

A PRACTICAL INTRODUCTION TO MICROCONTROLLER PROGRAMMING WITH S12

Pal-Ştefan MURVAY

Horaţiu Eugen GURBAN

Bogdan GROZA

Preface

Whether in home appliances, mobile devices or vehicles, microcontrollers are the drive behind an easier, more enjoyable user experience in a technologically advanced world. Initially built for the automotive industry, the HCS12 microcontroller family has provided derivatives which were quickly adopted in many industrial applications. Designed to provide common microcontroller functionalities (timer, PWM, ADC, serial communication, etc.) along with other application specific modules, the HCS12 is an ideal platform for learning embedded systems programming.

Motivated by the availability of HCS12 controllers in our laboratories at UPT, mostly due to the kind support that we received from Continental (the largest automotive entrepreneur in Timisoara), this platform became employed in many courses at our university. The main intention of this book is to provide a self-contained supporting material for learning the basic features of microcontrollers from the perspective of the HCS12 platform during a single semester course. Due to inherent space constraints, some specific features and modules found on the HCS12 were not included. The material is intended for both novice and experienced students. Students following their first course on microcontrollers could benefit from the specifics of the HCS12 family but also from the general presentation of common microcontroller features that are illustrated on HCS12. Students that already have a background in development with embedded devices may find the material useful for its succinct and structured presentation.

While our work is intended as a short introduction, more comprehensive materials on this subject can be found such as the two excellent books referenced as [1] and [2]. The first of them presents the most common features of the HCS12 family members in detail along with supporting examples of varying difficulty. The second also provides a detailed presentation of common HCS12 features and is accompanied by different application examples. We recommend these two books to more involved readers for an in-depth study of this platform.

The authors

June 2016

CONTENTS

Contents.....	3
1 Introducing the Freescale HCS12(X) microcontrollers and the ZK-S12-B starter kit	6
1.1 The Freescale HCS12(X) microcontroller family.....	6
1.1.1 Short historical note.....	6
1.2 The ZK-S12-B starter kit	8
1.2.1 Development board sections.....	8
1.2.2 Clock source configuration.....	10
1.3 CodeWarrior.....	11
1.3.1 CodeWarrior IDE	11
1.3.2 Creating a new project.....	16
1.3.3 True-Time Simulator & Real-Time Debugger.....	19
1.3.4 The Demo Project	26
2 S12 I/O ports.....	28
2.1 General I/O interfacing	29
2.2 Ports A, B, C and D	30
2.3 Port E.....	31
2.4 Port K.....	32
2.5 Port T.....	32
2.6 Port S.....	33
2.7 Port M	33
2.8 Ports H, J and P.....	34
2.9 Ports AD0 and AD1.....	34
3 S12 Programming model, instruction set and memory addressing	36
3.1 Programming model	36
3.2 Instruction set.....	37
3.2.1 Data transfer instructions	39
3.2.2 Arithmetic instructions	41
3.2.3 Logic and bit instructions	42
3.2.4 Branch instructions	43
3.2.5 Comparison instructions	44
3.2.6 Function call instructions	45
3.3 Memory addressing modes	47
4 The interrupt system, clock and reset generation.....	48
4.1 General concepts of interrupts.....	48

4.2	Interrupts on the S12 platform.....	48
4.2.1	The S12 Interrupt block.....	48
4.2.2	Using S12 interrupts and CodeWarrior.....	51
4.3	The S12 Clock and Reset Generator.....	54
4.3.1	General aspects on clock and reset generation and the S12 features.....	54
4.3.2	Clock generation.....	55
4.3.3	Reset generation.....	57
4.3.4	Using real-time interrupts.....	59
5	The Timer Module.....	60
5.1	TIM associated registers.....	61
5.1.1	Timer counter registers.....	61
5.1.2	Registers common for input capture and output compare.....	63
5.1.3	Registers related to the input capture function.....	63
5.1.4	Registers related to the output compare function.....	64
5.1.5	Registers related to the pulse accumulator function.....	65
5.2	ECT extended timer features.....	66
5.2.1	Registers related to the input-capture function.....	66
5.2.2	Registers related to the modulus down counter.....	67
5.2.3	Registers related to the pulse accumulator.....	68
6	Pulse Width Modulation (PWM) Module.....	70
6.1	General concepts of PWM.....	70
6.2	PWM generation on S12.....	70
6.2.1	PWM Clock configuration.....	72
6.2.2	PWM waveform configuration.....	73
6.2.3	PWM module control.....	76
6.2.4	16 bit resolution PWM.....	78
7	The Analog-To-Digital Converter Unit (ATD).....	80
7.1	General ADC concepts and the S12 ATD unit.....	80
7.1.1	Applications and some theoretical considerations on sampling rate, resolution and accuracy. 80	
7.1.2	The S12 ATD unit.....	80
7.1.3	Conversion algorithm.....	82
7.2	ATD associated registers.....	82
7.2.1	ATD control registers.....	82
7.2.2	ATD status registers.....	84
7.2.3	ATD test registers.....	85

7.2.4	ATD input enable register	86
7.2.5	ATD port data register.....	86
7.2.6	ATD conversion result registers	86
8	Internal memory	89
8.1	Basic concepts on memories	89
8.1.1	Common memory types.....	89
8.1.2	Memory mapping.....	89
8.2	The S12 memory system.....	90
8.2.1	S12 Flash memory.....	90
8.2.2	S12 EEPROM memory	95
8.2.3	Remapping memory sections.....	98
9	References	100

1 INTRODUCING THE FREESCALE HCS12(X) MICROCONTROLLERS AND THE ZK-S12-B STARTER KIT

This first chapter will introduce the Freescale HCS12(X) microcontroller and the ZK-S12-B development kit along with the included microcontrollers, the CodeWarrior IDE and the CodeWarrior Debugger as this is the setup used to support the examples discussed in throughout the chapters of this book.

1.1 THE FREESCALE HCS12(X) MICROCONTROLLER FAMILY

1.1.1 Short historical note

The HC9S12 microcontroller, subject of this book (briefly referred as S12), is part of a microcontroller family introduced by Freescale since the beginning of 2000. This line is built upon the success story of the HC11 platform developed in the '80s and enhanced in the mid 90's as HC12. The microcontroller is still under development at Freescale Semiconductors (a spinoff/division of Motorola). The most advanced embodiments of this line, feature and additional co-processor called XGATE and are referred as S12X. Briefly, this line of microprocessors can be described as a line of cost-efficient and reliable 16-bit processors that is highly used in the automotive industry but nonetheless in various industrial and home appliances. The clock speed varies through this family of microprocessors, you may find several specifications, but the main lines are: the HC9S12 designed to work at 25MHz and the stronger S12X that works at 50MHz. The first HC12 derivatives could achieve speeds of only 8MHz. For current needs in the automotive industry, even the top of the S12 line is reaching its limits for the most demanding tasks, but it is still widely deployed in areas within the car where cost rather than performance is a major issue.



Figure 1.1 The three S12 cores included in the ZK-S12-B kit: S12C128, S12DT256 and S12XDT512 (with XGATE co-processor)

Some additional terminology may be also useful. Please pay attention to the fact that we refer these as microcontrollers and not microprocessors. The difference between the two is that microprocessors (see for example the well-known lines of Intel Pentium and AMD) do not embed the memory and peripherals on the chip, while for microcontrollers this is clearly the case as memory and peripherals are embedded with the core in the same capsule. The application range for microcontrollers is usually centred on specific applications (e.g., engine control tasks) and thus lower speeds, lower memory, etc. are usually sufficient to fulfil the task, low cost and high reliability being

the more important constraint when designing these devices. This usually does not hold for microprocessors which are designed to fulfil a large variety of tasks (in personal computers for example) and their costs can reach prohibitive levels.

The S12 microcontroller has all the memory embedded on the chip. The additional numbers in the chip name specifies the amount of Flash memory available. They can be seen in Figure 1.1 for models dubbed: MC9S12C128CFU, S9S12DT256CFUE and MC9S12XDT512MFU which in turn have 128, 256 or 512 kilobytes of memory.

Although built around the same CPU, various members of the S12 family differ by the hardware configuration packaged with the S12 core. This means that available modules and ports will vary from one chip type to the other. Complete descriptions of chip capabilities and register definitions are available in corresponding datasheets. If you are using the SofTec Microsystems installation package, you can find datasheets on your system at `c:\Program Files\SofTec Microsystems\SK and ZK-S12(X) Series\DataSheets\ZK-S12-B\[<target MCU>]`, where <target MCU> stands in for the name of the microcontroller. Alternatively you can access the datasheets online directly from the producer's website [3, 4, 5]. The variety of modules that are present on this device are presented in Figure 1.2 which also points out the chapters which are going to detail each of them. Only the basic modules are going to be discussed as the communication modules are out of the scope of this book. These are detailed as follows:

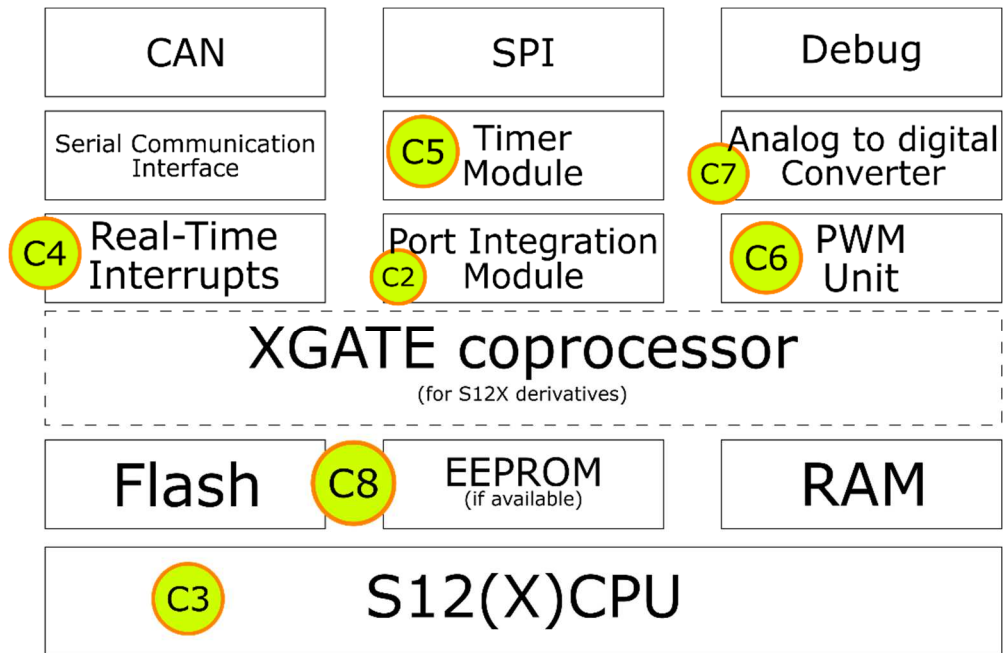


Figure 1.2 Block diagram of the HC9S12 with the associated chapter

- The first chapter introduces the ZK-S12-B development kit and the CodeWarrior environment, an accessible setup for starting up with S12 development;
- The second chapter will introduce the S12 ports which are used as general purpose input/outputs as well as for dedicated functionalities
- Chapter 3 addresses the programming model and the low-level (assembly) programming of the microcontroller. This is mainly intended to make you more familiar with how this microcontroller works.
- Chapter 4 continues somewhat on the same line by providing insights on the interrupt system, a subsystem which is fundamental for programming embedded applications.
- Chapter 5 addresses the Pulse Width Modulation (PWM) subsystem, this is employed for generating signals that can be used for various tasks such as the control of motor drives.

- Chapter 6 addresses the Analogue-to-Digital (ADC) conversion system that is used for signal acquisitions.
- Chapter 7 addresses the Timer module, fundamental for any real-time application.
- Chapter 8 introduces the internal Flash and EEPROM memory units which are included in microcontrollers for storing program and data.

1.2 THE ZK-S12-B STARTER KIT

1.2.1 Development board sections

The ZK-S12-B Starter KIT is an evaluation board designed for the Freescale HCS12(X) microcontroller family. The starter kit user's manual [6] (which is contained in the installation folder of the Softec Microsystems System Software – "c:\Program Files\SofTec Microsystems\SK and ZK-S12(X) Series\") contains a presentation of the development board and its configuration options. An electrical schematic of the board can be found in the accompanying schematics file [7] in the same install folder. The board is organised into the following sections: MCU, Power supply, USB to BDM interface, BDM, CAN, LIN, RS-232, Serial settings, Inputs, Reset, Outputs and a Prototype area. A short description regarding each section follows.

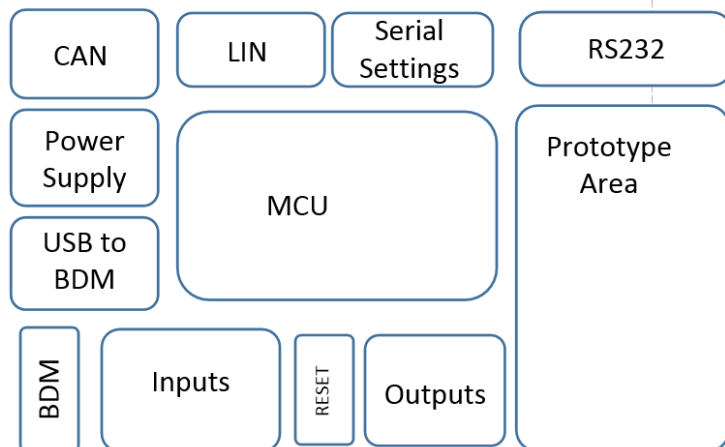


Figure 1.3 ZK-S12-B Starter kit PCB sections

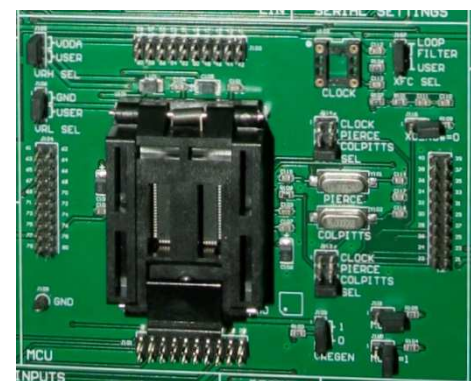


Figure 1.4 MCU section

MCU section – The Freescale HCS12(X) microcontroller is connected to the board using an 80 pin QFP ZIF (Zero Insertion Force) socket. All the microcontroller pins are accessible through four 20 pin header connectors which are mounted around the socket. The clock source can also be selected in this section as illustrated in Figure 1.4. By using the J13 and J14 jumpers, in the MCU section, the following selection can be made: Clock, Pierce and Colpitts. The clock source configuration is discussed in detail in section 1.2.2.

!!! Before performing the following operation make sure the board is disconnected from the power supply and also disconnect the USB cable if connected.

To change the microcontroller or to find out which microcontroller is currently installed in the development board socket, open the ZIF socket and check the writing on the inserted MCU. Write down the name of the microcontroller, you will need this when creating a new project or when loading the demo projects. Avoid touching the microcontroller because it can be easily damaged due to electrostatic discharges (ESD). Make sure you have closed the socket once you are done.

Power supply section – The evaluation board needs to be connected to a 12V DC power supply (used to power the LIN and CAN transceivers) provides regulated 3.3V or 5V to the MCU and to other circuits on the board. A power adapter is included within the kit content

USB to BDM interface and BDM interface – An USB to BDM convertor is installed on the board. By using this interface no external BDM in-circuit debugger is required. The evaluation board provides a BDM connector, so also an external in-circuit debugger can be employed if needed.

CAN section – This section contains two **MC33388** CAN transceivers (capable of up to 125 Kbaud speeds). The TX and RX signals for CAN0 and CAN4 can be individually connected or disconnected to/from the transceiver by using 4 jumpers.

LIN section – The LIN section contains two MC33661 LIN transceivers (100 Kbps - fast mode). By using the jumpers in this section the two LIN nodes can be configured as master nodes. Another jumper selection allows the selection of the LIN transceivers power supply between the ZK-S12 board power supply and by the LIN network.

Serial settings section – The MCU RX and TX lines can be connected to LIN section or to RS232 section.

RS-232 section – A MAX3232 transceiver is used to convert TTL to RS232 logic levels. Each of the two serial communication channels can be configured as Data Terminal Equipment (DTE) or Data Communication Equipment (DCE).

Inputs section – The inputs section provides the following elements: a potentiometer, eight DIP-switches, and four push-buttons. The potentiometer is connected to the PAD00 MCU pin; this connection can be disabled by removing a jumper. The push-buttons are connected to PP0, PP1, PP2 and PP3 while the dip-switches are connected to PT0-PT7.

Outputs section – The outputs section contains 8 LEDs which are connected to port B pins (PB0 – PB7). Each LED can be disconnected from the corresponding port B pin by removing the associated jumper.

Prototype sections – The prototype section is divided in two areas, a SMD area and a through-hole area. The 5V, 12V, VDD, and GND signals are routed to the prototype section and can be easily accessed in through-hole area.

Power supply section			
J202	VDD SEL	1-2	3.3 V
		2-3	5V
Inputs section			
J205	ENA	installed	Potentiometer connected to PAD00
		removed	Potentiometer disconnected
Outputs section			
J204	LED ENA	installed	LED connected
		removed	LED disconnected
MCU section			
J105	VRH SEL	1-2	VRH connected to VDDA
		2-3	VRH connected to J103
J106	VRL SEL	1-2	VRL connected to GND
		2-3	VRL connected to J104
J113-J114	OSC SEL	1-2	CLOCK
		3-4	PIERCE
		5-6	COLPITTS

Table 1.1 Jumper configuration

As presented, various features of the development board, such as the MCU supply voltage, potentiometer and LED availability, CAN, LIN and serial communication channels or clock source, can be configured by means of jumper settings. The configurations that affect the power supply, MCU, inputs and outputs sections are summarised in Table 1.1. For other settings please refer to the starter kit user manual [6]. The default values are illustrated in bold letters.

1.2.2 Clock source configuration

In the MCU section the user can select one of the three clock sources that are available on the development board: two 16 MHz crystals, in Pierce and in Colpitts configuration, or a CMOS compatible external oscillator. Table 1.2 summarises the clock configuration settings. Figure 1.5 illustrates the board's oscillator circuit.

Oscillator configuration	J113 & J114, connected pins	J115, "XCLK#=0"
External Clock	1-2	Installed
Pierce	3-4	Installed
Colpitts	5-6	Removed

Table 1.2 Jumper configuration for oscillator selection

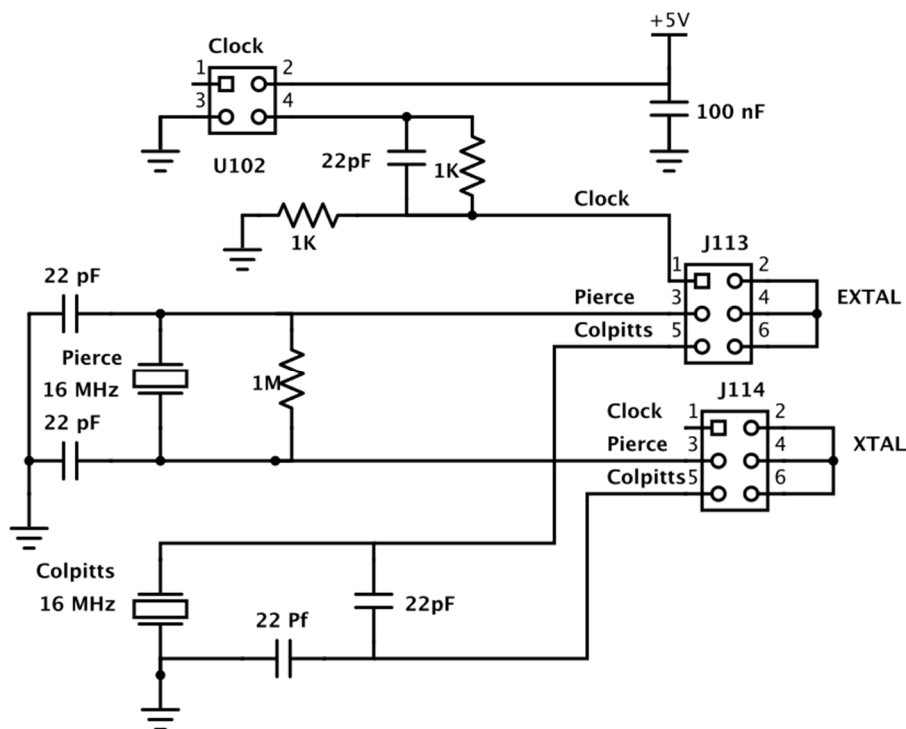


Figure 1.5 ZK-S12-B oscillator circuit

When the 16 MHz crystal needs to be used in Colpitts configuration, pins 5 and 6 have to be connected (by jumper) on connectors J113 and J114.

When an external crystal is used together with the internal, low power, Colpitts oscillator the XCLKS# signal has to be forced to high. XCLKS# signal is sampled on the rising edge of the RESET signal. For forcing the XCLKS# signal to high the jumper has to be removed from the J115 connector. When a Pierce oscillator or an external CMOS compatible external oscillator is used the XCLKS# signal has to be low during the rising edge of RESET signal, therefore the jumper should be installed on J115. When the 16 MHz crystal is used in Pierce configuration, pins 3 and 4 have to be connected (by jumper) on connector J113 and J114 (OSC SEL).

An external clock module can also be used. In this case the CMOS compatible external oscillator has to be inserted in the U102 socket and pins 1 and 2 have to be connected on connector J113. The XCLKS# signal has to be low during the rising edge of RESET signal (J115 connector “XCLKS#=0” – jumper installed).

1.3 CODEWARRIOR

This section provides a quick introduction to CodeWarrior IDE, a Freescale Semiconductor product built for editing, compiling and debugging software for various microcontroller platforms provided by the same manufacturer. CodeWarrior comes with various supporting packages according to the platforms which it has to be used for. We need to use CodeWarrior Development Studio for HCS12(X) Microcontrollers to build applications for microcontrollers in this family.

1.3.1 CodeWarrior IDE

The following guide was made based on version 5.7.0 of the CodeWarrior IDE. While window appearance and some features may vary between different versions of CodeWarrior, the main functionalities are as described in what follows.

For a quicker introduction to the CodeWarrior IDE a demo project provided with SofTec Microsystems Software package is used to present its functionalities.

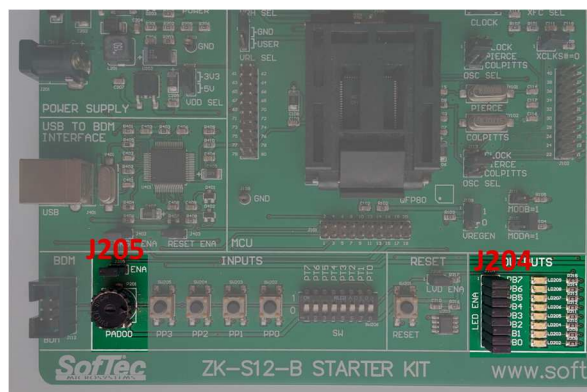


Figure 1.6 Locations of J204 and J205

The following steps should be taken before loading the demo project:

- Make sure the J204 “LED ENA” jumpers from the output section of the board are installed (Figure 1.6) – 8 jumpers.
- Make sure the J205 “POTENTIOMETER ENABLE” jumper from the input section of the board is installed (Figure 1.6).
- Connect the power supply to the ZK-S12-B board
- Connect the PC and the board via USB cable

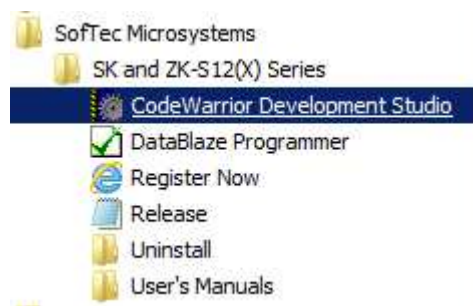


Figure 1.7 Launching CodeWarrior IDE from the start menu

To start CodeWarrior you can take one of the following steps:

- From the Start menu follow the path: Start → All Programs → SofTec Microsystems → SK and ZK-S12(X) Series → CodeWarrior Development Studio, as in Figure 1.7
- Alternatively follow the drive path to the Start menu shortcut: "C:\ProgramData\Microsoft\Windows\Start Menu\Programs\SofTec Microsystems\SK and ZK-S12(X) Series\CodeWarrior Development Studio"
- Follow the local installation path: "C:\Program Files (x86)\Freescale\<CW install folder>\bin\IDE.exe", where <CW install folder> should be replaced with the CodeWarrior install folder on your system.
- If your CodeWarrior installation is located elsewhere, start it by following your install path.

On opening CodeWarrior a startup window will appear (if this was not disabled) like shown in Figure 1.8. The appearance of this window can be disabled by unchecking the "Display on Startup" checkbox. If needed it can later be re-enabled from the CodeWarrior settings.

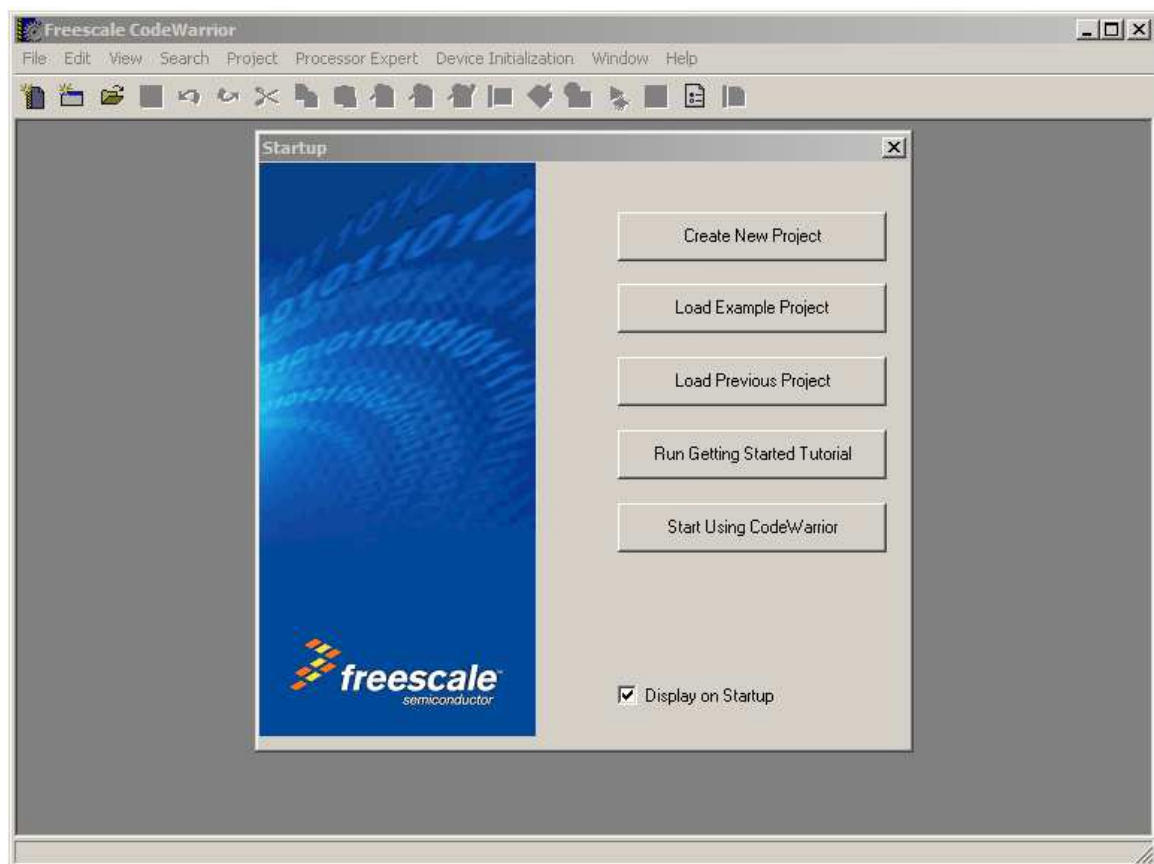


Figure 1.8 Freescale CodeWarrior IDE Start-up window

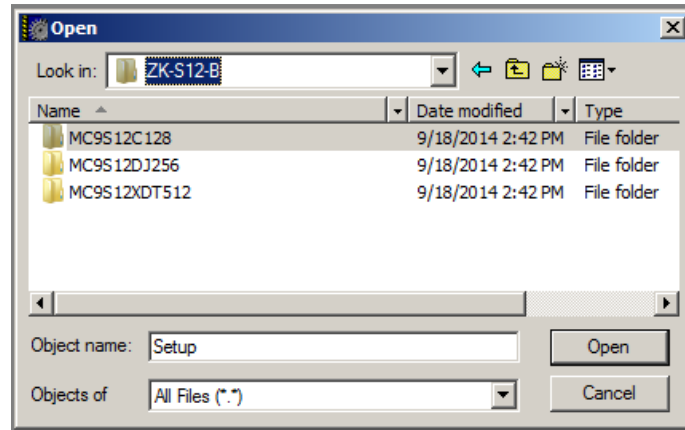


Figure 1.9 Open window, select ZK-S12-B board and the MCU

Open the CodeWarrior demo project Demo.mcp (*.mcp is the CodeWarrior project file) which should be located at the following path: c:\Program Files\SofTec Microsystems\SK and ZK-S12(X) Series\CodeWarrior Examples\ZK-S12-B\[<target MCU>]\Demo\Demo.mcp. Here the <target MCU> label stands for MC9S12C128, MC9S12DJ256 or MC9S12XDT512. Select the MCU that is installed on your board (Figure 1.9).

The project items are organized in 3 tabs: Files, Link Orders and Targets (Figure 1.10).

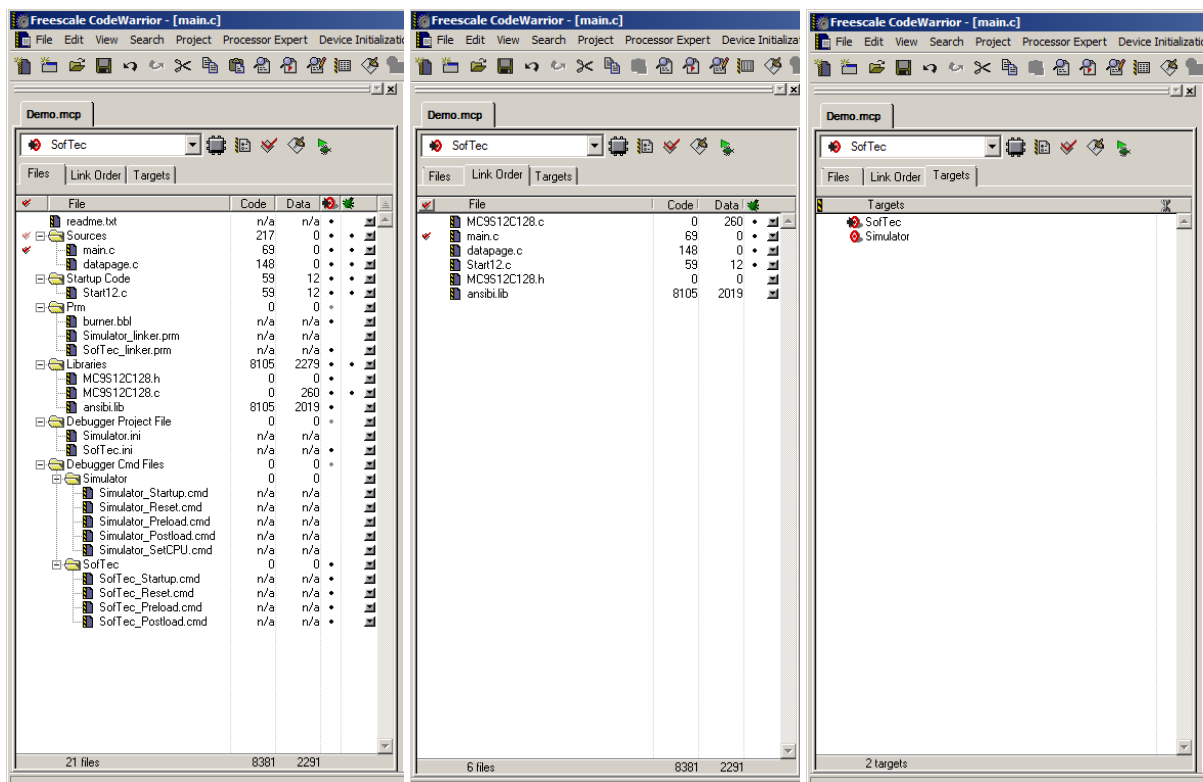


Figure 1.10 Files, Link order and Targets tabs

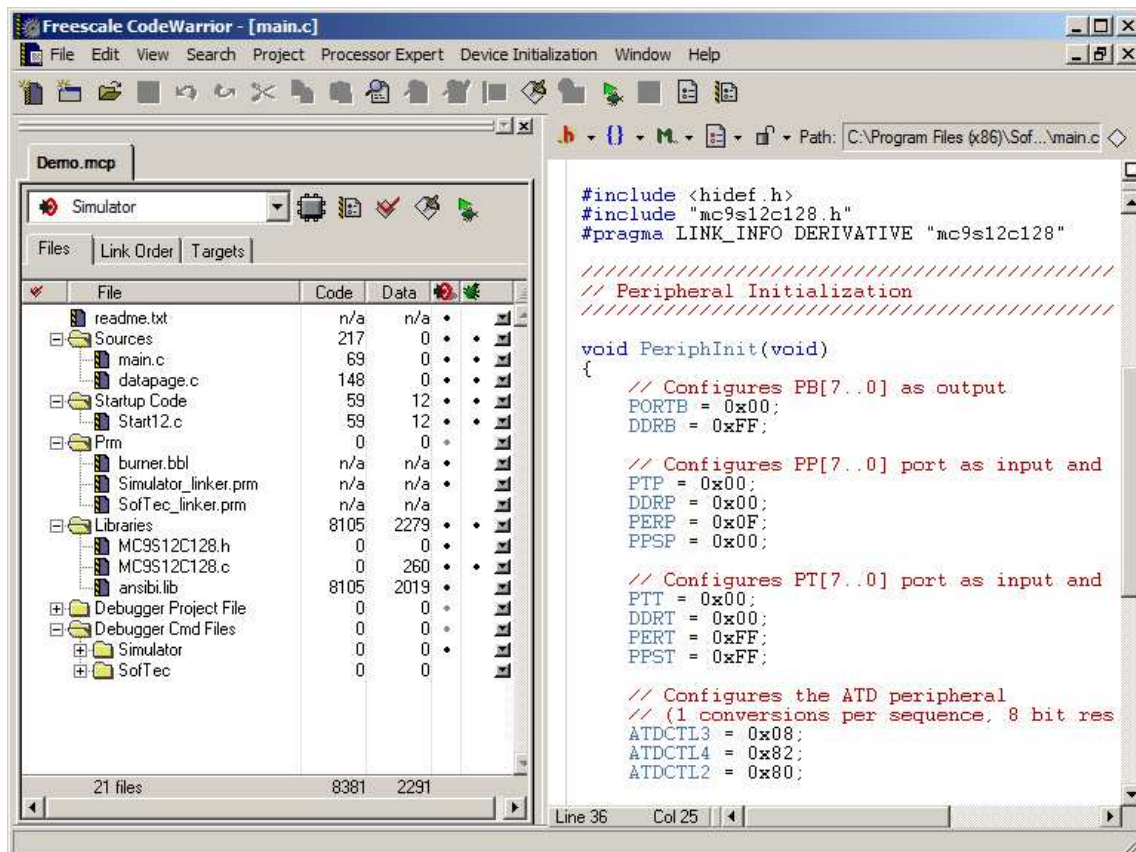


Figure 1.11 Demop.mcp project files

The Files tab of the project area lists all the files used in the project. These files are organized in a tree view manner, each main section in the view corresponding to a category of files. The following sections can be seen in the Files tab (Figure 1.11):

- **“Sources”** - contains the source code
- **“Startup Code”** - contains the startup code (“Start12.c”), used for initializing C library and calling the main() function
- **“Libraries”** - contains the library files (“ansibi.lib” is included as default), also the device header and file (“MC9S12C128.h”, “MC9S12C128.c” in the presented example as we employed an S12C128 MCU)
- **“Debugger Project File”**- contains the .ini file for the existing target connections, in this case simulator and SofTec USB to BDM debugger (“Simulator.ini”, “SofTec.ini”).
- **“Debugger Cmd Files”** - contains subfolders for the existing target connections (“Simulator”, “SofTec”). Each subfolder contains the associated command files.

