SMX® RTOS

Target Guide

Version 4.3
May 2015

by
David Moore

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

Revisions

date	ver	comments
5/04	3.6	first release; information taken from smx User's Guide, smx Porting Guide, and readme files
9/04	3.6	addition of CodeWarrior ARM and other information and corrections
12/04	3.6	corrections, ColdFire peripheral renumbering
5/05	3.7	update to v3.7 and addition of Diab ColdFire information
12/05	3.7	addition of IAR ARM information and other changes
5/06	3.7	update of ARM and ColdFire sections and addition of project file info for Borland and Microsoft
12/07	3.7	update of ARM, ColdFire, and common sections and addition of CrossWorks ARM
10/08	3.7	update of CrossWorks ARM and CodeWarrior ColdFire
4/09	3.7	addition of ARM-M (Cortex-M) and update of CrossWorks ARM
10/10	4.0	update to v4.0
10/12	4.1	update to v4.1 and new tool versions
1/14	4.2	update to v4.2 and removal of x86 information
5/15	4.3	update to v4.3 and new tool versions

Table of Contents

Common Notes

This section contains notes that are common to all processor versions of smx.

Introduction

This manual is a collection of target-related information, including tips about compilers and tools. There are different issues for each CPU and each tool suite, but the manual is organized as consistently as possible. Targets and tools are continually changing, so please consult the release notes in the DOC directory for additional and corrected information.

This manual is targeted to those using the smx multitasking kernel (rather than standalone releases of our middleware). However, some information about processor architecture and tools is useful to everyone.

Tip: Print just the Common section and the section for the CPU you are using.

Porting

Your release is most likely already ported to the hardware and tools you plan to use, in which case you can skip this section.

The smx multitasking kernel supports many CPUs and several compilers. However if you need to port to one that is not supported, the following is a summary of where to find the information you need:

New CPU architecture:
smx Porting Guide.

New CPU of an architecture already supported (e.g. ARM, ARM-M, CF):
Appropriate CPU section in **this manual** (e.g. ARM/ Porting to a New ARM or Board).

New compiler:
smx Porting Guide. Primarily what is relevant is information about the porting macros. These macros are quite compiler-dependent. Some compilers do not allow certain operations to be done in inline assembly, such as manipulating the stack pointer, so for them, such macros must be written as assembly routines in a separate file. Compilers can also differ in whether they create a stack frame pointer in function prologs. Because of differences such as

these, it is necessary to create a section for your compiler in the smx CPU header file (e.g. xarm.h, xarmm.h, xcf.h, or xppc.h). Copy the section that you think is closest and edit it for your compiler. When you try to compile the scheduler, the compiler will complain about any remaining problems in these macros.

The smxBase User's Guide covers porting SMX modules (middleware products) to different CPUs and tools.

BSP

BSP Notes

For ARM and ColdFire targets, we created PDF files in the DOC directory that summarize important information about the boards we support. These show memory layout, peripherals supported, and other details and tips about the board. One of these is provided in the DOC directory for the BSP you ordered. We recommend you print it and keep it close for reference.

BSP Configuration

The main configuration for the BSP is in **bsp.h**, **bsp.inc**, and **bsp.c**. See the beginning of the BSP API section in this manual for more details.

Protosystem

Files

These files are stored in the APP directory. Note that BSP files are also built and linked into the Protosystem. See the BSP Files section for your processor architecture for descriptions of the key BSP files.

1. **app.c**

   Sample application file. Replace this with your main application file. The two hook routines are appl_init() and appl_exit(). You must implement these.

2. **main.c**

   Contains main(), which calls smx_Go(). smx_Go() initializes smx, creates several smx objects, and starts *idle*, which is the first task. idle runs ainit() as its main function to perform application initialization. At the end of ainit(), idle's main function is changed to smx_IdleMain(). **ainit() must not call SSRs that suspend. Also, interrupts should be masked during initialization.** Generally, the startup code should mask all interrupts. main() ensures they are masked before calling smx_Go(), in case there is some reason you had to enable some interrupts. ainit() restores this mask. See *smx Startup and Scheduler Operation* in the SMX Quick Start for more discussion of these points. This file also statically initializes the ***smx_cf*** structure, based on settings in acfg.h. This is used by the

smx library. (It allows us to ship a binary version of smx). aexit() is used to exit. It can be made to infinite loop or do whatever is appropriate for your system on exit.

3 **main.h**

This file provides function prototypes and declarations for the Protosystem.

4 **mem.c**

Statically allocates large memory blocks for SDAR and ADAR.

5 **acfg.h**

acfg.h configures smx. Set the number of tasks, priority levels, stack size, etc. here. These settings directly affect memory requirements so keep these values small, but large enough for some growth in requirements.

6 **heap.c**

Provides simple functions to translate compiler heap calls into smx heap calls, when linking other pre-compiled libraries that make compiler heap calls. (Note: XSMX\xapi.h has macros to do similar translation at *compile* time.)

7 **initmods.c**

This file contains initialization code needed for some SMX modules (products). It is divided into sections for each module. It has 2 top-level routines, smx_modules_init() and smx_modules_exit() which call each module initialization and exit routine in turn. These 2 routines are called from ainit().

8 **smxaware.c**

Initialization file for smxAware. smxaware_init() is called by ainit().

9 **XXX.YYY Subdirectory** (e.g. IAR.ARM, CW.CF, etc.)

Build directory. Makefile, project file, batch file, locator build script, etc are stored here. Also the stripped (condensed) version of the log file is put here (for makefile builds).

BSP directory

1 **bsp.c, bsp.h**

Implements the BSP API routines documented in the APIs section of this manual. There is typically one of these for each board, stored in the subdirectory named for the board. For ARM-M, there is just one file, BSP\ARM\bspm.c.

2 **startup code** (file names vary)

The startup code performs some register and memory initialization, then calls main() in main.c. See the Protosystem section in the CPU section for a list of startup files for your CPU.

Target Defines

The project files and makefiles pass several target-related defines to the compiler and assembler to control conditional compilation/assembly of the code. For project files, these are in preinclude files in the CFG directory or in the project itself, in the case where the IDE does not support preinclude files. It is common for IDEs to support them for C files but not assembly. These are where key SB_BRD (board) and SB_CPU (processor) symbols are defined. See the Preinclude Files subsection in the section for your tools in this manual, for more information.

Coding Notes

ISRs

The Architectural Notes section for each processor in this manual has a subsection about ISRs specific to that processor. Here are some general tips for writing ISRs.

Some processors such as ARM have a single interrupt flag to enable or disable interrupts. Others, such as ColdFire have multiple bits that indicate an interrupt priority level for which interrupts are enabled. For the first case, use sb_INT_DISABLE()/sb_INT_ENABLE() to disable/enable interrupts. For the second case, use sb_IntStateSaveDisable() and sb_IntStateRestore(), which save and restore the interrupt priority level, unless you want to enable all interrupts at the end of the critical section.

smx ISRs must increment the smx global *srnest* before interrupts are enabled. This requires use of ISR enter/exit macros and often assembly shells to do the ISR prolog/epilog instead of using the compiler's interrupt keyword or other method to write an interrupt function.

On entry to ISRs, interrupts are disabled or disabled for lower and same priority levels, depending upon the processor. In the second case, if higher priority ISRs must be prevented from nesting in a critical section, use sb_IntStateSaveDisable() and sb_IntStateRestore(). If this must be prevented from the first statement of the ISR, it may be necessary for you to modify the assembly shell to disable all interrupts in the first instruction. Again, see the information about writing ISRs for your processor, in this manual.

To allow nesting, you must enable interrupts. However, before doing this, you should do at least the minimum operations necessary to service the interrupt:

```
//…
sb_IRQClear(IRQ_NUM);
/* read/write any peripheral controller registers that need to be handled,
   or full body of ISR. */
sb_IRQEnd(IRQ_NUM);
sb_INT_ENABLE();
//…
```

sb_IRQClear() acknowledges it so the same interrupt does not continue to be generated. sb_IRQEnd() tells the interrupt controller that processing is done and it is ok to generate the interrupt again. If you want to enable it sooner, use sb_IRQMask() to mask it before sb_INT_ENABLE() and unmask it with sb_IRQUnmask() before exiting the ISR.

The sequence of calls to hook an interrupt is as follows.

```
sb_IRQVectSet(IRQ_NUM, MyISR);
sb_IRQConfig(IRQ_NUM);
sb_IRQUnmask(IRQ_NUM);
```

For more information about these, please see the API section of the smxBase User's Guide. The smxBase OS porting layer has a function for hooking interrupts, but we do not recommend using it, except in special cases. See the section Porting Layer/ Interrupt Service Routines in the smxBase User's Guide for information about this function.

Important: smx ISRs (those that use smx_ISR_ENTER() and smx_ISR_EXIT()) must not run during initialization, since smx structures such as the LSR queue have not been created or initialized. Do not enable such ISRs until after interrupts are unmasked in ainit(). C++ users, keep in mind that static initializers run during the startup code before main().

Inline Assembly in C

C compilers generally support some degree of inline assembly within C files, but the syntax and rules vary for each compiler. We use inline assembly in smx scheduler porting macros to save the overhead of a function call and return. However, limitations of the tools have often forced us to write them as assembly functions. Some compilers do not allow changing certain registers, such as the stack pointer, from inline assembly. Newer versions have become more restrictive about this.

Another problem is register usage in inline assembly. The question is whether the compiler assumes the register will be unchanged following the inline assembly section or if it is expected to be preserved. Often the compiler documentation does not discuss this, and experimentation may not prove that something will always be ok, and at all optimization levels. Taking the safe approach and saving/restoring registers requires two memory references for each register, which may be more costly than the function call/return. Compilers do not expect volatile registers to be preserved across a function call, so implementing a porting macro as an assembly function guarantees you can use those registers without needing to save/restore them.

As a result of the limitations, we have re-implemented many of the smx porting macros as assembly functions in an assembly file.

Misc Notes

Configuration

The CFG directory contains preinclude files that pass settings and defines to the compiler and assembler to specify the target CPU, board, etc. Some tools call these "prefix files" or "via files". These are documented in the sections for each CPU.

APP\acfg.h in the Protosystem is where to specify the maximum number of various smx objects to allocate, such as tasks, stack pool stacks, and queue control blocks. The settings here are used to initialize the global *smx_cf* structure, which is referenced by the kernel. It is initialized in main.c.

XSMX\xcfg.h has kernel configuration settings. Most reduce the size of the kernel by removing features. These should usually be left alone, unless memory is very tight. Also you must have kernel source code (not provided in evaluation kits) and rebuild the smx library if you change any settings in these.

Main SMX modules each have their own configuration file. Examples:

smxFS:	XFS\fcfg.h
smxNS:	XNS\include\nscfg.h
smxUSBH:	XUSBH\ucfg.h

As part of building your release, we configure these files for the drivers or add-on modules you purchased. Other tuning can be done to them as well. See the SMX Quick Start for more discussion of the SMX Modules and the files involved.

Project Files and Makefiles

We typically supply either project files or makefiles, but not both. We recommend that you use what we provide. If you prefer to create your own, be sure to use all the switches we do! If you are in doubt about the need for a switch or setting, please ask. Unfortunately, IDEs make it hard to see what we have set, since settings are scattered across many dialog tabs and comparing each to a default project is tedious. Consult the section in this manual for the compiler you are using for any notes about necessary settings. Also see if you can get the pure Protosystem (as shipped) to build and run using your build files.

Project files often do not handle product modularity well (i.e. the ability for us to release a custom configuration of SMX modules per your order), so the Protosystem project file is set for the products you ordered. If you order more in the future, it is necessary for you to add other modules. This consists of adding its library and adding one or more defines to be passed to the compiler and assembler. These are listed in the SMX Quick Start, in section Global Concepts/ Module Defines. Also see Global Concepts/ IDE vs. Makefiles for more about this.

Processor Selection in Project Files and Makefiles

The library project files and makefiles that are not named for a particular target are set to build for a generic processor of the architecture you are using, e.g. ARM, so that they will work on any or most processors of that architecture. You should change the setting to match the processor you are using, so the compiler can generate more efficient code.

Make Utility

For compilers that do not include a make utility, our makefiles use the Microsoft NMAKE syntax. In the past, we supported the GNU DMAKE utility, but we found it to be inadequate because of its minimal conditional handling and poor diagnostics. Whenever we made a mistake in our makefiles, we often had a lot of trouble finding the problem — much more so than with other make utilities we have used such as NMAKE.

We chose to standardize on the NMAKE format because:

1. It works well.
2. It is likely you have a copy of it. If you have any version of Microsoft C++ or MASM, you have it.
3. It is commonly supported by make utilities such as Opus Make (www.opussoftware.com), and possibly shareware utilities from the web.

See Common Notes/ Tools/ NMAKE for some tips about using it.

C Run-Time Library

The following are issues to consider when using functions in the C run-time library.

reentrancy

Consult your compiler documentation to determine which functions in the C library are reentrant and which are not.

