MOTOROLA
intelligence everywhere™

digital dna™

8-Bit Software Development Kit for Motor Control Targeting M68HC08 Applications

User's Guide

M68HC08
Microcontrollers

SDKHC08AUG/D
7/2002

WWW.MOTOROLA.COM/SEMICONDUCTORS

8-Bit Software Development Kit for Motor Control Targeting M68HC08 Applications

User's Guide

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.motorola.com/semiconductors/

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Revision History

Date	Revision Level	Description	Page Number(s)
July, 2002	N/A	Original release	N/A

List of Sections

List of Sections

Table of Contents

Section 1. General Description

Section 2. Directory Structure

Section 3. Developing Software

Section 4. Core System Infrastructure

Section 5. On-Chip Drivers

Table of Contents

Section 6. Off-Chip Drivers

List of Figures and Tables

List of Figures and Tables

Section 1. General Description

1.1 Contents

1.2 Introduction

This user's guide is targeted for Motorola M68HC08 (HC08) application developers. Its purpose is to describe the development environment, the software modules, and the tools for the HC08 and the application programming interface (API). Simply, this manual describes how to use the Motorola HC08 SDK to develop software for the Motorola MCU processor.

1.3 Overview

The HC08 SDK development environment provides fully debugged peripheral drivers, examples, and interfaces that allow programmers to create their own C application code, independent of the core architecture. This environment has been developed to complement the existing development environment for Motorola 68HCMR32 processors. It provides a software infrastructure allowing development of efficient, ready to use, high-level software applications which are fully portable and reusable between different core architectures. Maximum portability is achieved for devices with comparable on-chip peripheral modules.

This manual only contains information specific to the HC08 SDK as it applies to the Motorola HC08 software development. Therefore, it is required that users of the HC08 SDK should be familiar with the HC08 Family in general, as is described in the *MC68HC908MR32/ MC68HC908MR16 Advance Information* (Motorola document order number MC68HC908MR32/D).

The Motorola HC08 SDK is designed for and is fully integrated with Metrowerks[1] CodeWarrior[2] development tools. Before starting to explore the full feature set of the HC08 SDK, one should install and become familiar with the CodeWarrior development environment.

All together, the HC08 SDK, the CodeWarrior, and the evaluation modules (EVMs) create a complete and scalable tool solution for easy, fast and efficient development.

1.3.1 Features

The HC08 SDK environment is composed of the following major components:

- Core system infrastructure
- On-chip drivers with defined API
- Sample example applications
- Off-chip drivers

This section provides very illustrative information about these components, while comprehensive descriptions can be found in specially targeted sections.

1.3.1.1 Core-System Infrastructure

The core-system infrastructure creates the fundamental infrastructure for the MC68HC908MR32 device operation and enables further integration with other components, e.g., on-chip drivers. The provided basic development support includes:

- Commonly used macro definitions
- Portable architecture-dependent register declaration
- Mechanism for static configuration of on-chip peripherals as well as interrupt vectors
- Project templates

1. Metrowerks® and the Metrowerks logo are registered trademarks of Metrowerks, Inc., a wholly owned subsidiary of Motorola, Inc.
2. CodeWarrior® is a registered trademark of Metrowerks, Inc., a wholly owned subsidiary of Motorola, Inc.

1.3.1.2 On-Chip Drivers

The on-chip drivers isolate the hardware-specific functionality into a set of driver commands with a defined API. The API standardizes the interface between the software and the hardware, see **Figure 1-1**. This isolation enables a high degree of portability or architectural and hardware independence for application code. This is mainly valid for devices with similar peripheral modules. The driver code reuses lead for greater efficiency and performance.

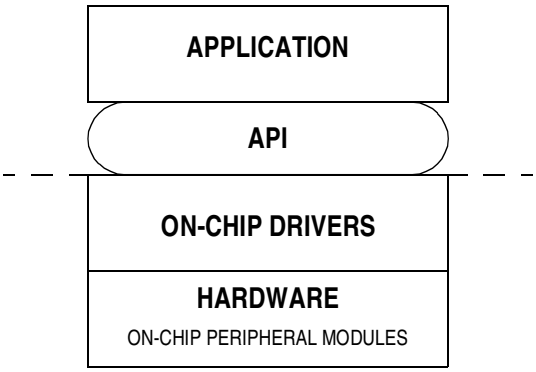

Figure 1-1. Software Structure

1.3.1.3 Off-Chip Drivers

Off-chip drivers isolate the hardware-specific functionality into a set of driver commands with a defined API. The API standardizes the interface between the software and the hardware, see **Figure 1-2**. This isolation enables a high degree of portability or architectural and hardware independence for both application and off-chip drivers code. This is mainly valid for standard external periphery such as: PC, display, keyboard, switches, light emitting diode (LED), etc. The driver code reuses leads for greater efficiency and performance.

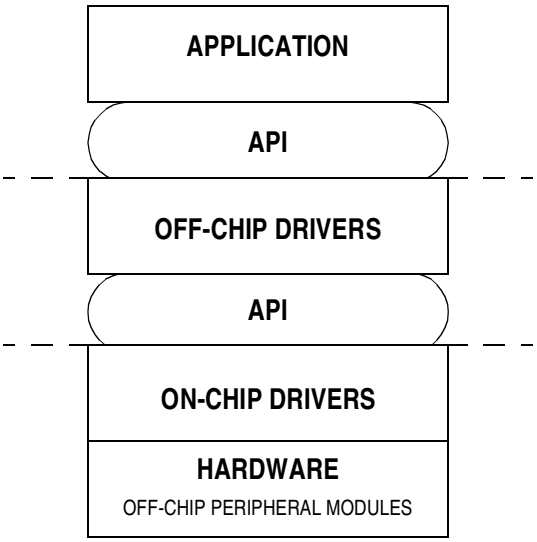

Figure 1-2. Software Structure

1.3.1.4 Sample Applications

The HC08 SDK contains a number of sample applications demonstrating how to use on-chip drivers and how to implement some user-specific tasks. These sample examples are kept simple, illustrative and their intention is to minimize the learning curve.

1.3.1.5 PC Master Software

PC master software is one of the off-chip drivers, supporting communication with a PC. This tool was initially created for developers of motor control applications, but it may be extended to any other application developments. This tool allows remote control of an application using a user-friendly graphical environment running on a PC. It also provides the ability to view some real-time application variables in both textual and graphical form.

Main features:

- Graphical environment
- Visual Basic Script or Java[1] Script can be used for control of target boards
- Easy to understand navigation
- Connection to target board possible over a network, including the Internet
- Demonstration mode with password protection support
- Visualization of real-time data in the Scope window
- Acquisition of fast data changes using integrated recorder
- Value interpretation using custom defined text messages
- Built-in support for standard variable types (i.e., integer, floating point, bit fields)
- Several built-in transformations for real type variables
- Automatic variable extraction from Metrowerks' CodeWarrior linker output files (MAP, ELF)
- Remote control of application execution

PC master software is a versatile tool to be used for multipurpose algorithms and applications. It provides a lot of excellent features, including:

- Real-time debugging
- Diagnostic tool
- Demonstration tool
- Education tool

The full description can be found in the *PC Master Software User Manual* attached to the PC master software tool.

1. Java™ is a trademark of Sun Microsystems, Inc., in the United States and other countries.

1.4 Quick Start

This subsection provides information needed to install the HC08 SDK and get it running.

1.4.1 Installing CodeWarrior Development Tools

CodeWarrior for Motorola MCU Embedded Systems provides a complete software development environment for Motorola general purpose HC08 processors. CodeWarrior for Motorola MCU is a windows based integrated development environment (IDE) with a highly efficient C compiler.

As previously mentioned, Motorola HC08 SDK is designed for and fully integrated with Metrowerks CodeWarrior development tools. With CodeWarrior tools, users can build applications and integrate other software included as part of the HC08 SDK release. Once the software is built, CodeWarrior tools allow users to download executable images into the target platform and run/debug downloaded code.

The installation process of CodeWarrior is described in the installation guide attached to the actual version of CodeWarrior.

1.4.2 Installing HC08 SDK

In order for HC08 SDK to integrate itself with the development tools, the CodeWarrior tools should be installed prior to the installation of HC08 SDK (see **1.4.1 Installing CodeWarrior Development Tools**). If HC08 SDK is installed while CodeWarrior is not present, users can only browse the install software package, but will not be able to build, download, and run the released code. However, installation of SDK can be simply completed once CodeWarrior is installed, see **1.4.2.1 Supplementary HC08 SDK Installation Steps**.

The installation of SDK itself consists of copying the needed files to the destination hard drive, checking the presence of CodeWarrior and creating the shortcut under Start->Programs menu.

NOTE: *Each HC08 SDK release can be installed in its own new directory named HC08 SDK_rX.X (where X.X denotes the release number). Thus, it allows you to maintain the older releases and projects. It gives free choice to select the active release.*

To start the installation process, perform the following steps:

1. Execute Setup.exe
2. Follow the HC08 SDK software installation instructions on your screen.

To integrate HC08 SDK with CodeWarrior, perform the following steps:

1. Set path to HC08 SDK source within CodeWarrior's IDE
2. Launch the CodeWarrior IDE from the Start->Programs->Metrowerks CodeWarrior menu
3. Open *IDE Preferences* dialog window using Edit->Preferences...
4. Select *Source Trees* panel from IDE Preferences Panels-General
5. Type *HC08 SDK src* to the Name box
6. Choose *Absolute Path* as a path type
7. Click *Choose* and locate the HC08 SDK installation directory, e.g., C:\Program Files\Motorola\HC08 SDK\src_mw
8. click *Add*
9. click *OK* to finish

1.4.2.1 Supplementary HC08 SDK Installation Steps

Now that CodeWarrior and HC08 SDK are successfully installed. The next step is to copy the content of the HC08 SDK stationery folder (e.g., ...Motorola\HC08SDK\stationery) into the CodeWarrior stationery folder (e.g., ...Metrowerks\CodeWarrior\Stationery) to "register" the HC08 SDK project templates for the newly created projects.

1.4.2.2 Installing PC Master

System Requirements

The PC master software application can run on any computer with Microsoft(1) Windows(2) operating system with Internet Explorer 4.0 or higher installed. The following requirements are for the Internet Explorer 4.0 application:

Computer: 486DX/66 MHz or higher processor (Intel(3) Pentium(4) recommended)

Operating system: Microsoft Windows + DCOM pack

Memory: Windows 95 or 98: 16 Mb RAM minimum (32 Mb recommended); for Windows NT: 32 Mb RAM minimum (64 Mb recommended)

Required software: Internet Explorer 4.0 or higher installed

Hard drive space: 6 MB

Other hardware requirements: Mouse, serial RS-232 port for local control, network access for remote control

Target Development Board Requirements

PC Master relies on the following to be provided by the target development board:

Interface: Serial communication port (required on all Motorola EVM boards)

1. Microsoft® is a registered trademark of Microsoft Corporation in the U.S. and/or other countries.
2. Windows®, Windows95® and subsequent versions, Windows NT® are registered trademarks of Microsoft Corporation in the U.S. and/or other countries.
3. Intel® is a registered trademark of Intel Corporation.
4. Pentium® is a registered trademark of Intel Corporation.

1.4.3 Required Hardware

The HC08 SDK for HC08MRxx has been designed and tested with the MC68HC908MR32 MC board target hardware. If the user wants to quickly exercise software applications included with the HC08 SDK, MC68HC908MR32 MC board hardware must be installed.

NOTE: *It is recommended that all HC08 SDK users read through this document, before proceeding with software development.*

1.4.4 Building and Running Sample Application

Once the HC08 SDK is installed, the user can build and run any of the released demo applications for the MC68HC908MR32 MC board by opening and building the project, using the CodeWarrior development environment. We will use *pwm_demo.mcp* as an example:

Step 1: Launch CodeWarrior IDE from:
Start->Programs->Metrowerks CodeWarrior menu

Step 2: Using File->Open command, open *pwm_demo.mcp* project located in:
src_mw\68HC908MR32\demos\pwm_demo\ directory

Step 3: Execute *Debug* by pressing the *Ctrl+F5* key or choose the *Debug* command from the *Project* menu.

Step 4: Run the application by pressing the green arrow (Run) in the debug window or choose the *Run* command from the *Project* menu.

At this point, the application is running — the LEDs associated to the pulse-width modulator (PWM) outputs are now flashing and the green LED is blinking periodically.

Subsequent sections describe:

- How to create a new application
- How to use interrupts
- Usage of on-chip drivers
- Other needed information to successfully create a new application

1.5 Rules and Coding Standards

In this section, some programming guidelines are described that apply to all applications and algorithms as well as to all MCU architectures supported by the HC08 SDK, regardless of the application area. Rules are the set of critical software development practices that must be followed to produce software with a high degree of portability and reusability. Software developed with these rules is considered HC08 SDK-compliant, since it conforms to the standards used to develop the HC08 SDK software. Coding standards, on the other hand, are a set of good software development practices that lead to a highly portable, consistent, and reusable end product. These rules and guidelines are the standards by which the HC08 SDK components are developed.

1.5.1 Rules

1.5.1.1 Use of the C Language

All algorithms and applications will follow the calling conventions imposed by the C programming language. This ensures the system integrator is free to use C to “bind” various algorithms together, to control the flow of data between algorithms and applications, and to interact easily with other processors in the system.

It is very important to note this does not mean that software must be written in C language. Software may be implemented entirely in Assembly language. However, it must be callable from the C language and respect the C language calling conventions. Refer to the CodeWarrior Help, Calling Conventions, for details.

Rule 1: All algorithms and applications must follow the calling conventions imposed by the CodeWarrior implementation of the C programming language.

1.5.1.2 Use of Peripherals by Algorithms

To ensure the inter-operability of algorithms, it is important no algorithm ever directly access any peripheral device directly.

Rule 2: Algorithms must never directly access any peripheral device.

1.5.1.3 Use of Peripherals by Applications

To ensure the inter-operability and portability of applications, no application can ever directly access any peripheral device. All data produced or consumed by an application must be explicitly passed by the on-chip driver API.

Rule 3: Application software must never directly access any peripheral device.

1.5.2 Coding Standards

If followed, the guidelines established here will lead to highly portable, consistent, and reusable end product.

1.5.2.1 Naming Conventions

The following subsections discuss the specifics of naming.

General guidelines:

- Every external identifier should begin with the lower-case name of the file that contains it. This helps eliminate naming conflicts and provide for a quick cross-reference mechanism from identifier to definition and implementation.
- Use meaningful names; i, j, k are meaningful and acceptable names for iterators or counters
- Use mixed-case instead of underscores to separate words
- Use all upper case for constants; underscores are permitted for readability

Table 1-1 provides guidance on naming conventions based upon properties of the entity being identified. These prefix characters are used to further identify attributes associated with the type being specified.

Table 1-1. Naming Conventions

Property	Prefix Characters	Property	Prefix Characters
Struc type	s	Typedef type	t
Union type	u	Pointer variable	p
Enum type	e		

Files:

- File names should remain short to facilitate the general naming guidelines

Macros:

- Functions name should begin with the lower cases
- Macros should be upper case

1.5.2.2 Formatting

Guidelines:

- Line Length for comments should have 80 or less characters
- Line Length for code should have 135 or less characters
- Do not rely on window line wrap
- Try to alphabetize methods for easier reference
- Try not to use tabs; however, if you must use tabs, set tabs to four spaces and use tabs only to the left of the first readable character.

1.5.2.3 Entry/Exit

Guidelines:

- Single entry and exit point for assembly language
- C code may have early exits for parameter or state checking
- Multiple exits may not be used for different processing paths

1.5.2.4 Self Modification

Do not use self-modifying code.

1.5.2.5 Source Statements per Line

Do not concatenate multiple source statements on a single line. Instead, use a block comment before a section of code.

1.5.2.6 Arithmetic Calculations

Guidelines:

- Consider issues related to integer length, signed versus unsigned, divide by zero, overflow/underflow, etc.
- Where possible, utilize intrinsics provided by the HC08 SDK libraries

1.5.2.7 Reserve Word Redefinition

Do not redefine reserved words.

1.5.2.8 Recursion

C functions may be used recursively. That is, a function may call itself either directly or indirectly.

- If recursion is used, it must be noted in the header
- Environment issues should be addressed; e.g., stack space
- Terminations must be guaranteed

1.5.2.9 Data Initialization

Guidelines:

- All local variables must be initialized to a valid value prior to their use
- If no other valid value is available, pointers should be initialized to NULL

1.5.2.10 Global Variables

Variables used in the library function should be passed through the header.

1.5.2.11 Use of Parentheses

Use parentheses to avoid ambiguity and improper evaluation in arithmetic and Boolean expressions.

1.5.2.12 GOTO

There are valid reasons to use a GOTO in C code. However, in general, it is used as a lazy alternative to restructuring and a GOTO should be avoided if at all possible.

1.5.2.13 Switch Statements

Switch statements must have a default clause. Each case statement must either have a separate break statement or be combined with another case statement with a separate break statement.

1.5.2.14 Headers

The following sections establish conventions for headers.

File Headers:

Determine a standard file header with a copyright notice to make the code appear more consistent between developers and to provide legal protection for source code. The following is an example of code intended for Motorola internal use only.

Example:

```
/*****************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************
*
* File Name: < source file name >
*
* Description: < Brief description >
*
* Modules Included:
*                           < modulename1 >
*                           < modulename2 >
*
*****************************************************************/
```

Function Headers:

- Module or functions should provide the function name for reference
- The description should describe the intent of the routine and any significant algorithms or complex coding schemes
- Returns should specify value to be returned
- Global data should identify the modification of data or state not local to this routine and how this may affect the execution of the application
- Arguments should identify the parameters and how they are used. i.e. let the code identify the actual types of data
- Range Issues should identify any argument or function range issues
- Special Issues should identify the following:
  - Designer notes specifying considerations for maintainability or implementation choices
  - Dependencies include portability issues (hardware or software), compiler workarounds, emulator nuances, etc.

Example:

```
/*****************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************
*
* Module: yyyy()
*
* Description: < Basic functionality performed by module >
*
* Returns: < Specify return values >
*
* Global Data:
* < define global data modifications and effects on application
      execution>
*
* Arguments: < describe parameters >
*
* Range Issues: < Specify argument or function range issues >
*
* Special Issues: < designer notes, dependencies >
*
*****************************************************************/
```

1.5.2.15 Data Typing

Guidelines:

- Be aware of signed versus unsigned and overflow/underflow when using integers
- Differentiate between handling of character arrays and string data; i.e., proper string termination
- Use the data types described in *types.h* to make your code more portable
- Use ANSI C's `size_t` data type when you need a portable way to hold the size of an entity; e.g., `sizeof` returns `size_t`

1.5.2.16 Portability

Guidelines:

- Rely on ANSI C whenever possible
- Any non-portable (i.e., dependency on OS or compiler specifics) should be noted in the header
- Do not insert computational statements in function calls which are removed on certain platforms, e.g., *printf()* and *assert()*; when these calls are removed or stubbed out, the computational statements may not be executed

1.5.2.17 Macro Usage

Guidelines:

- Non-function-style macros (e.g., #define) should be used to define symbolic constants that help to self-document code rather than hard-coding literals in place
- Function-style macros should take into account the syntactical result of inlining, as well as the proper functioning of the macro itself

1.5.2.18 Re-entrance

Functions that are re-entrant may be entered multiple times concurrently. For example, an application has called a subroutine, and during the execution of this subroutine an interrupt occurs, calling the same subroutine that was interrupted.

- If the function is re-entrant, it must be noted in the header
- Avoid static and non-local variables; e.g., file or global scope

1.5.2.19 Code Comments

Source files shall be commented to provide maintainability for future enhancements or bug fixes. Comments shall detail basic functionality of code sections, especially where implementation may not be obvious to an inexperienced developer.

Section 2. Directory Structure

2.1 Contents

2.2 Introduction

This section describes the directory structure of the Motorola HC08 SDK tool, located in the directory: **<...>\Motorola\HC08SDK**.

NOTE: *The root directory of HC08 SDK may be changed during installation by the user. In general, the HC08 SDK software is organized by supported devices as it is explained here.*

2.3 Root Directory

The root or main directory is organized as shown in **Figure 2-1**.

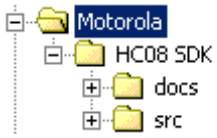

Figure 2-1. Root Directory Structure

Where:

- ***src*** contains the C source files; see also **2.5 Src Directory**
- ***docs*** contains the HC08 SDK User's Manual and other useful documentation

2.4 Applications Directory

This directory contains the large applications to demonstrate the HC08 SDK usage when developing for different hardware platforms. The structure of the ***applications*** directory is illustrated in **Figure 2-2**.

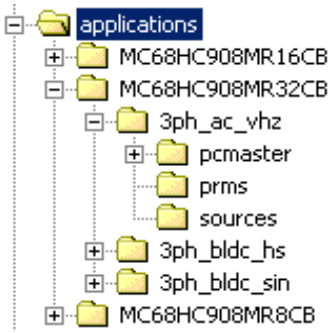

Figure 2-2. Applications Directory Structure

The applications reside on the directory corresponding to the target hardware — Motorola EVM boards.

NOTE: *Individual application directories are further structured with project specific folders which hold the configuration files, the project build files, and the CodeWarrior private data files.*

2.5 Src Directory

A ***src*** or “source” directory is intended to hold all source files. Its structure is shown in **Figure 2-3**. The ***src*** directory is further divided into the following subdirectories:

- ***68HC08MRxx*** is the directory specific for each of the supported devices.
  - Subdirectory ***drivers*** contains the source code for on-chip peripheral drivers and high-level drivers
  - Subdirectory ***system*** contains the device specific source files, the
  - Subdirectory ***config*** contains files for static configuration.
  - Subdirectory ***examples*** contains the sample applications demonstrating the usage of the SDK drivers.
- ***algorithms*** contains distributed and user algorithms
- ***applications*** is the directory containing the large applications targeted for specific EVM boards.
- ***include*** contains the common HC08 SDK header files, which define APIs and the implementation of generally used macros

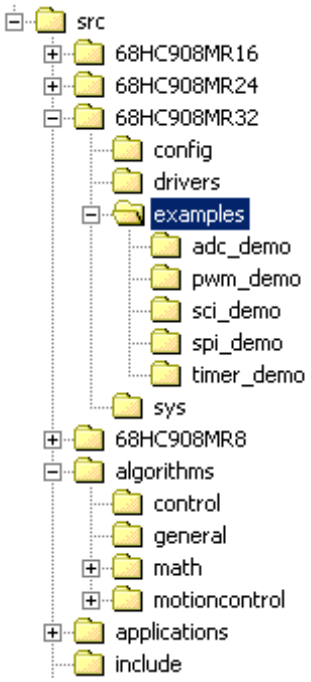

Figure 2-3. Src Directory Structure

2.6 Stationery Directory

The ***stationery*** directory contains the templates for the newly created HC08 SDK projects.

