

## Application Note

AN2485/D  
8/2003

HCS12 Software Stationery



By Eduardo Montañez  
8/16 Bit Applications Engineering  
Austin, Texas

---

## Introduction

This application note describes a standardized software stationery that takes advantage of system on a chip (SoC) capabilities to make software development for the HCS12 Family of microcontrollers more efficient and reusable.

Defining each microcontroller peripheral individually according to SoC ideology provides a library complete with all necessary MCU parameter and definition files. This stationery architecture promotes:

- Increased version control
- Easy upgrades
- Pre-silicon development
- Reduced development time
- Easy portability/compatibility/adaptability
- Software reuse

With all the MCU definition files readily available, you can begin developing applications immediately.

The software stationery, *HCS12\_Stationery\_VX\_X.zip*, and an example serial communication project example using the stationery, *AN2485SW.zip*, are available from the Motorola website, <http://motorola.com/semiconductors>.

**NOTE:** *With the exception of mask set errata documents, if any other Motorola document contains information that conflicts with the information in the device data sheet, the data sheet should be considered to have the most current and correct data.*

---

## Overview

The HCS12 software stationery was developed using Metrowerks' CodeWarrior® tool for HC(S)12 MCUs (version 1.2 or greater<sup>1</sup>) and was written in the C programming language. The stationery displays all the necessary project files for your target MCU in a CodeWarrior (CW) project window.

The stationery is accessible through two unique methods:

- Mass MCU Stationery (\_MC9S12\_ALL.mcp) — CW project that includes all HCS12 Family definitions. You must follow MCU selection steps discussed in [Selecting a Target MCU \(Steps 9, 10, 11\)](#) to select which MCU peripheral and parameter files will be compiled for your defined application. One advantage of using the mass MCU stationery is increased portability. Using the CW project window, you can modify your application's target MCU with a few clicks.
- Single MCU Stationery (\_PartNumber\_Maskset.mcp) — Separate CW projects for each supported HCS12 MCU. Project includes only the definition files necessary for that device, so there is no need for the target MCU selection process. An advantage of this method is that there are fewer files in the project window.

This application note will refer to the mass MCU stationery method. However, all stationery functionality remains the same for the single MCU stationery, with the exception of the MCU selection steps described in [Selecting a Target MCU \(Steps 9, 10, 11\)](#).

The stationery was developed to use the CodeWarrior interface, but the stationery's software architecture is unique and can be modified to function on different embedded tools.

The SoC architecture of the stationery allows for easy upgrades to support growth in the HCS12 Family. Also, the SoC structure allows you to develop your application software for a targeted MCU prior to receiving silicon. This advantage can boost efficiency and allow you more time for debugging and improving your application. This concept will be explained in more detail in [System on a Chip Ideology](#).

---

1. CodeWarrior® is a registered trademark of Metrowerks, Inc., a wholly owned subsidiary of Motorola, Inc.

---

## System on a Chip Ideology

The HCS12 Family is based on system on a chip ideology. SoC allows the reuse of existing peripherals in multiple HCS12 MCU derivatives. SoC provides design with a convenient way to mix-and-match peripherals to easily customize an MCU. Therefore, as more and more HCS12 MCU derivatives are introduced, compatibility across the family of parts increases.

By standardizing the software peripheral definitions, the stationery follows the SoC method. Therefore, when you develop an application for a target MCU, your project will include only the necessary module definitions for that part. For example, when designing with an MC9S12DP256 (K79X) in mind, only the module definitions listed in [Table 1](#) would be included in the compilation:

**Table 1. MC9S12DP256 (K79X) Peripherals**

Peripheral	Version	Document Number	Header File
HCS12 V1.5 Core	V1.2	S12CPU15UG/D <sup>(1)</sup>	S12CPU15V1_2.h
CRG	V02	S12CRGV2/D	S12CRGV2.h
ECT_16B8C	V01	S12ECT16B8CV1/D	S12ECT16B8CV1.h
ATD_10B8C	V02	S12ATD10B8CV2/D	S12ATD10B8CV2.h
IIC	V02	S12IICV2/D	S12IICV2.h
SCI	V02	S12SCIV2/D	S12SCIV2.h
SPI	V02	S12SPIV2/D	S12SPIV2.h
PWM_8B8C	V01	S12PWM8B8CV1/D	S12PWM8B8CV1.h
FTS256K	V02	S12FTS256KV2/D	S12FTS256KV2.h
EETS4K	V02	S12EETS4KV2/D	S12EETS4KV2.h
BDLC	V01	S12BDLCV1/D	S12BDLCV1.h
MSCAN	V02	S12MSCANV2/D	S12MSCANV2.h
PIM_9DP256	V02	S12DP256PIMV2/D	S12DP256PIMV2.h

1. The HCS12 Family MCU Core User Guide has been parted into sovereign documents. The naming convention for header files will continue to be derived from the title of the collective core document as demonstrated. See [Naming Convention](#) for further explanation.

### *Naming Convention*

Notice in [Table 1](#) that each peripheral name and version number corresponds with the appropriate document order number. The HCS12 Family follows this naming convention to create a concise correlation between the peripheral and the corresponding documentation. In order to keep up with peripheral revisions, the stationery module definition files (also known as header files) maintain a standard naming convention, which directly reflects the equivalent documentation. For example, the S12SCIV2.h definition file reflects all the registers in the S12SCIV2/D block guide. The software stationery uses this as a form of version control. When you develop an application based on the HCS12 V1.5 Core V1.2, you will be well aware by the equivalent header file (S12CPU15V1\_2.h) that you do not have access to additional features like the on-chip debug module included in the new HCS12 V1.5 Core V1.5 (S12CPU15V1\_5.h). In addition, if you were to receive an MCU with a revised mask set, the software stationery would identify any revisions in the peripherals from a previous mask set — enabling you to correct any functionality differences in your software. Overall, the software stationery SoC method allows you to develop using only functionality available in your target MCU.

### *Early Development*

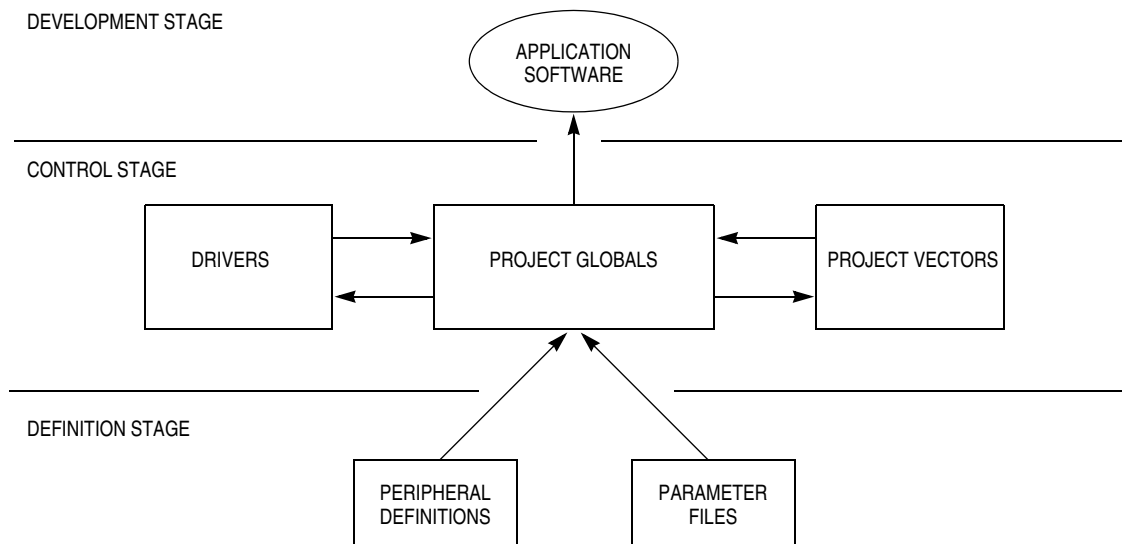
You can benefit from the SoC method when developing application software for a future MCU for which you have not yet received silicon. For example, if you wanted to program an application which involves using a 16-bit/ 8-channel enhanced capture timer (ECT), you could use the MC9S12DP256 (K79X) stationery to write and debug your software. In turn, when you receive your MC9S12D64 (L86D) silicon you can easily port your 16-bit, 8-channel ECT application that you had developed on the MC9S12DP256. In this example, both MCUs have identical ECT peripherals. The software stationery allows you to make easy transitions between MCUs by [Selecting a Target MCU \(Steps 9, 10, 11\)](#). The ability to change the target device for your development software on the fly will also allow for convenient reuse of existing software routines. Software written for a primary MCU can be easily transferred to a compatible target MCU.

## Software Stationery Architecture

The software stationery is composed of three stages with files that must be included in the compilation to access all the MCU features via software:

- Definition Stage — Contains files that are consistent to MCU documentation and need no modification by the user.
  - Peripheral definition files
  - Parameter files
- Control Stage — Contains files that require minimum user modification for access to stationery features.
  - Project globals file
  - Project vectors file
  - Driver files
- Development Stage — Contains files developed completely by the user using stationery resources.
  - Application software file

Each stage is essential to the stationery for developing a complete software application. **Figure 1** illustrates the three stages and the files creating the software stationery. The arrows represent where the files must be included for a successful software compilation.



**Figure 1. Software Stationery Architecture**

**Figure 1** demonstrates the software stationery's pyramid structure. The following sections will breakdown each individual stage and explain the role of each software file in the stationery's scheme. Also, the sections below will describe how each software file is created for a target MCU.

## Definition Stage

The definition stage is the backbone of the software stationery. This stage allows your software to access the hardware functionality of your MCU by associating labels with all register and bit field names from the MCU data sheet. The definition stage then sets these labels to the appropriate address and bit positions. The definition stage is composed of both peripheral definitions and parameter files. Both of these file types obey the HCS12 SoC architecture by defining a complete target MCU in software, dependent on the part number and mask set.