Calls to functions that are not reentrant need to be protected by a semaphore. The *in_clib* semaphore is provided in the Protosystem to deal with this. Calls to non-reentrant functions (and those you *suspect* may be non-reentrant) should be protected by this semaphore. That is, test in_clib before the call and signal after it. Macros have been provided for convenience: smx_CLibEnter() and smx_CLibExit() (see XSMX\xapi.h).

If having only one *in_clib* semaphore causes a bottleneck, replace it with a semaphore per group of C library functions. Grouping is dictated by shared, non-reentrant subroutines or use of a common global variable — study the C library source code to determine this.

stack usage

Some C library functions use a lot of stack. The printf() family of functions, for example, allocates large buffers on the stack — 1500 bytes or more. Tasks that use such functions need larger stacks. The stack usage checking and padding added in smx v3.6 is a big help in catching potential stack overflows such as this. When possible, use simpler functions; in this case, use itoa() instead of sprintf(). Alternatively, you can create a custom version of such functions, starting from the source code provided with the compiler.

Minimizing RAM Usage

stacks

Task stacks probably account for the largest RAM usage in a multitasking system, so it is desirable to have as few as possible. The smx stack pool is helpful since it allows minimizing the number of stacks required by allowing tasks to share stacks. When a task completes its work (i.e. it stops), it releases its stack for other tasks to use. Stacks are needed only for the tasks active *simultaneously*.

Also, you want to minimize the size of stacks in the stack pool as much as possible. The stack size used in the Protosystem, as shipped, is fairly large. When you get your system working, you may want to try to tune that size down.

Tip: Tune stack size when your application is working. Then, verify that it still works after reducing the stack size.

Use bound stacks for unusually large or small stacks, as stack pool stacks are intended to be the right size for the typical task in your system. Bound stacks are allocated by simply specifying a stack size as the last parameter to smx_TaskCreate(). They are allocated from the heap. Bound tasks keep their stacks even when stopped. The memory is freed only when the task is deleted.

heap and dynamically allocated regions (DARs)

Heap and DAR sizes are controlled in acfg.h. SDAR holds smx objects such as control blocks. ADAR and the heap are primarily for application use. mem.c shows clearly what is in ADAR and SDAR. See the Heap and Memory chapters in the smx User's Guide for more information.

control blocks

Control blocks are small, to minimize memory usage. Most are 12 to 36 bytes in 32-bit versions. The TCB is larger, currently about 80 bytes. See the control block definitions in xtypes.h to see their sizes and what fields they contain. The settings in acfg.h dictate how many control blocks of each type are allocated. You should tune this for your application, but set them generously initially for development to avoid SMXE_OUT_OF_ and SMXE_INSUFF_ errors.

SB_DEBUGGER_IRQ

If your debugger uses a software debug monitor on the target (i.e. not JTAG, BDM, etc.), set SB_DEBUGGER_IRQ in bsp.h to match the IRQ used by the debug monitor or else the debugger's Stop button won't work. We use this setting to unmask the interrupt in ainit(). The Protosystem and debug monitor are independently built, so this setting can't be determined automatically by including a header file.

Profiling

A simple profiler is built into smx. It monitors:

(1) overhead (smx)
(2) work (application)
(3) idle

See the profiling sections in the Diagnostics chapter of the smx User's Guide for full discussion.

Command Line Environment in Windows

This section does not apply to those using an IDE to build libraries and the application.

To use command-line compilers, it is necessary to add the compiler's BIN directory to the path, and it is likely that other environment variables need to be defined so that the compiler's include files and libraries can be found. Typically, when you install a compiler, it adds the necessary path to the *path* environment variable and adds any other environment variables that it needs. This is fine if you have only one compiler installed, but it doesn't work if you have several, since the names of some tools are the same for different compilers (e.g. link.exe), so the first one

encountered on the path is the one that is used. This result can be that the linker or other tool from one compiler is used when compiling with another.

To solve this problem, you can easily create a separate environment for each compiler. This can be done by creating shortcuts on the desktop that each run a different batch file. Each batch file sets the environment variables for one compiler (and associated tools such as locator). Then it either runs the Windows command prompt or a shell such as FAR (a clone of Norton Commander). This section explains how to do this. This technique should work on all 32-bit versions of Windows, but we have only tested it on 98, NT4, 2000, and XP. We recommend using FAR (or a similar shell) — see section Utilities/ Shell in the SMX Quick Start.

If you are using only one compiler and it did not add its path or other environment variables to the global Windows environment settings, see Windows Environment Variables below for a summary of how to set them.

Batch File

Start with one of the sample batch files provided in SMX\MISC\WINDOWS. Copy it to a convenient location (e.g. C:\), rename it as appropriate, and make these modifications:

1. Modify the paths to point to your compiler's directories.
2. Enable the line for the Windows command prompt or shell utility you wish to use. We recommend you use FAR — see section Utilities/ Shell in the SMX Quick Start for more about this. (You can initially set it to the command prompt and later change it to use FAR.)
3. Add any additional environment variables (e.g. INCLUDE, etc) required by your compiler.
4. Delete any environment variables or other lines that are not needed.

The batch file is very simple: It adds the compiler and assembler paths to the global path variable and then runs the Windows command prompt or your shell utility. Note that the path and other environment variables are only for this command-line session and do not affect the path and variables in others, so you could have several or all of these environments open simultaneously for the compilers you are using and you can then Alt-Tab between them.

A key point is to add the compiler's path to the **beginning** of the path statement since that will be seen before any other paths that were added to by other compiler installs. You could delete those from the global path in Windows, but it is unnecessary if the batch file adds this compiler's path first.

Desktop Shortcut

1. Right-click on the desktop. Select New, Shortcut.
2. A wizard pops up asking:
    a. Command line: Enter path to batch file (e.g. c:\sys\mc32.bat)
    b. Select a name for the shortcut: Enter name that will appear below the icon (e.g. MC32). (Don't worry, it can be renamed later.)

    Press Finish.

3. Right click on the shortcut and select Properties.
   a. Options tab. Set for Full screen.
   b. Layout tab: set the Height to 25 or 50. (Defaults to 300 in 2000/XP.) Press Ok.
4. Run the shortcut and test it:
   a. If you see “Out of environment space” in 95/98/ME, increase the Initial environment setting in the shortcut properties. It’s on the Memory tab. Start at 512 (bytes) and increase until the problem goes away.
   b. Type “path” at the command prompt to check the path to verify that only the desired compiler, assembler paths are present. Often when you install a compiler, it adds itself to the global Windows path, so if you've installed more than 1 compiler, you could have both in your path. This might not be a problem for 2 different compilers because the program names are different, so both could be in the path. However, if you have older versions of the same compiler installed, there may be problems if the path to the old version precedes the path to the new one. To delete extra paths from the Windows path environment variable, see “Windows Environment Variables” below.
   c. Type the name of the compiler exe and the assembler exe, to ensure they are found. You should see a sign-on banner and maybe a list of switches. If not, check the path carefully. Also ensure paths are separated by a semicolon.
   d. Try to build one of the SMX libraries or the Protosystem.
5. Initial Directory: You may wish to set the shortcut to automatically go to your SMX directory. This can be done from the batch file or the shortcut properties:
   a. Batch File:

      ```
      c:
      cd\smx
      ```

      Put this **before** the line that runs the shell (cmd.exe, command.com, or far.exe, etc.) since that must be last.
   b. Shortcut: Right click on the shortcut. Select Properties. On the Shortcut tab, set the Initial directory line to the desired directory. Be sure the batch file is not also changing the directory, or this setting will seem to have no effect (since the batch file runs after the shortcut changes directory.
6. Change the icon (optional):
   a. Right click on the shortcut and select Properties. On the Shortcut tab, press the Change Icon button. It will tell you the batch file has no icons. Press Ok.
   b. Then it pops up a dialog showing some common icons. You can either choose one of these, or better, get an icon from the one of the compiler’s exe files. To do that, press the Browse button, and change to the compiler’s BIN directory and select one of the exe files. A good one might be the exe you run to start the IDE. An exe file can hold many icons, so when you select the exe, you may be given a choice. Look in various EXEs until you find one you want. Select the desired icon in the lower pane and press Ok. After a delay of a couple seconds, you should see the icon change on the desktop.

Windows Environment Variables

Locations of global environment variables such as path:

1. 95/98/Millenium: c:\autoexec.bat
2. NT4/2000/XP: System applet in Control Panel. Then:

   NT4: Environment tab.
   2000/XP: Advanced tab, then press Environment Variables button.

   Tip: Windows Logo Key + Break is the keyboard shortcut to the System applet.

SMX Utilities

The following is a summary of the utilities provided with SMX. Only the utilities appropriate for your release are included in it.

BINTOC

Converts a file, such as an HTML page, into an array of hex digits so it can be compiled and linked with the application. Supplied with smxNet.

Syntax: bintoc <infile> <outfile>

DOS_CMD

Used to run DOS commands from a make utility if this is not supported by the make utility or if there are problems running particular commands.

Syntax: dos_cmd <command>

FlashImage

Creates a flash disk image for NAND or NOR flash. It is supplied with the NAND and NOR drivers. See the readme.txt in this directory for details and syntax.

MIBTOC

MIB to C translator for smxNS SNMP Agent. It is supplied with the smxNS SNMP Agent add on.

Syntax: mibtoc <infile> [outfile]

NSBLDPG

Converts HTML pages to C to add to the application, for the smxNS web server.

Syntax: nsbldpg <cfgfile>

cfgfile is the full path and name of the .cfg input file. Enclose it in quotes if it contains spaces.

PREFRMT

Converts dial scripts to C for use with smxNS PPP and SLIP. Supplied with smxNS.

Syntax: prefrmt <in.scr> <out.scr> <down.scr> {usrN.scr}

REALTIME

Creates a web page identical to the input web page except it adds an HTML comment at the top of the file that specifies whether the file contains real time data. This avoids the need for the Web server to do another pass of the web page to determine if there is any real time data. Supplied with smxNet.

Syntax: realtime <infile.htm> <outfile.htm>

STRIP

Creates a condensed log file from the log file generated by capturing all output from the make utility. Strip keeps only important lines, such as warnings and errors, to make it easy to see if the build was successful or what the errors were. See the section below about strip32.

Syntax: strip <inlog> [outlog] [delay]

If no outlog is specified, output is printed to the screen. delay is in seconds and occurs before stripping. Its purpose is to allow pausing a batch file that calls strip, since the DOS batch language does not have a command to pause for a length of time. (The DOS pause command requires pressing a key to continue.) SMX mak.bat files use this to pause briefly before they type the stripped log to the screen, so that any output from the make can still be seen before the log file is displayed.

strip.exe can only accept pathnames that use the 8.3 naming convention. strip32.exe is the Win32 version and supports long file and directory names. (Put quotes around the arguments if there are spaces in the names.) Our batch files that call strip use relative paths (e.g. ..\..), and the file names are short, so the DOS version works fine. The problem is when using it from an IDE or your own batch file, if you install SMX to a path that has a directory name longer than 8 characters. Both versions of strip produce identical results and both run from Windows. The non-32 version also runs from DOS.

TestComm

Windows program used to test the smxUSBD serial driver. Supplied with it.

TestSocket

Windows program used to test the smxUSBD Remote NDIS (RNDIS) driver. Supplied with it.

usbdfu

Windows console program used to test smxUSBD Device Firmware Upgrade. Supplied with it.

Tools

This section documents tools used for different CPU versions of SMX. Also see the Tools section in each CPU section.

NMAKE

Switches

/F	Specify name of makefile. Must have a space between it and makefile.
/D	Displays time stamp information during the NMAKE session.
/P	Displays NMAKE information, including all macro definitions, inference rules, target descriptions.
/X	Redirect errors to a file.
v=	Macro assignment that can be overwritten by assignment in makefile.

Errors and Problems

1. error U1033: syntax error : 'EOF' unexpected

   Cause: The closing set of angle brackets for an inline response file are not at the beginning of the line.

2. error U1077: command has returned non zero.

   Some real mode DOS programs do not run directly from nmake. Call them using dos_cmd.exe in SMX\BIN.

3. Out of environment space: Increase the environment space with the config.sys command:

   SHELL=command.com /E:1024 /p

4. NMAKE version 1.4 will not stop building if a command returns an error. Use nmake 1.5 or greater. See: http://support.microsoft.com/support/kb/articles/Q132/0/84.asp

Tips

Debugging

1. Open app.c and set a breakpoint in LED_LSR() or the loop in LED_task_main(). If you can run to this breakpoint repeatedly, the software and hardware are probably running fine.

2. If smx does not seem to be running correctly, set a breakpoint on **smx_EMHook()** in main.c. If this breakpoint is ever hit, an smx error has occurred. You can inspect smx_ct and the call stack to see who caused it. If the error is SMXE_OUT_OF_ or SMXE_INSUFF_, increase the appropriate setting in APP\acfg.h.