The order in which the project files are linked can be set in the Link order tab (Figure 1.12). To change the link order you have to grab and drag a file to the desired position. If some files should not be included in the build they must be deleted from this list. A red check mark in the first column of the list indicated a file that has changed or was *touched* and will be compiled during the next build process. Files can be manually touched by selecting this option from the dropdown that opens when pressing on the arrow on the last column.

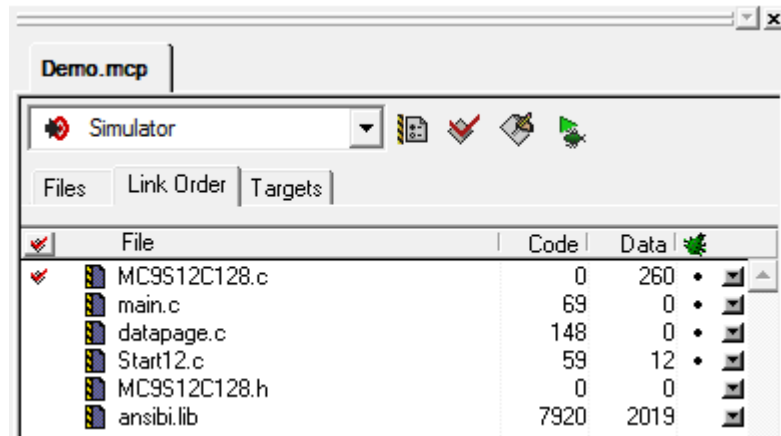


Figure 1.12 Link order tab

Two connection methods have been configured for this demo project. A target connection that uses a Full Chip Simulation and a connection target that uses the SofTec USB to BDM interface. You can set the default target connection from the main menu: “Project->Set Default Target”. An alternative is to use the CodeWarrior project panel as can be seen in Figure 1.13.

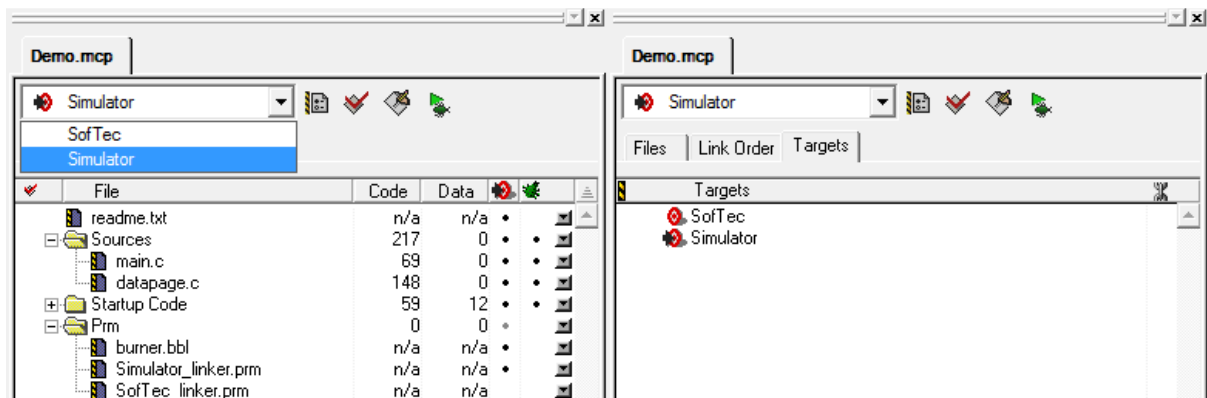


Figure 1.13 Selecting the target connection in the CodeWarrior project

The CodeWarrior toolbar provides quick access for most frequently accessed menu items (Figure 1.14) related to file access, editor and build process functionalities.

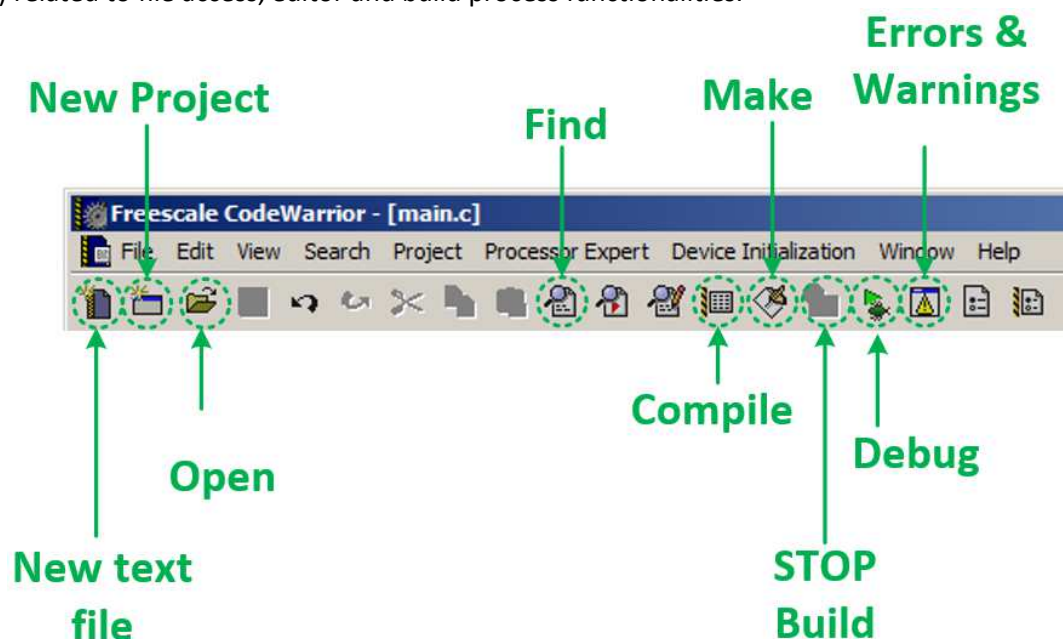


Figure 1.14 CodeWarrior toolbar

1.3.2 Creating a new project

For creating a new project you can use File -> New Project, the shortcut *CTRL+SHIFT+N* or the “New Project” icon on the menu bar (Figure 1.15).

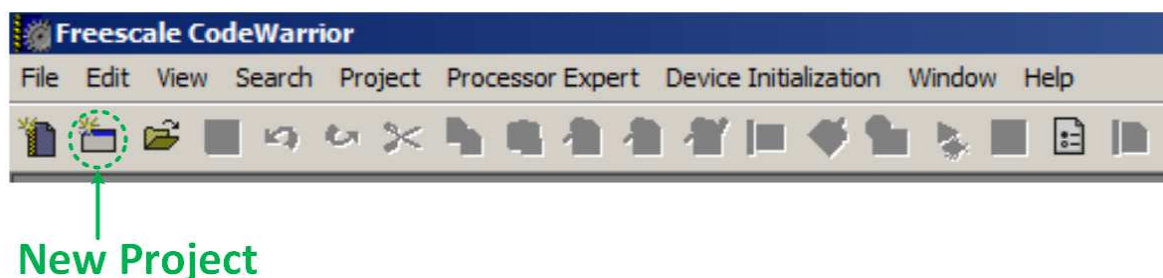


Figure 1.15 The create a new project icon on the menu bar

The new project window will open with a wizard that will guide you through the project setup phase. In the first wizard page select the microcontroller derivative that needs to be used and the connection types that should be included in the project. Choose the derivative according to the chip installed on your board. Selecting a wrong derivative can lead to undefined behaviour due to differences in memory mappings and feature availability. Select “Full Chip Simulation” if you want to use a MCU simulator, or use the SofTec HCS12 (inDART In-Circuit Debugger) for the ZK-S12-B development kit. Alternatively, if other debugger hardware is available you can select another connection type accordingly. After making the desired settings press the Next button to continue.

Note that each wizard window will generally contain an inactive text box displaying additional information about the selections made.

In the following wizard page, Figure 1.17, you can select the programming language between several options: Assembly, C or C++. In most cases for implementing examples and exercises in this book we will need a C type project. The project name and location will also be set in this step by filling the corresponding fields. From this point on the Finish button becomes active and by pressing it you can skip the next steps leaving the rest of the settings to their default values.

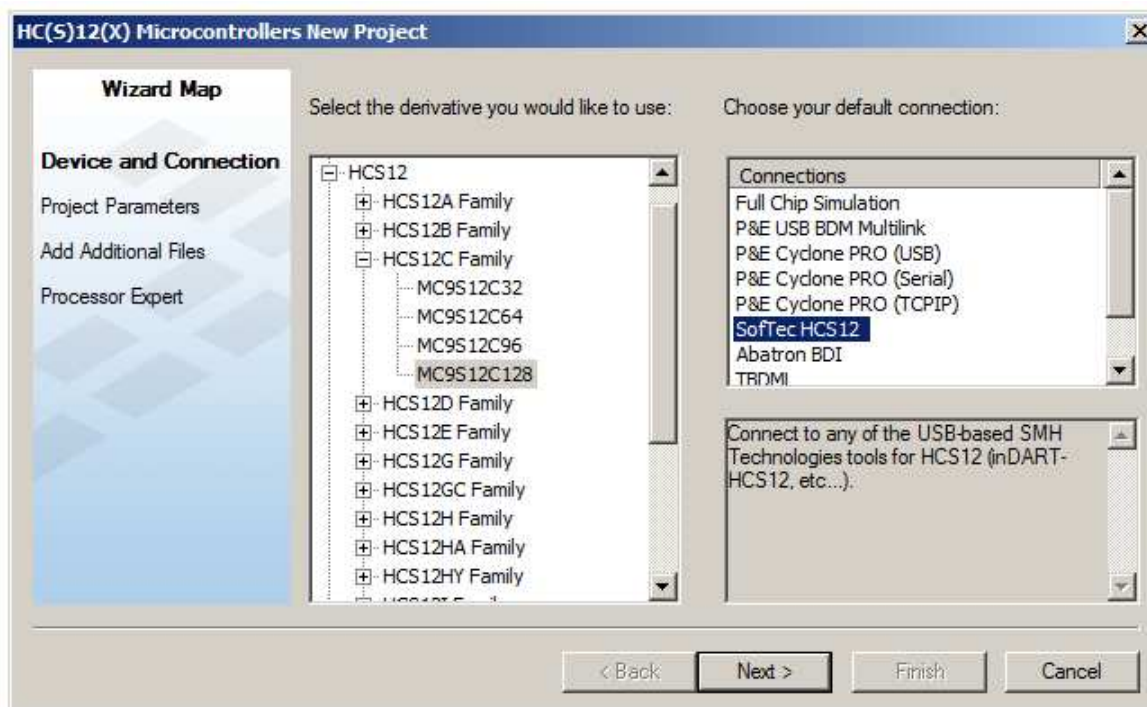


Figure 1.16 New Project - Device and Connection

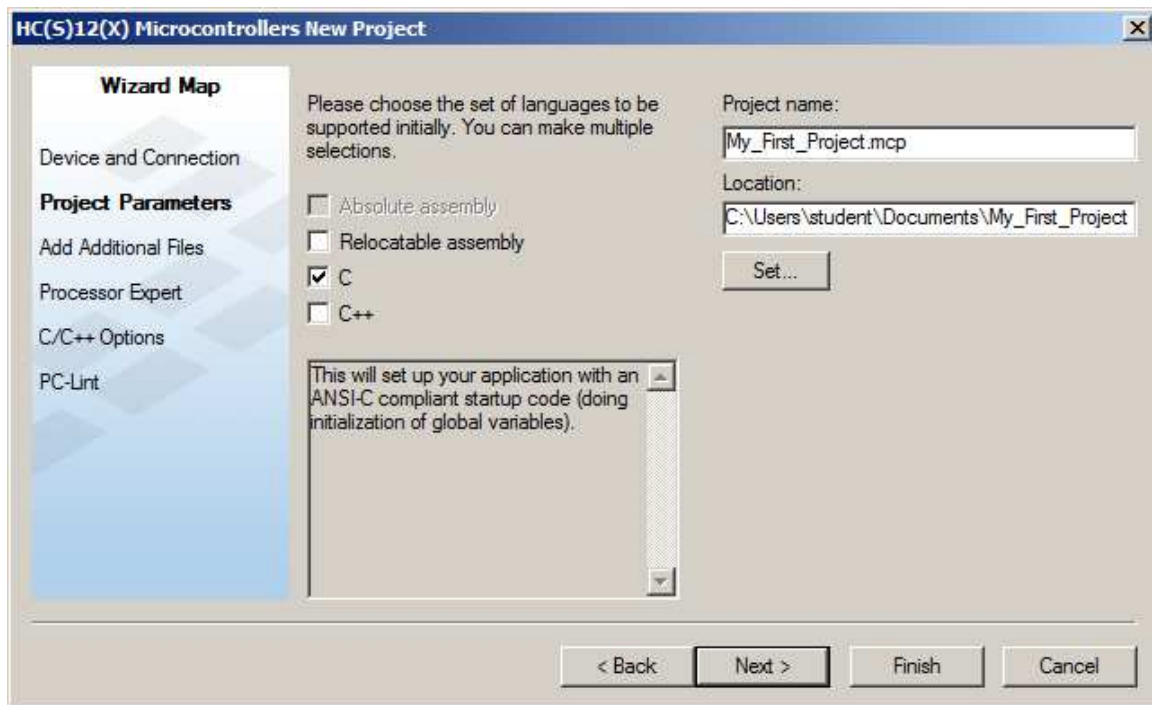


Figure 1.17 New Project - Project parameters

After going to the following step (Figure 1.18) you can select and add additional files in the project if they are needed. This step is generally needed when using already existing drivers or software libraries. These files can be automatically copied in the project if the corresponding checkbox is activated, otherwise they will be used as references from their existing location on disk. The main.c file will be generated if the corresponding checkbox is checked.

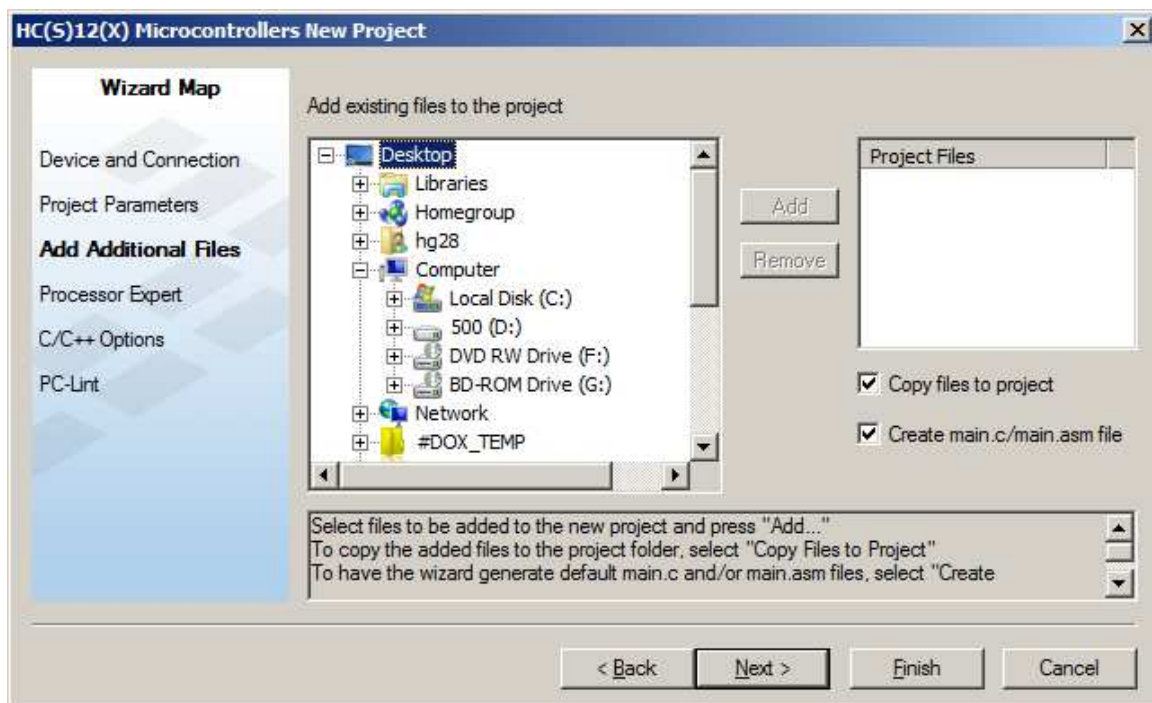


Figure 1.18 New Project - Add additional files

The next wizard step, called Processor expert, is illustrated in Figure 1.19. In this step you can select if CodeWarrior should generate some initialization code automatically (MCU peripherals configuration, ISR templates).

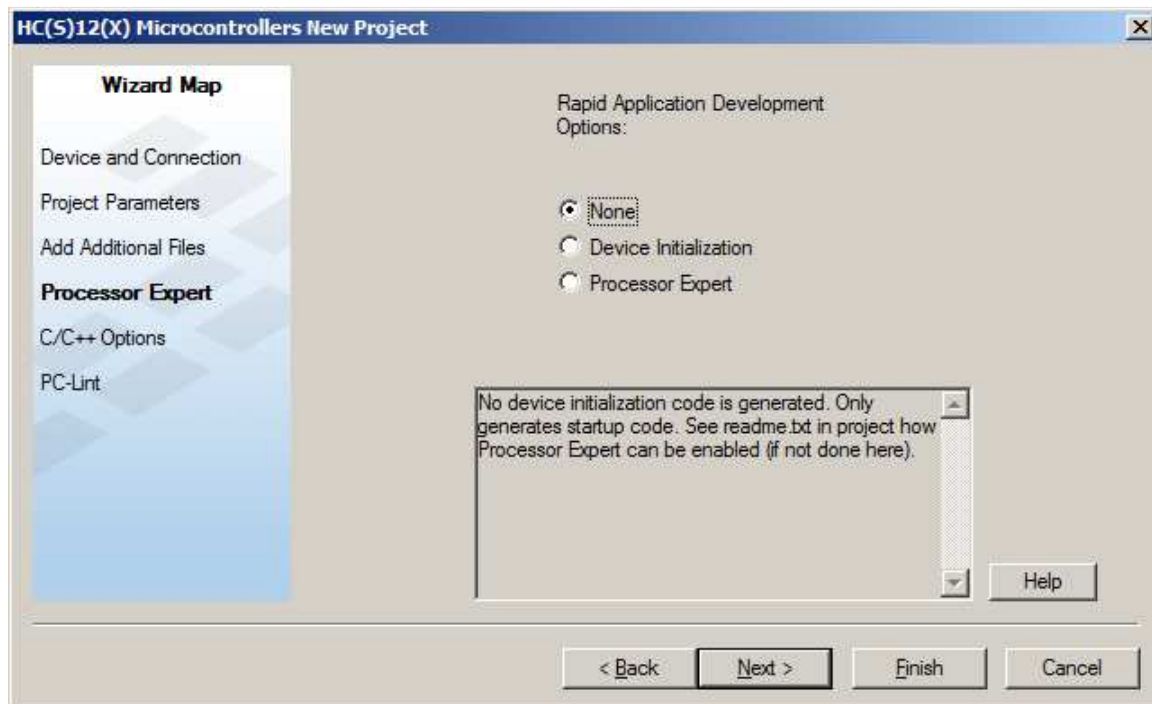


Figure 1.19 New Project - Processor expert

The C/C++ Options wizard page (Figure 1.20) is dedicated for setting various options related to the startup code, memory size and usage of floating point variables. By selecting “ANSI startup code” the global variable will be initialized and the main function called. A “Small” memory model can be selected if the code doesn’t exceed 64K. If no floating point variables are needed select “none”. The double datatype can be selected as 4 or 8 Bytes.

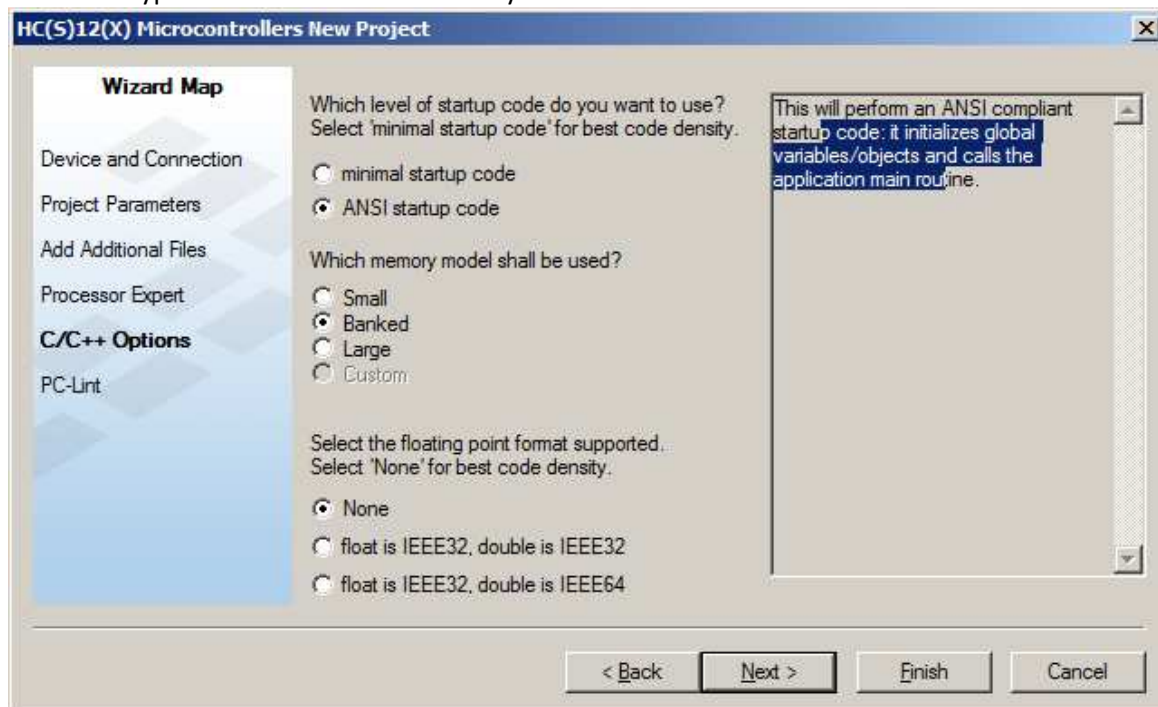


Figure 1.20 New Project - C/C++ options

In the last wizard page (Figure 1.21) the use of PC-Lint can be enabled for performing static code analysis and for checking the code compliance with MISRA C and C++ rules.

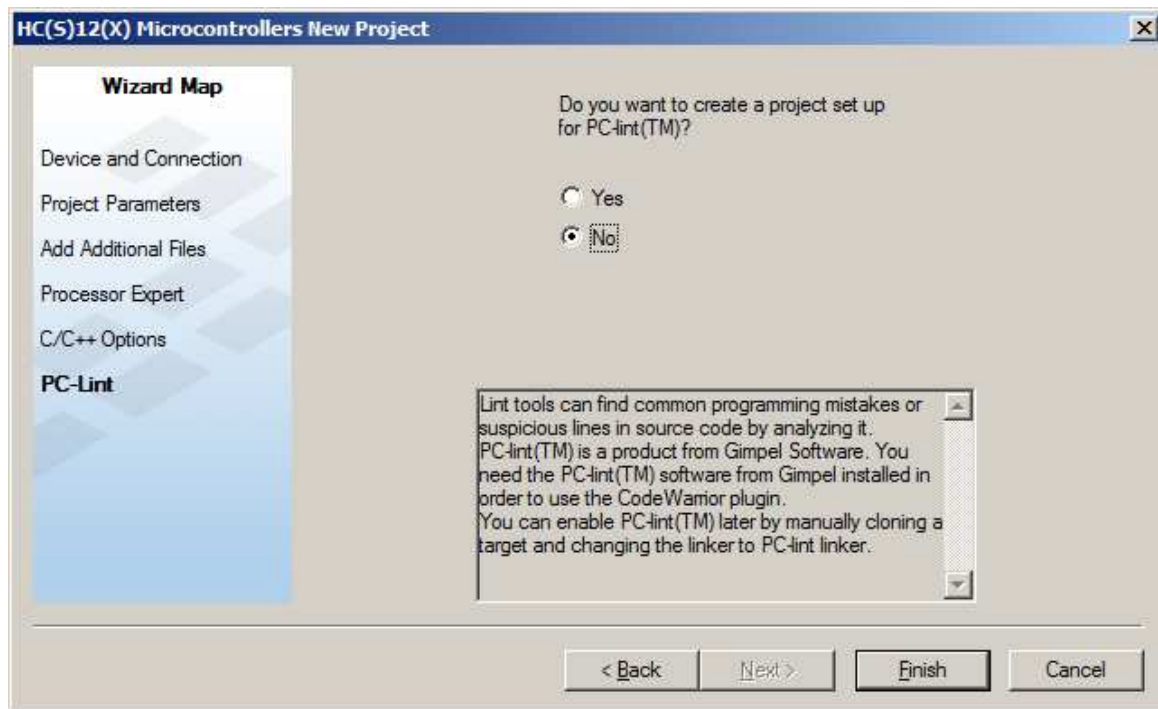



Figure 1.21 New Project - PC Lint

1.3.3 True-Time Simulator & Real-Time Debugger

To start debugging you can use the debug icon  from the CodeWarrior toolbar or the project toolbar. The True-Time Simulator & Real-Time Debugger application will open with settings and debug symbols preloaded as in Figure 1.22.

Alternatively you can manually open the debugger from its install folder: "C:\Program Files (x86)\Freescale\<CW_install_folder>\prog\hiwave.exe". In this case you have to load the configurations and the debug symbols manually:

- To load the configuration, go to "File → Open Configuration..." or directly press Ctrl+O. In the dialog that opens select the project file (*.ptj) which will be located in the root folder of your project after building the project. The file will be named according to the configuration it was built for (e.g., Simulator.ptj or SofTec.ptj).
- To load the debug symbols, go to "File → Load Application..." or directly press Ctrl+L. Select the binary file (*.abs or *.elf) in the dialog that opens. This file should be located in your project's bin folder (<project_path>\bin). As in the case of configuration files this file will also be named according to the configuration it was built for (e.g., Simulator.abs or SofTec.abs).

The True-Time Simulator & Real-Time Debugger default configuration is organised in 8 sub-windows: Source, Procedure, Data:1, Data:2, Assembly, Register, Memory, Command.

- Source window – displays the source code
- Procedure window – displays the chain of functions and procedures called until the current moment. This list is usually known as a *call stack* or *call chain* and is displayed in reversed order – the last function called is on the top.
- Data:1 and 2 windows – in the default configuration the Data:1 window displays the global variables and the Data:2 window displays the local variables.
- Assembly window – displays the source code in assembly

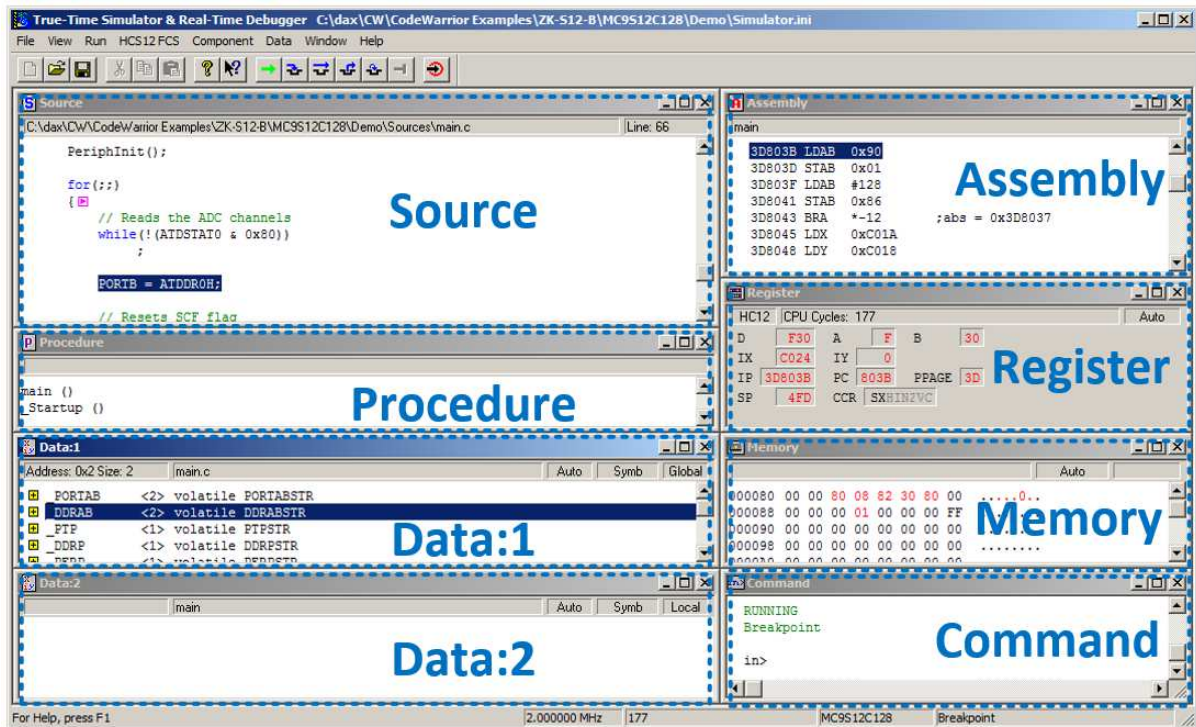


Figure 1.22 True-Time Simulator & Real-Time Debugger

- Registers window – displays the following MCU registers: D, A and B (accumulators), IX, IY (16 bit index registers), IP, PC (Instruction Pointer, Program Counter), PPAGE, SP (Stack Pointer), CCR (Condition Code Register). The value of the registers can be modified by double clicking in the value field. The registers display format can be modified from the contextual menu as shown in Figure 1.23.

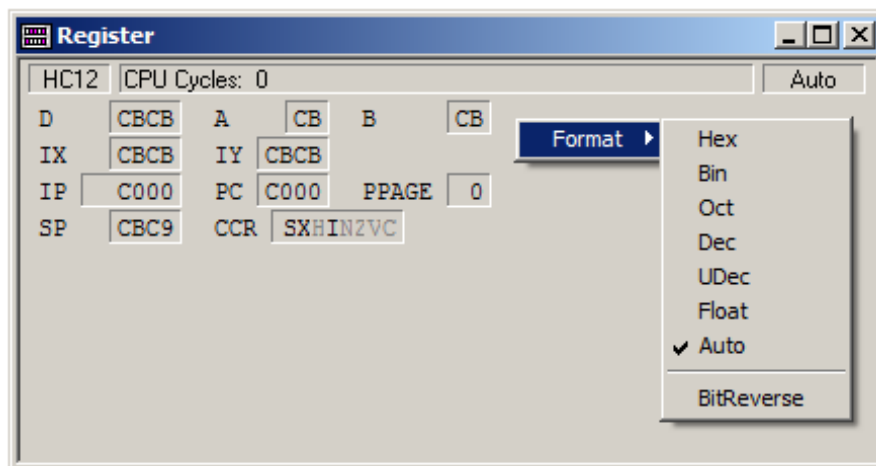


Figure 1.23 True-Time Simulator & Real-Time Debugger - Register window

- Memory window (Figure 1.24) – displays the MCU memory. Besides the normal memory data the following characters can also be seen in the memory window: “-” byte not configured, “r” byte not accessible (MCU is running) and “p” byte cannot be read or read and written.

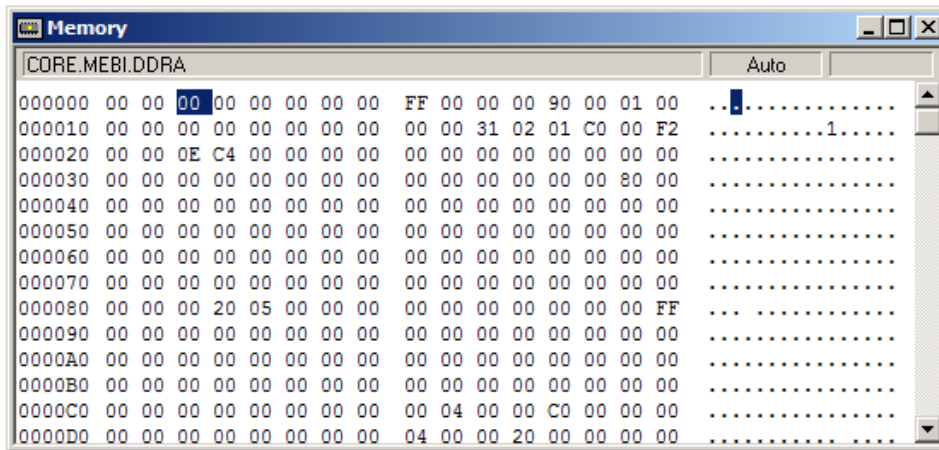


Figure 1.24 True-Time Simulator & Real-Time Debugger - Memory window

- Command window – the commands executed by the debugger are prompted here (Figure 1.25).

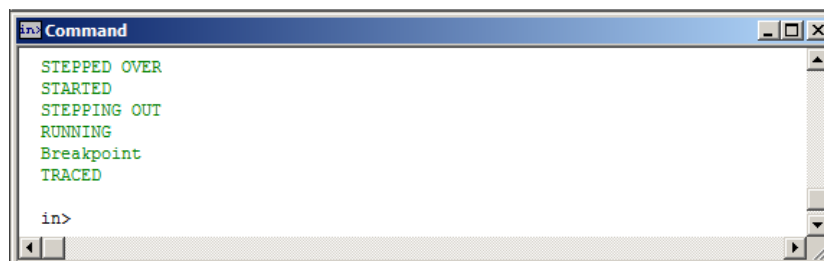


Figure 1.25 True-Time Simulator & Real-Time Debugger - Command window

For controlling the execution of the currently loaded program the following commands can be used: Start/Continue, Restart, Halt, Single Step, Step Over, Assembly Step, Assembly Step Over, Assembly Out and Halt. These commands can be accessed from the main menu (Figure 1.26), from the toolbar (Figure 1.27) or by using the associated shortcuts.

