

SoftConsole v3.1

User's Guide

Actel Corporation, Mountain View, CA 94043

© 2010 Actel Corporation. All rights reserved.

Printed in the United States of America

Part Number: 50200177-2

Release: February 2010

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Actel.

Actel makes no warranties with respect to this documentation and disclaims any implied warranties of merchantability or fitness for a particular purpose. Information in this document is subject to change without notice. Actel assumes no responsibility for any errors that may appear in this document.

This document contains confidential proprietary information that is not to be disclosed to any unauthorized person without prior written consent of Actel Corporation.

Trademarks

Actel, IGLOO, Actel Fusion, ProASIC, Libero, Pigeon Point and the associated logos are trademarks or registered trademarks of Actel Corporation. All other trademarks and service marks are the property of their respective owners.

Table of Contents

Introduction	5
Key Features	5
SoftConsole Package	5
Software and Hardware Tool Flows	7
Supported CPUs	8
1 SoftConsole Environment	11
Workbench	11
Perspectives, Views, and Editors	11
Workspace	11
Project	11
2 Software Installation	13
System Requirements	13
Installation Instructions	13
Licensing	15
Firmware Catalog Installation	16
3 Creating Embedded Applications with SoftConsole	17
Setting Up the Workspace	17
Setting Up a Project	17
Importing Peripheral Drivers and CMSIS into SoftConsole (Cortex-M3)	23
Importing Peripheral Drivers and CMSIS into a Compiler (Cortex-M3).	24
Importing Firmware Drivers and Hardware Abstraction Layers into SoftConsole (Cortex-M1 and Core8051s)	26
Importing Drivers and Hardware Abstraction Layers into a Compiler	26
Using the Editor to Create Source and Header Files	26
4 Project Settings	29
Project Settings for Cortex-M3	29
Project Settings for Cortex-M1	30
Project Settings for Core8051s	36
5 Building a Project	43
Creating the Release Version	44
Types of Project Build Methods	44
6 Debugging with SoftConsole	47
Debug Perspective	50
Debug View	51
7 Programming Flash Memory in Cortex-M1 Systems	59
Overview	59
Minimum Requirements	59

Flash Memory Programming Flow	59
A Appendix A – Programming Flash Memory in Cortex-M1 Systems	61
Overview	61
Minimum Requirements	61
Flash Memory Programming Flow Overview	61
Setting Up SoftConsole	64
Flash Programming Checklist	74
B Appendix B – Configuring the Debug Utility for CoreMP7 Projects	75
C Appendix C – Reference Documents	79
SoftConsole Documentation	79
D Product Support	81
Actel Customer Technical Support Center	81
Actel Technical Support	81
Website	81
Contacting the Customer Technical Support Center	81
Index	83

Introduction

SoftConsole is a free software development environment that enables the rapid production of C executables for the ARM[®] Cortex[™]-M3, Cortex-M1, CoreMP7, and Core8051s soft intellectual property (IP) processors. From a SoftConsole project you can write software that is compiled for use in debugging or programming into a nonvolatile memory device. SoftConsole includes a fully integrated debugger that offers easy access to memory contents, registers, and single-instruction execution.

SoftConsole includes a flexible and easy to use graphical user interface (GUI) for managing software development projects. The tool gives you the ability to organize files in a project, quickly develop and debug software programs, implement them in Actel devices, simultaneous access multiple tool windows, and quickly switch editing and debug views.

Key Features

- Eclipse-based integrated design environment (IDE)
- GNU C/C++ compiler (Cortex-M3, Cortex-M1 and CoreMP7)
- SDCC compiler (Core8051s)
- GDB debugger
- FlashPro4 debug interface
- Simultaneous access to multiple tool windows
- Quick switching between C/C++ and debug views
- One or more perspectives in a workbench window
- Perspectives can be customized by the user
- Direct integration with Actel's Firmware Catalog

SoftConsole Package

The following tools are included in the SoftConsole package:

- SoftConsole Eclipse-based embedded software development environment
- GCC compiler
- SDCC compiler
- GDB debugger
- Debug support for Cortex-M3, Cortex-M1, CoreMP7, and Core8051s
- Support for programming and debugging with FlashPro4

Eclipse IDE

The SoftConsole v3.1 Eclipse platform includes:

- Eclipse IDE v3.3
- Eclipse CDT v4.0.3 (C/C++ development tools)
- Sun[™] Java J2SE[™] Runtime v6, update 7

SoftConsole is built around the open source Eclipse IDE. SoftConsole is structured as a collection of plug-ins, each of which contains the code that provides some of SoftConsole's functionality. The code and other files for a plug-in are installed on the local computer and get activated automatically as required.

GCC Compiler

The SoftConsole v3.1 GCC compiler tools include the following:

- CodeSourcery™ Sourcery G++™ Lite (GNU Toolchain for ARM processors) 2008q1-126
- GCC (GNU Compiler Collection) v4.2.3
- Binutils (GNU binary utilities) v2.18
- GNU make v3.81

The CodeSourcery G++ GCC compiler in SoftConsole supports the Cortex-M3, Cortex-M1, and CoreMP7 processors. The G++ compiler includes everything you need to develop your application—optimizing GNU C/C++ compilers, a flexible assembler, a powerful linker, runtime libraries, and a source—assembly-level debugger, and a Debug Sprite for hardware debugging using the FlashPro4 programmer.

SDCC Compiler

- SoftConsole v3.1 SDCC tools include:
- SDCC Small Device C Compiler v2.6.3
- CodeSourcery G++ Line 2008q1-126, integrating port changes from CodeSourcery Tools for 8051 v1.0-7
- Binutils (GNU binary utilities) v2.18
- CodeSourcery omf2elf 1.0-7

SDCC is a retargettable, optimizing ANSI C compiler that targets the 8051 in SoftConsole. SDCC in SoftConsole supports a full range of functionality and has been set up to work with GDB and FlashPro4 to simplify the SoftConsole environment.

GNU Debugger (GDB)

SoftConsole v3.1 GDB tools include:

- GDB (GNU Debugger) v6.7.50 for Cortex-M3, Cortex-M1, and CoreMP7
- GDB (GNU Debugger) v6.7.50

The GNU Debugger in SoftConsole is a command-line source-level debugger. In addition to breakpoints and commands for controlling program execution, the SoftConsole GDB supports breakpoints and flash programming for Cortex-M1, is fully integrated into SoftConsole, and works seamlessly with the GCC compiler and FlashPro4 programmer.

Debug Sprites

SoftConsole Debug Sprite tools include the following:

- Cortex-M3: CodeSourcery ARM Debug Sprite v1.0-7 + Actel 1.3.2-M3 + Flash Programming 1.2.2
- Cortex-M1: CodeSourcery ARM Debug Sprite v1.0-7 + Actel 1.3.2-M3 + Flash Programming 1.2.2
- Core8051s: Actel C8051 Debug Sprite v1.0-7 + Actel 1.5.0 + Flash Programming 1.2.2
- CoreMP7: FS2 In-Target System Analyzer for ARM Processor Cores v1.2.0

The SoftConsole Debug Sprites interfaces between the relevant target processor and GDB to facilitate hardware-based debugging by translating between GDB Remote Serial Protocol and target processor-specific JTAG/debug commands/responses. This enables utilization of the GDB debugger with the supported Actel processors (Cortex-M3, Cortex-M1, CoreMP7, and Core8051s) in Actel devices. The sprites are automatically installed and configured as part of the SoftConsole installation.

FlashPro4

FlashPro4 targets the latest generation of flash-based devices offered by Actel, including IGLOO®, ProASIC®3, Fusion, SmartFusion™ and RT ProASIC3 families. FlashPro4 offers extremely high performance through USB 2.0 and is high-speed compliant for full use of the 480 Mbps bandwidth. Powered exclusively via USB, FlashPro4 provides a V_{PUMP} voltage of 3.3 V for programming these devices.

The FlashPro4 programmer can be used with SoftConsole to program both on-chip and off-chip flash memory (Cortex-M3, Cortex-M1) and perform debugging (all devices). Support for Core8051s program download to Fusion NVM and external flash given a suitable hardware platform.

Firmware Catalog

The Firmware Catalog is a standalone executable program that supports SoftConsole embedded processor development toolchains targeting the ARM Cortex-M3, Cortex-M1, CoreMP7, and Core8051s processors. The Firmware Catalog streamlines the locating and generating of firmware that is compatible with Intellectual Property (IP) cores used in Actel FPGA designs.

Software and Hardware Tool Flows

Two development flows must be followed when building a design using a processor in an FPGA: the FPGA hardware development flow using Libero® Integrated Design Environment (IDE) and the processor program (software) development flow with SoftConsole. [Figure 1](#) shows the two flows and the connections between them.

The hardware design is input with the editor or SmartDesign, place-and-route is performed, and then the design is programmed into the device using FlashPro4.

The software is coded, compiled, and debugged with FlashPro4. The Libero IDE Firmware Catalog contains drivers or other software components that are associated with the IP cores used in SmartDesign and these files are made available to SoftConsole for the software designer to use in the coding stage. From this point the flows are independent until the

programming stage, when both the hardware and software are programmed to the device using the FlashPro4 programmer. Debugging of the hardware and software can also be accomplished through the FlashPro4.

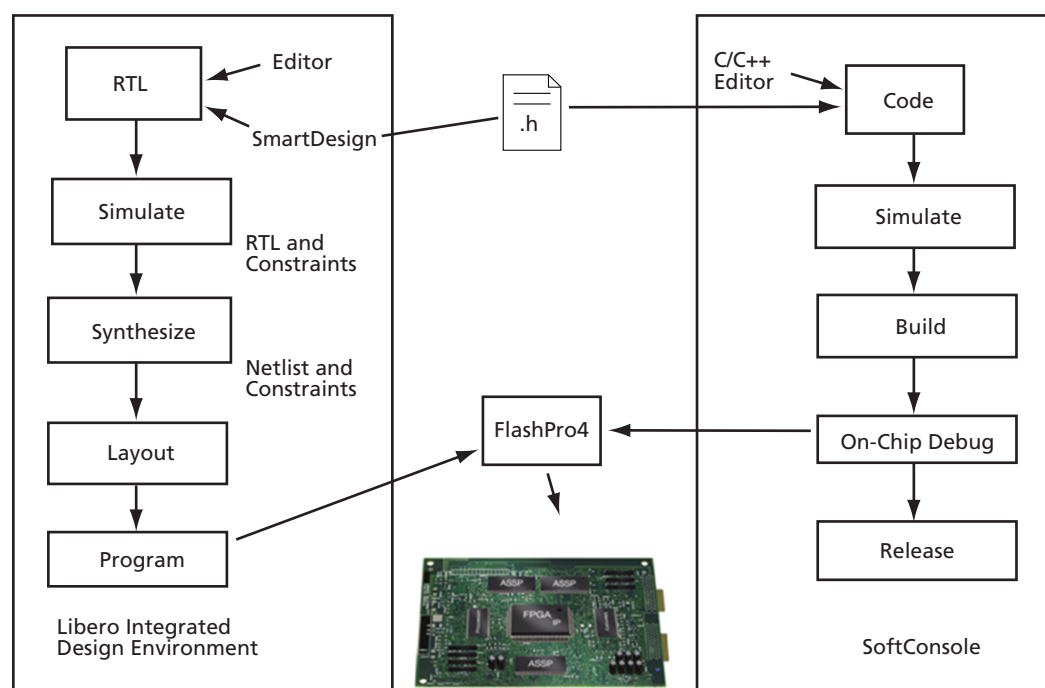


Figure 1 · Software and Hardware Tool Flows

Supported CPUs

Cortex-M3

The ARM Cortex-M3 processor is the industry-leading 32-bit processor for highly deterministic real-time applications and has been specifically developed to enable partners to develop high-performance low-cost platforms for a broad range of devices, including microcontrollers, automotive body systems, industrial control systems, wireless networking, and sensors. The processor delivers outstanding computational performance and exceptional system response to events, while meeting the challenges of low dynamic and static power constraints. The processor is highly configurable, enabling a wide range of implementations from those requiring memory protection and powerful trace technology through to extremely cost sensitive devices requiring minimal area.

Cortex-M1

Developed by ARM in collaboration with Actel, the 32-bit ARM Cortex-M1 processor is the first ARM processor designed for FPGA implementation. With a balance between size and speed, the free Cortex-M1 processor operates at up to 60 MHz and can be implemented in as few as 4,435 VersaTiles in M1 Fusion and M1 ProASIC3 flash-based FPGAs. A streamlined three-stage pipeline solution, the Cortex-M1 processor runs a subset of the ARM Thumb®-2 instruction set, so existing Thumb code can be utilized without change. The configurable Cortex-M1 processor connects to the advanced high performance bus (AHB), enabling designers to build their subsystem and add peripheral functionality. In addition to SoftConsole from Actel and µVision®3 tools from Keil™, third-party vendors offer supporting tools from compilers and debuggers to RTOS solutions. Cortex-M1 is available for use in Actel M1 devices free of charge with no license fees, royalties, or contracts to sign.

Core8051s

Core8051s is an ASM51-compatible microcontroller core which contains the main 8051 core logic but no peripheral logic. Core8051s has an advanced peripheral bus (APB) interface to expand the functionality of the core by connecting it to existing APB IP peripherals. This allows users to configure the core with the peripheral functions (timers, UARTs, and I/O ports) needed for the application. Core8051s is a pipelined architecture that can execute one 8051 instruction per clock cycle and is available for free usage in Actel devices.

CoreMP7

ARM7[™] is a 32-bit RISC microprocessor. Actel's CoreMP7 is a soft IP version of the ARM7TDMI-S[™] and has been optimized to maximize speed and minimize size in Actel's M7 Fusion and M7 ProASIC3 flash-based FPGAs. CoreMP7 executes the ARMv4T instruction set architecture and implements all 32-bit ARM7 instructions and all 16-bit Thumb[®] instructions. The processor has a 3-stage pipeline, 32-bit arithmetic logic unit (ALU), 32-bit register file, 32-bit external address and data bus interface, and JTAG debug interface. CoreMP7 is available for use in Actel M7 devices free of charge with no license fees, royalties, or contracts to sign.

SoftConsole Environment

The following definitions will help familiarize you with the SoftConsole environment.

Workbench

The term workbench refers to the development environment for the Actel SoftConsole. The workbench is where you edit, compile, and debug your application.

Perspectives, Views, and Editors

The workbench window contains one or more perspectives. A perspective is a group of views and editors in the workbench window laid out in a specific way. One or more perspectives can exist in a single workbench window. Each perspective can have a different set of views.

A view is a visual component within the workbench. It is typically used to navigate a list or hierarchy of information (such as the resources in the workbench), or display properties for the active editor. Modifications made in a view are saved immediately.

An editor is also a visual component within the workbench. It is typically used to edit source code, but it can be used to view any text file. Typically, editors are launched by clicking on a resource in a view. Modifications made in an editor follow an open-save-close lifecycle model.

Workspace

A workspace is the location on your machine where your work is stored. A workspace contains one or more SoftConsole projects that are visible in the workbench.

Project

Code development using the SoftConsole IDE is organized into projects. A project can be defined as the logical grouping of all the source files as well as the compiler, assembler, and linker settings required to compile and link a program. Individual files are not required to be physically located within the project folder, however.

Software Installation

System Requirements

SoftConsole can be run on the following Microsoft® Windows® operating systems:

- Microsoft Windows Vista® Business (U.S. Version)
- Windows XP Professional with SP3 (U.S. Version, cumulative)

Note: SoftConsole might run on other XP/Vista variants, but it is not supported on any platforms other than those listed above. SoftConsole is not supported on any non-U.S. version of Windows.

Note: Administrator privileges are required in order to install SoftConsole on Windows Vista or XP.

The following are the minimum system requirements needed to support SoftConsole:

- Pentium 1.0 GHz processor
- NTFS, FAT32 file system
- 400 MB free disk space
- 128 MB RAM
- 1024 x 768 resolution monitor

Installation Instructions

SoftConsole is available for free download from the Actel website: www.actel.com/download/software/softconsole/default.aspx.

To install SoftConsole:

1. Double-click the **SoftConsole_vX_X_setup.exe** file. This opens the installation wizard. Click **Next**.
2. Read through the license agreement, check the **I accept the agreement** radio button, and click **Next**.
3. If the default installation folder is okay, click **Next**. If you want to install SoftConsole in a different folder, either browse to it or type in the path and click **Next**.

4. By default, all SoftConsole components are selected (recommended) for installation. If you do not want to install a component, clear the selection for it. Click **Next**. This opens the Core8051s and SDCC Tools page (Figure 2-1).

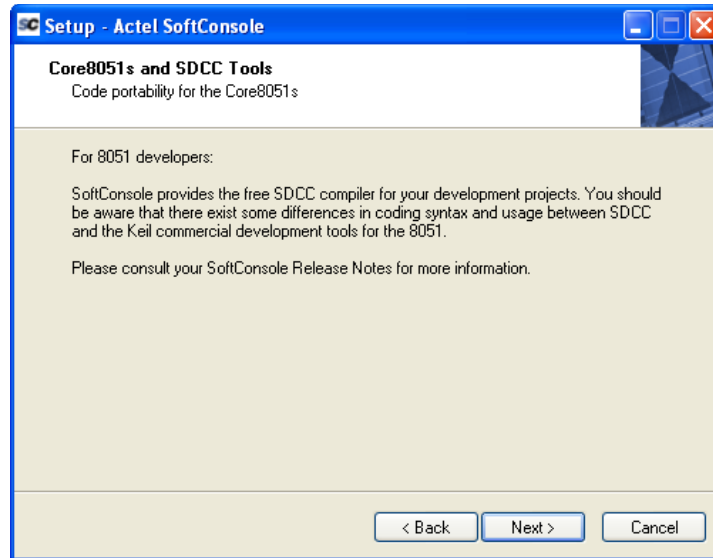


Figure 2-1 · Core8051s and SDCC Tools Page

This installation page is for your information only. When you have finished reading it, click **Next**.

5. By default, SoftConsole creates a Start Menu folder named *Actel SoftConsole vX.X*. You can either accept the default name or use your own name. If you want to rename the SoftConsole Start Menu folder, type in the new name or browse to a folder that you would like to use. Click **Next**.
6. To have the installation program create a desktop or Quick Launch icon, select the appropriate check box. If no additional icons are desired, clear the check box for both items. Click **Next**.

7. Click **Install** to complete the installation. During the installation, an information page appears (Figure 2-2). Click **Next**.

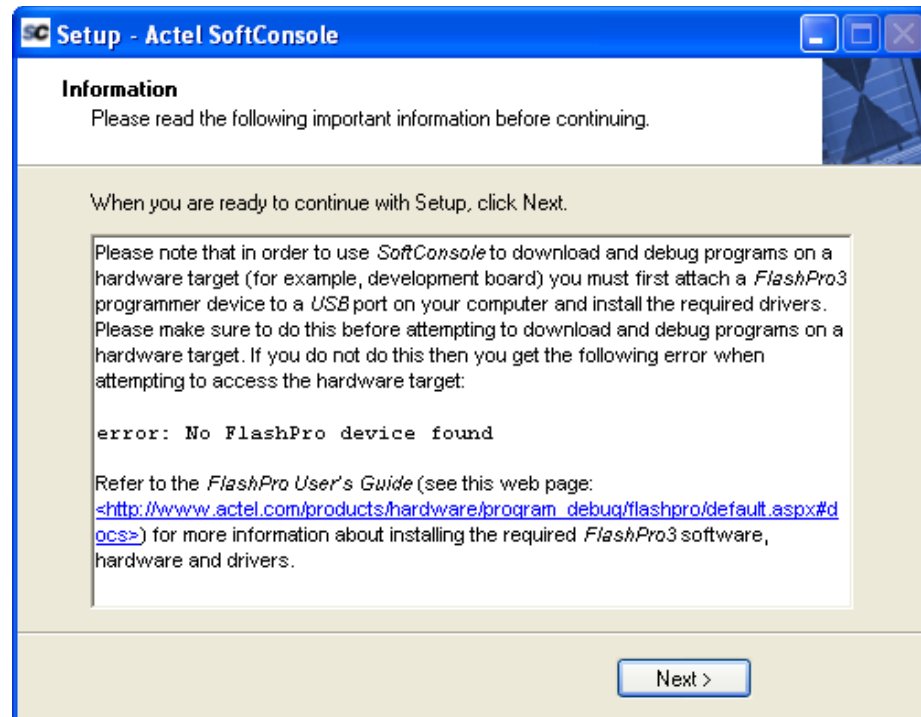


Figure 2-2 · SoftConsole Information Page

8. Click **Finish** to complete the installation.

Licensing

SoftConsole is freely licensed for use with Actel processors and devices. SoftConsole contains Open Source elements. Individual licenses for these elements are included in the SoftConsole License Agreement that is presented during the installation process.

Firmware Catalog Installation

Firmware Catalog is a tool that allows you to create firmware for software development. Typically, you use the tool to select the firmware that you need and Firmware Catalog places the firmware into your software development project (SoftConsole, IAR EWARM, or Keil).

Firmware Catalog is installed by default when Libero IDE is installed (v8.6 or newer) and it is integrated with SoftConsole v3.1. Firmware Catalog can be launched from the **Run** menu in SoftConsole.

If you do not want to install Libero IDE, you can download the Firmware Catalog installation only from Actel's Downloads web page: www.actel.com/download/default.aspx.

Regardless of which installation method you use, Firmware Catalog is a standalone tool with its own executable.

Creating Embedded Applications with SoftConsole

This chapter describes how to create an embedded application with SoftConsole.

Setting Up the Workspace

Start SoftConsole. SoftConsole will prompt you to select a workspace with the Workspace Launcher dialog box (Figure 3-1). Use the **Browse** button to navigate to a workspace folder and click **OK**.

Note: The Workspace Launcher enables you to set the selected workspace as your default workspace. If you set a default workspace, SoftConsole will not prompt you again for a workspace. To change workspaces, from the **File** menu, choose **Switch Workspace**.

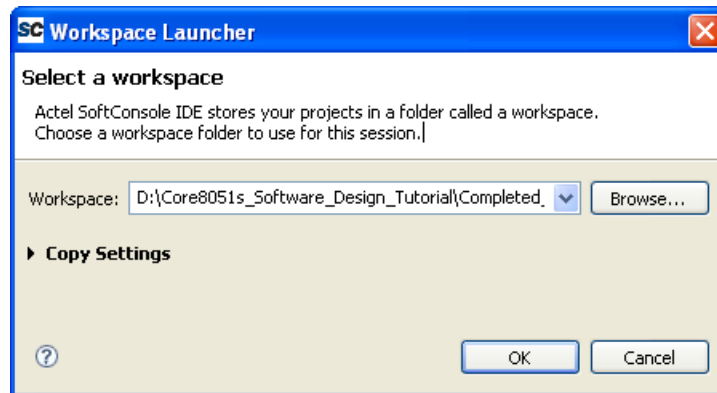


Figure 3-1 · Selecting a Workspace

Setting Up a Project

At this point, you can either create a new project or import an existing project into your workspace.

Creating a New Project

1. From the **File** menu, choose **New > C Project**. The C Project box opens (Figure 3-2 on page 18).
2. Type the name of your project in the Project name box, select **Executable (Managed Make)** in the Project types box, and select the appropriate toolchain in the Toolchain box. SoftConsole currently supports toolchains for the Cortex-M3, Cortex-M1, Core8051s, and CoreMP7 microcontroller cores.
3. Click **Next**. Verify that Release and Debug configurations are checked. Click **Finish**.

Note: In an Executable (Managed Make) project, SoftConsole will manage the makefiles. Actel recommends this option.