NOTE: *This directory is present in the HC08 SDK source directory only if the Metrowerks development tool is not installed prior to the installation of the HC08 SDK. The proper placement is within the Metrowerks installation directory. If Metrowerks was already installed, this directory will be automatically placed during HC08 SDK installation.*

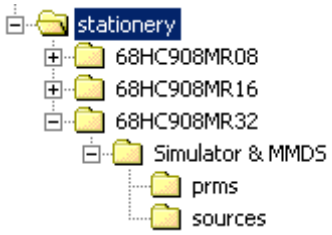

Figure 2-4. Stationery Directory Structure

The device specific subdirectories ***68HC908MRxx*** are covered by the HC08 SDK directory. The device specific subdirectory contains all needed support files for proper memory, system, and application configuration and initialization.

2.7 Docs Directory

The ***docs*** directory contains this HC08 SDK User's Manual as well as other relevant documentation.

Section 3. Developing Software

3.1 Contents

3.2 Introduction

This section describes:

- How to develop applications using 8-bit SDK in detail
- How to create new applications using 8-bit SDK
- How to initialize on-chip peripheral modules
- How to access on-chip peripheral modules in run-time by application code

In addition, an application configuration by an application specific configuration file *appconfig.h* is presented.

At this point, it is assumed CodeWarrior Development Tools and 8-bit SDK have been successfully installed and are running. If you need information regarding installation of these tools, refer to **1.4.1 Installing CodeWarrior Development Tools** and **1.4.2 Installing HC08 SDK**.

3.3 Creating a New Project

8-bit SDK supports two compilers Metrowerks CodeWarior and Cosmic[1] Software. Separate subsections for each compiler follow.

3.3.1 Metrowerks CodeWarrior IDE

To create a new project based on the HC08 SDK project templates (stationery), perform the following steps:

1. Launch CodeWarrior IDE from the **Start->Programs->Metrowerks CodeWarrior** menu.
2. Choose **File->New** command.
3. Click the **Project** tab and select **HC08 SDK Stationery** project type.
4. Type the project name (with a *.mcp* extension) and set the location for the new project.
5. Click **OK**.
6. Select the project stationery from the **New Project** window. This step includes selecting the type of processor (MR32) and the target configuration (Simulator, MMDS or both).
7. Click **OK**.

Upon completing all these actions the project window is displayed. The project window contains these predefined file groups:

- **Dependencies** — contains the following file subgroups:
  - **SDK Configuration** — contains *types.h, appconfig.h, config.c, sys.c, interrupts.c,Start08.c* files, and the target specific linker command files *default.prm*
  - **SDK Drivers** — relevant drivers for chosen chip
  - **SDK Algorithms** — empty by default
  - **MW Lib** — *ansi.lib*

1. Cosmic® is a registered trademark of Cosmic Software Inc. All Rights Reserved

- C Sources — contains an empty *main.c* file to be modified by the user and the peripheral file subgroup which includes all driver source files for an easy start
- Info — contains text files with default static initialization definitions. The definitions can be copy to appconfig.h file.

Now, you can start writing your code in the C source file *main.c* and to configure the on-chip peripherals into the include file *appconfig.h.*

3.3.2 Cosmic Software Idea CPU08

To create a new project based on the HC08 SDK project templates (stationery), perform the following steps:

1. Copy whole *new_app* folder from **src_cosmic\stationery\HC08_SDK\68HC908MR32 to** your application directory.
2. Rename the folder and project file *new_app.prj* to appropriate name.
3. In the new project file **.prj* replace all string *new_app* by your new name.
4. Launch Cosmic Idea CPU08 from the **Start->Programs->Cosmic Tools->Idea 6808** menu.
5. Choose **Project->Load** and find your application project file.
6. In **Setup->Working Directory** set working directory to 8-bit SDK source. E.g.C:\Program_Files\Motorola\8-bit SDK\src_cosmic

Upon completing all these actions the project window is displayed. The project window contains the predefined 8-bit SDK files.

3.4 On-Chip Peripheral Initialization

HC08 SDK provides a very effective mechanism illustrating how to initialize statically all on-chip peripherals. The static configuration of on-chip peripheral is provided by the application specific configuration file *appconfig.h*.

The configuration file *appconfig.h* is used to define the configuration items, which determine the configuration of the on-chip peripheral registers. Defined configuration items are written to peripheral registers in periphery initialization function. The function can be called by the *IOCTL* driver commands xx_INIT (xx is the peripheral prefix used in all *IOCTL* commands) which the user can use somewhere in the initialization code of the application. For example, PWM_INIT for Pulse Width Modulator, etc.

The periphery initialization function is called automatically if the INCLUDE_xxx is defined in *appconfig.h*. For example, #define INCLUDE_PWM. This way saves more because it can find influence between the used peripheral modules.

The step-by-step procedure to statically initialize the on-chip peripheral using the HC08 SDK is:

1. Define configuration items (register values) in the configuration file *appconfig.h*. In the *periphery.txt* file are predefined examples of all constants for specified periphery.
2. Initialize the selected on-chip peripheral by the INCLUDE_xxx in the appconfig or by the xx_INIT *IOCTL* command in your application code.

Tip: If you are editing the configuration file *appconfig.h* manually, you can copy the template of all configuration items intended for the *appconfig.h* file from the peripheral TXT file <*name_of_driver*>*.txt*. For example, *pwmdrv.txt* — Pulse Width Modulation driver TXT file, etc. **Example 1** shows this template for the timer/counter as extracted from the include file *timer.h*.

Example 1. Configuration items for Timer B Extracted from the Driver TXT File

```
/*****************************************************************************/
/*                              Timer B Initialization                       */
/*****************************************************************************/
/* TIMB Status and Control Register              (TBSC)                      */
#define TIMB_OVERFLOW_INT       TIM_DISABLE        /* TIM_DISABLE/TIM_ENABLE */
#define TIMB_STOP_BIT           TIM_STOP           /* TIM_STOP/TIM_COUNT     */
#define TIMB_RESET_COUNTER      TIM_OFF            /* TIM_OFF/TIM_ON         */
#define TIMB_PRESCALER          TIM_BUS_CLK_DIV_1  /* TIM_BUS_CLK_DIV_1      */
                                                      /* TIM_BUS_CLK_DIV_2   */
                                                      /* TIM_BUS_CLK_DIV_4   */
                                                      /* TIM_BUS_CLK_DIV_8   */
                                                      /* TIM_BUS_CLK_DIV_16  */
                                                      /* TIM_BUS_CLK_DIV_32  */
                                                      /* TIM_BUS_CLK_DIV_64  */
                                                      /* TIM_PTE0_TCLKB      */
/*---------------------------------------------------------------------------*/
/* TIMB Counter Modulo Register                  (TBMODH,TBMODL)             */
#define TIMB_MODULO      0xFFFF                    /* 0xFFFF..0x0000         */
/*---------------------------------------------------------------------------*/
/* TIMB Channel 0 Status and Control Register    (TBSC0)                     */
#define TIMB_CH0_INT        TIM_DISABLE          /* TIM_DISABLE/TIM_ENABLE    */
#define TIMB_CH0_MODE       TIM_OUTPUT_PRESET_H  /* TIM_OUTPUT_PRESET_H       */
                                                 /* TIM_OUTPUT_PRESET_L       */
                                                 /* TIM_INPUT_CAPTURE_R_EDGE  */
                                                 /* TIM_INPUT_CAPTURE_F_EDGE  */
                                                 /* TIM_INPUT_CAPTURE_FR_EDGE */
                                                 /* TIM_TOGGLE_ON_COMP        */
                                                 /* TIM_CLEAR_ON_COMP         */
                                                 /* TIM_SET_ON_COMP           */
                                                 /* TIM_TOGGLE_ON_COMP_BUFF   */
                                                 /* TIM_CLEAR_ON_COMP_BUFF    */
                                                 /* TIM_SET_ON_COMP_BUFF      */
#define TIMB_CH0_TOGGLE_ON_OVERFLOW TIM_NO          /* TIM_NO/TIM_YES         */
#define TIMB_CH0_MAXIMUM_DUTY_CYCLE TIM_NO          /* TIM_NO/TIM_YES         */
/*---------------------------------------------------------------------------*/
/* TIMB Channel_0 Register                       (TBCH0H,TBCH0L)             */
/*                !!!!!!!!   Unaffected by reset    !!!!!!!!                 */
#define TIMB_CH0_VALUE            0xFFFF          /* 0..0xFFFF              */
/*---------------------------------------------------------------------------*/
/* TIMB Channel 1 Status and Control Register    (TBSC1)                     */
#define TIMB_CH1_INT        TIM_DISABLE          /* TIM_DISABLE/TIM_ENABLE    */
#define TIMB_CH1_MODE       TIM_OUTPUT_PRESET_H  /* TIM_OUTPUT_PRESET_H       */
                                                 /* TIM_OUTPUT_PRESET_L       */
                                                 /* TIM_INPUT_CAPTURE_R_EDGE  */
                                                 /* TIM_INPUT_CAPTURE_F_EDGE  */
                                                 /* TIM_INPUT_CAPTURE_FR_EDGE */
                                                 /* TIM_TOGGLE_ON_COMP        */
                                                 /* TIM_CLEAR_ON_COMP         */
                                                 /* TIM_SET_ON_COMP           */
/* !  Output compare for channel_1 is buffered when the chnnel_0 is in a !   */
/*   buffered mode. In that case the channel_1 can not be an input capture.  */
#define TIMB_CH1_TOGGLE_ON_OVERFLOW TIM_NO          /* TIM_NO/TIM_YES         */
#define TIMB_CH1_MAXIMUM_DUTY_CYCLE TIM_NO          /* TIM_NO/TIM_YES         */
/*---------------------------------------------------------------------------*/
/* TIMB Channel_1 Register                       (TBCH1H,TBCH1L)             */
/*                !!!!!!!!   Unaffected by reset    !!!!!!!!                 */
#define TIMB_CH1_VALUE            0xFFFF          /* 0..0xFFFF              */
/*---------------------------------------------------------------------------*/
```

3.5 On-Chip Drivers Interface Description

HC08 SDK includes a set of on-chip drivers used to initialize, configure and access the on-chip peripherals. The on-chip drivers provide a C language application programming interface (API) to the peripheral module (see **Figure 3-1**). This interface is common for all input/output operations. Use of the *ioctl* command provides a very efficient and easy way to access a peripheral module. It increases the code portability and readability and, thus, decrease the number of bugs in the developed code.

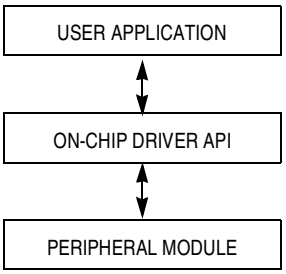

Figure 3-1. User Interface

This interface provides the following API statements:

```
ioctl(peripheral_module_identifier, command,
command_specific_parameter);
```

or, (if *ioctl* command returns a value):

```
var = ioctl(peripheral_module_identifier, command,
command_specific_parameter);
```

Where:

- *Peripheral_module_identifier* parameter specify module for example TIMA (timer A module), TIMB (timer B module), etc.
- *Command* parameter specifies the action, which will be performed on the peripheral module. The list of all commands available can be found in **Section 5. On-Chip Drivers**.
- *Command_specific_parameter* parameter specifies other data required to execute the command.

Example 2. Using IOCTL

```
IOCTL(PORTB, PORT_SET_PINS, PORT_PIN1|PORT_PIN2);
IOCTL(ADC, ADC_START, ADC_ATD2);
IOCTL(TIMA, TIM_CLEAR_CH0_FLAG, NULL);
```

This example shows miscellaneous IOCTL commands. The parameters are:

1. The first one specifies the peripheral module:
   - PORTB — input output port B
   - ADC — analog-to-digital (A/D) converter
   - TIMA — timer A
2. The second one is the command:
   - PORT_SET_PINS — to set pin
   - ADC_START — to start A/D conversion
   - TIM_CLEAR_CH0_FLAG — to clear flag
3. The third one is the command specific parameter:
   - PORT_PIN1 | PORT_PIN2 — to specify that pin 1 and 2 will be set
   - ADC_ADT2 — to specify input 2 is selected for ADC conversation
   - NULL — no parameter is used

See **Section 5. On-Chip Drivers** where all *IOCTL* commands and their detailed descriptions can be found.

Tip: To see all available IOCTL commands and their parameters from within CodeWarrior IDE, open the appropriate <*name_of_driver*>.*h* include file (e.g. *pwmdrv.h* — Pulse Width Modulation driver include file, etc.). At the beginning of the file, there is a list of all implemented *IOCTL* commands.

3.6 Interrupts and Interrupt Service Routines

Handling interrupts using HC08 SDK is described in detail in **4.7 Interrupts**. This section contains a practical guide on how to write a user interrupt service routine.

3.7 *appconfig.h* file

The *appconfig.h* include file is the application specific configuration file. It is used to define configuration items of both on-chip and off-chip drivers, users callback, and pin assignment. See **1.5 Rules and Coding Standards** for more information about how to initialize on-chip peripheral modules using the *appconfig.h* file and the respective ioctl xx_INIT command.

Section 4. Core System Infrastructure

4.1 Contents

4.2 Introduction

The core system infrastructure is one of the three main blocks composing HC08 SDK. Its purpose is to provide the fundamental infrastructure for the HC08 device operation (e.g., the interrupt handling, static configuration, etc.). It also provides some additional support (commonly used macros, data types) and enables further integration with on-chip drivers.

4.3 Boot Sequence

The core system infrastructure provides the fundamental code which is executed before the user's main function. This code provides basic settings needed to initialize the chip, settings required by the Metrowerks CodeWarrior 4.0 Compiler, and initialization of global variables. Finally, it passes control to the user's application code (*main* function).

The following subsections provide a detailed description of all initialization performed before user's *main()* function is called.

4.3.1 peripheralInit()

The *peripheralInit()* performs peripheral initialization according to the appconfig.h file. The *peripheralInit()* function has to be called at the beginning of the main application code (*main()* function.

4.3.2 *main()*- User's Application Code

The *main()* function is called after all the above described code is executed (i.e., the processor is initialized). It is the place where the user writes the application code. By default the function is located in *main.c* file, but the file can be renamed by the user.

4.4 Data Types

HC08 SDK defines some basic data types to support code portability between different hardware architectures and tools. These basic data types are defined in the C header file *types.h,* This is used throughout the interface definitions for the on-chip drivers.

NOTE: *In some development environments these data type definitions are located in the type.h file.*

Data type definitions:

1. Generic word types
   - SWord16 — 16-bit signed variable/value
   - UWord16 — 16-bit unsigned variable/value
   - SByte — 8-bit signed variable/value
   - UByte — 8-bit unsigned variable/value
2. Miscellaneous types
   - type_uBits — bit addressable UByte
   - type_uLowHigh — byte addressable UWord16
   - mc_s3PhaseSystem — 3 SWord16 structure
   - typefunc_p — pointer to a function
3. Constants
   - TRUE — true value
   - FALSE — false value
   - NULL — null pointer

4.5 *ArchIO* and *ArchCore* Register Structures

The global variables *ArchIO* and *archCore* provide a C-callable interface to all peripheral and core registers mapped in data memory. Access to all peripheral and core registers is provided via these structures. Thus, there is no need to know the concrete addresses of the registers to write/read them. This mechanism increases code readability and portability and simplifies access to registers. *ArchIO*, *ArchCore*, *FlashBlockProtectReg,* and *COPControlRegister* are defined in the header file *arch.h*.

The global variable *ArchIO* is of the type *arch_sIO* and *ArchCore* is of the type *Arch_sCore*. The structure type *arch_sIO* is comprised in another structures, where each structure corresponds to one on-chip peripheral module.

The examples shown below use *ArchIO* and *ArchCore* global variables.

Example 3. Using ArchIO and ArchCore

```
UWord16 RegValue;

RegValue = ArchIO.TimerA.Channel1.Value.Word;
ArchIO.TimerA.Channel1.Value.Word = 0x8000;
ArchCore.LVIStatusControlReg.Byte = 0x25;
```

This code reads timerA channel 1 register (TACH1), writes to the timer A channel 1 register (TACH1), and finally writes to the low-voltage inhibit (LVI) status and control register (LVISCR).

All peripheral registers also have defined short name substitutes for the long ArchIo name:

```
UWord16 RegValue;

RegValue = TACH1;
TACH1 = 0x8000;
LVISCR = 0x25;
```

This reading/ writing can be also performed using the *periphMemRead* and *periphMemWrite* functions. For more detailed information on these macros, see **4.6 General Periphery Functions**.

```
UWord16 RegValue;

RegValue = periphMemRead(&TACH1);
periphMemWrite(0x8000,&TACH1);
```

This method is recommended, as both periphMemRead() and periphMemWrite() functions guarantee the proper sequence to access each byte of the word.

4.6 General Periphery Functions

This section describes functions having direct access to the periphery. These functions are placed in the periph.asm file. periph.h has to be included while this functions are used.

4.6.1 periphMemRead() - memory read

Call(s):

UWord16 periphMemRead(UWord16 *pAddr);

Arguments:

Table 4-1. *periphMemRead* Arguments

pAddr	in	The memory address from which to read a 16-bit word

Description:

The *periphMemRead()* function reads a 16-bit word from the memory location addressed by parameter pAddr. The function read the high byte first and then the low byte. This sequence is required mainly for registers which have to be latched, or have a defined byte reading order, during the reading (e.g., TBCNT)

Example 4. periphMemRead() Usage

```
UWord16 RegValue;

RegValue = periphMemRead(&TBCNT);
```

This code reads the high byte of counter register (TBCNTH) which latches the content of the low byte (TBCNTL). Then the low byte (TBCNTL) is read to unlatch TBCNTL.

4.6.2 periphMemWrite() - memory write

Call(s):

UWord16 periphMemWrite(UWord16 Data, UWord16 *pAddr);

Arguments:

Table 4-2. *periphMemWrite* Arguments

Data	in	The 16-bit data to write to the memory
pAddr	in	The memory address from which to write a 16-bit word

Description:

The *periphMemWrite()* function writes a 16-bit word to the memory location addressed by parameter pAddr. The function writes the high byte first and then the low byte. This sequence is required for writing to registers (e.g., TBMOD).

Example 5. periphMemWrite() Usage

```
periphMemWrite(0x1234, TBMOD);
```

This code writes 0x1234 to the 0x12 to the high byte of the counter modulo register (TBMODH), which inhibits the TOF bit and overflow interrupts until the low byte (TMODL) is written. Then the 0x34 is written to the TMODL.

4.7 Interrupts

This subsection describes interrupt processing and interrupt configuration using the HC08 SDK. For detailed information on interrupts and interrupt processing for the MC68HC908MR32, refer to *MC68HC908MR32/ MC68HC908MR16 Advance Information* (Motorola document order number MC68HC908MR32/D).

4.7.1 Processing Interrupts

An interrupt is an event generated by a condition inside the MCU or from external sources. When this event occurs, the interrupt processing transfers control from the currently executing program to an interrupt service routine (ISR), with the ability to later return to the current program upon completion of the ISR.

The HC08 SDK structure allows:

- Calling user ISR before IFS (interrupt flag service)
- Calling user ISR after IFS
- Allowing user his own IFS
- Sharing HC08 SDK and user's ISR
- Generating debug strobes on specified I/O pins during the ISR
- Debugging unhandled interrupts

4.7.1.1 Interrupt Callbacks

The interrupt system of SDK handles all possible interrupts of the MC68HC908MR32. User callback enables users to share their own ISR with an ISR provided by the HC08 SDK. The user can define his callback before HC08 SDK ISR. For example:

```
#define INT_PWM_RELOAD_CALLBACK_1 IsrPWM_Reload
```

In this example, the IsrPWM_Reload() user function is called first. Then, the PWM Interrupt flag is cleared.

The user can also define his own callback after HC08 SDK ISR. For example:

```
#define INT_PWM_RELOAD_CALLBACK_2 IsrPWM_Reload
```

In this example, the PWM interrupt flag will be cleared and then the IsrPWM_Reload() user function will be called.

4.7.1.2 Interrupt Flag Service

Each on-chip peripheral interrupt source has its own interrupt flag, which must be cleared while the interrupt is serviced. This interrupt flag indicates an interrupt is pending and inside the ISR this flag must be cleared. The constant INT_*FlagName*_FLAG defined in the appconfig.h determine if the flag is cleared by HC08 SDK or if the user will have to take care of this flag himself.

Example 6

```
/* appconfig.h file */
#define INT_PWM_RELOAD_CALLBACK_1       IsrPWM_Reload
#define INT_PWM_RELOAD_FLAG             CLEAR_USER
```

This definition leaves flag service to the user. In this case, the user is responsible for servicing the interrupt flag.

Example 7. Clearing Interrupt Flags inside ISR

```
/* application code */
/*****************************************************
      PWM A Reload Interrupt Service Routine
*****************************************************/
void IsrPWM_Reload(void)
{
      /* ISR code */
      ...
      /* clear Reload interrupt flag */
      IOCTL(PWM, PWM_CLEAR_RELOAD_FLAG, NULL);
}
```

This example shows the PWM reload interrupt service routine.

NOTE: *The PWM_CLEAR_ RELOAD_FLAG IOCTL() command is used to clear the reload interrupt flag (PWMF) in control register 1 (PCTL1).*

Default value of *INT_PWM_RELOAD_FLAG* is *CLEAR_AUTO.* If the user does not define the INT_*FlagName*_FLAG HC08 SDK will take care of the flag. More information about *FlagName* for all ISR can be found in specified periphery driver descriptions in **Section 5. On-Chip Drivers**.