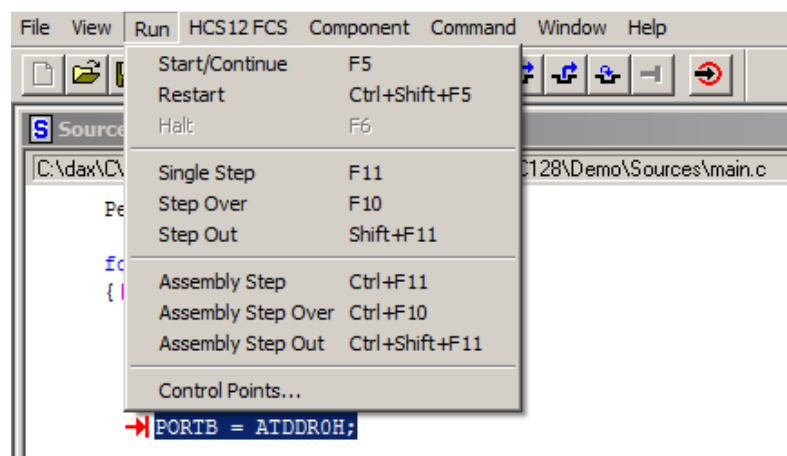


Figure 1.26 Debug commands - Run menu

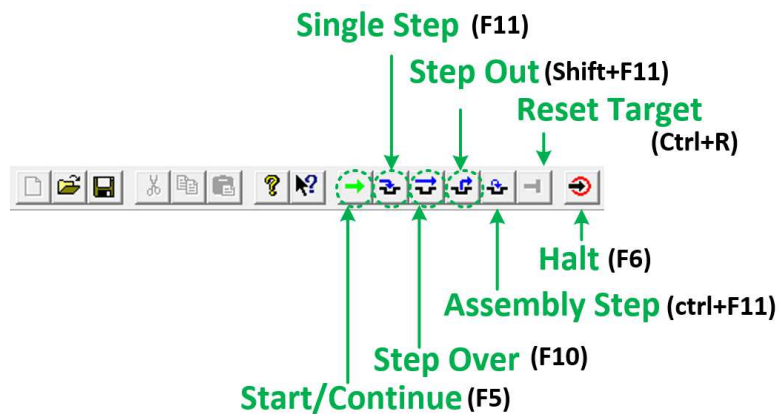


Figure 1.27 True-Time Simulator & Real-Time Debugger toolbar

- **Start/continue (F5)** - starts/continues the execution of the currently loaded code on the target. The application runs until a breakpoint or watch point is reached or a halt command is sent by the user.
- **Single step (F11)** – Can be used just when the target application is halted. It performs a single instruction from the Source window. If the next instruction is a function call the debugger will step into the function.
- **Step over (F10)** – Behaves in the same manner as the Single step command with the remark that when the next instruction is a function call it will execute the function.
- **Step out (Shift F11)** – In the case the application execution is stopped in a function, by using this command, the rest of the function code will be executed and the execution is halted at the first instruction that follows the function call.
- **Assembly step (Ctrl+F11)** – Executes a single assembly instruction then halts the execution.
- **Halt (F6)** – Halts the application execution.
- **Reset target (ctrl+r)**


Breakpoints

Breakpoints are the most important type of control points used in SW debugging. Several types of breakpoint are supported by the CodeWarrior debugger:

- *Temporary breakpoints* – they halt the code execution at the first encounter of the breakpoint
- *Permanent breakpoints* – they halt the code execution each time the breakpoint is reached
- *Conditional breakpoints* – they halt the code execution when the breakpoint is reached and a predefined condition is met
- *Counting breakpoints* – they halt the code execution after the associated instruction executes a specified number of times

Permanent break points can be set by using one of two possible approaches:

1) Right click+“Set Breakpoint”



- Put the mouse pointer in the row on which you want to insert the break point.
- Right click to display the Source window context menu
- Select “Set Breakpoint”,  symbol should appear in the code.

2) P + left mouse button

- Put the mouse pointer in the row on which you want to insert the break point
- hold down the left mouse button
- Press the P key, and release the mouse button

There are several different ways to set a temporary breakpoint:

1) Converting a permanent breakpoint to a temporary breakpoint

- Create a permanent breakpoint
- Right click anywhere in the Source window, select “Show Breakpoints” → the Controlpoints Configuration window should appear
- In the Controlpoints Configuration window select the breakpoint you want to activate as temporary and check the “Temporary” checkbox. Click Ok. The  should become .

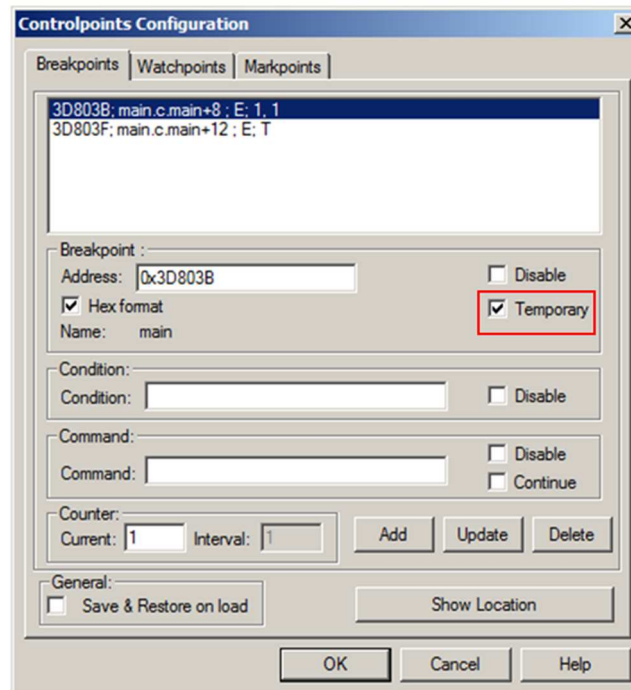


Figure 1.28 Configure a temporary breakpoint from Control Configuration, breakpoints tab

2) Run to Cursor

- Put the mouse pointer in the row with the instruction where you want to insert a breakpoint.
- Right click → the Source window context menu appears
- Select “Run to Cursor”

3) T + left mouse button

- Put the mouse pointer in the row with the instruction where you want to insert a breakpoint, hold down the left mouse button
- Press the T key, and release the mouse button

For setting conditional breakpoints follow these steps:



- Create a breakpoint
- Open the Controlpoints Configuration window
- Add the breakpoint activation condition (use C syntax) in the condition field
- The  should become .

Figure 1.29 illustrates how to set up a conditional breakpoint to break execution only if the value of the *enCnt* variable equals 0 at the time the breakpoint is reached.

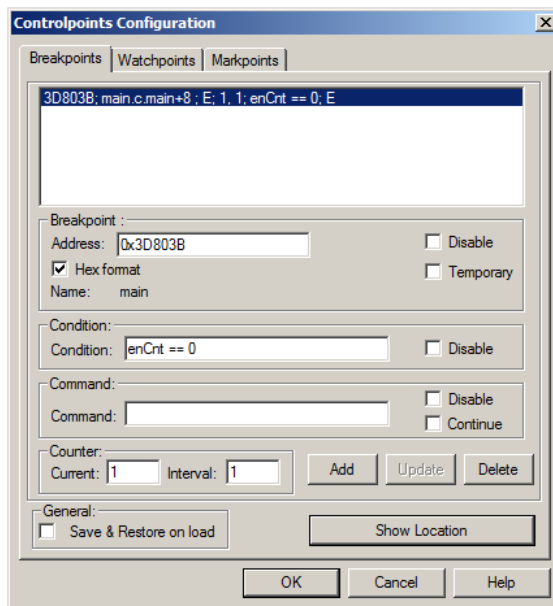


Figure 1.29 Configure a conditional breakpoint from Control Configuration, breakpoints tab

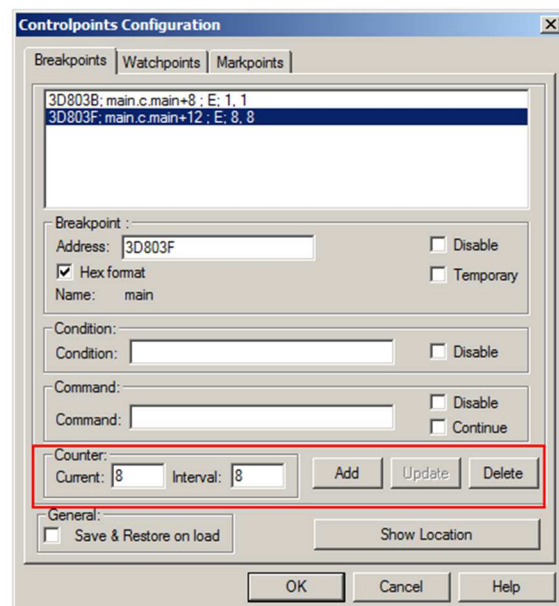




Figure 1.30 Configure a counting breakpoint from Control Configuration, breakpoints tab

When setting counting breakpoints the following steps have to be taken as suggested in Figure 1.30:

- Create a breakpoint
- Open the Controlpoints Configuration window
- Update the Counter interval filed, click update.
- The  should become .

Watchpoints

Watchpoints are control points associated with the MCU memory access. They offer the possibility to halt the program execution when a selected memory range is accessed. There are five types of watchpoints that can be set in the debugger:

- *Read Access*—halts the program when the “watched” memory locations are read.
- *Write Access* – halts the program when the “watched” memory locations are written.
- *Read/Write* – halts the program when the “watched” memory locations are read or written.
- *Counting* – applicable to any type of watchpoint: Read, Write, Read/Write.

A counter initialized by the user is associated to the memory access. This counter decrements when the specified type of memory access is detected. When the counter reaches 0 the program execution is halted. The counter is then automatically reloaded with the value initially set by the user.

- *Conditional* – applicable to any type of watchpoint: Read, Write, Read/Write.
The program execution is halted the specified type of memory access is detected and a condition imposed by the user is true.

For adding a watchpoint select the variable (in the Data window) or a memory range (in the Memory window) and right click the selected area (the contextual menu should appear). Next, select “Set Watchpoint” which will create a read/write watchpoint.

To configure a watchpoint:

- Right click on the Memory window, select “Show Watchpoints...”

- Specify the type of memory access: read, write, read/write. In the Memory window the memory locations that are monitored are underlined with different colours corresponding to the 3 access monitoring types.

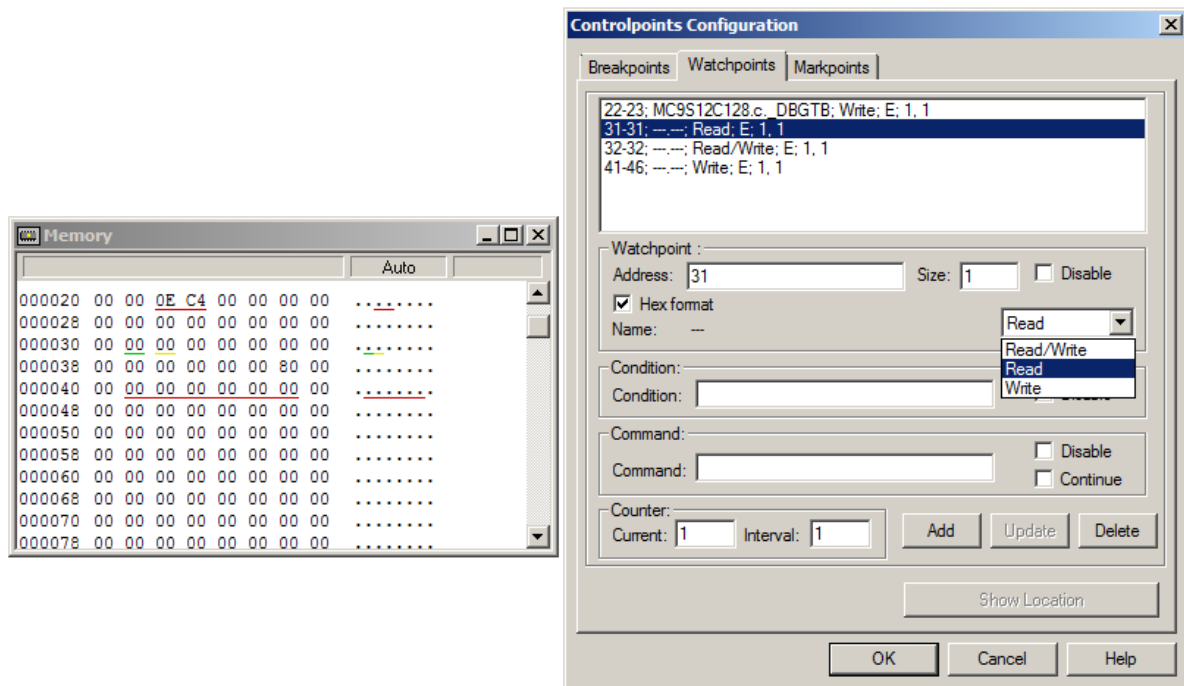


Figure 1.31 Configure a watchpoint (Read, Write, Read/Write access) from Control Configuration, watchpoints tab

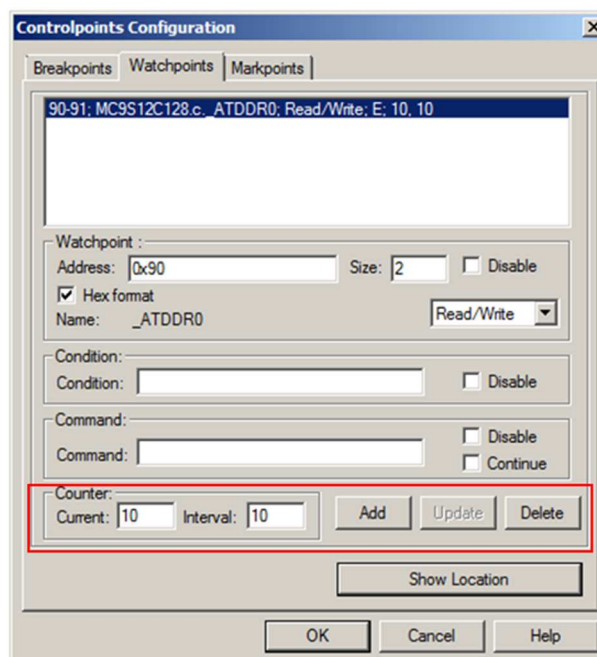


Figure 1.32 Configure a counting watchpoint from Control Configuration, watchpoints tab

The process of creating counting watchpoints is similar to the one used for counting breakpoints:

- Create a watchpoint
- Open the Controlpoints Configuration window, Watchpoint tab
- Update the Counter interval field, click update.

1.3.4 The Demo Project

The Starter Kit Demo (Code snippet 1) project provides a quick introduction in programming the S12 MCU. The demo project configures the analogue-to-digital converter module to capture and convert an input signal generated by the use of the PAD00 potentiometer located in the Inputs section of the development board. The conversion result is then displayed in binary format on the LEDs from the Outputs section.

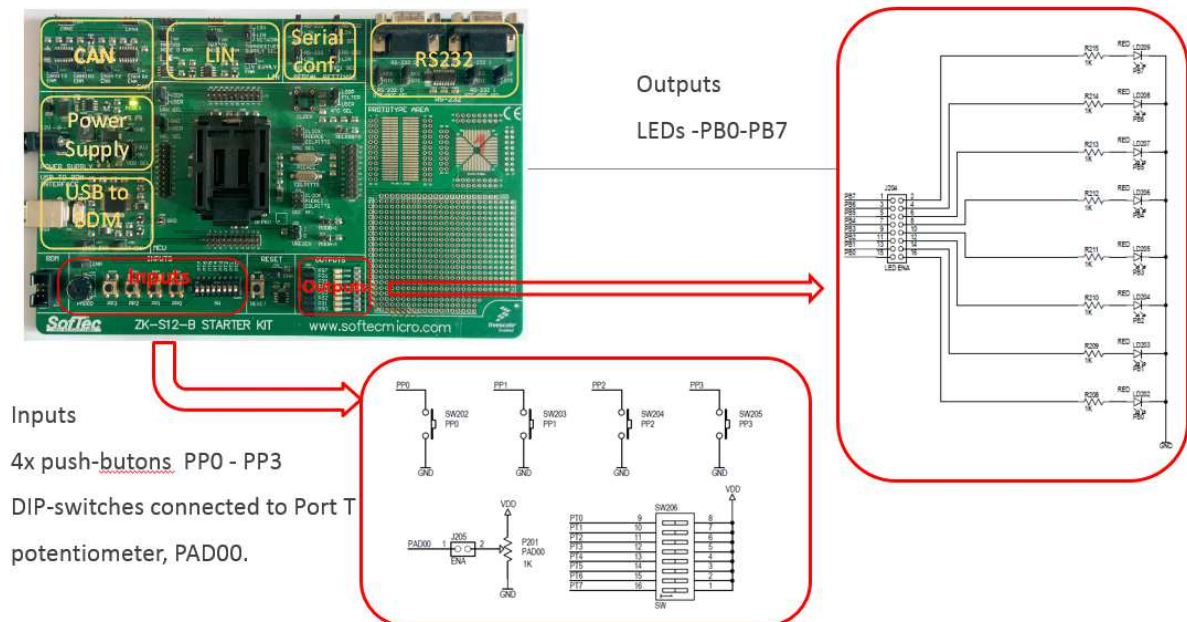


Figure 1.33 ZK-S12-B Input/Output sections

All module configurations are made in the *PeriphInit* function. Additionally to the configurations needed for Port B (connected to the LEDs) and for the ATD module, this function also configures ports P and T corresponding to the push-buttons and DIP switches from the Input section of the board.

Port B pins[7..0] are configured as output and are connected to the outputs section LEDs (Figure 1.33). Port P pins [3..0] are configured as inputs (pull-up enabled) and are connected to 4 four push-buttons (Inputs section). Port T pins [7..0] are configured as inputs (pull-down enabled) and are connected to 8 DIP-switches (Inputs section).

PAD00 (AN00) is configured as analog input channel for the ADC convertor (8 bit resolution, continuous conversion sequence mode). PAD00 is connected to the potentiometer in the inputs section.

EXERCISE 1.1 Open the Demo project corresponding to the microcontroller on your board. Depending on the microcontroller type, the project will include the corresponding header file which contains register definitions. Code snippet 1 shows the main.c file for the S12C128 microcontroller. For other targets, the difference will be in the included header (`#include "mc9s12c128.h"` will be replaced as necessary) and the pragma defining the employed microcontroller derivative. Open the main.c file and look over the configurations made to the different registers as indicated by the comments that sit beside them.

EXERCISE 1.2 Compile the project and download the binaries on the microcontroller using the debugger. Set a breakpoint at the beginning of the *main* function and run the program step by step looking at how the microcontroller registers change upon each step. Also check the assembler code generated for the corresponding C instructions in the Assembly window. How many assembler instructions are executed for each C instruction? Try to explain why for several assembler instructions are executed for a line of C code.

```

#include <hidef.h>
#include "mc9s12c128.h"
#pragma LINK_INFO DERIVATIVE "mc9s12c128"

// Peripheral Initialization

void PeriphInit(void)
{
    // Configures PB[7..0] as output
    PORTB = 0x00;
    DDRB = 0xFF;

    // Configures PP[7..0] port as input and enables pull-ups on PP[3..0] port
    PTP = 0x00;
    DDRP = 0x00; // set Port P as input
    PERP = 0x0F; // PERP (Port P Pull Device Enable Register) – Enable PP[3..0]
    PPSP = 0x00; // PPSP (Port P Polarity Select Register), 0 = pull-up

    // Configures PT[7..0] port as input and enables pull-downs on PT[7..0] port
    PTT = 0x00;
    DDRT = 0x00; // set Port T as input
    PERT = 0xFF; // Port P Pull Device Enable for PT[7..0]
    PPST = 0xFF; //PPSP (Port P Polarity Select Register), 1 = pull-down

    // Configures the ATD peripheral
    // (1 conversions per sequence, 8 bit resolution, continuous conversion)
    ATDCTL3 = 0x08; // 1 conversions per sequence
    ATDCTL4 = 0x82; // 8 bit resolution
    ATDCTL2 = 0x80; // Normal ATD functionality
    ATDCTL5 = 0x30; // analog input channel configured as AN00 (PAD00)

    EnableInterrupts;
}

// Entry point

void main(void)
{
    PeriphInit();

    for(;;)
    {
        // Reads the ADC channels
        while(!(ATDSTAT0 & 0x80)) // Check the Sequence Complete Flag (SCF) from Bit 7 from ATDSTAT0 reg
            ; // when SCF is set to 1 => conversion is completed

        PORTB = ATDDR0H;

        // Resets SCF flag
        ATDSTAT0 = 0x80;
    }
}

```

Code snippet 1.1 Demo Project, main.c file

EXERCISE 1.3 Make a copy of the Demo project and adapt it to turn on LEDs in a progress-bar manner, i.e. each LED is turned on when a voltage threshold is reached and stays on until the voltage drops under that threshold. Calculate intermediary thresholds knowing that the threshold for the first LED (PBO) is 0V and the threshold for the last LED (PB7) is 4.375V given a [0, 5]V input voltage range.

2 S12 I/O PORTS

All microcontrollers are equipped with a set of I/O pins which are used to assure the communication with peripherals. To simplify usage and configuration these pins are usually organised in groups of 8-bit (or 8-bit multiples, e.g.: 16, 32) named ports. Besides general purpose input/output functionalities ports may have additional dedicated functions (e.g.: ADC input, PWM output, serial LIN or CAN communication).

In S12 microcontrollers the interface between I/O ports and the peripheral modules is handled by the Port Integration Module (PIM). The available ports will vary depending on the S12 family member. Table 2.1 shows the available ports for each of the microcontrollers found in the ZK-S12 development kit.

MCU	Port													
	A	B	C	D	E	H	J	K	M	S	P	T	AD0/AD	AD1
S12C	✓	✓	-	-	✓	-	✓	-	✓	✓	✓	✓	✓	-
S12D	✓	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S12XD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.1 Port availability depending on microcontroller

The port availability can also vary depending on the chip packaging. Packages with a larger amount of pins will have more ports available. All port pins can act as general purpose input/output and some can act as special inputs or outputs for a certain module. For compatibility reasons combination of port pins and special module functionality available in lower pin count packages are the same in packages with a larger pin count.

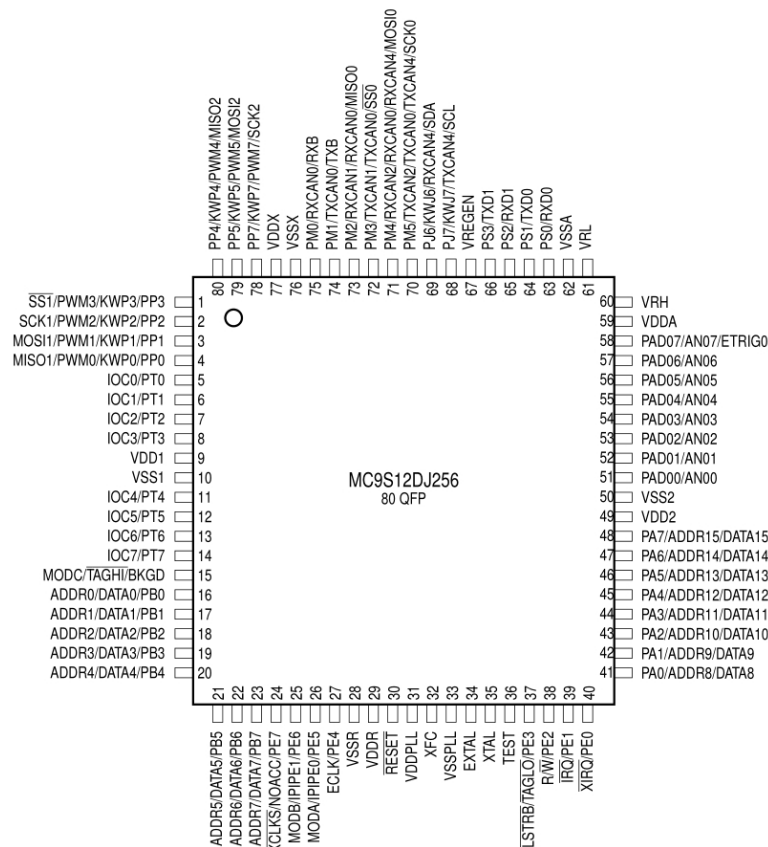


Figure 2.1 S12DJ256 80-pin QFP pinout, from [4]

To get pin locations check the data sheet for the pinout of your package configuration. For example, the ZK-S12-B development board uses 80-pin QFP chips so this is the packaging to be checked for the pin assignments. Figure 2.1 shows the pinout of an S12DJ256 chip in the 80-pin QFP configuration.

A set of registers is available for configuring each port. Some of these registers are common to all ports. Each bit in a port control register affects a corresponding pin, i.e. for a register controlling port B, bit n sets up pin n of the same port. In what follows, where x is used in naming a register it represents a placeholder for the port name.

2.1 GENERAL I/O INTERFACING

Circuit electrical characteristics of the I/O pins must be considered when interfacing the microcontroller to other electrical components. The electrical characteristics of each circuit are presented in their corresponding datasheets.

Three main characteristics should be considered when compatibility is assessed: voltage level, current drive and timing:

- *Voltage level compatibility* issues come from the fact that various integrate circuit technologies use different voltage levels. Voltage level characteristics are presented for each circuit as high and low values for the input and output voltages. For a circuit X to be capable of driving circuit Y it is necessary for the output high voltage of circuit X to be higher than input high level of Y and for the output low voltage of X to be lower than the input low voltage of circuit Y;
- *Current drive compatibility* issues arise from the inability of a microcontroller to supply or to sink the current needed in interfacing with another circuit. When connecting external circuitry to a microcontroller I/O pin one must assure that the pin can supply and sink the current needed by the interfacing circuit and that the total current required by all the connected devices does not exceed the microcontroller maximum rating;
- *Timing compatibility* are encountered when an I/O pin does not satisfy the sample and hold times while driving peripherals that contain latches or flip-flops. In this case the specified sample and hold times specified for the connected circuit must be considered. If multiple latches and flip flops are cascaded, the overall time delay must be satisfied.

Note that failing to satisfy the absolute maximum ratings of a microcontroller will lead to damaging the I/O interfacing circuitry or even the entire microcontroller. Relevant information on the effects of exceeding ratings given in the electrical specification of S12 microcontrollers is contained in [9]. This application note shows that S12 microcontrollers can handle electrical characteristics outside the recommended ratings. Even though negative effects of such operating conditions are not immediately visible the performance and lifetime of the device can be affected.

The electrical behaviour of I/O pins in relation to pull-up and drive capabilities can be configured by the user:

- *Pull-up* resistors protect high impedance input pins from oscillating when unconnected and for them to take a high or low state. The S12 chips provide internal pull circuitry for port pins. Core pins (ports A, B, E and K) are fitted with pull-up resistors that can be enabled or disabled. Other ports (eg. ports T or S) have more complex circuitry that can enable the use of either pull-up or pull-down circuitry or to disable the use of pull devices;
- *Reduced drive* can be configured on pins to lower the power consumption and reduce electromagnetic interference if the load connected on the pin permits it. The reduced drive strength is 1/3 of the full drive strength. On S12 the reduced drive mode can be configured on a per-port manner for the core ports (A, B, E and K) and in a per-pin manner for other ports.

2.2 PORTS A, B, C AND D

Ports A, B, C and D along with ports E and K are part of the Multiplexed External Bus Interface (MEBI) sub-block of the HCS12(X) core [8]. Ports C and D are part of the HCS12X core and are available only for S12X sub-family members. Besides their general purpose I/O functionality, ports A and B can be used for time-multiplexed addresses or data. On S12X microcontrollers ports A and B are used only for addressing while C and D are used for data I/O. To use these ports, or any other port, for general purpose I/O one must select if the port should be used as an output or as an input. This is done by using the data direction register (DDR_x).

Data direction register (DDR_x) – This register is used to set up a pin as an input, by writing ‘0’ to the corresponding bit, or as an output, by setting the bit to ‘1’.

The port data register (PT_x) should be used for reading from or writing to the port.

I/O register (PT_x) – This register holds the port value if the port is used as a general input/output port. Writing to this register when the port direction is set to output changes the output values on the corresponding pins while writing when the direction is set to input has no effect.

Reading from this register when the port direction is set as an input returns the values of the pins. When the port direction is configured as an output reading this register gives the register content.

EXAMPLE 2.1 Write the C code needed to configure all port B pins for output and set ‘1’ only on even pins.

Solution: To set the data direction we need to set the DDRB register for output usage. This means all bits have to be set to ‘1’. Next, to output the required value we need to write a ‘1’ only for the even bits from the PTB register. This means the binary value that must be written in the PTB register is “10101010” which give 0xAA when converted in hex. The following code snippet presents the solution:

```

DDRB = 0xFF; // set port direction to output
PTB = 0xAA; // set ‘1’ to even pins and ‘0’ to odd pins

```

When using the pull-up registers for the pins of the core ports, the pull-up control register (PUCR) should be configured. Pull-ups are assigned per port and will apply to any pin configured as an input in the corresponding port. Setting PUP_xE to ‘1’ will enable pull-up resistors on port x pins.

PUCR

7	6	5	4	3	2	1	0
PUPKE	-	-	PUPEE	-	-	PUPBE	PUPAE

The reduced drive register (RDRIV) is used to select reduced drive for the pins associated to core ports. This action results in reduced power consumption and reduced RFI (radio-frequency interference) at the cost of a slight increase in transition time. Set a ‘1’ in the corresponding RDP_x port to activate reduced drive for associated ports according to the register illustration below.

RDRIV

7	6	5	4	3	2	1	0
RDPK	-	-	RDPE	-	-	RDPB	RDPA

2.3 PORT E

Pins on port E can be used for controlling the external bus and as interrupt inputs. When not used for one of these functionalities port E pins can be used as general purpose I/O pins excepting PTE bits [1:0] which can only be used as inputs. For setting up general purpose I/O functionality, direction and data registers presented for ports A, B, C and D are also available for port E. To assign functions to the port pins when in the expanded mode the port E assignment register (PEAR) should be used.

PEAR

7	6	5	4	3	2	1	0
NOACCE	-	PIPOE	NECLK	LSTRE	-	RDWE	-

- **NOACCE (No Access output enable)** – set ‘0’ to this bit to use PTE7 as general purpose I/O pin. Otherwise, PTE7 is an output and will indicate if the cycle is a CPU free cycle.
- **PIPOE (Pipe signal Output Enable)** – a ‘0’ set to this bit will configure PTE [6:5] as general purpose I/O. Writing ‘1’ to this bit sets PTE [6:5] as outputs which indicate the state of the instruction queue.
- **NECLK (No external E-clock)** – write ‘0’ to this bit to set PTE4 as the input of the external E-clock. When set to ‘1’ PE4 is a general purpose I/O pin.
- **LSTRE (Low strobe (LSTRB) enable)** – Write ‘0’ for using PTE3 as a general purpose I/O pin or ‘1’ to have PTE3 as the LSTRB bus-control output, when the HCS12 is not in single-chip or normal expanded narrow modes.
- **RDWE (Read/Write Enable)** – for using PTE2 as a general purpose I/O pin write ‘0’ to this bit, otherwise PTE2 is configured as the R/W pin. In single-chip modes, RDWE has no effect and PTE2 is always a general purpose I/O pin. R/W is used for external writes.

The MODE register can be used to select between several chip operation modes. In addition it also provides options like the visibility of internal operations on the external bus and Port E and K emulation.

MODE

7	6	5	4	3	2	1	0
MODC	MODB	MODA	-	IVIS	-	EMK	EME

- **MODC, MODB, MODA** – mode select bits. Set these bits according to the Table 2.2 to configure various operation modes.

MODC	MODB	MODA	Mode
0	0	0	Special single chip
0	0	1	Emulation narrow
0	1	0	Special test
0	1	1	Emulation wide
1	0	0	Normal single chip
1	0	1	Normal expanded narrow
1	1	0	Special peripheral
1	1	1	Normal expanded wide

Table 2.2 Selecting chip operation modes

- **IVIS (Internal Visibility)** – ‘0’ configures no visibility of internal bus operations on external bus, while ‘1’ makes internal bus operations visible on external bus.

- **EMK (Emulate Port K)** – Set this bit to ‘0’ to make PTK and DDRK present in the memory map and port K usable in general I/O. When setting it to ‘1’ while in expanded mode, PTK and DDRK will be removed from the memory map.
- **EME (Emulate Port E)** – Set this bit to ‘0’ to make PTE and DDRE present in the memory map and Port E usable for general I/O. When setting it to ‘1’ while in expanded mode or special peripheral mode, PTE and DDRE will be removed from memory map, which allows the user to emulate the function of these registers externally.

External bus interface control register (EBICTL) – Only bit 0 of this register can be set and it controls the stretching of the external E-clock. Clock stretching can be achieved by setting this bit to ‘1’.

2.4 PORT K

Port K is available only in the H subfamily. Its pins can be used as general purpose I/O pins by employing the corresponding data direction register (DDRK) and port data register (PTK). In expanded mode Port K pins are used as expanded address, emulated chip select and external chip select signals.

2.5 PORT T

Port T also has an associated data direction register (DDRT) and a port data register (PTT). The input register PTIx is available for all registers which are not part of the S12 core.

Input register (PTIx) – This is a read-only register which returns the value of the pins.

The drive strength of the port T pins can be set RDRT register which is available also for other ports.

Reduced drive register (RDRx) – Use this register to configure the drive strength of ports. Write ‘0’ to select full drive strength as output or ‘1’ to configure 1/3 of the full drive strength.

Enabling pull-up and pull-down devices for non-core ports is configured by the use of the PERx register. To select which kind of device should be activated for each pin use the polarity select register (PPSx).

Pull device enable register (PERx) – This register turns on or off the usage of pull-up and pull-down devices. Writing a ‘1’ to the corresponding pin enables pull-up or pull-down device usage, while a ‘0’ disables it.

Polarity select register (PPSx) – This register is used to select between a pull-up and a pull-down being used for a pin. Write ‘0’ to the corresponding bit to have a pull-up device or a ‘1’ to have a pull-down device.

EXAMPLE 2.2 Give the instructions (written in C code) for setting the following characteristics for port T: reduced drive strength and pull-up device connected for all port pins.

Solution: Setting reduced port drive strength is straightforward and requires writing ‘1’ to all bits of the RDRT register. To set up the pull-up device, this must first be enabled in the PERT register and then the polarity set accordingly in the PPST register. The following code sequence satisfies the requirements:

```
RDRT = 0xFF; // set port T drive strength to 1/3 of full strength
PERT = 0xFF; // enable pull-up or pull-down devices on all pins
PPST = 0x00; // select pull-up device
```


In addition to general purpose I/O functionality, port T pins are associated to the Timer-counter unit and can be used for input capture or output compare action pins. Details on this functionality are presented in Chapter 6 which is dedicated to the timer-counter unit.

For the S12C sub-family members port T pins are also associated to the pulse width modulation (PWM) unit. To select whether pin functionality is to be controlled by the timer or by the PWM the port T module routing register (MODRR) must be used.

MODRR

7	6	5	4	3	2	1	0
-	-	-	MODRR4	MODRR3	MODRR2	MODRR1	MODRR0

- **MODRRx (Module Routing)** – If set to ‘1’ corresponding pin is connected to the PWM module. Otherwise it will be connected to the timer module.

2.6 PORT S

Port S pins can be used as serial interface signals for SCI or SPI communication. When used for general purpose I/O port S makes use of registers associated to port T: PTS, DDRS, PTIS, RDRS, PERS and PPSS. In addition, port S has a Wired-OR Mode register (WOMS).

Wired-Or Mode Port x Register (WOMx) – Set a bit to ‘1’ to make output buffers operate as open-drain outputs for the corresponding pin. When setting to ‘0’ output buffers behave as push-pull outputs.

2.7 PORT M

Besides having the general purpose I/O functionality supported by the registers which are also contained in port S (DDRM, PTM, PTIM, RDRM, PERM, PPSM and WOMM), port M also has a module routing register (MODRR) which defines rerouting of CAN 0, CAN4, SPI0, SPI1, and SPI2. In S12C sub-family members, which only have one CAN and one SPI channel, MODRR is used for port T pin routing.

MODRR

7	6	5	4	3	2	1	0
-	MODRR6	MODRR5	MODRR4	MODRR3	MODRR2	MODRR1	MODRR0

Depending on the required modules, MODRR routing should be configured according to the values presented in Table 2.3. As pin routing will vary depending on the options available on each individual S12 family member you should consult the datasheet of the particular chip you are using in your application.

