **ENTRY** command identifies a symbol (`_start`) as the place to start code execution. The symbol `_start` was used in the `crt0.o` file to locate the start of the compiled code within `.install0`

The **INCLUDE** command directs `ldscript-rtos.x` to fetch and *in-line* the contents of `memory.x`. The `memory.x` file contains a **MEMORY** command which overrides the linker's default behaviour which is to assign all available memory.

The **MEMORY** command has the following syntax ([RHELLD]):

```c
MEMORY
{
    name [(attributes)] : ORIGIN = origin, LENGTH = length
}
```

The name is a symbol for a *region* of memory. Attributes identify what type of memory is found in that region: r=read only, w=read/write, x=executable, a=allocatable, etc.

The contents of the `memory.x` file for the HC12_Banked demo is listed in Figure A-13.
The various regions of memory in the HC9S12DP256 are defined in the MEMORY command. Each region is defined with a start address and a length and possibly some attributes of the region (read, write, execute). For example, the data region consists of volatile memory. The HC9S12DP256 has 12K of RAM starting at 0x1000. The text region is where program code is written and consists of FLASH memory. The DP256 has many regions of FLASH memory (in 16K pages) and these are all mapped individually. The last flash region (bank 15) is defined with a size of 16K less 100 bytes to accommodate the vector table.

The Page0 region is identified as the first 256 bytes of RAM and may have had meaning when optimizing for older MPUs like the HC11 (the first 256 bytes can be addressed with a single byte). There is an excellent discussion of the utility of the page0 concept in [GLINK06].

The PROVIDE directive forces the symbol _stack to have the physical starting address at the top of the available RAM. Recall that crt0.o used the symbol _stack to load the stack pointer. That operation won't work unless _stack is defined using PROVIDE.
The symbol `vectors_addr` is also defined to have the start address of the vector table.

The `memory.x` file uses the SECTIONS command to tell the linker how to map input sections (sections from within source object files) to output sections in the executable. Normally, sections are defined in the linker script (we shall see these shortly). If no SECTIONS command is provided at link time, sections in the input object files are mapped to the equivalently named output sections starting at memory address 0 ([RHELDD]).

The SECTIONS command has the following format ([RHELDD]):

```
SECTIONS
{
  sections-command
  sections-command
  ....
}
```

In turn, each `sections-command` has the following format ([RHELDD]):

```
section name :
{
  output-section-command
  output-section-command
  ....
} [>region] [=fillexp]
```

Each `output-section-command` can consist of a symbol assignment, a special keyword, some data values to be included, or (most commonly) a mapping to an input section from one or more source object files. This mapping can make use of wildcards to identify multiple files. Using these definitions, we can now identify how the SECTIONS command in `memory.x` works. For example:

```
.bank2 :
{
  ../Common/Minimal/flash.o(.text .rodata)
  *(.bank2)
} > bank2
```

The section name is .bank2. The `output-section-command` is an input section description. In this case, we are specifically mapping the .text and .rodata (read-only data) sections from the `flash.o` source object to the .bank2 section in the executable image. The second line maps all sections called .bank2 in all input source files (using the wildcard) to the .bank2 section in the executable image. The wildcard matches any input source object file.
Defining sections and mapping input sections to output sections does not yet identify where in memory the section will go. The image file must be statically linked to a group of destination addresses. To do this, the output section option “>” is used. The last line of the the .bank2 section maps the .bank2 section (and all the code that has been mapped to it) into the physical region of memory defined as bank2 by the earlier MEMORY command. Thus, .bank2 will start at physical address 0x8000.

After the memory.x file is included, the remainder of the ldscript-rtos.x file provides additional configuration. The ldscript-rtos.x file maps each section defined to specific memory locations if not already done and ensures that garbage collection at the end of linking does not sweep up sections defined in crt0.o.

The final element of the compilation process is the makefile. The modified makefile for the FreeRTOS HC12_GCC_Banked Demo is shown in Figure A-14.