4.7.1.3 Interrupt Debug Strobes

Debug strobes are useful for displaying the execution time on the oscilloscope. The user can monitor not only extreme values of execution time, but also its time evolution.

Selected I/O pins are set at the beginning of an interrupt and cleared at the end. The required I/O port and pin must be defined in the appconfig.h by definition:

INT_*InterruptName*_STROBE_PORT PORT*x*

and

INT_*InterruptName*_STROBE_PIN *n*

where:

x is the port identifier (A, B, C, E, F) and the *n* is pin number (0, 1, 2, 3, 4, 5, 6, 7)

More information about *InterruptName* for all ISR can be found in specified periphery driver descriptions in **Section 5. On-Chip Drivers**.

Example 8. Users Debug Strobe definition for PWM interrupt

```
#define INT_PWM_RELOAD_STROBE_PORT      PORTB
#define INT_PWM_RELOAD_STROBE_PIN       4
```

4.7.1.4 Interrupt Debug Mode

Interrupt debug mode helps the user identify unhandled interrupts. If the user defines INT_DEBUG_MODE TRUE in the appconfig.h, then a jump to a never-ending cycle will be added to the all unhandled interrupts. If the execution of the application ends in this cycle, the user can easily find which interrupt is unhandled.

Example 9. Interrupt Debug Mode

```
/* appconfig.h */
INT_DEBUG_MODE              TRUE
```

The default value of the INT_DEBUG_MODE is FALSE. If the interrupt debug mode is not selected all interrupts will end by RTI.

4.7.1.5 Interrupt Processing Flow

Every ISR is divided into three parts with these functions:

- In the second part, HC08 SDK executive code is placed and the interrupt flag is cleared.
- The first and third parts are reserved for calling user's callback functions. The user can also choose to maintain interrupt flag control himself. At the beginning and ending of an interrupt (if required by the user) an assert and deassert of the debug strobe will be placed. A never-ending cycle will be called at the end of the interrupt if debug mode is selected.

Refer to **Figure 4-1** for a detailed processing flowchart.

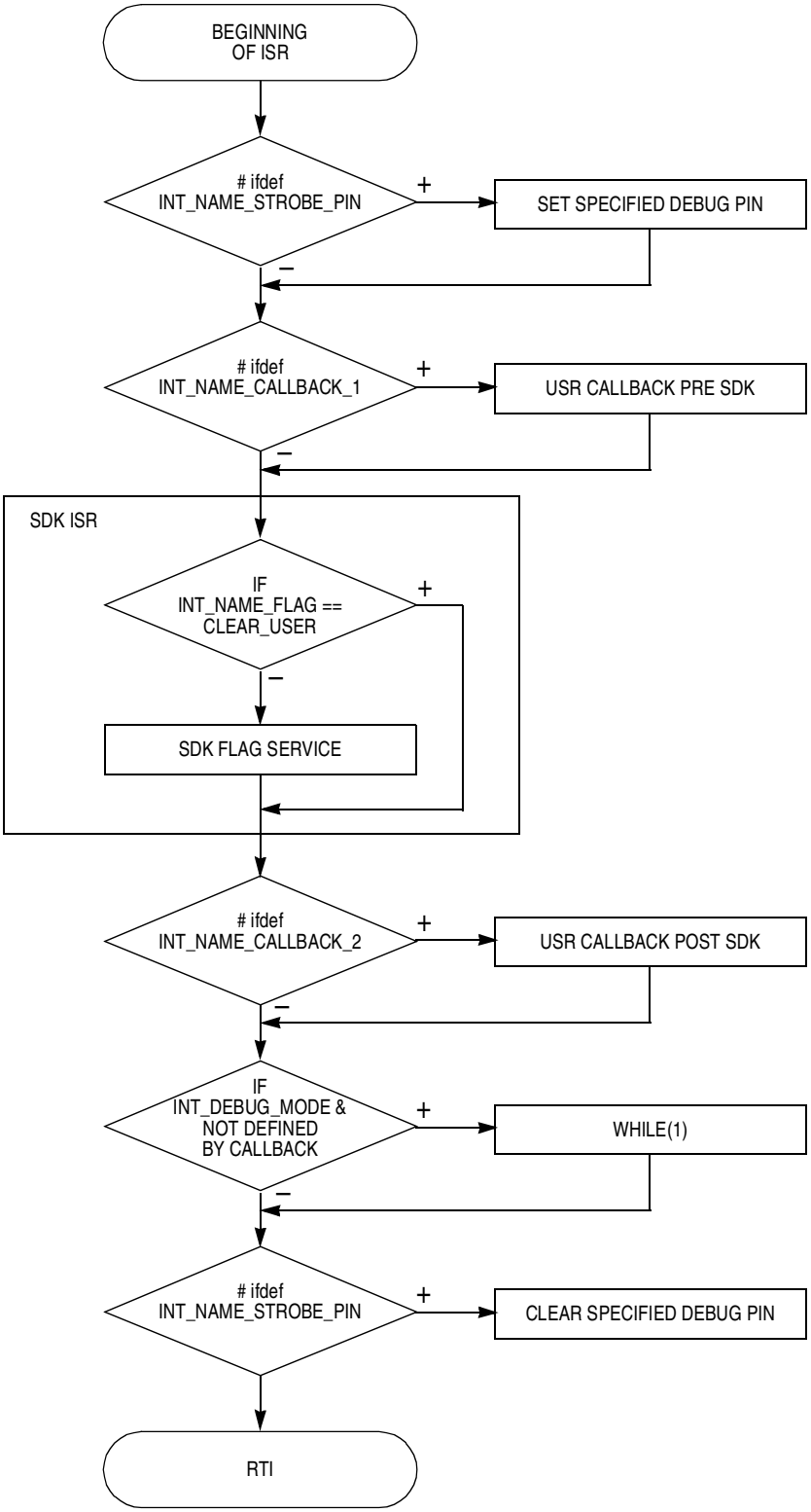

Figure 4-1. Interrupt Processing Flow

Section 5. On-Chip Drivers

5.1 Contents

5.2 Introduction

One strength of 8-bit SDK is that it provides a high degree of architectural and hardware independence for the application code. This portability is achieved by the modular design of 8-bit SDK. Which in this case, isolates all chip-specific functionality into a set of defined, tested, and documented application programming interfaces (APIs).

This section describes the APIs for on-chip drivers, forming the interface between hardware and application software. The source code implementation can be found at ***<...>\68HC908MR32\drivers*** of 8-bit SDK directory. It defines the API by identifying all public interface functions, commands, and data structures.

8-bit SDK on-chip driver's API are implemented as a *low-level device driver interface*. The low-level device driver interface was chosen mainly for its efficiency and also because it enables the utilization of the whole hardware functionality. Another reason is the non-standardized approach, on how to use most of the on-chip peripheral modules. The portability of the low-level device driver interface is not influenced so much by the lower abstraction level, but mainly by the capability of the peripheral module hardware. This means portability is ensured between devices, which involves the same or a very similar implementation of the peripheral module. In the case of quite different peripheral modules on target devices, the portability is much lower.

On-chip drivers usage considerations:

- Peripheral module hardware and functionality knowledge

  The only efficient and, in some cases, safe usage of the on-chip peripheral module is based on the user knowledge that the user has about the module itself. A comprehensive description can be found in the *MC68HC908MR32/ MC68HC908MR16 Advance Information* (Motorola document order number MC68HC908MR32/D) and in various Motorola application notes. The way in which the on-chip driver's API is designed takes advantage of the whole hardware capability. The self-explaining names of the driver commands will help the user to find the desired hardware feature.

- On-chip driver commands implemented as macros

  Almost all commands are implemented as efficient C macros. Exceptions are documented in each detailed description of the commands.

- Efficient use of the driver commands

  The general form of the driver command is:

  ```
  ioctl(peripheral_module_identifier,command,
  command_specific_parameter);
  ```

  Where, the peripheral_module_identifier parameter specifies the peripheral module by the predefined symbolic constants, like PWM, TIMA, TIMB, etc.

  The command parameter specifies the action, which will be performed on the peripheral module. It represents the command name as it is implemented for each on-chip driver.

  The command_specific_parameter parameter specifies other data required to execute the command. Generally speaking, it can be a pointer to the structure, the NULL value, or a variable-value in dependency with the specific command. If the required parameter is a variable value, it is recommended that, if possible,

a constant value be used because it influences the efficiency of the resulting code. This efficiency is illustrated by the following examples.

```
IOCTL(TIMB,TIM_SET_CH1_INT,TIM_ENABLE);
```

Constant used results in:

```
                BSET  6,89

IOCTL(TIMB,TIM_SET_CH1_INT,varU8);
```

Variable used results in:

```
                BIT   #1
                BNE   L1673 ;abs = 1673
                BCLR  6,89
                SKIP2 L1675 ;abs = 1675
L1673:          BSET  6,89
L1675:
```

NOTE: *Some macros expand just to a single assembly instruction (as illustrated in the above examples). Some other macros expand to more assembly instructions, e.g., the different mode setting where it is necessary to clear the previous setting and then, to set the new mode. This is illustrated by the following example:*

```
IOCTL(TIMA,TIM_SET_PRESCALER,TIM_BUS_CLK_DIV_32);
                BSET  2,14
                BCLR  1,14
                BSET  0,14
```

There can even be longer commands. These commands incorporate some higher functionality than only a simple access to the peripheral registers. An example can be commands which perform the mathematical calculations for data scaling to fit the results into the desired data range, e.g., recounting of the PWM duty cycle in percentage of the actual value to be written to the PWM value register.

- Implementation details

  The next example is intended to illustrate the macro expansion process. The corresponding items are highlighted by the same color through this macro expansion example.

  IOCTL command general syntax:

```
IOCTL(module_ID,cmd_name,cmd_spec_param)
```

  Example:

```
IOCTL(PWM,PWM_WRITE_MODULO,0xFF)
```

  Implementation:

  Common include file — sys.h

```
#define IOCTL(id,cmd,param) IOCTL_##id##__##cmd(param)
```

  On-chip driver include file — pwmdrv.h

```
#define IOCTL_PWM__PWM_WRITE_MODULO(param)PMOD = param
```

  Generated assembly code:

```
LDHX        #255
STHX        40
```

5.3 Phase Locked Loop (PLL) Drivers

The phase locked loop (PLL) driver performs both the initial configuration during startup and LOCTL commands for controlling the peripheral module.

The initial configuration sets required clock sources and waits for PLL lock. All required parameters for PLL configuration are expected in the **appconfig.h**. The commands for peripheral module control are performed by macros.

5.3.1 API Definition

Required Files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "plldrv.h"
```

NOTE: *The included files must be kept in order.*

5.3.2 Static Initialization

Call(s:)

```
SByte pllInit(void);
```

Description:

The pllInit function sets PLL peripheral module. The required parameters for peripheral module configuration are defined in **appconfig.h** (see **Table 5-1**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues:

All enters parameters are defined in **appconfig.h**

Example 10. Configuration items for appconfig.h

```
/* Modules for Static Configuration */
#define INCLUDE_PLL
/*************************************************************************
*          PLL Initialization
*************************************************************************/
/* PLL Control Register 1                          (PCTL)
#define PLL_ON_BIT            PLL_ON       /* PLL_ON / PLL_OFF */
#define PLL_BASE_CLOCK        PLL_CGMVCLK  /* PLL_CGMXCLK / PLL_CGMVCLK */
```

NOTE: *The previous definition determine using PLL clock source as the base of CPU bus clock. The function pllInit(), where the required setting occurs, is called automatically before main, if INCLUDE_PLL is defined in* ***appconfig.h***

Table 5-1. PLL Driver Constant Definitions

Constant Definition	Parameters[1]	Description	Note[2]
PLL_ON_BIT	PLL_ON / PLL_OFF	PLLON in PCTL = 1/0	d
PLL_INTERRUPT	PLL_DISABLE / PLL_ENABLE	PLLIE in PCTL = 0/1	d
PLL_BASE_CLOCK	PLL_CGMXCLK / PLL_CGMVCLK	BCS in PCTL = 0/1	d
PLL_FREQUENCY_MUL	PLL_MUL1 / PLL_MUL2 / PLL_MUL3 PLL_MUL4 / PLL_MUL5 / PLL_MUL6 PLL_MUL7 / PLL_MUL8 / PLL_MUL9 PLL_MUL10 / PLL_MUL11 / PLL_MUL12 PLL_MUL13 / PLL_MUL14 / PLL_MUL15	MUL7,MUL6,MUL5,MUL4 in PPG = 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15	d
PLL_VCO_FREQUENCY_MUL	PLL_MUL1 / PLL_MUL2 / PLL_MUL3 PLL_MUL4 / PLL_MUL5 / PLL_MUL6 PLL_MUL7 / PLL_MUL8 / PLL_MUL9 PLL_MUL10 / PLL_MUL11 / PLL_MUL12 PLL_MUL13 / PLL_MUL14 / PLL_MUL15	VRS7,VRS6,VRS5,VRS4 in PPG = 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9 / 10 / 11 / 12 / 13 / 14 / 15	d
PLL_BANDWIDTH	PLL_MANUAL / PLL_AUTOMATIC	AUTO in PBWC = 0/1	d
PLL_MODE	PLL_ACQUISITION / PLL_TRACKING	ACQ in PBWC = 0/1	d

1. First item in the parameters column is the default.
2. Configuration in **appconfig.h**:
   d — parameter with defined default reset state
   u — parameter with undefined default reset state
   o — parameter with write-once register

5.3.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*:

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
void        IOCTL (module, command, parameters);
UByte       IOCTL (module, command, parameters);
UWord16     IOCTL (module, command, parameters);
void        IOCTL (module, command, *parameters);
UByte       IOCTL (module, command, *parameters);
UWord16     IOCTL (module, command, *parameters);
```

Description:

A MACRO for operation with a PLL peripheral register is called according to the command and value parameters.

Example:

The following command sets bit PLLIE (PLL interrupt enable) in the register PCTL to 1.

```
IOCTL(PLL, PLL_SET_INT, PLL_ENABLE);
```

Arguments:

module (*in)* — specifies module, in this case, the module is PLL

command (*in)* — specifies target which has to be addressed

parameters (*in, inout, out)* — data passed to the IOCTL macro function

All commands and appropriate parameters are arranged in **Table 5-2**. The table uses the following conventions:

- Items separators:
  - / only one of the specified items is allowed
  - | consolidation of items is allowed (item1|item2|item4)
  - & intersection of items is allowed (item1&item2&item3)
- Implementation:
  - f — function
  - M — macro

Table 5-2. PLL Driver Macro and Function Commands

Command	Parameters[1]			Description	Notes[2]
PLL_INIT	NULL			Static configuration according appconfig return 0	f
PLL_GET_CONTROL_REG	NULL			return PCTL (UByte)	M
PLL_WRITE_CONTROL_REG	Ubyte	in	<0..0xFF>	PCTL = 0..0xFF	M
PLL_SET_ON_BIT	PLL_ON / PLL_OFF			PLLON in PCTL = 1/0	M
PLL_SET_BASE_CLOCK	PLL_CGMXCLK / PLL_CGMVCLK			BCS in PCTL = 0/1	M
PLL_GET_LOCK_BIT	NULL			return LOCK of PBWC	M

1. First item in the parameters column is the default.
2. f = function
   M = macro

5.4 PLL Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.4.1 Debug Strobes

Debug strobes allow the interrupt duration to be observed on the user specified GPIO port and pin. At the beginning of the interrupt, the strobe signal is set and when finished it is cleared.

Debug strobe port and pin specification:

```
#define INT_PLL_STROBE_PORT      PORT
#define INT_PLL_STROBE_PIN       Pin Number
```

Example of setting the debug strobe signal on port A pin 4:

```
#define INT_PLL_STROBE_PORT      A
#define INT_PLL_STROBE_PIN       4
```

5.4.2 Debug Mode

The debug mode helps the user to find the unhandled interrupts. If the INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and an unhandled interrupt occurs, the program will run an endless loop.

```
#define INT_DEBUG_MODE    TRUE
```

5.4.3 User Callbacks

Users can define two different types of callbacks for execution in his own interrupt code.

1. This definition in the **appconfig.h** file installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

   ```
   #define INT_PLL_RELOAD_CALLBACK_1 function_name_1
   ```

2. This definition in **appconfig.h** file installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

   ```
   #define INT_PLL_RELOAD_CALLBACK_2 function_name_2
   ```

5.5 Pulse-Width Modulator (PWM) Driver

The PWM driver performs the statical configuration of the PWM module and IOCTL commands for controlling the peripheral module.

The statical initialization sets the PWM peripheral module according to user settings in the **appconfig.h** file which overwrites the default configuration of the registers. Initialization consists of three parts:

1. Setting of the write-once registers (performed in the premain function).
2. Setting of the registers with default value defined after reset (function pwmInit sets the PWM peripheral module according to the **appconfig.h** file if the set value differs from the default reset state)
3. Setting of registers which have not defined any value after reset (function pwmInit set the PWM peripheral module according to the **appconfig.h** file)

Commands for peripheral module control are performed by:

1. Functions (suitable for longer and often used code)
2. Macros (suitable for short code)

5.5.1 API Definition

This section defines the application programming interface (API).

Public Data Structure(s):

```
typedef struct
{
      SWord16 PhaseA;     /*Current in phase A*/
      SWord16 PhaseB;     /*Current in phase B*/
      SWord16 PhaseC;     /*Current in phase C*/
} mc_s3PhaseSystem;
```

Required Files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "pwmdrv.h"
```

NOTE: *The included files must be kept in order.*

5.5.2 Static Initialization

Call(s:)

```
void pwmInit(void);
```

Description:

The pwmInit function sets PWM peripheral module. The required parameters for peripheral module configuration are defined in **appconfig.h** (see **Table 5-3**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: All enters parameters are defined in **appconfig.h**

Example:

These definitions have to be in **appconfig.h**

```
/* Modules for Static Configuration */
#define INCLUDE_PWM
/*************************************************************************
*          PWM Initialization
*************************************************************************/
/* PWM Control Register 1                    (PCTL1)*/
#define PWM_DISABLE_BANK_X    PWM_NO        /* PWM_NO/PWM_YES */
#define PWM_DISABLE_BANK_Y    PWM_NO        /* PWM_NO/PWM_YES */
#define PWM_RELOAD_INT        PWM_ENABLE    /* PWM_DISABLE /PWM_ENABLE */
```

NOTE: *The previous definition determines the setting of bits DISX and DISY in PCTL1 to 0 and PWMEN to 1. The function pwmInit(), where the required setting occurs, is called automatically before main if INCLUDE_PWM is defined in* ***appconfig.h****.*

Table 5-3. PWM Driver Constant Definitions (Sheet 1 of 2)

Constant Definition	Parameters[1]	Description	Notes[2]
PWM_DISABLE_BANK_X	PWM_NO / PWM_YES	DISX in PCTL1 = 0/1	d
PWM_DISABLE_BANK_Y	PWM_NO / PWM_YES	DISY in PCTL1 = 0/1	d
PWM_RELOAD_INT	PWM_DISABLE /PWM _ENABLE	PWMINT in PCTL1 = 0/1	d
PWM_CURRENT_CORRECTION	PWM_CORRECTION_NO / PWM_ CORRECTION_SOFTWARE / PWM_CORRECTION_DURING_DEADTIME/ PWM_CORRECTION_DURING_CYCLE	ISENS0, ISENS1 in PCTL1 = 0/1/2/3	d
PWM_LOAD_OK	PWM_NO / PWM_YES	LDOK in PCTL1 = 0/1	d[3]
PWM_MODULE	PWM_DISABLE /PWM _ENABLE	PWMEN in PCTL1 = 0/1	d
PWM_RELOAD_FREQUENCY	PWM_EVERY_1_CYCLE / PWM_EVERY_2_CYCLE / PWM_EVERY_4_CYCLE / PWM_EVERY_8_CYCLE /	**LDFQ0, LDFQ1** in PCTL2 = 0/1/2/3	d[4]
PWM_SOFTWARE_CORRECTION	PWM_VAL_1 | PWM_VAL_3 | PWM_VAL_5 | PWM_VAL_2 | PWM_VAL_4 | PWM_VAL_6	IPOL1, IPOL2, IPOL3 in PCTL2 = 0|0|0|4|2|1	d[4]
PWM_PRESCALER	PWM_FOP_DIV_1 / PWM_FOP_DIV_2 / PWM_FOP_DIV_3 / PWM_FOP_DIV_4	**PRSC0, PRCS1** in PCTL2 = 0/1/2/3	d[4]
PWM_FAULT4_INT	PWM_DISABLE / PWM_ENABLE	FINT4 in FCR	d
PWM_FAULT4_MODE	PWM_MANUAL / PWM_AUTOMATIC	FMODE4 in FCR	d
PWM_FAULT3_INT	PWM_DISABLE / PWM_ENABLE	FINT3 in FCR	d
PWM_FAULT3_MODE	PWM_MANUAL / PWM_AUTOMATIC	FMODE3 in FCR	d
PWM_FAULT2_INT	PWM_DISABLE / PWM_ENABLE	FINT2 in FCR	d
PWM_FAULT2_MODE	PWM_MANUAL / PWM_AUTOMATIC	FMODE2 in FCR	d
PWM_FAULT1_INT	PWM_DISABLE / PWM_ENABLE	FINT1 in FCR	d
PWM_FAULT1_MODE	PWM_MANUAL / PWM_AUTOMATIC	FMODE1 in FCR	d
PWM_OUTPUT_CONTROL	PWM_NONE / PWM_OUT1 | PWM_OUT2 | PWM_OUT3 | PWM_OUT4 | PWM_OUT5 | PWM_OUT6	OUT1, OUT2, OUT3, OUT4, OUT5, OUT6 in PWMOUT = 0/1|2|4|8|10|20	d
PWM_MANUAL_OUTPUT_CONTROL	PWM_DISABLE / PWM_ENABLE	OUTCTL in PWMOUT = 0/1	d
PWM_MODULO	<0..0x0FFF>	**Bit 0.. Bit 11** in PMODH, PMODL = 0..0xFFFF	u[4]
PWM_VALUE_1	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL1H, PVAL1L = 0..0xFFFF	d

Table 5-3. PWM Driver Constant Definitions (Sheet 2 of 2)

Constant Definition	Parameters[1]	Description	Notes[2]
PWM_VALUE_2	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL2H, PVAL2L = 0..0xFFFF	d
PWM_VALUE_3	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL3H, PVAL3L = 0..0xFFFF	d
PWM_VALUE_4	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL4H, PVAL4L = 0..0xFFFF	d
PWM_VALUE_5	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL5H, PVAL5L = 0..0xFFFF	d
PWM_VALUE_6	<0..0xFFFF>	**Bit 0.. Bit 15** in PVAL6H, PVAL6L = 0..0xFFFF	d
PWM_ALIGN (2)	PWM_CENTER / PWN_EDGE	EDGE in CONFIG = 0/1	o[5]
PWM_BOTTOM_POLARITY (2)	PWM_POSITIVE / PWM_NEGATIVE	BODNEG in CONFIG = 0/1	o[5]
PWM_TOP_POLARITY (2)	PWM_POSITIVE / PWM_NEGATIVE	TOPNEG in CONFIG = 0/1	o[5]
PWM_MODE (2)	PWM_COMPLEMENTARY / PWM_INDEPENDENT	INDEP in CONFIG = 0/1	o[5]
PWM_DEAD_TIME	<0..0xFF>	Bit 0..Bit 7 in DEADTM = 0..0xFF	o[5]
PWM_DISABLE_MAP	PWM_DPIN1_X | PWM_DPIN2_X PWM_DPIN2_Y | PWM_DPIN3_X | PWM_DPIN4_Y | PWM_DPIN5_X | PWM_DPIN5_Y | PWM_DPIN6_Y	Bit 0..Bit 7 in DISMAP = 0xFF/ 80|40|20|10|8|4|2|1	o[5]

1. First item in the parameters column is the default.
2. Configuration in **appconfig.h**:
   d — parameter with defined default reset state
   u — parameter with undefined default reset state
   o — parameter with write-once register
3. PCTL1 is set at the end of the pwmInit() function and confirms the setting by the Load OK Bit (LDOK)
4. The action takes effect when PWM_LOAD_OK is used to confirm the setting
5. See **5.15.2 Configuration Items**.

5.5.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: ***Inout*** *parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
void        IOCTL (module, command, parameters);
UByte       IOCTL (module, command, parameters);
UWord16     IOCTL (module, command, parameters);
void        IOCTL (module, command, *parameters);
UByte       IOCTL (module, command, *parameters);
UWord16     IOCTL (module, command, *parameters);
```

Description:

A MACRO for operation with a PWM peripheral register is called according to the command and value parameters.