Module	MODRR							Related pins	
	6	5	4	3	2	1	0	RXCAN	TXCAN
CAN0	-	-	-	-	-	0	0	PM0	PM1
	-	-	-	-	-	0	1	PM2 ¹	PM3 ¹
	-	-	-	-	-	1	0	PM4 ²	PM5 ²
	-	-	-	-	-	1	1	PJ6 ³	PJ7 ³
CAN4	-	-	-	0	0	-	-	PJ6	PJ7
	-	-	-	0	1	-	-	PM4 ⁴	PM5 ⁴
	-	-	-	1	0	-	-	PM6 ⁵	PM7 ⁵
	-	-	-	1	1	-	-	Reserved	

								MISO	MOSI	SCK	/SS
SPI0	-	-	0	-	-	-	-	PS4	PS5	PS6	PS7
	-	-	1	-	-	-	-	PM2 ⁶	PM4 ⁷	PM5 ⁷	PM3 ⁶
SPI1	-	0	-	-	-	-	-	PP0	PP1	PP2	PP3
	-	1	-	-	-	-	-	PH0	PH1	PH2	PH3
SPI2	0	-	-	-	-	-	-	PP4	PP5	PP7	PP6
	1	-	-	-	-	-	-	PH4	PH5	PH6	PH7

Table 2.3 Module pin routing configuration

Notes:

1. Routing to this pin takes effect only if CAN1 is disabled
2. Routing to this pin takes effect only if CAN2 is disabled
3. Routing to this pin is only possible on S12XD family members
4. Routing to this pin takes effect only if CAN2 is disabled; CAN0 is disabled if routed here
5. Routing to this pin takes effect only if CAN3 is disabled
6. Routing to this pin takes effect only if CAN1 is disabled; CAN0 is disabled if routed here
7. Routing to this pin takes effect only if CAN2 is disabled; CAN0 and CAN4 are disabled if routed here

2.8 PORTS H, J AND P

Ports J and P are available on all members of the three sub-families in the ZK-S12-B kit, while port H is not available on S12C sub-family members. These three ports can be configured using identical register sets. Basic registers found in previous ports are also used here: PTx, DDRx, PTIx, RDRx, PERx, PPSx. Two additional registers are available for ports H, J and P: the port interrupt enable register (PIEx) and the port interrupt flag register (PIFx).

Port interrupt enable register (PIEx) – Configures edge sensitive interrupts on associated port pins. Write ‘1’ to the corresponding bit to enable interrupts on a certain pin.

Port interrupt flag register (PIFx) – Interrupt flags that are set by an active edge (rising or falling depending on the PPSx register setting) on the associated input pin. Reading ‘1’ means that an active edge has occurred on the associated pin. Write ‘1’ to clear the associated flag.

These three ports are also associated to various modules:

- Port H is associated to SPI modules
- Port J is associated to CAN
- Port P pins can be used by the PWM and SPI modules.

2.9 PORTS AD0 AND AD1

Some S12 family members have one 8-channel A/D converter implemented while others have two. For those that have two A/D modules they are referred to as AD0 and AD1. When a single 8-channel converter is present it is referred as AD instead of AD0. These ports are used as analogue inputs to the A/D converters. When A/D conversion is not enabled they can be used as general purpose I/O ports.

Besides the usual port control registers AD0 and AD1 each have an ATD digital input enable register.

ATD input enable register (ATDxDIEN) – Configures a pin to be used as a digital input. Set a bit to ‘1’ to configure the corresponding pin as a digital input to PTADx.

EXERCISE 2.1 Display the state of the 8 DIP-switches that are connected to Port T (PT0-PT7) on corresponding LEDs connected to the Port B pins.

EXERCISE 2.2 Display the state of the four push-buttons that are connected to Port P (PP0-PP3) on the LEDs. When a Port P push-button is pressed the corresponding Port B LED should be turned ON.
Remark: Pins PP0 - PP3 are configured as pull-up inputs.

3 S12 PROGRAMMING MODEL, INSTRUCTION SET AND MEMORY ADDRESSING

This chapter brings into focus some notions on the low-level programming of the S12 microcontroller. Some motivation may be in order. When developing embedded applications you are usually required to code in a programming language such as C/C++ at a higher layer (this happens for the exercises in the other chapters here as well), which is fine because it clearly increases the speed at which you can code. However, coding at higher levels gives less insights on how the system actually works and in-depth knowledge is clearly useful when tackling a problem that is transparent to the higher layer (where you depend mostly on the compiler). Besides learning how things work, there are also a number of problems that can be more efficiently tackled from the assembly programming level, e.g., direct hardware manipulation (drivers), reducing code overhead, speed optimizations, etc.

3.1 PROGRAMMING MODEL

By programming model we refer the abstraction of the S12 core. This includes the registry structure that is discussed below and illustrated in Figure 3.1:

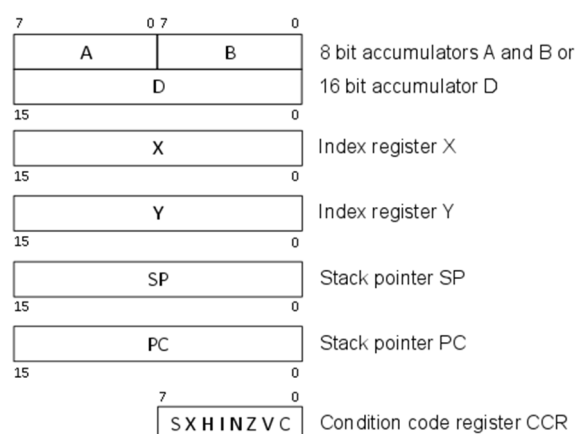


Figure 3.1 S12 core registers

- **A** and **B** – two 8 bit, i.e., 1 byte, accumulators which can be used as an extended 16 bit, i.e., 1 word, accumulator referred as **D** (the term accumulator comes from the fact that such registers were used to cumulate the results of computer operations, e.g., additions, etc.)
- **D** a 16-bit, i.e., 1 word, accumulator which results from the concatenation of **A** and **B** (see above) also referred as the double accumulator. Note that you should view **A**, **B** and **D** as your main working area of the microcontroller.
- **X** and **Y** (sometimes referred as **IX** and **IY**) two 16 bit index registers used as the name suggests in indexed addressing modes, usually involved in processing data from lists (stored in memory)
- **SP** a 16 bit stack pointer which contains the last used memory location from the stack, which is the top of the stack (this is obviously used to retrieve the information from the stack and closely resembles the other two index register)
- **PC** a 16 bit program counter which contains the address of the next instruction to be processed,
- **CCR** an 8 bit condition code register (or status register) were each bit can be 0 (reset) or 1 (set) with the following significance:

- **S (STOP Mask Bit)** - microcontroller ignores the STOP instruction and treats it as NOP (no operation),
- **X (XIRQ Mask Bit)** - enables non-maskable external interrupts,
- **H (Half-Carry bit)** - set when carry from lower to upper nibble occurs during arithmetic operations (a nibble is 4 bits),
- **I (Interrupt Mask Bit)** - enables maskable interrupts,
- **N (Negative Bit)** - set when the results of an arithmetic operation is negative,
- **Z (Zero Bit)** - set when the result of an arithmetic operation is zero,
- **V (Overflow Bit)** - set when the results of an arithmetic operation is overflow,
- **C (Carry Bit)** - set when a carry from the most significant bit occurs after some arithmetic operation.

Note that register access time is shorter than memory access time. This is why you can speed up your code if, when possible, registers are used instead of memory stored values.

3.2 INSTRUCTION SET

The instruction set of a microcontroller (or of any CPU in general) comprises all the instructions that its core can execute. This section is by no means intended to be exhaustive, please refer to reference manuals [8] and [10] for the complete instruction set. Here we only intend to give an overview of the instruction set and provide some examples. Summing up over the entire instruction set would need of course a more extensive presentation (this runs well over 30 pages in Chapter 4 of [8] so it is not our intention to cover the entire instruction set in detail). In Table 3.1 we give an overview of the MC9S12 instruction set by comparing it as presented in the [8] with what is included in this chapter (instructions from the sections written in grey italics from the core guide are not presented in this book).

Core user guide instruction set [8]	Instruction set as grouped and presented in these notes
Load and Store Instructions (§4.3.1), Transfer and Exchange Instructions (§4.3.2), Move Instructions (§4.3.3)	§3.2.1 Data transfer instructions
Add and Subtract Instructions (§4.3.4), Add and Subtract Instructions (§4.3.5), <i>Decrement and Increment Instructions (§4.3.6),</i> Multiply and Divide Instructions (§4.3.10),	§3.2.2. Arithmetic instructions
Bit Test and Bit Manipulation Instructions (§4.3.11), Shift and Rotate Instructions (§4.3.12), Boolean Logic Instructions (§4.3.8), <i>Clear, Complement, and Negate Instructions (§4.3.9)</i>	§3.2.3 Logic and bit instructions
Branch Instructions (§4.3.17), Short Branch Instructions (§4.3.17.1), Long Branch Instructions (§4.3.17.2), <i>Bit Condition Branch Instructions (§4.3.17.3),</i> <i>Loop Primitive Instructions (§4.3.17.4)</i>	§3.2.4 Branch instructions
Compare and Test Instructions (§4.3.7)	§3.2.5 Comparison instructions
Jump and Subroutine Instructions (§4.3.18), Interrupt Instructions (§4.3.19), <i>Index Manipulation Instructions (§4.3.20),</i> <i>Stacking Instructions (§4.3.21)</i>	§3.2.6 Function call instructions
<i>Load Effective Address Instructions (§4.3.22),</i> <i>Condition Code Instructions (§4.3.23),</i>	Not addressed.

<i>STOP and WAI Instructions (§4.3.24), Background Mode and Null Operation Instructions (§4.3.25)</i>	
<i>Fuzzy Logic Instructions (§4.3.13), Maximum and Minimum Instructions (§4.3.14), Fuzzy Logic Membership Instruction (§4.3.14.1), Fuzzy Logic Rule Evaluation Instructions (§4.3.14.2), Fuzzy Logic Averaging Instruction (§4.3.14.3), Multiply and Accumulate Instruction (§4.3.15), Table Interpolation Instructions (§4.3.16)</i>	Not addressed.

Table 3.1 Overview of MC9S12 instruction set as presented in the Core User Guide and in this book

How to write assembly code in CodeWarrior?

In order to write code in assembly mode you have two options. The first is to start an assembly mode project in CodeWarrior. If you are not going to use multiple files in your project starting with *Absolute assembly* rather than a *Relocatable assembly* project is in order. The CodeWarrior new project interface is pictured in Figure 3.2. The source file generated in this way for the project will contain only assembly code.

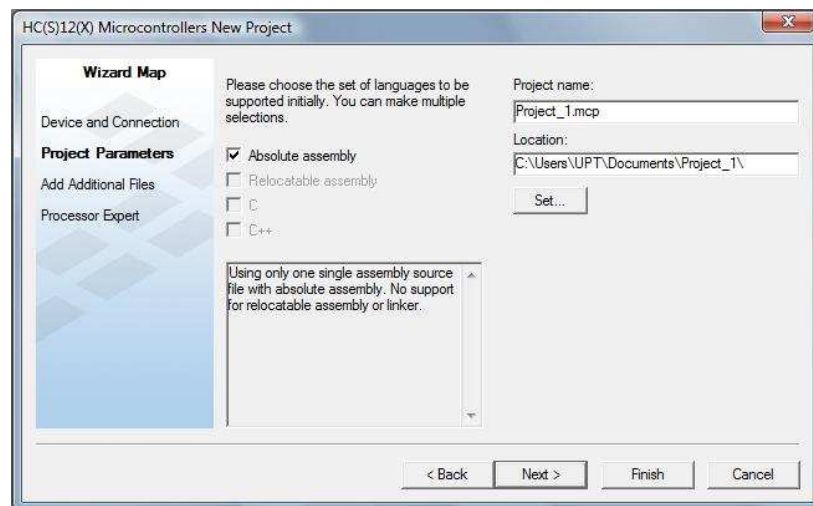


Figure 3.2 Starting a new assembly project for HC(S)12(X) in CodeWarrior

The second option is to insert asm code directly in C source files. For this simply insert the `asm{ ... }` keyword in your C code and write the instructions within the braces (curved brackets), see Code snippet 3.1. Note that some areas of the memory are protected, especially when using existing C examples from the SoftecMicro package, therefore whenever some load/store (or other) instructions do not work (i.e., memory location is not written with your specified value), the assembly instructions are not to blame but rather consider that you do not have the permission to perform that operation (starting a fresh assembly program should help you avoid such issues).

```

void main(void)
{
    asm {
        LDD #$1234
        STD $feee
        LDAA$feee
        LDAA 12
        LDAA #34
    }

    for (;;) {
    }
}

```

Code snippet 3.1 Using assembly instructions inline with C code

Structure of an assembly line

The structure of an assembly code line is detailed below. Underlined values (the label which identifies the code line and the comment field which starts with semicolon ;) are optional and the bold ones (operation code referred as mnemonic or opcode and operand) are mandatory (the operand(s) can be of course absent for instructions that do not require them):

Label **Mnemonic/Opcode** **Operand(s)** ;Comment

There are two special symbols of prime importance which are going to be use in most assembly lines. Be sure to keep them in mind:

- **\$** - indicates that the subsequent value is an hexadecimal one (without it, is treated as decimal)
- **#** - the number that follows must be treated as data and not as address (without it, data will be retrieved (stored) from (in) memory).

Some compiler directives are available for usage within assembly code. Note that these are not S12 instructions but directives for the compiler on which you will frequently rely:

- **EQU** - gives a value to a symbol (note that equ just defines a constant and does not allocate memory in contrast to the DB and DC directives next),
- **DB** (define byte) and **DC.[size]** (define constant) where size is **b** (byte), **w** (word) or **l** (4 bytes) is used to allocate storage for variables. A similar directive is **DS** but this one just allocated space for a variable without assigning it a value,
- **ORG** - sets value for the location counter where a piece of code will be, section defines a new program section.

3.2.1 Data transfer instructions

Load instructions can be used to load values in memory or immediate values into the CPU registers that were defined in the programming model. The available load instructions are listed in Table 3.2.

LDAA – loads A from memory or immediate value (8 bit)	
LDAB – loads B from memory or immediate value (8 bit)	
LDD – loads D from memory or immediate value (16 bit)	LEAS – loads effective address into SP (16 bit)
LDS – loads SP from memory or immediate value (16 bit)	LEAX – loads effective address into X (16 bit)
LDX – loads X from memory or immediate value (16 bit)	LEAY – loads effective address into Y (16 bit)
LDY – loads Y from memory or immediate value (16 bit)	

Table 3.2 Load instructions

In contrast store instructions, as presented in Table 3.3, have the ability to save the value of a CPU register to memory.

STAA – stores A in memory (8 bit)
STAB – stores B in memory (8 bit)
STD – stores D in memory (16 bit)
STS – stores SP in memory (16 bit)
STX – stores X in memory (16 bit)
STY – stores Y in memory (16 bit)

Table 3.3 Store instructions

Transfer and exchange instructions are used when we need to have CPU registers as both source and destination of the data transfer. Transfer instructions set the value of the destination register to the value in the source register without affecting the content of the source. As their name implies, exchange operations will exchange the content of one register with the content of the other one.

TAB – transfer A to B	TSY – transfer SP to Y
TAP – transfer A to CCR	TXS – transfer X to SP
TBA – transfer B to A	TYS – transfer Y to SP
TFR – transfer register (any to any of the A, B, CCR, D, X, Y, or SP)	EXG – exchange registers (any with any of the A, B, CCR, D, X, Y, or SP)
TPA – transfer CCR to A	XGDX – exchange D with X
TSX – transfer SP to X	XGDY – exchange D with Y

Table 3.4 Transfer and exchange instructions

Similarly to the register to register instructions there are also memory to memory move instructions as shown in Table 3.5.

MOVB - Move byte from memory to memory (8-bit)
MOVW - Move word from memory to memory (16-bit)

Table 3.5 Move instructions

EXERCISE 3.1 Given the following lines of assembly code, specify at the end of each step the values within the registers A and B both in hex and decimal. Please also use CodeWarrior to check that the values are stored in memory and the status of the registers is as shown in Figure 3.3 and Figure 3.4.

LDD #\$4321	A:h /.....	B:h /.....
STD \$1234	A:h /.....	B:h /.....
LDAA \$1234	A:h /.....	B:h /.....
LDAA #\$12	A:h /.....	B:h /.....
LDAA #34	A:h /.....	B:h /.....

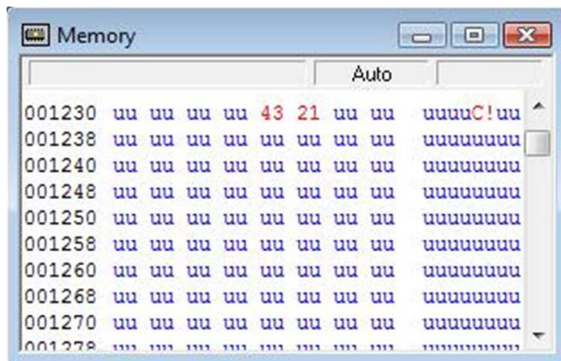


Figure 3.3 Memory as modified by some of the previous lines of code

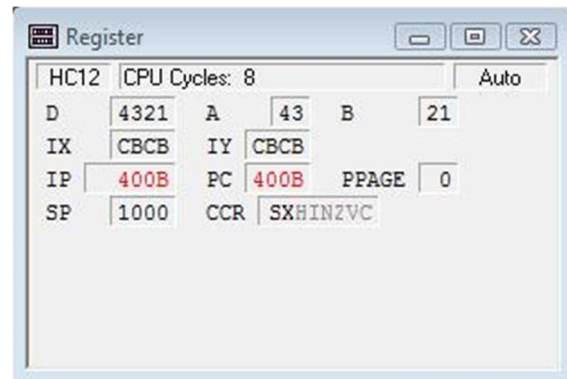


Figure 3.4 Registers as modified by some of the previous lines of code

3.2.2 Arithmetic instructions

We detail below only the basic arithmetic instructions: addition, subtraction, multiplication and divisions. Note however, that there are also instructions that can be used for incrementing and decrementing registers and even memory as well as instructions for Binary Coded Decimals (BCD).

ABA – add A to B ABX – add B to X ABY – add B to Y ADCA – add memory or imm. value and carry to A ADCB – add memory or imm. value and carry to B ADDA – add memory or immediate value to A ADDB – add memory or immediate value to B ADDD – add memory or immediate value to D	SBA – subtract B from A SBCA – subtract memory or imm. value and carry from A SBCB – subtract memory or imm. value and carry from B SUBA – Subtract memory or immediate value from A SUBB – subtract memory or immediate value from B SUBD – subtract memory or immediate value from D
---	---

Table 3.6 Add and subtract instructions

EMUL - 16 by 16 multiply (unsigned) EMULS - 16 by 16 multiply (signed) MUL - 8 by 8 multiply (unsigned) (A) EDIV - 32 by 16 divide (unsigned) EDIVS - 32 by 16 divide (signed) FDIV - 16 by 16 fractional divide (unsigned) IDIV - 16 by 16 integer divide (unsigned) (D) IDIVS - 16 by 16 integer divide (signed)

Table 3.7 Multiply and division instructions

EXERCISE 3.2 As you already know, the S12 core has 16 bit register width. Consider that you want to add the following two integers each of 32 bits: 0xEEFF FEE and 0xAABB BBAA and consider that they are stored in memory at locations \$E000 and \$0004 respectively (when programming this in CodeWarrior, since the memory is blank on your fresh project, you are requested to place these values in memory within your own code). Perform their addition on 32-bits (do not forget about the carry) and put the result starting from memory location \$0008 (highest word at the lowest address).

Solution: Below you are given the assembly source code for this addition. The code is commented but additional explanations are needed, show the values of the registers after each step (you are encouraged to use the code in CodeWarrior to see how the registers & memory change).

```
;store 1st integer in memory
```

LDD #SEEFF	A:h /..... B:h /.....
STD \$E000	A:h /..... B:h /.....
LDD #SFEE	A:h /..... B:h /.....
STD \$E002	A:h /..... B:h /.....
;store 2nd integer in memory	
LDD #SAABB	A:h /..... B:h /.....
STD \$E004	A:h /..... B:h /.....
LDD #SBBA	A:h /..... B:h /.....
STD \$E006	A:h /..... B:h /.....
;retrieve 1st word from 1st integer	
LDD \$E002	A:h /..... B:h /.....
;add to it 1st word from 2nd integer	
ADDD \$E006	A:h /..... B:h /.....
;store the result in memory	
STD \$E00B	A:h /..... B:h /.....
;retrieve 2nd word from 1st integer	
LDAB \$E001	A:h /..... B:h /.....
;add with carry to it the 2nd word from 2nd integer	
ADCB \$E005	A:h /..... B:h /.....
LDAA \$E000	A:h /..... B:h /.....
ADCA \$E004	A:h /..... B:h /.....
;store the result in memory	
STD \$E009	A:h /..... B:h /.....
;store the last carry in memory	
LDAA #\$00	A:h /..... B:h /.....
ADCA #\$00	A:h /..... B:h /.....
STAA \$E008	A:h /..... B:h /.....

EXERCISE 3.3 Consider the same exercise as previously, but this time implement 32 bit multiplication.

3.2.3 Logic and bit instructions

Below we only detail the basic logic operations: AND, OR and XOR and some bit set, clear and test instructions. Note that there are also instructions to complement and negate registers or memory.

<p>AND A - AND A with memory or immediate value</p> <p>AND B - AND B with memory or immediate value</p> <p>AND CCR - AND CCR with immediate value (clears CCR bits)</p> <p>EOR A - Exclusive OR A with memory or immediate value</p> <p>EOR B - Exclusive OR B with memory or immediate value</p>	<p>OR A - OR A with memory or immediate value</p> <p>OR B - OR B with memory or immediate value</p> <p>OR CCR - OR CCR with immediate value (sets CCR bits)</p> <p>BCLR - Clear bit(s) in memory</p> <p>BITA - Bit test A</p> <p>BITB - Bit test B</p> <p>BSET - Set bits in memory</p>
---	---

Table 3.8 Boolean instructions and bit instructions

Shift and rotate instructions are available as well. Remember that the difference between logical and arithmetic shifts is that in the arithmetic right shifting the bits are filled with the value of the carry flag C (not with 0). While left shifts behave identical on the value, whether logical or arithmetic, the flags they set in the CCR may be different in some deployments (they are identical on S12 according

to [8] and [10], but in some implementations arithmetic shifts will trigger overflows, etc., so you should pay attention on this).

LSL - Logic shift left of value from memory LSLA - Logic shift left of A LSLB - Logic shift left of B LSLD - Logic shift left of D LSR - Logic shift right of value from memory LSRA - Logic shift right of A LSRB - Logic shift right of B LSRD - Logic shift right of D	ASL - Arithmetic shift left of value from memory ASLA - Arithmetic shift left of A ASLB - Arithmetic shift left of B ASLD - Arithmetic shift left of D ASR - Arithmetic shift right of value from memory ASRA - Arithmetic shift right of A ASRB - Arithmetic shift right of B
--	--

Table 3.9 Logic and arithmetic shift instructions

ROL - Rotate left memory ROLA - Rotate left A ROLB - Rotate left B ROR - Rotate right memory RORA - Rotate right A RORB - Rotate right B

Table 3.10 Rotate instructions

3.2.4 Branch instructions

Branch instructions (and also jump instructions) can be usually split in two classes: unconditional (jump is always executed) and conditional (jumps are made only if a specific condition is met). This is the case for the S12 core which has three unconditional branch instructions BRA, its long version LBRA and the classical JMP. There are also negated versions of these, i.e., jumps that are never taken, and are useful for debugging purposes BRN, LBRN. Note that LBRA and JMP can use a larger address space (16-bit) when compared to BRA (8-bit) but these also require 1 additional clock cycle (are slower), thus you should always choose whatever is best fitted for your application. Being an 8-bit value, the parameter for the BRA instruction is an integer in the range [-128, 127]. For conditional branches there is a higher number of alternatives as pointed out in Table 1.7.

Further, besides the classical classification in unconditional vs. conditional branching, the S12 core guide [8] makes a distinct classification according to the condition that must be satisfied: i) unary branches are actions that are always taken (i.e., unconditional branches), ii) simple branches are taken when a specific bit of the CCR is set or clear, iii) unsigned branches are taken after comparing or testing unsigned values according to the bit state in the CCD, iv) signed branches are the same as unsigned branches but this time obviously with respect to signed integers.

Unary	Unary
BRA - Branch always 1 = 1 BRN - Branch never 1 = 0	LBRA - Long branch always 1 = 1 LBRN - Long branch never 1 = 0
Simple	Simple
BCC - Branch if carry clear C = 0 BCS - Branch if carry set C = 1 BEQ - Branch if equal Z = 1 BMI - Branch if minus N = 1 BNE - Branch if not equal Z = 0 BPL - Branch if plus N = 0 BVC - Branch if overflow clear V = 0 BVS - Branch if overflow set V = 1	LBCC - Long branch if carry clear C = 0 LBCS - Long branch if carry set C = 1 LBEQ - Long branch if equal Z = 1 LBMI - Long branch if minus N = 1 LBNE - Long branch if not equal Z = 0 LBPL - Long branch if plus N = 0 LBVC - Long branch if overflow clear V = 0 LBVS - Long branch if overflow set V = 1

Unsigned	Unsigned
BHI - Branch if higher (R > M) BHS - Branch if higher or same (R ≥ M) BLO - Branch if lower (R < M) BLS - Branch if lower or same (R ≤ M)	LBHI - Long branch if higher (R > M) LBHS - Long branch if higher or same (R ≥ M) LBLO - Long branch if lower (R < M) LBLS - Long branch if lower or same (R ≤ M)
Signed	Signed
BGE - Branch if greater than or equal (R ≥ M) BGT - Branch if greater than (R > M) BLE - Branch if less than or equal (R ≤ M) BLT - Branch if less than (R < M)	LBGE - Long branch if greater than or equal (R ≥ M) LBGT - Long branch if greater than (R > M) LBLE - Long branch if less than or equal (R ≤ M) LBLT - Long branch if less than (R < M) N Å V = 1

Table 3.11 Branching instructions

EXERCISE 3.4 Explain the effect of the following instructions: BRA \$FE and BNE \$FF.

At the end of next section which introduces comparison instructions, you will have the opportunity to use branches in a more practical exercise.

3.2.5 Comparison instructions

Compare and test instructions are used to set conditions for branch instructions. They consist of a subtraction performed on the pair of parameters without storing the result but with affecting the condition codes in the CCR.

CBA - Compare A to B CMPA - Compare A to memory or immediate value CMPB - Compare B to memory or immediate value CPD - Compare D to memory or immediate value CPS - Compare SP to memory or immediate value CPX - Compare X to memory or immediate value CPY - Compare Y to memory or immediate value	TST - Test memory for zero or minus TSTA - Test A for zero or minus TSTB - Test B for zero or minus
---	---

Table 3.12 Compare and test instructions

EXAMPLE 3.1 Write an ASM program which sorts a vector stored in memory. For simplicity consider the vector has a fixed size, for example 5 bytes.

Solution: For further simplicity we will use the basic bubble sort mechanism (this is good to serve as an example, but not really encouraged for practice due to lack of performance). To store data in memory we use the FCB (Form Constant Byte) directive which reserves a block of memory and sets it to predefined values and the DC.W directive which can be used to define an array of words. There are also variations of the two directives: FDB forms double byte (word), FCC forms constant character (ASCII enclosed in "") and DC.B which forms an array of bytes.

vector	FCB 3, 2, 5, 7, 1
switch	FCB 0
counter	FCB 0
addr1	DS.W 1
addr2	DS.W 1
len	FCB 5
loop_switch:	
LDAA #0	;set switch to 0
STAA switch	
LDAA #4	;set len to 4 (i.e., 5 elements from 0..4)
STAA len	

```

    LDX #0           ;set index register X and Y to walk through consecutive elements
    LDY #1
loop_vector:
    LDAA vector, X   ;load the i-th value in A
    LDAB vector, Y   ;load the i-th+1 value in B
    CBA              ;compare A to B
    BLS ok           ;branch without exchanging if lower or same
xchg:
    ;it did not branch so need to exchange A to B
    STAA vector, Y
    STAB vector, X
    LDAA #1          ;also set switch to 1 as the array is not sorted
    STAA switch
ok:
    INX              ;increment your index registers
    INY
    DEC len          ;decrement the len loop variable

    TST len          ;test if you reached the end of the array
    BNE loop_vector ;if len is not equal to 0 then loop

    TST switch
    BNE loop_switch ;if switch is 0 if not array is not sorted so loop again

```

EXERCISE 3.5 Write a sequence of asm code that can add 2 integers that are coded on n bytes (consider that n is fixed).

3.2.6 Function call instructions

The easiest way to access a subroutine is with the BSR instruction that will branch the program to the subroutine label. Note that the subroutine must end with instruction RTS. The return address from the subroutine is automatically saved and reloaded from the stack. The CALL instruction can also be used but note that this also allocates a new memory page for the subroutine.

BSR - Branch to subroutine (SP)	RTI - Return from interrupt
CALL - Call subroutine in expanded memory	SWI - Software interrupt
JMP - Jump Subroutine address	TRAP
JSR - Jump to subroutine	
RTS - Return from subroutine	
RTC - Return from call	

Table 3.13 Jump and subroutine instructions

EXAMPLE 3.2 Modify the code from the previous exercise to use a subroutine that exchanges two elements when they are unsorted. Use the stack to push all related parameters: values of the elements and their index, as well as the effective address of the vector.

Solution: Note that all parameters passed into the stack: this includes, the values, their address (of course you could simplify and pass only their address, this is just for illustration). Note the order in which they are pushed and then extracted and how the return address is saved when entering the subroutine. Also note that we are improving the previous code by: using CLR to clear switch and BSET to set the switch, use LEAX to load effective address. It may be a good idea to follow the stack with pen and paper (and the CodeWarrior IDE) at each step.

```

vector   FCB 3, 2, 5, 7, 1
switch   DC.B 0
counter  DC.B 0
addr1    DS.W 1
addr2    DS.W 1
len      DC.B 4

ret_addr DS.W 1

```

```

store_addr DS.W 1

loop_switch:
    CLR switch      ;set switch to 0, this time elegantly with clear memory instruction

    MOVB len, counter ;set len to 4 (i.e., 5 elements from 0..4)

    LDX #0          ;set index register X and Y to walk through consecutive elements
    LDY #1

loop_vector:
    LDAA vector, X  ;load the i-th value in A
    LDAB vector, Y  ;load the i-th+1 value in B
    CBA             ;compare A to B

    BLS ok          ;branch without exchanging if lower or same

    PSHX            ;save index registers, you will use them to pass the effective address
    PSHY

    PSHA            ;put data A, B, X, Y in the stack
    PSHB

    LEAX vector, X  ;put the address of the sorted vector in stack
    PSHX
    LEAY vector, Y
    PSHY

    BSR XCHG       ;branch to exchange subroutine
    BSET switch, $01 ;and also set switch to 1 as the array is not sorted, this time with a mask

    PULY            ;retrieve index registers
    PULX

ok:
    INX             ;increment your index registers
    INY
    DEC counter     ;decrement the len loop variable

    TST counter     ;test if you reached the end of the array
    BNE loop_vector ;if len is not equal to 0 then loop

    TST switch
    BNE loop_switch ;if switch is 0 if not array is not sorted so loop again

XCHG                ;exchange subroutine
    PULD            ;retrieve return address in D
    STD ret_addr    ;store D in memory

    ;PULD
    ;STD store_addr ;retrieve vector address

    PULY            ;pull Y, X, B, A - note the reverse order
    PULX
    PULB
    PULA

    STAA Y
    STAB X

    LDD ret_addr    ;retrive return address from memory
    PSHD            ;push D in stack

    RTS

```

3.3 MEMORY ADDRESSING MODES

Addressing modes are of prime importance for coding in assembly language, they refer to how the effective address of the operand is formed. The following addressing modes are present on S12:

- **Inherent addressing mode.** This is the case in which there is no address and happens for instructions that have no operand. For example: INX, CBA, etc.
- **Immediate addressing mode.** This is the case when the data is present right after the instruction and the data is always marked by the # sign. For example: LDAA #10, LDAB #\$AA, etc.
- **Direct addressing mode.** In this case the address follows right after the instruction and the address is specified by a single byte the higher one of the 16-bit address is always 0. That is, the address is in the range \$0000 to \$00FF. For example LDAA \$0F.
- **Extended addressing mode.** Here the address follows after the instruction but this time is in the full 16-bit range, e.g., LDAA \$1000, etc.
- **Relative addressing mode.** Is an addressing mode exclusively for branching instructions, that is, you will only encounter it with branching instructions. In this case the address is computed by adding to the program counter the value after the branch instruction. For example BEQ \$0A will add \$0A to the PC (note that when hitting the branch instruction, the PC is already incremented).
- **Indexed addressing mode.** As the name suggests this mode uses the index registers (IX, IY but also SP or PC) and is usually present when working with lists of elements. Here the instruction set is quite generous allowing 7 types of indexed addressing modes. These modes allow adding an increment/decrement of 3, 5, 9 or 16 bits. In the case of 3 bits increment/decrement this can be added/subtracted before or after, i.e., post/pre-incrementation and post/pre-decrementation. For example: LDAA \$2, +X will add \$2 to X and put the value from memory address [2+X] in A. On the contrary, LDAA \$2, X- will place the content from location [X] in A and only afterwards subtract (or adds if you use +) 2 from X. Since we are working on 3 bits, the value that can be added is from 1 to 8. To sum up, the 7 indexed addressing modes allowed by S12 are: i) indexed with constant 5 bit offset, ii) auto-increment/auto-decrement (on 3 bits as shown above), iii) 9-bit constant offset, iv) 16-bit constant offset, v) accumulator offset (in which values of A, B or D are added to the index), vi) 16-bit constant offset indexed-indirect and vii) accumulator D offset indexed-indirect. For vi) and vii) the indexed-indirect addressing means that the value in the index register plus the offset (constant or in D) point to the effective address.

EXERCISE 3.6 Modify the code from the previous exercise by using indexed addressing modes with predefined increments on the array.

EXERCISE 3.7 Explain in your own words the programming model of S12 and summarize the 7 types of instructions of S12.

EXERCISE 3.8 Write a program using assembly instructions that finds the minimum (or maximum) value in some predefined memory array.

EXERCISE 3.9 Starting from one of the previously solved exercises implement addition and multiplication for arbitrary n byte (or word) integers that will be treated as vectors stored in memory (we consider n to be a fixed constant in your code).

EXERCISE 3.10 Write a subroutine that returns the greatest common divisor of two integers, based on the well-known Euclidean rule: if $a=b$ return a, if $(a<b)$ $\text{gcd} = \text{gcd}(a, b-a)$ else $\text{gcd} = \text{gcd}(a-b, b)$.

4 THE INTERRUPT SYSTEM, CLOCK AND RESET GENERATION

This chapter begins by presenting fundamental concepts about interrupts – an efficient and indispensable mechanism for building reliable embedded applications – and specific features of the S12 Interrupt system. The second part of this chapter is dedicated to clock generation with the S12 Clock and Reset Generator (CRG) block.

4.1 GENERAL CONCEPTS OF INTERRUPTS

Interrupts are events that force the CPU to leave the normal execution path and perform specific activities related to the event. An interrupt can be generated externally by external circuitry or internally by the hardware associated to microcontroller subsystems and software exceptions. Interrupts have various applications:

- triggering time-critical operations
- executing periodic tasks
- performing reliable I/O operations
- handling of software errors.

In some cases it may be needed that certain interrupts are prevented from affecting the normal execution path. Most microcontrollers provide solutions for such situations by introducing maskable interrupts, i.e. interrupts that can be ignored by setting an enable flag. If no interrupts should affect program execution, the entire interrupt system can also be disabled by setting the appropriate value in the interrupt enable flag.

When an interrupt is generated a set of actions may need to be taken. For this the CPU provides service to the interrupt by executing a code sequence called an *interrupt service routine* (ISR). The start addresses of the ISRs associated to each interrupt are stored in *interrupt vectors*. Interrupt vectors are stored in a table called *interrupt vector table*. When the interrupt is generated the context data of the current execution path is first pushed onto the stack. Next, after identifying the start address of the ISR code associated with the issued interrupt, a jump is made to this address. After the provided ISR is executed, the context will be restored and the previous execution path can be resumed.