Figure A-14: ldscript-rtos.x File From HC12_GCC_Banked Demo
The first two-thirds of the makefile are pre-occupied with identifying the source directories and building variables and prefixes for the various commands and switches. This makes things a considerably easier to read.

Some of the `ldscript-rtos.x` file was modified to make the compilation process work with the development environment. These changes were primarily limited to re-mapping the source directory paths and identifying some OBJDUMP switches to provide a more comprehensive .lst file for debugging.

OBJDUMP and OBJCOPY are binary utilities that come with the GNU GCC compiler. The CC command builds the file main.elf. This is used by objdump.exe to build a .lst file and, depending on the switches to that command, will configure the content of the .lst file. The objcopy.exe binary utility is used to convert the main.elf file to an S19 file format suitable for transmission by serial loaders.
Compiling for the HC9S12C32 Target

This section details all of the steps that were taken to implement a minimal FreeRTOS on the HC9S12C32 CPU. Note that this process is not yet complete.

- Create a new Eclipse "C Standard Make" project named RTOSDG128 (named after the DG128 in the event that the C32 target does not work out).
- Populate the new project with a clean FreeRTOS Source directory tree.
- The GCC_HC12_Banked demo source contains sys and asm-m68hcs12 directories that specify configuration specific information for the HCS12DP256. These directories were copied into the new project verbatim with the intent of modifying them to the C32 and/or DG128. Since I took out the SIO functionality, I needed to make the following changes immediately:
  - Comment out "void ATTR_INT ATTR_NEAR vCOM_ISR( void );" in Vectors.c;
  - Change the handlers for sci1_handler: and sci0_handler: to fatal_interrupt in Vectors.c
- The following source files were borrowed from GCC_HC12_Banked and modified to obtain a compiled executable:
  - Startup.c: Provides the _premain required from Crt0.o to initialize the target. Otherwise, it is fairly benign and can be “absorbed” by the main body of code on the next code update. It “includes” cpu.h and ports.h;
  - CPU.h: Some modifications were performed in this file. This file defines a number of attributes including those for interrupts and memory banks. Otherwise, it is fairly benign and can be “absorbed” by the main body of code on the next code update. It “includes” cpu.h and ports.h;
  - Param.h: No changes required;
  - Ports.h: Provides definitions for system ports. It also wraps the definitions so that they can be compiled for a 6811 or 6812 target using C or C++. Ports2.h “includes” ports_def.h.
  - Ports_def.h: Defines the port mapping for the DP256. Modified extensively to reflect the C32;
o Vectors.c: Contains an interrupt_vectors structure containing a table of pointers to ISRs.

o Interrups.h: No change;

o Interrups_dp256.h: To make this file work with the C32, re-mapping of the C32 interrupts onto the interrupts found in this file was required. Interrupt table information was obtained from Chapter 5 of the Device User's Guide for both the C32 and the DP256;

o main.c: All accesses to demo code was commented out. Main.c simply starts the scheduler now – although it will be required to start one other task in order to get the scheduler running properly;

o memory.x: Re-wrote from scratch and removed all references to using banked memory.

o FreeRTOSConfig: This is the main configuration file for the running RTOS. Significant changes to this file include:
   ▪ Commenting out the line “define BANKED_MODEL” which was inserted in the HS12_GCC_Banked demo for use by the compiler and linker;
   ▪ For this particular target, some difficulty was encountered with fitting the code into the available 2K memory space. However, Jefferson Smith (who assisted greatly in this venture) made the suggestion below concerning the setting of configTOTAL_HEAP_SIZE. On his advice, the heap size was sent to 1024 bytes and the compile/link process was successful.
   ▪ In order to accommodate main.c without an idle hook, the following change was required:

```
#define configUSE_IDLE_HOOK 0
```
   ▪ In order to keep the heap allocation within reasonable size limits, the following change was required:

```
#define configMINIMAL_STACK_SIZE( ( unsigned portSHORT ) 70)
```

o ldscript-rtos.x: The m68hc12elfb.x startup script was used instead. These files are not significantly different;

o makefile: The demo makefile was modified in several ways to suit the requirements of the conversion to the C32. In particular, Jefferson Smith noted that –mlong-calls could be removed if the banked memory model.
was not being used. And, as already mentioned, the –T option was used to change the linker script from ldscript-rtos.x to m68hc12elfb.x.