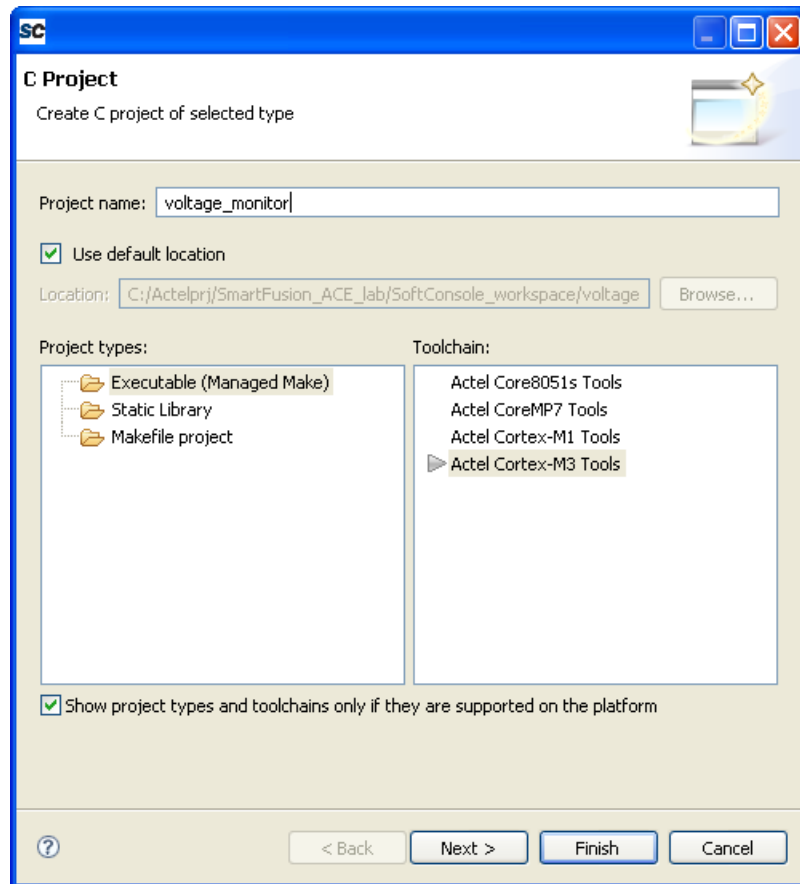


Figure 3-2 · Selecting Project Name, Type, and Toolchain

The next step in creating an application is to add some source and/or header files to the project. Files can be imported from another place in your file system or created using the built-in editor. The memory map created by SmartDesign provides useful information for creating header files. The `<component>.xml` file from the `<$project>/component/work/<component>` folder can be opened to view the memory map of the FPGA design.

4. From the **File** menu, choose **Import**. The Import box appears (Figure 3-3).

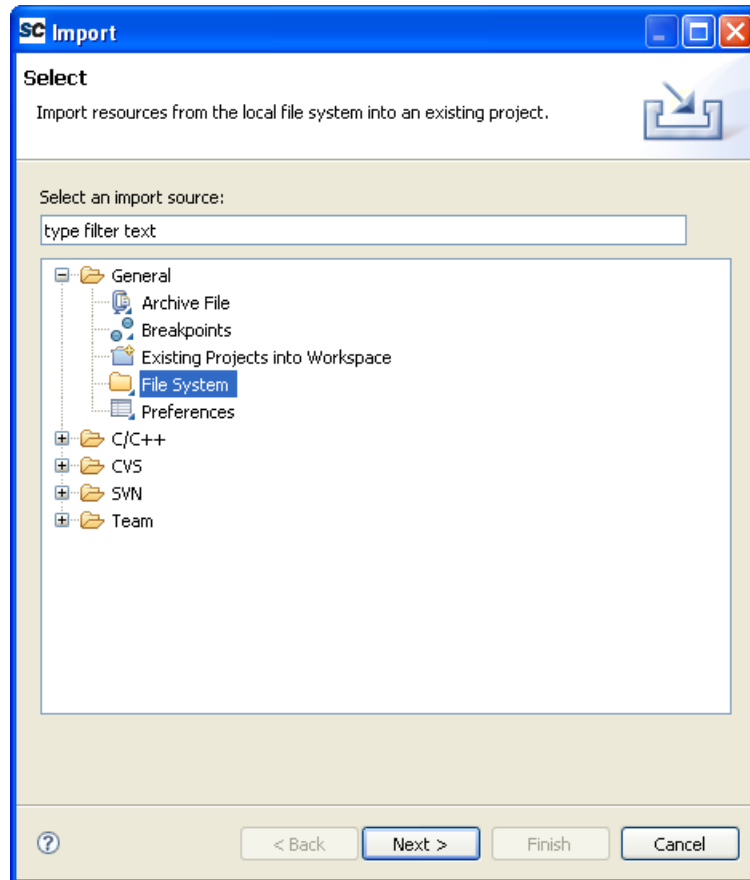


Figure 3-3 · Selecting File System as the Entity to Import

5. Select **General** to expand the options and select **File System**. Click **Next**. This opens the Import: File System window (Figure 3-4 on page 20).
 - Next, use the **Browse** button in the From directory box to select the folder that contains the source file(s). Click **OK**. The right side box shows the files in the selected folder. Check the boxes in front of the files you want to import.
 - Click the **Browse** button in the Into folder box to specify the destination of the project folder. Verify that **Create selected folders only** is selected in the options window. Click **Finish**.

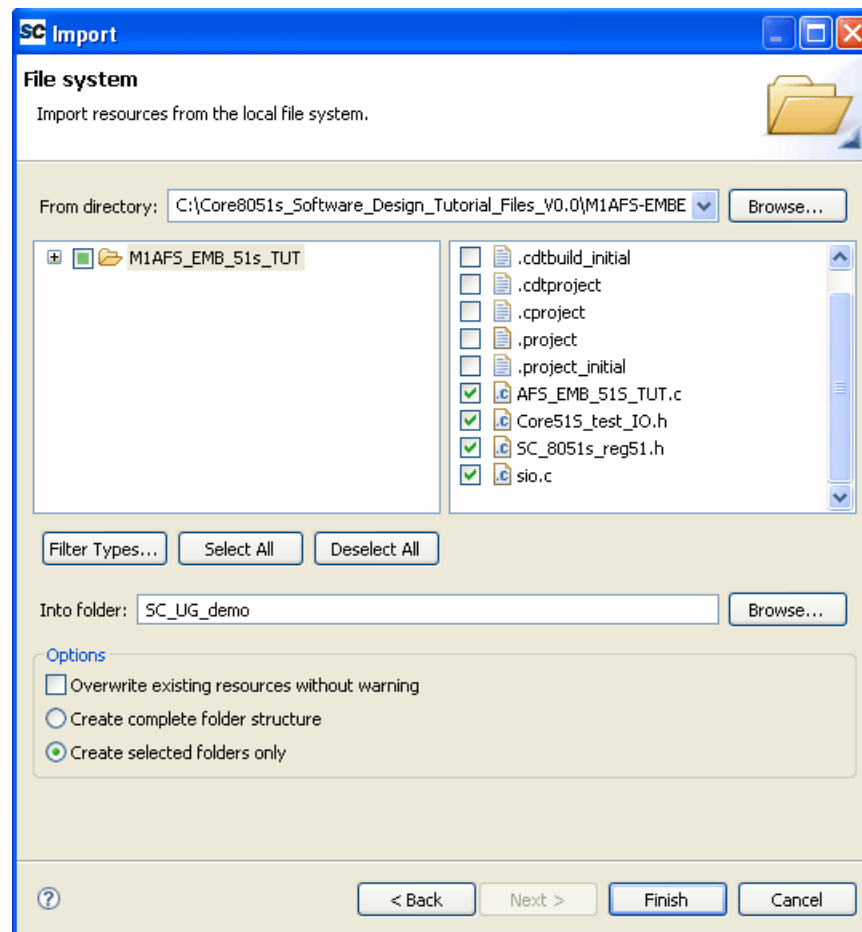


Figure 3-4 · Importing Files from a Project

- Expand the project by clicking the + sign next to the project name in the Project Explorer window (Figure 3-5). You can view any of these files in the editor window by double-clicking the file name.

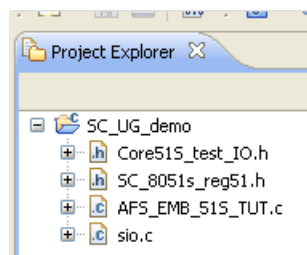


Figure 3-5 · Imported Files

Importing an Existing Project

1. From the **File** menu, choose **Import**. The Import box opens (Figure 3-6).
2. Select **General** to expand the options and then select **Existing Projects into Workspace**. Click **Next**.

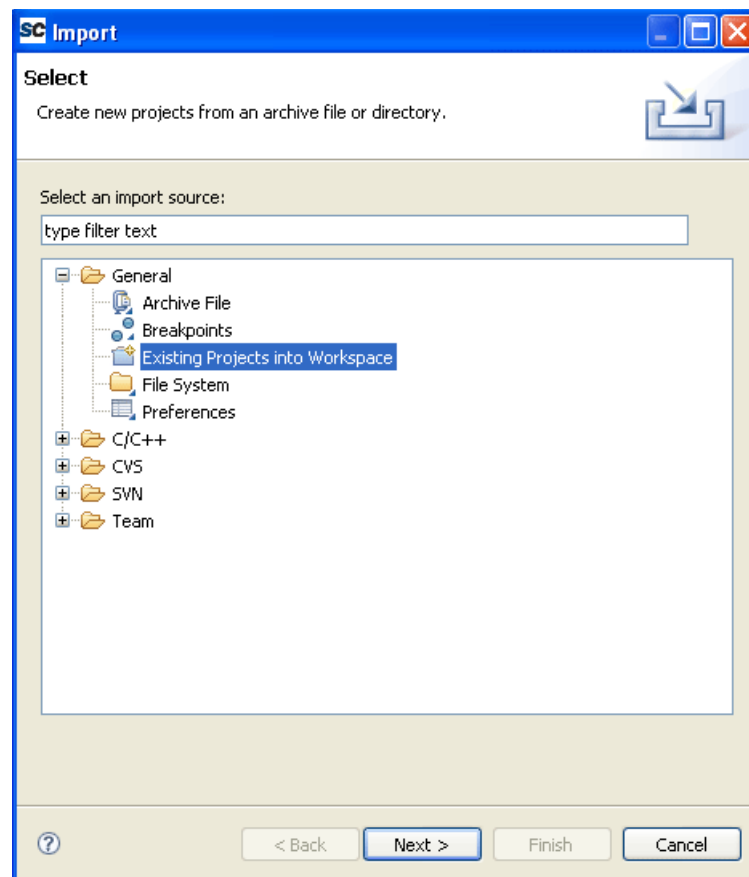


Figure 3-6 · Selecting Projects as the Type of Entity to Import

This opens the Import: File System window

- Use the **Browse** button in the Select root directory box to navigate to the folder that contains the project(s) to import and click **OK**. The project(s) are now listed in the Import Projects box (Figure 3-7). Select the projects you want to import and press **Finish**.

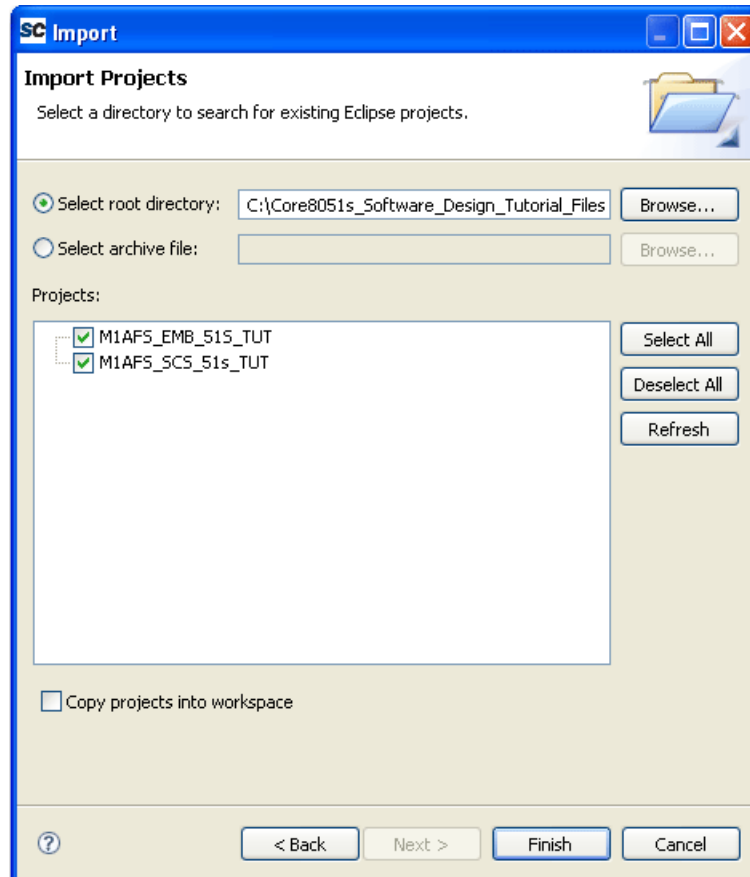


Figure 3-7 · Importing Projects into the Workspace

Note: A single project can also be imported by browsing to the specific project folder (instead of a folder which contains the project folder), selecting the project folder, and clicking **Finish**.

Checking the **Copy projects into workspace** box will make a copy of the project folder(s) inside the workspace folder and use these folders for project activities. Otherwise, a link to the original project folder(s) will be created.

You can expand and view the project contents by clicking the + sign next to the project name in the Project Explorer window. You can view any of these files in the editor window by double-clicking the file name (Figure 3-8).

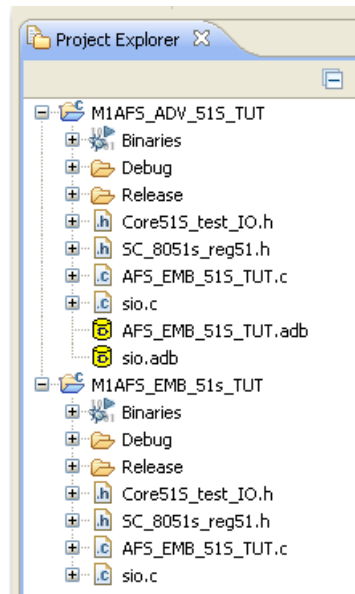


Figure 3-8 · Expanding the Imported Project Files

Importing Peripheral Drivers and CMSIS into SoftConsole (Cortex-M3)

The Cortex-M3 CMSIS is an abstraction layer that conforms to the ARM Cortex Microcontroller Software Interface Standard. The MSS Configurator generates the CMSIS code and drivers for the selected Cortex-M3 microcontroller peripherals. You will need to import the MSS peripheral drivers and CMSIS. If you launched the MSS Configurator from within Libero IDE, these drivers and the CMSIS are in the firmware folder of your Libero project.

For more information on how to use MSS Configurator, refer to the Libero IDE online help.

To import firmware drivers and CMSIS into SoftConsole:

1. Use the MSS Configurator to generate the CMSIS PAL and drivers.
2. From the **File** menu, choose **Import**. The Import dialog box will open.
3. Expand the **General** folder, select **File System**, and click **Next** to continue. The Import dialog box will open.
4. Click the **Browse** button to the right of the From directory field to set the folder to the folder containing the CMSIS and drivers. Click the + sign next to this folder to expand the path. Select the CMSIS and driver folders.
5. Click the **Browse** button for the Into folder field. Select the project name in the Import into Folder dialog box, and then click **OK**.
6. Click **Finish** to import the firmware drivers and the CMSIS.

Importing Peripheral Drivers and CMSIS into a Compiler (Cortex-M3)

For more information on how to use MSS Configurator, refer to the Libero IDE online help.

To import peripheral drivers and CMSIS into a compiler:

1. Right-click the project and select **Properties**.
2. Expand C/C++ BUILD on the left and select **Settings**.
3. Click GNU C Compiler > Directories (Figure 3-9).

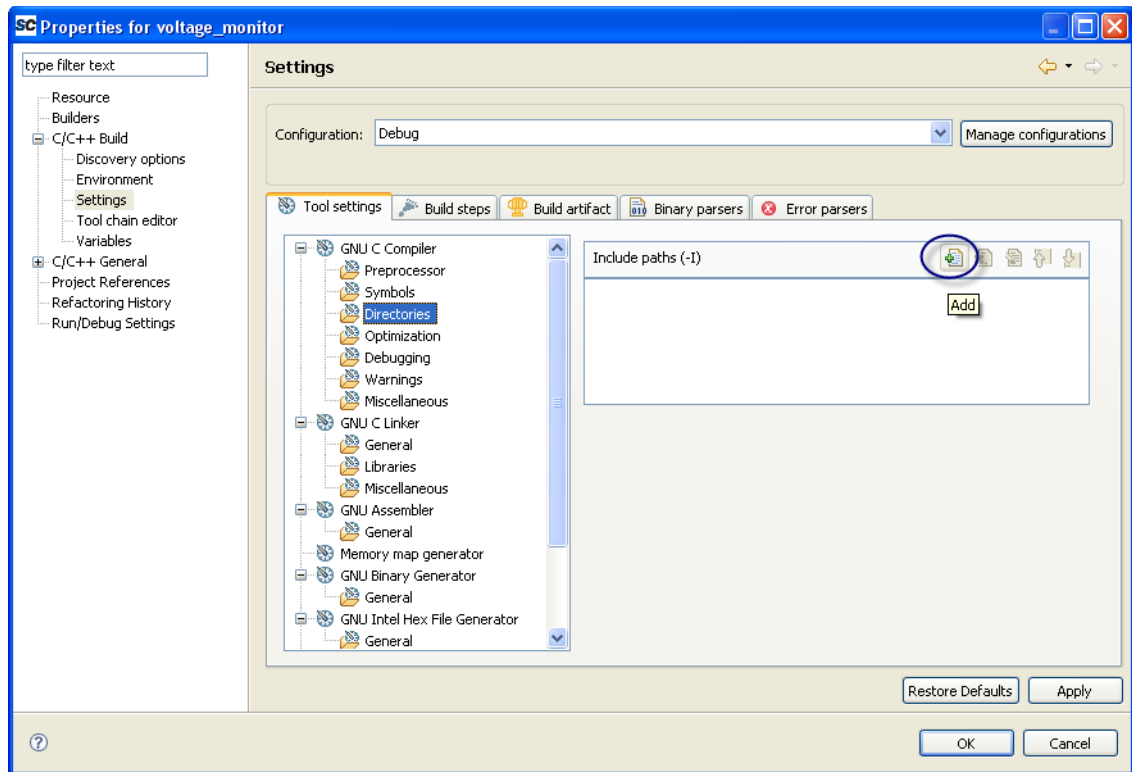


Figure 3-9 · GNU Compiler Directory Settings

4. Click the **Add** button in the include paths field.

5. Click the **Workspace** button (Figure 3-10).

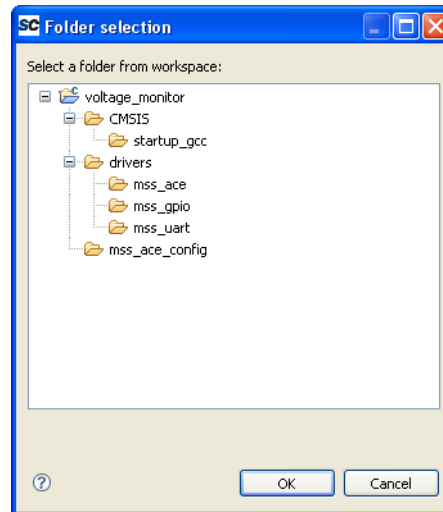


Figure 3-10 · SoftConsole Folder Selection Dialog Box

6. Expand the <project name> folder and select the **CMSIS** subdirectory.
7. Click **OK** in the Add directory path dialog box.
8. Repeat this process for all of the directories in the project . All of the directories in the project should be visible in the Include paths windows of the Settings dialog box (Figure 3-11).

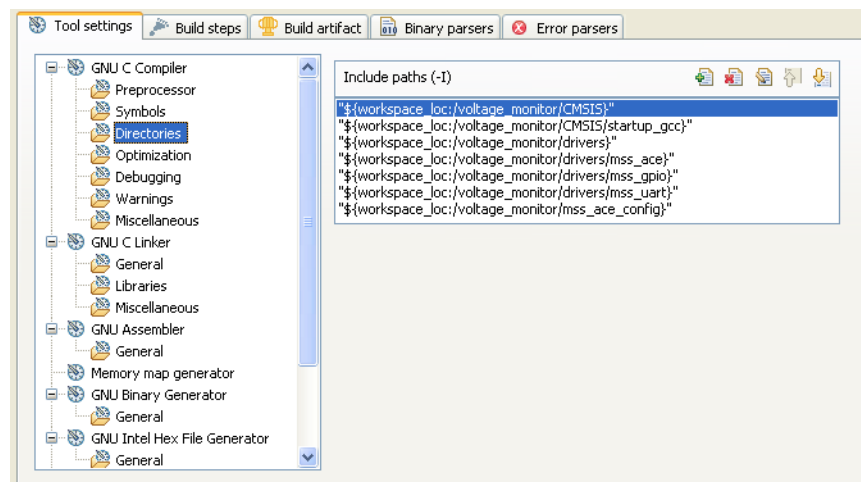


Figure 3-11 · Project Folder Settings

Importing Firmware Drivers and Hardware Abstraction Layers into SoftConsole (Cortex-M1 and Core8051s)

For information about accessing and generating drivers and Hardware Abstraction Layers (HAL), refer to the Firmware Catalog online help.

To import firmware drivers and HALs into SoftConsole:

1. Launch Firmware Catalog from SoftConsole by selecting **Run > Firmware Catalog**.
1. The configuration dialogs are specific to the selected drivers. Configure the drivers according to the needs for your design.
2. Generate the firmware cores and direct them into your project folder within the SoftConsole workspace. Close the Firmware Catalog.
3. In SoftConsole, right-click the project and click the **Refresh** button. The *HAL* and *Driver* folders will now appear in the Project Explorer.

Importing Drivers and Hardware Abstraction Layers into a Compiler

To import firmware drivers and HALs into a compiler:

1. Right-click the project and select **Properties**.
2. Expand **C/C++ BUILD** on the left and select **Settings**.
3. Click **GNU C Compiler > Directories**.
4. Click the **Add** button.
5. Click the **Workspace** button.
6. Expand the **<project name>** and select the *HAL* subdirectory.
7. Click the **OK** button on the Add Directory Path window.
8. Click the **Add** button.
9. Click the **Workspace** button.
10. Expand the **<project name>** and select the *HAL* directory.
11. Open the *HAL* subdirectory and select the *HAL/CortexM1* directory.
12. Repeat steps 7 to 9.
13. Repeat steps 4 to 6 and navigate to the *HAL/CortexM1/GNU* directory.
14. Repeat steps 7 to 9 but navigate to the *Drivers* directory, following the same sequence for all drivers within the directory.