If several interrupts sources are available then it is possible that two or more interrupts are issued simultaneously. To control the order in which the interrupts are serviced the microcontroller should use a *prioritization* mechanism. Most microcontrollers, including the S12 can prioritize interrupts in hardware but where this is not possible software implementations can be made to handle this task.

4.2 INTERRUPTS ON THE S12 PLATFORM

4.2.1 The S12 Interrupt block

The block diagram of the S12 interrupt block is shown in Figure 4.1. The S12 platform provides all basic interrupt features like interrupt masking and prioritisation.

There are two categories of interrupts in the S12 interrupt system:

- **maskable interrupts** which include the IRQ pin interrupt and all peripheral function interrupts. Different members of the S12 family implement various peripheral functions, therefore, the number of maskable interrupts available on each member is different.
- **nonmaskable interrupts** which include the XIRQ pin interrupt, the *swi* instruction interrupt, and the unimplemented opcode trap.

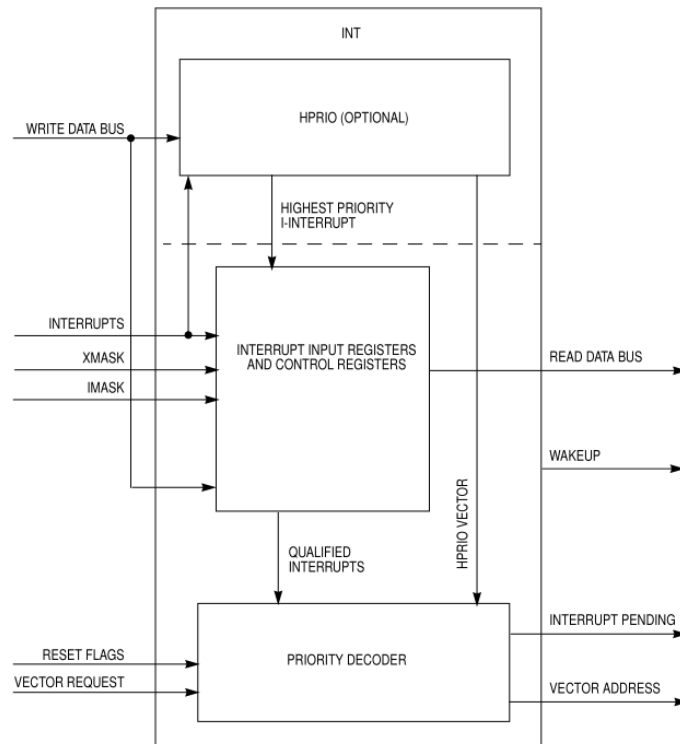


Figure 4.1 S12 Real-Time Interrupt System Block Diagram, according to [11]

The I flag of the CCR register is the global mask of all maskable interrupts. Setting this flag to ‘1’ disables all maskable interrupts. Additionally, maskable interrupts have local enable bits that can be used for individual masking. These can be found within the register sets controlling each particular block. For example, if we need to use the interrupts associated to the Port P pins the Port P Interrupt Enable Register (PIEP) bits have to be set to the appropriate value as shown in section 2.8.

The global mask flags in the CCR have no effect when one of the nonmaskable interrupts are issued. A nonmaskable interrupt can be generated by using the swi assembly instruction. This is commonly used for debugging purposes, eg. for implementing breakpoints. Each assembly instruction is encoded through an *opcode*. When an undefined opcode is assigned for execution to the S12 CPU a nonmaskable interrupt called unimplemented opcode trap is generated.

Each interrupt source on the S12 platform has an assigned priority and position in the interrupt vector table. The priorities and vector addresses of all S12 interrupt sources are listed below in Table 4.1 along with the corresponding register that holds the local enable bit. Detailed descriptions of the registers used to enable interrupts for S12 modules presented in this table are included in corresponding chapters as follows: PWM-related interrupts in chapter 5, ATD-related interrupts in chapter 6 and Enhance Capture Timer related interrupts in chapter 7.

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFFE	Reset	None	None	-
\$FFFC	Clock Monitor fail reset	None	COPCTL (CME, FCME)	-
\$FFFA	COP failure reset	None	COP rate select	-
\$FFF8	Unimplemented instruction trap	None	None	-
\$FFF6	SWI	None	None	-
\$FFF4	XIRQ / BF High Priority Sync Pulse	X-Bit	None / BFRIER (XSYNIE)	-
\$FFF2	IRQ	I-Bit	INTCR (IRQEN)	\$F2
\$FFF0	Real Time Interrupt	I-Bit	CRGINT (RTIE)	\$F0
\$FFEE	Enhanced Capture Timer channel 0	I-Bit	TIE (COI)	\$EE
\$FFEC	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	\$EC

\$FFE8	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	\$E8
\$FFE6	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	\$E6
\$FFE4	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	\$E4
\$FFE2	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	\$E2
\$FFE0	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	\$E0
\$FFDE	Enhanced Capture Timer overflow	I-Bit	TSCR2 (TOF)	\$DE
\$FFDC	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	\$DC
\$FFDA	Pulse accumulator input edge	I-Bit	PACTL (PAI)	\$DA
\$FFD8	SPIO	I-Bit	SPICR1 (SPIE, SPTIE)	\$D8
\$FFD6	SCIO	I-Bit	SCICR2 (TIE, TCIE, RIE, ILIE)	\$D6
\$FFD4	SCI1	I-Bit	SCICR2 (TIE, TCIE, RIE, ILIE)	\$D4
\$FFD2	ATD0	I-Bit	ATDCTL2 (ASCIE)	\$D2
\$FFD0	ATD1	I-Bit	ATDCTL2 (ASCIE)	\$D0
\$FFCE	Port J	I-Bit	PIEJ (PIEJ7, PIEJ6, PIEJ1, PIEJ0)	\$CE
\$FFCC	Port H	I-Bit	PIEH (PIEH7-0)	\$CC
\$FFCA	Modulus Down Counter underflow	I-Bit	MCCTL (MCZI)	\$CA
\$FFC8	Pulse Accumulator B Overflow	I-Bit	PBCTL (PBOVI)	\$C8
\$FFC6	CRG PLL lock	I-Bit	PLLCR (LOCKIE)	\$C6
\$FFC4	CRG Self Clock Mode	I-Bit	PLLCR (SCMIE)	\$C4
\$FFC2	BDLC	I-Bit	DLCBCR1 (IE)	\$C2
\$FFC0	IIC Bus	I-Bit	IBCR (IBIE)	\$C0
\$FFBE	SPI1	I-Bit	SPICR1 (SPIE, SPTIE)	\$BE
\$FFBA	EEPROM	I-Bit	ECNFG (CCIE, CBEIE)	\$BA
\$FFB8	FLASH	I-Bit	FCNFG (CCIE, CBEIE)	\$B8
\$FFB6	CAN0 wake-up	I-Bit	CANRIER (WUPIE)	\$B6
\$FFB4	CAN0 errors	I-Bit	CANRIER (CSCIE, OVRIE)	\$B4
\$FFB2	CAN0 receive	I-Bit	CANRIER (RXFIE)	\$B2
\$FFB0	CAN0 transmit	I-Bit	CANTIER (TXEIE[2:0])	\$B0
\$FFAE	CAN1 wake-up	I-Bit	CANRIER (WUPIE)	\$AE
\$FFAC	CAN1 errors	I-Bit	CANRIER (CSCIE, OVRIE)	\$AC
\$FFAA	CAN1 receive	I-Bit	CANRIER (RXFIE)	\$AA
\$FFA8	CAN1 transmit	I-Bit	CANTIER (TXEIE[2:0])	\$A8
\$FFA6	BF Receive FIFO not empty	I-Bit	BFRIER (RCVFIE)	\$A6
\$FFA4	BF receive	I-Bit	BFBUFCTL[15:0] (IENA)	\$A4
\$FFA2	BF Synchronization	I-Bit	BFRIER (SYNAIE, SYNIE)	\$A2
\$FFA0	BF general	I-Bit	BFBUFCTL[15:0] (IENA), BFGIER (OVRNIE, ERRIE, SYNEIE, SYNLIE, ILLPIE, LOCKIE, WAKEIE) BFRIER (SLMMIE)	\$A0
\$FF96	CAN4 wake-up	I-Bit	CANRIER (WUPIE)	\$96
\$FF94	CAN4 errors	I-Bit	CANRIER (CSCIE, OVRIE)	\$94
\$FF92	CAN4 receive	I-Bit	CANRIER (RXFIE)	\$92
\$FF90	CAN4 transmit	I-Bit	CANTIER (TXEIE[2:0])	\$90
\$FF8E	Port P Interrupt	I-Bit	PIEP (PIEP7-0)	\$8E
\$FF8C	PWM Emergency Shutdown	I-Bit	PWMSDN (PWMIE)	\$8C

Table 4.1 Interrupt vector map

EXAMPLE 4.1 In an application where Port P interrupt and ATD0 interrupt were both enabled which will be first served if they both are generated at the same time?

Solution: To answer this question we have to take a look at the interrupt priorities in Table 4.1. The priority for Port P interrupts is 0x8E while the priority for the ATD0 interrupt is 0xD2. Hence, the ATD0 interrupt service routine will be executed first.

One of the interrupts in the group of maskable interrupts can be promoted to the highest priority using the HPRIO (Highest Priority I Interrupt) register. The priorities of the other sources remain unchanged. To promote an interrupt, write the least significant byte of the associated interrupt vector address to the HPRIO register. For example, to raise the Port P interrupt to the highest priority, write the value of \$8E to the HPRIO register.

HPRIO

7	6	5	4	3	2	1	0
PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	-

4.2.2 Using S12 interrupts and CodeWarrior

To define an interrupt service routine in CodeWarrior the *interrupt* keyword must be used before the function name. The ISR must be located in the non-banked segment of the S12 memory so the `CODE_SEG __NEAR_SEG NON_BANKED` pragma directive must be used for signalling this. If the code following the definition of the ISR should be located in the default segment of the memory then the `CODE_SEG DEFAULT` pragma should be used. The code snippet below illustrates the template for defining an ISR in CodeWarrior. Here, *my_ISR* is the name of the routine.

```
#pragma CODE_SEG __NEAR_SEG NON_BANKED
interrupt void my_ISR(void)
{
    /*Add interrupt handling here*/
}

#pragma CODE_SEG DEFAULT
```

Code snippet 4.1 Declaration of interrupt service routine

When the interrupt is issued a corresponding flag is set to signal the event. This flag has to be reset after executing the ISR. Otherwise the interrupt will stay active and the ISR will be called repeatedly until the flag is reset.

After writing the ISR, the interrupt vector has to be set up. We do this by linking the created interrupt service routine to the corresponding position of the interrupt vector table. The linker file will have the **.prm* extension and the same name as the debug configuration. When your debug configuration is SofTec the linker file is called *SofTec_linker.prm* and can be found under the *Prm* folder of the project structure as illustrated by Figure 4.2.

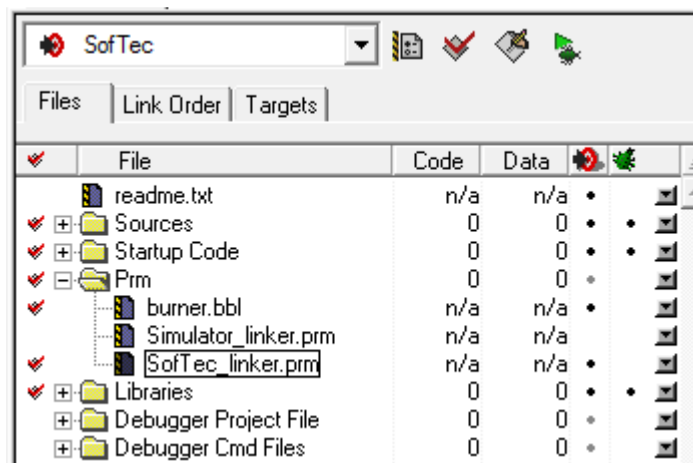


Figure 4.2 Location of the project linker file

```

NAMES END
SEGMENTS
RAM = READ_WRITE 0x1000 TO 0x3FFF;
/* unbanked FLASH ROM */
ROM_4000 = READ_ONLY 0x4000 TO 0x7FFF;
ROM_C000 = READ_ONLY 0xC000 TO 0xFEFF;
/* 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 ROM_4000 */
/* PAGE_3F = READ_ONLY 0x3F8000 TO 0x3FBFFF; not used: equivalent to ROM_C000 */
END

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

STACKSIZE 0x100

VECTOR 0 _Startup /* reset vector: this is the default entry point for a C/C++ application. */
//VECTOR 0 Entry /* reset vector: this is the default entry point for a Assembly application. */
//INIT Entry /* for assembly applications: that this is as well the initialisation entry point */
VECTOR ADDRESS 0xFF8E my_ISR

```

Code snippet 4.2 Linker file with added reference to the ISR

Linker files are used to add custom directives for the linker. Common uses will be for defining address and size of the available memory areas and organize memory content in segments. The stack and the interrupt vector content can be also configured by using the linker file.

An example for setting up the memory of a CodeWarrior project is given in Code snippet 4.2. The file starts with the definition of the RAM segment with its access rights (read and write) and address range. The FLASH memory is defined next as a series of 16Kbyte sectors. The arrangement of the various sections into memory is provided by within the PLACING section. The stack size is also set here. The last line of this snippet (marked in bold letters) is the one of interest for using interrupts as it assigns the `my_ISR` routine to the interrupt vector position corresponding to Port P interrupts. When adding

EXAMPLE 4.2 Modify the linker file line specifying the Port P interrupt ISR to set up a function called *your_ISR* as the routine to be executed when the ATD0 interrupt is generated.

Solution: The vector address associated with the ATD0 interrupt is 0xFFD2. This is the address that we have to provide after the VECTOR ADDRESS keywords followed by the name of our function. The actual line that will have to be written in the linker file is given below:

```
VECTOR ADDRESS 0xFFD2 your_ISR
```

As an alternative to using the linker file to set up the interrupt vector to point to a certain ISR this can be done directly in the C source code file. The content of either the entire vector or just a part of it can be defined as a constant variable with a compiler directive telling the linker the address at which to locate its content. The code snippet shows how to set the reference to the `my_ISR` function to the Port P interrupt vector location.

```
typedef void (*near tISR)(void);
const tISR _vect [ ] @0xFF8E { // Select the address for Port P interrupt
my_ISR
};
```

Code snippet 4.3 Setting interrupt vector routine reference in codeWarrior C source file

EXERCISE 4.1 Write an application that uses the interrupt system to capture press events of the 4 pushbuttons available on the S12 development board PTP0-PTP3. Each press will toggle the state of the corresponding LED (Port B 0-3).

EXERCISE 4.2 Modify the demo application that demonstrates the use of the analog-to-digital converter to provide the same functionality using interrupts. Interrupts will be used instead of polling to call an ISR that reads the conversion result and outputs it on the port B LEDs at the end of the conversion. You will need to use the ASCIE and ASCIF bits of the ATDCTL2 register.

EXERCISE 4.3 For this exercise you will have to use the function generator to generate square signals of various frequencies on Port J bit 7 pin. Use interrupts and configure the port to issue interrupts on the falling edge of the signal. Write an ISR that toggles another pin (e.g. one of the pins of port A) whenever it is called. Use an oscilloscope to check if the signal generated on this pin matches the one outputted by the signal generator.

4.3 THE S12 CLOCK AND RESET GENERATOR

4.3.1 General aspects on clock and reset generation and the S12 features

All microcontrollers need a clock signal for the core and modules operation. In most cases a clock signal is provided by an external crystal oscillator which has to be corrected by on-chip circuitry to generate a square waveform. In other cases the external circuit directly generates the square waveform alleviating the need for additional processing of the signal on the microcontroller side. Some microcontrollers can even be used without any external clock circuit as they provide integrated oscillators.

The base oscillator frequency needs to be derived to obtain the clock frequency needed by the application. This is done by the phase-locked-loop (PLL) circuit which has a low-frequency signal as an input and outputs a higher-frequency clock signal. The PLL circuit is built as a feedback loop system using the feedback to correct and stabilise the output signal.

Reset circuitry react on a certain set of stimuli and initiate actions corresponding to the reset sequence. Common reset sources are external reset pins, power-on, access exceptions or modules assuring the reliable execution of device firmware commonly known as watchdogs. When reset conditions are met the associated circuit executes initialization steps like setting registers to their initial values.

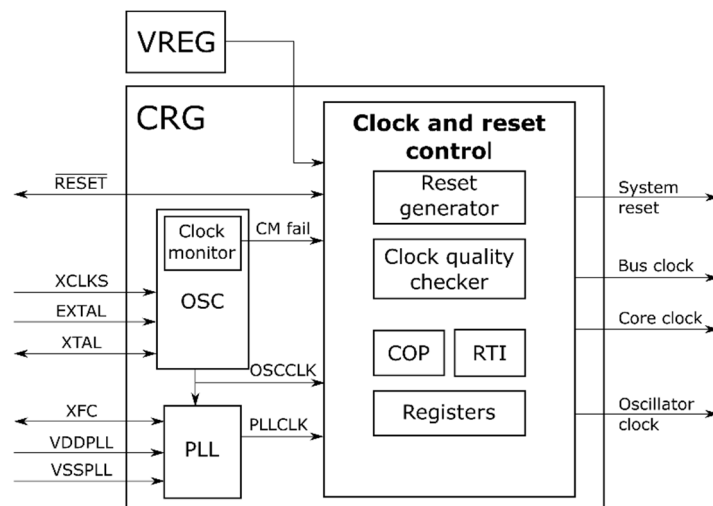


Figure 4.3 Block diagram of the Clock and Reset Generator block according to [12]

For handling the tasks previously described, the S12 microcontroller integrates a Clock and Reset Generator (CRG) module. The organization of the CRG block [12] is illustrated in Figure 4.3. The CRG module interacts with external (i.e. outside the chip) circuitry through a set of signals:

- **VDDPLL** and **VSSPLL** – these pins provide the supply voltage (VDDPLL) and ground (VSSPLL) for the PLL circuit allowing it to be powered independently of the rest of the circuitry. These connections have to be properly made even if PLL is not used;
- **XFC** – this pin must be connected to an external loop filter to eliminate the VCO (Voltage Controlled Output) input ripple. The needed filter is a second-order, low-pass filter that is used along with the reference frequency to determine the speed of the corrections and the stability of PLL. When the PLL is not used, the XCF pin must be tied to VDDPLL;
- **EXTAL** and **XTAL** – these pins are provided for the connection of a crystal or a CMOS compatible clock circuit to control the internal clock generator. EXTAL is the input for the external clock or the input for the crystal oscillator, while XTAL is the output of the crystal oscillator amplifier;

- **RESET** – this is a bidirectional, active low reset pin. As an input it leads to the initialization of the microcontroller to a known initial state. As an output it indicates that a system reset was triggered;
- **XCLKS** – this pin indicates if an external crystal in combination with the internal oscillator should be used or whether the oscillator clock will be provided by an external clock source.

4.3.2 Clock generation

The internal oscillator circuit (OSC) generates the internal reference clock OSCCLK based on the input source with the XCLKS pin. The reference clock is used by the PLL block to derive the PLLCLK if PLL is enabled or as an input to directly drive the system clock.

Phase-Locked-Loop (PLL)

The frequency value of the PLLCLK is determined by two components: the synthesizer (SYNR) and the reference divider (REFDV) using the following equation: $PLLCLK = 2 \times OSCCLK \times (SYNR + 1) / (REFDV + 1)$. The values of the SYNR and REFDV members are given by two corresponding registers which are presented below.

SYNR

7	6	5	4	3	2	1	0
-	-	SYN5	SYN4	SYN3	SYN2	SYN1	SYN0

REFDV

7	6	5	4	3	2	1	0
-	-	-	-	REFDV3	REFDV2	REFDV1	REFDV0

The PLL block can operate either in acquisition mode or in the tracking mode. After initially powering up the PLL its output frequency will not be within the expected target bounds. This is when the acquisition mode comes in handy as it allows the PLL to make large adjustments a fast correction of the output signal. After this signal reaches its target output frequency, the PLL switches to the tracking mode, in which it makes only small adjustments to keep the output from deviating from the target frequency.

The PLL operation can be controlled via four registers: CRGFLG, CRGINT, CLKSEL and PLLCTL. Some of these registers also contain fields for controlling other blocks of the CRC module.

The CRGFLG (CRG Flag) register provides status information on the operation of the CRG module.

CRGFLG

7	6	5	4	3	2	1	0
RTIF	PORF	-	LOCKIF	LOCK	TRACK	SCMIF	SCM

- **RTIF (Real Time Interrupt Flag)** – this bit is set to 1 after a Real Time Interrupt period timeout. This flag can be cleared by writing it to 1;
- **PORF (Power on Reset Flag)** – on the occurrence of a power on reset this flag is set to 1. To clear it write a one to the PORF bit;
- **LOCKIF (PLL Lock Interrupt Flag)** – this flag is set to 1 when the LOCK status bit changes. It can be cleared by writing a 1;
- **LOCK (Lock Status Bit)** – this bit reflects the status of the PLL lock. It reads to 0 when the PLL Voltage Controlled Output (VCO) does not fit the desired tolerance or 1 when the tolerance is met;

- **TRACK (Track Status Bit)** – this bit reflects the status of the PLL operation mode. When in acquisition mode this bit is 0. In track mode this bit reads to 1;
- **SCMIF (Self Clock Mode Interrupt Flag)** – this flag is set to 1 when the self-clock mode (SCM) status bit changes its value. To clear this flag write it to 1;
- **SCM (Self Clock Mode Status Bit)** – when the operating state is Self Clock Mode this bit is set to 1. This mode, if enabled, is entered when a loss of the external clock signal is detected.

The CRGINT (CRG Interrupt Enable) registers is used to configure the interrupts available for the CRG module.

CRGINT

7	6	5	4	3	2	1	0
RTIE	-	-	LOCKIE	-	-	SCMIE	-

- **RTIE (Real Time Interrupt enable)** – set this bit to 1 for enabling the Real Time Interrupt source;
- **LOCKIE (PLL Lock Interrupt Enable)** – set this bit to 1 to enable interrupt generation on the PLL LOCK signal changes;
- **SCMIE (SCM Interrupt Enable)** – when this bit is set to 1 interrupts will be generated on changes of the SCM status bit.

The CLKSEL (Clock Select) register controls the clock selection within the CRG module.

CLKSEL

7	6	5	4	3	2	1	0
PLLSEL	PSTP	SYSWAI	ROAWAI	PLLWAI	CWAI	RTIWAI	COPWAI

- **PLLSEL (PLL Select)** – set this bit to 1 to enable the PLL and derive the system clocks from PLLCLK or to 0 otherwise. This bit cannot be written when LOCK = 0 and AUTO=1 or when TRACK=0 and AUTO=0;
- **PSTP (Pseudo Stop)** – this bit controls the functionality of the oscillator in Stop Mode. Writing this bit to 0 will disable the oscillator in Stop Mode while setting it to 1 will let it run;
- **SYSWAI (System clock stop in Wait Mode)** – if this bit is set to 0 the clock will continue to run while in Wait Mode. When set to 1 the clock will be stopped in Wait Mode;
- **ROAWAI (Reduced Oscillator Amplitude in Wait Mode)** – if this bit is set to 1 the oscillator amplitude is reduced while in Wait Mode, otherwise it remains unchanged;
- **PLLWAI (PLL stops in Wait Mode)** – when this bit is set to 1 and the Wait Mode is active the PLL stops. Setting it to 0 lets the PLL continue operation;
- **CWAI (Core stops in Wait Mode)** – if this bit is set to 1 when in Wait Mode, the core clock stops running;
- **RTIWAI (RTI stops in Wait Mode)** – setting this bit to 1 stops the RTI block in Wait Mode;
- **COPWAI (COP stops in Wait Mode)** – if this bit is set to 1 and the Wait Mode is active the COP stops. 0 will keep the COP running in Wait Mode.

The PLLCTL (PLL Control Register) is dedicated to controlling the PLL functionality:

- **CME (Clock Monitor Enable)** – to enable the clock monitor write this bit to 1. This bit cannot be written when SCM is 1;

- **PLLON (Phase Lock Loop On)** – when set to 1 this bit enables the PLL circuitry while 0 will disable it. Write anytime except when PLLSEL is 1;
- **AUTO (Automatic Bandwidth Control)** – this bit selects if the PLL acquisition and tracking modes should be selected automatically or by software by using the ACQ bit. Writing the AUTO bit to 1 enables the automatic mode;
- **ACQ (Acquisition)** – when set to 1 the high bandwidth (tracking) mode is on. If set to 0 the low bandwidth (acquisition) mode is activated;
- **PRE (RTI Enable during Pseudo Stop)** – when this bit is set to 1 the RTI continues to run during Pseudo Stop Mode;
- **PCE (COP Enable during Pseudo Stop)** – when this bit is set to 1 the COP continues to run during Pseudo Stop Mode;
- **SCME (Self Clock Mode Enable)** – set this bit to 1 to enable the usage of Self Clock Mode. When set to 0 clock failures will result in a clock monitor reset.

PLLCTL

7	6	5	4	3	2	1	0
CME	PLLON	AUTO	ACQ	-	PRE	PCE	SCME

As previously stated, the system clock (SYSCLK) can be chosen as either the PLLCLK or the OSCCLK. The SYSCLK is directly connected to the core clock output of the CRG. The bus clock is used to drive external memory and peripheral modules. Its frequency is obtained by dividing the SYSCLK by 2.

EXAMPLE 4.3 Configure the CRG so that a bus clock of 20MHz is used. Consider that the implementation will be used on the ZK-S12-B board with the 16MHz crystal selected as the input clock. **Solution:** The input clock is lower than the required bus clock. Therefore, the PLL has to be used to obtain a higher frequency signal on the PLLCLK line. Since the frequency of the bus clock is half the frequency of SYSCLK (which equals PLLCLK in this case), the PLLCLK has to be configured to a frequency of 40MHz. The SYN_R and REF_{DV} parameters are determined by using the equation for computing the PLLCTL: $40\text{MHz} = 2 \times 16\text{MHz} \times (\text{SYNR}+1)/(\text{REFDV}+1)$. In our case SYN_R = 4 and REF_{DV} = 3 are the values that fit the equation. The following code snippet gives the required configuration:

```

SYNR = 0x04;
REFDV = 0x03;
PLLCTL = 0x60;           //Enable PLL and Automatic Bandwidth Control
while((CRGFLG & 0x08)==0x00); //Wait for the PLLCLK lock
CLKSEL = 0x80;           //Select the PLLCLK as the clock signal for the CRG

```

4.3.3 Reset generation

The CRG provides special handling for several types of reset: power-on reset, external reset, COP watchdog reset and clock monitor reset.

Power-On reset

A special circuit on the S12 chip monitors the level of the VDD power supply and asserts a reset when it reaches a certain level. This circuit is triggered by power line slew rate. When the power-on is triggered the CRG module performs a quality check of the input clock signal. The reset sequence will start after the clock quality check indicates a valid clock signal or the CRG enters the Self Clock Mode as a result of a failed clock check.

External reset

When any of the four reset events is detected the $\overline{\text{RESET}}$ pin is driven to low for a period of 128 SYSCLK cycles. This period may be increased by 3 to 6 additional SYSCLK cycles depending on the internal synchronization latency. After this period has elapsed, the $\overline{\text{RESET}}$ pin is driven to high after which the CRG block waits for another 64 SYSCLK cycles before reading the value of the pin. If the value read back is still low this means the reset was caused by an external source.

Computer Operating Properly (COP)

The COP is the watchdog timer feature provided by the S12 to assure proper execution of the firmware. The watchdog functionality, commonly available and employed on most embedded platforms, is implemented in the form of a timer with configurable period that has to be restarted by the user program before the configured time period elapses. If the software fails to restart the watchdog timer it is an indication that the intended execution sequence is no longer followed and a reset is issued.

The COPCTL (COP Control) register is used to enable and configure the S12 COP block as described below.

COPCTL

7	6	5	4	3	2	1	0
WCOP	RSBCK	-	-	-	CRG2	CRG1	CRG0

- **WCOP (Window COP Mode)** – when set to 1 writing to the ARMCOP register must occur in the last 25% of the selected period. Writing any value in the first 75% of the period resets the COP;
- **RSBCK (COP and RTI stop in Active BDM mode)** – writing this bit to 1 stops the COP and RTI counters while in Active BDM mode while setting it to 0 allows them to run;
- **CRGx (COP Watchdog Timer Rate select)** – these three bits allow the selection of the COP time-out period according to Table 4.2.

CR2	0	0	0	0	1	1	1	1
CR1	0	0	1	1	0	0	1	1
CR0	0	1	0	1	0	1	0	1
OSCCLK cycles to time-out	COP disabled	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{23}	2^{24}

Table 4.2 Setting COP time-out period

The COP reset and time-out period restart is done by using the 8-bit ARMCOP (Timer Arm/Reset) register. Writing any value to this register besides 0x55 or 0xAA will cause a COP reset. To restart the COP time-out period the value 0x55 has to be written to the ARMCOP register followed by a 0xAA. When read this register always returns 0x00.

Clock monitor

If enabled, the clock monitor circuit constantly checks the OSCCLK signal and signals its failure. Depending on the value set to the SCME bit of the PLLCTL, the OSCCLK failure can lead to an immediate switch to the Self Clock Mode or to a clock monitor reset. The reset event forces the registers to their default settings which means the SCME bit will be set to 1 after the reset resulting in the operation in the Self Clock Mode. The input clock signal is checked in parallel and, when the clock checker indicates that it meets the required state, the CRG leaves Self Clock Mode selecting OSCCLK as a source clock.

4.3.4 Using real-time interrupts

A special type of periodical interrupt is provided by the clock and reset generation block. The CRG includes a Real-Time Interrupt (RTI) module. This block can be used to generate periodical hardware interrupts. To enable the RTI, the RTIE bit of the CRGINT (previously presented in this chapter) register has to be set.

The interrupt occurrence rate is set via the RTICTL register, where the RTR[6:4] bits are used to select the prescale rate for RTI and the RTR[3:0] bits select the modulus counter target in order to provide additional granularity. Table 4.3 illustrates all possible divide rates. To calculate the frequency at which the interrupts occur use the frequency of the OSCCLK signal as this is the clock source that drives the RTI block.

RTICTL

7	6	5	4	3	2	1	0
-	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0

RTR[3:0]	RTR[6:4]							
	000 (OFF)	001 (2^{10})	010 (2^{11})	011 (2^{12})	100 (2^{13})	101 (2^{14})	110 (2^{15})	111 (2^{16})
0000 ($\div 1$)	OFF	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001 ($\div 2$)	OFF	2×2^{10}	2×2^{11}	2×2^{12}	2×2^{13}	2×2^{14}	2×2^{15}	2×2^{16}
0010 ($\div 3$)	OFF	3×2^{10}	3×2^{11}	3×2^{12}	3×2^{13}	3×2^{14}	3×2^{15}	3×2^{16}
0011 ($\div 4$)	OFF	4×2^{10}	4×2^{11}	4×2^{12}	4×2^{13}	4×2^{14}	4×2^{15}	4×2^{16}
0100 ($\div 5$)	OFF	5×2^{10}	5×2^{11}	5×2^{12}	5×2^{13}	5×2^{14}	5×2^{15}	5×2^{16}
0101 ($\div 6$)	OFF	6×2^{10}	6×2^{11}	6×2^{12}	6×2^{13}	6×2^{14}	6×2^{15}	6×2^{16}
0110 ($\div 7$)	OFF	7×2^{10}	7×2^{11}	7×2^{12}	7×2^{13}	7×2^{14}	7×2^{15}	7×2^{16}
0111 ($\div 8$)	OFF	8×2^{10}	8×2^{11}	8×2^{12}	8×2^{13}	8×2^{14}	8×2^{15}	8×2^{16}
1000 ($\div 9$)	OFF	9×2^{10}	9×2^{11}	9×2^{12}	9×2^{13}	9×2^{14}	9×2^{15}	9×2^{16}
1001 ($\div 10$)	OFF	10×2^{10}	10×2^{11}	10×2^{12}	10×2^{13}	10×2^{14}	10×2^{15}	10×2^{16}
1010 ($\div 11$)	OFF	11×2^{10}	11×2^{11}	11×2^{12}	11×2^{13}	11×2^{14}	11×2^{15}	11×2^{16}
1011 ($\div 12$)	OFF	12×2^{10}	12×2^{11}	12×2^{12}	12×2^{13}	12×2^{14}	12×2^{15}	12×2^{16}
1100 ($\div 13$)	OFF	13×2^{10}	13×2^{11}	13×2^{12}	13×2^{13}	13×2^{14}	13×2^{15}	13×2^{16}
1101 ($\div 14$)	OFF	14×2^{10}	14×2^{11}	14×2^{12}	14×2^{13}	14×2^{14}	14×2^{15}	14×2^{16}
1110 ($\div 15$)	OFF	15×2^{10}	15×2^{11}	15×2^{12}	15×2^{13}	15×2^{14}	15×2^{15}	15×2^{16}
1111 ($\div 16$)	OFF	16×2^{10}	16×2^{11}	16×2^{12}	16×2^{13}	16×2^{14}	16×2^{15}	16×2^{16}

Table 4.3 RTI Frequency divide rates

5 THE TIMER MODULE

A dedicated timer system is needed in many embedded applications to perform various tasks such as: creating time delays, measuring time, measuring signal period, frequency or pulse width, counting events or generating periodic interrupts.

The S12 Timer module is built around a 16-bit programmable counter. It can provide three main functions: input capture, output compare and pulse accumulator. The *input capture* function offers the possibility of measuring characteristics of an external signal and is triggered by an event represented by a signal edge which can be a rising or falling edge. It can be used to measure the duty cycle and frequency of a periodic signal or the length of an input pulse. The *output compare* makes a comparison of the internal timer value with the output compare register. If they are equal several actions can be performed: toggle a pin, generate an interrupt request and setting a flag. The last timer functionality, of *pulse accumulator* is used to count external events marked as signal pulses on a corresponding pin.

Some of the S12 family members (e.g. the S12C) provide a **Standard Timer Module (TIM)**. TIM consists of 8 input-capture/output-compare channels and one 16-bit pulse accumulator. On other S12 family members the timer module features additional functionality and is referred to as **Enhanced Capture Timer Module (ECT)**. The block diagram illustrated in Figure 5.1 depicts the features of TIM and ECT. Components that are marked with grey colour are additional features found only on ECT modules.