In addition to the changes listed above, a significant change to the default startup process was performed. On the 9S12C32, both RAM and register space needs to be defined early in the startup process. As shown in Figure A-5, the RAM bank is initially defined to start at 0x0800 which, as the Figure indicates, is not a useful address range. Therefore, the RAM bank needs to be re-defined (within the first 64 MCU clock cycles). This must be done before any other program code uses either the RAM or the stack (since the stack is within RAM).

Three methods of moving the RAM early in the startup were briefly investigated:

- Create a startup function as identified by Jefferson Smith [SJ06]. This doesn’t eliminate the startup script that already exists as part of the project because the function simply provides a code body to the application placeholder of crt0.o. Recall that crt0.o included a number of .install[] sections. The .install1 section was reserved for applications and exists just after the initialization of the stack pointer.
- Modify the existing startup script to perform the operations (as it does already to some extent).
- Borrow (verbatim) Eric Engler’s copy of crt0.o that has been directly modified to address the startup needs of the C32.

The first option was not pursued (although it would have been the most “elegant” solution) because it was 3 AM and looked harder than the others.

The second option (modifying the project startup script to set up the registers) was doomed before it was attempted. The startup script defines a body of code behind the _premain label which is inserted by crt0.o in the .install2 section. Unfortunately, as shown by the first two lines of the .lst file below, in order to get to the _premain code, a jsr instruction is executed. This instruction attempts to stack the return address and, although the stack pointer is correctly configured, the RAM for the stack has not been set up (which is what _premain was supposed to be doing.

```
00008000 <_start>:
  8000:  cf 40 00       lds    #4000 <_stack>
  8003:  16 80 37       jsr    8037 <__premain>
```

The next easiest thing to do was to borrow Eric’s code. This required the following changes to the makefile:

```
main.elf: $(OBJS) $(CC) $(LDFLAGS) -o $^ 9s12c32-crt0.o -nostartfiles -lc -lgcc
```

Eric’s replacement for crt0.o (9s12c32-crt0.o) is included as an object file to link in. To prevent the default crt0.o file from being used, the option –nostartfiles is also provided.
As an aside, while experimenting with Eric Engler’s Embedded GNU [EGNU07], it was discovered that libgcc did not natively provide `strncpy` and this was needed by `queue.c`. The solution was to ensure that libc was included and this is reflected in the linker directives in the makefile line shown above (`-lc -lgcc`).

Data Management

Variables that have been defined by the application at startup time are permanently stored in FLASH at the end of the `.text` section. This is done for a number of sensible reasons and is summed up best as follows:

"Placing data in ROM is useful to embedded applications: the data is always available and doesn't have to be generated at startup. Even more importantly, the data cannot get corrupted by an errant application, which reduces the number of considerations when debugging." [NES01]

A section of code in the linker script (`m68hc12elfb.x`) is responsible for putting the initialized data into FLASH at the end of the `.text` section. This code is shown in Figure A-15 below.

```
/* Start of the data section image in ROM. */
__data_image = ;
PROVIDE (__data_image = );
/* All read-only sections that normally go in PROM must be above.*/
We construct the DATA image section in PROM at end of all these
read-only sections. The data image must be copied at init time.
Refer to GNU Id, Section 3.6.8.2 Output Section LMA. */
.data : AT (__data_image)
{
__data_section_start = ;
PROVIDE (__data_section_start = );
*(sdata)
"(data)
*(data)
*(data)
*(gnu.linkonce.d")
CONSTRUCTORS
_data = ;
PROVIDE (_data = );
= ALIGN(2);
} > data =0xffffffff
```

Figure A-15: Initialization of Data in Flash

The label `_data_image` is defined as the current address pointer (using the `=` operator). The section of code in Figure A-15 appears immediately at the end of defining the `.text` sections so the address pointer will therefore be at the end of the `.text` section. This label is then made available using the PROVIDE command.