## Peripheral Definitions

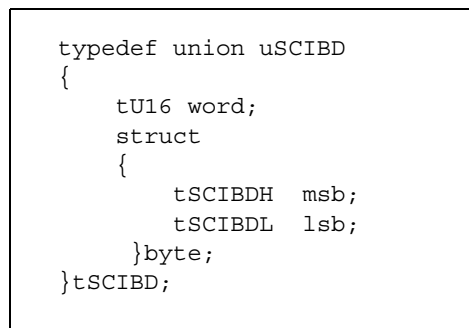
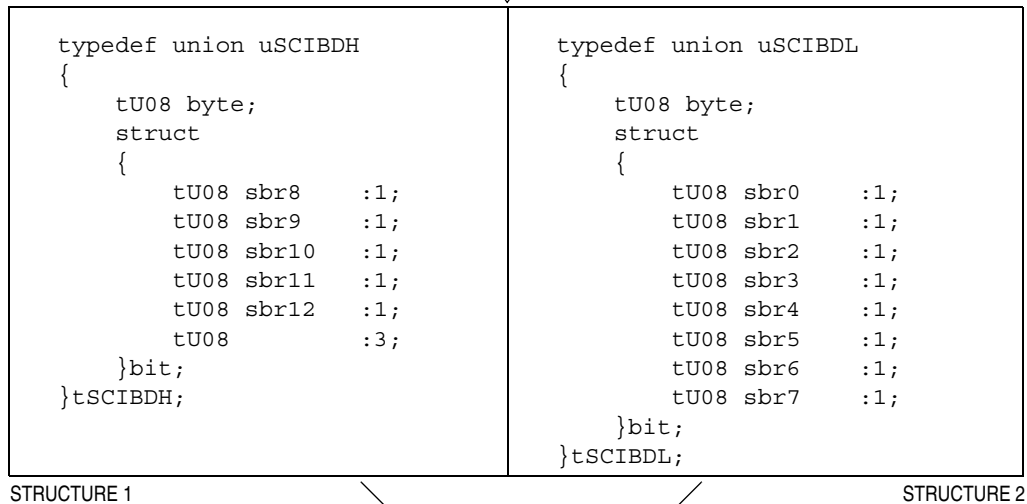
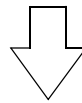
The first block in the definition stage contains the peripheral definitions, which is built in two levels:

- **Module Definitions (S12ModuleVersion.h)** — Create register-level definitions per module by using structures in C programming language. The sequential structures in header files also specify register locations. Structure definitions allow the user access to register space via software.
- **MCU Definitions** — (per\_partnumber\_maskset.c and per\_partnumber\_maskset.h) — Include only the necessary module definitions per part number and mask set for a single MCU. Develops a module memory map by defining the beginning address of each peripheral. The C source and header file are dependent on each other and contain almost identical information. However, the header files allow the peripheral structures to be accessible to the application software by making the peripheral definitions “extern.” (In the C programming language, the label “extern” makes an object accessible by other files.)

## Module Definitions

**Figure 2** illustrates the construction of a register structure in a module definition file based on the register level information provided by the MCU documentation. The example builds a C programming structure for the serial communication interface baud rate (SCIBD) register in the SCI peripheral. This particular example demonstrates three register structures. The first two structures are the SCIBDH (high byte — msb) and SCIBDL (low byte — lsb), which allow for both byte-wise and bit-wise access to the registers via software. The third structure concatenates SCIBDH and SCIBDL into a single SCIBD structure, enabling word access to the SCI baud rate register.

Addr. Offset	Register Name	Bit 7	6	5	4	3	2	1	Bit 0
\$_0	SCIBDH	Read: 0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8
		Write:							
		Reset: 0	0	0	0	0	0	0	0
\$_1	SCIBDL	Read: SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0
		Write:							
		Reset: 0	0	0	0	0	0	0	0



STRUCTURE 3

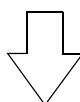
S12SCIV2.h

**Figure 2. Constructing the SCIBD Register Structure**

The module definition files characterize register structures for all peripheral registers with an identical format to [Figure 2](#). [Figure 2](#) only represents one 16-bit register in the SCI module definition file. Recall that structure 1 defines a byte-length register with three reserved bits. Next, structure 2 defines a byte-length register with all bits defined. Then, structure 3 concatenates structures 1 and 2 to define a word-length register.

To complete the module definition file, you must organize the registers within a larger structure sequentially (by address) after defining all the register structures for a peripheral. For example, the SCIBD register is the first of eight registers for the SCI peripheral. [Figure 3](#) shows an example structure organizing the SCI peripheral registers in sequential order.

Offset	Use	Access
\$_0	SCI baud rate register high (SCIBDH)	Read/Write
\$_1	SCI baud rate register low (SCIBDL)	Read/Write
\$_2	SCI baud rate register 1 (SCICR1)	Read/Write
\$_3	SCI baud rate register 2 (SCICR2)	Read/Write
\$_4	SCI status register 1 (SCISR1)	Read
\$_5	SCI status register 2 (SCISR2)	Read/Write
\$_6	SCI data register high (SCIDRH)	Read/Write
\$_7	SCI data register low (SCIDRL)	Read/Write



```
typedef struct                /*sci datastructure */
{
    volatile tSCIBD          scibd; /*sci baud rate registers */
    volatile tSCICR1          scicr1; /*sci control register 1 */
    volatile tSCICR2          scicr2; /*sci control register 2 */
    volatile tSCISR1          scisr1; /*sci status register 1*/
    volatile tSCISR2          scisr2; /*sci status register 2*/
    volatile tSCIDRH          scidrh; /*sci data register high */
    volatile tSCIDRL          scidrl; /*sci data register low*/
}tSCI;
```

S12SCIV2.h

**Figure 3. Sequentially Organizing the SCI Peripheral Registers**



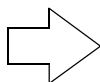
## MCU Definitions

After all the module definition files are complete, construct the MCU definition files. Recall that an MCU definition file includes the module definition files that make a particular MCU. It also specifies the peripheral's offset address from the documented module memory map. The SoC architecture allows you to include only the module definition files necessary to build an MCU by part number and mask set via software. [Figure 4](#) demonstrates how an MCU definition file is constructed from the module definition files for an MC9S12DP256 (K79X).

The MCU definition files mold the register space for all the supported MCUs. Note that in [Figure 4](#), only the peripheral definitions and header files pertaining to an MC9S12DP256 (K79X) are included in the MCU definition file. Also note that some peripherals are duplicated within MCUs. For example, an MC9S12DP256 (K79X) has two identical SCI peripherals. The peripheral definition structures accommodate the presence of multiple peripherals by allowing the structures to be renamed per peripheral occurrence. Renaming the structure definitions allows access to a single peripheral definition by multiple peripherals. Therefore, you can access each repeating peripheral individually — ensuring that you modify the correct peripheral by associating them with the appropriate name in the MCU documentation. SCI peripherals (Sci0 and Sci1) use the same tSCI structure defined by the SCI module definition file, but they access register space at two different memory locations. The MCU definition file shown in [Figure 4](#) is created for each new MCU mask set and completes the peripheral definitions block in the definition stage.

## MC9S12DP256 (K79X) Register Mapping

Address	Module
\$0000 – \$0017	CORE (Ports A, B, E, modes, inits, test)
\$0018 – \$0019	Reserved
\$001A – \$001B	Device ID register (PARTID)
\$001C – \$001F	CORE (MEMSIZ, IRQ, HPRI0)
\$0020 – \$0027	Reserved
\$0028 – \$002F	CORE (BDM)
\$0030 – \$0033	CORE (PPAGE, Port K)
\$0034 – \$003F	Clock and reset generator (PLL, RTI, COP)
\$0040 – \$007F	ECT 16-bit, 8 channels
\$0080 – \$009F	ATD 10-bit, 8 channels (ATD0)
\$00A0 – \$00C7	PWM 8-bit, 8 channels
\$00C8 – \$00CF	SCI0
\$00D0 – \$00D7	SCI1
\$00D8 – \$00DF	SPI0
\$00E0 – \$00E7	IIC
\$00E8 – \$00EF	BDLC
\$00F0 – \$00F7	SPI1
\$00F8 – \$00FF	SPI2
\$0100 – \$010F	FLASH control register
\$0110 – \$011B	EEPROM control register
\$011C – \$011F	Reserved
\$0120 – \$013F	ATD 10-bit, 8 channels (ATD1)
\$0140 – \$017F	CAN0
\$0180 – \$01BF	CAN1
\$01C0 – \$01FF	CAN2
\$0200 – \$023F	CAN3
\$0240 – \$027F	PIM
\$0280 – \$02BF	CAN4
\$02C0 – \$03FF	Reserved



```

ifndef REG_BASE
#define REG_BASE 0x0000
#endif

#include "S12ATD10B8CV2.h" //ATD
#include "S12BDLCV1.h" //BDLC
#include "S12CPU15V1_2.h" //CORE (PAGE/REG)
#include "S12CRGV2.h" //CRG
#include "S12EETS4KV2.h" //EEPROM
#include "S12FTS256KV2.h" //FLASH
#include "S12IICV2.h" //IIC
#include "MOTYPES.h" //TYPE DEFS
#include "S12MSCANV2.h" //MSCAN
#include "S12DP256PIMV2.h" //PIM
#include "S12PWM8B8CV1.h" //PWM
#include "S12SCIV2.h" //SCI
#include "S12SPIV2.h" //SPI
#include "S12ECT16B8CV1.h" //TIMER

```

```

extern tREGISTER Regs @ (0x0000 + REG_BASE);
extern tPAGE Page @ (0x0030 + REG_BASE);
extern tCRG Crg @ (0x0034 + REG_BASE);
extern tTIMER Tim0 @ (0x0040 + REG_BASE);
extern tATD Atd0 @ (0x0080 + REG_BASE);
extern tPWM Pwm @ (0x00A0 + REG_BASE);
extern tSCI Sci0 @ (0x00C8 + REG_BASE);
extern tSCI Sci1 @ (0x00D0 + REG_BASE);
extern tSPI Spi0 @ (0x00D8 + REG_BASE);
extern tIIC Iic @ (0x00E0 + REG_BASE);
extern tBDLC Bdlc @ (0x00E8 + REG_BASE);
extern tSPI Spi1 @ (0x00F0 + REG_BASE);
extern tSPI Spi2 @ (0x00F8 + REG_BASE);
extern tFLASH Flash @ (0x0100 + REG_BASE);
extern tEEPROM Eeprom @ (0x0110 + REG_BASE);
extern tATD Atd1 @ (0x0120 + REG_BASE);
extern tMSCAN Can0 @ (0x0140 + REG_BASE);
extern tMSCAN Can1 @ (0x0180 + REG_BASE);
extern tMSCAN Can2 @ (0x01C0 + REG_BASE);
extern tMSCAN Can3 @ (0x0200 + REG_BASE);
extern tPIM Pim @ (0x0240 + REG_BASE);
extern tMSCAN Can4 @ (0x0280 + REG_BASE);

```

per\_DP256\_K79X.h

Figure 4. Building an MCU Definition File for an HC9S12DP256 (K79X)

## Parameter Files

Parameter files:

- Specify memory locations for a target MCU.
- Provide the development tool linker with available memory locations to be programmed for your application software.
- Define an interrupt vector lookup table by assigning an interrupt handler label to each exception.

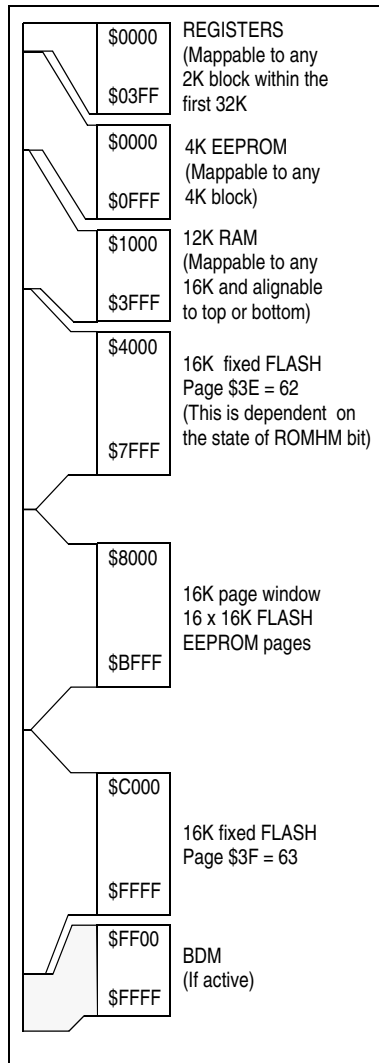
The parameter files block is composed of three types of parameter files per MCU part number. The three types of parameter files are:

- RAM parameter file (`_partnumber_RAM.prm`) — Specifies only RAM and EEPROM memory locations for a target MCU. No interrupt vector addresses are defined. File limits development of software applications to the maximum RAM size available in your target MCU.
- FLASH parameter file (`_partnumber_FLAT.prm`) — Specifies RAM, EEPROM, and non-banked FLASH memory locations for a target MCU. Defines an interrupt vector table. Intended for development of software applications with less than 48K of FLASH. (Does **not** use banked FLASH.)
- BANKED FLASH parameter file (`_partnumber_BANKED.prm`) — Specifies RAM, EEPROM, and banked FLASH memory locations for a target MCU. File defines an interrupt vector table. Most commonly used for development of software applications with more than 48K of FLASH and limits development to the maximum FLASH size available in your target MCU. File uses a PPAGE value, which allows access to multiple 16K FLASH windows. (Uses banked FLASH.)