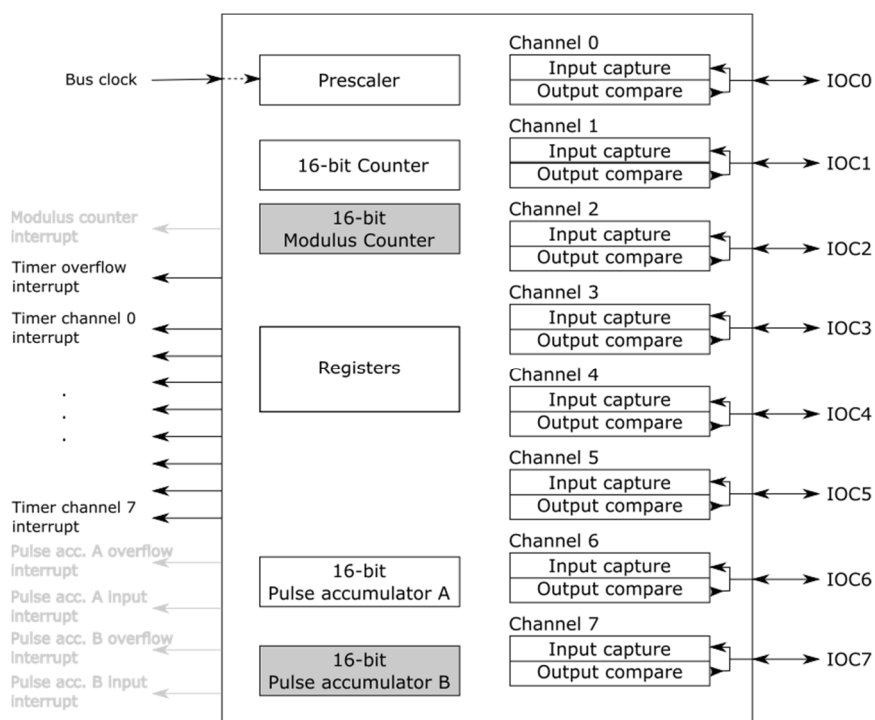


Figure 5.1 Diagram of the timer module TIM/ECT, according to [13] and [14]

There are three main components at the base of the standard timer counter module:

- the 16-bit free-running counter and the associated prescaling hardware
- the 8 capture/output compare channels
- the pulse accumulator

The 16-bit free-running counter sits at the heart of the timer module. The timing information used by all the input capture and output compare functions is derived from this counter. After the timer

system is enabled the counter starts at 0x0000 and is incremented at each pulse. Once the counter reaches 0xFFFF it rolls over back to 0x0000 and continues counting. The frequency at which this counter is incremented can be set to the needed value by deriving the system clock using the prescaler block.

The IOC[7:0] pins associated to the 8 dual-function 16-bit input-capture/output-compare channels are connected to the pins of Port T. Each channel can be configured to work either as an input capture or an output compare channel but not both at the same time. When configured as an input capture channel the corresponding register captures the 16-bit value of the free-running counter when an external event occurs. If configured in the output compare event, a 16-bit value has to be loaded in the corresponding 16-bit timer channel register. When the counter reaches this value a chain of events is triggered. First, the associated channel flag is set followed by the generation of the corresponding interrupt request (if the interrupt was enabled for the channel). Lastly, the user-specified event is generated on the associated pin.

The pulse accumulator can be configured in event-counting mode to count the number of events that appear on the associated pin (pin 7 of Port T). In the gated-time-accumulation mode, the pulse accumulator measures the duration of a pulse.

5.1 TIM ASSOCIATED REGISTERS

Table 5.1 provides an overview of the registers associated to the standard S12 timer module. Entries that are greyed out and marked with a strikethrough line are registers that are only used for factory testing purposes. The main registers used for configuring the timer module will be detailed next.

TIOS - Timer Input Capture/Output Compare Select	RW	TCTL4 - Timer Control 4	RW
CFORC - Timer Compare Force	RW	TIE - Timer Interrupt Enable	RW
OC7M - Output Compare 7 Mask	RW	TSCR2 - Timer System Control 2	RW
OC7D - Output Compare 7 Data	RW	TFLG1 - Main Timer Interrupt Flag1	RW
TCNT - Timer Count	RW	TFLG2 - Main Timer Interrupt Flag2	RW
TSCR1 - Timer System Control1	RW	TCx - Timer Input Capture/Output Compare x	RW
TTOV - Timer Toggle Overflow	RW	PACTL - 16-Bit Pulse Accumulator Control	RW
TCTL1 - Timer Control 1	RW	PAFLG - Pulse Accumulator Flag	RW
TCTL2 - Timer Control 2	RW	PACNT - Pulse Accumulator Count	RW
TCTL3 - Timer Control 3	RW	TIMTST - Timer Test	RW

Table 5.1 Registers associated to the standard timer module

5.1.1 Timer counter registers

There are four registers related to the main timer counter: TCNT, TSCR1, TSCR2 and TFLG2. The TCNT register is a 16-bit up counting register which can be read to get the current counter value. Both bytes of this register have to be read in a single cycle to get a correct value. Writing to this register has no effect.

TCNT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The TSCR1 register is used to enable the timer module and set the fast flag clearing mode. Its bits are described below:

- **TEN (Timer Enable)** - Setting this bit to 1 enables the timer module while a 0 value disables it;
- **TSWAI (Timer Module Stops While in Wait)** – 1 disables the timer counter while the microcontroller is in wait mode. 0 enables running also during wait state.

- **TSFRZ (Timer Stops While in Freeze Mode)** – Setting this bit to 1 disables the timer counter when the microcontroller is in the freeze mode. A 0 enables running of the timer module during freeze mode;
- **TFFCA (Timer Fast Flag Clear All)** – When this bit is set to 1 then a read from an input capture or a write to the output compare channel causes the corresponding channel flag – CnF – from TFLG1 to be cleared. In the case of TFLG2, any access to the TCNT register clears the TOF flag. Any access to the PACNT registers clears the PAOVF and PAIF flags in the PAFLG register. This has the advantage of eliminating software overhead in a separate clear sequence. 0 will lead to the normal functioning of the timer flag clearing.

TSCR1

7	6	5	4	3	2	1	0
TEN	TSWAI	TSFRZ	TFFCA	-	-	-	-

TSCR2 holds additional timer settings for using the overflow interrupt, enabling timer self-reset and configuring the prescaler. The structure of the TSCR2 register is presented below.

TSCR2

7	6	5	4	3	2	1	0
TOI	-	-	-	TCRE	PR2	PR1	PRO

- **TOI (Timer Overflow Interrupt Enable)** – Setting this bit to 1 generates an interrupt request when the TOF flag is set.
- **TCRE (Timer Counter Reset Enable)** – Setting this bit to 1 will lead to a reset of the timer counter after a successful output compare 7 event.
- **PRx (Prescaler)** – these bits are used for setting the prescaler value to derive the timer clock frequency from the bus clock. Set the prescaler values according to the table below.

PR2	PR1	PRO	Timer Clock
0	0	0	Bus Clock / 1
0	0	1	Bus Clock / 2
0	1	0	Bus Clock / 4
0	1	1	Bus Clock / 8
1	0	0	Bus Clock / 16
1	0	1	Bus Clock / 32
1	1	0	Bus Clock / 64
1	1	1	Bus Clock / 128

Table 5.2 Prescaler setting

The TFLG2 register indicates when interrupt conditions are met. To clear the interrupt flag, a 1 has to be written in the corresponding flag position.

- **TOF (Timer Overflow Flag)** – This bit indicates when the 16-bit counter overflows.

TFLG2

7	6	5	4	3	2	1	0
TOF	-	-	-	-	-	-	-

5.1.2 Registers common for input capture and output compare

The TIOS register is used to select whether a channel should function in the input capture or output compare mode. Setting the **IOSx** bit to 0 sets the corresponding channel x to work as an input capture, otherwise it will function as an output compare.

TIOS

7	6	5	4	3	2	1	0
IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0

To enable interrupts for input capture or output compare events the TIE register should be configured. Setting the **CxI** bit of this register to 1 enables the interrupt for the corresponding channel x.

TIE

7	6	5	4	3	2	1	0
C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I

When the interrupt condition has occurred on a channel, the corresponding bit of the TFLG1 register is set. To clear the interrupt flag its value should be set to 1.

TFLG1

7	6	5	4	3	2	1	0
C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F

The eight 16-bit registers TC0 through TC7 are each associated to one timer channel. When the corresponding channel is configured as input capture the TC register will hold the timer counter value latched when the defined event appears. If the channel is configured as output compare the condition value for output compare should be written in the corresponding TCx register.

TCx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

5.1.3 Registers related to the input capture function

TCTL3 and TCTL4 are used to configure the behaviour of the input capture edge detector circuits by setting appropriate values for each bit pair (**EDGxB**, **EDGxA**) according to Table 5.3. Setting this registers specifies the signal edge on which to record the capture.

TCTL3

7	6	5	4	3	2	1	0
EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A

TCTL4

7	6	5	4	3	2	1	0
EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0SA

If an input capture channel is configured but the capture is disabled, the associated pin can be used as a general purpose I/O pin.

EDGnB	EDGnA	Configuration
0	0	Capture disabled
0	1	Capture on rising edges only
1	0	Capture on falling edges only
1	1	Capture on any edge (rising or falling)

Table 5.3 Configuration of the edge detection circuit

5.1.4 Registers related to the output compare function

TCTL1 and TCTL2 are used to specify the output action to be taken as a result of a successful output compare. When either **OMx** or **OLx** is 1, the pin associated with output compare channel x becomes an output tied to this channel. Table 5.4 shows the outcomes of all possible combinations for setting the OM-OL pairs value.

TCTL1

7	6	5	4	3	2	1	0
OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4

TCTL2

7	6	5	4	3	2	1	0
OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0

OMx	OLx	Action
0	0	Timer disconnected from output pin logic
0	1	Toggle output compare x output line
1	0	Clear output compare x output line to zero
1	1	Set output compare x output line to one

Table 5.4 Compare result output action

To force an output compare action on a channel set the corresponding bit of the CFORC register to 1.

CFORC

7	6	5	4	3	2	1	0
FOC7	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0

Output compare function on channel 7 is special because it can control up to 8 output compare functions simultaneously. Register OC7M is used to set the channels that should be controlled. The values of the corresponding pins are specified in the OC7D register. In this way, on a successful output compare for channel 7 each affected pin (specified by OC7M) assumes the value in the OC7D register.

OC7M

7	6	5	4	3	2	1	0
OC7M7	OC7M6	OC7M5	OC7M4	OC7M3	OC7M2	OC7M1	OC7M0

OC7D

7	6	5	4	3	2	1	0
OC7D7	OC7D6	OC7D5	OC7D4	OC7D3	OC7D2	OC7D1	OC7D0

The pins corresponding to output compare channels can be forced to toggle when the counter timer register overflows. This is done by setting bits of the TTOV register corresponding to the channels on which this behaviour is wanted to 1.

TTOV

7	6	5	4	3	2	1	0
TOV7	TOV6	TOV5	TOV4	TOV3	TOV2	TOV1	TOV0

5.1.5 Registers related to the pulse accumulator function

PACTL

7	6	5	4	3	2	1	0
-	PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI

The PACTL register is used to configure the pulse accumulator function by setting each individual bit to proper values:

- **PAEN (Pulse Accumulator System Enable)** – Setting this bit to 1 enables the pulse accumulator functionality;
- **PAMOD (Pulse Accumulator Mode)** – If this bit is set to 0 the pulse accumulator functions in the event counter mode while if set to 1 it works in gated time accumulation mode;
- **PEDGE (Pulse Accumulator Edge Control)** – Setting this pin according to Table 5.5 affects the behaviour of the pin associated to the pulse accumulator.

PAMOD	PEDGE	Pin Action
0	0	Falling edge
0	1	Rising edge
1	0	Div. by 64 clock enabled with pin high level
1	1	Div. by 64 clock enabled with pin low level

Table 5.5 Setting pulse accumulator pin action

- **CLKx (Clock Select Bits)** – set them to select the clock according to Table 5.6

CLK1	CLK0	Timer Clock
0	0	Use timer prescaler clock as timer counter clock
0	1	Use PACLK as input to timer counter clock
1	0	Use PACLK/256 as timer counter clock frequency
1	1	Use PACLK/65536 as timer counter clock frequency

Table 5.6 Pulse Accumulator Timer clock selection

- **PAOVI (Pulse Accumulator Overflow Interrupt enable)** – Set this bit to 1 to enable an interrupt when the pulse accumulator overflows;
- **PAI (Pulse Accumulator Input Interrupt enable)** – Set this bit to 1 to issue an interrupt on an input signal change.

When an interrupt related to the pulse accumulator functionality is produced, a flag signalling the corresponding reason is set in the PAFLG register:

- **PAOVF (Pulse Accumulator Overflow Flag)** – this bit is set when the 16-bit pulse accumulator overflows. Write a 1 in this bit to clear the flag;
- **PAIF (Pulse Accumulator Input edge Flag)** – this flag is set when the selected edge is detected on the IOC7 pin. Write this bit to 0 to clear the flag.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

Delays can be introduced for the events on the input capture pins by configuring the DLYCT (Delay Counter Control) register.

DLYCT

7	6	5	4	3	2	1	0
-	-	-	-	-	-	DLY1	DLY0

The delay is set according to Table 5.8. When the delay is activated the delay counter will count the prescribed number of clock cycles and then generate the pulse for the input-capture circuit.

DLY1	DLY0	Delay
0	0	Disabled (bypassed)
0	1	256 bus clock cycles
1	0	512 bus clock cycles
1	1	1024 bus clock cycles

Table 5.7 Delay counter selection

The ICOVW (Input Control Overwrite) register prevents the overwriting of the content of the related capture or holding register. Setting bit NOVWx to 0 allows the overwrite of the associated channel input capture register while setting it to 1 only allows overwriting after the content was either latched or read.

ICOVW

7	6	5	4	3	2	1	0
NOVW7	NOVW6	NOVW5	NOVW4	NOVW3	NOVW2	NOVW1	NOVW0

5.2.2 Registers related to the modulus down counter

The modulus down counter can be used either as a time base for generating periodic interrupts or to latch values of the IC registers and pulse accumulators to their holding registers. The MCCTL (Modulus Down-Counter Control) register is used to setup the circuit.

MCCTL

7	6	5	4	3	2	1	0
MCZI	MODMC	RDMCL	ICLAT	FLMC	MCEN	MCPR1	MCPR0

- **MCZI (Modulus Counter Underflow Interrupt Enable)** – when set to 1 the modulus counter interrupt is enabled;
- **MODMC (Modulus Enable Mode)** – when this bit is set to 0 the counter will only count down once until reaching 0x0000. Setting it to 1 makes the counter reload with the previous value every time it reaches 0x0000;
- **RDMCL (Read Modulus Down-Counter Load)** – by setting this bit to 0 the values read from the modulus down counter will return the actual value of the counter while setting it to 1 returns the load register value;
- **ICLAT (Input Capture Force Latch Action)** – when input capture latch mode is enabled, writing 1 to this bit results in the immediate transfer of the TCO-3 input capture registers content and the content of the 8-bit pulse accumulators to their holding registers. This action leads to the clearing of the pulse accumulators;
- **FLMC (Force Load Register into Modulus Counter Count Register)** – writing 1 into this bit loads the load register into the modulus counter count register and resets the modulus counter prescaler

- **MCEN (Modulus Down-Counter Enable)** – when written to 1 this bit enables the modulus down counter while a 0 will disable it
- **MCPRx (Modulus Counter Prescaler)** – these bits are used to select the prescaler value for the modulus counter clock according to Table 5.8.

MCPR1	MCPR0	Prescaler division rate
0	0	1
0	1	4
1	0	8
1	1	16

Table 5.8 Modulus counter prescaler selection

The MCFLG register holds the flags associated to the 16-bit modulus down-counter:

- **MCZF (Modulus Counter Underflow Flag)** – this flag is set when the counter reaches 0x0000. Write 1 to this bit to reset the flag;
- **POLFx (First Input Capture Polarity Status)** – each POLFx bit gives the polarity of the first edge that caused an input capture to occur after a capture latch was read on timer PORTx input. 0 indicates a falling edge while 1 corresponds to a rising edge.

MCFLG

7	6	5	4	3	2	1	0
MCZF	-	-	-	POLF3	POLF2	POLF1	POLF0

The value of the modulus counter is available in the MCCNT register. This register should be read in a single clock cycle operation as separate readings of its low and high bytes will result in a different value.

MCCNT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

5.2.3 Registers related to the pulse accumulator

Each of the two 16-bit pulse accumulators of the ECT module are built by cascading two 8-bit pulse accumulators. The additional Pulse accumulator B is configured by the PBCTL register

PBCTL

7	6	5	4	3	2	1	0
-	PBEN	-	-	-	-	PAOVI	-

- **PBEN (Pulse Accumulator B Enable)** – when set to 1 this bit leads to the enabling of the two associated 8-bit pulse accumulators cascaded to form the 16-bit pulse accumulator;
- **PBOVI (Pulse Accumulator Overflow Interrupt enable)** – enables (when set to 1) the overflow interrupt on pulse accumulator B.

Interrupt flags associated to the pulse accumulator B are grouped in the PBFLG register and their function is similar to the corresponding flags in the PAFLG register.

PBFLG

7	6	5	4	3	2	1	0
-	-	-	-	-	-	PBOVF	-

The counter register associated to the pulse accumulator B is built by cascading two 8-bit registers PACN1 and PACN0. As in the case of PACNT, this register should also be read in a single clock cycle operation because separate readings of its low and high bytes will give different results as when reading as a word.

PACN1								PACN0							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit	bit
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Holding registers are available for all pulse accumulator registers. They are available as the 8-bit PAXH registers.

PAXH							
7	6	5	4	3	2	1	0
PAXH7	PAXH6	PAXH5	PAXH4	PAXH3	PAXH2	PAXH1	PAXH0

Exercise 7.1 Use the input capture functionality to measure the period of a signal that is generated on the pin associated to the Timer channel 0. Configure the signal generator to generate signals of various frequencies and shapes and connect its output to the Timer channel 0 input capture pin.

Exercise 7.2 Generate a 1kHz square waveform with a 50% duty cycle using the output compare function on the Timer channel 2. The generated signal should be checked by using the oscilloscope.

Exercise 7.3 Use the pulse accumulator functionality to generate an interrupt when a number of N pulses have been generated on the PT7 pin. Make connection between the PT7 pin and one of the four pushbuttons available on the development board. This will enable you to generate the pulses using one of the pushbuttons.

Exercise 7.4 Write an application that uses the timer to count the number of seconds elapsed since the start of the program. The least significant byte of the variable holding the number of seconds elapsed should be displayed on the LEDs associated to Port B.

6 PULSE WIDTH MODULATION (PWM) MODULE

The purpose of this chapter is to introduce basic concepts of pulse width modulation and the particularities of the S12 PWM module.

6.1 GENERAL CONCEPTS OF PWM

Pulse width-modulation (PWM) is the technique employed for encoding a given message in a pulsing signal. The main use of PWM is in electrical motor control. Although modulated signals could be generated by the use of other microcontroller components such as timers and GPIO this can be done more efficiently by using a hardware module dedicated to this purpose.

A PWM waveform is characterized by two parameters (as illustrated in Figure 6.1): *frequency* (or alternatively the *period*) and *duty cycle*. The PWM frequency will be derived from the microcontroller clock and can be configured by using prescalers. The duty cycle is the ratio between the active pulse duration and the signal period.

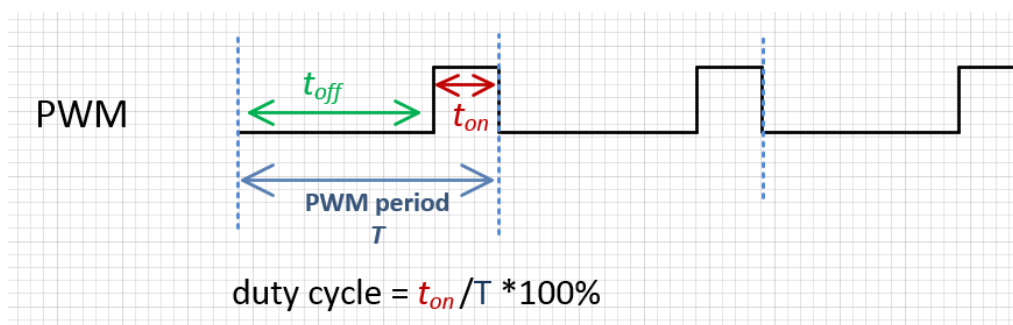


Figure 6.1 PWM waveform

6.2 PWM GENERATION ON S12

All members of the S12 family are equipped with PWM modules, however, the number of channels available on each member may differ. The S12C128 has 6 independent PWM channels which can be used in 8 bit mode or in 16 bit mode, where pairs of PWM channels are concatenated. The S12DT256 and S12XD512 have 8 PWM channels when used in 8-bit mode or a maximum of 4 PWM channels if used in 16-bit mode.

Figure 6.2 illustrates the block diagram of the PWM module as available on the S12C128. Other S12 family member will have 2 additional PWM channels. As the diagram suggests there are a set of common configuration registers for setting up the clock and controlling several signal parameters for all channels. Additionally, each channel has dedicated registers for individually configuring the signal period and duty cycle. The steps to take when configuring a PWM channel output on S12 are the following:

- configure PWM the clock
- configure the PWM waveform parameters as required: period, duty cycle, polarity and alignment
- enable the output on the selected PWM channel

The actual PWM period will be affected by both the PWM clock and the channel period register settings.

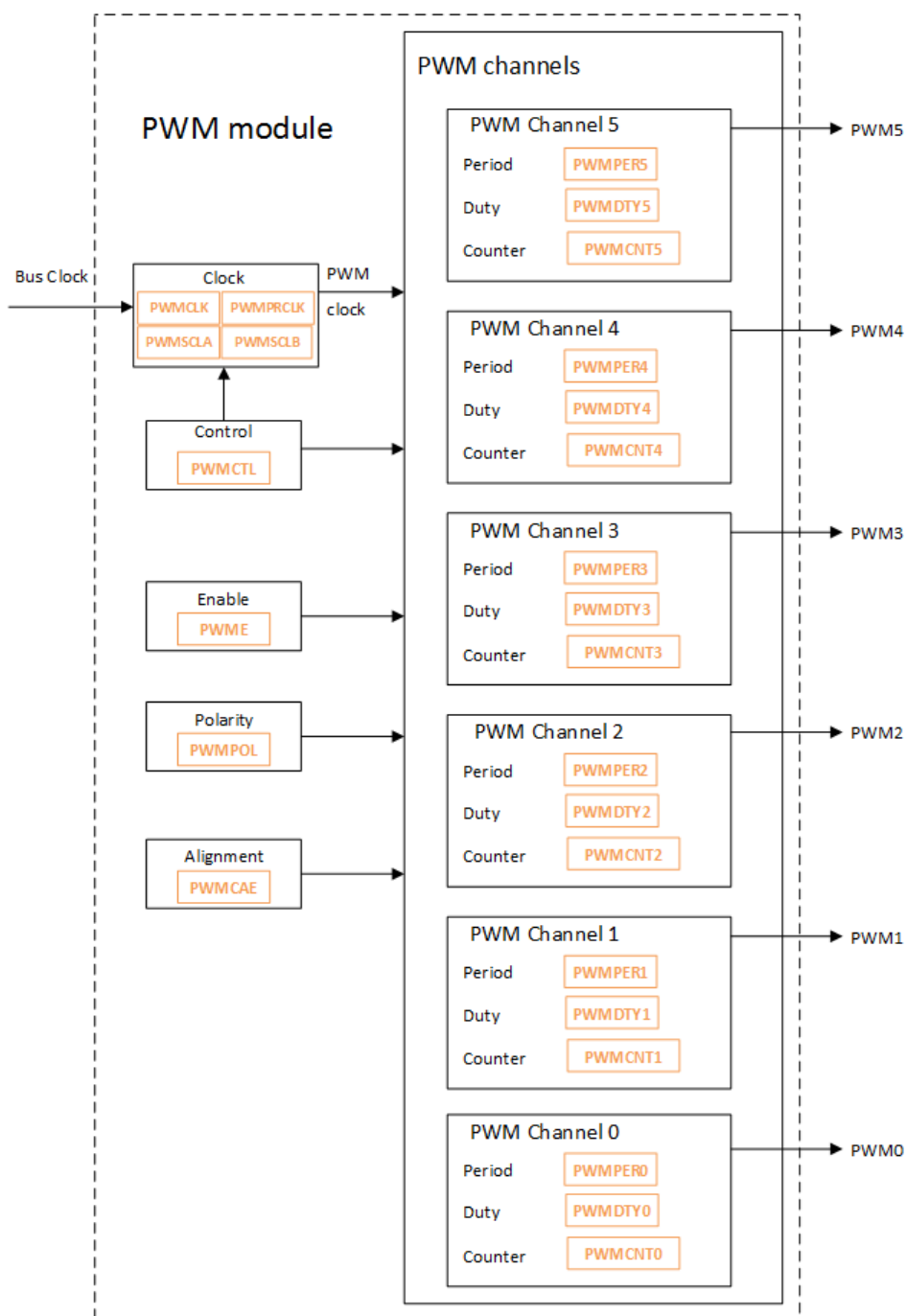


Figure 6.2 S12C128 PWM module Block Diagram

Table 6.1 provides an overview of the PWM module registers, the strikeout values are only available for factory testing purposes. A complete register description can be found in [15].

PWME - PWM Enable	R/W	PWMSCLA - PWM Scale A	R/W
PWMPOL - PWM Polarity	R/W	PWMSCLB - PWM Scale B	R/W
PWMCLK - PWM Clock Select	R/W	PWMSCNTA - PWM Scale A Counter	R/W
PWMPRCLK - PWM Prescale Clock Select	R/W	PWMSCNTB - PWM Scale B Counter	R/W
PWMCAL - PWM Center Align Enable	R/W	PWMCNTx - PWM Channel x Counter	R/W
PWMCNTL - PWM Control	R/W	PWMPERx - PWM Channel x Period	R/W
PWMTST - PWM Test	R/W	PWMDTYx - PWM Channel x Duty	R/W
PWMPRSC - PWM Prescale Counter	R/W	PWMSDN - PWM Shutdown	R/W

Table 6.1 Overview of the PWM associated registers (strikeout registers are for factory test purposes only)

Table 6.2 gives a summary of the registers needed to configure each of the PWM channels when working in 8 bit mode.

PWM channel	Enable	Polarity	Clock	Prescaler	Alignment	8/16 bit	
	PWME	PWMPOL	PWMCLK	PWMPRCLK	PWMCAE	PWNCTL	Output
PWM ch. 0	PWME0	PPOL0	PCLK0	PCKA2 PCKA1 PCKA0	CAE0	CON01	PWM0
PWM ch. 1	PWME1	PPOL1	PCLK1	PCKA2 PCKA1 PCKA0	CAE1	CON01	PWM1
PWM ch. 2	PWME2	PPOL2	PCLK2	PCKB2 PCKB1 PCKB0	CAE2	CON23	PWM2
PWM ch. 3	PWME3	PPOL3	PCLK3	PCKB2 PCKB1 PCKB0	CAE3	CON23	PWM3
PWM ch. 4	PWME4	PPOL4	PCLK4	PCKA2 PCKA1 PCKA0	CAE4	CON45	PWM4
PWM ch. 5	PWME5	PPOL5	PCLK5	PCKA2 PCKA1 PCKA0	CAE5	CON45	PWM5
PWM ch. 6	PWME6	PPOL6	PCLK6	PCKB2 PCKB1 PCKB0	CAE6	CON67	PWM6
PWM ch. 7	PWME7	PPOL7	PCLK7	PCKB2 PCKB1 PCKB0	CAE7	CON67	PWM7

Table 6.2 Summary for 8 bit mode PWM channel configuration

6.2.1 PWM Clock configuration

There are four clock sources that can be used for the PWM channels: Clock A, Clock B, Clock SA (scaled A) and Clock SB (scaled B). Clock A and B are obtained directly by dividing the bus clock by a factor of 1, 2, 4, 8, 16, 32, 64 or 128. The prescaling is done by configuring the **PWMPRCLK (PWM Prescale Clock Select)** register.

PWMPRCLK

7	6	5	4	3	2	1	0
-	PCKB2	PCKB1	PCKB0	-	PCKA2	PCKA1	PCKA0

- **PCKB[2, 0]** – used for selecting the prescaler value used to generate **clock B**. Table 6.3 presents the correspondence between the PCKB[2:0] value and **clock B** frequency
- **PCKA[2, 0]** – used to select the prescaler value used to generate **clock A**. The correspondence between the PCKA[2:0] value and the **clock A** frequency is presented in Table 6.3.

PCKx2	PCKx1	PCKx0	Value of clock x
0	0	0	Bus clock
0	0	1	Bus clock/2
0	1	0	Bus clock/4
0	1	1	Bus clock/8
1	0	0	Bus clock/16
1	0	1	Bus clock/32
1	1	0	Bus clock/64
1	1	1	Bus clock/128

Table 6.3 Prescaler options for the two main clock sources, x stands for A and B

If a lower clock frequency is needed clock A and B can be further scaled down by using an 8 bit down counter which loads a value configurable by the **PWMSCLA (PWM Scale A)** and **PWMSCLB (PWM Scale B)** registers respectively. The resulting clock signals are provided on the clock SA and SB lines.

PWMSCLA

7	6	5	4	3	2	1	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

For generating the clock SA a down counter loads the value from the **PWMSCLA** register, at each clock A cycle it decrements by 1 and. When it reaches 1 it generates a pulse and reloads the value from the PWMSCLA register. The obtained clock frequency is further divided by 2. Therefore clock SA is derived with respect with: $\text{clock SA} = \text{clock A} / (2 * \text{PWMSCLA})$, e.g., when $\text{PWMSCLA} = 0xFF$, $\text{clock SA} = \text{clock A} / (2 * 255)$.

A special case is considered when $\text{PWMSCLA} = 0x00$ because the down counter can never reach value 0 when decrementing. In this case clock A will be divided by 256 and then by 2. ($\text{clock SA} = \text{clock A} / (2 * 256)$).

PWMSCLB

7	6	5	4	3	2	1	0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

All the remarks presented about PWMSCLA are also applicable for PWMSCLB.

On each PWM channel we can select from two clock sources as follows: for PWM channels 0, 1, 4 and 5 clock A or clock SA can be selected while for PWM channels 2, 3, 6 and 7, clock B or clock SB can be set as a source. The clock source selection is made through the **PWMCLK (PWM Clock Select)** register.

PWMCLK

7	6	5	4	3	2	1	0
PCLK7	PCLK6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0

- **PCLKx** - used to select the corresponding PWM channel clock source for channel x, where $x \in \{0, 1, 4, 5\}$. When **PCLKx** is cleared to 0 **clock A** is selected as clock source, when set to 1 **clock SA** is selected as a clock source
- **PCLKy** - used to select the corresponding PWM channel clock source for channel y, where $y \in \{2, 3, 6, 7\}$ (note that channels 6 and 7 only available on some S12 family members). When **PCLKy** is cleared to 0 **clock B** is selected as clock source while setting it to 1 selects **clock SB** as a clock source for these channels.

6.2.2 PWM waveform configuration

The PWM waveform is controlled by registers associated to the four configurable properties: period, duty cycle, polarity and alignment.

PWM counters

Each 8 bit PWM channel has an associated 8 bit **PWMCNTx (PWM Channel Counter)** register used to implement the different operation modes of the module. The PWMCNTx is automatically configured as an up or up/down counter depending on the chosen PWM alignment mode (left aligned and center aligned). When using the left aligned mode for a PWM channel the counter register will act as an up counter, and when using a center aligned mode it will act as a up and down counter.

The **PWMCNTx** can be read at any time, special attention should be taken when writing this register as writing any value to the counter will:

- reset the counter to 0x00
- set the counting mode to up
- load the duty register (PWMDTYx) and the period register (PWMPERx) with the values from their corresponding buffers

PWM period

Each channel has a dedicated 8 bit **PWM period register PWMPERx** which determines the period of the signal generated by the corresponding channel. The period register is double buffered which means that a change to the register value will not affect the channel output unless one of the following occurs: the effective period ends, the channel counter is written (counter resets to 0x00) or the channel is disabled. The actual signal period can be calculated based on the channel clock source as follows:

- when in left alignment mode : $\text{PWMx period} = \text{PWMPERx} / (\text{PWM channel clock frequency})$
- when in center alignment mode: $\text{PWMx period} = 2 * \text{PWMPERx} / (\text{PWM channel clock frequency})$

Note that reading the PWMPERx register will not return the PWMPERx register value but will get the value from the PWMPERx buffer

PWM duty cycle

The duty cycle is also configured by a double buffered 8 bit **PWM channel duty register PWMDTYx** one allocated for each channel. Similar to the PWMPERx register a write to the PWMDTYx does not immediately change the wave duty cycle but only after the occurrence of one of the three events previously mentioned. Based on the register value and the waveform polarity the duty cycle is calculated as follows:

- when the PWM waveform start polarity equals 0: $\text{PWMx duty cycle} = (\text{PWMPERx} - \text{PWMDTYx}) / \text{PWMPERx} * 100\%$
- when the PWM waveform start polarity equals 1: $\text{PWMx duty cycle} = (\text{PWMDTYx}) / \text{PWMPERx} * 100\%$

PWM polarity

The PWM polarity represents the starting voltage level of a PWM period as illustrated in Figure 6.3. The **PWMPOL (PWM Polarity)** register is used to set the polarity on all PWM channels, each register bit corresponding to one channel. Clearing **PPOLx** sets a low starting voltage on the corresponding PWM channel output, while setting it to 1 makes the corresponding PWM channel output starts on high.

PWMPOL

7	6	5	4	3	2	1	0
PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0

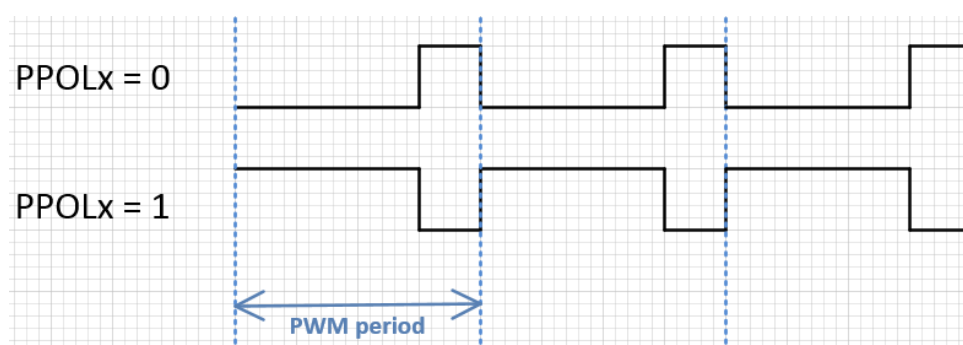


Figure 6.3 PWM waveform polarity

PWM waveform alignment

Two types of alignment of the PWM signal within a period are provided by the S12 PWM module: left alignment and center alignment. The alignment mode is configured by the **PWMCAE (PWM Center Align Enable)** register.

PWMCAE

7	6	5	4	3	2	1	0
CAE7	CAE6	CAE5	CAE4	CAE3	CAE2	CAE1	CAE0

CAEx bits are used to select PWM waveform alignment for each PWM channel. When **CAEx** is cleared to 0 a left aligned PWM waveform is generated for channel x. When **CAEx** is set to 1 a center aligned PWM waveform is generated for channel x.

When configuring a PWM channel in left aligned mode the PWM Channel Counter register will act as an up counter only. The **PWMCNTx** will be continuously compared with the PWM Channel Duty Register (**PWMDTYx**) and when a match occurs the corresponding PWM signal output is complemented as illustrated in Figure 6.4. **PWMCNTx** is also continuously compared with the PWM Channel Period register (**PWMPERx**). When a match occurs the **PWMCNTx** value is reset and both **PWMDTYx** and **PWMPERx** are updated with the values from their corresponding buffers.