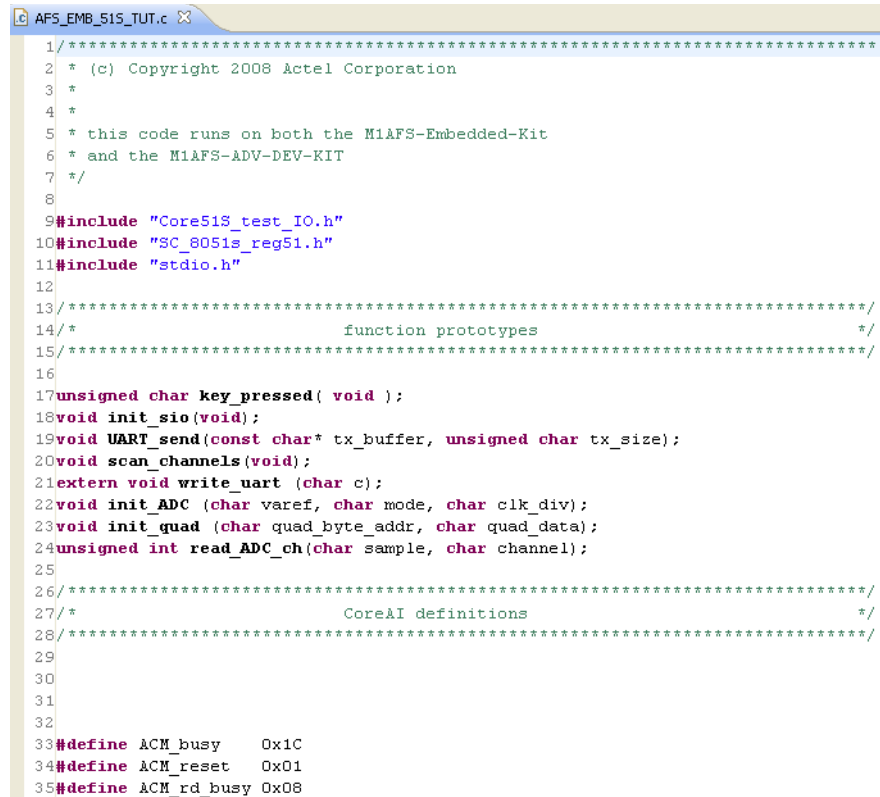
Using the Editor to Create Source and Header Files

The editor window can be used to create and edit header and source files.

To create a source file:

1. From the **File** menu, choose **New > Source File**. The new source file appears.
2. Use the **Browse** button to navigate to the folder where you want to save the source file and enter the name of the source file with a file extension.
Note: SoftConsole does NOT automatically assign a file extension.

3. Click **Finish**. If the source file was saved in your project folder, the Project Explorer file lists it. Double-click the file name in the Project Explorer to edit the source file (Figure 3-12).



```

1/*****
2 * (c) Copyright 2008 Actel Corporation
3 *
4 *
5 * this code runs on both the M1AFS-Embedded-Kit
6 * and the M1AFS-ADV-DEV-KIT
7 */
8
9#include "Core51S_test_IO.h"
10#include "SC_8051s_reg51.h"
11#include "stdio.h"
12
13/*****
14/*                                function prototypes                                */
15/*****
16
17unsigned char key_pressed( void );
18void init_sio(void);
19void UART_send(const char* tx_buffer, unsigned char tx_size);
20void scan_channels(void);
21extern void write_uart (char c);
22void init_ADC (char varef, char mode, char clk_div);
23void init_quad (char quad_byte_addr, char quad_data);
24unsigned int read_ADC_ch(char sample, char channel);
25
26/*****
27/*                                Core&I definitions                                */
28/*****
29
30
31
32
33#define ACM_busy    0x1C
34#define ACM_reset   0x01
35#define ACM_rd_busy 0x08

```

Figure 3-12 · Editing a Source File

To create a header file:

From the **File** menu, select **New > Header File**. This opens the editor window.

The editor supports common Windows commands for highlighting, cutting, and pasting text, as well as some development commands: automatic line numbering, block comment/uncomment, and line comment/uncomment.

Block comment/uncomment encloses a block of text between C-language comment markers ("/*" and "*/"). To add or remove block comments, highlight the block of text, right-click, and select **Source > Add Block Comment/Remove Block Comment**.

Line comment/uncomment will precede a line of text with the C++ line comment markers ("//"). To add or remove line comments, highlight the lines of text, right-click and select **Source > Comment / Uncomment**.

Project Settings

This chapter explains how to configure the compiler to match a Cortex-M3, Cortex-M1 or Core8051s application.

Project Settings for Cortex-M3

Import the CMSIS and peripheral drivers into your project and set the include directory paths if not already performed.

1. Right-click the project and choose **Properties**.
2. In the Properties for <project_name> window, select Settings under C/C++ Build and then select **Miscellaneous** under the GNU C Linker heading.
3. Enter `-T./CMSIS/startup_gcc/debug-in-actel-smartfusion-esram.ld` in the Linker Flags field (Figure 4-1. This command directs SoftConsole to use a linker script that builds an executable that will run from the SmartFusion internal SRAM.

Note: The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled into the linker executable. You can supply your own linker script by using the “-T” command line option.

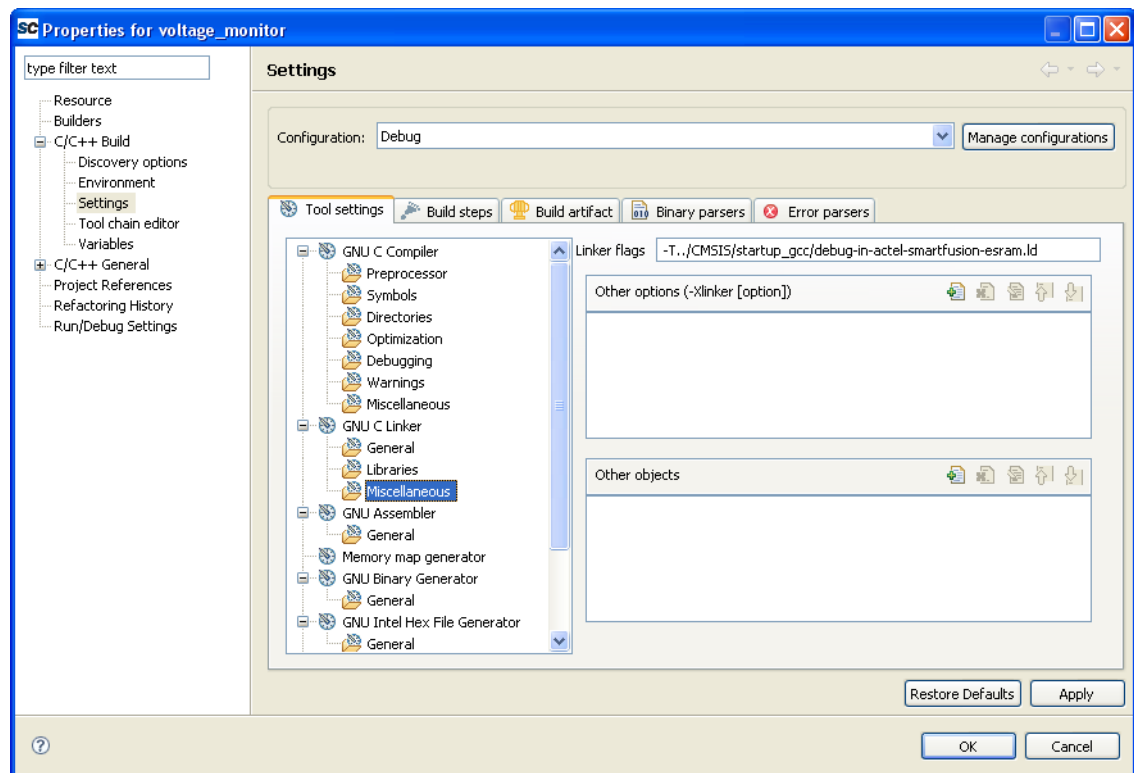


Figure 4-1 · Specifying the Linker Script

4. Click **Apply** then **OK** to close the Properties dialog box.

- Perform a clean build by selecting **Clean** from the **Project** menu. Accept the default settings in the Clean dialog box and click OK (Figure 4-2).

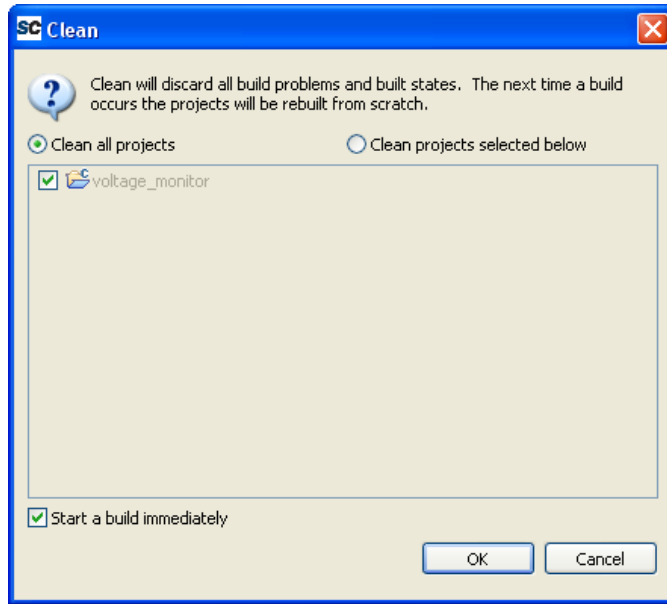


Figure 4-2 · SoftConsole Clean Dialog Box

- Confirm that there are no errors listed in the **Problems View**.

Project Settings for Cortex-M1

Cortex-M1 Hardware Abstraction Layer (HAL)

The Cortex-M1 HAL provided in the SoftConsole installation is used by the drivers. Therefore, you must include the HAL in your project when using the Actel drivers and the flash programmer utility. There are three directories in the HAL:

- hal*
- hal/CortexM1*
- hal/CortexM1/GNU*

The HAL source code is in the *<SoftConsole v3.1>\src\Cortex-M1\Cortex-M1_hal* directory.

Linker Scripts for Specifying Program Code and Data Memories

Actel provides linker scripts you can use for a Cortex-M1 embedded processor target application. Table 4-1 gives descriptions of the Cortex-M1 embedded processor linker scripts provided with SoftConsole.

Table 4-1 · Linker Scripts Provided with SoftConsole

Linker Script	Description
boot-from-actel-coreahbnvm.ld	Used to build software images where the processor boots from internal NVM (exists only in Actel Fusion® FPGA devices). Copies the code to RAM and then runs from RAM.

Table 4-1 · Linker Scripts Provided with SoftConsole

boot-from-intel-flash.ld	Used to build software images where the processor boots from external flash (Intel® JS28F640J3D). Copies the code to RAM and then runs from RAM.
only-ram-memory.ld	Used to build software for debugging code running from SRAM. The data and program code are stored in SRAM.
run-from-actel-coreahbnvm.ld	Used to build software images where the processor boots and runs from internal NVM (exists only in Actel Fusion FPGA devices). Only data is stored in RAM.
run-from-intel-flash.ld	Used to build software images where the processor boots and runs from external flash (Intel JS28F640J3D). Only data is stored in RAM.

All of the linker scripts place the processor data in RAM. You may have to modify the linker script to match your target hardware system. Specifically, the address location and size of the program and data memory peripherals as well as the stack size need to match your system. The only portion of the linker scripts that you need to modify is the board customization section. As an example, the board customization section from the boot-from-actel-coreahbnvm.ld linker script is shown below (Figure 4-3).

```

/*****
 * Start of board customization.
 *****/
MEMORY
{
    /*
     * WARNING: The words "SOFTCONSOLE", "FLASH", and "USE", the colon ":", and
     *           the name of the type of flash memory are all in a specific order.
     *           Please do not modify that comment line, in order to ensure
     *           debugging of your application will use the flash memory correctly.
     */

    /* SOFTCONSOLE FLASH USE: actel-coreahbnvm */
    rom (rx) : ORIGIN = 0x00000000, LENGTH = 1M

    /* Normal SRAM */
    ram (rwx) : ORIGIN = 0x10000000, LENGTH = 1M
}

RAM_START_ADDRESS = 0x10000000; /* Must be the same value MEMORY region ram ORIGIN above. */
RAM_SIZE = 1M; /* Must be the same value MEMORY region ram LENGTH above. */
MAIN_STACK_SIZE = 256k; /* Cortex main stack size. */
PROCESS_STACK_SIZE = 16k; /* Cortex process stack size (only available with OS extensions).*/

/*****
 * End of board customization.
 *****/

```

Figure 4-3 · Linker Script Example

Flash Memory

The following comment in the linker script tells the flash programmer which type of flash you are targeting. Do not modify this comment. There is a file which describes the flash commands in the installation of SoftConsole in the <SoftConsole v3.1>\Sourcery-G++\share\sprite\flash directory. This file has the same name as the name in this comment (actel-coreahbnvm.xml in this case).

Stack Size

The stack size is adjusted by modifying the value assigned to MAIN_STACK_SIZE.

Note: PROCESS_STACK_SIZE is only used with versions of Cortex-M1 that support OS extensions. OS extensions are not currently available as part of the Actel Cortex-M1 implementation.

Memory Size

The nonvolatile memory size and location is specified by editing the following line:

```
rom (rx) : ORIGIN = 0x00000000, LENGTH = 1M
```

The RAM size and location is specified by editing the following line:

```
ram (rwx) : ORIGIN = 0x00000000, LENGTH = 1M
```

The values for RAM_START_ADDRESS and RAM_SIZE must also be edited so that they match the values specified in the line above.

These linker scripts work in conjunction with the startup files and the flash programmer in SoftConsole. Based on the contents of the linker script, SoftConsole creates a memory_map.xml file and stores it in the current active build configuration. The flash programmer uses this file to learn the addresses of the flash (internal NVM or external flash) and SRAM peripherals. Make sure that the debug configuration you are using points to the build configuration with the current memory-map.xml file.

Specifying Linker Script

After you have modified the linker script to match your program code and data memory peripherals, you need to point the linker to your linker script.

1. Right-click the project and choose **Properties**.
2. Navigate to **C/C++ Build > Settings**.

3. Highlight **GNU C Linker > Miscellaneous** and enter **-T <path to linker script>** in the Linker flags text box, as shown in Figure 4-4.

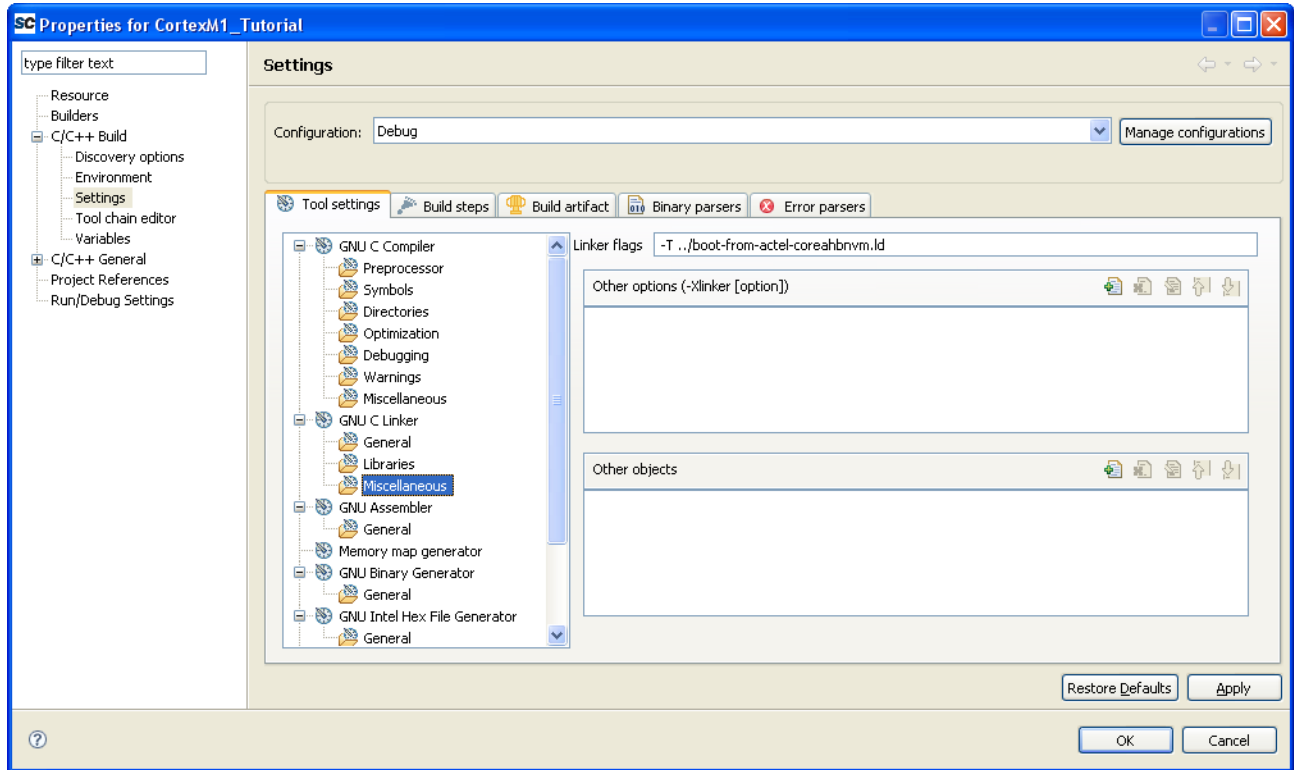


Figure 4-4 · Specifying Linker Script Location

Using boot-from-actel-coreahbnvm.ld and boot-from-intel-flash.ld

When using the boot-from-actel-coreahbnvm.ld or boot-from-intel-flash linker script, you must compile the project using the `-mlong-calls` compiler flag. This is needed so the compiler can create the call to `main()` from within the `_start()` function. These linker scripts are used when your boot code is at address `0x0` and the processor runs the application code from another memory location; typically `0x10000000` or higher.

To add the compiler flag:

1. Right-click the project and choose **Properties**.
2. Select **C/C++ Build > GCC C Compiler**.

3. Add `-mlong-calls` at the end of the command, as shown in Figure 4-5.

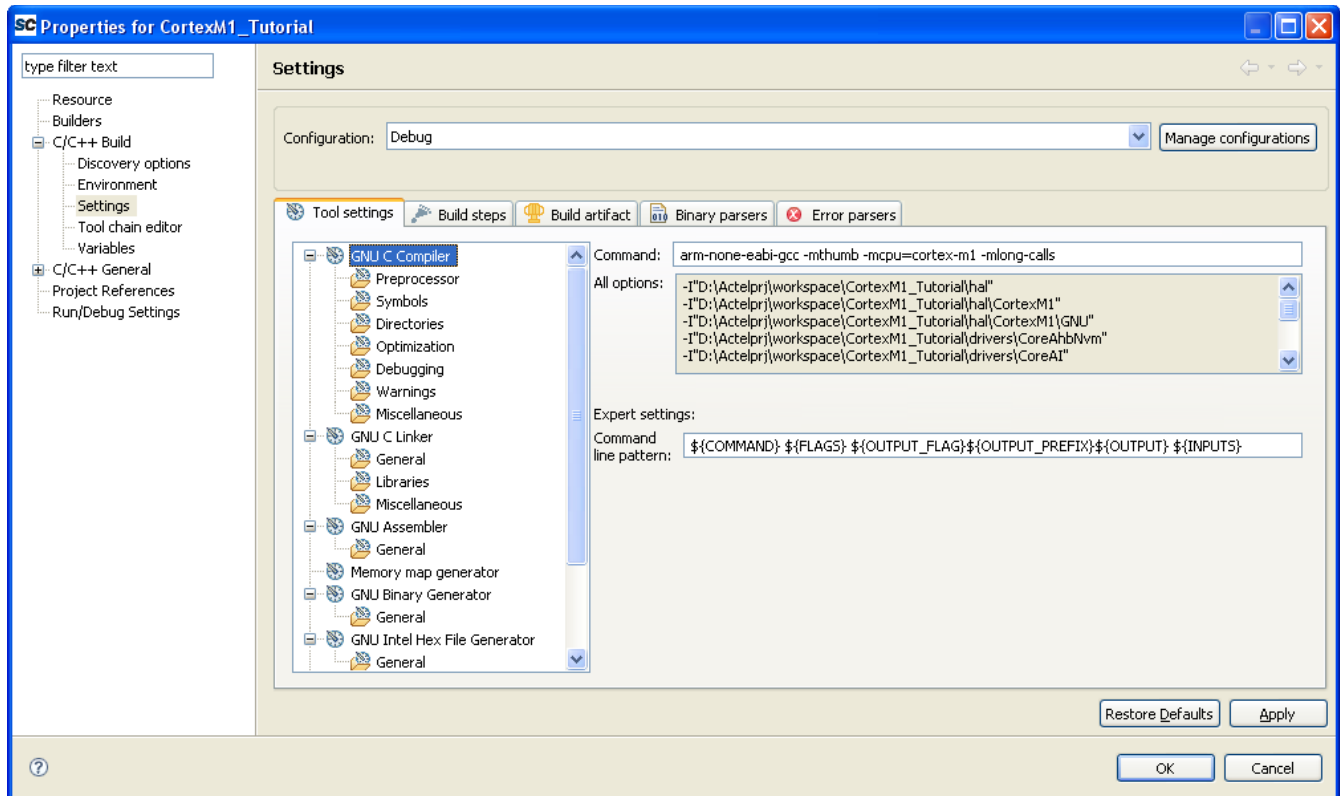


Figure 4-5 · Setting `-mlong-calls` Compiler Flag

Memory Map and Interrupt Numbers

You must create a system header file that contains the addresses of the peripherals in your system as well as the interrupt numbers. While this is not strictly necessary to create a working Cortex-M1 system, Actel recommends this practice so that application code is more readable and flexible in case any changes are needed. This information can be obtained by looking at the system in SmartDesign. The `<component>.xml` file from the `<$project>/component/work/<component>` folder can be opened to view the memory map of the FPGA design.

Startup Files

Cortex-M1 allows writing an application in C, including the boot code which usually requires some assembly code. The code presented here replaces the usual GNU crt0 boot code. The boot code is split between three files:

- `vector_table.s`
- `sys_boot.c`
- `default_handlers.c`

The `vector_table.s` file contains the Cortex-M1 vector table. The vector table is implemented in assembly code. The vector table contains the initial value of the Cortex-M1 stack pointer and the addresses of the various exception handlers. The `sys_boot.c` file contains the `_start()` function, which is called on power-up and warm reset. The `main()` function is called from `_start()` once the executable code has been relocated (if required), the data section has been initialized, and the non-initialized variables have been set to 0.

The `sys_boot.c` file is written to work in conjunction with one of five possible linker scripts:

- `boot-from-actel-coreahbnvm.ld`
- `boot-from-intel-flash.ld`
- `only-ram-memory.ld`
- `run-from-actel-coreahbnvm.ld`
- `run-from-intel-flash.ld`

These linker scripts provide symbols used in the `sys_boot.c` file for locating the various sections of the executable image. The linker scripts are provided in the SoftConsole installation in the `<SoftConsole v3.1>\src\Cortex-M1\linker-script-examples` directory. For information about the purpose of each linker script and how to modify them to match your hardware, refer to [“Linker Scripts for Specifying Program Code and Data Memories” on page 30](#).

The default `handlers.c` file contains the implementation of the default exception handlers. The Cortex-M1 HAL uses weak linking for exception handlers. This allows easy implementation of exception handlers without having to modify the vector table. Using this method, an exception handler can simply be implemented by creating a function with a predefined name. The address of the user-implemented exception handler is automatically included in the vector table as long as the predefined handler function names are used. [Table 4-2](#) lists the predefined exception handler names.

Table 4-2 · Predefined Exception Handler Names

Exception Type	Handler Function Name
NMI handler void	<code>cortex_nmi_handler(void)</code>
Fault handler void	<code>cortex_fault_handler(void)</code>
System service call with SVC instruction void	<code>cortex_sv_call(void)</code>
Pendable request for system service void	<code>P cortex_pend_sv(void)</code>
System tick timer void	<code>cortex_systick_isr(void)</code>
External interrupt 0 void	<code>cortex_irq_0_isr(void)</code>
External interrupt 1 void	<code>cortex_irq_1_isr(void)</code>
External interrupt 2 void	<code>cortex_irq_2_isr(void)</code>
External interrupt 3 void	<code>cortex_irq_3_isr(void)</code>
External interrupt 4 void	<code>cortex_irq_4_isr(void)</code>
External interrupt 5 void	<code>cortex_irq_5_isr(void)</code>
External interrupt 6 void	<code>cortex_irq_6_isr(void)</code>
External interrupt 7 void	<code>cortex_irq_7_isr(void)</code>

Note: The following exception sources are not available in the current Cortex-M1 implementation: system tick timer, external interrupt 1 to 7.