3. The Diagnostic window in smxAware shows a list of the errors that occurred, in order. If you don’t have smxAware, you can look at the global smx_xerrno, which indicates the number of the most recent smx kernel error. 0 means no error has occurred. See xdef.h for the error numbers. To see the error buffer (without smxAware), inspect *smx_ebi to *smx_ebn (smx_ebi[0] is the first error).
4. If the debugger’s Stop button does not work, and the debugger uses a software debug monitor running on the target (not JTAG or BDM), ensure SB_DEBUGGER_IRQ is set (in bsp.h) to match the IRQ used for debugging. ainit() unmasks this interrupt.

ARM

See section ARM-M for information about Cortex-M. This section is for traditional ARM processors (ARM7, 9, etc). Since the same tools are used for both, tool information is presented only in this section.

Architectural Notes

ISRs

See the section ISRs in the Common Notes/ Coding Notes section at the beginning of this manual for general information about writing and hooking ISRs.

The interrupt controller on ARM chips varies because this is not part of the ARM core. The ARM architecture only specifies the format of the Exception Vector Table and that there is only one IRQ vector in it.

ARM chips all seem to use one of two ways of dealing with this. Some hook a single, master ISR to that one IRQ vector that prioritizes and dispatches the user's ISR, **in software**. For some ARMs, this involves a fair bit of code that runs for each interrupt, which hurts performance. Many newer ARMs of this type improve on it by doing prioritization in hardware so only the dispatch must be done in software. Other ARM chips implement their own internal vector table and do the prioritization **in hardware**. These are discussed in turn. Fortunately, the newer ARMs seem to have either a vectored interrupt controller or at least do prioritization in hardware so only a simple dispatcher is needed.

For ARMs that require software vectoring, a dispatcher routine is needed to call the appropriate ISR function. For smx, we encapsulate this with code that does the equivalent of smx_ISR_ENTER() and smx_ISR_EXIT(). Only this one master routine uses these macros; user ISRs are normal C functions.

XSMX\xarm_*.s implements this master ISR, called smx_irq_handler. At a high level, it looks like this:

```
ISR enter code
call dispatcher to run appropriate user ISR
ISR exit code
```

Since the ISR enter and ISR exit code is done in this single hardware ISR, your ISRs are to be written as simple C functions that are called by the dispatcher. In your functions:

1. Do not use the interrupt keyword (or __irq, etc).
2. Do not use smx_ISR_ENTER() or smx_ISR_EXIT().

You may call smx_LSR_INVOKE() from your ISR function, as usual. Also, the usual rule about not calling SSRs from ISRs still applies. For these ARM chips, a user ISR looks like this:

```
void MyISR(void)
{
    //...
    smx_LSR_INVOKE(my_lsr);
    //...
}
```

Notice that it is a normal function and does not use smx_ISR_ENTER() or smx_ISR_EXIT().

The dispatcher can be complicated or simple depending upon the processor. It is complicated and slow if the prioritization must be done in software. In this case, hopefully the processor vendor supplies this routine. This is true for the LH7A400 for example (not the LH7A404).

The dispatcher is simple for ARMs that have a register that indicates the IRQ number of the highest priority pending interrupt. In this case, we maintain a vector table in software and simply call into it using the register value as the index. This is true for the DragonBall MX1/MXL and STMicro STR7, for example.

ARMs that do hardware vectoring, such as the Atmel AT91 family, use a clever technique: The interrupt controller on these processors has an internal vector table that is set when you hook your ISR, and they have a register with the address of the highest priority ISR that should run. The single ARM IRQ slot is programmed with an instruction that does an indirect branch via the chip's register that holds the address of the highest priority ISR. In this case, ISRs are written as is typical of other smx versions: Each is hooked to its own vector and uses smx_ISR_ENTER and smx_ISR_EXIT. They cannot be fully coded in C, however. Instead, the outer shell that uses smx_ISR_ENTER and smx_ISR_EXIT must be written in assembly, and it calls the C function to do the real work. The smx_ISR_ENTER and smx_ISR_EXIT macros in XSMX\xarm_*.inc are the same macros used in the dispatcher in xarm_*.s.

For ARMs that do hardware vectoring, an ISR looks like this:

```
; file.s

IMPORT  MyISR
EXPORT MyISRShell

MyISRShell
    smx_ISR_ENTER
    BL      MyISR
    smx_ISR_EXIT

/* file.c */

void MyISR(void)
{
    //...
    smx_LSR_INVOKE(my_lsr);
    //...
}
```

Notice that the C function is a normal function, and that smx_ISR_ENTER() and smx_ISR_EXIT() are done for each ISR, in assembly.

The following files are provided for interrupt handling.

Software Vectoring:

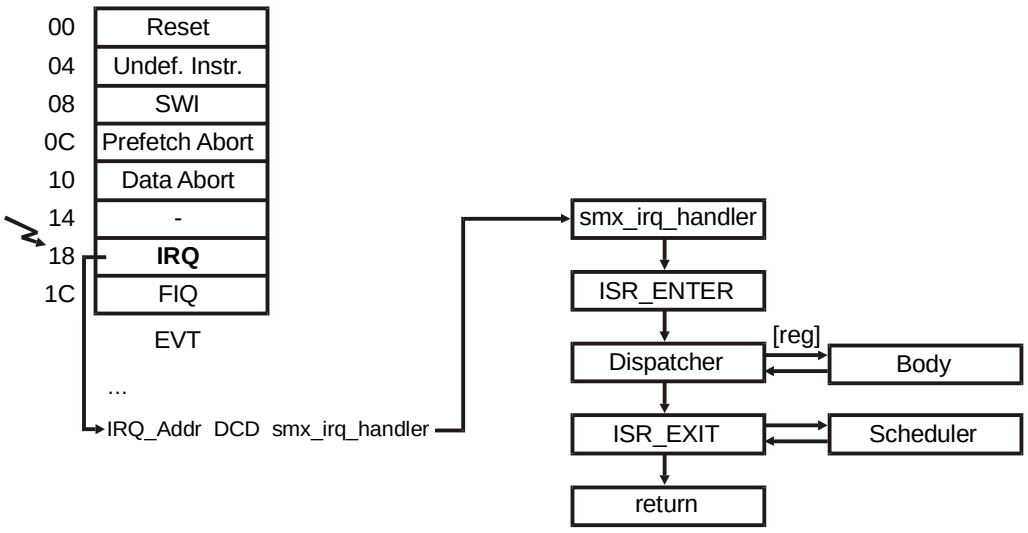
XSMX
xarm_ads.s: single ISR calling dispatcher; ARM DS/RealView/MDK assembler
xarm_gcc.s: single ISR calling dispatcher; GNU C preprocessor then assembler
xarm_gnu.s: single ISR calling dispatcher; GNU assembler
xarm_iar.s: single ISR calling dispatcher; IAR assembler

Hardware Vectoring:

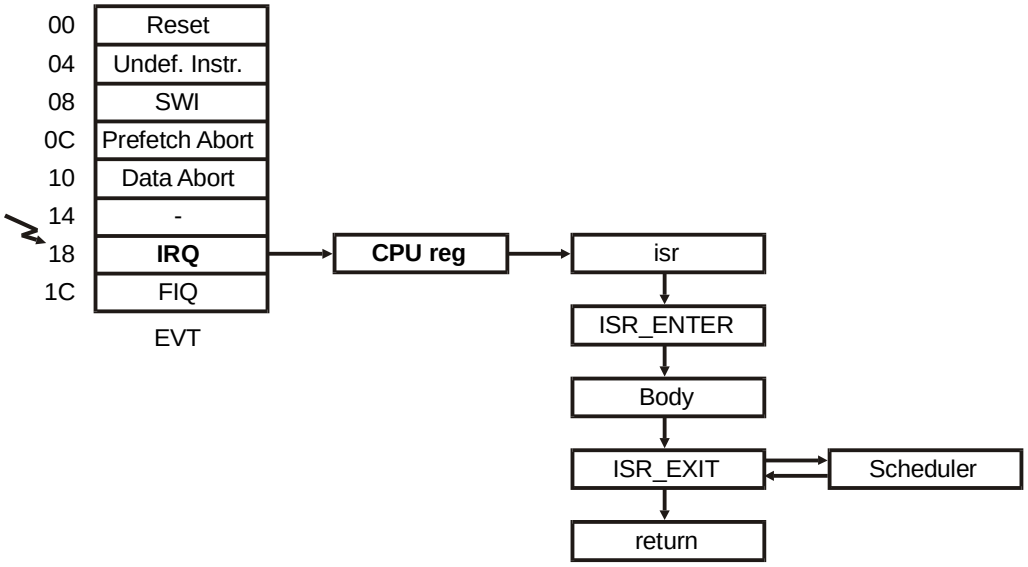
XSMX
xarm_ads.inc: smx_ISR_ENTER/EXIT macros; ARM DS/RealView/MDK assembler
xarm_gcc.inc: smx_ISR_ENTER/EXIT macros; GNU C preprocessor then assembler
xarm_gnu.inc: smx_ISR_ENTER/EXIT macros; GNU assembler
xarm_iar.inc: smx_ISR_ENTER/EXIT macros; IAR assembler

BSP\ARM
isrshells_ads.s: ISR shells that use macros (add your shells); ARM DS/RealView/MDK assembler
isrshells_gcc.s: ISR shells that use macros (add your shells); GNU C preprocessor then assembler
isrshells_gnu.s: ISR shells that use macros (add your shells); GNU assembler
isrshells_iar.s: ISR shells that use macros (add your shells); IAR assembler

The code for smx_ISR_ENTER/EXIT() is fairly complicated because it must switch out of IRQ mode back to the task's mode (i.e. Supervisor Mode (SVC)) and check whether to branch to the LSR and task schedulers. The following diagrams summarize operation of this code:



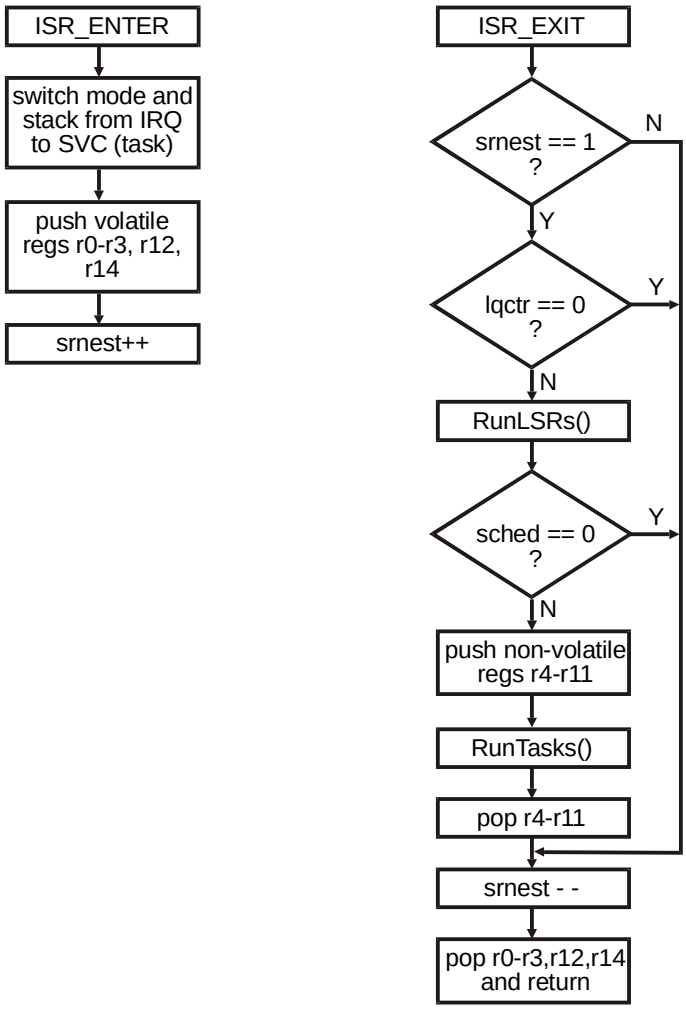
Software Vectoring



Hardware Vectoring

Notice that in the case of software vectoring, the branch is done via the literal pool (table of addresses) following the EVT, but in the case of hardware vectoring the branch is done via a register in the CPU's interrupt controller. **The key point is that the CPU register can change** (to be the address of the next ISR to run). In the software vectoring diagram, [reg] means a possible branch through a CPU register; i.e. prioritization is done by the CPU's interrupt controller, not in software.

Operation of smx_ISR_ENTER and smx_ISR_EXIT is shown below. Note that these show the main steps and omit complexities such as switching to the system stack and calling the pre-scheduler code.



Thumb Code

We have tested that the smx library, smxNS library, and Protosystem can be compiled and run in Thumb mode. A few changes were necessary, mainly to force ISR-related functions to be compiled for ARM mode. Changes were made to other SMX modules too, but these have not yet been tested in Thumb mode. SMX also supports linking other Thumb code, such as yours or in other libraries.

Alignment of Memory Access

Traditional ARM processors do not support unaligned accesses to memory. It is necessary to access a 32-bit value on a 4-byte boundary. Attempting to read or write at a byte or halfword address results in the access being done at the next lower aligned address, producing wrong data.