## Memory Mapping

Even though there are three types of parameter files per MCU, only one is selected for use when developing your application software. Choose the parameter file according to the size and requirements of your application software. [Figure 5](#) demonstrates how an MC9S12DP256 memory map is translated into a BANKED FLASH parameter file.

MC9S12DP256 (K79X) MEMORY MAP



```

NAMES
END

SECTIONS
    RAM      = READ_WRITE 0x1000 TO 0x3FFF; /* 12K */

    EEPROM   = NO_INIT    0x0400 TO 0x0FFF /* 3K - first 1K hidden under registers */
              ALIGN 2 [<= 2: 2] [>2: 4]; /* default alignment = 2 */
              /* char & int assigned to x2 boundary */
              /* elements > 3 bytes to x4 boundary */
              /* < 100% use of EEPROM if byte data used */

    /* unbanked FLASH ROM */
    ROM_4000 = READ_ONLY 0x4000 TO 0x7FFF; /* 16K */
    ROM_C000 = READ_ONLY 0xC000 TO 0xFEFF; /* ~16K */
    SECURITY = READ_ONLY 0xFF00 TO 0xFF0F;
    ROM_FF10 = READ_ONLY 0xFF10 TO 0xFF7F;
    /* banked FLASH ROM */
    PAGE_30 = READ_ONLY 0x308000 TO 0x30BFFF;
    PAGE_31 = READ_ONLY 0x318000 TO 0x31BFFF;
    PAGE_32 = READ_ONLY 0x328000 TO 0x32BFFF;
    PAGE_33 = READ_ONLY 0x338000 TO 0x33BFFF;
    PAGE_34 = READ_ONLY 0x348000 TO 0x34BFFF;
    PAGE_35 = READ_ONLY 0x358000 TO 0x35BFFF;
    PAGE_36 = READ_ONLY 0x368000 TO 0x36BFFF;
    PAGE_37 = READ_ONLY 0x378000 TO 0x37BFFF;
    PAGE_38 = READ_ONLY 0x388000 TO 0x38BFFF;
    PAGE_39 = READ_ONLY 0x398000 TO 0x39BFFF;
    PAGE_3A = READ_ONLY 0x3A8000 TO 0x3ABFFF;
    PAGE_3B = READ_ONLY 0x3B8000 TO 0x3BBFFF;
    PAGE_3C = READ_ONLY 0x3C8000 TO 0x3CBFFF;
    PAGE_3D = READ_ONLY 0x3D8000 TO 0x3DBFFF;
    /* PAGE_3E = READ_ONLY 0x3E8000 TO 0x3EBFFF; not used: equivalent to MY_ROM_1 */
    /* PAGE_3F = READ_ONLY 0x3F8000 TO 0x3FBFFF; not used: equivalent to MY_ROM_2 */
END

PLACEMENT
    PRESTART, STARTUP,
    ROM_VAR, STRINGS,
    NON_BANKED, COPY
    DEFAULT_ROM
    INTO ROM_4000, ROM_C000, ROM_FF10;
    INTO PAGE_30, PAGE_31, PAGE_32, PAGE_33, PAGE_34, PAGE_35, PAGE_36,
    PAGE_37, PAGE_38, PAGE_39, PAGE_3A, PAGE_3B, PAGE_3C, PAGE_3D;

    DEFAULT_RAM
    INTO RAM;
    EEPROM_DATA
    INTO EEPROM;
END

STACKSIZE 0x0600 /* 1.5K bytes = 1/8 of RAM */

```

\_MC9S12Dx256\_BANKED.prm

Figure 5. Building an MCU Parameter File

Note that each individual memory location on the memory map is allocated in the parameter file (RAM, EEPROM, non-banked FLASH, and banked FLASH). Because this example shows a banked parameter file, each FLASH page window is defined to total 16 x 16K paged FLASH memory locations on the 256K FLASH device.

## Vector Table

To complete the parameter file, [Figure 6](#) illustrates how the interrupt vector lookup table is defined. Each exception is labeled with an interrupt service routine handler and defined with the appropriate vector address location.

MC9S12DP256 (K79X) Vector Table

Vector Address	Interrupt Source
\$FFFE, \$FFFF	Reset
\$FFFC, \$FFFD	Clock monitor fail reset
\$FFFA, \$FFFB	COP failure reset
\$FFF8, \$FFF9	Unimplemented instruction trap
\$FFF6, \$FFF7	SWI
\$FFF4, \$FFF5	XIRQ
\$FFF2, \$FFF3	IRQ
\$FFF0, \$FFF1	Real-time interrupt
\$FFEE, \$FFEF	Enhanced capture timer channel 0
\$FFEC, \$FFED	Enhanced capture timer channel 1
\$FFEA, \$FFEB	Enhanced capture timer channel 2
\$FFE8, \$FFE9	Enhanced capture timer channel 3
\$FFE6, \$FFE7	Enhanced capture timer channel 4
\$FFE4, \$FFE5	Enhanced capture timer channel 5
\$FFE2, \$FFE3	Enhanced capture timer channel 6
\$FFE0, \$FFE1	Enhanced capture timer channel 7
\$FFDE, \$FFDF	Enhanced capture timer overflow



```

VECTOR ADDRESS 0xFFFF _Startup          /* 0xFFFF Reset */
VECTOR ADDRESS 0xFFFC clockmonitor_isr    /* 0xFFFC Clock monitor fail reset */
VECTOR ADDRESS 0xFFFA cop_isr             /* 0xFFFA COP failure reset */
VECTOR ADDRESS 0xFFF8 trap_isr            /* 0xFFF8 Unimplemented instruction trap */
VECTOR ADDRESS 0xFFF6 swi_isr             /* 0xFFF6 SWI */
VECTOR ADDRESS 0xFFF4 xirq_isr            /* 0xFFF4 XIRQ */
VECTOR ADDRESS 0xFFF2 irq_isr             /* 0xFFF2 IRQ */
VECTOR ADDRESS 0xFFF0 rti_isr             /* 0xFFF0 real Time Interrupt */
VECTOR ADDRESS 0xFFEE ect_ch0_isr         /* 0xFFEE Timer channel 0 */
VECTOR ADDRESS 0xFFEC ect_ch1_isr         /* 0xFFEC Timer channel 1 */
VECTOR ADDRESS 0xFFEA ect_ch2_isr         /* 0xFFEA Timer channel 2 */
VECTOR ADDRESS 0xFFE8 ect_ch3_isr         /* 0xFFE8 Timer channel 3 */
VECTOR ADDRESS 0xFFE6 ect_ch4_isr         /* 0xFFE6 Timer channel 4 */
VECTOR ADDRESS 0xFFE4 ect_ch5_isr         /* 0xFFE4 Timer channel 5 */
VECTOR ADDRESS 0xFFE2 ect_ch6_isr         /* 0xFFE2 Timer channel 6 */
VECTOR ADDRESS 0xFFE0 ect_ch7_isr         /* 0xFFE0 Timer channel 7 */
VECTOR ADDRESS 0xFFDE ect_overflow_isr    /* 0xFFDE Timer overflow */

```

\_MC9S12Dx256\_BANKED.prm

Figure 6. Creating an Interrupt Vector Table for an MC9S12DP256 (K79X)

**Figure 6** only shows a portion of the MCU interrupt vectors for an MC9S12DP256 (K79X). Note that the labeled interrupt handler routines will be executed when the CPU requests the corresponding interrupt.

## Control Stage

The purpose of the control stage is to construct the software stationery. The stage provides a bridge between the development and definition stages. The settings defined in the control stage enable you to specify the definition files necessary for your application software. Also, this stage gives you flexibility to mold your projects by providing an area to define global parameters, macro definitions, driver inclusions, and interrupt service routines. The control stage is comprised of three blocks:

- Project globals
- Software drivers
- Project vectors

### *Project Globals*

The project globals block is made up of a single file labeled projectglobals.h, which gives you the option to select which MCU to include in the compilation with your application software (for mass MCU method only). Recall from the **Definition Stage** that an MCU definition file is created per MCU, which in turn includes each corresponding module definition file and the relative base address for each peripheral. **Figure 7** demonstrates the piece of software in projectglobals.h in which the user selects an MCU/maskset for software development (for mass MCU method only). For step-by-step instructions on how to set up your software application for development on a specific MCU, refer to **Developing Using the HCS12 Software Stationery**.

```

#ifdef Dx512_L00M
#include "per_Dx512_L00M.h"
#define Flash_Sector_Size 0x400
#endif /*Dx512_L00M*/

#ifdef DP256_K79X
#include "per_DP256_K79X.h"
#define Flash_Sector_Size 0x200
#endif /*DP256_K79X*/

#ifdef Dx256_L91N
#include "per_Dx256_L91N.h"
#define Flash_Sector_Size 0x200
#endif /*Dx256_L91N*/

#ifdef Dx128_L40K
#include "per_Dx128_L40K.h"
#define Flash_Sector_Size 0x200
#endif /*Dx128_L40K*/

#ifdef Dx64_L86D
#include "per_Dx64_L86D.h"
#define Flash_Sector_Size 0x200
#endif /*Dx64_L86D*/

#ifdef A512_L00M
#include "per_A512_L00M.h"
#define Flash_Sector_Size 0x400
#endif /*A512_L00M*/

#ifdef A256_L91N
#include "per_A256_L91N.h"
#define Flash_Sector_Size 0x200
#endif /*A256_L91N*/

#ifdef A128_L40K
#include "per_A128_L40K.h"
#define Flash_Sector_Size 0x200
#endif /*A128_L40K*/

#ifdef A64_L86D
#include "per_A64_L86D.h"
#define Flash_Sector_Size 0x200
#endif /*A64_L86D*/


#ifdef H256_K78X
#include "per_H256_K78X.h"
#define Flash_Sector_Size 0x200
#endif /*H256_K78X*/

#ifdef H128_K78X
#include "per_H128_K78X.h"
#define Flash_Sector_Size 0x200
#endif /*H128_K78X*/

#ifdef E128_L15P
#include "per_E128_L15P.h"
#define Flash_Sector_Size 0x400
#endif /*E128_L15P*/

#ifdef C32_L45J
#include "per_C32_L45J.h"
#define Flash_Sector_Size 0x200
#endif /*C32_L45J*/

```



```

//*****
//* MCU_Maskset Selection:
//*****

/*D-Family*/
//#define Dx512_L00M
//#define DP256_K79X
//#define Dx256_L91N
//#define Dx128_L40K
//#define Dx64_L86D

/*A-Family*/
//#define A512_L00M
//#define A256_L91N
//#define A128_L40K
//#define A64_L86D

/*H-Family*/
//#define H256_K78X
//#define H128_K78X

/*E-Family*/
//#define E128_L15P

/*C-Family*/
//#define C32_L45J

```

To define the device and mask set for your application, remove the comment indicator ("//") in front of "#define".

projectglobals.h

**Figure 7. MCU/Mask Set Selection**

The second function of the projectglobals.h file allows you to identify any global parameters or macro definitions. Since the definition stage is standardized and should remain unmodified until any module revisions, the control stage is used to develop additional definitions directly related to the application software and not the MCU. **Figure 8** illustrates some example macro definitions that would be created within the projectglobals.h file.