Project Settings for Core8051s

SoftConsole provides a way for you to specify options for the application. In the case of a Core8051s project, these include paths for folders related to the code, memory models, and the memory limits for the target application. These memory limits include the code memory size, the external data memory size, internal data RAM size, minimum stack allocation, a pseudo stack, and whether to pack the internal data memory.

Setting Memory Limits

To set memory limit options:

1. Right-click the project name in the C/C++ Projects View box and select **Properties**.
2. Select and expand **C/C++ Build** in the Properties window.
3. Select **Settings**.
4. In the **Tool Settings** tab, under **SDCC Compiler**, select **Memory Options** (Figure 4-6).

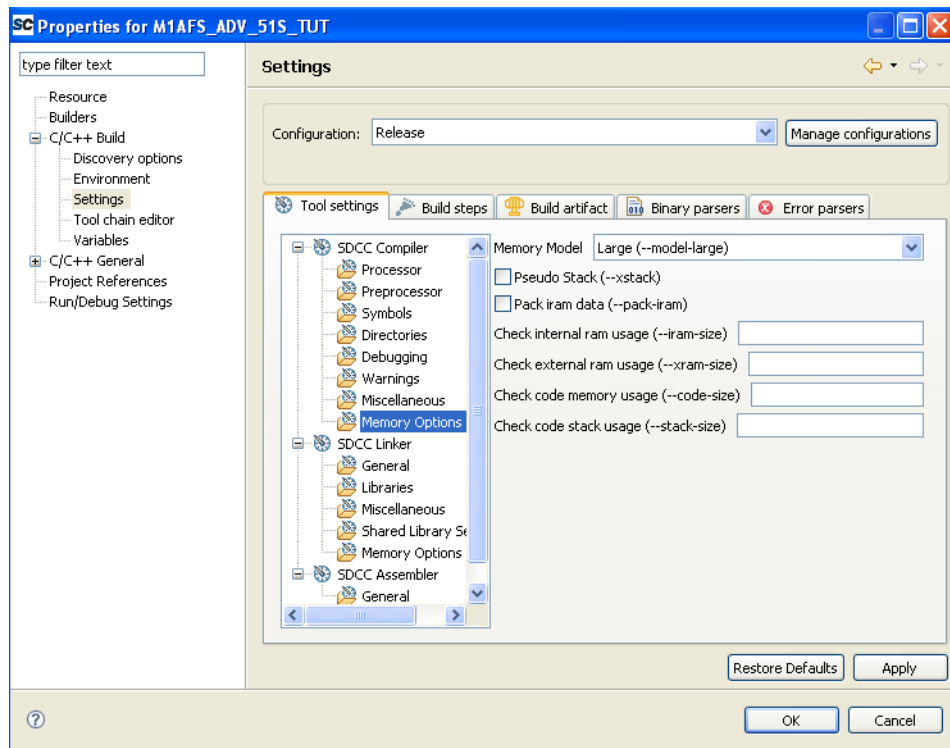


Figure 4-6 · Memory Options for 8051

5. Select the **Pseudo Stack** check box to enable the `--xstack` option. This option creates a pseudo stack in the first page (256 bytes) of external data memory. Port2 (of a standard 8051) is used for the high byte of the address. Core8051s maps the APB peripherals into the upper 4KB of external memory space and therefore is not compatible with this option.
6. Select the **Pack iram data** check box to enable the `--pack-iram` option. This option tells the linker to use unused register banks for data and to pack data, idata, and the stack together.

7. To set the internal RAM limit, select the **Check internal ram usage** check box and type in the size of the internal data memory. This should be 0x100 (256 bytes) for Core8051s. This causes the linker to check whether the internal data memory usage is within this limit.
8. To set the external RAM limit, select the **Check external ram usage** check box and type in the size of the external data memory (depends on your hardware design). Checking this box causes the linker to check whether the external data memory usage is within this limit.
9. To set the code memory limit, select the **Check code memory usage** check box and type in the size of the code memory (depends on your hardware design). Checking this box causes the linker to check whether the code memory usage is within this limit.
10. To set the minimum stack limit, select the **Check code stack usage** check box and type in the size of the stack. This causes the linker to check that there is at least the specified amount of space available for the stack.

Memory Models

In addition to these options, you can choose between a large and a small memory model. In the large model, all variables declared without a storage class will be in external RAM. This includes all parameters and local variables (for non-reentrant functions). When the small model is used, all variables declared without a storage class will be in internal RAM.

The memory model size (large or small) must be selected in two places: for the compiler and for the linker.

To select the memory model size for the compiler:

1. Right-click the project name in the C/C++ Projects View box and select **Properties**.
2. Select and expand **C/C++ Build** in the Properties window. Select **Settings**.
3. In the Tool Settings tab, under SDCC Compiler, select **Memory Options**.
4. In the Memory Model box, use the pull-down menu and select either Large or Small (Figure 4-7).
5. Click **Apply** and **OK**.

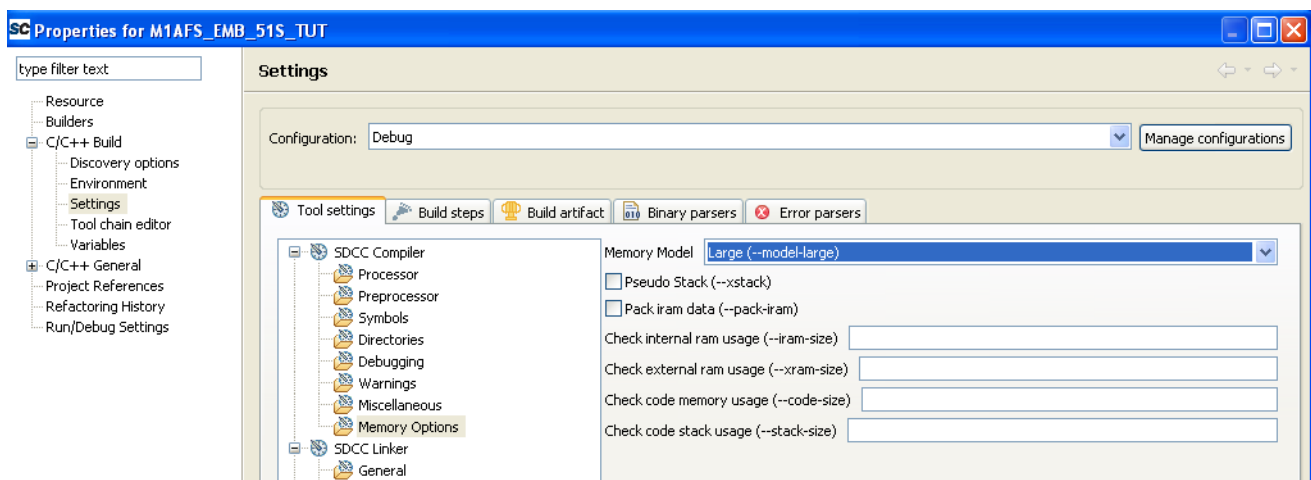


Figure 4-7 · Setting the Compiler Memory Model To Large

To select the memory model size for the linker:

1. Right-click the project name in the C/C++ Projects View box and select **Properties**.
2. Select and expand **C/C++ Build** in the Properties window. Select **Settings**.
3. In the Tool Settings tab, under SDCC Linker, select **Memory Options**.
4. In the Memory Model box, use the pull-down menu and select either Large or Small (Figure 4-8).
5. Click **Apply** and **OK**.

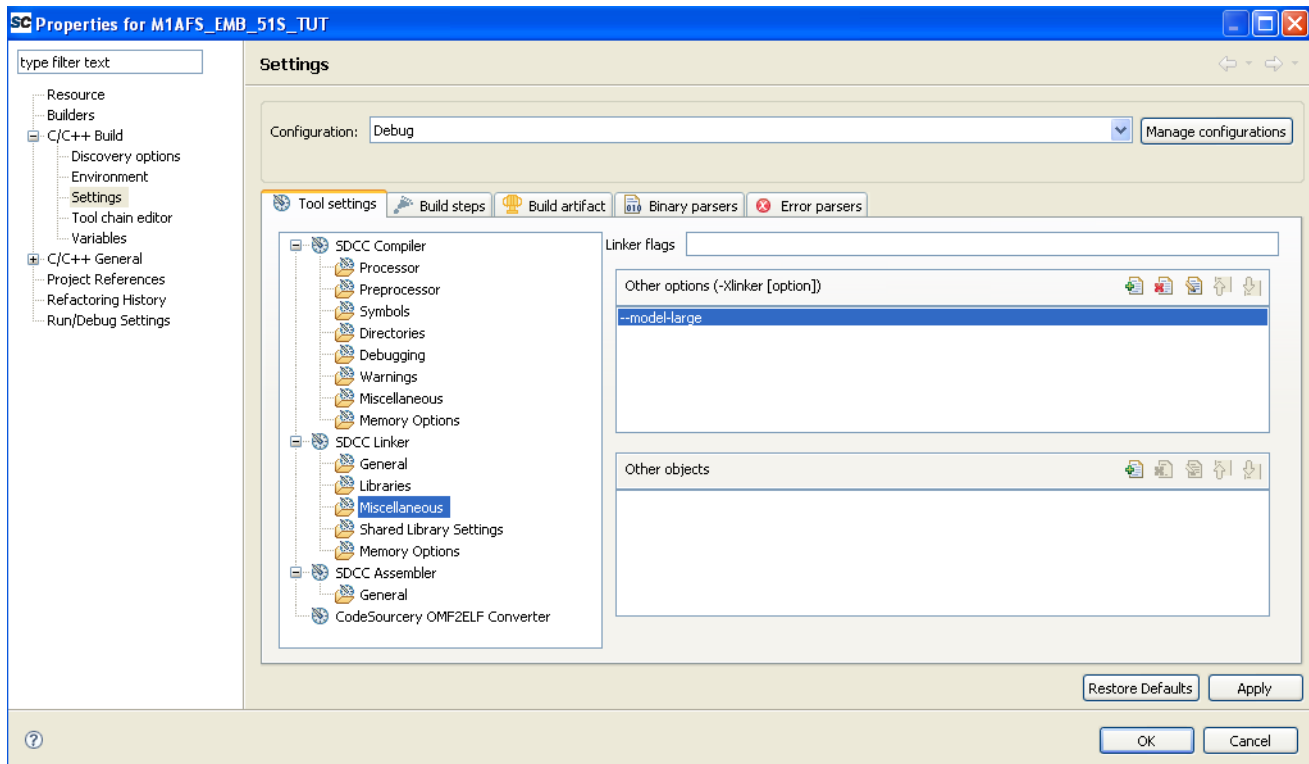


Figure 4-8 · Setting the Compiler Linker Option To Large

Both the compiler and the linker settings must be set to the same value. Setting different values can give unpredictable results.

Setting Paths to Files

If you have source code or header files in folders other than the project folder (including folders inside the project folder) it is necessary to set paths to these folders.

To set paths to files:

1. Right-click the project name in the C/C++ Projects View box and select **Properties**.
2. Select and expand **C/C++ Build** in the Properties window. Select **Settings**.
3. In the Tool Settings tab, under SDCC Compiler, select **Directories**.
4. Click the + icon to add a path. This will open the Add directory path box (Figure 4-9)
5. Click **File system** to specify a path outside of the workspace. Click **Workspace** to specify a path within your current workspace.

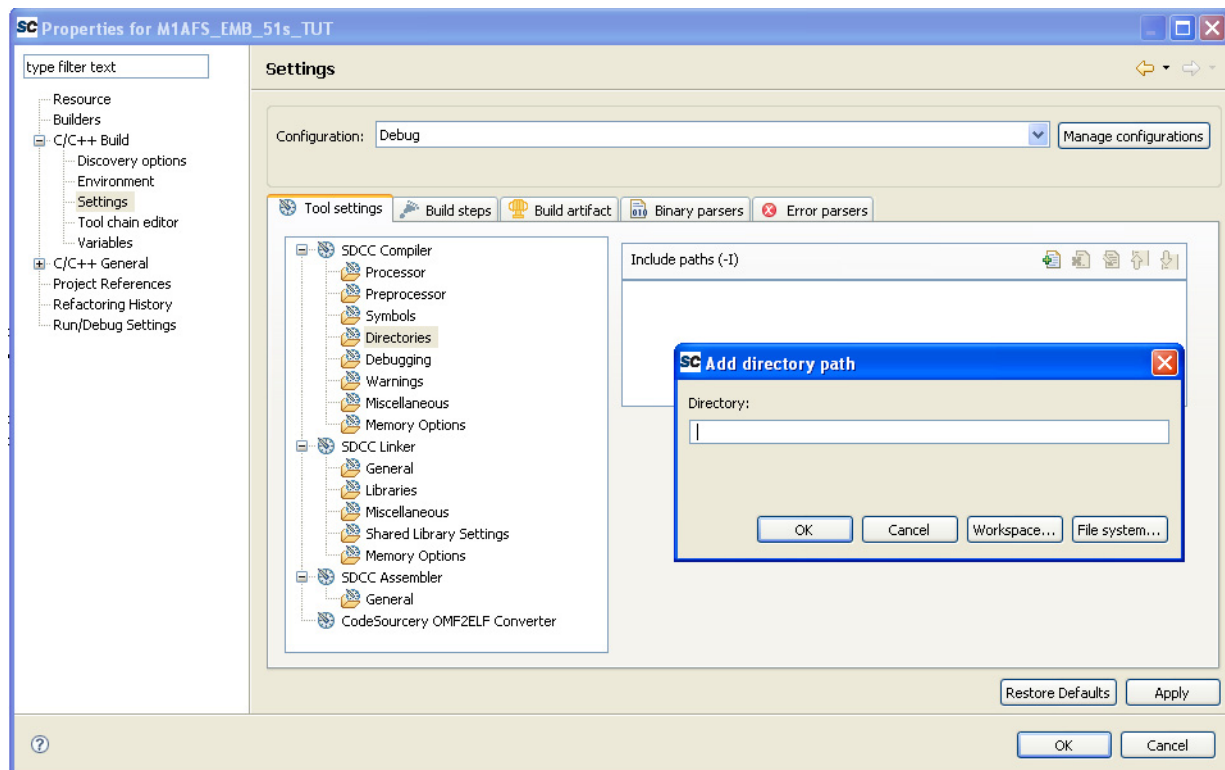


Figure 4-9 · Setting the Directory Paths for the SDCC Compiler

You can click the X icon to delete a path.

Startup Files

The compiler triggers the linker to link certain initialization (startup) modules from the runtime library. Only the necessary modules are linked. One of these modules performs static and global variable initialization before the function main is invoked. In general, there is no user intervention required to link in startup files for the SDCC compiler.

Using Libraries

The linker will normally link to standard C libraries to provide standard library functions. You simply need to reference the appropriate header file(s) using `#include` statements.

You can disable the automatic linking in of startup and/or library files by selecting the SDCC Linker General setting in the project properties box (Figure 4-10).

- Select the **Do not use standard start files (-nostartfiles)** check box to disable linking in of startup files.
- Select the **Do not use default libraries (-nodefaultlibs)** check box to disable linking in of standard library files.
- Select the **No startup or default libs (-nostdlib)** check box to disable linking of startup files and standard library files.
- Select the **Omit all symbol information (-s)** check box to disable all symbol information.
- Select the **No shared libraries (-static)** check box to disable the shared libraries.

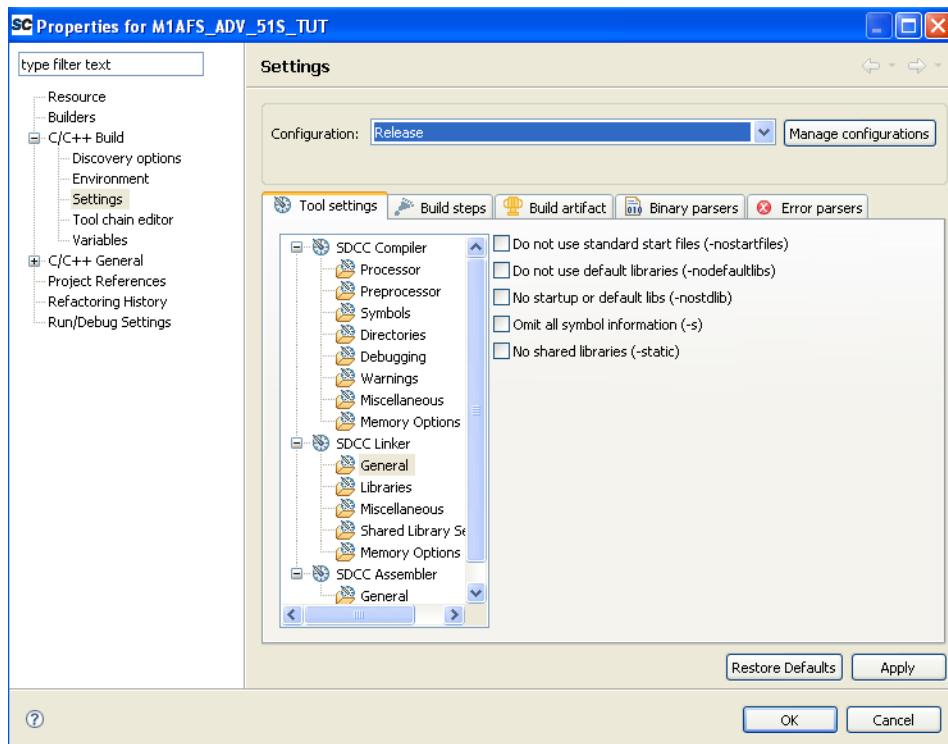


Figure 4-10 · General Linker Settings

Standard Library Functions for I/O (printf, scanf, etc.)

Standard library functions for I/O, such as `printf` and `scanf`, default to using the standard input (stdin) and standard output (stdout) devices. The SDCC compiler has no knowledge of the intended stdin and stdout devices for any given application. The software developer must write a `getchar()` routine for the stdin device and a `putchar()` routine for the standard output device. Any initialization required of the stdin and stdout device(s), such as with a UART, must be called prior to calling any function that uses stdin or stdout.

Math Functions

The SDCC library includes math functions by default and it is only necessary to include the math.h header file in your code.

Obtaining Header File Information from SmartDesign

Smart Design can be used to obtain information useful for making your header file for your I/O assignments.

1. In Libero IDE, open the SmartDesign component that contains Core8051s.
2. Click the SmartDesign tab from the main menu. Select **Show Memory Map/Data Sheet**. The datasheet and memory map document will open.
3. Click **Memory Map** in the table of contents or scroll down to the memory map. The Base Address shown for the peripherals is the base address of the peripheral on the APB3 bus, extended to 32 bits. However, note that on Core8051s, the APB3 bus has an external memory address starting at 0xF000. The address of the peripheral in external data memory space is the addition of the lower 16 bits of the APB3 base address (Figure 4-11) + 0xF000.

- CORE8051S_0

	Base Address
CoreTimer_0 : RegisterMap	0x00000000
CoreInterrupt_0 : RegisterMap	0x00000100
CoreGPIO_0 : RegisterMap	0x00000200
CoreUARTapb_0 :	0x00000300
CoreTimer_1 : RegisterMap	0x00000400
COREAI_0 : RegisterMap	0x00000600
CoreWatchdog_0 : RegisterMap	0x00000e00

[subsystem list, top of page](#)

Figure 4-11 · Typical APB3 Memory Map from SmartDesign

Building a Project

There are two types of output files created by the compiler: a debug version and a release version. The debug file includes the your application code along with additional information used by the debugger. The debug option creates a file, default.elf, and is placed in the *Debug* folder of the project. This code is used along with the FlashPro4 debugger to run your application in debug mode.

To create the debug version:

1. Right-click the project name and scroll down to Build Configurations.
2. Scroll to Set Active.
3. Select **Debug** to select a debug build (Figure 5-1).

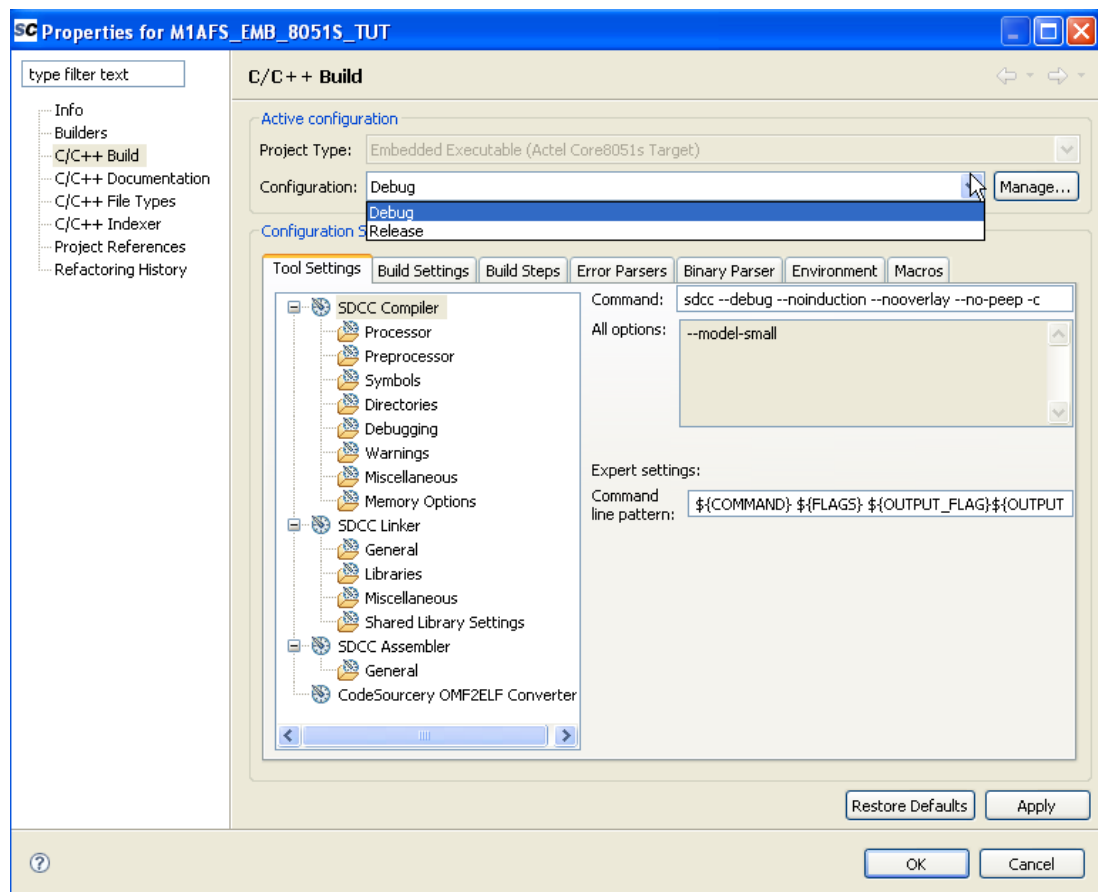


Figure 5-1 · Setting the Build Configuration

Creating the Release Version

The release file contains your application code in hex format and has the extension *.ihx. This file is placed in the *Release* folder of the project and should be used to program NVM code memory. The release file will have the name of the project with the extension *.ihx.

To create the release version:

1. Right-click the project name and scroll down to Build Configurations.
2. Scroll to Set Active.
3. Select **Release** to select a release build.

Types of Project Build Methods

There are three options for building the project (in addition to changing from release to debug, etc.). These are Build Project, Clean, or Build Automatically. These options are available from the Project pull-down menu.