ARMv7-A adds support for unaligned data access. CP15 c1 SCTLR U bit is always 1. Setting the A bit of this register to 1 enables alignment fault checking so the processor will fault on any unaligned access. However, our experience on the TI AM335x and AM35x, Renesas RZ, and Freescale Vybrid VFxx processors is that even with alignment checking off (A bit set to 0), it still generates the fault for an unaligned access. Unfortunately, IAR EWARM generates code for this architecture assuming unaligned accesses are ok, so this causes faults. We had to add the switch --no_unaligned_access to all project files for these processors. Apparently, it is needed for all ARM-A processors.

Semihosting

Semihosting uses a software interrupt to interact with the host PC, such as to direct console output to a debugger window. Although it can be convenient for debugging, it can cause problems due to inhibiting interrupts awhile, causing your system to run differently than expected. For example, IAR EWARM v6.50 implements the time() function to make a semihosting call to get the time from the PCs clock, but this causes a long period where interrupts are blocked and one of our customers spent a couple days to find out why their regularly occurring interrupt would sometimes not occur regularly. As a result of this experience, we disabled semihosting in all IAR projects starting in SMX v4.1.1. If you have a problem with interrupts like this, you should verify this setting is disabled, since it is possible we could have created a new project by copying an old one from before the fix, by mistake.

Porting to a New ARM or Board

If you are using an ARM that we do not support, please follow this guide to adapt one of our existing BSPs to your particular ARM. Also refer to the Protosystem section, which follows. Only refer to the smx Porting Guide if you are porting to a new compiler or CPU family that is not yet supported by smx. See the section Common Notes/ Porting in this manual for an overview of porting.

1. Build the Protosystem project even if you don't have the board that our BSP targets, to ensure the tools are set up ok. See the appropriate Getting Started section in the SMX Quick Start for directions, if you have not done this already.
2. BSP\ARM\<cpu>\<board> contains BSP code, including some code from the board vendor. Replace that directory with your own, for your CPU and board. bsp.* and led.* are our files. Create new versions for your board. The main work is bsp.c — it is the implementation of the smx BSP API. Some routines will map onto the BSP code supplied with your board. See the section APIs/ BSP API in this manual, or comments in XBASE\bbsp.h if you are unclear about the purpose of a function.
3. CFG directory:
    a. ARM Developer Suite: Create new board .via files similar to the .via files provided (e.g. at91eb40a.via and at91eb40aa.via). Modify arm.via and arma.via to include your new ones.
    b. CrossWorks: Create a new board preinclude file similar to the .h file provided (e.g. at91sam7x256ek.h). Modify gcwarm.h to include it.

c. GNU X-Tools: Create new board .mki files similar to the .mki files provided (e.g. gcat91eb40a.mki and gcat91eb40aa.mki). Modify gcarm.mki to include your new ones.

e. IAR Embedded Workbench: Create a new board preinclude file similar to the .h file provided (e.g. at91sam7x256ek.h). Modify iararm.h to include it.

4. Create a new build directory for your board, under APP\ADS.ARM, GCW.ARM, or IAR.ARM. Copy the project files or makefile and other build files to the new directory and rename them for your target. Then modify the project in the IDE or modify the makefile to build your BSP files instead of the BSP files we provided.

BSP Files

1 **armdefs.h, armdefs.inc** ("_ads", "_gcc", "_gnu", "_iar" versions)

Master include file to include the appropriate BSP header files for the target. armdefs.inc is for assembly files. It has only a small subset of what is in armdefs.h.

2 **bsp.c**

Implements the BSP API routines documented in the APIs section of this manual. Some that are the same for most targets are implemented in XBASE\bbsp.c instead.

3 **bsp.h**

BSP-specific defines, types, prototypes, and configuration settings.

4 **isrshells.s** ("_ads", "_gcc", "_gnu", "_iar" versions)

These do ISR enter and exit. This is discussed at length in ARM/ Architectural Notes/ ISRs, in this manual. Please read that.

5 **lcd.c, lcd.h**

Simple API for writing messages to LCD. Used by lcddemo.c.

6 **lcddemo.c**

Simple test/demo for text LCD for boards that have one.

7 **led.c, led.h**

Simple API for writing LEDs. Used by LED_task and LED_lsr in app.c.

8 **term.c**

Terminal I/O routines for message output and keyboard input. Interfaces to UART driver.

9 **uart.c, uarti.c**

Polled and interrupt-driven low-level routines. The latter are used by high-level interrupt-driven UART driver.

10 **AM, AT91, LPC, STM32, TM, …** (subdirectories)

Subdirectories containing BSP files for the indicated board or family.

Evaluation boards provided by ARM vendors typically come with board support code (i.e. drivers). In some BSPs, we have interfaced our BSP layer (bsp.c) to the code they provide. Since smx requires only a few services, such as a timer, interrupt masking, unmasking, and hooking, and a UART for console output, much of the code they provide is unused. We have copied only the files we needed to subdirectories under BSP\ARM, and we have made any necessary changes to them. Look at the other files provided on their web site or the CD supplied with the board to see what else might be useful to you. Because every board has a different set of BSP files, it is too much to document here. Refer to any documentation the vendor provides, or simply look at the comments at the top of those files.

It is typical for the code to assume compilation with a C compiler not a C++ compiler, so you may need to wrap each file with extern "C" { }, to avoid name mangling so the linker can resolve references from assembly files. This is necessary if you compile with a C++ compiler. See the BSP files we used to see how we did this, if necessary.

BSP API Extensions

BOOLEAN **sb_IRQTableEntryWrite**() — parameters vary

> Changes an entry dynamically in irq_table[]. Generally, irq_table should be initialized statically and left alone, but this function is provided in case there is a need to change it dynamically. After calling this, call sb_IRQConfig() to make the change in the interrupt controller. The parameters vary because the fields in irq_table vary depending on the interrupt controller for a particular processor.
>
> This function is primarily provided for assembly access, since it may not be possible to access a C structure in assembly. Even for C, it is better style to call this function rather than modifying the structure directly.

ARM Developer Suite (ADS.ARM)

Last updated for ADS v1.1. DS, RealView, and MDK-ARM are based on these tools.

Build Targets

The project files have the standard 3 build targets (Debug, Release, and ROM), but the ROM target is currently not supported.

Preinclude Files (Via Files (.via))

Several "via" files are provided in the CFG directory that define global options for the build such as the CPU, board, and which SMX libraries to link. These contain switches that are passed on the compiler/assembler command line.

These override IDE settings. In other words, don't be surprised if you change one of these settings in the IDE and it has no effect; you have to change it in the via files. We specify these in

via files rather than in the IDE to ensure that the same settings are used for all builds (all libraries and the application). Also, using via files is convenient since it allows us to provide 1 via file for each target board that has all of the pertinent settings. Otherwise, switching to a different target would require changing the setting in 4 places in each target of each project file!

After changing a setting in a via file, such as enabling or disabling an smx library, do a clean build by cleaning object code out of targets before rebuilding. Any changes to global settings (CPU, calling convention, etc) require rebuilding all libraries clean. Changes to select libraries and demos require only rebuilding the Protosystem/application clean.

Note: All files are provided in 2 forms, for the compiler and assembler. Assembler versions are suffixed with "a".

Main Via Files:

arm.via	Includes board via files (select the one you are using). Specifies other global switches and the smx module libraries and demos to link.

Board/Processor Via Files:

lpc2378keil.via	Settings for Keil LPC2378 board
…etc.	

Startup Sequence

assembly startup code -> __main -> main() -> …

__main is the assembly startup code in C library (clears BSS, etc). Following main() is the standard sequence shown in the SMX Quick Start, in section SMX Startup and Scheduler Operation.

__packed Keyword

Don't use this keyword for structures used to overlay IO ports. If used, the compiler will not assume that elements are properly aligned on 32-bit boundaries, so it will call a helper routine, __rt_uwrite4, which chops the write into 4 byte-sized writes. Typically, an IO write will fail unless done as a single 32-bit or 16-bit write.

Troubleshooting

1. Problem: A setting change in the IDE seems to have no effect.

   Cause: The option may be set differently in a via file (.via files in the CFG directory), and these override settings in the IDE. See the discussion of via files in the section ARM/ ARM Developer's Suite/ Via Files for the reasons we use via files.

   Solution: Check the .via files.

GNU ARM

Distributions

Despite sharing a common name, all distributions of the GNU C/C++ compiler are different. Some replace components with proprietary tools, such as the IDE and debugger. Some may change the interface to the underlying tool. For example, assembly files may be run through the C preprocessor rather than the assembler's preprocessor, requiring use of #ifdef rather than assembler preprocessor directives, or there may be other syntax differences, such that a different version of each assembly file must be created. A bigger problem is that there is forking among the releases, so they have different sets of switches. To make things more confusing, some distributions supply documentation from another distribution that does not exactly match their own, so that it documents switches that don't exist, or the usage differs. Veteran GNU users undoubtedly know all this, but if you are new to it, please expect these frustrations, if you have to do some work to port our release to your particular GNU tools.

We cannot support every distribution of the GNU tools. Instead, we have selected Rowley CrossWorks since it is the cleanest and most professional we've seen, and it runs natively on Windows (no Cygwin!). Several years ago, we supported GNU X-Tools from Microcross, and we retain that section below, in case it is helpful. We may also support other GNU distributions packaged with particular processors, just for those processors.

If you are using different tools, you will need to create the project files, makefiles, etc. and possibly make syntax changes to the code. We cannot provide much support for this. In GNU releases, we provide all GNU files for all versions of GNU we support, so you can use the files that are closest to what your tools require. You should simplify your release by deleting the files you don't need.

We recommend that you study the CrossWorks project files (.hzp) carefully in your text editor and note all options such as defines, include paths, and other settings, and then make similar settings in yours.

GNU / CrossWorks ARM (GCW.ARM)

Last updated for CrossWorks v2.1.1. v1 information has been removed from this section.

Please read the GNU ARM section first for an overview of our GNU support.

SMX supports CrossWorks ARM from Rowley Associates (www.rowley.co.uk), starting with v1.7 Build 3. It may work on older version, but we have not tested this.

With v2, the user interface has been completely redone, so it looks like a new IDE. Windows and settings have changed a lot, so v1 information has been removed from this section to avoid complicating the discussion with two descriptions for making every setting. The project file format has remained nearly the same, which would be convenient if you needed to revert to the older tools for some reason.

The nicest thing about CrossWorks is that the Windows version of the tools is natively hosted on Windows and does not require Cygwin! It also has a nice IDE and debugger built in, unlike some

GNU distributions. These tools are proprietary, not Eclipse-based. (The Eclipse IDE has some surprising and frustrating limitations on things that are basic features of an IDE.)

The version of GCC used in each version of CrossWorks is indicated in the release notes. In v1.7, it was based on the Sourcery C/C++ compiler from Mentor Graphics. There is no indication of this in the v2 release notes.

GCW: We use this in the name of the build directory. The G is for GNU and is useful to keep all of them sorted together.

Installation

Follow the directions in the CrossWorks documentation. Then download the ARM Support Package for your hardware using Tools | Install Packages… in the IDE or from www.rowleydownload.co.uk/arm/packages.

Project Files

Project settings are saved in two files. The **.hzp** file is the key project file. It should never be deleted. **.hzs** stores session settings such as open files, breakpoints, watches, etc. It can be deleted, and clean one will be automatically generated when you open the project.

Important Notes:

1. See Build Targets below for explanation of what CrossWorks calls Configurations.
2. The list of options displayed changes depending on which node you have selected in the Project Explorer window. For example, the File Type setting only appears when you select an individual source file (not a folder nor a higher-level node).
3. Difficulty finding settings: Select the **Common** target (from the drop list in the Properties Window) to see many global settings. Also, due to the hierarchy of the project, you may need to click on the top level Solution node or the project node under it. The IDE does not show some inherited settings, such as Additional Compiler Options and User Include Directories. For example, Release is based on Common, but if you have the Release target selected, you do not see the Common settings. Sometimes you may prefer to just look at the .hzp file in your text editor, as we often do.
4. If you have any problems with options not taking effect, open the project file (.hzp) in your editor and study it. It could be that conflicting settings are being put at different nodes of the hierarchy. We had some trouble with this early in our work. The project file format is so simple, it is easiest to make some changes by editing it. Also, when you temporarily make a setting and then reverse it, remnants get left behind which can clutter the file. You may want to periodically review the project file in your editor and clean it up so any important overrides are more easily seen.

Build Targets

CrossWorks calls these **"Configurations"**. The project files have our standard 3 build targets (Debug, Release, and ROM), which can be selected in the IDE from the drop list at the top of the Project Explorer window. In the drop list in the Properties Window, it lists the others that these

are built from: ARM, Common, and Flash. See Build | Build Configurations in the menu. These configurations are built-into CrossWorks. Debug, Release, and ROM are targets we created.

In general, the application Debug and Release targets use linker settings to locate the code for RAM; the ROM target has different settings for ROM/Flash. The Debug target locates to ROM/Flash for small SOCs that have only a small internal SRAM and do not support external RAM.