Example:

```
IOCTL(PWM, PWM_BANK_X, PWM_YES); /* Sets DISX in PCTL1 to 1 */
```

Arguments:

module	*in*	specifies module, in this case the module is PWM
command	*in*	specifies target which has to be addressed
parameters	*in, inout, out*	data passed to the IOCTL macro function

All commands and appropriate parameters are shown in **Table 5-4**. The table uses the following conventions:

- Items separators
  - / only one of the specified items is allowed
  - | consolidation of items is allowed (item1|item2|item4)
  - & intersection of items is allowed (item1&item2&item3)
- Implementation:
  - f — function
  - M — Macro
  - **Bold** — Buffered

Table 5-4. PWM Driver Macros and Functions Commands (Sheet 1 of 4)

Command[1]	Parameters[2]			Description	Notes[3]
PWM_INIT	NULL			Static configuration according appconfig return 0	f
PWM_GET_CONTROL_REG_1	NULL			return PCTL1 (UByte)	M
PWM_WRITE_CONTROL_REG_1	Ubyte	in	<0..0xFF>	PCTL1=0..0xFF	M
PWM_SET_DISABLE_BANK_X	PWM_NO / PWM_YES			DISX in PCTL1=0/1	M
PWM_SET_DISABLE_BANK_Y	PWM_NO / PWM_YES			DISY in PCTL1=0/1	M
PWM_SET_RELOAD_INT	PWM_DISABLE / PWM_ENABLE			PWMINT in PCTL1=0/1	M
PWM_CLEAR_RELOAD_FLAG	NULL			PWMF in PCTL1= 0	M
PWM_SET_LOAD_OK	NULL			LDOK in PCTL1= 1	M
PWM_SET_MODULE	PWM_DISABLE / PWM_ENABLE			PWMEN in PCTL1=0/1	M
PWM_GET_DISABLE_BANK_X	NULL			return DISX of PCTL1	M
PWM_GET_DISABLE_BANK_Y	NULL			return DISY of PCTL1	M
PWM_GET_RELOAD_INT	NULL			return PWMINT of PCTL1	M
PWM_GET_RELOAD_FLAG	NULL			return PWMF of PCTL1	M
PWM_GET_MODULE	NULL			return PWMEN of PCTL1	M
PWM_SET_CURRENT_CORRECTION	PWM_CORRECTION_NO / PWM_CORRECTION_SOFTWARE / PWM_CORRECTION_DURING_DEADTIME / PWM_CORRECTION_DURING_CYCLE			ISENS0, ISENS1 in PCTL1=0/1/2/3	M
PWM_GET_CURRENT_CORRECTION	NULL			return ISENS0, ISENS1 of PCTL1	
PWM_GET_CONTROL_REG_2	NULL			return PCTL2 (UByte)	M
PWM_WRITE_CONTROL_REG_2	Ubyte	in	<0..0xFF>	PCTL2 = 0..0xFF	M
PWM_SET_RELOAD_FREQUENCY	PWM_EVERY_1_CYCLE / PWM_EVERY_2_CYCLE / PWM_EVERY_4_CYCLE / PWM_EVERY_8_CYCLE			**LDFQ0, LDFQ1** in PCTL2 = 0/1/2/3	M[4],[5]

Table 5-4. PWM Driver Macros and Functions Commands (Sheet 2 of 4)

Command[1]	Parameters[2]			Description	Notes[3]
PWM_SET_SOFTWARE_CORRECTION	PWM_VAL_1 | PWM_VAL_3 | PWM_VAL_5 | PWM_VAL_2 | PWM_VAL_4 | PWM_VAL_6			IPOL1, IPOL2, IPOL3 in PCTL2 = 0|0|0|4|2|1	M[4],[5]
PWM_SET_PRESCALER	PWM_FOP_DIV_1 / PWM_FOP_DIV_2 / PWM_FOP_DIV_3 / PWM_FOP_DIV_4			**PRSC0, PRCS1** in PCTL2 = 0/1/2/3	M[4],[5]
PWM_GET_RELOAD_FREQUENCY	NULL			return **LDFQ0, LDFQ1** of PCTL2	M
PWM_GET_SOFTWARE_CORRECTION	NULL			return **IPOL1, IPOL2, IPOL3** of PCTL2	M
PWM_GET_PRESCALER	NULL			return **PRSC0, PRCS1** of PCTL2	M
PWM_GET_FAULT_CONTROL_REG	NULL			return FCR (UByte)	M
PWM_WRITE_FAULT_CONTROL_REG	Ubyte	in	<0..0xFF>	FCR = 0..0xFF	M
PWM_SET_FAULT4_INT	PWM_DISABLE / PWM_ENABLE			FINT4 in FCR	M
PWM_SET_FAULT4_MODE	PWM_MANUAL / PWM_AUTOMATIC			FMODE4 in FCR	M
PWM_SET_FAULT3_INT	PWM_DISABLE / PWM_ENABLE			FINT3 in FCR	M
PWM_SET_FAULT3_MODE	PWM_MANUAL / PWM_AUTOMATIC			FMODE3 in FCR	M
PWM_SET_FAULT2_INT	PWM_DISABLE / PWM_ENABLE			FINT2 in FCR	M
PWM_SET_FAULT2_MODE	PWM_MANUAL / PWM_AUTOMATIC			FMODE2 in FCR	M
PWM_SET_FAULT1_INT	PWM_DISABLE / PWM_ENABLE			FINT1 in FCR	M
PWM_SET_FAULT1_MODE	PWM_MANUAL / PWM_AUTOMATIC			FMODE1 in FCR	M
PWM_GET_FAULT4_INT	NULL			return FINT4 of FCR	M
PWM_GET_FAULT4_MODE	NULL			return FMODE4 of FCR	M
PWM_GET_FAULT3_INT	NULL			return FINT3 of FCR	M
PWM_GET_FAULT3_MODE	NULL			return FMODE3 of FCR	M
PWM_GET_FAULT2_INT	NULL			return FINT2 of FCR	M
PWM_GET_FAULT2_MODE	NULL			return FMODE2 of FCR	M
PWM_GET_FAULT1_INT	NULL			return FINT1 of FCR	M
PWM_GET_FAULT1_MODE	NULL			return FMODE1 of FCR	M
PWM_GET_FAULT_STATUS_REG	NULL			return FSR (UByte)	M
PWM_GET_FAULT_PIN	PWM_FPIN1/PWM_FPIN2/ PWM_FPIN3/PWM_FPIN4			return FPINx in FSR	M
PWM_GET_FAULT_FLAG	PWM_FPIN1/PWM_FPIN2/ PWM_FPIN3/PWM_FPIN4			return FFLAGx in FSR	M
PWM_GET_CURRENT_SENSING	NULL			return FTACK (UByte)	M
PWM_WRITE_FAULT_ACKNOWLEDGE	Ubyte	in	<0..0xFF>	FTACK = 1|4|16|64	M
PWM_SET_FAULT_ACKNOWLEDGE	FTACK1 FTACK2 FTACK3 FTACK4			FTACK1 in FTACK = 1 FTACK2 in FTACK = 1 FTACK3 in FTACK = 1 FTACK4 in FTACK = 1	M
PWM_SET_OUTPUT_CONTROL	NONE / PWM_OUT1 | PWM_OUT2 | PWM_OUT3 | PWM_OUT4 | PWM_OUT5 | PWM_OUT6			OUT1, OUT2, OUT3, OUT4, OUT5, OUT6 in PWMOUT = 0/1|2|4|8|10|20	M[5]
PWM_SET_MANUAL_OUTPUT_CONTROL	PWM_DISABLE/PWM_ENABLE			OUTCTL in PWMOUT = 0/1	M

Table 5-4. PWM Driver Macros and Functions Commands (Sheet 3 of 4)

Command[1]	Parameters[2]			Description	Notes[3]
PWM_GET_COUNTER	NULL			return UWord16 <0..0x0FFFH> Bit 0.. Bit 11 in PMODH, PMODL	f[6]
PWM_GET_MODULO	NONE			return UWord16 <0..0x0FFFH> **Bit 0.. Bit 11** in PMODH,PMODL	f
PWM_WRITE_MODULO	UWord16	in	<0..0x0FFF>	**Bit 0.. Bit 11** in PMODH, PMODL = 0..0xFFFF	M[4]
PWM_UPDATE_MODULO	UWord16	in	<0..0x0FFF>	**Bit 0.. Bit 11** in PMODH, PMODL = 0..0xFFFF PCTL1_LDOK = 1	M
PWM_GET_VALUE_n where n = <1..6> example: PWM_GET_VALUE_1	UWord16	in	<0..0x0FFF>	return UWord16 <0..0x0FFFH> Bit 0.. Bit 11 in PVALnH, PVALnL in the example: return **Bit 0.. Bit 15** in PVAL1H, PVAL1L	M[4]
PWM_WRITE_VALUE_n where n = <1..6> example: PWM_WRITE_VALUE_1	UWord16	in	<0..0x0FFF>	**Bit 0.. Bit 15** in PVALnH, PVALnL = 0..0xFFFF in the example: **Bit 0.. Bit 15** in PVAL1H, PVAL1L = 0..0xFFFF	M[4]
PWM_UPDATE_VALUE_n where n = <1..6> example: PWM_UPDATE_VALUE_1	UWord16	in	<0..0x0FFF>	**Bit 0.. Bit 15** in PVALnH, PVALnL = 0..0xFFFF CTL1_LDOK = 1 in the example: **Bit 0.. Bit 15** in PVAL1H, PVAL1L = 0..0xFFFF CTL1_LDOK = 1	M
PWM_UPDATE_VALUE_REGS_COMPL	&mc_s3PhaseSystem	in	3xUWord16	Bit 0..15 in PVAL1,3,5 **PVAL1L,H** = (*Value).PhaseA **PVAL3L,H** = (*Value).PhaseB **PVAL5L,H** = (*Value).PhaseC PCTL1_LDOK = 1	M
PWM_UPDATE_VALUE_CONSTANT	const	in	<0..0x0FFF>	Bit 0..15 in PVAL1,3,5 **PVAL1L,H** = Value **PVAL3L,H** = Value **PVAL5L,H** = Value PCTL1_LDOK = 1	M

Table 5-4. PWM Driver Macros and Functions Commands (Sheet 4 of 4)

Command[1]	Parameters[2]			Description	Notes[3]
PWM_UPDATE_SCALED_VALUE_REGS	&mc_s3PhaseSystem	in	3xUWord16	Bit 0..15 in PVAL1,3,5 **PVAL1L,H** = (*Value).PhaseA*scale **PVAL3L,H** = (*Value).PhaseB*scale **PVAL5L,H** = (*Value).PhaseC*scale PCTL1_LDOK = 1	M[7] [8] (7,8)
PWM_CHARGE_BOOT_STRAP	UByte	in	<0..0xFF>	if PWM_ENABLE = _NO switch on PWM2,4,6 for n = value PWM Reload cycles	f [9](6)

1. PACTL1 is set at the end of the pwmInit() function and confirms the setting by the Load OK Bit (LDOK)
2. First item in the parameters column is the default.
3. f = function
   M = macro
4. The action takes effect when PWM_LOAD_OK is used to confirm the setting
5. User have to make sure that no interrupt, where is same register affected, can occur during this command
6. User have to make sure that no interrupt, where is same register read, can occur during this command
7. IOCTL(PWM, PWM_UPDATE_SCALED_VALUE_REGS, mc_s3PhaseSystem *pHandle)
   Description:
   Update value reg with scaling according to constant PWM_MODULO defined in **appconfig.h**
   if PWM_MODULO is in {0x0100, 0xFF} use H bytes of inputs
   if PWM_MODULO < 0xFF call PwmUpdateScaledValue_8
   if PWM_MODULO > 0x100 call PwmUpdateScaledValue
8. Use constant PWM_MODULO defined in the **appconfig.h**
9. Detailed explanation is in **5.5.4 Functional Description**

Table 5-5. Memory Consumption an Execution Time

Command	Function Name	Size	Cycles (Min)	Cycles (Typ)	Cycles (Max)	Notes
PWM_READ_COUNTER	periphMemRead16	9	23	23	23	(1)
PWM_CHARGE_BOOT_STRAP	PwmChargeBootStrap	41	n*PWM Rel.	n*PWM Rel.	n*PWM Rel.	(2),(3)
PWM_UPDATE_SCALED_VALUE_REGS_16	PwmUpdateScaledValue	70	216	216	216	—
PWM_UPDATE_SCALED_VALUE_REGS_8	PwmUpdateScaledValue_8	45	TBD	TBD	TBD	TBD

1. The function periphMemRead16 is from periph.c library file
2. n = third parameter of IOCTL command
3. PWM Rel. = period of PWM reload

5.5.4 Functional Description

5.5.4.1 PwmChargeBootStrap

Call(s:)

```
void PwmChargeBootStrap (UByte reloadNumb)
```

Description:

In this case the PWM_OUT2, PWM_OUT4 and PWM_OUT6 are set and kept active for ‘ReloadNumb’ period of PWM reload cycles and allows the precharging of the bootstrap capacitors. During that period PWM reload interrupt is disabled (PWMINT = 0). After the function is finished the PWMINT is restored.

Range Issues: None

Special Issues:

The function is performed only when the PWMEN in PCTL1 = 1

The function is used in following IOCTL command

```
IOCTL(PWM, PWM_CHARGE_BOOT_STRAP, reloadNumb);
```

Example 11. PwmChargeBootStrap Usage

```
#define RELOAD_NUMB 10
...
Before execution PWMEN in PCTL1 = 0

IOCTL(PWM, PWM_CHARGE_BOOT_STRAP, RELOAD_NUMB);

During execution:
      PWM_ENABLE in PCTL1 = 1 (_YES)
      PWMINT in PCTL1 = 0 (Disabled)
                              PWMOUT = OUT2|OUT4|OUT6|OUTCTL
After execution:
      PWM_ENABLE in PCTL1 = 1
                              PWMOUT = OUT2|OUT4|OUT6
```

5.5.4.2 PwmUpdateScaledValue

```
void PwmUpdateScaledValue(mc_s3PhaseSystem * pHandle, UByte pwmModulo)
```

Description:

The function scale UWord16 value from mc_s3PhaseSystem variable to the PWM_MODULO range, place results to PWM value registers and confirm new values by setting the LDOK bit.

Represents: input * PWM_MODULO/256^2

Real Calculation:

PVAL1,3,5 = (pHandle -> PhaseA,B,C * high of PWM_MODULO)/256

Range Issues: None

Special Issues:

The function is used in IOCTL command:

```
IOCTL(PWM, PWM_UPDATE_SCALED_VALUE_REGS, mc_s3PhaseSystem * pHandle);
```

The function is used only when PVM_MODULO > 0x100.

Example 12. PwmUpdateScaledValue Usage

in **appconfig.h**

```
#define PWM_MODULO 0x300
```

in **main.c**

```
        mc_s3PhaseSystem motorVoltage;

        motorVoltage.PhaseA = 3000;
        motorVoltage.PhaseA = 30000;
        motorVoltage.PhaseA = 200;

        IOCTL(PWM, PWM_UPDATE_SCALED_VALUE_REGS, &motorVoltage)

    results:
        PVAL1: 35
        PVAL3: 351
        PVAL5: 2
```

5.5.4.3 PwmUpdateScaledValue_8

```
void PwmUpdateScaledValue(mc_s3PhaseSystem * pHandle, UByte pwmModulo)
```

Description:

The function scales UWord16 value from mc_s3PhaseSystem variable to the PWM_MODULO range, place results to PWM value registers and confirm new values by setting the LDOK bit.

Represents: input * PWM_MODULO/256^2

Real Calculation: PVAL1,3,5 = (low of (pHandle -> PhaseA,B,C) * low of PWM_MODULO)/256

Range Issues: None

Special Issues:

The function is used in IOCTL command:

```
IOCTL(PWM, PWM_UPDATE_SCALED_VALUE_REGS, mc_s3PhaseSystem * pHandle);
```

The function is used only when PVM_MODULO < 0xFF.

Example 13. PwmUpdateScaledValue_8 Usage

In **appconfig.h**:

```
#define PWM_MODULO 0x95
```

In **main.c**:

```
        mc_s3PhaseSystem motorVoltage;

        motorVoltage.PhaseA = 3000;
        motorVoltage.PhaseA = 30000;
        motorVoltage.PhaseA = 200;

        IOCTL(PWM, PWM_UPDATE_SCALED_VALUE_REGS, &motorVoltage)

    results:
        PVAL1: 6
        PVAL3: 68
        PVAL5: 0
```

5.6 PWM Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.6.1 Debug Strobes

Debug strobes allows the observing of the interrupt duration on the user specified GPIO port and pin. At the beginning of the interrupt the strobe signal is set and when finished it is cleared.

Debug Strobe Port and Pin Specification:

```
#define INT_PWM_RELOAD_STROBE_PORT      PORT
#define INT_PWM_RELOAD_STROBE_PIN       Pin Number
```

Example: Setting the debug strobe signal on port A pin 4:

```
#define INT_PWM_RELOAD_STROBE_PORT      A
#define INT_PWM_RELOAD_STROBE_PIN       4
```

5.6.2 Debug Mode

The debug mode helps the user to find the unhandled interrupts. If the INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and any unhandled interrupt occurs, the program will run an endless loop.

```
#define INT_DEBUG_MODE     TRUE
```

5.6.3 User Callbacks

Users can define two different types of callback for executing in his own interrupt code.

1. This definition in **appconfig.h** file installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_PWM_RELOAD_CALLBACK_1 function_name_1
```

2. This definition in **appconfig.h** file installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_PWM_RELOAD_CALLBACK_2 function_name_2
```

5.6.4 PWM Reload Flag

SDK automatically takes care of the PWM reload flag clearing it between pre- and post-SDK user callbacks.

If INT_PWM_RELOAD_FLAG_CARE_USER is defined in **appconfig.h,** SDK allows the user to take care of the interrupt flag clearing.

```
#define INT_PWM_RELOAD_FLAG_CARE_USER
```

5.7 Timer Drivers

The timer driver performs both the statical configuration of the timer module and LOCTL commands for controlling the peripheral module.

The statical initialization sets the timer peripheral module according to the user setting in the **appconfig.h** file which overwrites the default configuration of the registers. Commands for peripheral module control are performed by both functions and macros.