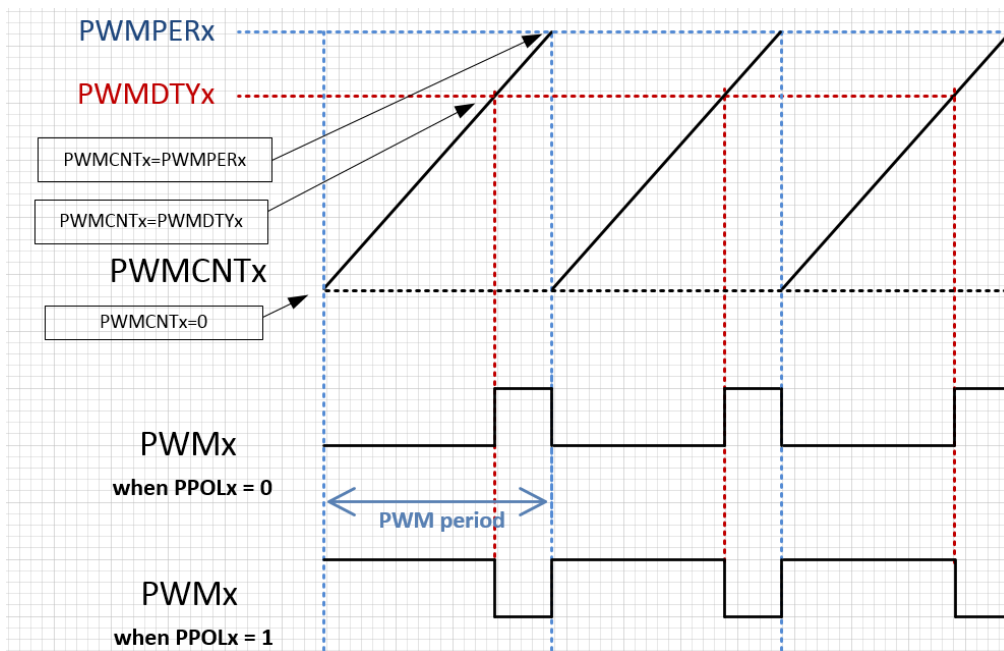


Figure 6.4 Left aligned PWM waveform

When configuring a PWM channel in center aligned mode the PWM Channel Counter register will act as a up and down counter. Similarly to the left aligned mode, the PWM Channel Counter register (**PWMCNTx**) is continuously compared with the PWM Channel Duty register (**PWMDTYx**). When a match occurs the corresponding PWM signal output is complemented. When a match between the **PWMCNTx** and the PWM Channel Period Register (**PWMPERx**) occurs, the **PWMCNTx** changes the counting direction from UP to DOWN. When the **PWMCNTx** reaches 0 while counting DOWN the **PWMDTYx** and **PWMPERx** are updated with the values from their corresponding buffers and the counter register changes the counting direction back from DOWN to UP. This process along with the resulting waveform is illustrated in Figure 6.5. While in this operation mode the waveform parameters on channel x are as follows:

- $\text{PWMx frequency} = \text{ch_x_clock_frequency} / (2 * \text{PWMPERx});$
- $\text{PWMx period} = \text{ch_x_clock_period} * \text{PWMPERx} * 2 ;$
- $\text{PWMx duty cycle} = (\text{PWMCNTx} - \text{PWMDTYx}) / \text{PWMCNTx} * 100\%$, when starting polarity is 0;
- $\text{PWMx duty cycle} = (\text{PWMDTYx}) / \text{PWMCNTx} * 100\%$, when starting polarity equals 1.

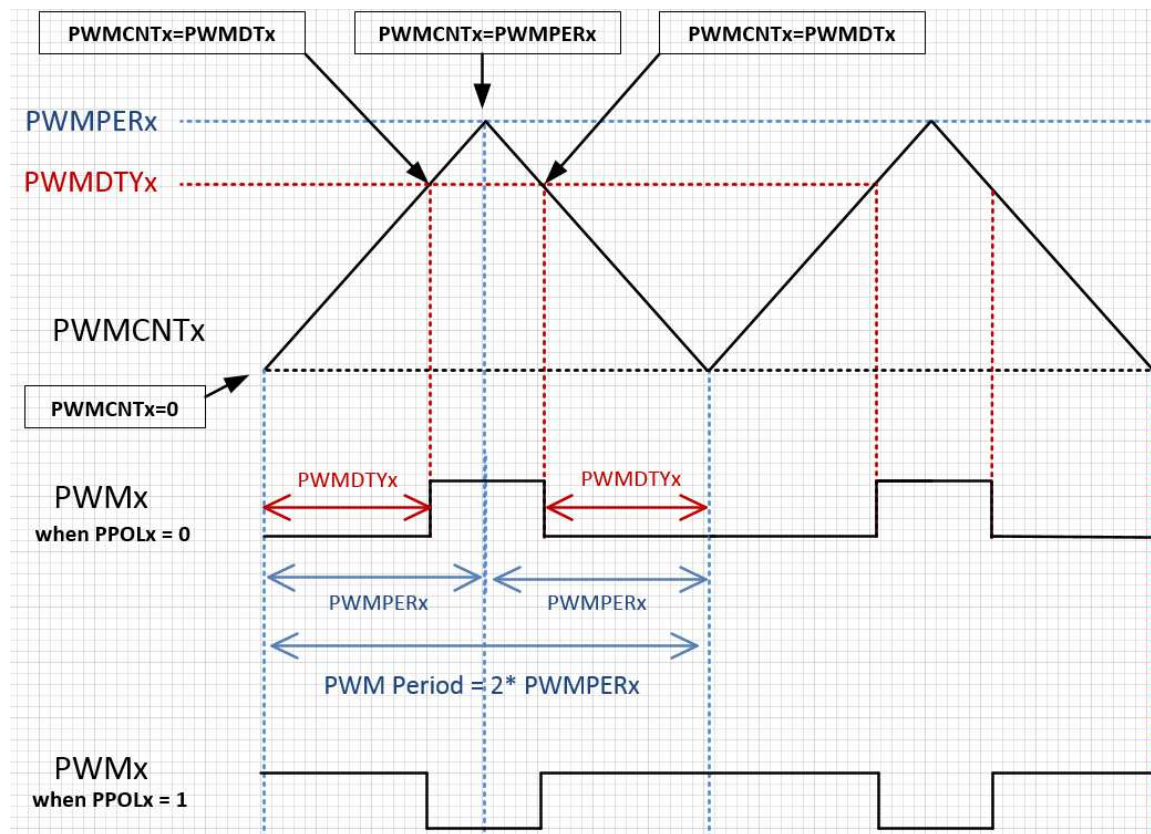


Figure 6.5 Center aligned PWM waveform

6.2.3 PWM module control

The **PWM Control (PWMCTL)** register is used to control the PWM signal resolution and the PWM module behavior in wait and freeze mode.

PWMCTL

7	6	5	4	3	2	1	0
CON67	CON45	CON23	CON01	PSWAI	PFRZ	0	0

- **CON67, CON45, CON23, CON01** – are used to concatenate two 8 bit PWM channels into a 16 bit PWM channel. When set to 1, **CON01** concatenates PWM channel 0 and PWM channel 1. When set to 1 **CON23** concatenates PWM channel 2 and PWM channel 3. When set to 1, **CON45** concatenates PWM channel 4 and PWM channel 5. **CON67** acts in a similar manner for channels 6 and 7. When the register concatenation bits (CON67, CON45, CON23, CON01) are cleared to 0 the corresponding PWM channel has a 8 bit resolution.
- **PSWAI** (PWM Stops in Wait Mode)—when cleared to 0, the bus Clock is fed into the prescaler when the MCU is in **wait mode**, when set to 1 the bus Clock is no longer fed to the prescaler (this leads to the disabling of the PWM module => lower power consumption)
- **PFRZ** (PWM Counters Stop in Freeze Mode) - when cleared to 0, the bus Clock is fed into the prescaler when the MCU is in **freeze mode**, when set to 1 the bus Clock is no longer fed to the prescaler (this leads to the disabling of the PWM module => lower power consumption).

The **PWMSDN (PWM Shutdown)** register provides emergency shutdown functionality for the PWM module. The pin used to implement this functionality is the pin associated with the last PWM channel output, i.e. channel 7 for most devices or channel 5 for devices with only 6 PWM channels.

PWMSDN

7	6	5	4	3	2	1	0
PWMIF	PWMIE	PWMRSTRT	PWMLVL	0	PWM7IN	PWM7INL	PWM7ENA

- **PWM7ENA** – when set to 1, enables the emergency shutdown functionality and forces the PWM Channel 7 corresponding output pin to input (PP7/PWM7).
 - **PWM7INL** – indicates the shutdown active level to react on when the emergency shutdown is active. A low active level is considered when PWM7INL is 0, while a high level is indicated by a 1
 - **PWM7IN** – this bit reflects the current status of the PWM channel 7 pin
 - **PWMLVL** – when the expected level is found on the last channel pin the outputs of the other PWM channels will be set to the value indicated by PWMLVL
 - **PWMRSTRT** – writing a 1 to this bit restores the PWM operation but only when the asserted input is changed
 - **PWMIE** – writing a 1 to this bit enables the PWM interrupt for emergency shutdown
 - **PWMIF** – this interrupt flag indicates the asserted state. Write a logic 1 to this bit to reset it
- The **PWME (PWM Enable)** register is used to individually enable/disable the PWM channels.

PWME

7	6	5	4	3	2	1	0
PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0

- **PWME_x** – when cleared to 0 the corresponding PWM channel is **disabled**, when set to 1 the corresponding PWM channel is **enabled**.

EXAMPLE 5.1 Configure the PWM module to generate a 1.25MHz PWM signal on channel 0 with a 25% duty cycle. A 10 MHz bus clock should be considered.

Solution: Clock A or SA can be a source for PWM channel 0, therefore we first configure a prescaler for clock A. By using a Bus Clock/2 prescaler we obtain a 5MHz (200 ns period) clock for the PWM channel 0. Further period scaling will be made by using the dedicated period register. The PWM frequency/period is determined as follows:

- PWM0 period = PWM channel 0 clock period * PWMPER0
- PWM0 frequency = PWM channel 0 clock frequency / PWMPER0

thus, PWM 0 frequency = 5 MHz / 4 = 1.25 MHz.

To get the required duty cycle, the PWMDTY0 register must be configured according to: PWM duty cycle = [(PWMPER0 – PWMDTY0) / PWMPER0] * 100% = 25%.

The PWM waveform polarity will be set to 0. The last step is to enable the output on the PWM channel by setting the PWME register. The C code segment needed to configure PWM registers as presented is given below.

```
PWMCLK=0;           // select Clock A as clock source for PWM channel 0 (PCLK0=0)
PWMPRCLK=1;        // configure the prescaler for Clock A =Bus Clock / 2
PWMPOL=0;          // configure PWM starting polarity as 0 (PWMPOL0=0)
PWMPER0=0x04;     // configure PWM channel 0 period
PWMDTY0=0x03;     // configure PWM channel 0 duty cycle
PWME=0x01;        // Enable PWM channel 0
```

The resulting PWM signal should look as illustrated in Figure 6.6.

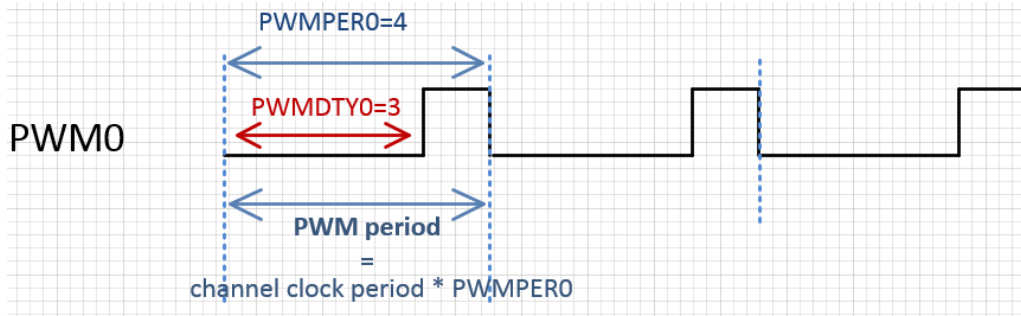


Figure 6.6 PWM waveform generated in exercise 5.1

6.2.4 16 bit resolution PWM

When a higher resolution is desired for the PWM waveform, the 8 bit PWM channels can be concatenated in pairs, to obtain 16 bit PWM channels. The concatenation is controlled by the CONx bits of the PWMCTL register (CON01 for channels 0 and 1, CON23 for channels 2 and 3, CON45 for channels 4 and 5 and CON67 for channels 6 and 7).

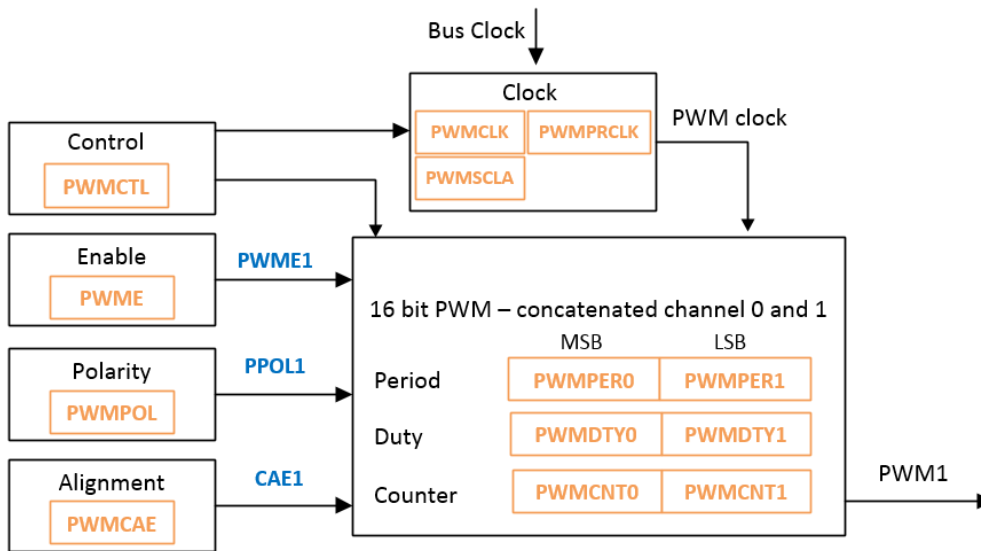


Figure 6.7 Diagram of the 16 bit PWM channel obtained by concatenating channels 0 and 1

When two PWM channels are concatenated the lower numbered register in the pair becomes the high-order byte of the double byte channel. As shown in Figure 6.7, for the case of concatenating channels 0 and 1, the PWMCNT0, PWMDTY0 and PWMPER0 registers take the role of high order byte. All the other control features (enable, polarity, alignment, clock source -including the prescaling functions) of the 16 bit PWM channels are controlled by the low order bytes (higher number in the pair). in the case of concatenating channel 0 and 1 the following configuration bits are used: PWME1, PPOL1, PCLK1, CAE1.

PWNCTL	PWME	PWMPOL	PWMCLK	PWMRCLK	PWMCAE	PWM Output
CON67	PWME7	PPOL7	PCLK7	PCKB2 PCKB1 PCKB0	CAE7	PWM7
CON45	PWME5	PPOL5	PCLK5	PCKA2 PCKA1 PCKA0	CAE5	PWM5
CON23	PWME3	PPOL3	PCLK3	PCKB2 PCKB1 PCKB0	CAE3	PWM3
CON01	PWME1	PPOL1	PCLK1	PCKA2 PCKA1 PCKA0	CAE1	PWM1

Table 6.4 16 bit PWM channels configuration registers

EXERCISE 5.1 Generate two 12.5 KHz PWM signals, with a 20% duty cycle on channel 1 and an 80% duty cycle channel 5. Use the oscilloscope to view the generated signals.

EXERCISE 5.2 Repeat the previous exercise but for channels 2 and 3.

EXERCISE 5.3 Remove the jumper mounted to connect the LED associated to the Port B 0 pin on the ZK-S12-B board and connect the PWM channel 0 output to the LED-side pin. Use the PWM channel 0 output to control the brightness of this LED. The brightness change will be controlled by using the 4 switches on the board as follows:

- pushing SW0 sets the brightness to 20%
- pushing SW1 sets the brightness to 40%
- pushing SW2 sets the brightness to 60%
- pushing SW3 sets the brightness to 80%

Hint: The brightness is controlled by changing the duty cycle. A 100% duty cycle will lead to a full brightness level while a 0% duty cycle will lead to a turned off LED.

7 THE ANALOG-TO-DIGITAL CONVERTER UNIT (ATD)

Depending on the chosen variant, the S12 chip has 1 or more identical ATD units (while ATD is the name of the S12 converter, ADC is the more often employed acronym for analog to digital converters). Each ATD unit is able to perform conversions on 8 or 10 bit resolution at a sample rate that varies between 500 kHz and 2 MHz. A single 8 bit conversion can be performed as quickly as 6 μ s while 10-bit conversions require 7 μ s. These conversions are accurate at ± 1 in the Least Significant Bit (LSB). The use of 10-bit resolution is encouraged only when the application mandates it, otherwise note that this requires 16 bits to store the result and also increases the acquisition time; while 16 bits are not necessary a high demand, the main idea behind embedded applications is to be rational when spending resources. For further reading, see [16] as the reference manual for the ATD unit and also [17] as an introduction on how this unit works.

Practical consideration. Always use the 8-bit conversion in favour of 10-bit if this is enough for your application. Using a lower conversion rate will always increase the accuracy of the conversion.

7.1 GENERAL ADC CONCEPTS AND THE S12 ATD UNIT

7.1.1 Applications and some theoretical considerations on sampling rate, resolution and accuracy.

In general, the ADC converter is used whenever acquiring data from sensors is needed. This is because a sensor can represent the value of some continuous physical quantity (there are countless examples: temperature, humidity, pressure, acceleration, weight, etc.) by means of electric voltage and then the ADC unit can further convert this voltage in a digital (binary) format. Briefing through some theoretical notions is in order:

- A strong theoretical result is the *Nyquist-Shannon sampling theorem* which states that in order to perfectly reconstruct a continuous signal (analog) of limited bandwidth B, it is sufficient to sample it at a rate that is twice the bandwidth 2B. This sampling rate of 2B (which is twice the highest frequency for baseband signals) is also called the *Nyquist rate*. While this gives you a hint on the rate at which signals should be sampled, it does not mean that if you do so you will be able to perfectly reconstruct the signal because in practice your conversion is also limited by the resolution of the measurement (you will always sample on a predefined number of bits, but a continuous signal has an amplitude that is part of the real domain, thus a perfect reconstruction may not be practically possible).

- The resolution of the ADC is the number of discrete values that it can produce over the analog range. For ADC this is usually expressed in bits, for an 8-bit ADC you have $2^8 = 256$ values and for 10-bit ADC you have $2^{10} = 1024$ values. The resolution can also be expressed in volts by dividing the full input range of the signal to the number of discrete values, i.e., $(VRH - VRL)/2^b$, where b is the resolution in bits. For example, for b = 8, we have a $\frac{5-0}{256} = 19.53mV$ resolution.

- The precision and accuracy are two relevant (and distinct) notions related to the measurement. Accuracy means how close your measurement is to the real (true) value. Precision is the degree in which your measurement, if repeated in the same conditions, will yield the same result.

7.1.2 The S12 ATD unit

Figure 7.1 presents the block diagram of the ATD module according to [16], we will detail this in what follows. To begin with, the following need to be defined:

- ANx are the analog signal input pins,

- PADx are the pins used for general purpose digital input,
- VDDA and VSSA are the power (voltage) supplies of the circuit,
- VRH and VRL are the high and low reference voltages. For simplicity VRL is usually set to ground (0V) and VRH to VDD (5V).

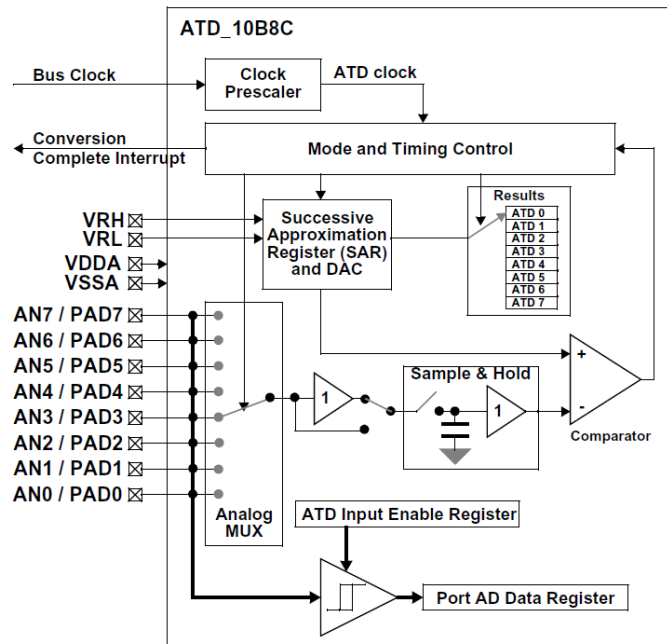


Figure 7.1 Block diagram of the ATD unit from S12 (as described in [16])

The main elements of the ADC are now described based on [17]. Let us group them in two categories:

- The analog subsystem** – contains the analog part that is necessary for conversion:
 - MUX – a multiplexer that selects one of the 8 inputs,
 - an input sample amplifier is connected next to the MUX,
 - RC-DAC – a resistor-capacitor digital-to-analog converter,
 - Comparator – a high-gain comparator that indicates if each successive output of the RC-DAC is higher or lower than the sampled input.
- The digital control subsystem** – contains the registers and logic for controlling the conversion:
 - SAR – the successive approximation register that stores one bit at a time and when finishing it transfers the result to the result register,
 - Mode and Timing control – used to specify the format of the result, the multiplexer input, sample time, etc.,
 - ATD Input Enable register – controls the input buffer from the analog input pin ANx.

Prerequisites for using the ATD

Before performing a conversion VDDA must be connected to 5V, VSSA to 0V, VRH must be less or equal to 5V and VRL must be higher than 0 but less than VRH. The circuit needs about 20 μ s to stabilize and the result of the conversion can be read only after the SCF flag is set, make sure to wait at least 20 μ s when powering up the ATD (make sure to implement a small sub-routine to guarantee this delay).

Steps when acquiring conversion results.

Make sure to follow the following steps whenever performing conversions (registers mentioned below are described in the following section):

- a. power up the ATD by setting ADPU bit in ATDCTL2,
- b. configure the ATD as desired for your application by setting up ATDCTL3 and ATDCTL4 (wait at least 20 μ s after this step),
- c. select the channel on which you want to perform conversion by writing to ATDCTL5,
- d. whenever you want to read the result of the conversion, make sure to test the SCF flag in the ATDSTAT register.

7.1.3 Conversion algorithm.

Clarifying how conversion actually works may be useful. The ATD circuit uses the successive-approximation method for conversion. That is, it first initializes the SAR register to 0 and then makes bit-by-bit guesses starting from the most-significant bit. Concretely, starting from the most-significant bit, the bit is set to 1, then the value from the SAR is converted to an analog voltage and compared via the Comparator to the original value. If the value from the SAR is larger, then clears the bit to 0. The operation is repeated for each subsequent bit until reaching the least significant one.

7.2 ATD ASSOCIATED REGISTERS

Table 7.1 provides an overview of the ATD registers. The striked-out values are only available for factory testing purposes. From the rest of the registers, the ones that are of more interest to us are the control registers, status registers and result registers. These are going to be explained next.

ATDCTL0 – ATD Control 0	R	ATDTEST0 – ATD Test 0	R
ATDCTL1 – ATD Control 1	R	ATDTEST1 - ATD Test 1	RW
ATDCTL2 - ATD Control 2	RW	ATDSTAT1 - ATD Status 1	R
ATDCTL3 - ATD Control 3	RW	ATDDIEN - ATD Input Enable	RW
ATDCTL4 - ATD Control 4	RW	PORTAD - Port Data	R
ATDCTL5 - ATD Control 5	RW	ATDDRHx - ATD Conversion Result High	RW
ATDSTAT0 - ATD Status 0	RW	ATDDRL - ATD Conversion Result Low	RW

Table 7.1 Overview of the ATD associated registers and their (striked-out registers are for factory test purposes only)

7.2.1 ATD control registers

ATDCTL2

7	6	5	4	3	2	1	0
ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF

The ATDCTL2 registers controls power-down, triggers and interrupts as follows:

- **ADPU (ATD Power Down/UP)** - cleared to 0 powers down the ATD, when set to 1 results in normal ATD functionality,
- **AFFC (ATD Fast Flag Clear All)** - cleared to 0 results in ATD flag clearing to operate normally (read the status register ATDSTAT1 before reading the result register), when set to 1 any access to a result register will cause the associate CCF flag to clear automatically,
- **AWAI (ATD Power Down in Wait Mode)** - when cleared to 0 ATD continues to run while S12 is in Wait mode, when set to 1 it halts conversion and power down ATD during Wait mode
- **ETRIGLE (External Trigger Level/Edge Control)** and **ETRIGP (External Trigger Polarity)** control the trigger edge and polarity as seen in table below:

ETRIGLE	ETRIGP	External Trigger Sensitivity
0	0	Falling edge
0	1	Rising edge
1	0	Low level

1	1	High level
---	---	------------

- **ETRIGE (External Trigger Mode Enable)** - when cleared to 0 disables external trigger, when set to 1 enables external trigger. The external trigger must be applied on channel 7, i.e., PAD7.
- **ASCIE (ATD Sequence Complete Interrupt Enable)** - when cleared to 0 ATD Sequence Complete interrupt requests are disabled, when set to 1 an ATD Interrupt will be requested whenever ASCIF=1 is set.
- **ASCIF (ATD Sequence Complete Interrupt Flag)** - cleared to 0 when no ATD interrupt occurred and set to 1 when ATD sequence complete interrupt pending (writes have no effect on it)

ATDCTL3

7	6	5	4	3	2	1	0
-	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0

ATDCTL3 is responsible for the length of the conversion, behaviour in freeze mode and the FIFO mode as follows:

- **SxC (Conversion Sequence Length)** - set the length of the conversion as follows:

S8C	S4C	S2C	S1C	Conversion Sequence Length
0	0	0	0	8 conversions
0	0	0	1	1 conversion
0	0	1	0	2 conversions
0	0	1	1	3 conversions
0	1	0	0	4 conversions
0	1	0	1	5 conversions
0	1	1	0	6 conversions
0	1	1	1	7 conversions
1	x	x	x	8 conversions

- **FIFO (Result Register FIFO Mode)** - when cleared to 0 the result of the first conversion appears in the first result register, the result of the second in the second result registers, etc., when set to 1 conversion results are placed in consecutive result register and wrap around at the end,
- **FRZx (Background Debug Freeze Enable)** - operates as follows:

FRZ1	FRZ0	Freeze mode behavior
0	0	Continues conversion
0	1	Reserved
1	0	Finish current conversion then freeze
1	1	Freeze immediately

ATDCTL4

7	6	5	4	3	2	1	0
SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0

ATDCTL4 is responsible for the length of the conversion, sample time and clock pre-scaler as follows:

- **SRES8 (A/D Resolution Select)** - when cleared to 0 it gives 10 bit resolution, when set to 1 it gives 8 bit resolution

- **SMP1 and SMP0 (Sample Time Select)** - selects the length of the second phase of the sample time according to the following table.

SMP1	SMP0	Sample time
0	0	2 A/D conversion clock periods
0	1	4 A/D conversion clock periods
1	0	8 A/D conversion clock periods
1	1	16 A/D conversion clock periods

- **PRSx (ATD Clock Prescaler)** - used to divide the Bus Clock which yields an ATD clock equal to $\text{BusClock}/(\text{PRS}+1)*0.5$. For example if PRS is set to 00011 (3) the ATDclock is the BusClock divided by 8.

ATDCTL5

7	6	5	4	3	2	1	0
DJM	DSGN	SCAN	MULT	-	CC	CB	CA

ATDCTL5 is responsible for the type of conversion and the inputs that are sampled (any write to this register aborts the current activity and starts a new conversion) as follows:

- **DJM (Result Register Data Justification)** - when cleared to 0 sets left justified data, when set to 1 selects right justified data in the result register,
- **DSGN (Result Register Data Signed or Unsigned Representation)** - when cleared to 0 unsigned data is selected and when set to 1 signed data representation is available in the result registers,
- **SCAN (Continuous Conversion Sequence Mode)** - when cleared to 0 a single conversion takes place, when set to 1 continuous conversions are taken,
- **MULT (Multi-Channel Sample Mode)** - when cleared to 0 a single channel is sampled, when set to 1 multiple channels are sampled (the number of channels is given by the SxC bits),
- **CC, CB, CA (Analog Input Channel Select Code)** - selects the input channel, when MULT = 0 this is precisely the channel that is measured, when MULT = 1 this is the first channel to be measured and subsequent channels are measured in incremental order (wrapping around the minimum value), e.g., 000 is AN0, 001 is AN1, 010 is AN2, etc.

EXERCISE 6.1 The following sequence of code can be seen in the demo that comes with your S12 development kit. Please explain how the ADC works when set via the following lines. Detail the effect of each line

```
ATDCTL3 = 0x08;
ATDCTL4 = 0x82;
ATDCTL2 = 0x80;
ATDCTL5 = 0x30;
```

7.2.2 ATD status registers

ATDSTAT0

7	6	5	4	3	2	1	0
SCF	-	ETORF	FIFOR	-	CC2	CC1	CC0

The ATDSTAT0 register provides a set of indicator flags for the status of various functions. It contains the following fields:

- **SCF (Sequence Complete Flag)** – will be set to 1 when a conversion is completed. When continuous conversions are enabled this flag is set after the completion of each conversion. The flag is cleared by setting it to 1 or as a consequence of writing to ATDCTL5 (starting a new conversion) or when a result register is read while AFFC = 1.
- **ETORF (External Trigger Overrun Flag)** – this flag is set if additional active edges are detected during a conversion sequence when edge trigger mode is active. The flag is cleared by setting it to 1 or if any of the non-reserved ATDCTLx registers are written to.
- **FIFOR (FIFO Overrun Flag)** – this flag is set when a result register is written to before clearing the conversion complete flag associated to it. To clear this flag set it to 1 or start a new conversion sequence.
- **CCx (Conversion Counter)** – The conversion counter bits indicate the result register that holds the result of the current conversion. The counter is always reset to 0 at the beginning and end of a conversion in non-FIFO mode but will not be reset in FIFO mode. When the maximum value is reached the counter wraps around to 0. The conversion counter is cleared by a write to a non-reserved ATDCTL register.

ATDSTAT1

7	6	5	4	3	2	1	0
CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0

The ATDSTAT1 register is a read-only register that contains the **Conversion Complete Flags (CCF)** for all channels. Each conversion flag CC is automatically set at the end of a conversion in a sequence of conversions. Each flag is associated to a position in the conversion sequence, eg. CCF0 is associated to the first conversion in a sequence, CCF1 to the second one and so on. When CCFx is 0 the xth conversion in the sequence is not completed while when it is set to 1 the conversion is completed and the result is available in the ATDDRx register.

7.2.3 ATD test registers

ATDTEST0 is reserved, writing to it can be done only in special modes and can alter functionality. Reading it can be done at any time but returns undefined values.

ATDTEST1

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	SC

ATDTEST1 register has a single implemented bit – the **SC (Special Channel Conversion) Bit**. This bit can be set to 1 to enable special channel conversions. If it is set to 0 the special channel conversions will be disabled. When in special channel conversion mode, the analog input behaves as illustrated in the table below.

SC	CC	CA	CB	Analog Input Channel
1	0	X	X	Reserved
1	1	0	0	V_{RH}
1	1	0	1	V_{RL}
1	1	1	0	$(V_{RH}+V_{RL})/2$
1	1	1	1	Reserved

7.2.4 ATD input enable register

ATDDIEN1

7	6	5	4	3	2	1	0
IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0

The ATDDIEN register is used to configure the analog input pins as digital inputs. Setting IENx to 1 enables digital input buffer from ANx to the PTADx data register.

7.2.5 ATD port data register

PORTAD

7	6	5	4	3	2	1	0
PTAD7	PTAD6	PTAD5	PTAD4	PTAD3	PTAD2	PTAD1	PTAD0

PORTAD is used to read the digital signal levels of the pins (the pins are shared with analog inputs AN0..AN7). If the digital input is enabled on pin x, that is IENx=1, then PTADx returns the logic level on ANx, otherwise PTADx will read 1.

7.2.6 ATD conversion result registers

The results of the conversion are stored in pairs of 8-bit read-only registers ATDDRHx/ATDDRLx. The result data is formatted according to the justification and sign settings. When left justification is selected the result is positioned so that the MSB of the result is located in the MSB of the ATDDRHx register. If right justification is used the result starts with its LSB situated in the ATDDRLx LSB. Signed data is stored as 2's complement and can be used only with left justification. Signed data setting is ignored when right justification is active.

EXAMPLE 6.1 In this exercise you will have to use an oscilloscope (Figure 7.2 shows a Tektronix TDS1001B oscilloscope) and a function generator like the TG120 pictured in Figure 7.3. You are requested to generate rectangular waves of [0, 5V] with a period of 1 to 8 ms and compute the length of this period by using your S12 development kit. You will light up LEDs 1 to 8 (LD0-7) according to the period of the signal. Figure 7.4 shows the board setup for this exercise. Note the following:

- a wire is connected to pin 56 which is channel 05 of the ATD
- the oscilloscope and function generator probes are connected to the other end of the wire
- set the common ground pin on the board.

Solution: The next step to take after making the HW setup is to set up the ATD module. We decided to use 8-bit resolution, and continuous conversions on channel 5. All this is configured by using the ATDCTLx registers. The period of the signal will be calculated by incrementing a counter after each conversion until the first encountered falling edge. The counter value along with the ATD timing settings will give a measure on the signal period. This can be then used to light up the LEDs according to the exercise requirements.

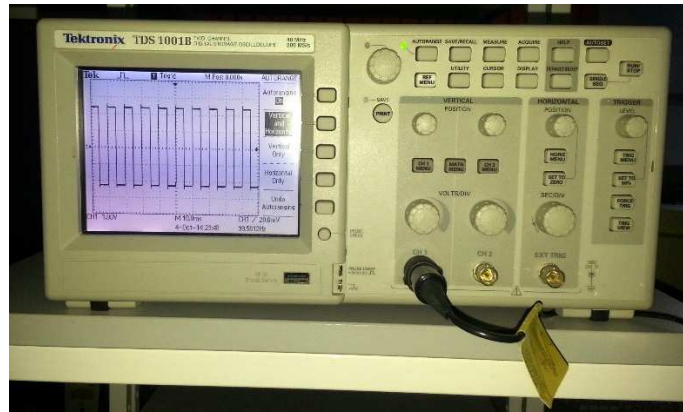


Figure 7.2 A Tektronix TDS1001 oscilloscope (use it to visualise the signal generated by the function generator)



Figure 7.3 TG120 function generator (use it to generate the square wave that is the input to the ADC)

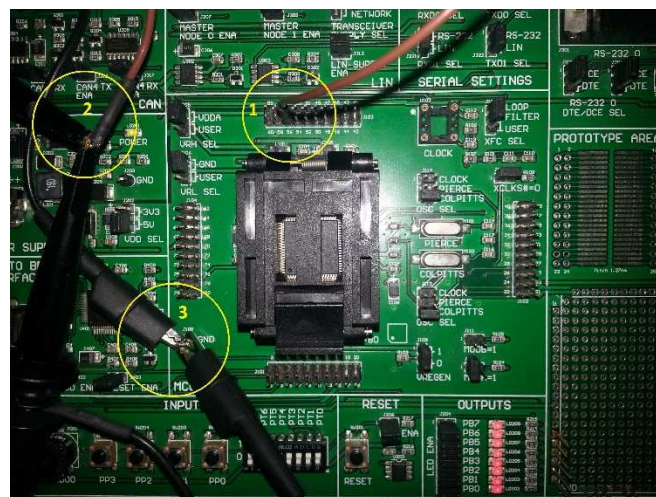


Figure 7.4 The board setup for this exercise

The complete solution for implementing this example is given in the following code snippet.

```
void wait1()
{
    asm{
        LDX #$230
loop2:
        DEX
        BNE loop2
    }
```

```

}
}

void main(void)
{
    byte current_value = 0, previous_value = 0;
    int period = 0;

    ATDCTL2 = 0x80; //power up the ADC by setting ADPU bit in ATDCTL2
    wait1();      //wait 100us before using the ATD
    ATDCTL3 = 0x00; //configure ATDCTL3, ATDCTL4 and ATDCTL5
    ATDCTL4 = 0x81; //8-bit resolution, ATDClk=BusClk/4
    ATDCTL5 = 0x25; //contious conversions, i.e., SCAN, on channel 5: PAD05/AN05, i.e. pin 56

    for(;;)
    {
        while((ATDSTAT0 & 0x80) == 0) {} //waits while SCF bit is set to 1
        //result is in ATDDR0
        current_value = ATDDR0H;
        if ((current_value - previous_value) < 5) {
            period++;
        }
        else {
            period = 2*period;
            // f = 1Hz => T = 1000ms
            if (period > 1000) { PORTB = 0x01;}
            // f = 2Hz => T = 500ms
            else if ((period > 500) && (period < 1000)) { PORTB = 0x03;}
            // f = 3Hz => T = 333ms
            else if ((period > 333) && (period < 500)) { PORTB = 0x07;}
            // f = 4Hz => T = 250ms
            else if ((period > 250) && (period < 333)) { PORTB = 0x0F;}
            // f = 5Hz => T = 200ms
            else if ((period > 200) && (period < 250)) { PORTB = 0x1F;}
            // f = 6Hz => T = 166ms
            else if ((period > 166) && (period < 200)) { PORTB = 0x3F;}
            // f = 7Hz => T = 140ms
            else if ((period > 140) && (period < 166)) { PORTB = 0x7F;}
            // f = 8Hz => T = 125ms
            else if (period < 140) { PORTB = 0xFF;}
            period = 0;
        }
        previous_value = current_value;
        wait1();
    }
}

```

EXERCISE 6.2 Using the code from the previous example, measure the characteristics of the square wave and output an identical one by the use of the PWM circuit. Use channel 2 of the oscilloscope to plot the second square wave. Consider also asymmetrical signals

8 INTERNAL MEMORY

This chapter will present basic memory concepts and guide you through the S12 internal memory system by presenting the types of available memory, its organization and mapping.