---

```

/*****
/*Macro Definitions
/*****

#define int_enable() {asm andcc #0xEF;} //interrupts enabled
#define int_disable() {asm orcc #0x10;} //interrupts disabled
#define wait() {asm wait;} //enter wait mode
#define stop_enable() {asm andcc #0x7F;} //stop mode enabled
#define stop() {asm stop;} //enter stop mode
#define nop() {asm nop;} //enter NOP asm instruction
#define bgnd() {asm bgnd; asm nop;} //enter BGND asm instruction
#define ON 1 //ON
#define OFF 0 //OFF

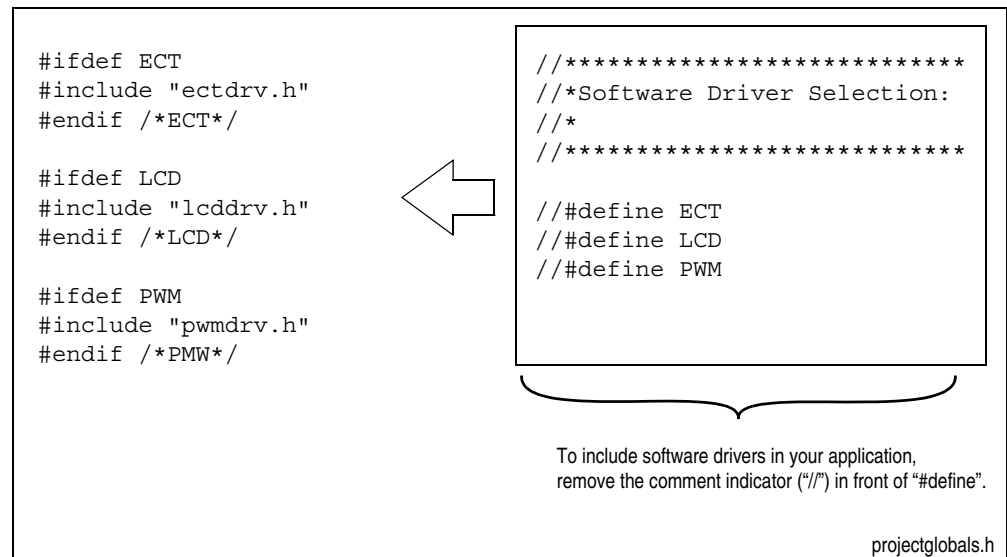
```

---

**Figure 8. Macro Definition Samples**

The third function of the projectglobals.h file allows you to add software drivers to the compilation of your application software. The addition of software drivers is not necessary in all applications, but an area in the projectglobals.h file is allocated for the selection of software drivers, if available. The majority of software drivers included in an application serve as low-level drivers made up of module initialization subroutines, which set up the MCU modules so they can function when executing the application. **Figure 9** demonstrates how a software driver can be included in the application. For more information on how to develop using additional software drivers, refer to [Developing Using the HCS12 Software Stationery](#).





**Figure 9. Software Driver Selection Examples**

### Software Drivers

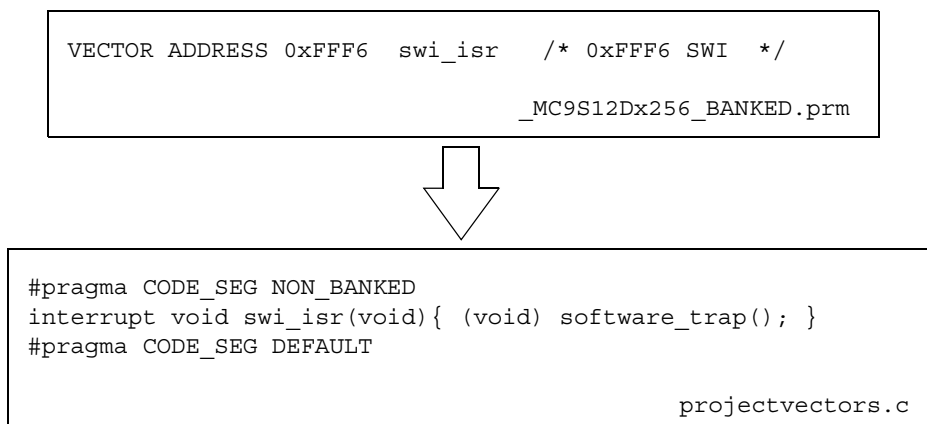
The drivers block is directly related to the software driver selection in the project globals block. It provides an accessible directory in the stationery architecture to maintain a library of software drivers, which can be included in your application software via the projectglobals.h file as mentioned in [Project Globals](#). Once the driver is included in the application project, you can access any driver subroutine.

### Project Vectors

The project vectors block stores both projectvectors.c and projectvectors.h files. These two files establish an interrupt vector handler corresponding to all the supported HCS12 MCU exceptions declared in the parameter files (when using the mass MCU method). In single MCU method, only the vectors pertaining to that MCU are included. Usually, interrupt vector handlers would be labeled as an unused interrupt service routine (unused\_isr) or a dummy interrupt service routine (dummy\_isr) if they were not applied in the application software. These routines would not execute a trap in order to capture an unexpected interrupt and would be left to you to declare.

This stationery predefines all the interrupt vector handler labels per MCU. Therefore, the stationery reduces the amount of user effort to re-label the interrupts for different software applications. Also, the stationery places these interrupt vector handlers within non-banked memory so they can be accessed regardless of the executing banked address. Refer to [Figure 10](#) for an example interrupt service handler declared in a parameter file and made accessible within the projectvectors.c file. Note that the interrupt vector handler in [Figure 10](#) is unimplemented. Therefore the interrupt includes only a subroutine

labeled `software_trap ( )`, which executes a trap if it unexpectedly occurs. This routine is a safeguard to prevent software runaway. For more information on how to set up an interrupt service routine in your software application, refer to [Developing Using the HCS12 Software Stationery](#).



**Figure 10. Example Unimplemented Interrupt Service Handler**

## Development Stage

The development stage allows you the ability to focus strictly on developing your application. With the standardized module definitions in the definition stage and the predetermined global settings in the control stage, the user can begin development almost immediately. This stage initially includes only a single `main.c` file, which is made accessible to build routines and functionality for the target MCU. Due to the stationery architecture, all module registers are readily accessible in the development stage.

For a complete software project example using all the features in the HCS12 Software Stationery, refer to [Developing Using the HCS12 Software Stationery](#).

## Developing Using the HCS12 Software Stationery

This section will provide:

- Step-by-step instructions on how to set up the stationery
- Detailed steps on how to develop an application using the stationery
- Example using the stationery's features (low-level drivers, global variables, interrupt vectors, macro definitions, and register accesses) to write a serial communication software example
- Demonstration on how to port your application software from one target MCU to another target MCU

### Setting Up the Stationery

The steps below assume that Metrowerks' CodeWarrior for HC(S)12 (version 1.2 or greater) is installed on your computer.

1. Download the HCS12 software stationery.
2. Extract HCS12\_Stationery.zip file into any directory.
3. Open the generated HCS12\_Stationery root directory.

**NOTE:** Steps 4-6 are not necessary. However, they allow the user to create a working directory while preserving the stationery template for subsequent applications.

4. Create a new project using either the mass or single MCU stationery method by duplicating (copy and paste) the appropriate directory within the HCS12\_Stationery root directory.
  - Mass MCU - duplicate the \_MC9S12\_ALL directory generating a working copy labeled Copy of \_MC9S12\_ALL.
  - Single MCU - duplicate the \_PartNumber\_Maskset directory generating a working copy labeled Copy of \_PartNumber\_Maskset.
5. Rename the new project directory (Copy of \_MC9S12\_ALL or Copy of \_PartNumber\_Maskset) to a name more appropriate to your project. (ex. ProjectName\_PartNumber -> MotorControlDemo\_E128)
6. Open the new project directory (ProjectName\_Partnumber -> MotorControlDemo\_E128) and rename the \*.mcp file to match the new project directory name (ex. ProjectName\_PartNumber.mcp -> MotorControlDemo\_E128.mcp)
7. Open the \*.mcp file (CW project file) in the new project directory (if you executed Steps 4-6) or in the original Mass or Single MCU directories as shown in Fig. 11. This will load a ready-to-go template project using the stationery (Mass or Single MCU) that you selected.
8. Select build target among RAM, FLASH and BANKED applications from the drop-down menu at the top of the CW project window.

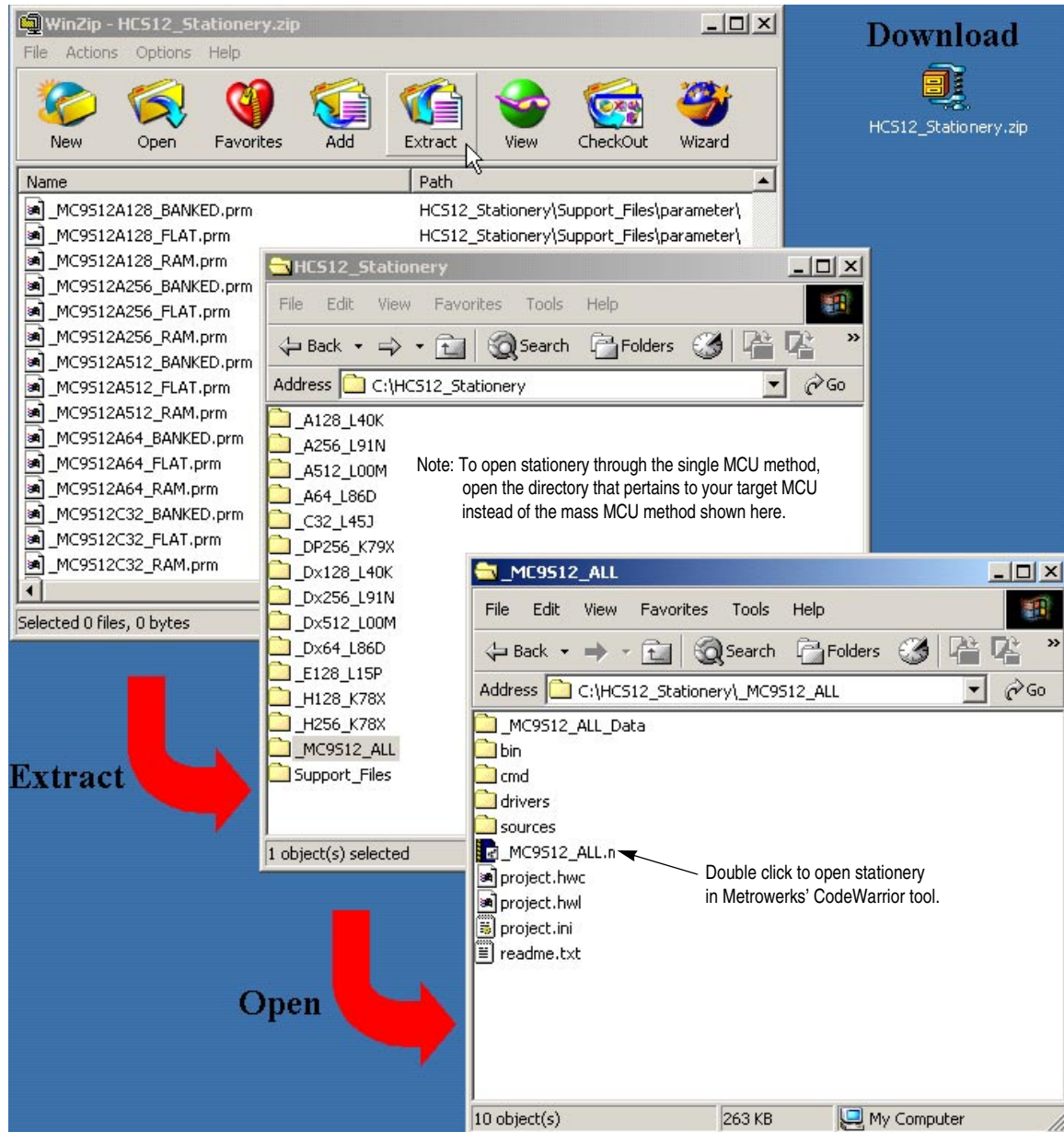
**NOTE:** *Steps 9 through 11 are not necessary in the single MCU stationery method. Selections are already configured.*