For our libraries, you should build the Release target. Unfortunately, it is not possible for us to set the default target in the IDE, so when you open the project, you need to change the selection from Debug to Release. Only if you need to debug our library code should you build and link the Debug library. It is also not possible to change the order of the targets in the drop list for selecting the active configuration.

Preinclude Files

Preinclude files are header files that are included ahead of every source file. We use them to define settings that should be used across all projects (libraries and application). Mostly, they define preprocessor symbols to indicate the processor and board, and they have defines to indicate which SMX module libraries and demos to link. The following is a summary:

C/C++

gcwarm.h	Master preinclude file. Configuration of libraries and demos is done here.
<board>.h	Board preinclude files; included by gcwarm.h.

Assembly

same files

How this works: The project setting Additional C/C++ Compiler Options that specifies the -include switch and points to CFG\gcwarm.h. It includes the board header file that is uncommented in it (also in CFG). The reason this works for the assembler too is because assembly files also go through the C preprocessor. To see this setting in v2, select the Project node in the Project Explorer window, and then look at the Compiler Options in the Properties Window.

Note that the reason for using preinclude files even though we can put all defines into the IDE is that this makes it easy to use the same defines in all projects (e.g. libraries and application). This makes it easy for us to switch from one board to another and avoids the need for us to repeat the same defines in every build target of every project — and maintain them.

Note: If you are using some version of GNU other than CrossWorks, you can use these same preinclude files, if your tools support them, or else copy the defines into your project files or makefiles.

Startup Sequence

assembly startup code -> main() -> …

In the assembly startup code, we copied the needed code from the CrossWorks startup code, crt0.s that clears BSS, copies initialized data from ROM to RAM, runs C++ initializers, and then

branches to main(). Following main() is the standard sequence shown in the SMX Quick Start, in section SMX Startup and Scheduler Operation.

Usually, we use the compiler's startup code (we call it from the end of ours), but in this case, the CrossWorks code has some overlap with ours and also does things we don't need, so we decided it was simplest to just copy what we needed into ours.

Optimization

Probably, you can use any optimization level for your code and our libraries, except for the smx scheduler in the smx kernel library. As of CrossWorks v1.7 Build 13, **it is still necessary to compile the smx scheduler at Level 1 or None**. (Note: This should be re-tested with the latest CrossWorks v2.x and smx v4.1.) This is done by explicit project settings just for xsched.c (look at the .hzp file). We have not studied the cause of failure when using other optimization levels for the scheduler. Most likely it is due to some optimization and the nature of the scheduler. Switching stacks is not expected by the compiler and is the main cause of difficulty, as well as inline assembly. Of course, if you have run-time problems, try lowering the optimization level of your code and our libraries.

By default, our project files set the optimization level to None for Debug targets and Optimize For Size for Release and ROM targets. In our brief testing, we found that the performance of Optimize For Size is only slightly lower than Level 3 (maximum optimization), but code savings is substantial.

Note that the optimization levels offered by the IDE are actually groups of compiler optimizations. There are many optimizations that are not in these groups. Please refer to section 3.10 Options That Control Optimization in the CrossWorks online help for details about these switches.

C++

There seems to be no switch or pragma that can be used to globally force a C++ compile in the CrossWorks IDE. GNU compilers have traditionally compiled as C or C++ based on the filename extension. More recently, the –x switch was added to the gcc.exe interface driver but not to the compiler itself, and the CrossWorks IDE bypasses that and calls the compiler directly. It seems the only solution to force a C++ compile for a .c file in the IDE is to set the file type as C++ for each file in the project properties:

Right-click on the file and select Properties. In the dialog box, find the File Options settings and set **File Type** to **C++**.

Other compilers we have supported have a switch or pragma to set it globally, so when we link smx++ or PEG+, we just compile everything for C++. In the past it was necessary to compile the smx kernel for C++ because some of its API calls use default parameters and pass by reference. Now, it is only necessary to force a C++ compile for .c files that contain any C++ code, such as initmods.c, and files that call smx services using default parameters or pass by reference.

Exception Handling

To enable exception handling, select the Common target in the Properties Widow. Find the Code Generation Options and set **Enable Exception Support** to **Yes**.

It is also necessary to set **Use GCC Libraries** to **Yes** in the Library Options.

Assembler

CrossWorks first runs assembly files through a C preprocessor, so it is necessary to use C-style preprocessor directives (#if rathe than .if), so we had to create a new variant of our assembly files for this case. These are suffixed with _gcc.

Linker

The linker uses a Memory Map File and a Section Placement File to control where code and data are located. These are XML files in the Protosystem build directory. CrossWorks creates a linker command file (.ld) from these files and puts it in the build directory along with the object files. This is the actual input to the linker. If you get a build error in the .ld file, look at this file, and then correct the problem in the map or placement file.

The Memory Map File and Section Placement Files come from or were derived from files in the ARM Support Packages you can download via Tools | Install Packages… in the IDE or from Rowley's website (www.rowleydownload.co.uk/arm/packages). They are specified in the Linker Options in the project properties. The files to use are specified independently for each build target. The Memory Map file is in the System Files folder shown in Project Explorer. This file is target-specific and specifies the addresses for the start of each segment.

We copied these files from the CrossWorks release (targets directory) to our build directory under APP\GCW.ARM and set our project files to use them. This way they are included with SMX releases, and the project continues to build with these same files regardless of whether CrossWorks modifies their files.

Debugger

The debugger built into the IDE is good. It supports various target connection types, but we've only used it with the J-Link JTAG unit.

We provide **threads.js** in APP\GCW.ARM to support the Threads window (Debug | Debug Windows | Threads). The Protosystem project points to this file. This is the best kernel awareness that is possible; CrossWorks does not support kernel-aware DLLs such as smxAware. The fields in the Threads window are controlled by the IDE; we can only supply the data. The IDE can display thread registers, but we did not support this because it is not useful. The registers at the time of a task suspend would be what they were when saved by the tail of the ISR or in the scheduler, not what they were on the last statement that executed from the task code itself. Even if they were, it is questionable how useful that would be.

Tips for setting up and using the debugger:

1. First set up the JTAG connection. Here are the steps for J-Link: Target | Targets, then select Segger J-Link in the scroll list. In the Properties Window, on the line J-Link DLL File, click the button "…" and browse to the location of JLinkARM.dll. This assumes you have already installed the J-Link driver that you got with your J-Link or downloaded from Segger. Close the Targets window.

2. Connect to the JTAG unit. Steps for J-Link: Target | Connect Segger J-Link.

3. Press the Start Debugging button to download the program and run to main(). Or press the Step Into button to stop at the first line of the assembly startup code.
4. Mixed C/Assembly and Disassembly window: Debug | Disassembly. This opens a new window. For interleaved source display, click the down arrow at the right of its local toolbar and select Show Source in Disassembly.
5. Locals, Globals, Threads, etc: Debug | Debug Windows | …

Flash Loader

CrossWorks provides the LIBMEM PRC Loader (with source code), which can program the image into flash. It is included in the ARM Support Packages, which you can download from www.rowleydownload.co.uk/arm/packages/.

To use the flash loader, go into Project Properties for the ROM target and scroll down to the Target Options section. Make these settings:

Loader File Path: Set to the path of the LIBMEM RPC Loader file in the targets directory.

Loader File Type: Set to "LIBMEM RPC Loader"

In Target Script Options:

Reset Script: Point to a script file that does "FlashReset()" to reset the board.

Thumb Support

You can select ARM or Thumb in the IDE project properties. In the Code Generation Options section, change the "Instruction Set" setting, and in the Library Options section, change Library Instruction Set to Thumb. Or you can manually edit the project file (.hzp) and set all occurrences of "arm_instruction_set" and "arm_library_instruction_set" to "ARM" or "Thumb".

It seems the compiler has no #pragma to allow forcing specific functions to be compiled for ARM as some other compilers have; it is necessary to compile the whole file for ARM. We compile the smx scheduler, xsched.c, for ARM because some functions in it need to be in ARM mode.

Using CrossWorks

1. The Protosystem project files are located in the board directories under APP\GCW.ARM.
2. See the Debugger section above for tips for using it.
3. Creating a new project file for your board: Copy our Protosystem project file (.hzp) and rename it as appropriate. It is often easiest to edit it with a text editor rather than in the IDE:

    a. Replace the solution name, project Name, and Targets with the correct processor.
    b. Set the RAMEND address, which is used in boot_gcw.s to set stacks. It is the end of RAM.
    c. Change the file name and select the correct memory map file for the processor.

d. Add preprocessor defines in the Assembler settings for symbols defined in the board preinclude (.h) file (see step 2).
e. Add your BSP source code to the project file.

4. Assembly Listings: Right click on the file and select Compile. Then right click and select Disassemble. This opens a mixed source/disassembly listing window. You can save it to a file. If you just want to generate the assembly code, look under Compiler Options in project properties, set Keep Assembly Source to Yes.

5. The link map is specified by a section placement file and a memory map file. The placement files are in the build directory (APP\GCW.ARM and the memory map files are one level down in the directory for your board. The tools automatically create the linker command file (.ld) from these, which is put in the directory with the object files. The placement files are intended to be general and shared by multiple targets (boards). They are provided by CrossWorks in the targets directory but we copied them so they are part of the SMX release.

Tips

1. HTML help files: Open <cwdir>**html\index.htm** in your browser. The left pane has the contents links to all sections. It is fully expanded if your browser is set to block active content. Look for the bar at the top of the browser window to allow blocked content and allow it. Then the tree will collapse. It is very hard to use when fully expanded.

Troubleshooting

1. Problem: You are unable to expand any smx global data structures (e.g. smx_cf, smx_ct, etc), and CrossWorks complains it doesn't know the type of the variable.

   Cause: Probably there is no symbolic information for them.

   Solution: In the smx library project, in the Release target, ensure that debug symbolics are enabled for xglob.c or the whole library.

IAR Embedded Workbench ARM (IAR.ARM and IAR.AM)

Last updated for IAR v7.40. IAR v4 information has been removed.

Version

Use the version of IAR EWARM indicated by the readme.txt file in the root of your release or by the suffix of the project files. For example, iar740 in the name App_am3359sk_iar740.* means v7.40. We may have provided project files for multiple versions, in which case you have a choice. The suffix is necessary because changes in the IDE from version to vesion require it to convert the project files, but once converted the changes cannot be reversed. It is often possible to use a newer version, but there could be build problems, so save a copy of the project files in case you need to revert.

Beginning in v5, the tools moved from IAR's proprietary UBROF object file format to the industry-standard ARM EABI 2.0 ELF/Dwarf object format. This was a major change to the tools, particularly the linker and assembler.

Project Files

Project settings are saved in several files. The **.ewp** and **.eww** files are the key project files. They should never be deleted. **.ewd** stores debug settings. If it is deleted it will be regenerated, but you have to reconfigure all the debug settings such as the path to the startup macro file you are using and the JTAG device selection and settings). Other project files, such as .dep and the whole settings dir can be deleted. The settings files hold less-important information such as window sizes and placement, positions in files, etc.

In this manual we refer to these files generally as project files, even though technically, the .ewp file is the project file. When we say to open the project, we mean to open the workspace file, .eww, using File | Open | Workspace.

Build Targets

The project files have the standard 3 build targets (Debug, Release, and ROM). The Debug and Release targets are linked with the _ram version of the linker command file (.icf); the ROM target is linked with the _rom version. For SoCs that have no external memory interface and only a small on-chip SRAM, there are only Debug and ROM targets, and both use the _rom linker command file.

Preinclude Files

Preinclude files are header files that are included by the IDE ahead of every source file. We use them to define settings that should be used across all projects (libraries and application). Mostly, they define preprocessor symbols to indicate the processor, board, etc., and they have defines to indicate which SMX module libraries and demos to link. The following is a summary:

C/C++

iararm.h	Master preinclude file. Configuration of libraries and demos is done here.
<board>.h	Board preinclude files; included by iararm.h.

Assembly

none

Only the compiler supports preinclude files, so for assembly, the defines are specified in the IDE, on the Preprocessor tab of the Assembler settings for each build target.

How this works: In the project options, select C/C++ Compiler in the left pane. In the right pane, select the Preprocessor tab. The line Preinclude file points to CFG\iararm.h and it includes the board header file that is uncommented in it (also in CFG).

Note that the reason for using preinclude files even though we can put all defines into the IDE is that this makes it easy to use the same defines in all projects (e.g. libraries and application). This makes it easy for us to switch from one board to another and avoids the need for us to repeat the same defines in every build target of every project — and maintain them.

Relative Paths

In order to allow you to install SMX to any place in your directory tree (and on any drive), it is necessary that the project use relative paths to locate the source code and other files in the project. EWARM does not have a checkbox to enable this as many other IDEs do. However, it provides "argument variables" that can be used to specify the paths relative to the project, compiler, etc. directory. We use the variable **$PROJ_DIR$** in many of the paths we specify. For a full list of these variables, see the Embedded Workbench User Guide, Part 7: Reference Information/ IAR Embedded Workbench IDE Reference/ Menus/ Project Menu/ Argument Variables Summary.