- Build Automatically builds the project when you save a file. If this option is selected, the Build Project option is grayed out. The build process is incremental and only the components affected by modified files in the project are built.
 - Clean provides an option to build a single project or build all projects in the workspace. A clean build discards the previously built state and rebuilds all components.
 - Build Project builds the current project. This option is grayed out if Build Automatically is selected.
1. Set the build configuration to **Release** (see “Creating the Release Version”).
 2. Perform a project clean by selecting **Project** from the SoftConsole menu, then selecting **Clean** (Figure 5-2)

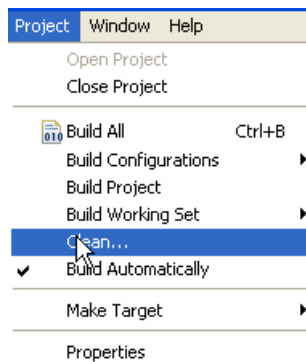


Figure 5-2 · Project Clean

3. The Clean box will open. You can clean all the projects in your workspace or only selected projects (Figure 5-3).

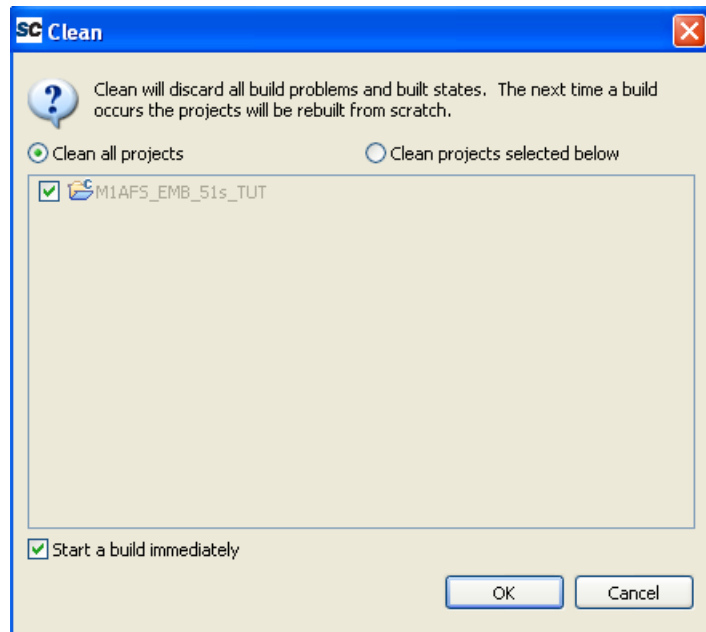


Figure 5-3 · Cleaning Options

4. The project will be compiled and linked, with results indicated in the Console window (Figure 5-4).

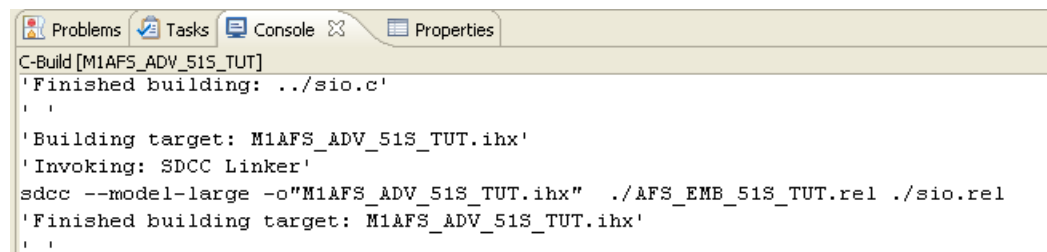


Figure 5-4 · Successful Completion of the Release Build

Debugging with SoftConsole

This chapter shows you how to debug using SoftConsole.

1. In Project Explorer, right-click the project and choose **Debug As**.
2. Select **Open Debug Dialog** (Figure 6-1). The Debug window opens.

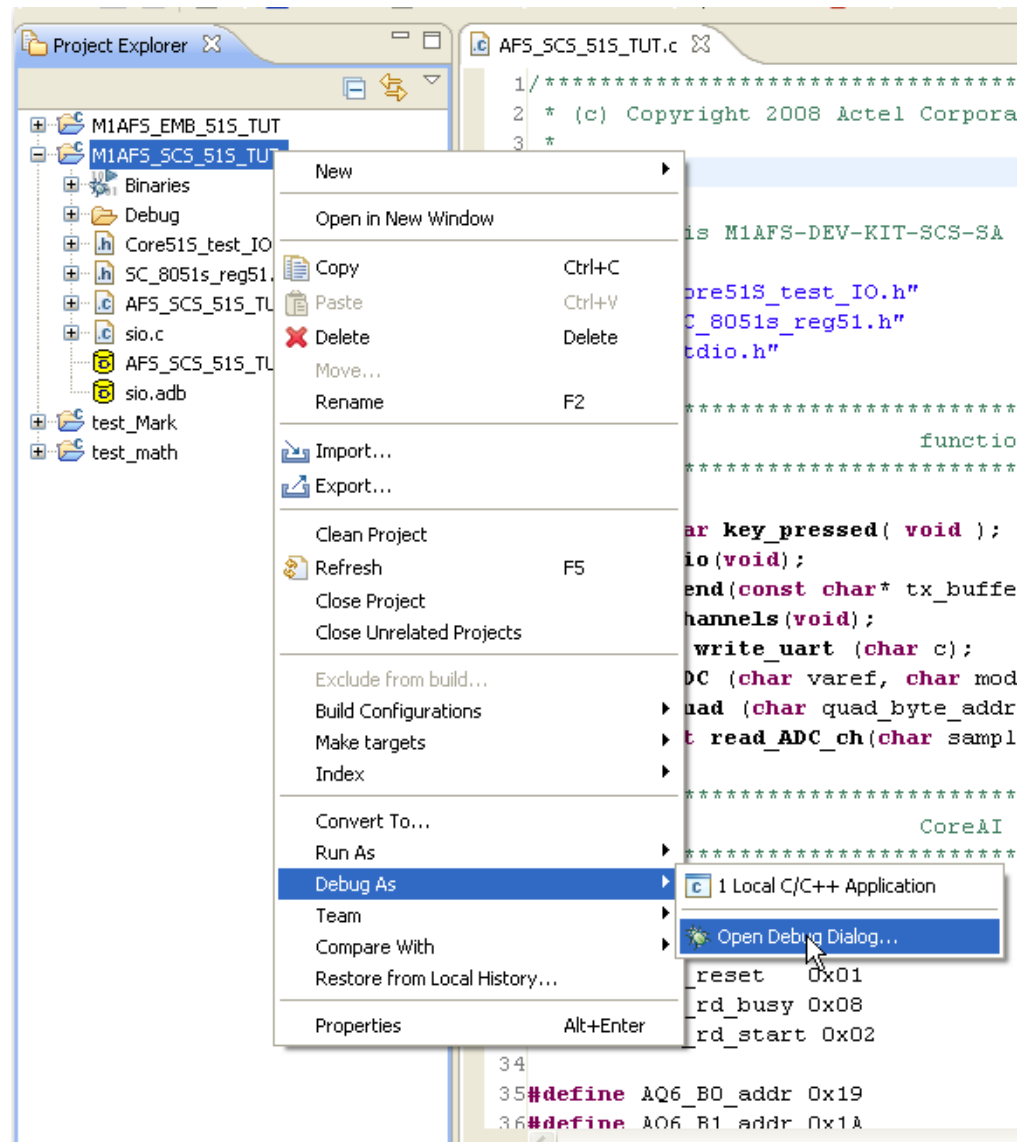


Figure 6-1 · Opening Debug Dialog

3. Right-click the relevant target (Actel Cortex-M3 eNVM Target, Actel Cortex-M3 RAM Target, Actel Cortex-M1 Target, or Actel Core8051s Target) and choose **New** (Figure 6-2 on page 48). SoftConsole automatically configures the debug utility.

Note: When targeting a CoreMP7 design, the debug utility must be configured manually. See “[Appendix A – Programming Flash Memory in Cortex-M1 Systems](#)” on page 61.

4. Click the **Debug** button. The Debug window will close and the debug sprite will launch.

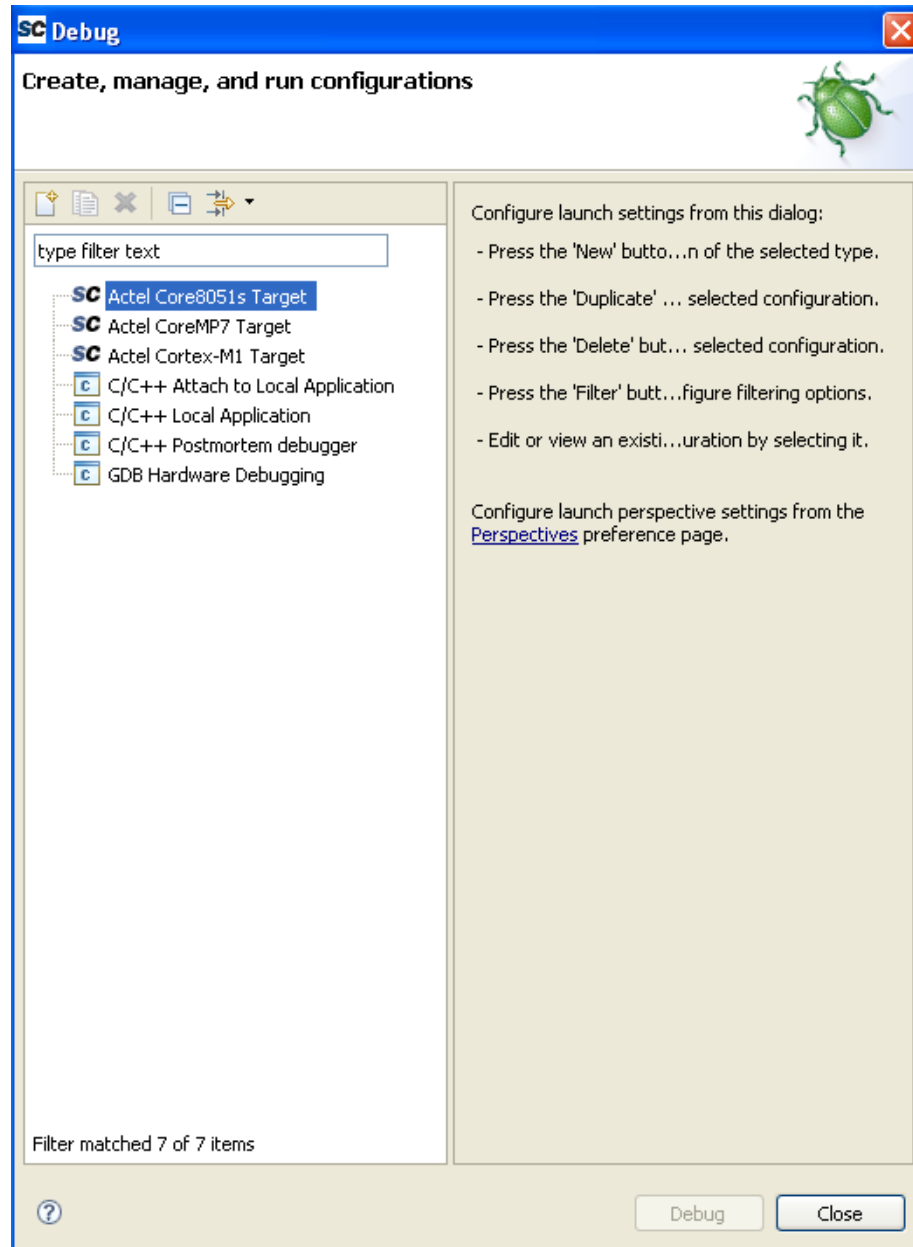


Figure 6-2 · Selecting the Debug Target

The sprite will begin to communicate with the debugger and results are displayed in the console view (Figure 6-3). The exact communications will vary based on the target processor. The file loading status is shown in the lower right area of the console view. (Figure 6-4).

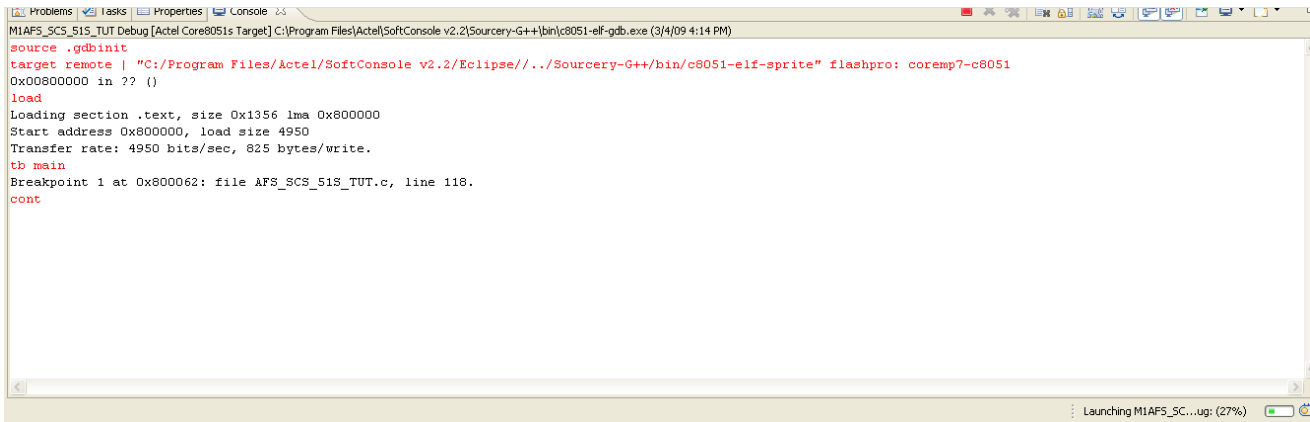


Figure 6-3 · Console Communications

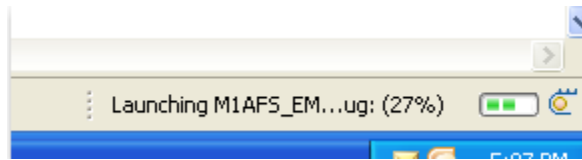


Figure 6-4 · File Loading Status

After the file loads, the Debug window (shown in Figure 6-2 on page 48) will disappear and the Confirm Perspective Switch window will open (Figure 6-5). Click the Yes button. The debug perspective will open.

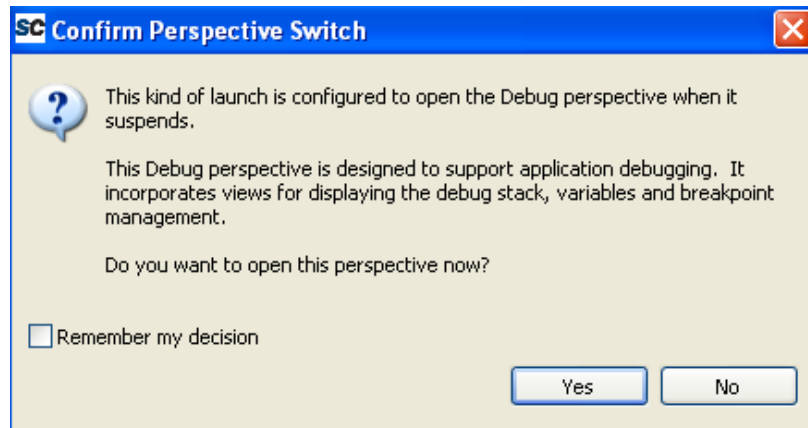


Figure 6-5 · Confirming Switch to Debug Perspective

Debug Perspective

The debug perspective is shown in [Figure 6-6](#). It consists of a number of views distributed over five regions of the perspective. Each view has a tab with which to select that view.

The top left region of the debug perspective contains the debug view. The top right region of the debug perspective contains the Variables, Breakpoints, Registers, and Modules views. The far right middle region contains the Outline view. The middle left and center region of the debug perspective contains the Source Code view and Editor. The bottom region of the debug perspective contains the Console, Tasks, Problems, and Memory views.

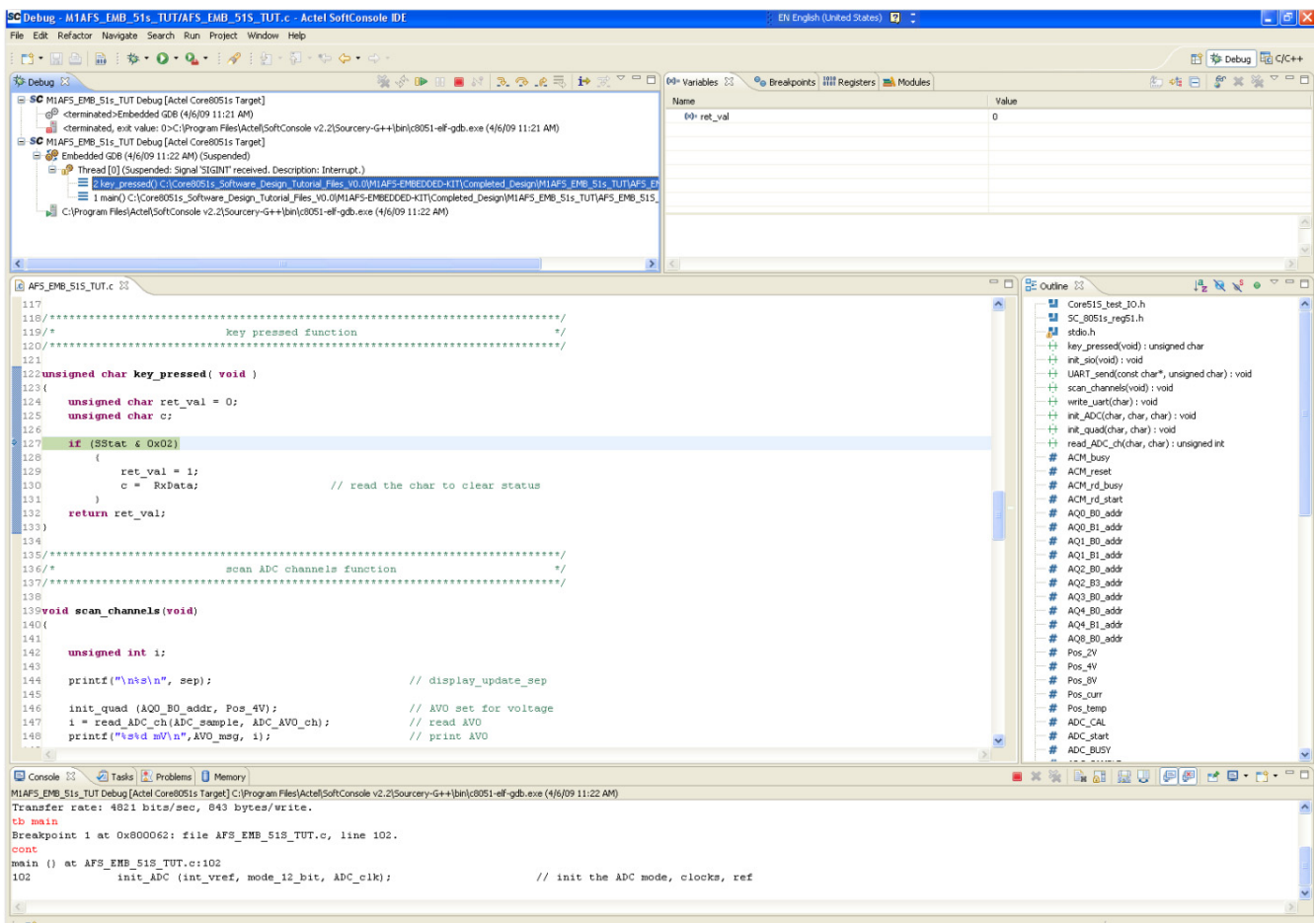


Figure 6-6 · Debug Perspective

Debug View

The debug view shows the debug utility in use (ActelCore8051s Target), the debugger in use (Embedded GDB), the current thread (application code), and the debugger sprite (c8051-elf-gdb.exe). [Figure 6-7](#) shows the ActelCore8051s Target debug utility, the Embedded GDB debugger, the current application code thread ('main ()'), and the c8051-elf-gdb.exe debugger sprite.



Figure 6-7 · Debug View

Variables, Breakpoints, Registers, and Modules Views

The top right area of the debug perspective contains the Variables, Breakpoints, Registers, and Modules views.

The Variables view shows variables in your program ([Figure 6-8](#)). You can view and modify variables, which can be useful for debugging code

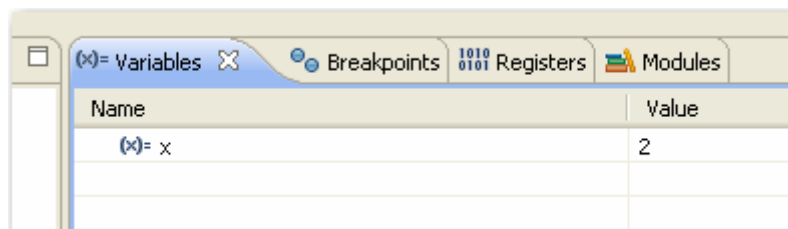


Figure 6-8 · Variables View

The Breakpoints view shows breakpoints in your program and their current enabled/disabled status ([Figure 6-9](#)).

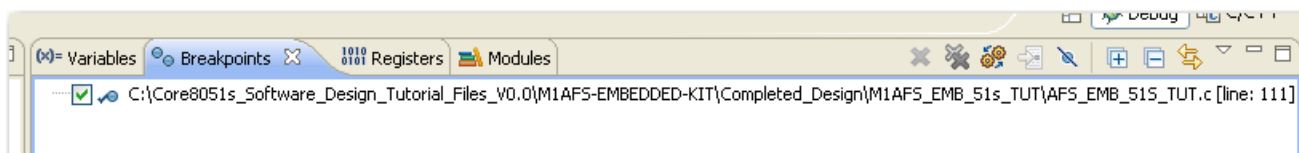


Figure 6-9 · Breakpoints View

A breakpoint in your code is set by clicking on a line of code in the Editor (while the code is not executing). This highlights the line of code. Double-click the line number to the left of the line of code. A checkmark and a circle appear to the left of the line number and a breakpoint appears in the Breakpoints view.

A breakpoint shown in the Breakpoints view can be enabled or disabled by clicking its square icon in the Breakpoints view, or by right-clicking its description in the Breakpoints view. Disabled breakpoints are indicated by a white circle. Enabled breakpoints are indicated by a colored circle.

The Registers view shows information about processor registers. Values that have changed are highlighted in the Registers view when your program stops.

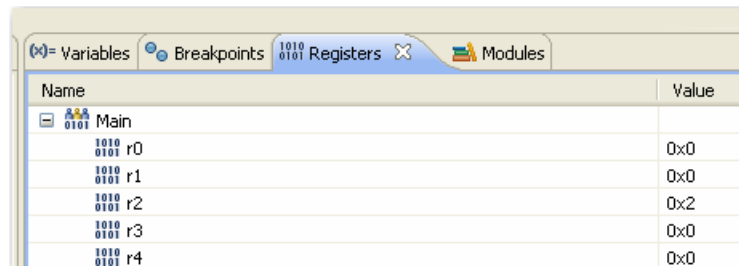


Figure 6-10 · Registers View

The Modules view displays information about the modules loaded in the current debug session, including executables and shared libraries (Figure 6-11). The view has two areas: the modules tree and a detail pane. The detail pane displays the information for the selected module. Expanding a module enables you to view the module's internal functions, global variables, and source files.

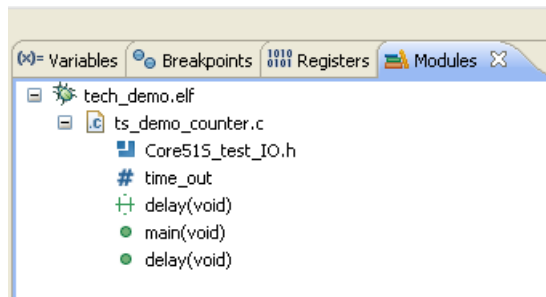


Figure 6-11 · Modules View

Source Code View

The middle left and center portion of the debug perspective contains the Source Code view and Editor (Figure 6-12).