9. Select the target MCU within the projectglobals.h file under the MCU/mask set selection area by removing the “//” in front of the #define statement of your MCU.
10. Select the target MCU peripheral file by clicking under the red bull’s eye column next to your desired MCU peripheral file (included file is marked by a bullet) in the CW project window.
11. Select the target MCU parameter file by clicking under the red bull’s eye column next to your desired MCU parameter file (included file is marked by a bullet) in the CW project window.
12. Begin application software development within the unimplemented main.c file. Interrupt vectors can be added within the projectvectors.c file.

*Download, Extract,  
and Open*  
(Steps 1, 2, 3)

The first three steps of setting up the stationery consist of attaining and placing the stationery on the computer that you use for software development.

**Figure 11** illustrates the first three steps.



**Figure 11. Downloading, Extracting, and Opening Stationery**

Note that by extracting the HCS12\_Stationery.zip, a root folder labeled HCS12\_Stationery is created with the following subdirectories:

- \_MC9S12\_All directory — Contains main project source files for mass MCU stationery method (\*.c and \*.mcp)
- \_PartNumber\_Maskset — Contains main project source files for single MCU stationery method (\*.c and \*.mcp)
- Support\_Files directory — Contains the following subdirectories
  - Definitions directory — Contains all module definition files (\*.h files)
  - Documentation directory — Contains stationery documentation (guides)
  - Parameter directory — Contains all MCU parameter files (\*.prm files)
  - Peripherals directory — Contains all MCU definition files (\*.h files)

*Duplicate Template  
(Optional Steps 4–6)*

If you decide to duplicate the stationery template to preserve it for subsequent use, follow the steps detailed in steps 4–6 of [Setting Up the Stationery](#). The examples and instructions in this document refer to files by their original template names. These names may differ from the names used in your project if you follow these steps.

*Open the Project File  
(Step 7)*

By double-clicking on the MC9S12\_All.mcp file (mass MCU method) in the HCS12\_Stationery\\_MC9S12\_All directory, the stationery opens in the Metrowerks' CodeWarrior environment and brings up the main project window and displays all necessary global, source, peripheral, parameter, and definition files needed to develop your application (as shown in [Figure 12](#)).

Recall that the \*.mcp file that you double-click to open the project might differ in name or location if you follow the optional steps 4–6.

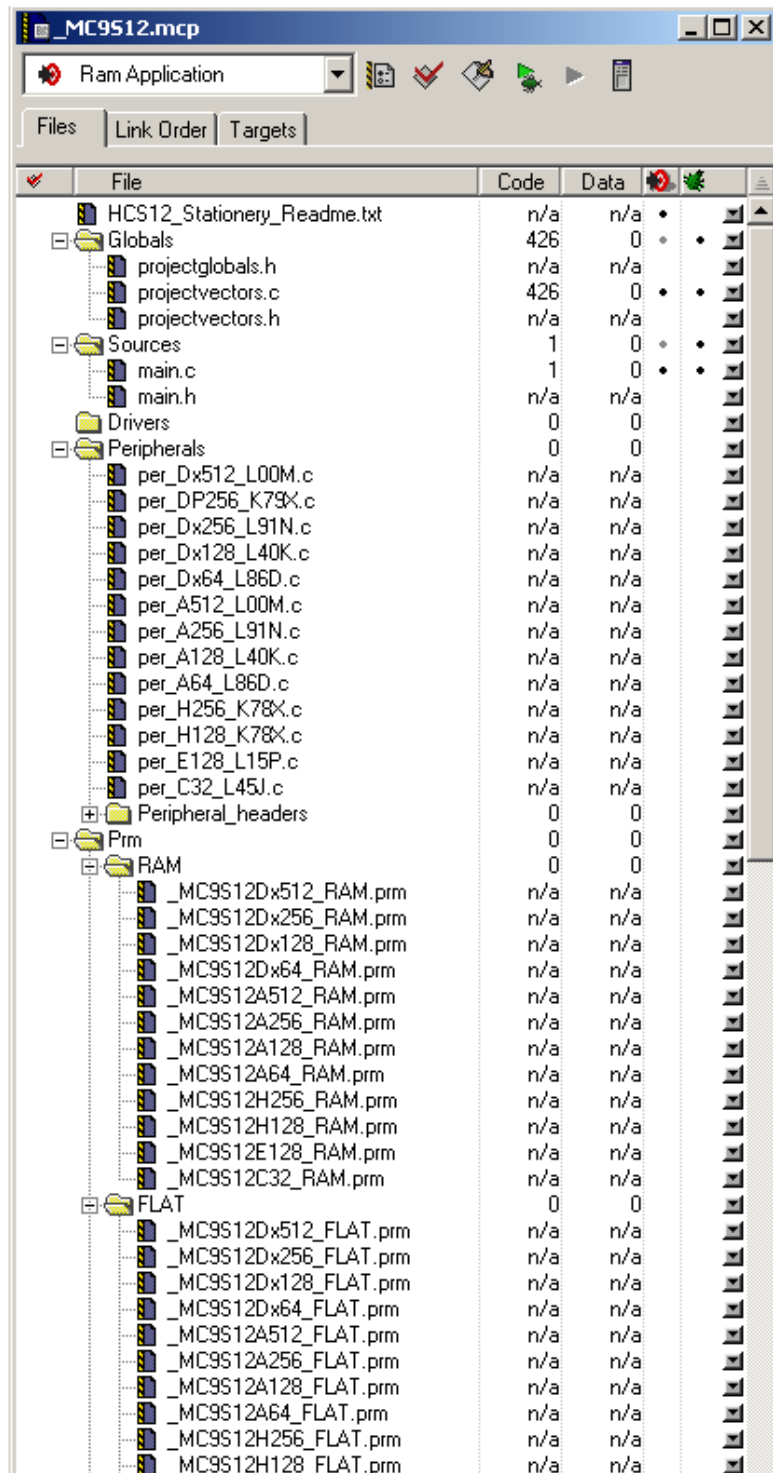


Figure 12. CodeWarrior Main Project Window

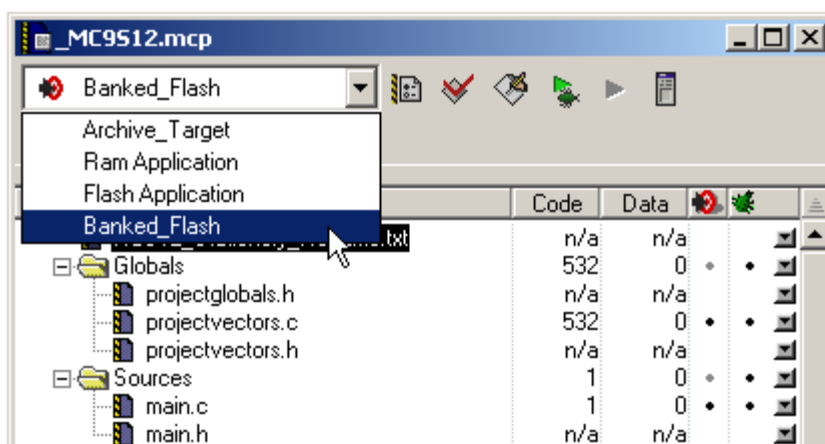
### Selecting a Build Target (Step 8)

Once you have established the stationery in your desired directory and have launched the project in the Metrowerks' CodeWarrior environment, you can focus on selecting the correct files to develop for your target MCU. Step 4 requires that you select the desired build target for your application. A build target is the intended memory space for which the compiler will build your software. When selecting a build target, you should match the target with the corresponding parameter file. For example, the available build targets listed below match up with the parameter files associated next to them.

- Archive\_Target — None (used as a null target and not used for development)
- Ram application — RAM parameter file (\_partnumber\_RAM.prm)
- Flash application — FLASH parameter file (\_partnumber\_FLAT.prm)
- Banked\_Flash — BANKED FLASH parameter file (\_partnumber\_BANKED.prm)

To determine which build target best fits your application, refer to [Parameter Files](#).

**Figure 13** demonstrates how to select your build target via the Metrowerks' CodeWarrior main project window. Note that the build target is selected via a pull-down menu.