Predefined Symbols

The IAR compiler and assembler define quite a few symbols that can be used in the code. These are clearly documented in the respective manuals. For the compiler, see the C Compiler Reference, Part 2: Compiler Reference/ The Preprocessor/ Predefined Symbols. We use the following:

__IAR_SYSTEMS_ICC__	Used for sections we assume are the same for all IAR compilers regardless of target processor.
__ICCARM__	Used for sections that are ARM-specific.

Startup Sequence

assembly startup code -> ?main -> main() -> …

?main is the routine in the IAR runtime library (DLIB) that clears BSS, copies initialized data from ROM to RAM, runs C++ initializers, etc and then branches to main(). Following main() is the standard sequence shown in the SMX Quick Start, in section SMX Startup and Scheduler Operation.

Assembler

In order to make it easier to assemble code you have already written for another assembler, the IAR assembler can be set to be more flexible about syntax. From the Language tab of the Assembler settings panel in the IDE, check "Allow alternative register names, mnemonics and operands" or use command line switch –j. We have not tried this switch. See the IAR ARM Assembler Reference Guide.

Linker Command Files (.icf)

We created our linker files based on the samples provided by IAR.

In smx v4.1, we changed the .icf files for many targets to locate SDAR and ADAR (.smx_sdar and .smx_adar sections, respectively). When possible, we locate SDAR and the System Stack to internal SRAM for better performance, and we locate ADAR to external SDRAM to separate it from SDAR. SDAR holds smx objects and queues; ADAR is intended for application use. In cases where all the SRAM is reserved for SMX middleware products (e.g. smxNS and smxUSB), we could not locate SDAR to SRAM. In your case, space may be available due to not using one or more of those products, or you may have additional memory areas. You should change the .icf file to achieve the best location for these for your system. (Note that we did not modify the .icf

files for old processors that we feel are unlikely to be used in new systems, so if your .icf file does not locate these, just add “section .smx_sdar” and “section .smx_adar” to the “place” commands at the end of the file.)

Note that we typically located the System Stack after some other data, such as SDAR, so it would not be at the start of a region of memory. A stack overflow into non-existent memory is likely to cause a processor fault, which would halt the system, while overflow into other data may not be catastrophic, especially if there is unused space at the end, as is likely to be the case with SDAR.

Also note that only the IRQ and SVC stacks are used. These are arranged so that the unused stacks are before the SVC stack (used as the smx System Stack), so that any overflow would go into these unused stacks.

Link Map

Generation of a link map is controlled in project Options. Select Linker and then the List tab. It can be enabled or disabled. The link map produced is short and easy to navigate.

Binary Files

Some flash programmers require that the program be a simple binary image. The project Options specify what to create. Select Output Converter, and on the Output tab, check Generate additional output. Then select the output format from the drop list.

Debugger (C-SPY)

JTAG Units

C-SPY supports a wide variety of JTAG units, including any that use the RDI protocol, and even the low-cost Macraigor wiggler. We successfully use and recommend IAR I-jet or J-Link.

See the Embedded Workbench User Guide, Part 6: C-SPY Hardware Debugger Systems/ Hardware-Specific Debugging for directions to set up your debug hardware. Also, see the table at the end of the release notes which points to documents for the different IAR C-SPY drivers.

You should first install the software that came with your JTAG unit. For RDI devices, for example, the RDI driver is supplied with that software and you must point to it in the IAR project settings.

Breakpoints

When running from ROM/Flash, breakpoints can be severely limited, sometimes to 2, and some options in IAR use breakpoints. If you try to set multiple breakpoints and IAR’s Debug Log window reports “Failed to set breakpoint: Driver error.” it probably means you have exceeded the number supported.

To see all breakpoints in use during a debug session, select from the menu: J-Link | Breakpoint Usage. In addition to any breakpoints you have set, you may also see these:

“Stack window trigger”
“C-SPY Terminal I/O & libsupport module”.

The “Stack window trigger” breakpoint is associated with Project | Options | Debugger | Plugins | Stack. Turning this off frees a breakpoint, and the Stack view becomes unavailable. (The Call Stack view is still available.)

The “C-SPY Terminal I/O & libsupport module” breakpoint is needed for the feature to direct printf() to a terminal window in the debugger, and it may disable other support associated with C library exception conditions. We have not found how to disable it.

Flash Loader

EWARM has a built-in flash loader interface and includes pre-made flash loaders for several specific processors. They provide the source code for these and directions how to create a new loader for your processor. This is documented in the C-SPY Debugging Guide, Part 3: Advanced Debugging/ Flash Loaders/ Using Flash Loaders. Information about writing your own flash loader is given in a separate PDF file in the IAR arm\doc\FlashLoaderGuide.pdf.

The flash loader is part of the debugger. There is no menu choice to run it. To download an app into flash you initiate a debug session just like when debugging to RAM. EWARM automatically loads the flash loader into RAM on the board and then runs it to download a binary version of your application. When it is done you can either debug the application in flash or kill the debug session and run free-standing. (For that, power off the board, disconnect the JTAG unit, power the board on, and the application will start running.)

The EWARM documentation does a good job describing how to set up for flash loading, but here we give some additional guidance:

1. Project setup: The ROM target of SMX project files should already be set up properly. However, ensure the checkboxes are set for Verify download and Use flash loader(s) in the project options Debugger settings | Download tab. The project is automatically set to use the correct flash loader based on the selected processor, if a flash loader for it is provided.
2. Failure to Program Flash: If you get verify errors, try resetting or cycling power to the board and try again.

Using IAR EWARM

1. The Protosystem project files are located in the board directories under APP\IAR.ARM. For example, the workspace file for Atmel AT91SAM9G35-EK is here:

   APP\IAR.ARM\AT91\App_at91sam9g35_iar740.eww

   Open the project file for the eval board you are using, do a make, and then press the Debug button to download it to the board and debug. It should run as shipped. If not, contact us for help.
2. The Protosystem project files are set for Embedded C++ in the Compiler Language setting, but it can be unchecked if you are not using C++ modules such as smx++ and PEG.
3. Files are added to a project with by right-clicking on a node and selecting Add | Add Files….
4. File Organization: The organization of file nodes in an IDE project has no relation to their location on disk. This gives you the flexibility to add new groups and drag files into them in

the IDE without any worry about what directories they are in on the disk. (However, the IDE must be told the paths to all files it needs.)

If you do want to move a file on disk, the IDE will not be able to find it. You can either remove the node from the project and re-add it or manually edit the project file since it is in text format (XML).

5. Excluded Files: If a file in the project has a blank page icon next to it, it is excluded from the build. To change this, open project Options and check/uncheck Exclude from build in the upper-left corner of the dialog. This is a convenient way for us to exclude optional files so they may be re-enabled easily without having to browse to add them back to the project. Each build target (Debug, Release, and ROM) sets this independently.

Debugging with C-SPY

1. Some targets require initialization steps to be performed before the debugger is able to download code to RAM on the board. This can be handled with a C-SPY .mac file. If one is necessary for one of our BSPs, we provide the .mac file in the same directory as the project files, and the project file points to it (and runs it each time you initiate a debug session). In the C-SPY Debugging Guide, see Part 3: Advanced Debugging/ C-SPY Macros for documentation about the macros that are available. Also, see the subsection Reference Informationon Reserved Setup Macro Function Names to learn which macros the debugger calls and when during the setup process.

2. By default, the debugger runs through the startup code automatically and stops at main(). If you want to debug the assembly startup code, open the project settings and select Debugger in the left pane. In the right pane, check the Run to box and enter main in the text input box under it.

3. smxAware, included with smx, is a DLL that plugs into C-SPY to display smx objects. It includes graphical display features to show event timelines, stack usages, profiling, and memory usage and layout. See the smxAware User's Guide for full information.

Tips

1. The Multi-file Compilation option can be used to reduce code space. We found when used for the smx library, it reduced code size by about 4KB. When enabled, the IDE does not show compiling each file; instead there is a long pause while it compiles all files in the project. It may appear the IDE is hung, so be patient. To enable it, right-click the top node of the project and select Options…, then click the Multi-file Compilation checkbox in the C/C++ Compiler settings.

2. The compiler switch --no_const_align can be used for files that have string literals, such as XSMX\xem.c to reduce ROM usage, since it causes each string to start on a byte boundary, instead of a word boundary. There is no checkbox in the IDE; it is necessary to enter this on the Extra Options tab of the C/C++ Compiler settings in project options.

3. The compiler switch --no_unaligned_access is needed for ARM-A processors. See section ARM/ Architectural Notes/ Alignment of Memory Access.

Troubleshooting

—

Tools

JTAG Units

You need a JTAG unit to connect to your target board for debugging. These range from high-end units that do tracing and have other advanced features to low-cost wigglers that provide minimal support. They connect to the board with a standard header, and to the PC via USB, Ethernet, or serial. Some use the RDI protocol defined by ARM, which is supported by most tools.

Abatron BDI2000

BDI2000 is a low-cost JTAG RDI unit that works well and supports both serial and Ethernet connection to the host. We have only tested it with AXD in ARM Developer Suite.

In the BDI configuration utility, set it to Stop not Reset, on the BDI Working Mode dialog box, if you want the startup code already in ROM to initialize the board. If your application contains the startup code, you can instead set it to Reset. Our .bdi files for Evaluator-7T and Sharp LH7A400 use the Stop method so that the on-board startup code will run; those for the Atmel boards use Reset since the startup code is in the application.

When using the Stop method, if there is a problem running after powering on the first time, try resetting the board. Also try increasing the Run Time setting in the BDI Working Mode dialog box. This gives more time for the boot code to run before the BDI unit stops it.

IAR (Signum) I-jet

I-jet is a low-cost JTAG unit that is integrated well with IAR Embedded Workbench. I-jet Trace is a higher-cost model that supports the ETM (Embedded Trace Macrocell), to store an execution trace.

IAR (Segger) J-Link/J-Trace

J-Link is a low-cost JTAG unit that is integrated well with IAR Embedded Workbench. J-Trace is a higher-cost model that supports the ETM (Embedded Trace Macrocell), to store an execution trace.

Lauterbach TRACE32

TRACE32 is a fairly expensive JTAG RDI unit with advanced capabilities. At least a few SMX customers use it and praise it. Lauterbach added smx kernel awareness to it themselves. This is one to investigate if you are in the market for a JTAG unit. We have never used this unit ourselves.

Signum JTAGjet

JTAGjet is a low-cost, high-speed JTAG RDI unit that connects to the host by USB. IAR acquired Signum and added built-in support for it starting in v6.30. As of the v6.30 release, they are continuing to work to make all JTAGjet and Chameleon functionality available from within EWARM. In our past attempts to use JTAGjet with older versions of IAR, we had mixed results. We expect now integration will be as good as J-Link, and this is a superior unit. We don't know, however, whether this will be true for the old JTAGjet you might already have, or whether it is necessary to buy a new one or more likely, upgrade its firmware. We have little experience using JTAGjet with IAR at this time. Follow the instructions provided by IAR. Also note that they have introduced a new unit called I-jet.

If you have problems, be sure your PC has a USB 2.0 controller. If not, buy an adapter card.

Drivers

Disk

See smxFS documentation.

Ethernet

See smxNS or smxNet documentation.

LED

Simple LED routines are provided in **led.c** in the board directory in the BSP (e.g. BSP\ARM\STM32\STM32F4xx\STM3240G-EVAL\led.c). See APIs/ LED API at the end of this manual.

UART and Terminal

We provide the UART drivers supplied by the board/processor vendor, with any modifications needed to integrate them with smx. These are in the subdirectories under BSP\ARM. Each vendor's driver is different, so we cannot document them all here. Please study the source code to see how to use them.

If you wish to connect a terminal to one of these for input and output, ensure XBASE\bcfg.h is set so that:

```
#define SB_CON_IN 1
#define SB_CON_OUT 1
```

Specify the port for each in bsp.h as follows:

```
#define SB_CON_IN_PORT 1      /* 1 or 2 */
#define SB_CON_OUT_PORT 1     /* 1 or 2 */
```

These settings are independent. Input or output can be individually enabled and the port can be different for each.

term.c in the BSP directory interfaces to the UART driver to do terminal i/o. Also, sb_PeripheralsInit() in **bsp.c** calls the driver initialization routine.

By default, the drivers are configured for 115200-8-N-1. Turn off flow control in your terminal or terminal emulator.

Video (Graphics)

See PEG or C/PEG documentation.

Video (Terminal)

sb_ConWriteString(), and other functions in XBASE\bcon.c are mapped onto the UART driver API so text output goes out the serial port to a terminal. See the section APIs/ Video API at the end of this manual.

Other Notes

—

Tips

—

ARM-M (Cortex-M)

Architectural Notes

Overview

The ARM-M architecture is significantly different from the traditional ARM architecture used for ARM7, ARM9, etc. Despite the fact that it is called “ARM” and is supported by ARM tools, you should consider it a new processor architecture.