8.1 BASIC CONCEPTS ON MEMORIES

8.1.1 Common memory types

Most microcontrollers require different types of memories to fulfill their tasks. Some are built for being used along with external memories but more commonly they come equipped with the needed memories. Memories can be divided in two main categories: random access memory (RAM) and read only memory (ROM).

RAM memories are volatile memories (their content is lost once the power supply is turned off) which can be written to or read from. They have shorter access times in comparison to ROM memories and are usually used to store temporary data during program execution. Several types of RAM memory are available such as the dynamic RAM (DRAM) and the static RAM (SRAM). SRAM has a smaller density and more power consuming than DRAM but it offers faster memory access.

On the other side, ROM memories are nonvolatile (their content is retained even when the power supply is removed), with a much longer access time than RAM and are typically used to store the program object code and constants. The first ROM memory types were not reprogrammable (ROM and PROM). They could only be programmed once and if changes were made to the firmware they had to be completely replaced. Erasable programmable ROMs (EPROMS) can be erased by exposure to UV light and reprogrammed afterwards. The electrically erasable programmable read only memory (EEPROM) can be erased electronically.

The most commonly used ROMs currently available are based on this technology. There are two categories of EEPROMs: flash EEPROM (commonly known as flash memory) and byte-erasable EEPROM (simply called EEPROM). The main difference between these two types is that flash can be erased and reprogrammed either completely or in a block-wise manner while the byte-erasable EEPROM, as its name suggests, allows the erasing and reprogramming of individual bytes. Flash memories are typically employed for holding the embedded software as it eases the development process due to its easy reprogrammable mechanism. EEPROMs are usually used for storing data such as constants or system configuration and calibration data.

8.1.2 Memory mapping

Memory content is accessed through addressing requiring the association of addresses to memory locations. Memories available on a microcontroller are all mapped in an address space the size of which is given by the addressing capabilities of the CPU, i.e. the width of the address bus. A part of this address space is reserved for registers and the interrupt vector.

This apparently limits the amount of memory that can be used on certain configuration. For example in a 16 bit addressing space we can address 0x10000 (65536) memory locations (bytes). However, there is a solution to alleviate this problem which implies using a memory paging or banking scheme. This means that certain memory sections will act as windows in which at a given time we can access one portion of a larger memory area. If for example we would need to work with a 256Kbyte flash memory using a single 16Kbyte page we will only be able to access 16Kbytes from the entire flash memory at a time. For accessing another part of the flash memory a page/bank change operation is needed. Even though it is functional, this mechanism adds overheads that should be avoided if possible by choosing a chip with a larger address space.

8.2 THE S12 MEMORY SYSTEM

The S12 family members come with specific memory configurations, each having a specified amount of RAM, Flash and EEPROM. These memories can be accessed from the S12 address space in which they are assigned a default position. Figure 8.1 illustrates the memory maps of the three S12 microcontrollers included in the ZK-S12 development kit. This shows that the memory organization is very similar on all S12 devices, the variations being given only by the register count and memory sizes.

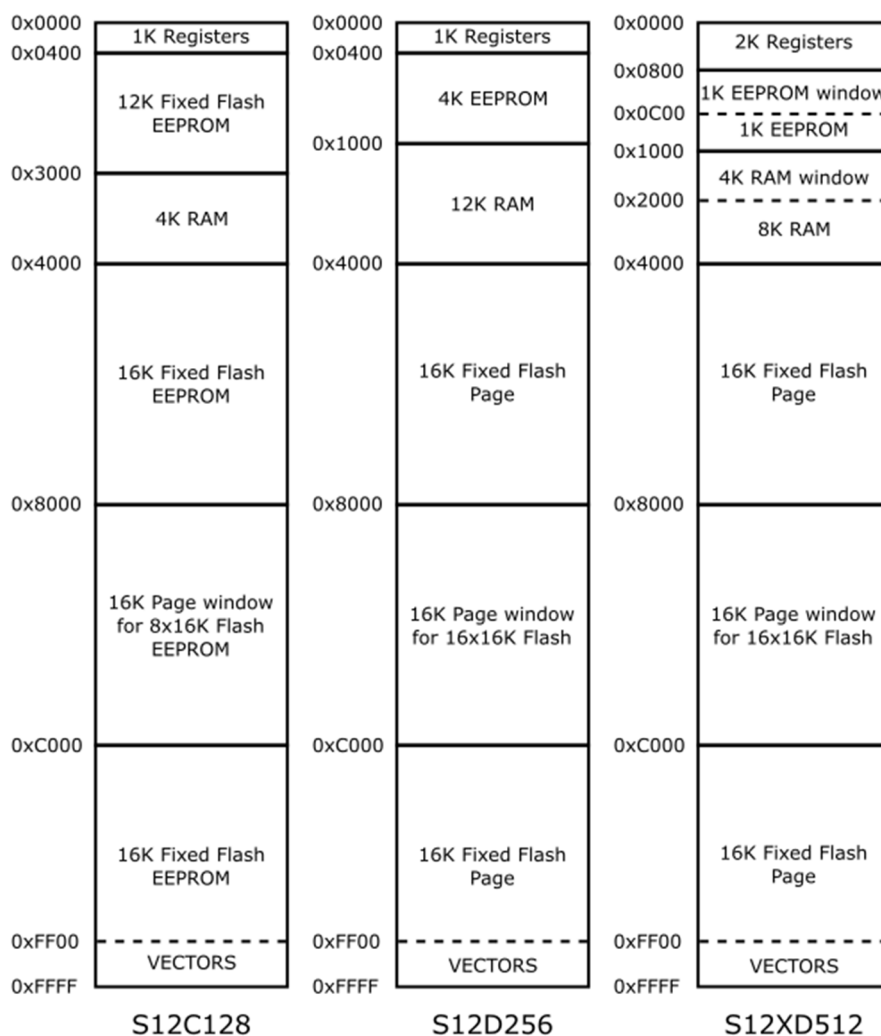


Figure 8.1 Memory map of various S12 microcontrollers according to [3], [18] and [5]

Due to the limited addressable memory space (16-bit address bus) some S12 family members use memory paging schemes for being able to address the entire memory available. This applies to all memory types available: RAM, Flash and EEPROM. The following presentation of the S12 memory system will be based on the S12DT256 and can be used as a reference for other S12 chips. For specifics on other family members please consult the corresponding datasheets and user manuals.

8.2.1 S12 Flash memory

The amount of Flash memory available on each S12 family member can vary in size between 32kB and 1MB. Memories larger than 64kB are divided in two or more blocks 64kB in size (4 such blocks in the case of 256K flash) which can be read, erased or programmed concurrently. The entire Flash memory is organized into 16kB pages as illustrated in Figure 8.2. The last two pages in Block 0 will be directly mapped to fixed memory locations. Page 0x3E will be directly accessible in the 0x4000-0x7FFF interval while page 0x3F will always be directly accessible at 0xC000-0xFFFF. All the pages can selected

to be mapped in the 0x8000-0xBFFF interval by setting the PPAGE register. This includes pages 0x3E and 0x3F even though this might seem redundant.

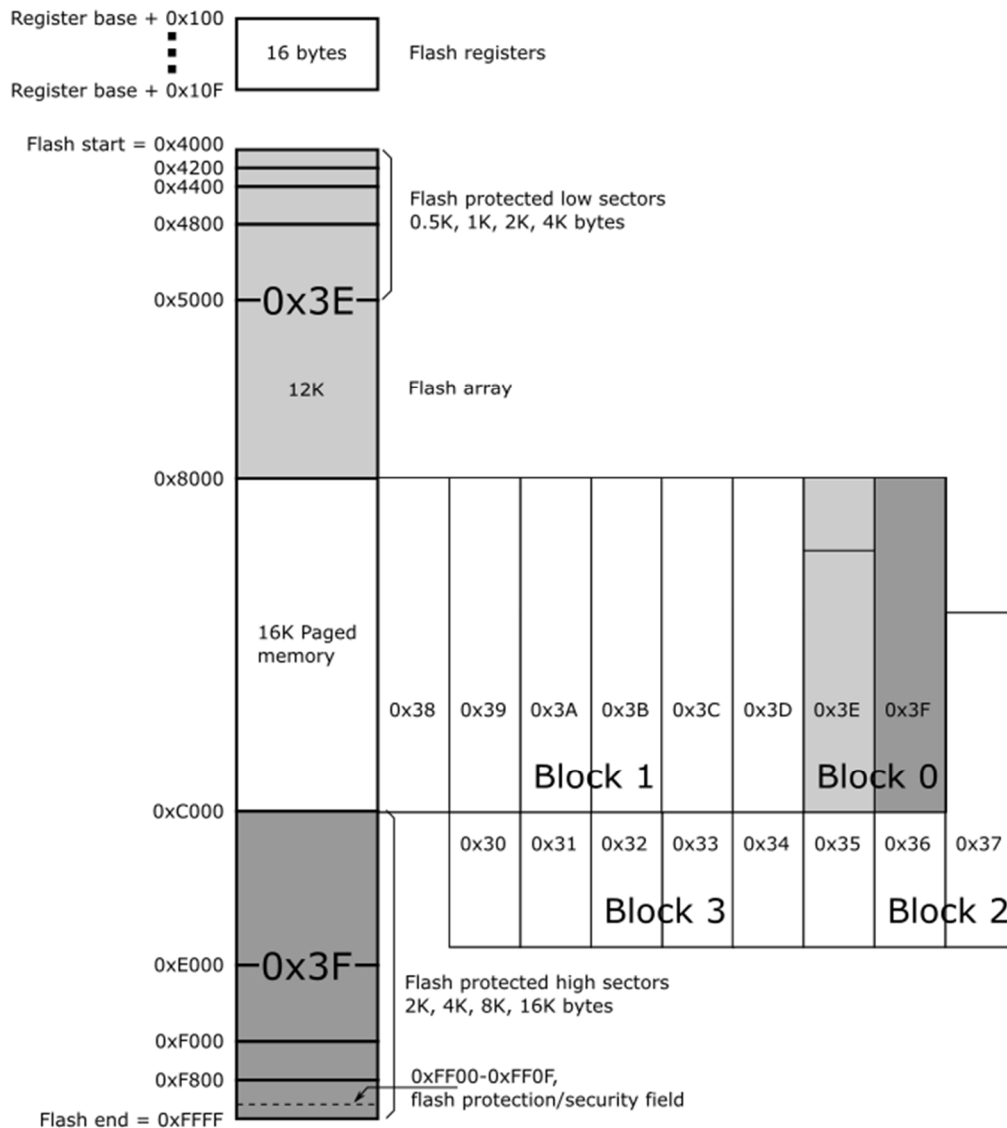


Figure 8.2 S12D256 Flash memory layout according to [18]

As for all previously described modules, a set of register dedicated for the configuration of the flash memory module is provided. These registers are described in what follows.

The Flash Clock Divider (FCLKDIV) register is dedicated for the configuration of timings in the programming and erasing process.

FCLKDIV

7	6	5	4	3	2	1	0
FDIVLD	PRDIV8	FDIV5	FDIV4	FDIV3	FDIV2	FDIV1	FDIV0

- **FDIVLD (Clock Divider Loaded)** – when this flag is set to 1 it indicates that the FCLKDIV register has been written to since the last reset. This bit is read-only;
- **PRDIV8 (Enable Prescaler by 8)** – enables (when set to 1) a prescaling by 8 of the flash module input clock before feeding it into the CLKDIV divider;
- **FDIVx (Clock Divider Bus)** – The flash module clock will be divided by (1 + FDIV). In conjunction with PRDIV8 this field is used to divide the flash module input clock to a frequency of 150-

200kHz with a maximum divide ratio of 512. For more details on the sequence to be followed when setting these fields refer to [18].

The security settings of the flash module are indicated by the FSEC (Flash Security) register. All bits of this register are read-only.

FSEC

7	6	5	4	3	2	1	0
KEYEN1	KEYEN0	NV5	NV4	NV3	NV2	SEC1	SEC0

- **KEYENx (Backdoor Key Security Enable)** – set the bits of this field to ‘10’ to enable the backdoor key access to the flash module. All other values (‘00’, ‘01’, ‘11’) will disable this option;
- **NVx (Non-Volatile Flag)** – these bits are available as non-volatile flags;
- **SECx (Flash Security)** – when the two bits of this field are set to ‘10’ the state is set to unsecured. All other bit combinations (‘00’, ‘01’, ‘11’) will secure the chip. After a backdoor key access, the flash module is unsecured and the value of this field is automatically set to ‘10’.

The FTSMOD (Flash Test Mode) register is used to control Flash special modes.

FTSMOD

7	6	5	4	3	2	1	0
-	-	-	WRALL	-	-	-	-

- **WRALL (Write to All)** – setting this bit to 1 enables the simultaneous writing of all banked registers.

Other configuration options for the flash module are available in the FCNFG (Flash Configuration) register.

FCNFG

7	6	5	4	3	2	1	0
CBEIE	CCIE	KEYACC	-	-	-	BKSEL1	BKSEL0

- **CBEIE (Command Buffer Empty Interrupt Enable)** – when set to 1 this bit enables interrupts in case of empty command buffers in the Flash module;
- **CCIE (Command Complete Interrupt Enable)** – this bit should be set to 1 to enable interrupts when all commands are being completed in the flash module;
- **KEYACC (Enable Security Key Writing)** – set this bit to 1 when interpret writes to the flash array as keys for opening the backdoor. When this bit is 0 flash writes are interpreted as the start of program or an erase sequence
- **BKSELx (Register Bank Select)** – these bits are used to select the available register bank. Table 8.1 shows the bit combinations needed to select each of the 4 banks of a 256KB Flash memory. Bank 0 is selected by default.

BKSEL[1:0]	Selected Register Bank
00	Flash 0
01	Flash 1
10	Flash 2
11	Flash 3

Table 8.1 Flash register bank selection

The FPROT (Flash Protection) register is a banked register that specifies which sectors are protected against programming or erase. Each Flash block has its own FRPOT register all sharing the same access address. The accessible protection register is selected by setting the bank bits. The default accessible protection register is the one for Block 0. The upper sector of Flash has to be unprotected to change the Flash protection settings loaded on reset.

FPROT

7	6	5	4	3	2	1	0
FPOPEN	NV6	FPHDIS	FPHS1	FPHS0	FPLDIS	FPLS1	FPLS0

- **FPOPEN (Open Flash for Program or Erase)** – writing this bit to 1 removes protection and enables programming or erasing on the flash sectors. When this bit is set to 0 protection is enabled and all the other register field are ignored;
- **NV6 (Non Volatile Flag)** – this read only bit is used for non-volatile flags;
- **FPHDIS (Flash Protection Higher address range Disable)** – a 0 in this bit activates the protection on the higher space of the flash address map (0xC000-0xFFFF);
- **FPHSx (Flash Protection Higher address Size)** – these bits determine the size of the protected sector in the higher Flash address space according to Table 8.2 also illustrated in Figure 8.2.

FPHS	Address range	Size
00	0xF800-0xFFFF	2K bytes
01	0xF000-0xFFFF	4K bytes
10	0xE000-0xFFFF	8K bytes
11	0xC000-0xFFFF	16K bytes

Table 8.2 Setting flash protection for the higher address range

- **FPLDIS (Flash Protection Lower address range Disable)** - a 0 in this bit activates the protection on the lower space of the flash address map (0x4000-0x7FFF);
- **FPLSx (Flash Protection Lower address Size)** – these bits determine the size of the protected sector in the lower Flash address space according to Table 8.3 also illustrated in Figure 8.2.

FPHS	Address range	Size
00	0x4000-0x41FF	512 bytes
01	0x4000-0x43FF	1K bytes
10	0x4000-0x47FF	2K bytes
11	0x4000-0x4FFF	4K bytes

Table 8.3 Setting flash protection for the lower address range

The command status of the Flash state machine and flash array access, protection, and bank verify status are defined by the FSTAT (Flash status) banked register.

FSTAT

7	6	5	4	3	2	1	0
CBEIF	CCIF	PVIOL	ACCERR	-	BLANK	-	-

- **CBEIF (Command Buffer Empty Interrupt Flag)** – reading 1 from this bit indicates that the buffers are ready for a new command while a 0 indicates that the buffers are full. This flag is cleared by writing the CBEIF bit with 1;
- **CCIF (Command Complete Interrupt Flag)** – this flag indicates whether a command is in progress (value is 0) or all previous commands are completed (value is 1). Writing this flag has no effect as it will be cleared automatically when CBEIF is cleared;

- **PVIOL (Protection Violation)** – this flag is set to 1 by an attempt to write or to program a protected Flash memory area. This flag is cleared by writing 1 the PVIOL bit;
- **ACCERR (Flash Access Error)** – when read as 1 this flag indicates an illegal access to the selected Flash block and is cleared by writing it to 1;
- **BLANK (Array verified as erased)** – this flag indicates that an erase verify command has checked a Flash blocked and found that it was erased. If after issuing an erase verify command, the BLANK bit is 0 and the CCIF bit indicates a completed command the Flash block is not erased. Writing to this bit has no effect.

The Flash commands are given by writing the FCMD (Flash Command) register. This register is also banked so commands written to it will only affect the associated Flash block.

FCMD

7	6	5	4	3	2	1	0
-	CMDB6	CMDB5	-	-	CMDB2	-	CMDB0

Valid commands are formed by setting the CMDBx bits according to Table 8.4. Writing any other commands other than these will cause an access error and the setting of the ACCERR flag.

Command	Meaning
0x05	Erase verify
0x20	Word program
0x40	Sector erase
0x41	Mass erase

Table 8.4 Valid Flash commands

Addressing of the Flash memory locations on which the commands are performed is done by using the 16-bit **FADDR (Flash Address)** banked register. FADDR consists of two 8-bit registers: FADDRHI and FADDRLO. Bit 15 of FADDR (bit 7 of FADDRHI) is tied to 0. In normal modes reading the FADDR returns zero.

Data to be written with Flash commands is written in the 16-bit **FDATA (Flash Data)** register. FDATA is also a banked register composed of two 8-bit registers: FDATAHI and FDATALO. In normal modes FDATA is not writable and reads to zero. In special modes, FDATA is readable and writable when writing to an address in Flash.

EXAMPLE 8.1 Write a function that receives a flash address as a parameter and erases the Flash sector containing it.

Solution: The function implementing the erase should follow the operation sequence described next. First the CBEIF flag should be tested to ensure that the address, data and command buffers are empty. If they are, write an address inside the sector to be erased in the FADDR register and then write the sector erase command (0x40) to the FCMD register. To start the command execution write the CBEIF bit to 1. To ensure no errors have occurred check the ACCERR and PVIOL bits. The CCIF flag will indicate the end of the command execution. The following code snippet illustrates the C implementation of the sector erase function.

```

int EraseSector(unsigned int addr)
{
    if (!(FSTAT & CBEIF))
        return 1; /* Command buffer not empty, return error */

    FADDR = addr; /* Write address location from the sector */
    FCMD = 0x40; /* Write sector erase command */
    FSTAT = CBEIF; /* Clear CBEIF to start the erase command */

    /* Check if command was successfully issued*/
    if (FSTAT & (ACCERR | PVIOL))
        return 1; /* Access error or protection violation detected, return error */

    while(!(FSTAT & CCIF)); /* Wait until the erase command is completed */

    return 0; /* Return success */
}

```

8.2.2 S12 EEPROM memory

The amount of EEPROM available on S12 chips also varies from none to 8KB. The EEPROM is organized as an array of 2 byte words. The 4K EEPROM of the S12D256 chip has 2048 rows. The size of the erase sector is 2 words or 2 rows (4 bytes). Erased bits will be read as 1 while a programmed bit reads 0. The operations that can be performed on the EEPROM are similar to the ones available for Flash. Figure 8.3 shows the memory layout of the on-chip EEPROM of the S12D256 microcontroller, for other family derivative refer to their corresponding datasheets or user manuals.

Registers associated with the EEPROM module offer similar functionalities as the ones provided for the Flash module. The available EEPROM registers are presented next.

The EEPROM Clock Divider (ECLKDIV) register is dedicated for the configuration of timings in the EEPROM program and erase process.

ECLKDIV

7	6	5	4	3	2	1	0
EDIVLD	PRDIV8	EDIV5	EDIV4	EDIV3	EDIV2	EDIV1	EDIV0

- **EDIVLD (Clock Divider Loaded)** – when this flag is set to 1 it indicates that the ECLKDIV register has been written to since the last reset. This bit is read-only;
- **PRDIV8 (Enable Prescaler by 8)** – enables (when set to 1) a prescaling by 8 of the EEPROM module input clock before feeding it into the ECLKDIV divider;
- **EDIVx (Clock Bivider Bus)** – The EEPROM module clock will be divided by (1 + EDIV). In conjunction with PRDIV8 this field is used to divide the EEPROM module input clock to a frequency of 150-200kHz with a maximum divide ratio of 512. For more details on the sequence to be followed when setting these fields refer to [19].

The ECNFG (EEPROM Configuration) register provides additional interrupt configuration options.

ECNFG

7	6	5	4	3	2	1	0
CBEIE	CCIE	-	-	-	-	-	-

- **CBEIE (Command Buffer Empty Interrupt Enable)** – when set to 1 this bit enables interrupts in case of empty command buffers in the EEPROM module;
- **CCIE (Command Complete Interrupt Enable)** – this bit should be set to 1 to enable interrupts when all commands are being completed in the EEPROM module.

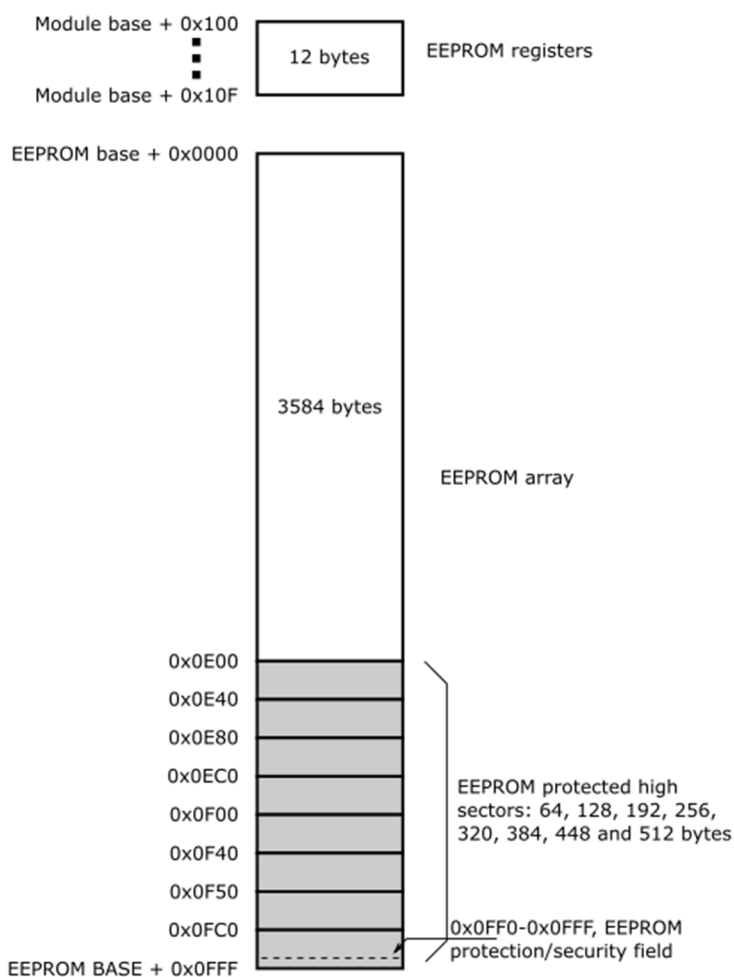


Figure 8.3 S12D256 4K EEPROM memory layout according to [19]

The EPROT (EEPROM Protection) register specifies which sectors are protected against program or erase. The upper sector of EEPROM has to be unprotected and the EEPROM protect byte located at address 0x0FFD must be written to for changing the EEPROM protection settings loaded on reset.

EPROT

7	6	5	4	3	2	1	0
EPOPEN	NV6	NV5	NV4	EPDIS	EP2	EP1	EPO

- **EPOPEN (Open EEPROM for Program or Erase)** – writing this bit to 1 removes protection and enables programming or erasing on the EEPROM sector. When this bit is set to 0 protection is enabled and all the other register field are ignored;
- **NVx (Non Volatile Flags)** – these read only bits are used for non-volatile flags;
- **EPDIS (EEPROM Protection address range Disable)** – a 0 in this bit activates the protection on higher address space of the EEPROM address map (0x0E00-0x0FFF);
- **EPx (EEPROM Protection address Size)** – these bits determine the size of the protected sector in the higher EEPROM address space according to Table 8.5 also illustrated in Figure 8.3.

EP	Address range	Size
000	0x0FC0-0x0FFF	64 bytes
001	0x0F80-0x0FFF	128 bytes
010	0x0F40-0x0FFF	192 bytes
011	0x0F00-0x0FFF	256 bytes
100	0x0EC0-0x0FFF	320 bytes
101	0x0E80-0x0FFF	384 bytes
110	0x0E40-0x0FFF	448 bytes
111	0x0E00-0x0FFF	512 bytes

Table 8.5 Setting EEPROM protected address range

The command status of the EEPROM state machine and EEPROM array access, protection, and bank verify status are defined by the ESTAT (EEPROM status) register. Some bytes of this register (marked with a grey background) are only available in special mode

ESTAT

7	6	5	4	3	2	1	0
CBEIF	CCIF	PVIOL	ACCERR	-	BLANK	FAIL	DONE

- **CBEIF (Command Buffer Empty Interrupt Flag)** – reading 1 from this bit indicates that the buffers are ready for a new command while a 0 indicates that the buffers are full. This flag is cleared by writing the CBEIF bit with 1;
- **CCIF (Command Complete Interrupt Flag)** – this flag indicates whether a command is in progress (value is 0) or all previous commands are completed (value is 1). Writing this flag has no effect as it will be cleared automatically when CBEIF is cleared;
- **PVIOL (Protection Violation)** – this flag is set to 1 by an attempt to write or to program a protected EEPROM memory area. This flag is cleared by writing 1 the PVIOL bit;
- **ACCERR (Flash Access Error)** – when read as 1 this flag indicates an illegal access to the selected EEPROM block and is cleared by writing it to 1;
- **BLANK (Array verified as erased)** – this flag indicates that an erase verify command has checked the EEPROM array and found that it was erased. If after issuing an erase verify command, the BLANK bit is 0 and the CCIF bit indicates a completed command the EEPROM block is not erased. Writing to this bit has no effect;
- **FAIL (Failed EEPROM operation Flag)** – this flag is 1 when an EEPROM erase verify operation has failed. Writing 1 in this bit erases the flag. This bit is available only in special mode;
- **DONE (Complete EEPROM operation flag)** – when this flag is 1 it indicates that an EEPROM operation is completed. This bit is only available in special mode and writing to it has no effect.

The EEPROM commands are given by writing the ECMD (EEPROM Command) register.

ECMD

7	6	5	4	3	2	1	0
-	CMDB6	CMDB5	-	-	CMDB2	-	CMDB0

Command	Meaning
0x05	Erase verify
0x20	Word program
0x40	Sector erase
0x41	Mass erase
0x60	Sector modify

Table 8.6 Valid EEPROM commands

Valid commands are formed by setting the CMDbX bits according to Table 8.6. Writing any other commands other than these will cause an access error and the setting of the ACCERR flag.

Addressing of the EEPROM memory locations on which the commands are performed is done by using the 16-bit **EADDR (EEPROM Address)** register. EADDR consists of two 8-bit registers: EADDRHI and EADDRLO. Bits 15 through 10 of EADDR (bits 7-3 of EADDRHI) are tied to 0. In normal modes reading the EADDR returns zero.

Data to be written with EEPROM commands is written in the 16-bit **EDATA (EEPROM Data)** register. EDATA is also composed of two 8-bit registers: EDATAHI and EDATALO. In normal modes EDATA is not writable and reads to zero. In special modes, EDATA is readable and writable when writing to an address in EEPROM.

8.2.3 Remapping memory sections

The default mapping of the register space, RAM, Flash and EEPROM may be changed by using module mapping registers of the S12 chip. The register block, RAM and EEPROM can be assigned to different memory locations by using the INITRG, INITRM and INITEE registers respectively [20]. It is considered good practice to make this configuration during the program initialization phase. If conflicts occur due to overlapping module mapping the allocation will be done according to the priorities in Table 8.7.

The INITRG register initializes the position of the internal register within the available address space. The registers can occupy 1 or 2 K byte of space depending on the S12 derivative and can be mapped to any 2KB space within the first 32KB.

INITRG

7	6	5	4	3	2	1	0
-	REG14	REG13	REG12	REG11	-	-	-

- **REG1x (Internal Register map position)** – these bits preceded by a 0 give the upper 5 bits of the base address for the system registers.

Priority	Resource
1	BDM firmware or register space
2	Internal register space
3	RAM memory block
4	EEPROM memory block
5	Flash memory block
6	Remaining external space

Table 8.7 Memory space allocation priorities

The INITRM register initializes the position of the internal RAM memory inside the system memory map.

INITRG

7	6	5	4	3	2	1	0
RAM15	RAM14	RAM13	RAM12	RAM11	-	-	RAMHAL

- **RAM1x (Internal RAM map position)** – these bits give the upper 5 bits of the base address for the RAM;
- **RAMHAL (RAM High Align)** – this bit sets the RAM alignment. Setting it to 0 aligns the RAM to the lowest address of the mapable space, while setting it to 1 aligns it to the highest address.

The INITEE register initializes the position of the on-chip EEPROM within the system memory map.

INITEE

7	6	5	4	3	2	1	0
EE15	EE14	EE13	EE12	EE11	-	-	EEON

- **EE1x (EEPROM map position)** – these bits give the upper 5 bits of the base address for the EEPROM;
- **EEON (Enable EEPROM)** – this bit enables the RAM in the memory map. Setting it to 0 disables EEPROM from the memory map, while setting it to 1 enables it.

As previously mentioned the on-chip Flash memory that cannot be directly addressed due to its size is accessed through the 16K byte page window. The PPAGE (Program Page Index) register is used to select which flash block to be loaded in this window.

PPAGE

7	6	5	4	3	2	1	0
-	-	PIX5	PIX4	PIX3	PIX2	PIX1	PIX0

- **PIXx (Program Page Index)** – these bits give the number of the 16K Flash page to be loaded in the page window.

EXAMPLE 8.2 Select Flash page 3 to be accessible in the flash page window.

Solution: The PIX bits of the PPAGE register have to be set to point at Flash page 3. Therefore the PPAGE register has to be written with the value 0x03. The following code snippet give the solution.

```
PPAGE = 0x03; // set flash page register value
```

EXERCISE 8.1 Write functions that provides an easy to use interface to all Flash operations.

EXERCISE 8.2 Write functions that provides an easy to use interface to all EEPROM operations.

EXERCISE 8.3 Write a program that remaps the internal registers, RAM and EEPROM so that they start at the same base address. Read values from that address space and explain the content. Choose different addresses and repeat the experiment.

9 REFERENCES

- [1] Han-Way Huang, *The HCS12/9S12: An Introduction to Software and Hardware Interfacing*, Cengage Learning, 2009.
- [2] Daniel J. Pack and Steven F. Barrett, *Microcontroller theory and applications: HC12 and S12*, Prentice Hall Press, 2007.
- [3] Freescale Semiconductor, *MC9S12C, MC9S12GC Family Reference Manual*, MC9S12C128, Rev. 01.24, May 2010, http://www.nxp.com/files/microcontrollers/doc/data_sheet/MC9S12C128V1.pdf
- [4] Freescale Semiconductor, *MC9S12DT256 Data Sheet and Reference Manual*, 08 Jul 2010, cache.freescale.com/files/microcontrollers/doc/data_sheet/9S12DT256_ZIP.zip
- [5] Freescale Semiconductor, *MC9S12XDP512 Data Sheet*, MC9S12XDP512RMV2, Rev. 2.21, Oct 2009, http://www.nxp.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf
- [6] SofTec Microsystems, *ZK-S12-B Starter Kit for Freescale HCS12(X) Family (80-Pin QFP ZIF Socket) – User’s Manual*, 2005.
- [7] SofTec Microsystems, *ZK-S12-B Schematic and Bill of Material*, 2005.
- [8] Freescale Semiconductor, *HCS12 V1.5 Core user guide, Version 1.2*, Original Release Date: 12 May 2000, Revised: 17 August 2000.
- [9] Paul Atkinson, *AN2434, Input/Output (I/O) Pin Drivers on HCS12 Family MCUs*, Revised September 2004, Freescale Semiconductor, Inc., 2004.
- [10] Freescale Semiconductor, Inc., *S12 CPU Reference Manual*, April, 2002, Revised: March 2006.
- [11] Motorola Inc., *HCS12 Microcontrollers Interrupt (INT) Module V1*, 01 May 2003.
- [12] Motorola Inc., *CRG Block User Guide V02.07*, Original Release Date: 29 Feb 2000, Revised: 11 Mar 2002.
- [13] Motorola Inc., *TIM_16B8C Block User Guide*, Original Release Date: 28 Jul 2000, Revised: 11 Oct 2001.
- [14] Freescale Semiconductor Inc., *ECT_16B8C Block User Guide V01.06*, Original Release Date: 2-Sep-1999, Revised: 30-Apr-2010
- [15] Freescale Semiconductor Inc., *PWM_8B8C Block User Guide V01.17*, Aug, 2004.
- [16] Motorola, Inc., *ADC Block User Guide, Version V02.10*, DOCUMENT NUMBER S12ATD10B8CV2/D, Original Release Date: 27 OCT 2000, Revised: 21 Feb. 2003.
- [17] *Martyn Gallop, AN2428/D, An Overview of the HCS12 ATD*, Revised: January 2003, Freescale Semiconductor, Inc., 2004

[18] Motorola Inc., *FTS256K Block User Guide V03.01*, Original Release Date: 8-Feb-2001, Revised: 8-Apr-2003.

[19] Freescale Semiconductor, *EETS4K Block User Guide V02.08*

[20] Motorola Inc. *HCS12 Microcontrollers, Module Mapping Control (MMC) V4*, February, 2003.