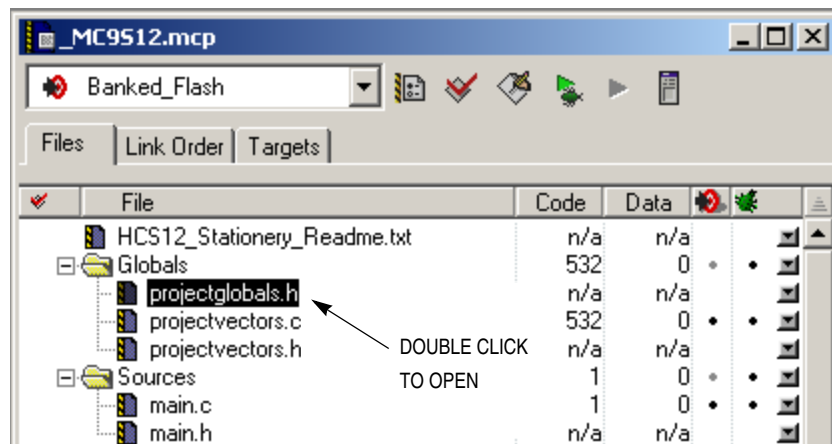
**Figure 13. Selecting a Build Target**



### Selecting a Target MCU (Steps 9, 10, 11)

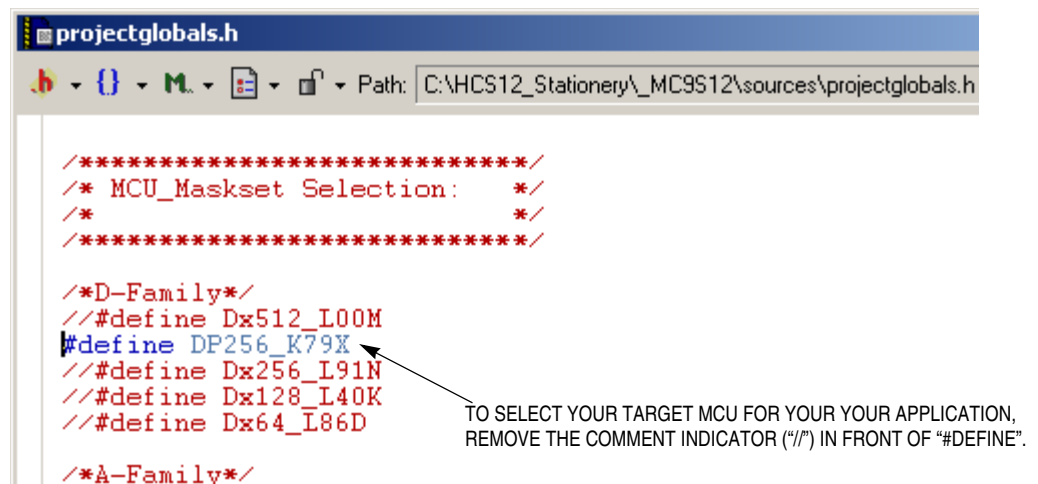
Following the selection of the build target, you must set up the stationery to include all necessary definition files for your application's target device. To complete step 5, you must select your MCU part number and mask set within the projectglobals.h file. To do so, open projectglobals.h by double-clicking on the file as shown in [Figure 14](#) and scroll down to the MCU\_maskset selection area.

**NOTE:** Steps 9 through 11 are not necessary in the single MCU stationery method. Selections are already configured.



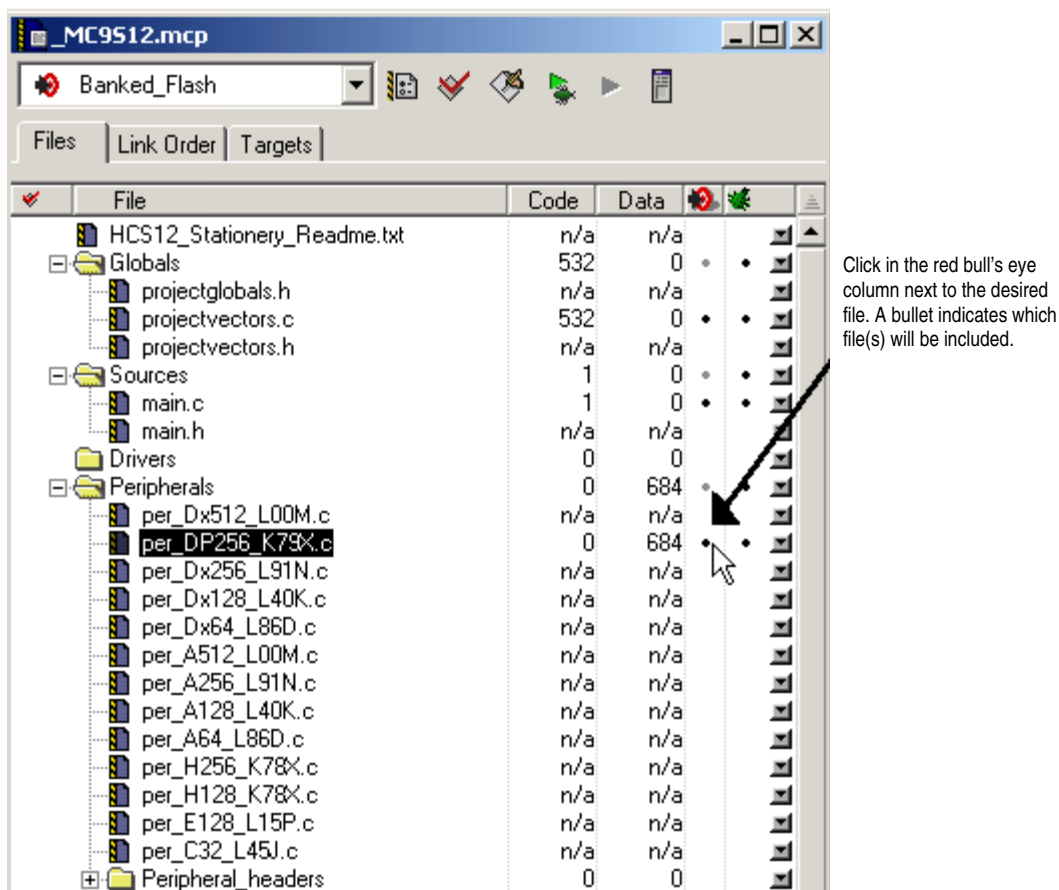
**Figure 14. Opening Project Globals File**

To include and compile MCU files for a device, remove the double forward slashes ("//") in front of the #define command for that device. See [Figure 15](#).



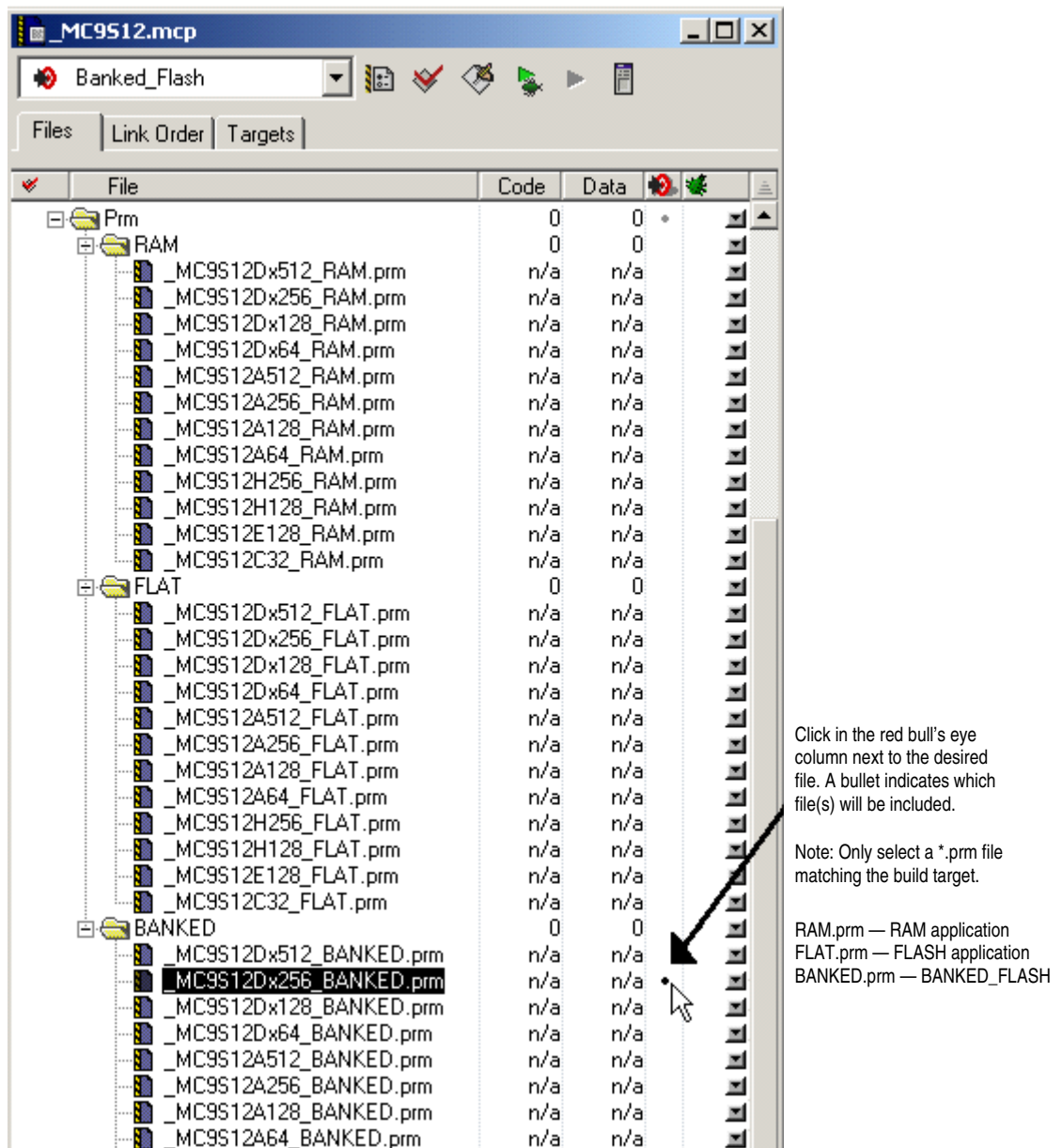
**Figure 15. Selecting a Target MCU**

You must include the respective MCU definition and parameter files that are necessary for the tool to compile and link your application software successfully for your chosen target MCU. Step 6, illustrated by [Figure 16](#) shows how to add the MCU definition file by placing a bullet next to the file name. For this example, continue to select an MC9S12DP256 (K79X) for your target MCU by selecting the per\_DP256\_K79X.c file.



**Figure 16. Selecting an MCU Definition File**

Complete the stationery setup process by selecting the MCU parameter file corresponding to your chosen target MCU. Recall that the MCU parameter file must match the target build chosen. Include the MCU parameter file by placing a bullet next to the relevant file. **Figure 17** shows step 7 (selecting the MCU parameter file), and lists the relationship between the target build and parameter files. Note, that an MC9S12DP256 (K79X) is still our primary device for development when the user selects the MC9S12Dx256\_BANKED.prm file.



**Figure 17. Selecting an MCU Parameter File**

## Developing Software Using the Stationery Features

This section will use a serial communication application example built on the HCS12 software stationery to highlight the following features:

- Developing a main routine — Including calls to function subroutines, references to global variables, and application sequences
- Using global variables — Allowing access to variables throughout the software project
- Using software drivers — Including initialization routines and function subroutines
- Accessing register space — Reading and writing to memory registers via words, bytes, and bits
- Using macro definitions — Creating memory masks for register initialization, conditional statements, and in-line assembly inclusion
- Using interrupt service routines — Developing routines that service a CPU interrupted request

## Overview of Serial Communication Application Example

The example used in this application note is a serial communication project built on a MC9S12DP256 (K79X) target MCU and uses a banked FLASH target build. The project initializes an SCI module to transmit and receive data at 19200 baud rate via an RS-232 interface connected to a computer's serial port. The project is similar to the common "Hello World" C++ programming language example. The difference is that this project uses the MCU's SCI module to output characters to a display on a computer terminal window. Characters are also received by the MCU through computer keyboard inputs entered on the terminal window display. These input characters are echoed back through the SCI module for display on the terminal window.

This example project prompts you for your name. Following the input, the MCU recalls the user's name and returns a greeting using the input name. [Figure 18](#) captures the complete terminal window display following the user's input.