5.7.1 API Definition

Required Files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "timerdrv.h"
```

NOTE: *Included files must be kept in order.*

5.7.2 Static Initialization

Call(s:)

```
SByte timaInit(void);
SByte timbInit(void);
```

Description:

The timaInit and timbInit functions set the timer peripheral module. The required parameters for peripheral module configuration are defined in **appconfig.h** (see **Table 5-6**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: All enter parameters are defined in **appconfig.h**

Example 14. Static Initialization of Timer Driver

```
Next definition have to be in appconfig.h

/* Modules for Static Configuration */
#define INCLUDE_TIMA
/*************************************************************************
*       Timer Initialization
*************************************************************************/
/* TIMA Status and Control Register           (TASC)
#define TIMA_OVERFLOW_INT TIM_ENABLE    /* TIM_DISABLE / TIM_ENABLE */
#define TIMA_STOP_BIT     TIM_COUNT     /* TIM_STOP / TIM_COUNT */
```

NOTE: *The previous definition determine setting of bit TOIE in TASC to 1 and TSTOP to 0. The function timaInit(), where the required setting occur, is called automatically before main, if INCLUDE_TIMA is defined in* ***appconfig.h***

Table 5-6. Timer Driver Constants Definition (Sheet 1 of 3)

Constant Definition	Parameters[1]	Description	Notes[2]
TIMA_OVERFLOW_INT	TIM_DISABLE / TIM_ENABLE	TOIE in TASC= 0/1	d
TIMA_STOP_BIT	TIM_STOP / TIM_COUNT	TSTOP in TASC=1/0	d
TIMA_RESET_COUNTER	NULL	TRST in TASC = 0/1	d
TIMA_PRESCALER	TIM_BUS_CLK_DIV_1/ TIM_BUS_CLK_DIV_2/ TIM_BUS_CLK_DIV_4/ TIM_BUS_CLK_DIV_8/ TIM_BUS_CLK_DIV_16/ TIM_BUS_CLK_DIV_32/ TIM_BUS_CLK_DIV_64/ TIM_PTE3_TCLKA	PS2,PS1,PS0 in TASC = 1/0	d
TIMA_MODULO	0x0000..0xFFFF	TAMOD = 0x0000..0xFFFF	d
TIMA_CH0_INT	TIM_DISABLE / TIM_ENABLE	CH0IE in TASC0 = 0/1	d
TIMA_CH0_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP TIM_TOGGLE_ON_COMP_BUFF TIM_CLEAR_ON_COMP_BUFF TIM_SET_ON_COMP_BUFF	MS0A,ELS0B,ELS0A in TASC0 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMA_CH0_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV0 in TASC0 = 0/1	d
TIMA_CH0_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH0MAX in TASC0 = 0/1	d

Table 5-6. Timer Driver Constants Definition (Sheet 2 of 3)

Constant Definition	Parameters[1]	Description	Notes[2]
TIMA_CH1_INT	TIM_DISABLE / TIM_ENABLE	CH1IE in TASC1 = 0/1	d
TIMA_CH1_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP	MS1A,ELS1B,ELS1A in TASC1 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMA_CH1_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV1 in TASC1 = 0/1	d
TIMA_CH1_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH1MAX in TASC1 = 0/1	d
TIMA_CH2_INT	TIM_DISABLE / TIM_ENABLE	CH2IE in TASC2 = 0/1	d
TIMA_CH2_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP TIM_TOGGLE_ON_COMP_BUFF TIM_CLEAR_ON_COMP_BUFF TIM_SET_ON_COMP_BUFF	MS2A,ELS2B,ELS2A in TASC2 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMA_CH2_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV2 in TASC2 = 0/1	d
TIMA_CH2_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH2MAX in TASC2 = 0/1	d
TIMA_CH3_INT	TIM_DISABLE / TIM_ENABLE	CH3IE in TASC3 = 0/1	d
TIMA_CH3_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP	MS3A,ELS3B,ELS3A in TASC3 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMA_CH3_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV3 in TASC3 = 0/1	d
TIMA_CH3_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH3MAX in TASC3 = 0/1	d
TIMB_OVERFLOW_INT	TIM_DISABLE / TIM_ENABLE	TOIE in TBSC= 0/1	d
TIMB_STOP_BIT	TIM_STOP / TIM_COUNT	TSTOP in TBSC=1/0	d
TIMB_RESET_COUNTER	NULL	TRST in TBSC = 0/1	d

Table 5-6. Timer Driver Constants Definition (Sheet 3 of 3)

Constant Definition	Parameters[1]	Description	Notes[2]
TIMB_PRESCALER	TIM_BUS_CLK_DIV_1/ TIM_BUS_CLK_DIV_2/ TIM_BUS_CLK_DIV_4/ TIM_BUS_CLK_DIV_8/ TIM_BUS_CLK_DIV_16/ TIM_BUS_CLK_DIV_32/ TIM_BUS_CLK_DIV_64/ TIM_PTE3_TCLKB	PS2,PS1,PS0 in TBSC = 1/0	d
TIMB_MODULO	0x0000..0xFFFF	TBMOD = 0x0000..0xFFFF	d
TIMB_CH0_INT	TIM_DISABLE / TIM_ENABLE	CH0IE in TBSC0 = 0/1	d
TIMB_CH0_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP TIM_TOGGLE_ON_COMP_BUFF TIM_CLEAR_ON_COMP_BUFF TIM_SET_ON_COMP_BUFF	MS0A,ELS0B,ELS0A in TBSC0 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMB_CH0_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV0 in TBSC0 = 0/1	d
TIMB_CH0_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH0MAX in TBSC0 = 0/1	d
TIMB_CH1_INT	TIM_DISABLE / TIM_ENABLE	CH1IE in TBSC1 = 0/1	d
TIMB_CH1_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP	MS1A,ELS1B,ELS1A in TBSC1 = 0000 / 0100 / 0001 / 0010 / 0011 / 0101 / 0110 / 0111 / 1101 / 1110 / 1111 /	d
TIMB_CH1_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES	TOV1 in TBSC1 = 0/1	d
TIMB_CH1_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES	CH1MAX in TBSC1 = 0/1	d

1. First item in the parameters column is the default.
2. Configuration in **appconfig.h**:
   d — parameter with defined default reset state
   u — parameter with undefined default reset state
   o — parameter with write-once register

5.7.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
void        IOCTL (module, command, parameters);
UByte       IOCTL (module, command, parameters);
UWord16     IOCTL (module, command, parameters);
void        IOCTL (module, command, *parameters);
UByte       IOCTL (module, command, *parameters);
UWord16     IOCTL (module, command, *parameters);
```

Description:

A MACRO for operation with a timer peripheral register is called according to the command and value parameters.

Example:

```
The following command sets bit TIOE(Timer Overflow Interrupt
Enable) in the register TASC to 1.
IOCTL(TIMA, TIM_SET_OVERFLOW_INT, TIM_ENABLE);
```

Arguments:

module	*in*	specify module, in this case the module is TIMA or TIMB
command	*in*	specify target which have be addressed
parameters	*in, inout, out*	data passed to the IOCTL macro function

All commands and appropriate parameters are shown in **Table 5-2**. The table uses these conventions:

- Items separators

  / only one of the specified items is allowed

  | consolidation of items is allowed (item1|item2|item4)

  & intersection of items is allowed (item1&item2&item3)

- Implementation

  f — function

  M — Macro

I

Table 5-7. Timer Driver Macros and Functions Commands (Sheet 1 of 4)

Command	Parameters(1)			Description	Note (2)
TIM_INIT	NULL			Static configuration according appconfig return 0	M(3)
TIM_WRITE_CONTROL_REG	Ubyte	in	<0..0xFF>	TASC,TBSC = 0..0xFF	M(2)
TIM_GET_CONTROL_REG	NULL			return TASC,TBSC(UByte)	M(2)
TIM_CLEAR_OVERFLOW_FLAG	NULL			TOF in TASC,TBSC = 0	M(2)
TIM_SET_OVERFLOW_INT	TIM_DISABLE / TIM_ENABLE			TOIE in TASC,TBSC = 0/1	M(2)
TIM_SET_STOP_BIT	TIM_STOP / TIM_COUNT			TSTOP in TASC,TBSC = 1/0	M(2)
TIM_SET_RESET_COUNTER	TIM_OFF / TIM_ON			TRST in TASC,TBSC = 0/1	M(2)
TIM_SET_PRESCALER	TIM_BUS_CLK_DIV_1/ TIM_BUS_CLK_DIV_2/ TIM_BUS_CLK_DIV_4/ TIM_BUS_CLK_DIV_8/ TIM_BUS_CLK_DIV_16/ TIM_BUS_CLK_DIV_32/ TIM_BUS_CLK_DIV_64/ TIM_PTE3_TCLKA			PS0,PS1,PS2 in TASC,TBSC = 0/1/2/3/4/5/6/7	M(2)
TIM_GET_OVERFLOW_FLAG	NULL			return TOF of TASC,TBSC	M(2)
TIM_GET_OVERFLOW_INT	NULL			return TOIE of TASC,TBSC	M(2)
TIM_GET_STOP_BIT	NULL			return TSTOP of TASC,TBSC	M(2)
TIM_GET_COUNTER	NULL			return TASC,TBSC(UWord16)	M(2)
TIM_WRITE_MODULO	UWord16	in	<0..0xFFFF>	TAMOD,TBMOD(UWord16)	M(2)
TIM_GET_MODULO	NULL			return TAMOD,TBMOD(UWord16)	M(2)
TIM_WRITE_CH0_CONTROL_REG	Ubyte	in	<0..0xFF>	TASC0,TBSC0 = 0..0xFF	M(2)

Table 5-7. Timer Driver Macros and Functions Commands (Sheet 2 of 4)

Command	Parameters[1]			Description	Note (2)
TIM_GET_CH0_CONTROL_REG	NULL			return TASC0,TBSC0(UByte)	M[2]
TIM_CLEAR_CH0_FLAG	NULL			CH0F in TASC0,TBSC0 = 0	M[2]
TIM_SET_CH0_INT	TIM_DISABLE / TIM_ENABLE			CHOIE in TASC0,TBSC0 = 0/1	M[2]
TIM_SET_CH0_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES			TOV0 in TASC0,TBSC0 = 0/1	M[2]
TIM_SET_CH0_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES			CH0MAX in TASC0,TBSC0 = 0/1	M[2]
TIM_GET_CH0_FLAG	NULL			return CH0F of TASC0,TBSC0	M[2]
TIM_GET_CH0_INT	NULL			return CHOIE of TASC0,TBSC0	M[2]
TIM_SET_CH0_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP TIM_TOGGLE_ON_COMP_BUFF TIM_CLEAR_ON_COMP_BUFF TIM_SET_ON_COMP_BUFF			MS0B,MS0A,ELS0B,ELS0A in TASC0,TBSC0 = 0000/0100/ 0001/0010/0011/ 0101/0110/0111/ 1001/1010/1011	M[2]
TIM_WRITE_CH0_VALUE	UWord16	in	<0..0xFFFF>	return TACH0,TBCH0(UWord16)	M[2]
TIM_GET_CH0_VALUE	NULL			return TACH0,TBCH0(UWord16)	M[2]
TIM_WRITE_CH1_CONTROL_REG	Ubyte	in	<0..0xFF>	TASC1,TBSC1 = 0..0xFF	M[2]
TIM_GET_CH1_CONTROL_REG	NULL			return TASC1,TBSC1(UByte)	M[2]
TIM_CLEAR_CH1_FLAG	NULL			CH1F in TASC1,TBSC1 = 0	M[2]
TIM_SET_CH1_INT	TIM_DISABLE / TIM_ENABLE			CH1IE in TASC1,TBSC1 = 0/1	M[2]
TIM_SET_CH1_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES			TOV1 in TASC1,TBSC1 = 0/1	M[2]
TIM_SET_CH1_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES			CH1MAX in TASC1,TBSC1 = 0/1	M[2]
TIM_GET_CH1_FLAG	NULL			return CH1F of TASC1,TBSC1	M[2]
TIM_GET_CH1_INT	NULL			return CH1IE of TASC1,TBSC1	M[2]
TIM_SET_CH1_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP			MS1A,ELS1B,ELS1A in TASC1,TBSC1 = 0000/0100/ 0001/0010/0011/ 0101/0110/0111/	M[2]
TIM_WRITE_CH1_VALUE	UWord16	in	<0..0xFFFF>	return TACH1,TBCH1(UWord16)	M[2]

Table 5-7. Timer Driver Macros and Functions Commands (Sheet 3 of 4)

Command	Parameters[1]			Description	Note (2)
TIM_GET_CH1_VALUE	NULL			return TACH1,TBCH1(UWord16)	M[2]
TIM_WRITE_CH2_CONTROL_REG	Ubyte	in	<0..0xFF>	TASC2 = 0..0xFF	M[4]
TIM_GET_CH2_CONTROL_REG	NULL			return TASC2(UByte)	M[3]
TIM_CLEAR_CH2_FLAG	NULL			CH2F in TASC2 = 0	M[3]
TIM_SET_CH2_INT	TIM_DISABLE / TIM_ENABLE			CH1IE in TASC2 = 0/1	M[3]
TIM_SET_CH2_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES			TOV2 in TASC2 = 0/1	M[3]
TIM_SET_CH2_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES			CH2MAX in TASC2 = 0/1	M[3]
TIM_GET_CH2_FLAG	NULL			return CH2F of TASC2	M[3]
TIM_GET_CH2_INT	NULL			return CH2IE of TASC2	M[3]
TIM_SET_CH2_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP TIM_TOGGLE_ON_COMP_BUFF TIM_CLEAR_ON_COMP_BUFF TIM_SET_ON_COMP_BUFF			MS2B,MS2A,ELS2B,ELS2A in TASC2 = 0000/0100/ 0001/0010/0011/ 0101/0110/0111/ 1001/1010/1011	M[3]
TIM_WRITE_CH2_VALUE	UWord16	in	<0..0xFFFF>	return TACH2(UWord16)	M[3]
TIM_GET_CH2_VALUE	NULL			return TACH2(UWord16)	M[3]
TIM_WRITE_CH3_CONTROL_REG	Ubyte	in	<0..0xFF>	TASC3 = 0..0xFF	M[3]
TIM_GET_CH3_CONTROL_REG	NULL			return TASC3(UByte)	M[3]
TIM_CLEAR_CH3_FLAG	NULL			CH3F in TASC3 = 0	M[3]
TIM_SET_CH3_INT	TIM_DISABLE / TIM_ENABLE			CH3IE in TASC3 = 0/1	M[3]
TIM_SET_CH3_TOGGLE_ON_OVERFLOW	TIM_NO/TIM_YES			TOV3 in TASC3 = 0/1	M[3]
TIM_SET_CH3_MAXIMUM_DUTY_CYCLE	TIM_NO/TIM_YES			CH3MAX in TASC3 = 0/1	M[3]
TIM_GET_CH3_FLAG	NULL			return CH3F of TASC3	M[3]
TIM_GET_CH3_INT	NULL			return CH3IE of TASC3	M[3]

Table 5-7. Timer Driver Macros and Functions Commands (Sheet 4 of 4)

Command	Parameters[1]			Description	Note (2)
TIM_SET_CH3_MODE	TIM_OUTPUT_PRESET_H TIM_OUTPUT_PRESET_L TIM_INPUT_CAPTURE_R_EDGE TIM_INPUT_CAPTURE_F_EDGE TIM_INPUT_CAPTURE_FR_EDGE TIM_TOGGLE_ON_COMP TIM_CLEAR_ON_COMP TIM_SET_ON_COMP			MS3A,ELS3B,ELS3A in TASC3 = 0000/0100/ 0001/0010/0011/ 0101/0110/0111/	M[3]
TIM_WRITE_CH3_VALUE	UWord16	in	<0..0xFFFF>	return TACH3(UWord16)	M[3]
TIM_GET_CH3_VALUE	NULL			return TACH3(UWord16)	M[3]

1. First item in the parameters column is the default.
2. M = macro
3. The command is supported for both timer A and timer B (port identifier TIMA and TIMB).
4. The command is supported only for timer A (port identifer TIMA).

5.8 Timer Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.8.1 Debug Strobes

Debug strobes allow observation of the interrupt duration on the user specified GPIO port and pin. At the beginning of the interrupt the strobe signal is set and when finished it is cleared.

The following subsections provide more specific information regarding the debug strobe port and pin.

5.8.1.1 Timer Overflow Interrupts

The following definition has to be in **appconfig.h.**

```
#define INT_TIMA_OVERFLOW_STROBE_PORT   PORT
#define INT_TIMA_OVERFLOW_STROBE_PIN    Pin Number
#define INT_TIMB_OVERFLOW_STROBE_PORT   PORT
#define INT_TIMB_OVERFLOW_STROBE_PIN    Pin Number
```

Example of setting the debug strobe signal on port A pin 4:

```
#define INT_TIMA_OVERFLOW_STROBE_PORT A
#define INT_TIMA_OVERFLOW_STROBE_PIN 4
```

5.8.1.2 Channel Interrupts

The following definition in the **appconfig.h** enables the interrupt strobe signal on a specific port and pin.

```
#define INT_TIMA_CH0_STROBE_PORT        PORT
#define INT_TIMA_CH0_STROBE_PIN         Pin Number
#define INT_TIMA_CH1_STROBE_PORT        PORT
#define INT_TIMA_CH1_STROBE_PIN         Pin Number
#define INT_TIMA_CH2_STROBE_PORT        PORT
#define INT_TIMA_CH2_STROBE_PIN         Pin Number
#define INT_TIMA_CH3_STROBE_PORT        PORT
#define INT_TIMA_CH3_STROBE_PIN         Pin Number
#define INT_TIMB_CH0_STROBE_PORT        PORT
#define INT_TIMB_CH0_STROBE_PIN         Pin Number
#define INT_TIMB_CH1_STROBE_PORT        PORT
#define INT_TIMB_CH2_STROBE_PIN         Pin Number
```

Example of setting the debug strobe signal on port A pin 4:

```
#define INT_TIMA_CH0_STROBE_PORT        A
#define INT_TIMA_CH0_STROBE_PIN         4
```

5.8.2 Debug Mode

Debug mode helps the user to find unhandled interrupts. If INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and an unhandled interrupt occurs, the program will run in an endless loop.

```
#define INT_DEBUG_MODE     TRUE
```

5.8.3 User Callbacks

The user can define two different types of callbacks for executing his own interrupt code.

5.8.3.1 Timer Overflow Interrupts

This definition in **appconfig.h** installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_TIMA_OVERFLOW_CALLBACK_1 function_name_1
#define INT_TIMB_OVERFLOW_CALLBACK_1 function_name_1
```

This definition in **appconfig.h** installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_TIMA_OVERFLOW_CALLBACK_2 function_name_2
#define INT_TIMB_OVERFLOW_CALLBACK_2 function_name_2
```

5.8.3.2 Channel Interrupts

This definition in **appconfig.h** installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_TIMA_CH0_CALLBACK_1 function_name_1
#define INT_TIMA_CH1_CALLBACK_1 function_name_1
#define INT_TIMA_CH2_CALLBACK_1 function_name_1
#define INT_TIMA_CH3_CALLBACK_1 function_name_1
#define INT_TIMB_CH0_CALLBACK_1 function_name_1
#define INT_TIMB_CH1_CALLBACK_1 function_name_1
```

This definition in **appconfig.h** installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_TIMA_CH0_CALLBACK_2 function_name_2
#define INT_TIMA_CH1_CALLBACK_2 function_name_2
#define INT_TIMA_CH2_CALLBACK_2 function_name_2
#define INT_TIMA_CH3_CALLBACK_2 function_name_2
#define INT_TIMB_CH0_CALLBACK_2 function_name_2
#define INT_TIMB_CH1_CALLBACK_2 function_name_2
```

5.9 Serial Peripheral Interface (SPI) Drivers

The serial peripheral interface (SPI) driver performs both the statical configuration of the SPI module and the IOCTL commands for controlling the peripheral module.

The statical initialization sets the SPI peripheral module according to the user setting in **appconfig.h** which overwrites the default configuration of the registers. Commands for peripheral module control are performed by both functions and macros.

5.9.1 API Definition

Required files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "spidrv.h"
```

NOTE: *The included files must be kept in order.*

5.9.2 Static Initialization

Call(s:)

```
SByte spiInit(void);
```

Description:

The spiInit function sets the SPI peripheral module. The required parameters for peripheral module configuration are defined in **appconfig.h** (see **Table 5-8**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: None

Example 15. Static Initialization of the SPI Module

```
Next definition have to be in appconfig.h

/* Modules for Static Configuration */
#define INCLUDE_SPI
/*************************************************************************
*          SPI Initialization
*************************************************************************/
/* SPI Control Register       (SPCR)
#define SPI_CLOCK_POLARITY    SPI_POSITIVE /* SPI_POSITIVE / SPI_NEGATIVE */
#define SPI_WIRED_OR          SPI_ENABLE   /* SPI_DISABLE / SPI_ENABLE */
```

NOTE: *The previous definitions determine setting of bit CPOL and SPWOM in SPCR to 1. The function spiInit(), where the required settings occur, is called automatically before main if INCLUDE_SPI is defined in* ***appconfig.h***

Table 5-8. SPI Driver Constants Definition

Constant Definition	Parameters[1]	Description	Notes[2]
SPI_RX_INT	SPI_DISABLE / SPI_ENABLE	SPRIE in SPCR = 0/1	d
SPI_MASTER_BIT	SPI_MASTER / SPI_SLAVE	SPMSTR in SPCR=1/0	d
SPI_CLOCK_POLARITY	SPI_POSITIVE / SPI_NEGATIVE	CPOL in SPCR = 0/1	d
SPI_CLOCK_PHASE	SPI_F_EDGE / SPI_R_EDGE	CPHA in SPCR = 1/0	d
SPI_WIRED_OR	SPI_DISABLE / SPI_ENABLE	SPWOM in SPCR = 0/1	d
SPI_MODULE	SPI_DISABLE / SPI_ENABLE	SPE in SPCR = 0/1	d
SPI_TX_INT	SPI_DISABLE / SPI_ENABLE	SPTIE in SPCR = 0/1	d
SPI_ERROR_INT	SPI_DISABLE / SPI_ENABLE	ERRIE in SPSCR = 0/1	d
SPI_MODE_FAULT	SPI_DISABLE / SPI_ENABLE	TCIE in SPSCR = 0/1	d
SPI_BAUD_RATE	SPI_DIV_2 / SPI_DIV_8/ SPI_DIV_32 / SPI_DIV_128	SPR1, SPR0 in SPSCR = 0/1/2/3	d

1. First item in the parameters column is the default.
2. Configuration in **appconfig.h**:
   d — parameter with defined default reset state
   u — parameter with undefined default reset state
   o — parameter with write-once register

5.9.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
void   IOCTL (module, command, parameters);
UByte  IOCTL (module, command, parameters);
UWord16IOCTL (module, command, parameters);
void   IOCTL (module, command, *parameters);
UByte  IOCTL (module, command, *parameters);
UWord16IOCTL (module, command, *parameters);
```

Description:

A MACRO for operation with a peripheral register is called according to the command and value parameters.