The `.data` section is then defined to be located at the start of `_data_image` using the AT command. All of the initialized data sections are then included in the `.data` section. Any
un-initialized portions of the *data region* are filled with $0xFF$ using the `$>data=0xffffffff$` command.

The affect of this code is reflected in the LMA and VMA addresses for the *data* section. The example in Section 3.6.8.2 of the ld manual [LDMAN07] points out that the LMA (logical memory address) and VMA (virtual memory address) are usually the same when the compiler executes and the *.lst* file is observed. They will often differ when referring to the data section. A section of a *.lst* file revealing this difference is shown Figure A-16.

The LMA for the data section points to the end of the *text* section in FLASH (0x9342). The VMA points to the actual start of RAM (0x3800).

Somehow, at boot time, the data found in FLASH must be manually moved to the start of RAM at 0x3800.

Happily, if `crt0.o` (or a derivative such as Eric Engler’s) is being used, as the default startup, this operation is handled. The last two lines of `crt0.o` (97 and 98) from Figure A-12 are:

93 ;-----------------------------------------
94 ; end startup code
95 ;-----------------------------------------
96 ;; Force loading of data section mapping and bss clear
97 .2byte __map_data_section
98 .2byte __init_bss_section
The \_map\_data\_section and \_init\_bss\_section code segments are inserted on an even byte boundary at the end of the startup code and handle moving the data FLASH to RAM as well as cleaning the .bss section. Figure A-17 shows a code snippet from a disassembled version of FreeRTOS clearly identifying the functions that move data from the end of the .text section to the .data section.

```
_start():
  0000:  cf 28 00  lds #2800 <_stack>
  0003:  16 80 39  jsr 0039 <__premain>
  0006:  cc 93 36  ldx #9336 <__data_image>
  0009:  ec 20 00  ldy #2000 <__data_section_start>
  000c:  cc 01 f0  ldd #f0f0 <__data_section_size>
  000f:  27 07  beq 0018 <__init\_bss\_section>

00000011 <Loop>:
  Loop():
  config/m68hc11/laixith.asm:1100
  8011:  10 0a 30 70  movb  1,x+, 1,y+
  config/m68hc11/laixith.asm:1101
  8015:  04 34 f9  dbne D,8011 <Loop>

00000018 <__init\_bss\_section>:
  __init\_bss\_section():
  config/m68hc11/laixith.asm:1125
  8018:  cc 07 65  ldd #765 <__bss\_size>
  config/m68hc11/laixith.asm:1126
  801b:  27 08  beq 8025 <Done>
  config/m68hc11/laixith.asm:1127
  801d:  ce 21 f0  ldx #21f0 <__edata>

00000020 <Loop>:
  Loop():
  config/m68hc11/laixith.asm:1130
  8020:  69 30  clr 1,x+
  config/m68hc11/laixith.asm:1131
  8022:  04 34 fb  dbne D,8020 <Loop>
```

Figure A-17: Disassembly Showing Initialized Data Movement

In the \_map\_data\_section(), the x register is loaded with the start address of the initialized data in FLASH (the LMA) while the y register is loaded with the start address of RAM (the VMA). The d register is loaded with the total number of data items to be moved. A loop is then entered in which data bytes are moved one at a time into RAM. Note that this section of disassembled code was intended for a DG128 target and, therefore, the RAM start address is at 0x2000.
FreeRTOS Analysis on 9S12C32

This section is pending the acquisition of the proper debugging tool support.
References


[CRT0] Configuration file crt0.o accessed from OpenSolaris.org March 07 located at: src.opensolaris.org/source/xref/sfw/usr/src/cmd/gcc/gcc-3.4.3/gcc/config/m68hc11/m68hc11-crt0.S


[FUG04] MC9S12C Family Device User Guide V1.05, 2004 (9S12C128DGV1.pdf)