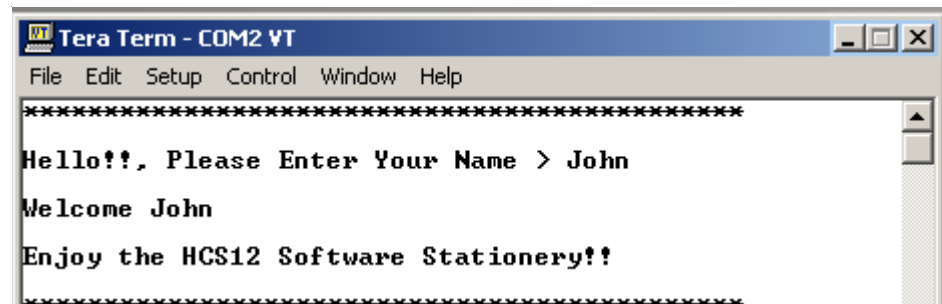


Figure 18. Serial Communication Example — Terminal Window Display

## Developing a Main Routine

To commence a software application, the flow of events must be programmed in the main source file (main.c). The main source file is considered to be the top level of development since it contains the main routine. The main routine typically calls initialization and other function routines in a specific order to fulfill the application. To begin your software, open the main.c file by double clicking on the filename in the main project window. The file should initially look like **Figure 19**.

---

```
#include "main.h"

void main ()
{
    //Insert Application Software Here.
    for (;;)
}
main.c
```

---

**Figure 19. Unimplemented Main Source File**

A header file (\*.h) should be created for each application source file (\*.c). In this example, the corresponding header file for main.c file is named main.h file. You should include the header file (main.h) in the corresponding source file (main.c). The example above shows the main.h file being included within the main.c file through the #include “filename” declaration. The corresponding header file is typically used to list the function prototypes for any routines programmed in the source file. These function prototypes are commonly prefixed with an “extern” command, which allows the functions to be accessible by other files in the project. The header file is also used to include project files that contribute routines or variables to the source file. For example, the main.h file includes the projectglobals.h file, which in turn includes all the necessary MCU peripheral definitions, global variables, and software drivers included in the projectglobals.h file. When using the stationery, a common header file would include the projectglobals.h file as shown in **Figure 20**.

---

```
#ifndef MAIN_H    /*prevent duplicated includes*/
#define MAIN_H

#include "projectglobals.h"

#endif /*MAIN_H*/
main.h
```

---

**Figure 20. Traditional Main Header File**

All function prototypes would be inserted in the header file, between the `#ifndef` and `#endif` condition commands. Those commands are to prevent duplicate file, subroutine, and variable includes.

Next in the development of the serial communication example, we insert the appropriate local variables and fill in the main routine with functions that create the communication described in [Overview of Serial Communication Application Example](#). Refer to [Figure 21](#) for the updated main.c file.

---

```

/*****
 *   DESCRIPTION:   Serial Communication Application Example
 *   SOURCE:        main.c
 *   COPYRIGHT:     © 04/2003
 *   AUTHOR:        rat579
 *****/

#include "main.h"

/* Local Variable Declarations */
uchar temp_string[20];           // Temporary Array to Store ASCII Characters

/* Main Routine */
void main ()
{
    SEI();                       // Disable Interrupts
    oscclk = 16000;              // Set Oscillator Freq. = 16 MHz
    busclk = oscclk/2;          // Set Bus Freq. = (1/2) * Oscillator Freq.
    (void) InitSCI(19200);       // Initialize SCI module @ 19200 Baud Rate
    CLI();                      // Enable Interrupts

    for (;;)                    // Interactive Terminal Display Prompt
    {
        (void) printf("*****\n\r\n\r");
        (void) printf("Hello!!, Please Enter Your Name > ");
        (void) scanf("%s",&temp_string);    // Gets User Info
        (void) printf("\n\r\n\rWelcome ");
        (void) printf("%s",&temp_string);    // Displays User Info
        (void) printf("\n\r\n\rEnjoy the HCS12 Software Stationery!!\n\r\n\r");
    }
}

```

---

**Figure 21. Implemented Main Source File**

Note that the implemented main.c file contains several C programming language basics. For example, the main.c file demonstrates the use of a local variable declaration, macro definitions, references to global variables, calls to external routines, and a conditional “for” loop.

Also, notice that the global variables, macro definitions, and routines are not defined within the main.c file, so an additional source file (typically called a software driver) must be existent in the project. This driver consists of the external routines called by the main.c file. Also, the projectglobals.h file maintains the global variables and macro definitions referenced in the main.c file. In the creation of the software driver for the serial communication example, several stationery methods will be covered.

### Using Global Variables

Global variables are very common in projects involving MCUs that allocate memory space associated with a label. They differ from local variables, because multiple functions in a project reference them as opposed to being reference by a single function. In the serial communication example, there are two global variables referenced, which are oscclk (oscillator clock frequency) and busclk (bus clock frequency). Both of these variables are referenced in main.c and sci.c files. In the main.c file, you set the variables to a specific value. In the sci.c file, the values are used to calculate the SCI baud rate value to write into the SCI baud rate register (SCIBD). The sharing of both these variables by multiple functions means that they must be declared globally. To declare a global variable in the stationery, the variable declaration must be made within the projectglobals.h file. By placing the variable declaration within the projectglobals.h file, the variable becomes propagated throughout all the source files that include the projectglobals.h file in their corresponding header files as recommended in [Developing a Main Routine](#). Refer to [Figure 22](#) for the source code line declaring these two global variables.

---

```
extern ulong oscclk, busclk;
```

projectglobals.h

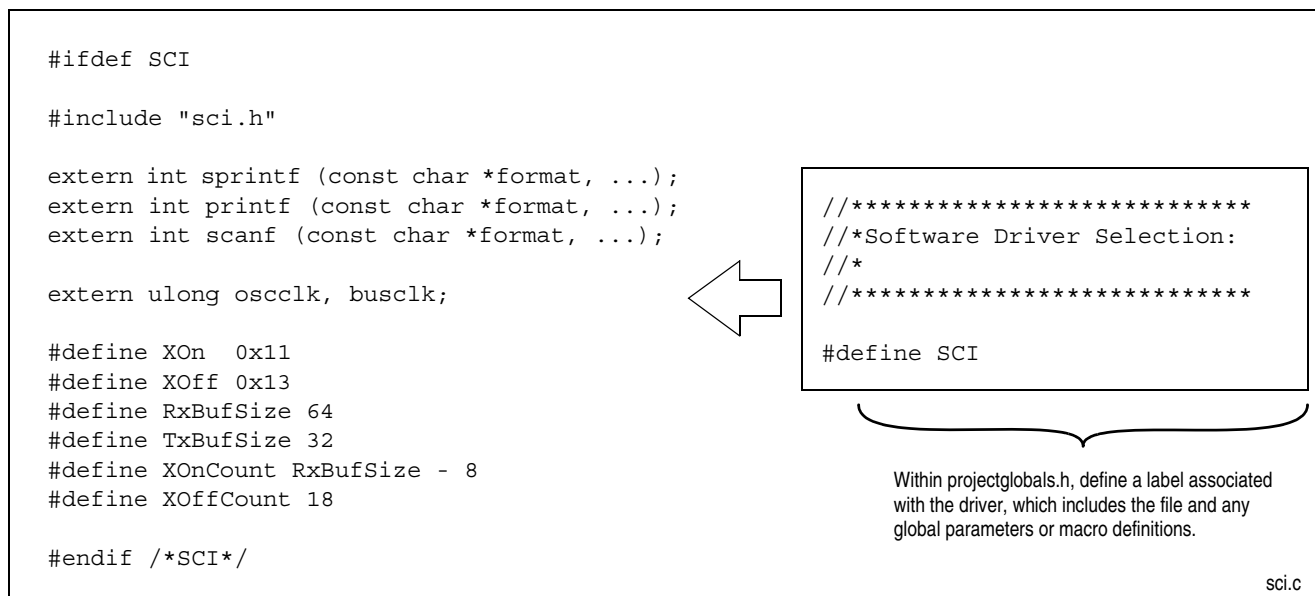
---

**Figure 22. Global Variable Declaration**

### Using Software Drivers

Notice that the main routine in the main.c file called a routine labeled (void) InitSCI (19200). The InitSCI routine is intended to initialize the SCI module to communicate at a 19200 baud rate. The InitSCI routine is not defined within the main.c file, but it is defined at a lower level, within a software driver (sci.c). Software drivers are source files most commonly used to define function routines, which either initialize an MCU peripheral or execute a series of instructions. Drivers are intended to be like stationery definition files and typically remain untouched by the user. These files, when included in projectglobals.h, allow your main routine in main.c to call specific routines

defined in the drivers. Refer to [Figure 23](#) for an illustration of how to include a software driver (like the sci.c file) in your application project.



**Figure 23. Including a Software Driver**

You must include all the necessary software drivers with any global parameters needed by your application. Notice that the `#include "sci.h"` instruction makes all the routines within the SCI driver available to any file, which includes the `projectglobals.h` file. The instructions declared by the `#define` are macro definitions used within the SCI software driver. These macro definitions are specific to the SCI driver, but were made global in this example in order to adjust them without modifying the `sci.c` file. When including a software driver, you must add the file to the CodeWarrior main project window and select the source file (\*.c) by placing a bullet next to the file's label. For the serial communication example, we will place a bullet next to the only driver labeled `sci.c`.

### Accessing Register Space

There are three main types of register accesses using the module definition files. They are provided with a prototype and example from our serial communication example project, showing how they would be entered in a C language program:

- Word (16-bit) access
  - `Modulename.registername.word = 0x1234;`
    - `Sci0.scibd.word = 0x1234;`
    - `Sci0.scibd.word = (uint)((((busclk/16)*1000)/baud));`



- Byte (8-bit) access
  - Modulename.registername.byte = 0x12;
    - Sci0.scicr2.byte = 0x2C;
    - Sci0.scicr2.byte = TE + RE + RIE;
- Bit (1-bit) access
  - Modulename.registername.bit.bitname. = 1;
    - Sci0.scicr2.bit.te = 1;

The types of register accesses above write to a register location in memory, but differ in the amount (words, bytes, bits) that they manipulate. These registers can be written to with a numerical value of that type size, solutions to equations ( $SCIBD = ((busclk/16)*1000)/baud$ ), or masked with predefined macro definitions (TE + RE + RIE). For more information regarding the use of macro definitions, refer to [Using Macro Definitions](#). These structures can also be read and stored in local variables of equal type size. These examples, showing methods for using structures in order to access registers in memory, are used in the sci.c file to initialize the SCI module. For more information on how these module definition files are constructed, refer to [MCU Definitions](#).