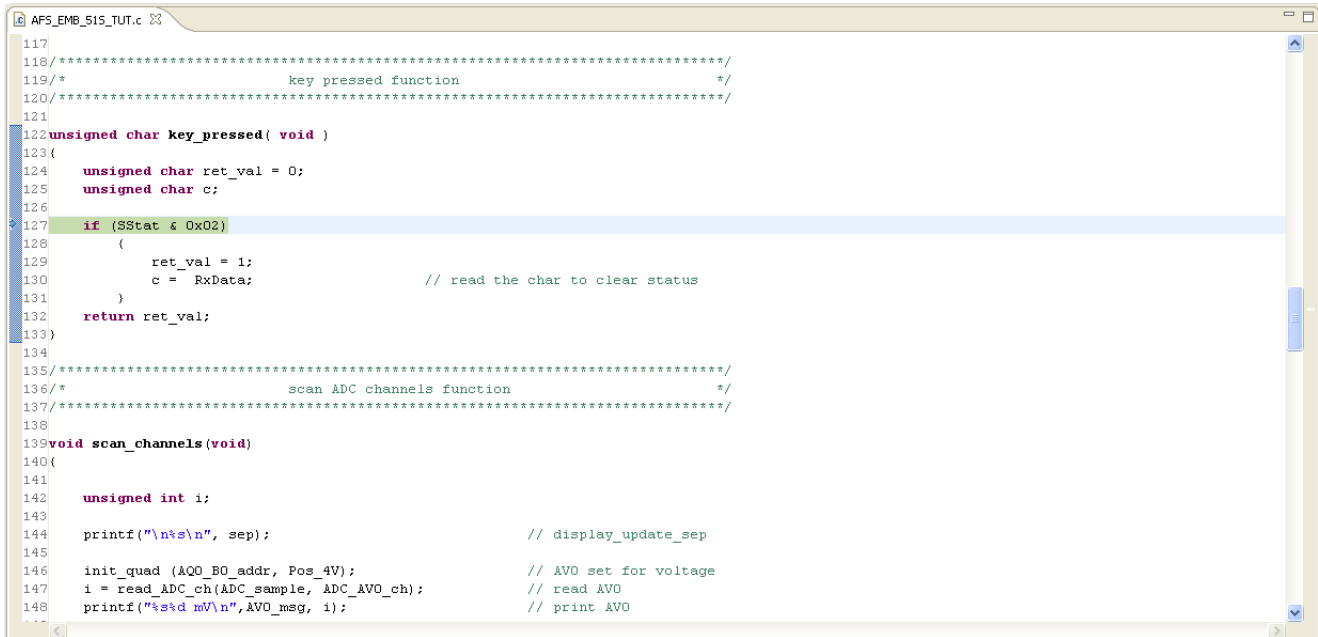


Figure 6-12 · Source Code View

This view displays the application's source code. When the application code is not executing, the next statement in the code to be executed is highlighted. This will be the case when the debugger has just been launched, when you have hit a breakpoint, when you have clicked the Suspend icon, or when a single step operation has finished.

A breakpoint in your code is set by clicking a line of code (while the code is not executing). This highlights the line of code. Double-click the line number to the left of the line of code. A checkmark and a circle appear to the left of the line number and a breakpoint appears in the Breakpoint view.

The Debug view uses the same editor used in the C/C++ perspective of SoftConsole and thus has the same editing capabilities.

Outline View

The middle far right side contains the Outline view (Figure 6-13).

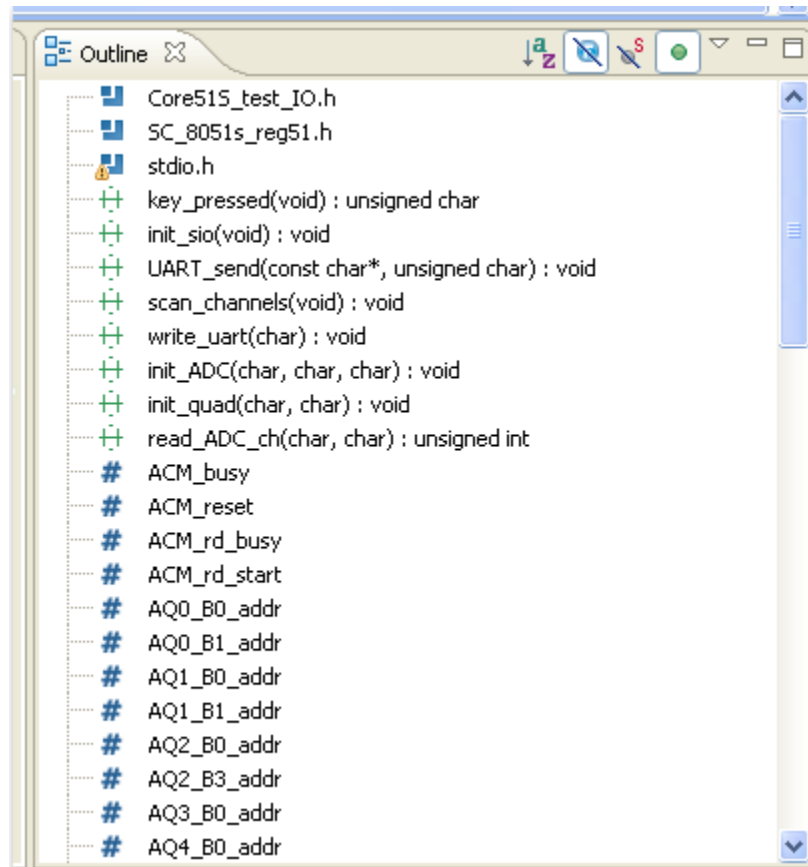


Figure 6-13 · Outline View

The Outline view is associated with the editor in the Source Code view. An outline of the structure of the code shown in the Source code view is shown in the Outline view. It contains references to include files, function prototypes, symbol definitions, and functions. Clicking a reference in the Outline view causes the Source Code window to display the portion of the source code that contains that item. It is a very quick way to advance the cursor to specific points in your code.

Console, Tasks, Problems, and Memory Views

The bottom portion of the debug perspective contains the Console, Tasks, Problems, and Memory views.

The Console view shows status information when launching the debugger. You can enter GDB commands in the Console view if the code is suspended by selecting **arm-none-eabi-gdb** in the Debug view. A list of GDB commands is available under the SoftConsole installation: `<SoftConsole install folder>\Sourcery-G++\share\doc\arm-2007q1-21-arm-none-eabi\pdf\gdb.pdf`.

The Task view is a place to store a list of tasks (Figure 6-14). You can add a task by right-clicking anywhere in the Task field and selecting **Add Task**. Alternately you can click the **ADD Task** icon (clipboard with a + sign). The Add Task box opens and allows you to enter a description of the task and set the priority to low, normal, or high. You can also mark the task as completed in this box.

A task can be edited by clicking in any Task field other than Description. This highlights the task. Right-click and select **Properties** to open the Add Task box. Edit the task in the ADD Task box.

A task can be deleted by highlighting the task field and using clicking the **Delete** icon or by right-clicking and selecting **Delete**.

A task can be marked as completed by highlighting the task, right-clicking, and selecting **Mark Completed**.

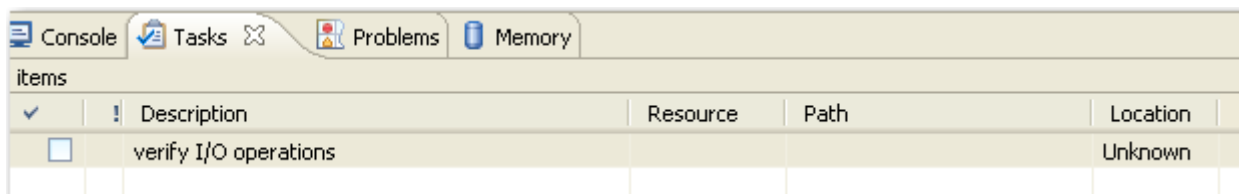


Figure 6-14 · Task View

The Problems view displays system-generated information, warnings, or errors.

The Memory view of the Debug perspective lets you view and modify the contents memory (Figure 6-15). The Memory view contains two panes: the Monitors pane and the Renderings pane.

Each monitor represents a section of memory specified by its base address location. Each memory monitor's data (rendering) can be displayed in different data formats. The debugger supports five data formats, hexadecimal (default), ascii, signed integer, and unsigned integer. The default format is displayed automatically in the monitor. It's possible to use expressions for the memory monitor.

The Monitors pane displays the list of memory monitors added to the debug session. The content of the Renderings pane is controlled by the selection in the Monitors pane. The Memory Renderings pane can be configured to display two renderings simultaneously.

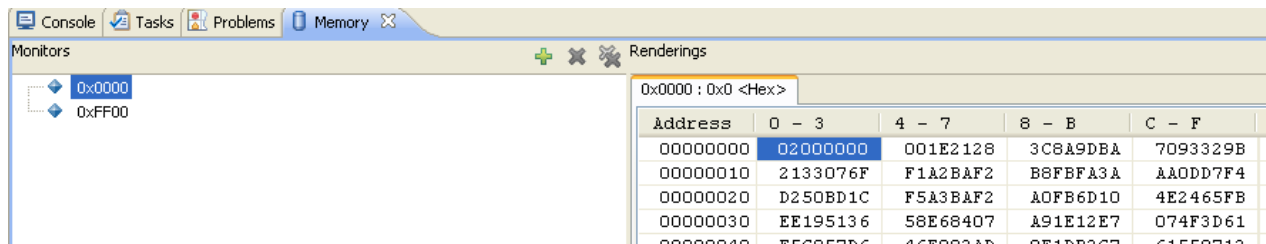


Figure 6-15 · Memory Monitor and Renderings Example

Clicking the + sign in the Monitors pane opens the Memory Monitor box, allowing you to specify a base address for the monitor. Clicking a monitor displays the memory contents for that monitor in the Renderings pane.

Clicking the X sign in the Monitors pane deletes the currently selected (highlighted) the memory monitor. Clicking the XX sign in the Monitors pane deletes all the memory monitors.

Some processors used with SoftConsole have separate code and data memory. A prefix for the memory address might be needed to distinguish between code and data memory spaces when using the memory monitors and renderings. Consult the SoftConsole release notes for specific details, as such prefixes may be processor-specific.

Running the Application

The Resume, Suspend, and Terminate icons are to the right of the Debug view's tab. Clicking the Resume icon (Figure 6-16) causes your program to run from its current position.



Figure 6-16 · Resume, Suspend, and Terminate Icons

When the debugger is first started, the program counter is set to the first line of code in your application.

The Suspend icon (middle icon in Figure 6-16) causes your executing user program to stop running, returning control of your application to the debugger.

The Terminate icon (right icon in Figure 6-16) stops your application code and the debug sprite from running.

Single Stepping

The single step icons (Figure 6-17) are located above the Debug view, to the right of the Resume, Suspend, and Terminate icons.



Figure 6-17 · Step Into, Step Over, and Step Return Icons

Clicking the Step Into icon causes the debugger to execute the current line of code. If the current line of code is a function call, the debugger advances to the first line of code in that function.

Clicking the Step Over icon executes the current line of code. If the current line of code is a function call, the function is executed and the debugger advances to the next line of code after the function call. This allows you to step through a block of code without having to step through all of the functions called from within that block.

The Step Return icon provides a method for finishing execution of a function and advancing the debugger to the next line of code after the function call. It effectively converts a Step Into mode inside a function to a Step Over mode.

Viewing Assembly Code

Assembly code for the application can be viewed by selecting **Window > Show View > Disassembly** in the main toolbar of the Debug perspective (Figure 6-18)

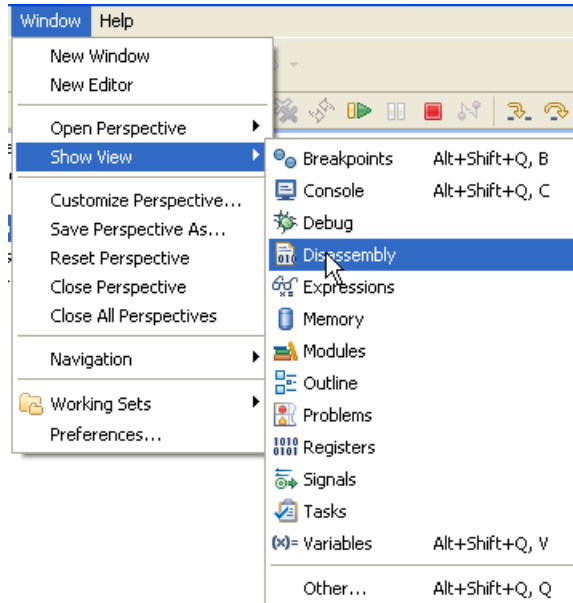


Figure 6-18 · Enabling Disassembly View

Figure 6-19 shows an example of the Disassembly view.

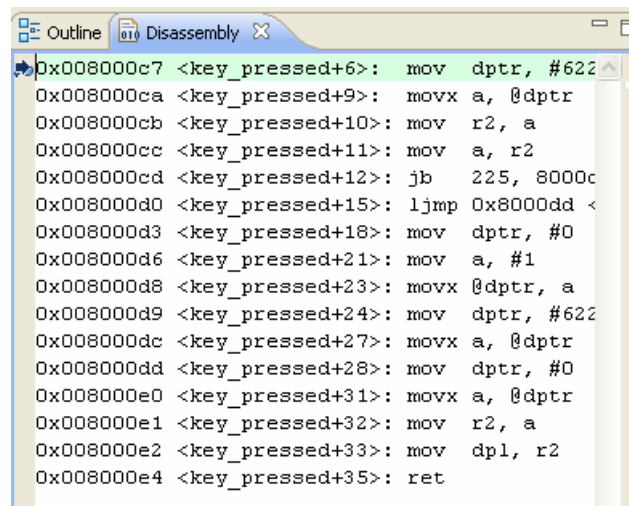


Figure 6-19 · Disassembly View

Stepping in Assembly Code

Clicking the Instruction icon to the right of the Step icons (Figure 6-20) toggles the debugger between C language and assembly language stepping.



Figure 6-20 · Stepping Instruction Icon

The Disassembly window opens, if it is not currently open, when assembly language stepping is chosen.

Modifying Code

The best strategy for modifying code is to suspend the debugger if the code is currently running. Then click the Terminate icon to terminate the debugger sprite. Click the C/C++ button to revert to the code development perspective. Make the code changes, compile, and launch the debugger.

Exiting the Debugger

Prior to exiting the debugger you should terminate the application by clicking the Terminate icon (above the Debug view). Failure to do so can leave a debug sprite running and interfere with using the debugger later.

Programming Flash Memory in Cortex-M1 Systems

Overview

SoftConsole can load an executable file into the program memory of a Cortex-M1 target system, for the purposes of debugging or executing the program on that system.

The program memory of the target system can be flash or SRAM program memory, whichever type is located at base address 0x00000000 in the target system's memory map. Flash memory can be external or embedded:

- External flash memory (external to the FPGA) accessed via CoreMemCtrl
- Actel Fusion embedded flash memory (eNVM) accessed via CoreAhbNvm

Minimum Requirements

The minimum requirements for flash memory programming with SoftConsole are as follows:

- SoftConsole v3.1 or higher
- A Cortex-M1 target system with flash program memory at base address 0x00000000 (flash program memory connected to CoreAHBLite slot 0)
- A GNU C linker script that accurately describes the location, size, and type of the flash program memory in the Cortex-M1 target system. A number of example linker scripts are provided with SoftConsole.
- A flash device description file. This file describes the sequence of programming commands for a particular Flash device. A number of Flash device description files are provided with SoftConsole.

Flash Memory Programming Flow

This section describes the flash memory programming flow using SoftConsole. Detailed step-by-step description of the flash memory programming flow is included in [“Appendix A – Programming Flash Memory in Cortex-M1 Systems”](#) on page 61.

Create a SoftConsole C Project

1. Create a SoftConsole C project.
 - Project type: Executable
 - Toolchain: Cortex-M1
2. Add the following files to the project:
 - Any source files (C or assembler), or Actel IP core software drivers that may be required to build a Cortex-M1 application program
 - The Actel Cortex-M1/GNU Hardware Abstraction Layer (HAL), which the drivers rely upon to access the hardware and which also provides boot code for the Cortex-M1
 - A suitable linker script. A number of example linker scripts are provided with SoftConsole. Each linker script is compatible with the Cortex-M1/GNU HAL, and a typical Cortex-M1 target system. Each linker script specifies a Flash device description file that is compatible with the target system's program memory. Refer to [“Linker Scripts for Specifying Program Code and Data Memories”](#) on page 30 for a description of linker scripts that are included with SoftConsole and instructions on how to modify the linker scripts to match the memory in your system.
3. Add include paths to the GNU C Compiler > Directories settings in the project properties for any HAL or Driver subfolders that contain source files.

4. Build the project and create the executable file. Building the project also creates a memory-map.xml file, which is used to pass the structure of the memory map and the flash device to the GDB debugger and the Cortex-M1 sprite.

Create a Debug Launch Configuration for the Project

Create a debug launch configuration for the project, as described in “[Debugging with SoftConsole](#)” on page 47 to support loading and debugging of programs for the Cortex-M1 processor via PlashPro3. The executable file must be loaded into program memory before debugging can take place.

Load the Executable File to the Flash Program Memory

Launch a debug session to load the executable file to flash program memory. Messages in the Console view will indicate that flash programming is in progress ([Figure 7-1](#)).

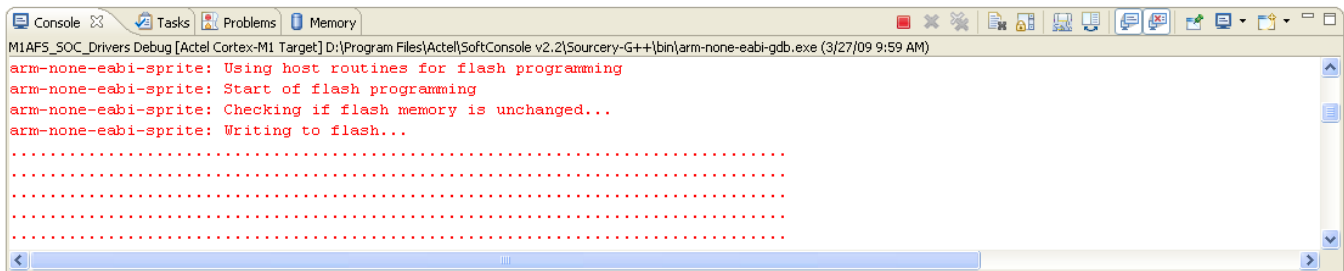


Figure 7-1 · Messages in Console View

Run and Debug the Program in Flash Memory

When flash programming is complete, the Confirm Perspective Switch dialog box opens, as described in “[Debugging with SoftConsole](#)” on page 47. The Debug perspective opens with the program ready to continue running from the temporary main() breakpoint. All the usual debug operations are available, including stepping, setting and running to breakpoints, viewing register and memory content, etc.

Notes:

1. Cortex-M1 supports the use of only two hardware breakpoints at any one time. So, when debugging a program that is running from flash, you may set many breakpoints but you **MUST ONLY ENABLE TWO BREAKPOINTS AT ANY ONE TIME** (that is, check only two breakpoints in the Breakpoints view). Remember also that GDB Step commands use a breakpoint, which counts towards the limit of two hardware breakpoints.
2. Flash memory blocks are erased and rewritten only if there is any change to the contents. If no changes are made to the executable between debug sessions, SoftConsole does not rewrite the flash memory. This reduces the level of wear that the flash memory is subjected to over multiple debug sessions.

Appendix A – Programming Flash Memory in Cortex-M1 Systems

Overview

SoftConsole can load an executable file into the program memory of a Cortex-M1 target system, for the purposes of debugging or executing the program on that system.

The program memory of the target system can be flash or SRAM program memory, whichever type is located at base address 0x00000000 in the target system's memory map. Flash memory can be external or embedded:

- External flash memory (external to the FPGA) accessed via CoreMemCtrl
- Actel Fusion embedded flash memory (eNVM) accessed via CoreAhbNvm

Minimum Requirements

The minimum requirements for flash memory programming with SoftConsole are as follows:

- SoftConsole v3.1 or higher
- A Cortex-M1 target system with flash program memory at base address 0x00000000 (flash program memory connected to CoreAHBLite slot 0)
- A GNU C linker script that accurately describes the location, size, and type of the flash program memory in the Cortex-M1 target system. A number of example linker scripts are provided with SoftConsole.
- A flash device description file. This file describes the sequence of programming commands for a particular flash device. A number of flash device description files are provided with SoftConsole.

Flash Memory Programming Flow Overview

This section provides an overview of the Flash Memory Programming Flow. For a more detailed step-by-step description, see “Setting Up SoftConsole” on page 64.

Create a SoftConsole C Project

1. Create a SoftConsole C project
 - Project type: Executable
 - Toolchain: Cortex-M1
2. Add the following files to the project:
 - Any source files (C or assembler), or Actel IP core software drivers that may be required to build a Cortex-M1 application program
 - The Actel Cortex-M1/GNU Hardware Abstraction Layer, which the drivers rely upon to access the hardware and which also provides boot code for the Cortex-M1
 - Add include paths to the GNU C Compiler > Directories settings in the project properties for any HAL or Driver subfolders that contain source files

Add a Linker Script and Build the Project

Before an executable file can be loaded to any type of program memory, it must first be created by building the SoftConsole project. For most target systems, the GNU C Linker build settings must be modified to specify a linker script that matches the memory map of the system.

For a system with flash program memory, the project must be set up as follows:

1. Add a suitable linker script to the project.

2. Modify the MEMORY command section of the linker script to match the target system.
3. Specify the details of target system's memory map, including the location and size of the flash memory.
4. Specify the flash device description file. This is the key enabler for flash memory programming with SoftConsole.
5. Use the -T linker option in the project properties to specify the linker script. Save the project properties.
6. Build the project. This creates the executable file. Building the project also creates a memory-map.xml file, which is used to pass the structure of the memory map and the flash device to the GDB debugger and the Cortex-M1 sprite.

A number of example linker scripts are provided with SoftConsole. Each linker script is compatible with the Cortex-M1/GNU HAL, and a typical Cortex-M1 target system. Each linker script specifies a flash device description file that is compatible with the target system's program memory.

The flash device description file name is specified with a commented USE: directive in the MEMORY section of the linker script:

```
/* SOFTCONSOLE FLASH USE: .... */
```

The GNU C Linker ignores this directive, but SoftConsole recognizes it and uses the named flash device description file to provide the sequence of commands for flash programming.

A set of flash device description files is provided with SoftConsole. The flash device description files include a description file for the Actel Fusion eNVM (CoreAhbNvm), and a description file for the Intel 28F640 (J3 v.D) flash device that is used on many Actel development kit boards. The flash device description files do not need to be imported into the project; the appropriate file name should simply be specified in the linker script.

Create a Debug Launch Configuration for the Project

SoftConsole uses the GNU GDB debugger, in conjunction with the Cortex-M1 Sprite, to support loading and debugging of programs for the Cortex-M1 processor. The executable file must be loaded into program memory before debugging can take place.

For a Cortex-M1 system with SRAM program memory located at base address 0x00000000, this is simply a matter of using the GDB load command. SoftConsole automatically inserts the load command (as one of the Initialize commands) when the debug launch configuration for Cortex-M1 is first created, so that when the debug launch configuration is subsequently run, it loads the executable file into SRAM at the start of the debug session.

For a Cortex-M1 system with flash program memory located at base address 0x00000000, loading the executable file to program memory is typically not as straightforward as for SRAM program memory, because of the requirement to send mode setup commands to the flash device. However, SoftConsole seeks to minimize this complexity by enabling the GDB load command to load the executable file directly to flash program memory, for an appropriately configured Cortex-M1 project.

To create a debug launch configuration for the project:

When the debug launch configuration is created, the Initialize commands box on the Commands tab is automatically populated with the following:

```
target remote | "${eclipse_home}/../Sourcery-G++/bin/arm-none-eabi-sprite" flashpro:
"${build_loc}"
load
tb main
```

Figure A-1 shows the debug launch configuration.