Example:

The following command sets the CPOL bit (clock polarity) in the SPCR register to 1.

```
IOCTL(SPI, SPI_CLOCK_POLARITY, SPI_POSITIVE);
```

Arguments:

module	*in*	specifies the module, in this case the module is SPI
command	*in*	specifies target which has be addressed
parameters	*in, inout, out*	data passed to the IOCTL macro function

All commands and appropriate parameters are shown in **Table 5-9**. The table uses these conventions:

- Items separators

  / only one of the specified items is allowed

  | consolidation of items is allowed (item1|item2|item4)

  & intersection of items is allowed (item1&item2&item3)

- Implementation

  f — Function

  M — Macro

Table 5-9. SPI Driver Macros and Functions Commands (Sheet 1 of 2)

Command	Parameters[1]			Description	Note[2]
SPI_INIT	NULL			Static configuration according appconfig return 0	f
SPI_GET_CONTROL_REG	NULL			return SPCR (UByte)	M
SPI_WRITE_CONTROL_REG	Ubyte	in	<0..0xFF>	SPCR = 0..0xFF	M
SPI_SET_RX_INT	SPI_DISABLE / SPI_ENABLE			SPRIE in SPCR = 0/1	M
SPI_SET_MASTER_BIT	SPI_MASTER / SPI_SLAVE			SPMSTR in SPCR = 1/0	M
SPI_SET_CLOCK_POLARITY	SPI_POSITIVE / SPI_NEGATIVE			CPOL in SPCR = 0/1	M
SPI_SET_CLOCK_PHASE	SPI_F_EDGE / SPI_R_EDGE			CPHA in SPCR = 1/0	M
SPI_SET_WIRED_OR	SPI_DISABLE / SPI_ENABLE			SPWOM in SPCR = 0/1	M
SPI_SET_MODULE	SPI_DISABLE / SPI_ENABLE			SPE in SPCR = 0/1	M
SPI_SET_TX_INT	SPI_DISABLE / SPI_ENABLE			SPTIE in SPCR = 0/1	M
SPI_GET_RX_INT	NULL			return SPRIE of SPCR	M
SPI_GET_MASTER_BIT	NULL			return SPMSTR of SPCR	M
SPI_GET_CLOCK_POLARITY	NULL			return CPOL of SPCR	M
SPI_GET_CLOCK_PHASE	NULL			return CPHA of SPCR	M
SPI_GET_WIRED_OR	NULL			return SPWOM of SPCR	M
SPI_GET_MODULE	NULL			return SPE of SPCR	M
SPI_GET_TX_INT	NULL			return SPTIE of SPCR	M
SPI_GET_STATUS_REG	NULL			return SPSCR (UByte)	M
SPI_WRITE_STATUS_REG	Ubyte	in	<0..0xFF>	SPSCR = 0..0xFF	M

Table 5-9. SPI Driver Macros and Functions Commands (Sheet 2 of 2)

Command	Parameters[1]			Description	Note[2]
SPI_SET_ERROR_INT	SPI_DISABLE / SPI_ENABLE			ERRIE in SPSCR = 1/0	M
SPI_SET_MODE_FAULT	SPI_DISABLE / SPI_ENABLE			MODFEN in SPSCR = 0/1	M
SPI_GET_RX_FULL	NULL			return SPRF of SPCR	M
SPI_GET_ERROR_INT	NULL			return ERRIE of SPCR	M
SPI_GET_OVERFLOW	NULL			return OVRF of SPCR	M
SPI_GET_FAULT_FLAG	NULL			return MODF of SPCR	M
SPI_GET_TX_EMPTY	NULL			return SPTE of SPCR	M
SPI_GET_MODE_FAULT	NULL			return MODFEN of SPCR	M
SPI_GET_DATA_REG	NULL			return SPDR (UByte)	M
SPI_WRITE_DATA_REG	Ubyte	in	<0..0xFF>	SPDR = 0..0xFF	M

1. First item in the parameters column is the default.
2. f = function
   M = macro

5.10 SPI Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.10.1 Debug Strobes

Debug strobes allow the observation of the interrupt duration on the user specified GPIO port and pin. At the beginning of the interrupt the strobe signal is set and when finished it is cleared.

The following subsections provide more specific information pertaining to the debug strobe port and pin

5.10.1.1 SPI Receive Interrupt

This definition has to be in **appconfig.h**.

```
#define INT_SPI_RX_STROBE_PORT   PORT
#define INT_SPI_RX_STROBE_PIN    Pin Number
```

Example of setting the debug strobe signal on port A pin 4:

```
#define INT_SPI_RX_STROBE_PORT   A
#define INT_SPI_RX_STROBE_PIN    4
```

5.10.1.2 SPI Transmit Interrupt

This definition in the **appconfig.h** enables the interrupt strobe signal on a specified port and pin.

```
#define INT_SPI_TX_STROBE_PORT   PORT
#define INT_SPI_TX_STROBE_PIN    Pin Number
```

Example of setting the debug strobe signal on port A pin 4:

```
#define INT_SPI_TX_STROBE_PORT   A
#define INT_SPI_TX_STROBE_PIN    4
```

5.10.2 Debug Mode

The debug mode helps the user to find the unhandled interrupts. If INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and an unhandled interrupt occurs, the program will run in an endless loop.

Example:

```
#define INT_DEBUG_MODE     TRUE
```

5.10.3 User Callbacks

The user can define two different types of callbacks for executing his own interrupt code.

5.10.3.1 SPI Receive Interrupt

This definition in **appconfig.h** installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_SPI_RX_CALLBACK_1 function_name_1
```

This definition in **appconfig.h** installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_SPI_RX_CALLBACK_2 function_name_2
```

5.10.3.2 SPI Transmit Interrupt

This definition in **appconfig.h** installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_SPI_TX_CALLBACK_1 function_name_1
```

This definition in **appconfig.h** installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_SPI_TX_CALLBACK_2 function_name_2
```

5.11 Serial Communications Interface (SCI) Driver

This section describes the API for the 68HC908MRxx serial communication interface (SCI) on-chip module. The functionality of the SCI module itself is described in the specific device data sheet.

The SCI driver is dedicated to controlling the on-chip SCI module. It is comprised of the initialization routine for statical module configuration and IOCTL commands for controlling the module during run-time.

IOCTL commands for controlling the SCI module are implemented as:

1. Macros (in-line code — short code)
2. Functions (called function — longer code)

5.11.1 API Definition

Required Header File(s):

The following header files are needed in order to use ADC driver:

```
#include "sys.h"
#include "arch.h"
#include "periph.h"
#include "appconfig.h"
#include "config.h"

#include "plldrv.h"
#include "scidrv.h"
```

5.11.2 Configuration Items

This subsection summarizes the symbols used in definitions for the static configuration of the SCI module. This initialization is performed by the *sciInit*() function during initialization process.

The statical configuration routine *sciInit()* sets the SCI module according to the user settings specified in **appconfig.h** which overwrites the default configuration of the registers.

The initialization process is divided into the following three parts:

1. Setting of the write-once registers (performed in the premain function).
2. Setting of the registers with defined reset states.
   - The function *sciInit()* sets the SCI registers according to the application static configuration file **appconfig.h** if the user settings differ from the default reset state of the register.
   - Refer to **Table 5-10** for the configurable items.
3. Setting of the registers with undefined reset states.
   - The function *sciInit()* sets the SCI registers according to the **appconfig.h** file.
   - Refer to **Table 5-10** for the configurable items.

See **Example 16** and **Example 17** for more details.

Code to initialize the SCI driver is automatically included in the SDK project by inserting the following into **appconfig.h** file.

```
#define INCLUDE_SCI     /* include SCI initialization code
                           to application */
```

Items Separators:

/ only one of the specified items is allowed

| consolidation of items is allowed (item1|item2|item4)

& intersection of items is allowed (item1&item2&item3)

CAPITAL = constant (*in*)

Table 5-10. Configuration Items for appconfig.h

Symbol	Parameters	Description
SCI_MODULE	SCI_DISABLE / SCI_ENABLE	Enable or disable SCI module and the SCI baud rate generator. ENSCI in SCC1 = 0 / 1
SCI_LOOP_MODE	SCI_DISABLE / SCI_ENABLE	Sets SCI loop mode operation LOOPS in SCC1 = 0 / 1
SCI_DATA_POLARITY	SCI_NOT_INVERTED / SCI_INVERTED	Sets the data polarity of transmitted signal TXINV in SCC1 = 0 / 1
SCI_WAKEUP_COND	SCI_WAKE_BY_IDLE / SCI_WAKE_BY_ADDRESS	Set wakeup condition of the SCI module. WAKE in SCC1 = 0 / 1
SCI_IDLE_LINE	SCI_AFTER_START / SCI_AFTER_STOP	Select idle line type ILTY in SCC1 = 0 / 1
SCI_DATA_FORMAT	SCI_8BIT_NONE / SCI_7BIT_EVEN / SCI_7BIT_ODD / SCI_8BIT_EVEN / SCI_8BIT_ODD / SCI_9BIT_NONE	Select the format of data Sets M, PEN, PTY in SCC1 refer toTable 5-11 Default and reset state is SCI_8BIT_NONE.
SCI_BAUD_RATE	UWord32	Select the desired communication speed in bauds. Refer to Table 5-12 with valid baud rates
SCI_PRESCALER	1, 3, 4, 13	Select the desired SCI prescaler SCBR
SCI_DIVIDER	1, 2, 4, 8, 16, 32, 64, 128	Select the desired SCI divider SCBR
SCI_TX_EMPTY_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when data register is empty. SCTIE in SCC2 = 0 / 1.
SCI_TX_IDLE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when transmission complete. TCIE in SCC2 = 0 / 1.
SCI_RX_FULL_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when receiver is full. SCRIE in SCC2 = 0 / 1.
SCI_RX_IDLE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when receiver is idle. SCRIE in SCC2 = 0 / 1.
SCI_TRANSMITTER	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI transmitter TE in SCC2 = 0 / 1.
SCI_RECEIVER	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI receiver RE in SCC2 = 0 / 1.
SCI_RX_WAKEUP	SCI_DISABLE / SCI_ENABLE	Wakeup or standby the SCI receiver RWU in SCC1 = 0
SCI_OVERRUN_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX overrun IRQ ORIE in SCC3 = 0 / 1
SCI_NOISE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX noise IRQ NEIE in SCC3 = 0 / 1
SCI_FRAMING_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX framing IRQ FEIE in SCC3 = 0 / 1
SCI_PARITY_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX parity IRQ PEIE in SCC3 = 0 / 1

Table 5-11. Character Format Selection

Parameter	M	PEN:PTY	Parity	Stop Bits	Character Length
SCI_7BIT_EVEN	0	10	Even	1	10 bits
SCI_7BIT_ODD	0	11	Odd	1	10 bits
SCI_8BIT_NONE	0	0X	None	1	10 bits
SCI_8BIT_EVEN	1	10	Even	1	11 bits
SCI_8BIT_ODD	1	11	Odd	1	11 bits
SCI_9BIT_NONE	1	0X	None	1	11 bits

Table 5-12. Baud Rates

Bus Clock	Valid Baud Rates for Given Bus Clock
8000000	75, 150, 300, 600, 1200, 2400, 4800, 9600
4915200	150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 76800 200, 400, 800, 1600, 3200, 6400, 12800, 25600
7372800	300, 600, 1200, 2400, 4800, 9600, 19200, 38400 225, 450, 900, 1800, 3600, 7200, 14400, 28800, 57600, 115200

5.11.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Items Separators:

/ only one of the specified items is allowed

| consolidation of items is allowed (item1|item2|item4)

& intersection of items is allowed (item1&item2&item3)

CAPITAL = constant (*in*)

First item or items in the column is default

Implemented as:

f = function

M = Macro

5.11.3.1 SCI Input/Output Control Commands

The SCI input/output control (IOCTL) commands are defined in **Table 5-13**.

Table 5-13. SCI Input/Output Control Commands (Sheet 1 of 3)

Command	Parameters[1]	Description	Note[2]
SCI_INIT	NULL	Initializes SCI module by data from configuration file (**appconfig.h**)	f
SCI_SET_MODULE	SCI_DISABLE / SCI_ENABLE	Enables or disables SCI module and the SCI baud rate generator. ENSCI in SCC1 = 0 / 1	M
SCI_SET_LOOP_MODE	SCI_DISABLE / SCI_ENABLE	Sets SCI loop mode operation LOOPS in SCC1 = 0 / 1	M
SCI_SET_DATA_POLARITY	SCI_NOT_INVERTED / SCI_INVERTED	Sets the data polarity of transmitted signal TXINV in SCC1 = 0 / 1	M
SCI_SET_DATA_FORMAT	SCI_8BIT_NONE / SCI_7BIT_EVEN / SCI_7BIT_ODD / SCI_8BIT_EVEN / SCI_8BIT_ODD / SCI_9BIT_NONE	Select the format of data Sets M, PEN, PTY in SCC1 refer to **Table 5-11**	M
SCI_SET_WAKEUP_COND	SCI_WAKE_BY_IDLE / SCI_WAKE_BY_ADDRESS	Select the wakeup condition WAKE in SCC1 = 0 / 1	M
SCI_SET_IDLE_LINE	SCI_AFTER_START / SCI_AFTER_STOP	Select idle line type ILTY in SCC1 = 0 / 1	M

Table 5-13. SCI Input/Output Control Commands (Sheet 2 of 3)

Command	Parameters[1]	Description	Note[2]
SCI_SET_TX_EMPTY_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when data register is empty. SCTIE in SCC2 = 0 / 1.	M
SCI_SET_TX_IDLE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when transmission complete. TCIE in SCC2 = 0 / 1.	M
SCI_SET_RX_FULL_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when receiver is full. SCRIE in SCC2 = 0 / 1.	M
SCI_SET_RX_IDLE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the IRQ generation when receiver is idle. SCRIE in SCC2 = 0 / 1.	M
SCI_SET_TRANSMITTER	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI transmitter TE in SCC2 = 0 / 1.	M
SCI_SET_RECEIVER	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI receiver RE in SCC2 = 0 / 1.	M
SCI_SET_RX_STANDBY	NULL	Puts SCI receiver into standby state RWU in SCC1 = 1	M
SCI_SET_RX_WAKEUP	NULL	Wakeup SCI receiver from standby state RWU in SCC1 = 0	M
SCI_SET_OVERRUN_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX overrun IRQ ORIE in SCC3 = 0 / 1	M
SCI_SET_NOISE_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX noise IRQ NEIE in SCC3 = 0 / 1	M
SCI_SET_FRAMING_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX framing IRQ FEIE in SCC3 = 0 / 1	M
SCI_SET_PARITY_INT	SCI_DISABLE / SCI_ENABLE	Enable or disable the SCI RX parity IRQ PEIE in SCC3 = 0 / 1	M
SCI_CLEAR_STATUS_REG1	NULL	Clear the SCI status register 1	M
SCI_GET_STATUS_REG1	NULL		M
SCI_GET_TX_EMPTY	NULL	Return the status of the SCI Transmitter Empty Flag SCTE in SCS1	M
SCI_GET_TX_IDLE	NULL	Return the status of the SCI Transmission Complete Flag TC in SCS1	M
SCI_GET_RX_FULL	NULL	Return the status of the SCI Receiver Full Flag SCRF in SCS1	M
SCI_GET_RX_IDLE	NULL	Return the status of the SCI Receiver Idle Flag IDLE in SCS1	M

Table 5-13. SCI Input/Output Control Commands (Sheet 3 of 3)

Command	Parameters[1]	Description	Note[2]
SCI_GET_RX_ERROR	SCI_OR | SCI_NF | SCI_FE | SCI_PE	Return the status of error flags specified by input mask OR, NF, FE, PE in SCC1	M
SCI_GET_STATUS_REG2	NULL	Return the content of the SCI Status Register 2 SCS2	M
SCI_READ_8BIT_DATA	NULL	Read the 8 bit data from the SCI data register SCDR	M
SCI_READ_9BIT_DATA	NULL	Read the 9bit data from the SCDR and R8 in SCC3	M
SCI_WRITE_8BIT_DATA	UByte	Write the 8 bit data to the SCI data register SCDR	M
SCI_WRITE_9BIT_DATA	UWord16	Write the 9bit data to the SCDR and T8 in SCC3	M
SCI_GET_STATUS	NULL	Get operation status of read/write functions	M
SCI_READ_CANCEL	NULL	Cancel non-blocking read operation	M
SCI_WRITE_CANCEL	NULL	Cancel non-blocking write operation	M
SCI_CLEAR_EXCEPTION	NULL	Clear exception of read/write functions if exist	M

1. First item in the parameters column is the default.
2. f = function
   M = macro

Example 16. SCI Usage without Buffering

```
/*****************************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: application configuration file
*
* Modules Included:  None
*
*****************************************************************************/

#ifndef __APPCONFIG_H
    #define __APPCONFIG_H

/*****************************************************************************
* Include needed driver initialization routines
*****************************************************************************/

#define   INCLUDE_PLL           /* PLL support */
#define   INCLUDE_SCI           /* SCI support */

/* Specify the your Xtal clock frequency                                    */
#define XTAL_CLOCK             8000000L

/*****************************************************************************/
/*                          PLL Initialization                               */
/*****************************************************************************/
/* BUS_CLOCK  = (XTAL_CLOCK * PLL_FREQUENCY_MUL) / 4                        */

/* PLL Control Register 1                                  (PCTL)           */
#define PLL_ON_BIT             PLL_ON                /* PLL_ON / PLL_OFF     */
#define PLL_BASE_CLOCK         PLL_CGMVCLK           /* PLL_CGMXCLK /
                                                        PLL_CGMVCLK          */
/*.........................................................................*/
/* PLL Programing Register 1                               (PPG)            */
#define PLL_FREQUENCY_MUL      PLL_MUL4              /* PLL_MUL1..PLL_MUL15  */
#define PLL_VCO_FREQUENCY_MUL  PLL_MUL3              /* PLL_MUL1..PLL_MUL15  */
/*.........................................................................*/
/* PLL Control Register 2                                  (PBWC)           */
#define PLL_BANDWIDTH          PLL_AUTOMATIC         /* PLL_MANUAL /
                                                        PLL_AUTOMATIC        */

/*****************************************************************************/
/*                          Watch dog initialization                         */
/*****************************************************************************/
/* Config Register   !!! Write-Once-Register !!!           (CONFIG)         */
#define WDO_COPD               WDO_DISABLE           /* WDO_ENABLE /
                                                        WDO_DISABLE          */
```

```
/*****************************************************************************/
/*                           SCI Initialization                              */
/*****************************************************************************/

#define SCI_BAUD_RATE          9600

#define SCI_DATA_FORMAT        SCI_8BIT_NONE         /* SCI_8BIT_NONE /
                                                        SCI_7BIT_ODD  /
                                                        SCI_7BIT_EVEN /
                                                        SCI_8BIT_EVEN /
                                                        SCI_8BIT_ODD  /
                                                        SCI_9BIT_NONE        */

#define SCI_WAKEUP_COND        SCI_WAKE_BY_IDLE      /* SCI_WAKE_BY_IDLE /
                                                        SCI_WAKE_BY_ADDRESS  */

#define SCI_DATA_POLARITY      SCI_NOT_INVERTED      /* SCI_NOT_INVERTED /
                                                        SCI_INVERTED         */

#define SCI_IDLE_LINE          SCI_AFTER_STOP        /* SCI_AFTER_START /
                                                        SCI_AFTER_STOP       */

#define SCI_LOOP_MODE          SCI_DISABLE           /* SCI_DISABLE /
                                                        SCI_ENABLE           */

/* Install the SCI callbacks */
#define INT_SCI_TX_CALLBACK_1  mySciTxISR

#endif /* __APPCONFIG_H*/


/*****************************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: sci_demo.c
*
* Description: SCI device driver demo application
*
* Modules Included:
*           < main >
*           < mySciTxISR >
*
*****************************************************************************/

#include "types.h"
#include "arch.h"
#include "hidef.h"
#include "sys.h"
#include "appconfig.h"

#include "plldrv.h"
#include "scidrv.h"

/* function prototypes */
void mySciTxISR(void);
```

```
/* global variables */
volatile UByte    position = 0;
volatile UByte    idt[] = "Serial Communication Interface";

void main(void)
{
    UByte sci_tmp;
    UByte rxChar;

    IOCTL(SCI, SCI_SET_MODULE,  SCI_ENABLE);
    IOCTL(SCI, SCI_TRANSMITTER, SCI_ENABLE);
    IOCTL(SCI, SCI_RECEIVER,    SCI_ENABLE);

    EnableInterrupts;   /* Authorise all interrupts */

    while(1)
    {
       /* wait for received character */
        while(!IOCTL(SCI, SCI_GET_RX_FULL, NULL));

        sci_tmp = IOCTL(SCI, SCI_GET_STATUS_REG1, NULL);
        rxChar  = IOCTL(SCI, SCI_READ_8BIT_DATA,  NULL);

        if (rxChar != '*')  {
            while(!IOCTL(SCI, SCI_GET_TX_EMPTY, NULL));

            sci_tmp = IOCTL(SCI, SCI_GET_STATUS_REG1, NULL);
            /* send back received character */
            IOCTL(SCI, SCI_WRITE_8BIT_DATA, rxChar);
        }
        else    {
        /* is SCI Transmitter in use ? */
        if (IOCTL(SCI, SCI_GET_TX_EMPTY, NULL)) {
             position = 0;
             /* enable user ISR to send answer */
             IOCTL(SCI, SCI_SET_TX_EMPTY_INT, SCI_ENABLE);

        }
        }
    }
}

void mySciTxISR(void)
{
    UByte sci_tmp;

    if (!(position == (sizeof(idt)/sizeof(idt[0]) - 1)))  {
        /* clear status register and write to data register */
        sci_tmp = IOCTL(SCI, SCI_GET_STATUS_REG1, NULL);
        IOCTL(SCI, SCI_WRITE_8BIT_DATA, idt[position++]);
    }
    else    {
        IOCTL(SCI, SCI_SET_TX_EMPTY_INT, SCI_DISABLE);
    }
}
```

5.11.3.2 Read — Non-Blocking or Blocking Read from SCI Module

Call(s):

```
void read (SCI, NON_BLOCKING,UByte *address, UByte size);
void read (SCI, BLOCKING, UByte *address, UByte size);
```

Arguments:

Table 5-14. Read Function Call Arguments

SCI	in	SCI module identifier
BLOCKING / NON_BLOCKING	in	Specify whether the data is read in blocking or non-blocking mode
UByte *address	in / out	Pointer of the user buffer
UByte size	in	the number of bytes read from SCI to user buffer

Description:

The *read* function is implemented as a macro and calls related *read* functions for non-blocking and blocking mode dependent on the *mode* parameter.