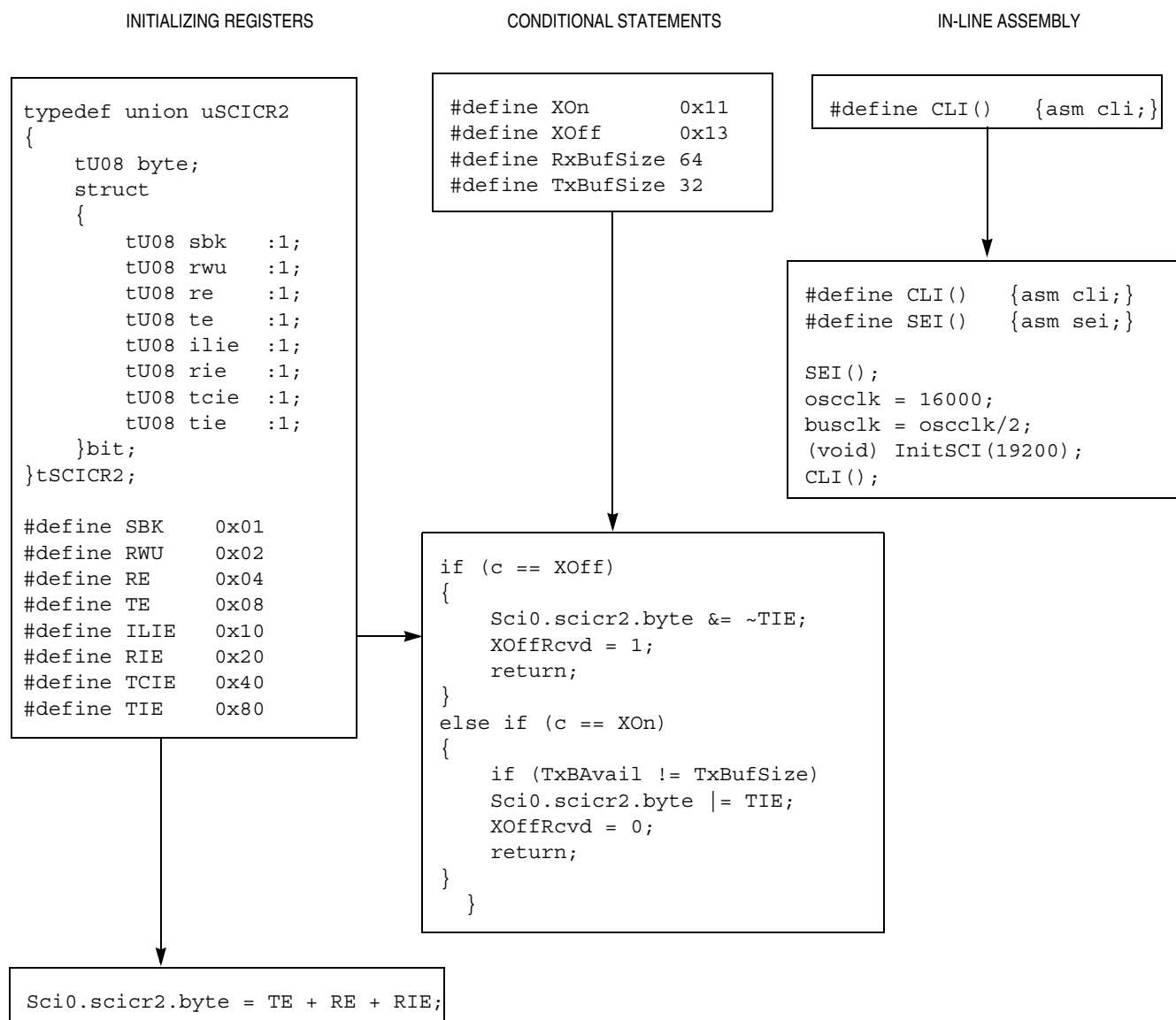
There are a few exceptions within the module definition files that require a slightly different method for accessing register space:

- Modulename.registername[index].word = 0x1234;
  - Tim0.tc[3].word = 0x1234;
- Modulename.registername[index].byte = 0x12;
  - Pwm.pwmper[4].byte = 0x12;
- Modulename.registername.byte.msb.byte = 0x12;
  - Sci0.scibd.byte.msb.byte = 0x12;
- Modulename.registername.byte.lsb.byte = 0x34;
  - Sci0.scibd.byte.lsb.byte = 0x34;

The exceptions listed above differ from the main types in two ways. The first two examples use an array structure with an index to a single word or byte in a series of multiple registers. The last two examples access the most significant byte (msb) or least significant byte (lsb) individually within a word register structure. These four structure examples are not commonly used and are unique to a few modules.

### Using Macro Definitions

Macro definitions are labels associated with a value or instructions that are inserted within your software to take the place of the defined parameter. They are useful when an instruction is used repetitively. Their symbolic representation also simplifies the visibility of your software by making it more self-documented. Macro definitions are usually used as memory masks for register initialization, conditional statements, and in-line assembly inclusion. Refer to [Figure 24](#) for several examples using macro definitions. These examples come directly from the serial communication example project.



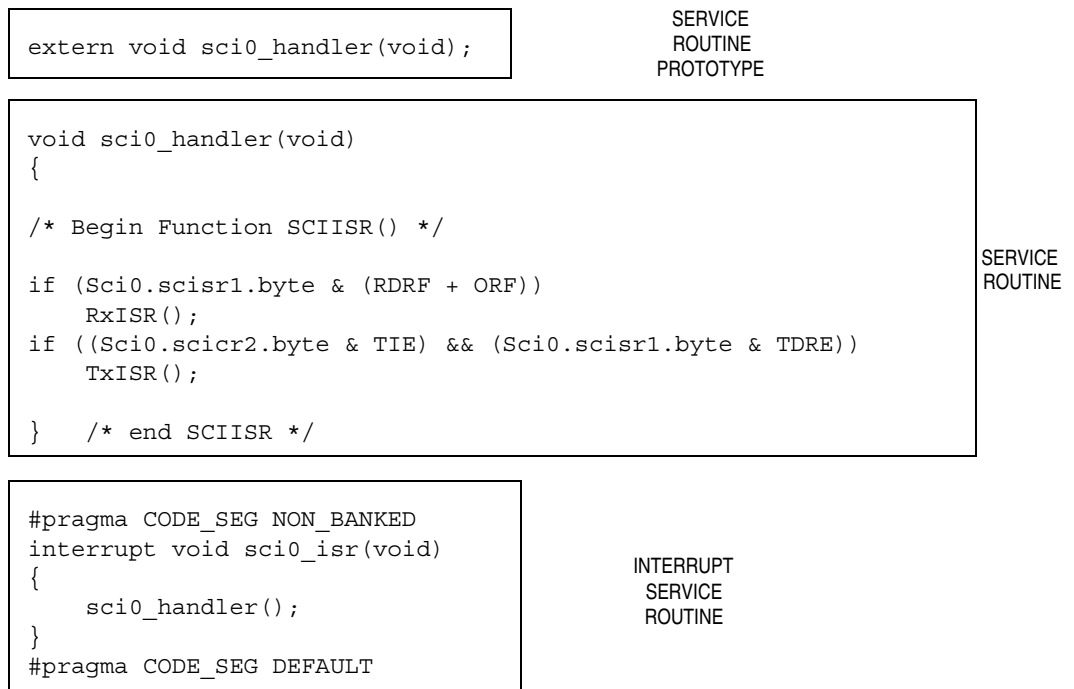
**Figure 24. Macro Definitions in Software**

Note that macro definitions can be declared for local or global use, depending on where they are included. For global use, you should place macro definitions within the projectglobals.h file. Also, each module definition file contains macro definitions for each register structure intended to be memory masks most commonly used for register initialization.

## Using Interrupt Service Routines

Interrupt service routines (ISRs) in the stationery are managed like an ordinary function routine. The difference is that the ordinary function routine gets called in a specific location in software as instructed by the program sequence, but an ISR occurs when a predetermined combination of events becomes true. This combination of events messages the CPU for permission to interrupt the ongoing program sequence. Once the CPU grants the interrupt request, the interrupt's vector address is fetched. Within the interrupt vector address is stored the location of the service routine, which will service the request. The stationery dictates the vector handler label for the ISR associated with that interrupt in the MCU parameter file. The ISRs for all the HCS12 MCUs are located in the projectvectors.c file. For more information regarding the projectvectors.c file, refer to [Control Stage](#).

To demonstrate setting up an ISR within a project, we will use the same serial communication example, which only calls for a single SCI module interrupt. The routine that services the interrupt should be written within the software driver and the prototype should be labeled "extern" in the driver's corresponding header file. This enables easier portability of your software routines. Once the routine that services the interrupt is written ((void) sci0\_handler ()), you can enter the prototype label within the corresponding ISR (interrupt void sci0\_isr ( )) in the projectvectors.c file. Note that the unimplemented ISRs initially have a software trap routine in them. You must remove this routine prior to adding your new routine. [Figure 25](#) shows how a routine to service an interrupt is set up to execute when an ISR is fetched.



**Figure 25. Implementing an ISR**

## Porting Your Application Software to a Target MCU

The stationery makes porting an application from one HCS12 device to another easy. Before porting your software onto another MCU, you must first check that both devices are compatible. Each MCU contains a set of peripherals that might not match in type or version. Only the peripherals accessed by your software application must be compatible. For example, the serial communication example only uses a single SCI module. Because this example application was developed for the MC9S12DP256 (K79X) as a target MCU, it can be ported to any MCU with the identical SCI module with no software modifications. The SCI module definition file pertaining to this target MCU is the S12SCIV2.h file. A compatible MCU that contains at least a single SCI module (S12SCIV2.h) is the MC9S12E128 (L15P). To port your software (when using mass MCU stationery method) to the new MCU, you must repeat the steps described in [Selecting a Target MCU \(Steps 9, 10, 11\)](#). Make sure that when you select a different target MCU within projectglobals.h, you add the comment indicator “//” in front of the label for any previously selected MCU(s) that you want to exclude. Also, when adding a bullet next to the label for the new MCU peripheral and parameter files, make sure to remove the bullet next to the previously selected MCU peripheral and parameter files by clicking on the respective bullet in the red bull’s eye (see [Figure 17](#)).

Software applications that do not contain identical peripherals between MCUs can also be ported within the software stationery. However, software modifications will be necessary to guarantee proper software execution.

---

## Conclusion

The HCS12 software stationery enables you to develop software applications for your target MCU faster and with more ease. All the MCU definition files are readily available, enabling you to begin software development immediately. The stationery provides a standardized architecture that produces well organized and better documented software. Plus, the strict file naming convention eliminates confusion during development. By following the SoC ideology, the stationery will accommodate new HCS12 MCUs and future stationery platforms. This application note introduces the stationery architecture, explains the SoC ideology used by the HCS12 Family, and gives detailed examples covering the proper methods for developing your application using the stationery. Overall, this software stationery improves on limitations encountered on previous software stationery while emphasizing a more comfortable development environment for the user.

The software stationery, *HCS12\_Stationery\_VX\_X.zip*, and an example serial communication project example using the stationery, *AN2485SW.zip*, are available from the Motorola website, <http://motorola.com/semiconductors>.







## **HOW TO REACH US:**

### **USA/EUROPE/LOCATIONS NOT LISTED:**

Motorola Literature Distribution  
P.O. Box 5405, Denver, Colorado 80217  
1-800-521-6274 or 480-768-2130

### **JAPAN:**

Motorola Japan Ltd.  
SPS, Technical Information Center  
3-20-1, Minami-Azabu  
Minato-ku  
Tokyo 106-8573, Japan  
81-3-3440-3569

### **ASIA/PACIFIC:**

Motorola Semiconductors H.K. Ltd.  
Silicon Harbour Centre  
2 Dai King Street  
Tai Po Industrial Estate  
Tai Po, N.T. Hong Kong  
852-26668334

### **HOME PAGE:**

<http://motorola.com/semiconductors>

Information in this document is provided solely to enable system and software implementers to use Motorola 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.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola 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. Motorola does not convey any license under its patent rights nor the rights of others. Motorola 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 Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the part.



Motorola and the Stylized M Logo are registered in the U.S. Patent and Trademark Office. digital dna is a trademark of Motorola, Inc. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola, Inc. 2003