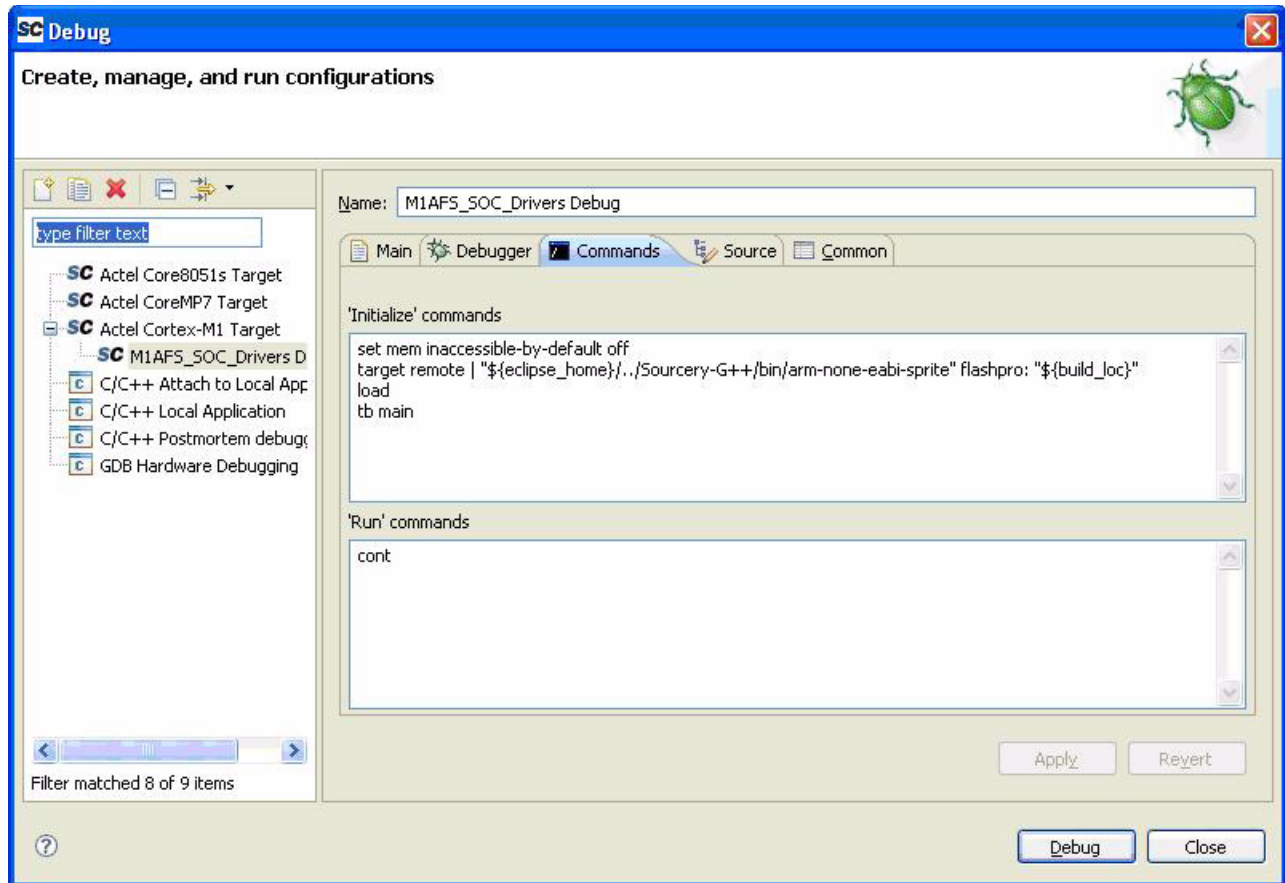


Figure A-1 · Programming

The GDB target remote command launches the Cortex-M1 Sprite, through which GDB communicates with the target system. The Cortex-M1 Sprite finds the memory-map.xml file in the current build folder (`${build_loc}`) and points it at the Debug or Release folder, whichever is the current active build configuration. It uses this memory map description to inform GDB that the target system is using flash program memory, so the specified flash device description file should be used when programming the flash memory, and hardware breakpoints should be used when debugging.

The GDB load command directs the Cortex-M1 Sprite to load the executable file into flash program memory, using the programming commands from the flash device description file.

Load the Executable File to Flash Program Memory

To load the Executable file to flash program memory, launch a debug session. Messages in the Console view will indicate that flash programming is in progress (Figure A-2).



Figure A-2 · Flash Programming in Progress Console Message

Setting Up SoftConsole

This section describes the steps required to set up a SoftConsole project, so that the executable file, which is the output of a Project Build, is automatically loaded into flash program memory when a debug session is launched.

Note: The description is based on using the Cortex-M1/GNU HAL and the set of example linker scripts supplied with SoftConsole. However, any linker script and corresponding boot code may be used, as long as the linker script includes the commented USE: directive to declare the flash device to be programmed.

Referenced Files

The description in this section contains several references to files provided with SoftConsole. The default install folder for SoftConsole is typically *C:\Program Files\Actel\SoftConsole vX.Y*, where vX.Y refers to the version of SoftConsole (for example, SoftConsole v3.1). The paths to the referenced files are listed here for convenience.

The example linker scripts are located in the following folder:

<SoftConsole install folder>\src\Cortex-M1\linker-script-examples

The flash device description files are located in the following folder:

<SoftConsole install folder>\Sourcery-G++\share\sprite\flash

The Cortex-M1/GNU HAL is located in the following folder:

<SoftConsole install folder>\src\Cortex-M1\Cortex-M1_hal

Create a SoftConsole Project

1. Open SoftConsole.
2. Select **File > New > C Project** from the SoftConsole menu to open the Create C project dialog.
3. Enter the project details on the C project dialog.
 - Project name: Type a name for the project.
 - Project types: Choose Executable (Managed Make)
 - Toolchain: Choose Actel Cortex-M1 Tools
4. Click **Next**, and **Finish**.
5. Turn off Automatic Building.

On the Project menu, ensure that the Build Automatically check box is cleared.
6. Set the Active Build Configuration to **Debug**.

On the Project menu, select **Build Configurations > Set Active > Debug**.

7. Add source files to the project.
 - Select **File > New > Source file** to create new C files.
 - Select **File > Import** to import existing source files.

Note: Alternatively, import an existing project, such as one of the example projects supplied with the Actel IP core software drivers. These example projects already contain the Cortex-M1/GNU HAL, but the linker scripts included with the HAL would have to be modified to enable flash memory programming, or replaced with one of the example linker scripts provided with SoftConsole.

If the Active Build Configuration is set to Release, the executable will be built with high optimization and without debug symbols. The executable may be loaded to flash program memory, but debugging will only be possible at the assembler level.

Add a Linker Script to the Project

A number of example linker scripts are provided under the SoftConsole install folder. Each example linker script has a detailed comment section at the top, which explains its purpose and how to use it in a SoftConsole project to support flash memory programming and debugging.

The example linker scripts are located in the following folder:

<SoftConsole install folder>\src\Cortex-M1\linker-script-examples

The main example linker scripts provided are described below.

run-from-intel-flash.ld

This linker script supports a target system with 2x16-bit Intel 28F640 Flash devices, accessed via CoreMemCtrl, at base address 0x00000000. The two flash devices are connected in parallel for a 32-bit data bus width. CoreMemCtrl is configured for 32-bit flash data bus width.

run-from-actel-coreahbnvm.ld

This linker script supports a target system with Actel Fusion FPGA embedded flash memory (eNVM), accessed via CoreAhbNvm, at base address 0x00000000.

only-ram-memory.ld

This linker script supports a target system with SRAM at base address 0x00000000. The SRAM can be Actel FPGA embedded SRAM accessed via CoreAhbSram or external SRAM memory (external to the FPGA) accessed via CoreMemCtrl.

Two additional example linker scripts are provided for use when program code is to be copied from flash to SRAM at boot time, and then run from SRAM.

boot-from-intel-flash.ld

This linker script supports a target system with 2x16-bit Intel 28F640 flash devices, accessed via CoreMemCtrl, at base address 0x00000000. The two flash devices are connected in parallel for a 32-bit data bus width. CoreMemCtrl is configured for 32-bit flash data bus width.

boot-from-actel-coreahbnvm.ld

This linker script supports a target system with Actel Fusion FPGA embedded flash memory (eNVM), accessed via CoreAhbNvm, at base address 0x00000000.

Choose the example linker script that most closely matches your target system memory map. For the most part, the selection of the appropriate linker script is all that will be required and no editing of the linker script will be necessary.

Import the Linker Script

1. Select the project in Project Explorer.
2. Select **File > Import** to open the Import dialog.
3. Expand General, select **File System** and then click the **Next** button.
4. Click the **Browse** button, navigate to the linker-script-examples folder, and click **OK**.
Folder location: <SoftConsole install folder>\src\Cortex-M1\linker-script-examples
5. Check the box beside the linker script (*.ld) of interest in the right-hand pane and select the **Create selected folders only** radio button.
Alternatively check the box beside the linker-script-examples folder in the left-hand pane to import all of the example linker scripts, and select the **Create selected folders only** radio button.
6. Click **Finish**. The selected linker script will appear in the Project Explorer view (Figure A-3).

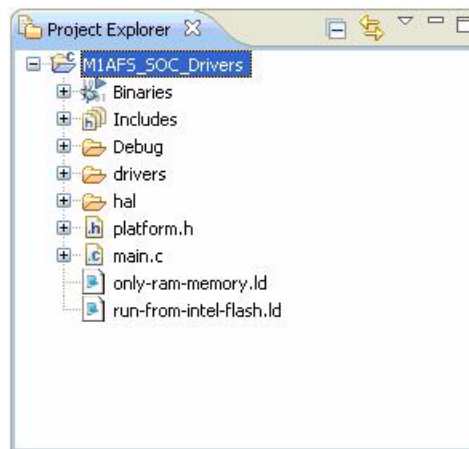


Figure A-3 · Linker Script in Project Explorer View

Set Up the Memory Map in the Linker Script

The MEMORY command section of the linker script is crucial for successful programming and debugging from flash memory. When using the example linker scripts provided with SoftConsole, there is, for the most part, no need to modify the content of the linker script, if the memory map and flash device described by the MEMORY command section matches the target system hardware.

However, if adjustment of the memory map is required, then the settings in the section between the /*Start of board customization*/ and /*End of board customization*/ comments are likely to be the only edits necessary.

An example of the MEMORY command section from the run-from-intel-flash is shown here.

```

/*****
 * Start of board customization.
 *****/
MEMORY
{
    /*
     * WARNING: The words "SOFTCONSOLE", "FLASH", and "USE", the colon
     *          ":", and the name of the type of flash memory are all in
     *          a specific order. Please do not modify that comment line,
     *          in order to ensure debugging of your application will use
     *          the flash memory correctly.
     */
    /* SOFTCONSOLE FLASH USE: intel-28f640-2x16 */
    rom (rx) : ORIGIN = 0x00000000, LENGTH = 1M

    /* Normal SRAM */
    ram (rwx) : ORIGIN = 0x10000000, LENGTH = 1M
}
RAM_START_ADDRESS = 0x10000000; /* Must be the same value as MEMORY region ram ORIGIN
above. */
RAM_SIZE = 1M; /* Must be the same value as MEMORY region ram LENGTH above. */
MAIN_STACK_SIZE = 256k; /* Cortex main stack size. */
PROCESS_STACK_SIZE = 16k; /* Cortex process stack size (only available with OS
extensions). */

/***** * End of board
customization.
 *****/

```

The following points should be noted if customization of the linker scripts is required:

The specifications of “rom” and “ram” in the MEMORY command section of the linker script must match the target system memory map.

If the ORIGIN or LENGTH of the “ram” specification are changed, then the RAM_START_ADDRESS and RAM_SIZE must be changed to match them.

The string “rom” must be used to specify the flash memory region in the MEMORY command section.

The commented directive /* SOFTCONSOLE FLASH USE:.... */ must appear on the line immediately preceding the “rom” specification, with the appropriate flash device named.

The string used to name the flash device must be the name of the flash device description file for that flash device (without the *.xml extension).

The flash device description file is an XML file that describes the programming sequence and commands for a particular flash device and the physical connectivity of the flash device (examples: 1x32-bit device, 1x16-bit device, 2x16-bit device).

A number of flash device description files are provided with SoftConsole, for the flash devices that are supported for flash memory programming. They are located in the following folder:

<SoftConsole install folder>\Sourcery-G++\share\sprite\flash

- actel-coreahbnvm.xml is for Actel Fusion FPGA embedded Flash memory (eNVM)
- intel-28f640-2x16.xml is for 2x16-bit Intel 28F640 Flash devices connected in parallel

DO NOT MOVE OR COPY the flash device description files from this folder into the project. It is only necessary to specify the file name correctly (without the *.xml extension) in the commented USE: directive in the linker script and SoftConsole will find the file in this folder.

Set up the Linker Script in the Project Build Settings

1. Select the project in Project Explorer.
2. From the File menu select **Properties** to open the project Properties dialog.
3. Select **Debug** from the Configuration drop-down menu.
4. Navigate to C/C++ Build > Settings.
5. On the Tool Settings tab, navigate to **GNU C Linker > Miscellaneous**.
6. In the Linker flags box, use the -T linker option to specify the linker script to be used (Figure A-4).

Linker flags

Figure A-4 · -T Linker Option

- For run-from-intel-flash.ld enter: -T../run-from-intel-flash.ld
 - For run-from-actel-coreahbnvm.ld enter: -T../run-from-actel-coreahbnvm.ld
 - For only-ram-memory.ld enter: -T../only-ram-memory.ld
7. Click **Apply** and **OK**.
- Note:** Do not specify the linker script as an -Xlinker option (in the Other options (-Xlinker [option]) box); this will cause the compiler to use your linker script incorrectly.

Add the Actel Cortex-M1\GNU HAL to the Project

A copy of the Actel HAL, for the Cortex-M1 processor and the GNU toolchain, is provided under the SoftConsole install folder. The HAL must be imported into the SoftConsole project and the project settings must be updated to specify the HAL folders as Include paths to the GNU C Compiler. The HAL provides Cortex-M1 boot code, which is required by the example linker scripts, and is used by the GNU C Linker when building the executable file.

Import the HAL

1. Select the project in Project Explorer.
2. Select **Import** from the File menu to open the Import dialog.
3. Expand General, select **File System** and then click the **Next** button.
4. Click the **Browse** button, navigate to the *Cortex-M1_hal* folder and click **OK**.
Folder location: <SoftConsole install folder>\src\Cortex-M1\Cortex-M1_hal
5. Select the **Cortex-M1_hal** check box and select the **Create selected folders only** radio button.
6. If the Into folder: box is empty, click the **Browse...** button beside it. Select the project and click **OK**.

7. Click **Finish**. The HAL folders and source files (hal, hal\CortexM1 and hal\CortexM1\GNU) will appear in the SoftConsole Project Explorer view (Figure A-5).

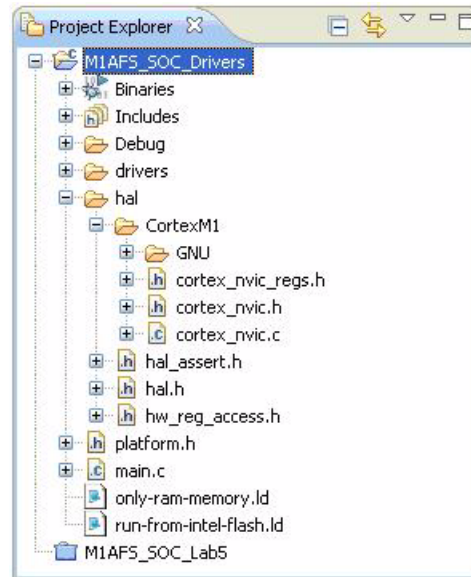


Figure A-5 · HAL Folders in Project Explorer View

Set up the HAL in the Project Build Settings

Specify the HAL folders as Include paths for the GNU C Compiler, to include the HAL source files in the Compilation.

1. Select the project in Project Explorer.
2. From the File menu, select **Properties** to open the project Properties dialog.
3. Select **Debug** from the Configuration drop-down menu.
4. Navigate to C/C++ Build > Settings.
5. On the Tool Settings tab, navigate to GNU C Compiler > Directories.
6. Add the three HAL folders—hal, hal\CortexM1 and hal\CortexM1\GNU—to the Include paths (-I) box:
 - Click the Add (+) icon, and an Add directory path dialog will appear.
 - Click the **Workspace** button.
 - Click the + to expand the project. Select the hal folder. Click **OK**.
 - In the Add directory path dialog, click **OK**.
 - Repeat these steps to add the hal\CortexM1 and hal\CortexM1\GNU folders.
7. The Include paths (-I) box should now contain the following:


```

${workspace_loc:/<project name>/hal}
${workspace_loc:/<project name>/hal/CortexM1}
${workspace_loc:/<project name>/hal/CortexM1/GNU}
      
```

These are shown in [Figure A-6](#).

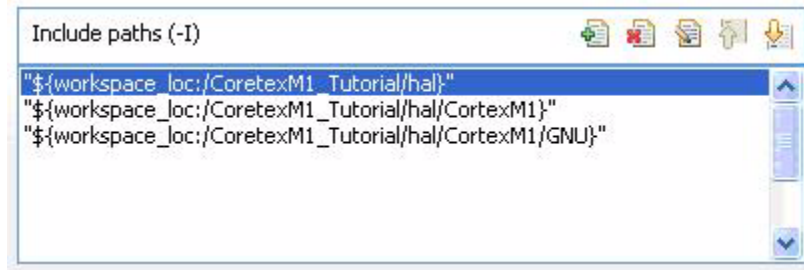


Figure A-6 · Include Paths

8. Click **Apply** and **OK** to close the Properties dialog box.

Build the Project

1. Select the project in Project Explorer.
2. From the Project menu select **Build Project**.
3. The project should build, without reporting any errors in the Console view.

A Debug folder should appear under the project root folder ([Figure A-7](#)).

The Debug folder contains all the files generated by the build, including the executable file and a memory-map.xml file. This memory-map.xml file is used by SoftConsole to direct the debugger (GDB and the Cortex-M1 Sprite) to program the executable to the flash program memory, when the Debug session is launched. The memory-map.xml file also directs GDB to use hardware breakpoints for debugging.

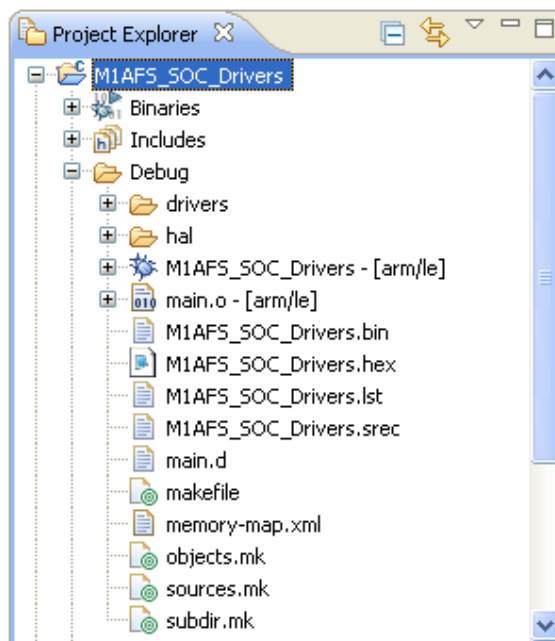


Figure A-7 · Debug Folder

Note: The project must be built so that the executable file and the memory-map.xml file are present in the current build folder (Debug or Release folder, whichever is the current active build configuration), before a debug launch configuration is set up for the project, or a configured debug session is launched.

Set Up the Debug Launch Configuration for the Project

1. Select the project in Project Explorer.
2. From the Run menu, select **Open Debug Dialog**.
3. Select the **Actel Cortex-M1 Target** and create a new debug launch configuration.
4. A debug configuration named <project name> Debug is created with all fields correctly populated.
The C/C++ Application box on the Main tab should contain the executable file (Figure A-8).

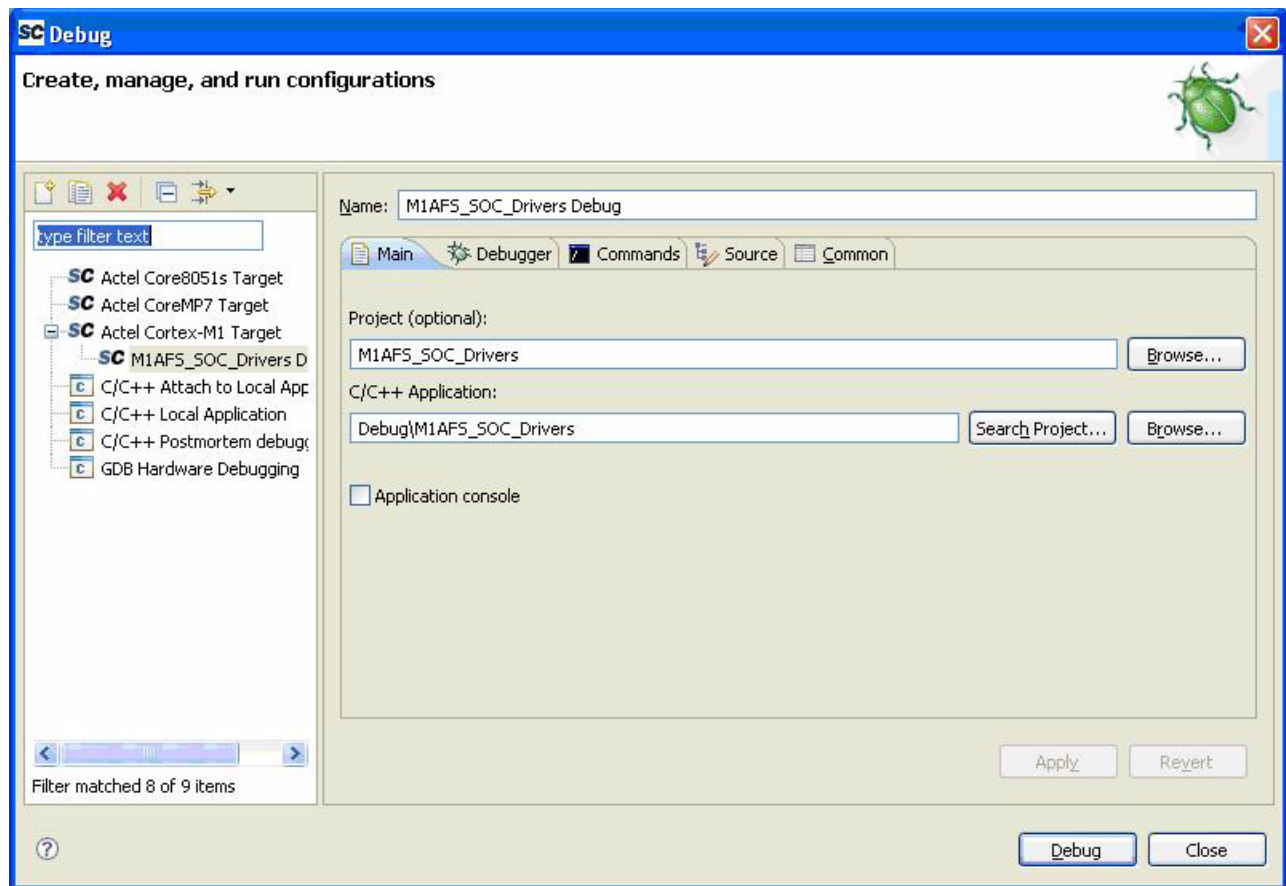


Figure A-8 · Executable in C/C++ Application Box

1. If the C/C++ Application box is empty, click the **Search Project...** button and select the executable file.
2. Click **Apply** if any changes were made.
3. Click **Close**.