- Non-Blocking Mode

  The *read* function in non-blocking mode initializes some internal status variables and pointers of the SCI driver, modifies the RIE (receiver full interrupt enable) bit as well as the REIE (receive error interrupt enable) bit in the SCI control register and returns immediately to the main program. All characters are sent within the interrupt service routine.

NOTE: *Corresponding interrupt routines must be installed. See* ***Example 17****.*

- Blocking Mode

  The *read* function in blocking mode uses a polling technique, interrupts are not used. The *read* function in blocking mode waits until all characters are read from the SCI module.

Returns: None.

Range Issues: Parameter *size* must greater than 1.

Special Issues:

The *read* function in non-blocking mode is not reentrant.

- Non-Blocking Mode

  If the non-blocking mode of the *read* function is used, then interrupt functions of the SCI driver must be installed to obtain proper functionality of the *read* function. See **Example 17**.

- Blocking Mode — None

Design/Implementation:

The non-blocking and blocking mode of the *read* macro are implemented as a function call.

5.11.3.3 Write — Non-Blocking or Blocking Write to SCI Module

Call(s):

```
void write (SCI, NON_BLOCKING, UByte *address, UByte size);
void write (SCI, BLOCKING, UByte *address, UByte size);
```

Arguments:

Table 5-15. Write Function Call Arguments

SCI	in	SCI module identifier
BLOCKING / NON_BLOCKING	in	Specify whether the data is written in blocking or non-blocking mode
UByte *address	in / out	Pointer of the user buffer
UByte size	in	the number of bytes to write send through SCI

Description:

The *write* function is implemented as a macro and calls related *write* functions for non-blocking and blocking mode, dependent on the *mode* parameter.

- Non-Blocking Mode

  The *write* function in non-blocking mode initializes internal status variables and pointers of the SCI driver, modifies the TEIE

(transmitter empty interrupt enable) bit in the SCI control register and returns immediately to the main program. All characters are received within the interrupt service routines.

- Blocking Mode

  The *write* function in blocking mode uses a polling technique, interrupts are not used. The *write* function in blocking mode waits until all characters are written to the SCI.

Returns: None

Range Issues: Parameter *size* must greater than 1.

Special Issues:

The *write* function in non-blocking mode is not re-entrant.

- Non-Blocking Mode

  If the non-blocking mode of the *write* function is used, then interrupt functions of the SCI driver must be installed to obtain proper functionality of the *write* function. See **Example 17**.

- Blocking Mode — None

Design/Implementation:

The non-blocking and blocking mode of the *write* macro are implemented as a function call.

Example 17. Read/Write Function

```
/*****************************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: appconfig.h
*
* Description: application configuration file
*
* Modules Included:  None
*
*****************************************************************************/

#ifndef __APPCONFIG_H
    #define __APPCONFIG_H

/*****************************************************************************
* Include needed driver initialization routines
*****************************************************************************/

#define   INCLUDE_PLL           /* PLL support */
#define   INCLUDE_SCI           /* SCI support */

/* Specify the your Xtal clock frequency                                    */
#define XTAL_CLOCK              8000000L

/*****************************************************************************/
/*                          PLL Initialization                               */
/*****************************************************************************/
/* BUS_CLOCK  = (XTAL_CLOCK * PLL_FREQUENCY_MUL) / 4                        */

/* PLL Control Register 1                                   (PCTL)          */
#define PLL_ON_BIT              PLL_ON                  /* PLL_ON / PLL_OFF     */
#define PLL_BASE_CLOCK          PLL_CGMVCLK             /* PLL_CGMXCLK /
                                                           PLL_CGMVCLK          */
/*.........................................................................*/
/* PLL Programing Register 1                                (PPG)           */
#define PLL_FREQUENCY_MUL       PLL_MUL4                /* PLL_MUL1..PLL_MUL15  */
#define PLL_VCO_FREQUENCY_MUL   PLL_MUL3                /* PLL_MUL1..PLL_MUL15  */
/*.........................................................................*/
/* PLL Control Register 2                                   (PBWC)          */
#define PLL_BANDWIDTH           PLL_AUTOMATIC           /* PLL_MANUAL /
                                                           PLL_AUTOMATIC        */

/*****************************************************************************/
/*                          Watch dog initialization                         */
/*****************************************************************************/
/* Config Register   !!! Write-Once-Register !!!            (CONFIG)        */
#define WDO_COPD                WDO_DISABLE             /* WDO_ENABLE /
                                                           WDO_DISABLE          */
```

```
/*****************************************************************************/
/*                           SCI Initialization                              */
/*****************************************************************************/

#define SCI_BAUD_RATE          9600

#define SCI_DATA_FORMAT        SCI_8BIT_NONE         /* SCI_8BIT_NONE /
                                                        SCI_7BIT_ODD  /
                                                        SCI_7BIT_EVEN /
                                                        SCI_8BIT_EVEN /
                                                        SCI_8BIT_ODD  /
                                                        SCI_9BIT_NONE        */

#define SCI_WAKEUP_COND        SCI_WAKE_BY_IDLE      /* SCI_WAKE_BY_IDLE /
                                                        SCI_WAKE_BY_ADDRESS  */

#define SCI_DATA_POLARITY      SCI_NOT_INVERTED      /* SCI_NOT_INVERTED /
                                                        SCI_INVERTED         */

#define SCI_IDLE_LINE          SCI_AFTER_STOP        /* SCI_AFTER_START /
                                                        SCI_AFTER_STOP       */

#define SCI_LOOP_MODE          SCI_DISABLE           /* SCI_DISABLE /
                                                        SCI_ENABLE           */

/* Assign the SCI callbacks for use in read / write functions */
#define INT_SCI_TX_CALLBACK_1 SciTxEmptyISR
#define INT_SCI_RX_CALLBACK_1 SciRxFullISR
#define INT_SCI_ERR_CALLBACK_1 SciRxErrorISR

#endif /* __APPCONFIG_H*/


/*****************************************************************************
*
* Motorola Inc.
* (c) Copyright 2001 Motorola, Inc.
* ALL RIGHTS RESERVED.
*
*****************************************************************************
*
* File Name: sci_demo1.c
*
* Description: Example of usage read / write commands.
*
* Modules Included: < main >
*
*****************************************************************************/

#include "types.h"
#include "arch.h"
#include "hidef.h"
#include "sys.h"
#include "appconfig.h"

#include "plldrv.h"
#include "scidrv.h"

/* global variables */
volatile UByte    buffer[10];
```

```
void main (void)
{
/* enable all maskable interrupts */
      EnableInterrupts;

/* read 10 characters in blocking mode */
read(SCI,BLOCKING, buffer, 10);
/* send back received characters in blocking mode */
write(SCI, BLOCKING, buffer, 10);

/* read 10 characters in non-blocking mode */
read(SCI, NON_BLOCKING, buffer, 10);

/* test if the characters are already received */
while(ioctl(SCI, SCI_GET_STATUS, NULL) & SCI_STATUS_READ_INPROGRESS);

      * send back received characters in non-blocking mode */
write(SCI, NON_BLOCKING, buffer, 10);

while(1);
}
```

This code:

- Installs receiver full, receiver error, and transmitter empty interrupts for the SCI module
- Initializes the PLL and SCI module

First, 10 characters are received and sent back to show how to use the *read* and *write* functions in blocking mode, then 10 characters are received and sent back again to show the usage of the *read* and *write* function in non-blocking mode.

5.12 SCI Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.12.1 Debug Strobes

Debug strobes allow the measuring of the interrupt duration on the user specified port and pin utilizing scope. At the beginning of the interrupt the strobe signal is asserted and when finished it is deasserted. To specify the debug strobes, the following code has to be added into **appconfig.h**.

Debug Strobe Port and Pin Specification:

```
#define INT_SCI_TX_STROBE_PORT      PORT
#define INT_SCI_TX_STROBE_PIN       Pin Number
#define INT_SCI_RX_STROBE_PORT      PORT
#define INT_SCI_RX_STROBE_PIN       Pin Number
#define INT_SCI_ERR_STROBE_PORT     PORT
#define INT_SCI_ERR_STROBE_PIN      Pin Number
```

The following code is an example of setting the debug strobes for SCI Tx interrupts on port A pin 4.

```
#define INT_SCI_TX_STROBE_PORT      A
#define INT_SCI_TX_STROBE_PIN       4
```

5.12.2 Debug Mode

The debug mode helps the user to find the unhandled interrupts. If INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and an unhandled interrupt occurs, the program will run in an endless loop.

Example:

```
#define INT_DEBUG_MODE    TRUE
```

5.12.3 User Callbacks

The user can define two different types of callbacks for execution in his own interrupt code.

This definition **appconfig.h** installs the user callback function ***function_name_1*** before the SDK routine and SDK flag service.

```
#define INT_SCI_TX_CALLBACK_1 function_name_1
```

This definition in *a***ppconfig.h** installs the user callback function ***function_name_2*** after the SDK routine and SDK flag service.

```
#define INT_SCI_TX_CALLBACK_2 function_name_2
```

5.13 Port Drivers

The port driver performs both the statical configuration of the input/output (I/O) ports and LOCTL commands for controlling the peripheral.

Before main is called the ports direction and initial state are set according to the user settings in **appconfig.h**, which overwrites the default configuration of the registers.

Initialization contains two parts:

1. Setting of the registers with default values defined after reset. Initialization sets the registers according to **appconfig.h** if the set value differs from the default reset state.
2. Setting of the registers with not defined reset value. Initialization sets the registers according to **appconfig.h**.

Commands for peripheral control are performed by:

1. Function — Usually more general, all parameters are passed through variables.
2. Macro — The code is usually shorter, but depends on compilation and type used. In some cases, macros can support only constant parameters.

5.13.1 API Definition

Required Files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "spidrv.h"
```

NOTE: *The included files must be kept in order.*

5.13.2 Static Initialization

Call(s:)

```
void portInit(void);
```

Description:

The portInit function sets PORT peripheral module. The required parameters for peripheral module configuration are defined in **appconfig.h** (see **Table 5-16**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: None

Example 18. Static Initialization of the Port Driver

```
Next definition have to be in appconfig.h

/* Modules for Static Configuration */
#define INCLUDE_PORT
/*************************************************************************
*          PORT Initialization
*************************************************************************/
/* SPI Control Register       (SPCR)
#define PORTA_DIRECTION    PORT_OUT0|PORT_OUT5     /* PORT_OUT0|..|PORT_OUT7 */
#define PORTA_DATA         PORT_PIN5               /* PORT_PIN0|..|PORT_PIN7 */
```

NOTE: *The previous definition determines the setting of pin 0 and pin 5 on port A as output. Pin 5 is set to level H and pin 0 is set to level L. The function portInit(), where the required setting occurs, is called automatically before main if INCLUDE_PORT is defined in* ***appconfig.h****.*

Table 5-16. Port Driver Constants Definition

Constant Definition	Parameters[1]	Description	Notes[2]
PORTA_DATA	PORT_PIN0|PORT_PIN1| PORT_PIN2|PORT_PIN3| PORT_PIN4|PORT_PIN5| PORT_PIN6|PORT_PIN7|	PTA = 0..0xFF	u
PORTA_DIRECTION	PORT_OUT0|PORT_OUT1| PORT_OUT2|PORT_OUT3| PORT_OUT4|PORT_OUT5| PORT_OUT6|PORT_OUT7, PORT_OUT, PORT_IN	DDRA = 0..0xFF	d
PORTB_DATA	PORT_PIN0|PORT_PIN1| PORT_PIN2|PORT_PIN3| PORT_PIN4|PORT_PIN5| PORT_PIN6|PORT_PIN7|	PTB = 0..0xFF	u
PORTB_DIRECTION	PORT_OUT0|PORT_OUT1| PORT_OUT2|PORT_OUT3| PORT_OUT4|PORT_OUT5| PORT_OUT6|PORT_OUT7, PORT_OUT, PORT_IN	DDRB = 0..0xFF	d
PORTC_DATA	PORT_PIN0|PORT_PIN1| PORT_PIN2|PORT_PIN3| PORT_PIN4|PORT_PIN5| PORT_PIN6	PTC = 0..0x7F	u
PORTC_DIRECTION	PORT_OUT0|PORT_OUT1| PORT_OUT2|PORT_OUT3| PORT_OUT4|PORT_OUT5| PORT_OUT6, PORT_OUT, PORT_IN	DDRC = 0..0x7F	d
PORTE_DATA	PORT_PIN0|PORT_PIN1| PORT_PIN2|PORT_PIN3| PORT_PIN4|PORT_PIN5| PORT_PIN6|PORT_PIN7	PTE = 0..0xFF	u
PORTE_DIRECTION	PORT_OUT0|PORT_OUT1| PORT_OUT2|PORT_OUT3| PORT_OUT4|PORT_OUT5| PORT_OUT6|PORT_OUT7, PORT_OUT, PORT_IN	DDRE = 0..0xFF	d
PORTF_DATA	PORT_PIN0|PORT_PIN1| PORT_PIN2|PORT_PIN3| PORT_PIN4|PORT_PIN5|	PTC = 0..0x3F	u
PORTF_DIRECTION	PORT_OUT0|PORT_OUT1| PORT_OUT2|PORT_OUT3| PORT_OUT4|PORT_OUT5, PORT_OUT, PORT_IN	DDRC = 0..0x3F	d

1. First item in the parameters column is the default.
2. Configuration in **appconfig.h**:
   d — parameter with defined default reset state
   u — parameter with undefined default reset state
   o — parameter with write-once register

5.13.3 Input/Output Control (IOCTL)

Call(s:)

```
return type IOCTL (PORTx, command, parameters);
```

Description:

A MACRO for operation with a PORT peripheral register is called according to the command and value parameters.

Returns:

Return type (depend on required command)

Global Data: None

Arguments:

command — specifies targets which have to be addressed

parameters — depends on required command, for more information see **Table 5-16**

Range Issues: None

Special Issues: None

Example:

```
IOCTL(PORTA, PORT_SET_PINS, PORT_PIN1); /* Sets pin1 on the
PORT A to level H*/
```

5.13.4 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: ***Inout*** *parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
void   IOCTL (module, command, parameters);
UByte  IOCTL (module, command, parameters);
UWord16IOCTL (module, command, parameters);
void   IOCTL (module, command, *parameters);
UByte  IOCTL (module, command, *parameters);
UWord16IOCTL (module, command, *parameters);
```

Description:

A MACRO for operation with a peripheral register is called according to the command and value parameters.

Example:

The following command sets bit CPOL (clock polarity) in the register SPCR to 1.

```
IOCTL(SPI, SPI_CLOCK_POLARITY, SPI_POSITIVE);
```

Arguments:

module (*in)* — specify module, in this case the module can be: PORTA, PORTB, PORTC, PORTD, PORTE, or PORTF

command (*in)* — specify target which have be addressed

parameters (*in, inout, out)* — data passed to the IOCTL macro function

All commands and appropriate parameters are shown in **Table 5-17**. The table uses these conventions:

- Items separators
  - / only one of the specified items is allowed
  - | consolidation of items is allowed (item1|item2|item4)
  - & intersection of items is allowed (item1&item2&item3)
- Implementation
  - f = function
  - M = macro

Table 5-17. Port Driver Macros and Functions Commands (Sheet 1 of 2)

Command	Parameters[1]			Description	Notes (2)
PORT_INIT	NULL			Static configuration according appconfig return 0	f
PORT_WRITE_DATA Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	Ubyte	in	<0..0xFF>	PTA = 0..0xFF	M
SPI_GET_CONTROL_REG	NULL			return PTA (UByte)	M
PORT_SET_PINS Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			PTx = PTx | BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M[3], [4]
PORT_CLEAR_PINS Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			PTx = PTx & ~BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M[3], [4]
PORT_TOGGLE_PINS Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			PTx = PTx ^BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M[3]
PORT_GET_PINS Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			return (PTx & BITy)&&BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M[5]
PORT_SET_PIN0 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx0 = 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN1 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx1 = 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN2 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx2 = 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN3 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx3 = 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN4 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx04= 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN5 Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_L / PORT_H			PTx_PTx0 = 0 / 1 where x is A,B,C,E or F	M
PORT_SET_PIN6 Port identifier: PORTA, PORTB, PORTC, PORTE	PORT_L / PORT_H			PTx_PTx0 = 0 / 1 where x is A,B or E	M

Table 5-17. Port Driver Macros and Functions Commands (Sheet 2 of 2)

Command	Parameters[1]			Description	Notes (2)
PORT_SET_PIN7 Port identifier: PORTA, PORTB, PORTE,	PORT_L / PORT_H			PTx_PTx0 = 0 / 1 where x is A,B,C or E	M
PORT_WRITE_DIR Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	Ubyte	in	<0..0xFF>	DDRx = 0..0xFF	M
PORT_SET_DIR_OUT Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			DDRx = DDRx | BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M(4)
PORT_SET_DIR_IN Port identifier: PORTA, PORTB, PORTC, PORTE, PORTF	PORT_PIN0 | PORT_PIN1 | PORT_PIN2 | PORT_PIN3 | PORT_PIN4 | PORT_PIN5 | PORT_PIN6 | PORT_PIN7|			DDRx = DDRx & ~BITy where x is A,B,C,E or F and y is 0|1|2|3|4|5|6|7	M(4)

1. First item in the parameters column is the default.
2. f = function
   M = macro
3. When the parameter contains more than one pin the hole register is read and rewrite. It means that if some pins are defined as inputs, than its actual state is read and write to the output register. It can change state of port if the port direction is changed. User also have to make sure that no interrupt can occur, where the same register is affected.
4. If parameter is constant and contains less than 4 pins, then using more commands with one pin setting is recommended. The pins will not be set at the same time, but total time will be lower and the command will not have any affect to other pins
5. The commands return true if one or more of selected pins are at level H, else return false. If only one pins selected the command return state of this pin.

5.14 WDO Driver

The WDO driver performs both the initial configuration during startup and IOCTL commands for controlling the peripheral module.

The initial configuration allows the disabling of WDO by defining the WDO_COPD WDO_DISABLE in **appconfig.h**. The commands for peripheral module control are performed by macros.

5.14.1 API Definition

Required files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "wdodrv.h"
```

NOTE: *The included files must be kept in order.*

The static initialization of WDO allows disabling the COP by setting the COPD in the CONFIG register. The CONFIG register is writing in PreMainInit(), which is located in config.c. The PreMainInit() is called automatically before main().

Example 19. Static Initialization of the WDO Driver

```
Next definition have to be in appconfig.h

/*****************************************************************************
*       COP Initialization
*****************************************************************************/
/* !!!!!!!! Write-Once-Register !!!!!!!! */
/* Config Register         (CONFIG)
#define WDO_COPD    WDO_DISABLE         /* WDO_ENABLE/WDO_DISABLE */
```

NOTE: *The previous definition disables watchdog during CPU initialization.*

5.15 Analog-to-Digital Converter (ADC) Driver

This section describes the API for the 68HC908MRxx analog-to-digital converter (ADC) on-chip module. The functionality of the ADC module itself is described in the specific device data sheet.

The ADC driver is dedicated to controlling the on-chip ADC module. It comprises an initialization routine for statical configuration refer to of the module data sheet and to the IOCTL commands for controlling the module during run-time.

IOCTL commands for controlling the ADC module are implemented as:

1. Macros (in-line code — short code)
2. Functions (called function — longer code)

NOTE: *Some commands are implemented both as macros and as functions. In this case, the implementation which best suits an application can be chosen.*

5.15.1 API Definition

Required Header File(s):

These header files are needed in order to use the ADC driver.