“Cortex” does not mean the new architecture; it is the “M” that matters. The Cortex-A and Cortex-R (ARM-A and ARM-R architecture) processors have the traditional ARM architecture. To summarize:

ARM-M:
Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4

Traditional ARM:
ARM7, ARM9, ARM11, StrongARM, XScale, etc
Cortex-A8, Cortex-R4

Architecture vs. Implementation: What has been confusing in the ARM world is that ARM numbered both the architecture and the implementation. The little “v” was how you could tell them apart. For example, ARM7 and ARM9 are based on the ARMv4 architecture. The name “Cortex” was introduced to break this pattern. Cortex-M4, M3, and M0 are implementations based on the ARMv7 architecture. Cortex-M1 is based on the ARMv6 architecture.

The key point is that ARM-M is basically a new processor, and as such, we assigned a different processor ID to it and created a separate set of build directories for it (xxx.AM instead of xxx.ARM). We have taken the more general view of calling it ARM-M rather than Cortex-M, in the hopes that it will support whatever future ARM-M processors are introduced, which could be named something other than Cortex.

ARM-M was designed for embedded systems, unlike ARM, and fixes the annoyances in ARM and goes further to offer new useful features. It is also simpler in some regards.

Since the same tools are used for ARM-M and ARM, we do not repeat tool information here. **Instead, please refer to the tool section in the ARM section of this manual**. Any additional notes for each tool are presented in sections here.

For information about the ARM-M architecture, we recommend the book “The Definitive Guide to the ARM Cortex-M3,” Joseph Yiu, ISBN 978-0-7506-8534-4, and the ARMv7-M manuals from ARM.

ISRs

See the section ISRs in the Common Notes/ Coding Notes section at the beginning of this manual for general information about writing and hooking ISRs.

For ARM-M, ISRs are simple C functions that require no interrupt keyword. For smx, you must wrap ISRs with smx_ISR_ENTER() and smx_ISR_EXIT(). No assembly shells are required, in contrast to traditional ARM, which required complex shells for smx. (The complexity for ARM is necessary due to the way mode switching works. It was necessary to switch out of IRQ mode immediately before allowing nested interrupts to occur.)

A difference from other processor versions of smx is that it is not necessary to increment srnest in smx_ISR_ENTER(), thanks to the RETTOBASE flag in the NVIC.

A C ISR simply looks like this:

```
void MyISR(void)
{
    smx_ISR_ENTER();
    //...
    smx_LSR_INVOKE(my_lsr, par);  /* optional */
    //...
    smx_ISR_EXIT();
}
```

Assembly macros are not provided and may never be, unless there is demand to write assembly ISRs. If there were some need to write an ISR in assembly, one could create a simple ISR shell in C and use the compiler to generate an assembly listing to start from.

ISR Priority Level

Basics:

1. Lower number is higher priority.
2. Priorities are generally not 0, 1, 2… but 0x00, 0x20, 0x40,… or similar. This is controlled by the number of priority bits, which are the high bits of the priority byte. For a processor that uses 3 bits, they are 0x20 apart. For 4 bits, they are 0x10 apart. Consult an ARM-M reference for more discussion.
3. The BASEPRI register allows disabling interrupts at a certain threshold and lower priority. PRIMASK and FAULTMASK disable all priorities.

If SB_ARMM_DISABLE_WITH_BASEPRI (barmm.h) is set to 0, PRIMASK will be used to disable interrupts instead of BASEPRI, and then there are no reserved priority levels, so all can be used for smx ISRs.

If SB_ARMM_DISABLE_WITH_BASEPRI (barmm.h) is set to 1, **the highest priority level (lowest value) you should use for your smx ISRs is SB_ARMM_BASEPRI_VALUE** (defined in XBASE\barmm.h and barmm*.inc). (smx ISRs are those that use

smx_ISR_ENTER/smx_ISR_EXIT, so they may run the scheduler upon completion.) Higher levels (lower numbers) are non-maskable and reserved for short non-smx ISRs, since they will run even during critical sections of code where we use sb_INT_DISABLE(). Such an ISR must not invoke an LSR or access any kernel data. Reserved priority level(s) are needed for ISRs that must run with no latency (no jitter) for things such as stepper motor control or collection of data at precise intervals.

Starting with v4.2, use of PRIMASK is the default, because using BASEPRI without the user being aware often led to run-time problems. Often, user's set the priority of an ISR above the threshold, not realizing this made it non-maskable. This caused various kinds of strange behavior which could waste days to resolve. Now the user must knowingly enable the more sophisticated feature.

Nested Vectored Interrupt Controller (NVIC)

The interrupt controller is built into the ARM-M core, unlike traditional ARMs, so it is the same for all processors, even from different vendors. In the BSP, vectors.c and irqtable.c contain the default vectors and configuration table.

Stacks

smx takes advantage of the dual stack model of ARM-M. Prior to smx v4.1, this was the only processor architecture for which smx could have a system stack for ISRs, LSRs, and the scheduler to use. For other processors, ISRs, LSRs, and the scheduler all had to run on the current task's stack, which meant the worst-case overhead had to be added to all task stack sizes.

Because of the way ARM-M was designed, smx can run ISRs, LSRs, and the scheduler using the Main Stack (MSP) and tasks using the Process Stack (PSP) . (There is a process stack for each task.) This way, only the main stack needs to be large enough for maximum interrupt and LSR nesting.

Files

Because ARM-M is significantly different from traditional ARM, most of the porting files are separate and named "armm" not "arm". However, some files are shared, such as the Protosystem files and the top-level preinclude file (e.g. iararm.h). We created a new build directory with extension .AM for ARM-M (e.g. IAR.AM). We keep extensions to three or fewer characters.

ARMM Conditionals

The ARMM conditional is used around code specifically for ARM-M processors. Note that ARM is also defined, so those conditionals apply too. (The compiler defines ARM or arm or similar, so there is no choice whether it is defined.) It is necessary to check ARMM first (before ARM) for sections that are only for ARM-M.

Because ARM-M is significantly different than other processors we have supported, the scheduler porting layer was not sufficient, and it was necessary to add ARMM conditionals in the code. There is not a lot of porting code, but it is more subtle than it might appear at first. If you are studying the code, you need to consider:

1. What stack is being used by the code that is running, and which stack is being modified (MSP or PSP)? Remember that unlike some processors, the return address of a function call is stored in a register, not on the stack.
2. The scheduler runs in an exception (the PendSV handler), which is a significant difference from other processor versions, which run at the task level.

Peripheral Initialization

Cortex-M processors are concerned with minimizing power usage, so power to peripherals is disabled at startup. In order to use a peripheral it is necessary to enable the clock to each peripherals you want to use before accessing any of its registers. Otherwise you will get a Bus Fault. This is true even to access GPIOs. Even GPIO ports have to be enabled. The vendor-supplied BSP code provides a function to do this. For example, for Stellaris processors, use SysCtlPeripheralEnable(). See examples of use in our bspm.c.

Flash Locking

Some processos have the ability to lock the flash for security. Unfortunately this sometimes happens by accident and this prevents you from downloading code to it. See if the vendor provides a flash programming utility. If so, look for an option to erase and unlock it. For example, Texas Instruments provides LM Flash Programmer on its website.

Floating Point (CM4 and CM7 FPU)

The Cortex-M4 and M7 floating point units have the ability to auto save registers on an exception. smx supports this hardware mechanism to save the floating point registers on a task switch. The processor does this if bit ASPEN = 1 in FPU->FPCCR. Unfortunately, this saves only the first half of the registers, s0-s15. If the compiler uses s16-s31, those must be saved in software, using smx hooked task exit/entry routines. APP\DEMO\fpudemo.c demonstrates three methods of saving the registers: software saves s0-s31; hardware saves s0-s15; and hardware saves s0-15 and software saves s16-s31. Currently, IAR EWARM does not have a switch to control which registers are used, so it may not be safe to save only s0-s15.

If you save all registers, you should compare performance of saving all in software or half and half. Note that with the hardware mechanism, once a task uses floating point, it will forever save the registers on a task switch, adding significantly to task switching time. Using the software method of smx hooked exit/entry routines, you can limit this to sections of the code that use floating point by hooking at the start of the section and unhooking at the end.

Lazy stacking is a special feature of the hardware mechanism that only reserves space to store the registers and doesn't actually write them to the stack, until it becomes necessary to do so (i.e. when the interrupting code executes a floating point instruction), thus eliminating unnecessary overhead. However, it appears to have been designed more with ISRs in mind than tasks, and it appears to be difficult to support for multitasking, so smx currently does not support it. A line in startup.c sets bit LSPEN = 0 in FPU-> FPCCR to disable it.

Porting to a New ARM-M or Board

Information is the same as for traditional ARM. See that section.

BSP Files

1. **armdefs.h, armdefs.inc**

   Master include file to include the appropriate BSP header files for the target. armdefs.inc is for assembly files. It has only a small subset of what is in armdefs.h.

2. **bspm.c**

   Implements the BSP API routines documented in the APIs section of this manual. This file is shared by all ARM-M processors and located in the BSP\ARM root directory, because all are so similar. This is unlike BSPs for traditional ARMs which each have their own copy of bsp.c. See the notes below.

3. **bsp.h**

   BSP-specific defines, types, prototypes, and configuration settings. This file is in the BSP directories, but may be shared by several related BSPs.

4. **irqtable.c**

   Contains just irq_table[] which defines the priority and any other properties for all interrupt vectors. In other BSPs (e.g. traditional ARM, ColdFire), this is in each bsp.c, but since bspm.c is shared for all ARM-M processors, this had to be split out. It is one of the few differences for each processor.

5. **lcd.c, lcd.h, oled.c, oled.h**

   Simple API for writing LCDs and OLEDs. Used by lcddemo_task_main() and oleddemo_task_main().

6. **lcddemo.c, oleddemo.c**

   LCD and OLED demos.

7. **led.c, led.h**

   Simple API for writing LEDs. Used by LED_task and LED_lsr in app.c.

8. **startup.c**

   Contains startup code. For IAR, it holds __low_level_init(), which is called by the compiler startup code to do any hardware init. Add any early init code that is necessary for your hardware. Use sb_PeripheralsInit() in bsp.c later init code.

9. **term.c**

   Terminal I/O routines for message output and keyboard input. Interfaces to UART driver.

10. **uart.c, uarti.c**

    Polled and interrupt-driven low-level routines. The latter are used by high-level interrupt-driven UART driver.

11 vectors.c

Exception Vector Table and default handlers. The BSP provides routines for dynamically hooking vectors, but you could statically hook your by modifying this table.

12 AT91\SAM3, EFM32, LM3S, LPC17, STM32, … (subdirectories)

Subdirectories containing BSP files for the indicated family.

We wrote the code that is common for all ARM-M processors, and it does direct register accesses. Code for specific chips mostly calls the chip vendor's library functions. In some BSPs, we brought over only the files we needed. You probably want to use more of the library, so you may want to copy their whole library tree somewhere in your project and change the project to use those files instead of the files in our BSP directory. When doing this or updating to their newer BSP files, search the files in our BSP for "MDI:" tags before you replace them and transfer those changes to the new files.

It is common for the chip vendor's code to assume compilation with a C compiler not a C++ compiler, so you may need to wrap each file with extern "C" { }, unless you change our project to compile for C. This is necessary to avoid name mangling so the linker can resolve references from assembly files. See the BSP files we used, to see how we did this, if necessary.

BSP files are organized by how hardware-specific they are. The more deeply nested in the directory structure, the more hardware-specific they are. From general to specific, directory nesting is: ARM common, vendor processor family, specific processor, specific board. Normally bsp.c is kept in the specific processor directory, but since most of what it handles is common to all ARM-M processors, even from different vendors, it is kept in the most general directory, BSP\ARM and it is named bspm.c, with the "m" to designate ARM-M. Similarly, bsp.h is at a higher level than the board directory. Sharing these files avoids duplicating the code many times, which is error-prone. Doing this requires using some conditionals, though, so it is a balance between duplication of code and simplicity.

BSP API Extensions

BOOLEAN **sb_IRQTableEntryWrite**() — parameters vary

Changes an entry dynamically in irq_table[]. Generally, irq_table should be initialized statically and left alone, but this function is provided in case there is a need to change it while running. After calling this, call sb_IRQConfig() to make the actual change in the interrupt controller. The parameters vary because the fields in irq_table vary depending on the interrupt controller for a particular processor.

This function is primarily provided for assembly access, since it may not be possible to access a C structure in assembly. Even for C, it is better style to call this function rather than to modify the structure directly.

Troubleshooting

1. Problem: Bus Fault when you run your application.

   Cause: You need to enable peripherals before you can use them.

   Solution: For TI Tiva processors, for example, use the Tiva driver library function SysCtlPeripheralEnable(). See examples of use in our BSP code, e.g. bspm.c.

2. Problem: The debugger prompts for the location of source files in the vendor's BSP library.

   Cause: The library was built with debug symbolics enabled, and the path to the files on our system is different than yours. We probably forgot to turn off debug symbolics before we built their library.