When the debug launch configuration is created, the Initialize commands box on the Commands tab is automatically populated with the following commands:

```
target remote | "${eclipse_home}/../Sourcery-G++/bin/arm-none-eabi-sprite" flashpro:
"${build_loc}"
load
tb main
```

The GDB target remote command launches the Cortex-M1 Sprite, via which GDB communicates with the target system. The Cortex-M1 Sprite finds the memory-map.xml file in the current build folder (\${build_loc}) and points it at the Debug or Release folder, whichever is the current active build configuration. It uses this memory map description to inform GDB that the target system is using flash program memory, so the specified flash device description file should be used when programming the flash memory, and hardware breakpoints should be used when debugging.

The GDB load command directs the Cortex-M1 Sprite to load the executable file into flash program memory, using the programming commands from the flash device description file (Figure A-9).

These commands are executed by GDB when the debug session is launched.

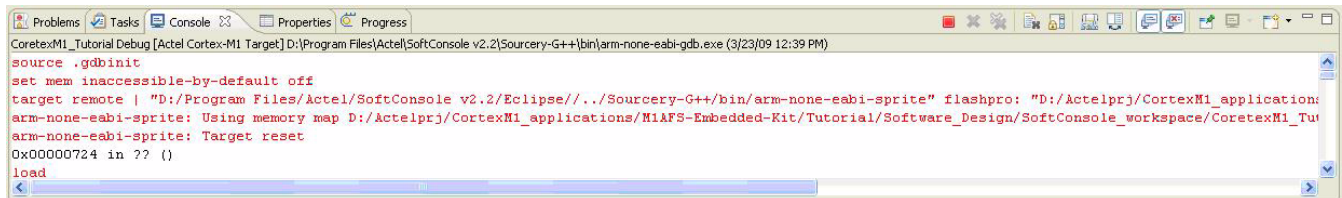


Figure A-9 · GDB Commands

Programming the Flash Memory

When the SoftConsole project is set up as described in the previous section, all that is required to program the flash memory is to launch a Debug session. The load command is one of the Initialize commands for the debug launch configuration. The debugger executes the load command as it starts up and begins the programming of the project's executable file to the flash memory. Prior to launching the debug session confirm the following:

- Power is applied to the target board.
- The FlashPro4 programmer is connected to the target board.

Start a Debug Session to Program the Flash Memory

1. Select the project in Project Explorer. From the Run menu, select **Open Debug Dialog**.
2. Select the debug launch configuration that was previously created under Actel Cortex-M1 Target.
3. Click the **Debug** button.

The debug session is launched and the messages in the Console view should indicate that flash programming is in progress, and eventually that it is completed (Figure A-10).

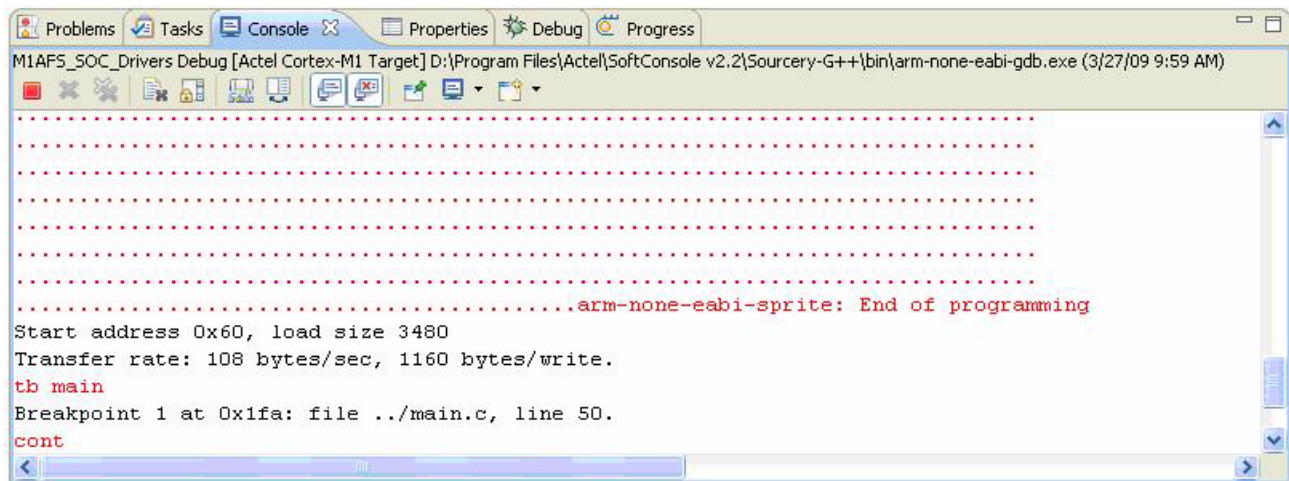


Figure A-10 · Console View Messages During Flash Programming

When flash programming is complete, the debugger processes any remaining Initialize commands for the debug launch configuration and on completion, SoftConsole switches to the Debug perspective.

4. The Confirm Perspective Switch dialog may ask if you wish to switch to the Debug perspective. Click **Yes**.
5. The Debug perspective opens and the program (running from flash) suspends at the temporary main() breakpoint.

Run and Debug the Program in Flash Memory

Loading of the executable to the flash program memory is now complete and the program is ready to continue running from the temporary main() breakpoint. All the usual debug operations are available, including stepping, setting and running to breakpoints, and viewing register and memory content.

Note: Cortex-M1 supports the use of only two hardware breakpoints at any one time. So, when debugging a program that is running from flash, you may set many breakpoints but you **MUST ONLY ENABLE TWO BREAKPOINTS AT ANY ONE TIME** (that is, check only two breakpoints in the Breakpoints view). Remember also that GDB Step commands use a breakpoint, which counts towards the limit of two hardware breakpoints.

Note: Flash memory blocks are erased and rewritten only if there is any change to the contents. If no changes are made to the executable between debug sessions, SoftConsole does not rewrite the flash memory. This reduces the level of wear that the flash memory is subjected to over multiple debug sessions.

Flash Programming Checklist

1. Are you using SoftConsole v3.1 or higher?
2. Is the project a Cortex-M1, Executable (Managed Make), C project?
3. Did you set the Active Build Configuration to Debug?
4. Did you add your application program source files to the project?
5. Did you add a linker script to the project?
6. Does the linker script match your target hardware system?
7. Does the linker script specify the correct flash memory?
8. Did you specify the linker script with the -T linker option in the project Properties?
9. Did you add the Cortex-M1/GNU HAL (or another source of Cortex-M1 boot code) to the project?
10. Did you specify the include paths to the Cortex-M1/GNU HAL folders in the project Properties?
11. If you are using your own Cortex-M1 boot code (that is, not using the Cortex-M1/GNU HAL), ensure that any constants defining system memory origin and size are compatible with the MEMORY section of the linker script.
12. Did you build the project?
13. Did the project build proceed without reporting errors in the Console view?
14. Does the Debug folder contain the executable file (same name as the project, with no file extension)?
15. Does the Debug folder contain the memory-map.xml file?
16. Does the memory map description in the memory-map.xml file match the MEMORY section of the linker script?
17. Does the debug launch configuration point to the correct executable in the C/C++ Application box (Debug\<project name>)?
18. Does the Console view report that flash programming has completed without reporting errors?
19. Does the Console view report that GDB is automatically using hardware breakpoints?
20. Has SoftConsole switched to the Debug view, with the program suspended at the main() breakpoint?
21. Are there at most two breakpoints enabled in the Breakpoints view?

Loading of the executable to the flash program memory should now be complete and all the usual debug operations should be available, including stepping, setting and running to breakpoints, and viewing register and memory content.

Appendix B – Configuring the Debug Utility for CoreMP7 Projects

The hardware target debug utility used for CoreMP7 requires manual configuration.

1. Select **Run > Open Debug Dialog** and select Actel CoreMP7 as a target.
1. Select **Run> External Tools > Open External Tools Dialog** box.
2. Click **New Configuration**.
3. In the External Tools, Create, manage, and run configurations box, change the Name (in the Name: rectangle) from New_configuration to **FlashPro4 debugger for MP7**. The exact name does not matter, but it is useful to choose a name that indicates the CPU.
4. For the location rectangle, indicate the location of the MP7 sprite. This should be *<softconsole installation path> \Arm\Bin\cliarm.exe*.
5. For the working directory rectangle, indicate the directory of the MP7 sprite. Note that this is the same as the Location without the filename specified (that is, leave off the \cliarm.exe).
6. In the Arguments box type **initarm.tcl**.
7. Click **Apply**.
8. Click **Close**.

Figure B-1 shows an example of setting the Debug configuration.

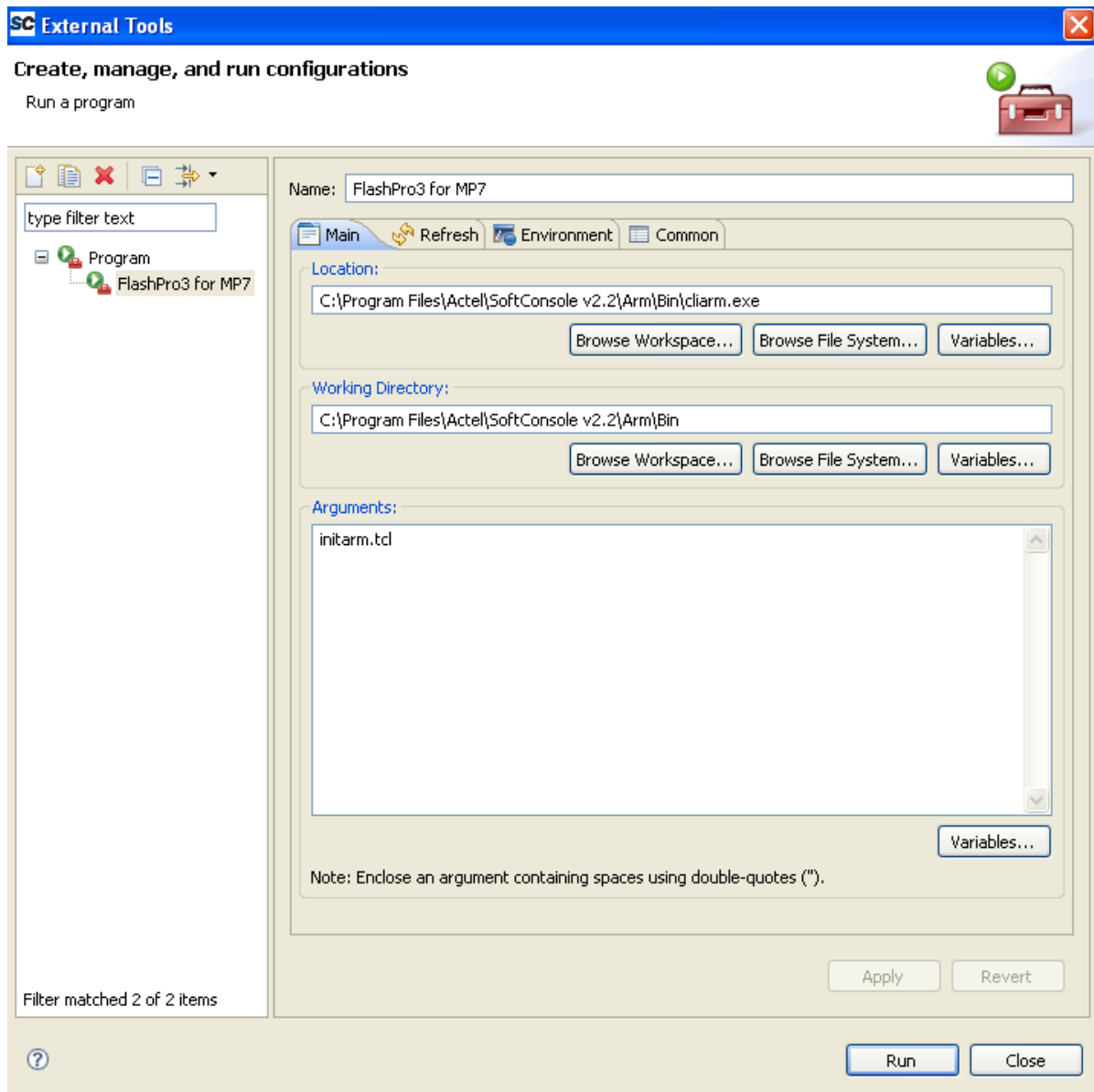


Figure B-1 · Setting the Debug Configuration for CoreMP7

9. Select **Run > External Tools > FlashPro4 for MP7** (or whatever you named your MP7 debug configuration), as shown in [Figure B-2](#).

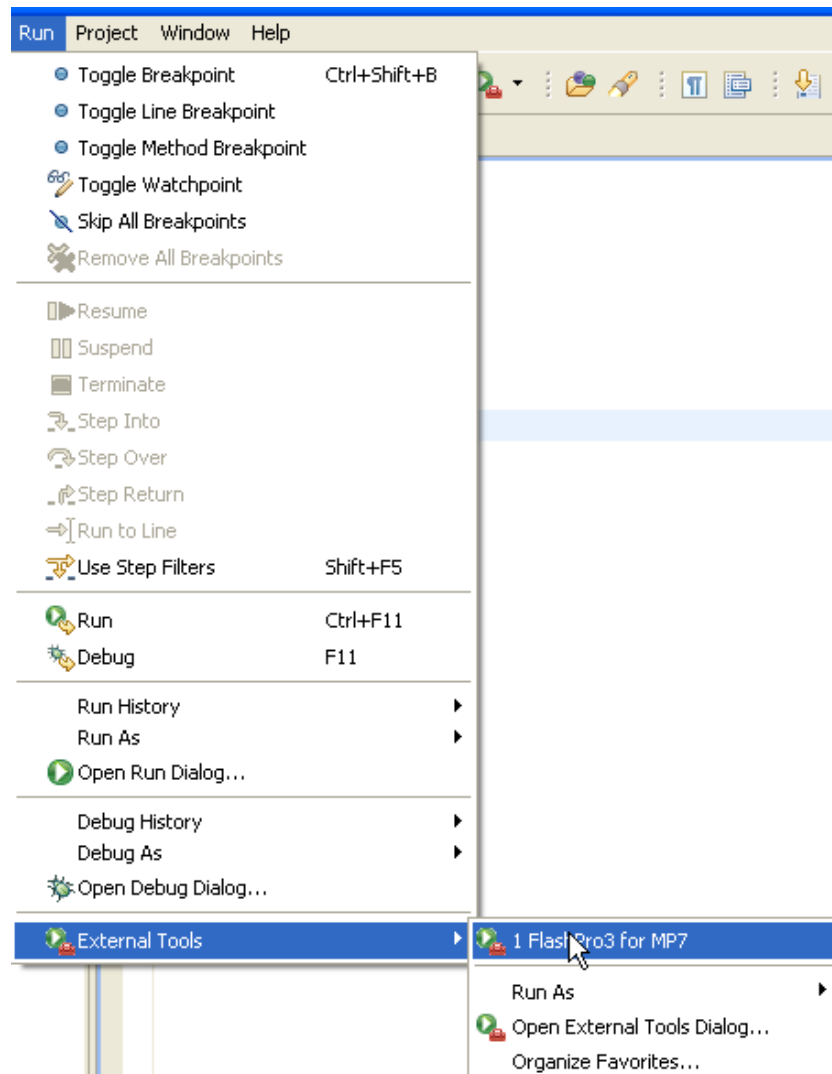


Figure B-2 · Launching the Debugger for CoreMP7

10. On the C/C++ application, click **Search project**.

11. Click the name of your project in the Binaries box (Figure B-3).

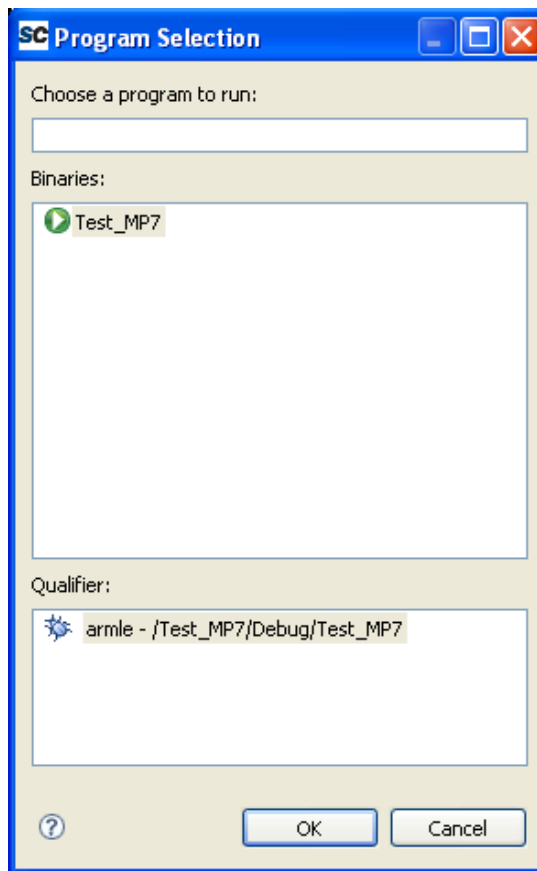


Figure B-3 · Program Selection

12. Click OK.
13. Click the **Debug** button.
14. Answer **Yes** to the Confirm Perspective Switch box.

Appendix C – Reference Documents

SoftConsole Quick Start Guide

www.actel.com/documents/SoftConsole_QS_UG.pdf

Libero User's Guide

www.actel.com/documents/libero_ug.pdf

Cortex-M1 Handbook

www.actel.com/documents/CortexM1_HB.pdf

CoreMP7 Users Guide

www.actel.com/documents/CoreMP7_UG.pdf

Core8051s Handbook

www.actel.com/ipdocs/Core8051s_HB.pdf

SoftConsole Documentation

The following documentation is available on your PC after SoftConsole is installed.

Access the documents using **Start > Programs > SoftConsole 3.1 > Reference Documents**.

- *SDCC User's Manual*
- *Cortex-M1 Bare Metal Boot Code Guide*
- *GNU Binutils manual*
- *GNU GCC Compiler Manual*
- *Programming Flash Memory Guide*
- *SDCC User Manual*

Additional documents:

- *GNU Linker Manual (ld.pdf)* - contains a description of Linker scripts (<SoftConsole v3.1 install>\Sourcery-G++\share\doc\arm-none-eabi\pdf)
- *GNU binary utilities (binutils.pdf)* - Page 28 lists options for objdump (<SoftConsole v3.1 install>\Sourcery-G++\share\doc\arm-none-eabi\pdf)
- *GNU Compiler Manual (gcc.pdf)* (<SoftConsole v3.1 install>\Sourcery-G++\share\doc\arm-none-eabi\pdf\gcc)
- List of GDB commands (<SoftConsole install folder>\Sourcery-G++\share\doc\arm-2007q1-21-arm-none-eabi\pdf\gdb.pdf)

Product Support

Actel backs its products with various support services including Customer Service, a Customer Technical Support Center, a web site, an FTP site, electronic mail, and worldwide sales offices. This appendix contains information about contacting Actel and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From Northeast and North Central U.S.A., call **650.318.4480**

From Southeast and Southwest U.S.A., call **650.318.4480**

From South Central U.S.A., call **650.318.4434**

From Northwest U.S.A., call **650.318.4434**

From Canada, call **650.318.4480**

From Europe, call **650.318.4252** or **+44 (0) 1276 401 500**

From Japan, call **650.318.4743**

From the rest of the world, call **650.318.4743**

Fax, from anywhere in the world **650.318.8044**

Actel Customer Technical Support Center

Actel staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions. The Customer Technical Support Center spends a great deal of time creating application notes and answers to FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Actel Technical Support

Visit the [Actel Customer Support website \(www.actel.com/support/search/default.aspx\)](http://www.actel.com/support/search/default.aspx) for more information and support. Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on the Actel web site.

Website

You can browse a variety of technical and non-technical information on Actel's [home page](http://www.actel.com), at www.actel.com.

Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. Several ways of contacting the Center follow:

Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.

The technical support email address is tech@actel.com.

Phone

Our Technical Support Center answers all calls. The center retrieves information, such as your name, company name, phone number and your question, and then issues a case number. The Center then forwards the information to a queue where the first available application engineer receives the data and returns your call. The phone hours are from 7:00 A.M. to 6:00 P.M., Pacific Time, Monday through Friday. The Technical Support numbers are:

650.318.4460
800.262.1060

Customers needing assistance outside the US time zones can either contact technical support via email (tech@actel.com) or contact a local sales office. [Sales office listings](#) can be found at www.actel.com/company/contact/default.aspx.

Index

A

Actel
 electronic mail 81
 telephone 82
 web-based technical support 81
 website 81
assembly code 57
 stepping 58

B

Breakpoints view 51
build methods 44

C

C project 59
compiler
 configure to match Cortex-M1 application 30
 GCC 6
 SDCC 6
contacting Actel
 customer service 81
 electronic mail 81
 telephone 82
 web-based technical support 81
Core8051s 9
CoreMP7 9
 debug utility 75
Cortex-M1 8
 programming 59
CPUs supported 8
customer service 81

D

debug
 CoreMP7 projects 75
 create debug launch configuration 60
 create debug version 43
 sprites 6
 start session 73
 version 43
 view 51
debug utility 51
debugger 6
debugging 47

E

Eclipse IDE 5
editors 11
exception handlers 35

F

features, key 5
flash
 flash memory programming flow 61
 programming 59
flash programming checklist 74
FlashPro3 7

G

GCC compiler 6
GDB debugger 6

H

HAL 30, 68
 import 68
 source code 30
header file
 create 27
 obtaining information 41

I

import project 21
installation 13
interrupt numbers 34

L

libraries 40
licensing 15
linker 32
linker scripts
 add to project 65
 example 31
 examples 65
 import 66
 modification needed 31
 program code and data memories 30
 set up memory map 67

M

math function 41
MEMORY command section example 67

- memory limits, setting 36
- memory map 34
 - set up 67
- memory model size 37
- memory size 32
- modifying code 58
- Modules view 52

N

- new project 17

O

- output files 43

P

- pack iram data 36
- perspectives 11
- Problems view 55
- product support 81–82
 - customer service 81
 - electronic mail 81
 - technical support 81
 - telephone 82
 - website 81
- programming
 - flash 59
- project 11
 - build 70
- project build
 - set up linker script 68
- projects
 - build methods 44
 - C 59
- pseudo stack 36

R

- references 79
- Registers view 52
- release version 43

S

- SDCC compiler 6
- single stepping 56
- SoftConsole
 - create embedded applications 17
 - documentation 79
 - environment 11
 - installation 13
 - licensing 15
 - referenced files 64
 - setting up 64
- SoftConsole package 5
- Source Code view 53
- source file
 - create 26
- sprite 63
- startup files 34, 39
- system requirements 13

T

- Task view 55
- technical support 81
- tool flows 7

V

- Variables view 51
- vector table 34
- version
 - release 44
- views 11

W

- web-based technical support 81
- workbench 11
- workspace 11
 - copy projects into 22
 - setting up 17
- workspace launcher 17



Actel is the leader in low-power FPGAs and mixed-signal FPGAs and offers the most comprehensive portfolio of system and power management solutions. Power Matters. Learn more at www.actel.com.

Actel Corporation • 2061 Stierlin Court • Mountain View, CA 94043 • USA

Phone 650.318.4200 • Fax 650.318.4600 • Customer Service: 650.318.1010 • Customer Applications Center: 800.262.1060

Actel Europe Ltd. • River Court, Meadows Business Park • Station Approach, Blackwater • Camberley Surrey GU17 9AB • United Kingdom

Phone +44 (0) 1276 609 300 • Fax +44 (0) 1276 607 540

Actel Japan • EXOS Ebisu Building 4F • 1-24-14 Ebisu Shibuya-ku • Tokyo 150 • Japan

Phone +81.03.3445.7671 • Fax +81.03.3445.7668 • <http://jp.actel.com>

Actel Hong Kong • Room 2107, China Resources Building • 26 Harbour Road • Wanchai • Hong Kong

Phone +852 2185 6460 • Fax +852 2185 6488 • www.actel.com.cn