```
#include "sys.h"
#include "arch.h"
#include "adcdrv.h"
#include "periph.h"
#include "appconfig.h"
#include "config.h"
```

5.15.2 Configuration Items

The symbols used in macro definitions for the static configuration of ADC module are summarized here. Initialization is performed by the *adcInit*() function during the initialization process.

The statical configuration routine *adcInit()* sets the ADC module according to the user setting specified in ***appconfig.h*** which overwrites the default configuration of the registers.

Initialization process, divided into three parts:

1. Setting of the write-once registers (performed in the premain function).
2. Setting of the registers with defined reset states.
   - The function *adcInit()* sets the ADC registers according to the application statical configuration file **appconfig.h** if the user setting differs from the default reset state of the register.
   - Refer to table **Table 5-10** with configurable items.
3. Setting of the registers with undefined reset state.
   - The function *adcInit()* set the ADC registers according to the ***appconfig.h*** file.
   - Refer to table **Table 5-18** for configurable items.

See **Example 16** and **Example 21** for more details.

Code to initialize the ADC driver is automatically included in the SDK project by inserting the following line into **appconfig.h**:

```
#define INCLUDE_ADC    /* includes ADC code to application */
```

Items Separators:

/ only one of the specified items is allowed

| consolidation of items is allowed (item1|item2|item4)

& intersection of items is allowed (item1&item2&item3)

CAPITAL = constant (*in*)

Table 5-18. Configuration Items for appconfog.h

Configurable Items	Parameters	Description
ADC_INPUT_CLOCK	/ ADC_CGMXCLK / ADC_BUS_CLK	Set the ADC clock source write to ADICLK bit of the ADC Clock Register (ADCLK) during initialization
ADC_CLOCK_PRESCALER	/ ADC_CLK_DIV_1 / ADC_CLK_DIV_2 / ADC_CLK_DIV_4 / ADC_CLK_DIV_8 / ADC_CLK_DIV_16	Set ADC prescaler in the ADC Clock Register (ADCLK) to derive the internal ADC clock during initialization
ADC_RESULT_MODE	/ ADC_TRUNCATE_8BIT / ADC_JUSTIFY_RIGHT / ADC_JUSTIFY_LEFT / ADC_JUSTIFY_LEFT_SIGN	Set the result justification mode in the ADC Clock Register (ADCLK) during initialization
ADC_CONVERSION	/ ADC_SINGLE (1) / ADC_CONTINUOUS (2)	Select the type of conversion (1) after conversion waits for new ADC start (2) ADC works in loop mode — ISR at the end of conversion is not generated
ADC_INT	/ ADC_DISABLE / ADC_ENABLE	Enable or disable interrupt after end of conversion
ADC_COMPLETE_CALLBACK	Pointer to the user callback function	Create the statical binding to the used conversion complete function
ADC_ENABLE_SCAN_CHANNELS	NONE	Enable buffered mode of ADC device when ADC converts the channels specified in ADC_CHANNEL_LIST and store it to ADC buffer. After finishing the conversion the ADC_COMPLETE_CALLBACK is generated to enable user to process the converted data from the buffer.
ADC_CHANNEL_LIST	Pointer to the user channel list	Contains the list of channels to be converted (statical binding) if the switching of the channel lists is required during run time, do not define this item in **appconfig.h**
ADC_BUFFER_SIZE	UByte Value	Specifies the number of items to be stored in buffer
ADC_SAMPLE_TYPE	/ UByte / UWord16 / SByte / SWord16	ADC_SAMPLE_TYPE specifies the size of items stored in buffer

5.15.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

1. *in* argument means that the parameter value is an input only to the function
2. *out* argument means that the parameter value is an output only from the function.
3. *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Item Separators:

/ only one of the specified items is allowed

| consolidation of items is allowed (item1|item2|item4)

& intersection of items is allowed (item1&item2&item3)

CAPITAL = constant (*in*)

First item or items in the column is default

Implemented as:

f = function

M = macro

5.15.3.1 ADC — Non-Buffered Mode

See **Table 5-19** for the ADC input/output control commands in on-buffered mode.

Table 5-19. ADC Input/Output Control Commands Non-Buffered Mode (Sheet 1 of 2)

Command	Parameters[1]	Description	Notes[2]
ADC_WRITE_ADCLK	UByte	Write UByte value to the ADC Clock Register (ADCLK)	M
ADC_GET_ADCLK	NULL	Get the content of the ADC Clock Register (ADCLK)	M
ADC_SELECT_CLOCK	/ ADC_CGMXCLK / ADC_BUS_CLK	Select the ADC clock source write to ADICLK bit of the ADC Clock Register (ADCLK)	M
ADC_SET_PRESCALER	/ ADC_CLK_DIV_1 / ADC_CLK_DIV_2 / ADC_CLK_DIV_4 / ADC_CLK_DIV_8 / ADC_CLK_DIV_16	Set ADC prescaler in the ADC Clock Register (ADCLK) to derive the internal ADC clock	M
ADC_GET_PRESCALER	NULL	Get ADC prescaler from the ADC Clock Register (ADCLK)	M
ADC_SET_MODE	/ ADC_TRUNCATE_8BIT / ADC_JUSTIFY_RIGHT / ADC_JUSTIFY_LEFT / ADC_JUSTIFY_LEFT_SIGN	Set the result justification mode in the ADC Clock Register (ADCLK)	M
ADC_GET_MODE	NULL	Get the current result justification mode from the ADC Clock Register (ADCLK)	M
ADC_WRITE_ADSCR	UByte	Write UByte value to the ADC Status and Control Register (ADSCR)	M
ADC_GET_ADSCR	NULL	Get the content of the ADC Status and Control Register (ADSCR)	M

Table 5-19. ADC Input/Output Control Commands Non-Buffered Mode (Sheet 2 of 2)

Command	Parameters[1]	Description	Notes[2]
ADC_START (1) ADC_START_ID (2) ADC_START_IE (3)	/ ADC_ATD0 / ADC_ATD1 / ADC_ATD2 / ADC_ATD3 / ADC_ATD4 / ADC_ATD5 / ADC_ATD6 / ADC_ATD7 / ADC_ATD8 / ADC_ATD9 / ADC_VREFH / ADC_VREFL / ADC_POWER_OFF (4)	Start the conversion of selected channel by write to the ADC Status and Control Register (ADSCR) (1) without affecting the Interrupt Enable bit (AIEN) (2) with disabling the ADC Interrupt (AIEN = 0) (3) with enabling the ADC Interrupt (AIEN = 1) (4) One conversion cycle is required to recover from disable state	M
ADC_ENABLE_ISR	NULL	Enable ADC interrupt at the end of ADC conversion in the ADC Status and Control Register (ADSCR) (AIEN = 1)	M
ADC_DISABLE_ISR	NULL	Disable ADC interrupt at the end of ADC conversion in the ADC Status and Control Register (ADSCR) (AIEN = 0)	M
ADC_GET_CONVERSION_COMPLETE	NULL	Get the status of current conversion in the ADC Status and Control Register (ADSCR)	M
ADC_GET_RESULT8	NULL	Get the 8 bit result from the ADC Data Register (ADR) Note: Macro implementation works only for ADC_TRUNCATE_8BIT	M, f
ADC_GET_RESULT16	NULL	Get the 16 bit result from the ADC Data Register (ADR)	M, f

1. First item in the parameters column is the default.
2. f = function
   M = macro

Example 20. Example of Using ADC in Non-Buffered Mode

```
/*************************************************************************
      Statical ADC Initialization in appconfig.h
*************************************************************************/

#define ADC_INT                  ADC_DISABLE
#define ADC_CONVERSION           ADC_SINGLE

#define ADC_INPUT_CLOCK          ADC_BUS_CLK
#define ADC_CLOCK_PRESCALER      ADC_CLK_DIV_8

#define ADC_RESULT_MODE          ADC_JUSTIFY_LEFT


/*************************************************************************
      Application program
*************************************************************************/

#include "sys.h"
#include "arch.h"
#include "adcdrv.h"
#include "periph.h"
#include "appconfig.h"
#include "config.h"

static volatile SWord16          channel1;

void main (void)
{

      while (1)
      {
            /* Start conversion of the channel 0 with disabled interrupt */
            IOCTL(ADC, ADC_START_ID, ADC_ATD0);

            /* Wait until conversion of the Channel 0 is completed */
            while(!IOCTL(ADC, ADC_CONVERSION_COMPLETE, NULL));

            /* Get the result of conversion from ADC Data Registers
            ADRH and ADRL and store it to channel1 */
            channel1 = IOCTL(ADC, ADC_GET_RESULT16, NULL);
      }
}
```

5.15.3.2 ADC — Buffered Mode

In this mode, the ADC driver samples data according to the user specified channel list and pools them to the ADC software buffer. At the end of the channel list conversion the user defined callback is called to allow processing of the converted data. Refer to **Table 5-18** for a list of configuration items and **Table 5-20** for input/output control commands.

Table 5-20. ADC Input/Output Control Commands — Buffered Mode

Command	Parameters(1)	Description	Notes(2)
ADC_SET_CHANNEL_LIST	Pointer to user channel list	Set the ADC driver to use the specified channel list for conversion	f
ADC_SCAN_CHANNELS	NULL	Starts the conversion of specified channel list.	M
ADC_GET_SAMPLE	UByte	Returns the sample specified by the parameter.	M
ADC_GET_SCAN_IN_PROGRESS	NULL	Return the state of buffered conversion. If scanning of samples from channel list is finished returns 1 otherwise 0	M

1. First item in the parameters column is the default.
2. f = function
   M = macro

Example 21. Example of Using ADC in Buffered Mode

```
/*************************************************************************
      Statical ADC Initialization in appconfig.h
*************************************************************************/

#define ADC_INT                 ADC_DISABLE
#define ADC_CONVERSION          ADC_SINGLE

#define ADC_INPUT_CLOCK         ADC_BUS_CLK
#define ADC_CLOCK_PRESCALER     ADC_CLK_DIV_8

#define ADC_RESULT_MODE         ADC_JUSTIFY_LEFT

#define ADC_ENABLE_SCAN_CHANNELS
#define ADC_SAMPLE_TYPE         SWord16
#define ADC_BUFFER_SIZE         3

#define ADC_CHANNEL_LIST        adcChannelList
#define ADC_COMPLETE_CALLBACK   AdcCompleteCallback

/*************************************************************************
      Application program
*************************************************************************/

#include "sys.h"
#include "arch.h"
#include "adcdrv.h"
#include "periph.h"
#include "appconfig.h"
#include "config.h"


static volatile SByte           channel1;
static volatile SByte           channel2;
static volatile SWord16         channel3;

#pragma CONST_SEG CONST_ROM
const  UByte adcChannelList[ ] = {
ADC_ATD0,           /* channel 0 */
ADC_ATD1,           /* channel 1 */
ADC_ATD2,           /* channel 3 */
};
#pragma CONST_SEG DEFAULT

/* function prototype */
void AdcCompleteCallback(void);
```

```
void main (void)
{
      /* start the conversion of adcChannelList */
      IOCTL(ADC, ADC_SCAN_CHANNELS, NULL);
}

/* Conversion complete function */
void AdcCompleteCallback(void)
{
      /* Read Sample 0 (Channel0) from sample buffer */
      Channel1 = (SByte)(IOCTL(ADC, ADC_GET_SAMPLE, 0)>>8);

      /* Read Sample 1(Channel0) from sample buffer */
      Channel2 = (SByte)(IOCTL(ADC, ADC_GET_SAMPLE, 1)>>8);

      /* Read Sample 1(Channel0) from sample buffer */
      Channel3 = (SWord16)IOCTL(ADC, ADC_GET_SAMPLE, 2);
}
```

5.16 ADC Interrupt Handling

Refer to **3.6 Interrupts and Interrupt Service Routines** for a detailed description of interrupt handling.

5.16.1 Debug Strobes

Debug strobes allows the measuring of the interrupt duration on the user specified GPIO port and pin utilizing scope. At the beginning of the interrupt the strobe signal is asserted and when finished it is deasserted. To specify the debug strobes, the following code has to be added into ***appconfig.h*** file.

Debug Strobe Port and Pin Specification:

```
#define INT_ADC_DEBUG_PORT      PORT
#define INT_ADC_DEBUG_PIN       Pin Number

example of setting the debug strobe signal on PortA - pin4:
#define INT_ADC_DEBUG_PORT      A
#define INT_ADC_DEBUG_PIN       4
```

5.16.2 Debug Mode

The debug mode helps the user to find the unhandled interrupts. If the INTERRUPT_DEBUG_MODE is defined in **appconfig.h** and an unhandled interrupt occurs, the program will run an endless loop.

```
#define INT_DEBUG_MODE
```

5.16.3 User Callbacks

User can define two different types of callbacks for executing in his own interrupt code.

1. This definition in **appconfig.h** file installs the user callback function ***function_name_1*** before the SDK routine and SDK reload flag service.

```
#define INT_ADC_CALLBACK_1 function_name_1
```

2. This definition in ***appconfig.h*** file installs the user callback function ***function_name_2*** after the SDK routine and SDK reload flag service.

```
#define INT_ADC_CALLBACK_2 function_name_2
```

Section 6. Off-Chip Drivers

6.1 Contents

6.2 Introduction

One strength of the 8-bit SDK is that it provides a high degree of architectural and hardware independence for the application code. This portability is achieved by the 8-bit SDK modular design which, in this case, isolates external periphery into a set of defined, tested, and documented application programming interfaces (APIs).

This section describes the APIs for off-chip drivers, forming the interface between hardware and application software. All drivers for external periphery allow pin assignments which, provides the drivers universal usage.

6.3 Light-Emitting Diode (LED) Driver

The light-emitting diode (LED) driver performs the standard LED functions (e.g., LED on, LED off, and blinking). The user can assign a pin and polarity for each LED.

6.3.1 API Definition

Required Files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "portdrv.h"
#include "leddrv.h"
```

NOTE: *The included files must be kept in order.*

6.3.2 Static Initialization

Call(s:)

```
SByte ledInit(void);
```

Description:

The ledInit function sets port direction as an output for all pins assigned to LED's. The required pin and LED polarity assignment are defined in **appconfig.h** (see **Table 6-1**).

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: None

Example 22. Static Initialization of the LED Driver

```
Following definition have to be in appconfig.h to assign the LED name to specified I/O
port and pin.

/* Modules for Static Configuration */
#define INCLUDE_LED
/*****************************************************************************
*      LED Initialization
*****************************************************************************/
#define LED_RED                  C_PTC6
#define SET_LED_RED_POLARITY     LED_POSITIVE /* LED_POSITIVE / LED_NEGATIVE */
...
#define LED_MASK_PORTC           BIT4|BIT5|BIT6
```

NOTE: *The previous definition assigns pins 4, 5, and 6 of port C for LED and the LED name for pin 6.*

Table 6-1. LED Driver Constants Definition

Constant Name	Value	Description
LED_FLASHING	0..255	Number of calling function RefreshLed() between toggle of flashing LED's
LED_LEDName	PTA_PTA0 .. PTF_PTF5	Pin assignment example: #define LED_GREEN PTC_PTC6
SET_LEDName_POLARITY	LED_POSITIVE / LED_NEGATIVE	Polarity of LED definition example: #define SET_LED_GREEN_POLARITY LED_POSITIVE
LED_MASK_PORTx x=A,B,C,D,E,F	BSET0|BSET1|...|BSET7	Mask of pins used for LED at the port example: #define LED_MASK_PORTC BSET4|BSET5|BSET6

6.3.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
LED_GET_STATE(led)
LED_SET_STATE(led, value)
LED_SET_ON(led)
LED_SET_OFF(led)
LED_TOGGLE(led)
LED_SET_FLASHING(led)
LED_CLEAR_FLASHING(led)
void ledRefresh(UByte LedFlashing);
```

Description:

All commands and appropriate parameters are given in **Table 6-2**. The table uses the following conventions:

- Implementation
  f — function
  M — Macro

Table 6-2. LED Driver Macros and Functions

Function / MACRO	Parameters	Description (Value in Hex)	Notes[1]
Sbyte ledInit()	None	DDRx = DDRx|LED_MASK_PORTx	f
void ledRefresh(UByte ledFlashing)	ledFlashing — perioda of LED's flashing. The LED_FLASHING defined in **appconfig.h** can be used	read actual state of all ports assigned to a switch have to be called in a timer interrupt.	f
LED_OFF	LEDName (User pin name) (Const)	PTxPINy = LED_POLARITY	M
LED_ON	LEDName (User pin name) (Const)	PTxPINy = !LED_POLARITY	M
LED_TOGGLE	LEDName (User pin name) (Const)	PTxPINy = !PTx_PINy	M
LED_SET_FLASHING	LEDName (User pin name) (Const)	Relevant bit in LedFlash=1; Switch on the LED	M
LED_CLEAR_FLASHING	LEDName (User pin name) (Const)	Relevant bit in LedFlash=0; Switch off the LED	M
LED_SET_STATE (LEDName, NewState) example: LED_SET_STATE (LED_GREEN, _ON);	1) LEDName (User pin name) (Const) 2) value: _ON/ _OFF/ (Ubyte, in)	PTxPINy = !LED_POLARITY PTxPINy = LED_POLARITY	M[2], [3], [4]
return LED_GET_STATE (LEDName)	LEDName (User pin name)	return PTxPINy ^ LED_POLARITY	M[4]

1. f = function
   M = macro
2. Users have to make sure that no interrupt, where the same register is affected, can occur during this command
3. The port is read before writing, that means that if some pins are defined as inputs its actual state is read and written to the output register. It can change port state if the port direction is changed.
4. Name and polarity of the LED must be defined (see **6.3.2 Static Initialization**).

6.3.4 Functional Description

Call(s:)

```
void ledRefresh (void)
```

Description:

The function toggles all flashing LED's every ledFlashing call of the function.

Return: None

Range Issues: None

Special Issues: None

NOTE: *In **Example 23** the LED_GREEN will be toggled every 200 ms (20 function calls).*

Example 23. Static Initialization of the LED Driver

```
/* appconfig.h */

/* LED port assignment */
#define LED_RED                         C_PTC6
#define SET_LED_RED_POLARITY            LED_POSITIVE
#define LED_GREEN                       C_PTC4
#define SET_LED_GREEN_POLARITY          LED_POSITIVE

#define LED_MASK_PORTC                  BIT4|BIT6

#define LED_FLASHING 20  /* Periode of led flashing */

/* main.c */
void main(void)
{

      LED_SET_FLASHING(LED_GREEN) /* Set the geen led as flashing led */

}
void IsrTimerB_Overflow(void)
{     /* 100Hz ISR */


      ledRefresh(LED_FLASHING);

}
```

6.4 Switch Driver

The switch driver performs standard switch controlling. The user can assign pin and polarity for each switch.

6.4.1 API Definition

Required files:

```
#include "types.h"
#include "sys.h"
#include "arch.h"
#include "appconfig.h"
#include "config.h"
#include "portdrv.h"
#include "switchdrv.h"
```

NOTE: *The included files must be kept in order.*

6.4.2 Static Initialization

Call(s:)

```
SByte switchInit(void);
```

Description:

The switchInit function fill switch state structure by constants defined in **appconfig.h**. In **appconfig.h**, the required pin, switch polarity assignment and switch debounce are defined. See **Table 6-1**.

Returns: 0

Global Data: None

Arguments: None

Range Issues: None

Special Issues: None

Example 24. Static Initialization of the Switch Driver

The following definition has to be in **appconfig.h** in order to assign the switches to a specific I/O port and pin.

NOTE: *This definition assigns pins 4 and 5 of port A for switch and name of switch.*

```
/* Modules for Static Configuration */
#define INCLUDE_SWITCH
/*****************************************************************************
*                    LED Initialization
*****************************************************************************/
#define SWITCH_REV_FWD                          SWITCH_PTA4
#define SET_SWITCH_REV_FWD_POLARITY             SWITCH_POSITIVE
#define SWITCH_START_STOP                       SWITCH_PTA5
#define SET_SWITCH_START_STOP_POLARITY          SWITCH_POSITIVE
#define SWITCH_MASK_PORTA                       BIT4|BIT5

#define SWITCH_DEBOUNCE 5                       /*Filter for Switch ports*/
```

Table 6-3. Switch Driver Constants Definition

Constant Name	Value	Description
SWITCH_DEBOUNCE	0..255	Number of unchanged pin states for accepting new state
SWITCH_*SwitchName* The *SwitchName* can be arbitrary name defined by user	SWITCH_PTA0 .. SWITCH_PTF5	bit assignment example: #define SWITCH_START_STOP SWITCH_PTA5
SET_*SwitchName*_POLARITY The *SwitchName* can be arbitrary name defined by user	SWITCH_POSITIVE / SWITCH_NEGATIVE	polarity of switch definition example: #define SET_SWITCH_START_STOP_POLARITY SWITCH_POSITIVE
SWITCH_MASK_PORTx x=A,B,C,D,E,F	BSET0|BSET1|...|BSET7	Mask of pins used for switches at the PORT example: #define SWITCH_MASK_PORTA BSET4|BSET5

6.4.3 API Specification

Function arguments for each routine are described as *in*, *out*, or *inout*.

- *in* argument means that the parameter value is an input only to the function
- *out* argument means that the parameter value is an output only from the function.
- *inout* argument means that a parameter value is an input to the function, but the same parameter is also an output from the function.

NOTE: *Inout parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its results within that data structure. The actual value of the inout pointer parameter is not changed.*

Call(s:)

```
SWITCH_GET_STATE(switchpin)
UByte switchCheck (void);
UByte SwitchFilt(switch_sState * switchState, UByte
portState);
```

Description:

All commands and appropriate parameters are shown in **Table 6-4**. The table uses the following conventions:

- Implementation
  f — function
  M — Macro

Table 6-4. Switch Drivers Macros and Functions

Function / MACRO	Parameters	Description (Value in Hex)	Notes[1]
InitSWITCH()	None	DDRx = DDRx|LED_MASK_PORTx	f
switchCheck()	None	Read actual state of all ports assigned to a switch which has to be called in a timer interrupt.	f
SwitchFilt()	switchState — structure with state of switches at actual port and required parameters. Insert: switchStatePTx, where x can be A,B,C,D,E,F portState — state of actual port Insert: IOCTL(PORTx, PORT_GET_DATA, NULL), where x can be A,B,C,D,E,F	Read actual state of defined port which has to be called in a timer interrupt.	f
return SWITCH_GET_STATE (SwitchName)	SwitchName (User pin name)	Return portFiltState.Bits.Bitx ^ SWITCH_POLARITY	M[2]

1. f = function
   M = macro
2. Name and polarity of the SWITCH must be defined (see **6.4.2 Static Initialization**)

Table 6-5. Memory Consumption and Execution Time of Functions

Function	Size	Cycles			Note
		Min	Typ	Max	
switchCheck()	nx16	nx94	nx94	nx114	(1)
SwitchFilt()	58	82	82	102	

1. n is the number of ports used for switches

6.4.4 Functional Description

6.4.4.1 switchCheck

Call(s):

```
void switchCheck (void)
```

Description:

The function calls the function SwitchFilt() for all ports assigned to a switch.

Return:

0 if all switches are in stable position.

Range Issues: None

Special Issues: None

Example:

```
void IsrTimerB_Overflow(void)
{             /* 1kHz ISR */
              UByte checkSwitchResult;

              checkSwitchResult = switchCheck();
              ...
}
```

6.4.4.2 switchFilt

Call(s):

```
UByte switchFilt(switch_sState * switchState, UByte portState)
```

Description:

The function compares the port's current state and previous state when the counter expires function and updates the filtered state of the port.

Return:

Number of function calls necessary to update the filtered state of the switches. If all swithes at the port are stable, returns 0.

Parameters:

*switchState (in/out) — pointer to a structure where the status of switches is for an actual port. Choose one from the next list:

```
&switchStatePTA
&switchStatePTB
&switchStatePTC
&switchStatePTD
&switchStatePTE
&switchStatePTF
```

portState(in) — actual state of port. To get the port state use one of the following commands:

```
IOCTL(PORTA, PORT_GET_DATA, NULL)
IOCTL(PORTB, PORT_GET_DATA, NULL)
IOCTL(PORTC, PORT_GET_DATA, NULL)
IOCTL(PORTD, PORT_GET_DATA, NULL)
IOCTL(PORTE, PORT_GET_DATA, NULL)
IOCTL(PORTF, PORT_GET_DATA, NULL)
```

Range Issues: None

Special Issues: None

Example:

```
    void IsrTimerB_Overflow(void)
    {     /* 1kHz ISR */
          UByte checkSwitchResult;

          SwitchFilt(&switchStatePTA, IOCTL(PORTA, PORT_GET_DATA, NULL));
          ...
}
```

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution;
P.O. Box 5405, Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

JAPAN:

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

ASIA/PACIFIC:

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

TECHNICAL INFORMATION CENTER:

1-800-521-6274

HOME PAGE:

http://www.motorola.com/semiconductors

MOTOROLA

SDKHC08AUG/D