   Solution: Rebuild the library on your system, with debug symbolics off so that you can share the project with others in your group without them having the same problem.

3. Problem: Run-time failure related to LSRs or ISRs, or that is difficult to diagnose.

   Cause: You may have hooked an smx ISR (one using smx_ISR_ENTER() and smx_ISR_EXIT() to one of the reserved top priority level(s). These are non-maskable when BASEPRI is used to disable interrupts in the sb_INT_DISABLE() macro. The smx scheduler depends on sb_INT_DISABLE() blocking all smx ISRs.

   Solution: First look at the priorities in irq_table[], in irqtable.c in the BSP, but since your application may configure interrupts elsewhere, you could try changing the smx config setting to use PRIMASK instead, since it masks all interrupts. Setting SB_ARMM_DISABLE_WITH_BASEPRI to 0 in XBASE\barmm.h and barmm*.inc and rebuild the smx library. See the section ARM-M/ Architectural Notes/ ISR Priority Level for more discussion about BASEPRI and PRIMASK.

4. Problem: The debugger is unable to download code to the board anymore.

   Cause: On-chip flash may have become locked accidentally.

   Solution: Look on the vendor's website for a flash programming utility. Install it and look for an option to erase flash and clear the lock. For Tiva processors, use LM Flash Programmer for this. We have had this happen on a couple different Tiva boards.

ColdFire

Architectural Notes

ISRs

See the section ISRs in the Common Notes/ Coding Notes section at the beginning of this manual for general information about writing and hooking ISRs.

On entry to an ISR on a ColdFire, all interrupts are inhibited by the processor for the first instruction of the ISR. This behavior is documented in the ColdFire manuals where they discuss exception processing: “ColdFire processors inhibit sampling for interrupts during the first instruction of all exception handlers. This allows any handler to effectively disable interrupts, if necessary by raising the interrupt mask level contained in the status register.” After the first instruction, interrupts of higher priority than the current one are allowed to nest. In order for the ISR to prevent nested interrupts, the very first statement must disable all interrupts with move #$2700,sr. This is unlike the x86, where interrupts remain disabled until the user re-enables them explicitly.

It is a general rule of smx that the **srnest** global must be incremented before any other ISR is allowed to nest. This ensures that the nested ISRs return to the point of interrupt, and only the outermost ISR will branch to the scheduler. Otherwise, if a nested ISR were allowed to branch to the scheduler it could be a long time before execution returned to the outer ISR(s). Tasks could run in the interim.

Starting in v3.7, srnest is defined as a 32-bit variable, so it is possible to directly increment it in memory in a single instruction and without altering any registers. If this is done as the first statement of the ISR, it is not necessary to disable all interrupts; the interrupt priority set in SR by the processor will pass through. This is what the smx assembly smx_ISR_ENTER macro does.

The C compiler has no knowledge of srnest, so for an interrupt function, it will not generate code in the prolog to increment it. Instead, it must disable all interrupts. In order to write an ISR fully in C, the compiler must provide an *interrupt* keyword or pragma that generates a prolog for the function whose first instruction sets the interrupt priority to 7 in SR (i.e. move.w 0x2700,sr) to disable all interrupts. Alternatively, it must provide a way to define a “naked” function that has no prolog so this can be done in inline assembly. See the ISRs section in the section for your compiler, below for details of what is supported. The simplest scheme is to use the assembly shell approach. Implement the body of the ISR as a normal C function and call it from a simple assembly routine:

```
        .xref       _SomeISR
        .global     _SomeISRShell
_SomeISRShell:
        smx_ISR_ENTER
        jsr     _SomeISR    ; body is C function
        smx_ISR_EXIT
```

Important: The priority level in SR will be different depending on which method you use. For example, if you write your ISR in C and the compiler disables all interrupts in the prolog, the priority level in SR will be 7. When using the assembly shell approach, the priority in SR will be the priority of the ISR (i.e. what the processor set in SR due to the interrupt), which allows higher priority interrupts to run (same priority and lower are blocked). It is probably a good practice for you to explicitly set the desired priority level in SR following the smx_ISR_ENTER macro.

Note that priority level 7 does not block a level 7 interrupt. See the section ISR Priority Level below for more discussion.

ISR Priority Level

The highest priority level you should use for your smx ISRs is 6. (smx ISRs are those that use smx_ISR_ENTER/smx_ISR_EXIT, so they branch to the scheduler upon completion.) **Level 7 is non-maskable and should be used only for short non-smx ISRs**, since they will run even during critical sections of code where we use the sb_INT_DISABLE() macro. We have verified that this is true. Such an ISR must not invoke an LSR or access any kernel data. Priority level 7 would be needed for ISRs that must run with no latency (no jitter) for things such as stepper motor control or collection of data at precise intervals.

Porting to a New ColdFire or Board

If you are using a ColdFire that we do not support, please follow this guide to adapt one of our existing BSPs to your particular ColdFire. Also refer to the Protosystem section, which follows. Only refer to the smx Porting Guide if you are porting to a new compiler or CPU family that is not yet supported by smx. See the section Common Notes/ Porting in this manual for an overview of porting.

1. Build the Protosystem project even if you don't have the board that our BSP targets, to ensure the tools are set up ok. See the appropriate Getting Started section in the SMX Quick Start for directions, if you have not done this already.
2. BSP\CF\<cpu>\<board> contains BSP code, mostly from Freescale. Replace that directory with your own, for your CPU and board. bsp.* and led.* are our files. Create new versions for your board. The main work is bsp.c — it is the implementation of the smx BSP API. Some routines will map onto the BSP code supplied with your board. See the section APIs/ BSP API, or comments in XBASE\bbsp.h if you are unclear about the purpose of a function.
3. CFG directory:
    a. CodeWarrior: Modify the board/CPU prefix files (cwcfhdw.h and cwcfhdw.inc). Add a new case for your board and select it.

b. Diab: Modify the makefile include file (confcf.mki). Add a new case for your board and select it.

4. APP\xx.yy directory:

a. CodeWarrior: Create a new build directory for your board, under APP\CW.CF. Copy the project file, linker command files, and other build files to the new directory. Then modify the project in the IDE to build the new BSP files instead of the files for the BSP we provided.

b. Diab: Create a new build directory for your board, under APP\DC.CF. Copy the linker command files to the new directory. Then modify the makefile to build the new BSP files instead of the files for the BSP we provided.

BSP Files

1. **bsp.c**

   Implements the BSP API routines documented in the APIs section of this manual.

2. **bsp.h**

   BSP-specific defines, types, prototypes, and configuration settings.

3. **bsp.inc**

   Contains configuration settings for timers and UARTs and any other BSP settings needed by assembly files.

4. **isrs.s**

   Assembly ISRs and ISR shells.

5. **led.c, led.h**

   Simple API for writing LEDs. Used by LED_task and LED_lsr in app.c.

6. **m5xxxevb.h** or **init.h**

   Contains memory map (address) constants. These must agree with the linker command file (.lcf). Also contains function prototypes, etc.

7. **mcf5xxx.s, mcf5xxx.h**

   Misc low-level ColdFire routines and definitions.

8. **mcf5xxx_lo.s**

   Assembly startup code. Entry point is asm_startmeup. Initializes some ColdFire registers and then calls mcf5xxx_init() in sysinit.c to do main hardware init. Then branches to __start in the CodeWarrior library. This ultimately leads to main().

9. **mcf5282.h, mcf5282.inc** (and similar)

   I/O Register Map (for on-chip peripherals, etc). Includes macros for reading and writing them. The .h file is provided by Freescale; the .inc file is a small subset of defines we created for assembly code. Add to it as necessary.

10 **mcfdefs.h, mcfdefs.inc**

Selects the proper include file for the target processor, which specifies i/o port addresses, etc. Other processor-specific settings are defined here.

11 **sysinit.c**

Main hardware initialization code. Initializes chip selects, SDRAM, and some peripherals. We commented out initialization of timers and UARTs, since we provide our own drivers that initialize them.

12 **term.c**

Terminal I/O routines for message output and keyboard input. Interfaces to UART driver.

13 **timer0.s, timer1.s**

Control functions for timers 0 and 1. Timer 0 is used for the smx tick; Timer 1 can be used for any purpose. bsp.inc contains configuration settings for the timers.

14 **uartn*.s**

Two or more UART drivers are provided that are identical except for the UART number in names in the code. n indicates the UART number. * is either "i" for interrupt-driven or "p" for polled. bsp.inc contains configuration settings for the UART drivers, such as baud rate.

15 **vectors.s** and **int_handlers.c**

vectors.s statically defines the Exception Vector Table. Entries point to simple default ISRs in int_handlers.c.

16 **5206E**, **5225x**, **523x**, **525x**, **527x**, **5282**, **532x**, **5441x**, **5445x,** etc. (subdirectories)

CPU directories. Contains CPU and board support code for your target. One CPU directory is provided in your order. Each has subdirectories for all boards we support for this CPU. See CPU Files below.

17 **M523xEVB, M5251C3, M5282EVB, TWR-MCF5441x,** etc.

Board directories. Contain board support code. Each CPU directory (e.g. 5282) contains board directories for each board we support for that CPU. See Board Files below.

BSP API Extensions

BOOLEAN **sb_IRQTableEntryWrite**() — parameters vary

Changes an entry dynamically in irq_table[]. Generally, irq_table should be initialized statically and left alone, but this function is provided in case there is a need to change it while running. After calling this, call sb_IRQConfig() to make the change in the interrupt controller. The parameters vary because the fields in irq_table vary depending on the interrupt controller for a particular processor.

This function is primarily provided for assembly access, since it may not be possible to access a C structure in assembly. Even for C, it is better style to call this function rather than modifying the structure directly.

BOOLEAN **sb_IRQ2TableEntryWrite** () — parameters vary

Does the same as the first function, in the case where there are 2 different interrupt controllers (requiring a second irq_table of a different format), such as on the 5249.

CodeWarrior (CW.CF)

Last updated for CodeWarrior v7.2.

Version

Use the version of CodeWarrior indicated by the readme.txt file in the root of your release or by the suffix of the project files. For example, cw72 in the name App_m5441xtwr_cw72.mcp means v7.2. We may have provided project files for multiple versions, in which case you have a choice. The suffix is necessary because changes in the IDE from version to vesion require it to convert the project files, but once converted the changes cannot be reversed. It is often possible to use a newer version, but there could be build problems, so save a copy of the project files in case you need to revert.

CodeWarrior v6.4 is not much different from v6.3, and v7.0 was released shortly after, so we did not create project files for v6.4. The main change we see is they removed the EC++ libraries. To use v6.4:

1. Start with our v6.3 (_cw63) project files and let the IDE convert them.
2. Delete the EC++ library (EC++_4i_CF_MSL.a) from the Protosystem project file(s). Add C++_4i_CF_MSL.a if you are using C++ and the project doesn't link without it. It is in <CW>\E68K_Support\msl\MSL_C++\MSL_E68k\Lib.

CodeWarrior v6 is not much different from CodeWarrior v5. Other than the prefix files improvement discussed below, it mostly just adds support for new ColdFires.

CodeWarrior v7.0 eliminates the MetroTRK libraries. Instead add ColdFire_Support\msl\MSL_C\MSL_ColdFire\srcs\console_io_cf.c to your project if you want console i/o to go to a debugger window rather than the actual UART.

CodeWarrior v7.1 fixes some problems in v7.0. For example, there was a codegen problem related to the volatile keyword. Also, the v7.0 debugger does not display most local variables, which is a huge problem for debugging. (Turning off register coloring in the project options seems to work around this.) Note that the IDE is the same version, just a higher build number, and it doesn't convert the project file, so it is possible to go back to v7.0 if you were to have some problem with v7.1. This was true in our testing, at least.

CodeWarrior v7.2 is a big change. Freescale said they rewrote the compiler to make the code it generates more efficient for microcontrollers. This is the same compiler used in the newer v10 tools, which are Eclipse-based. (We currently do not support v10.) The biggest change that affected SMX was in the calling convention. The new compiler supports only register passing of parameters. The old compiler also supported passing parameters on the stack, and that is the only model we supported. See the section Calling Convention below for more information.

Another change was they left the “extras” files out of the library, it seems deliberately. We had been using their ultoa(). If you need the extras functions, you should modify EWL .mak file or project file to include it and rebuild the EWL library.

Although Freescale discontinued development on the v7 tools after they released v10, they added support for the new MCF5441x in the v7.2.2 patch.

Even v7.2 seems to have a lot of trouble with displaying local variables, which makes debugging very difficult. Possibly experimenting with code generation and optimization settings may improve it.

Build Targets

The project files have the standard 3 build targets (Debug, Release, and ROM). Note that the ROM target is linked with the _ROM or _ROM_CodeToRAM version of the linker command file (.lcf). The CodeToRAM version copies code from ROM to RAM to execute at higher speed out of RAM. See the section ROM Target and Copying Code to RAM, below.

Preinclude Files (Prefix Files)

Starting with CodeWarrior v6, there is now a dialog where you can enter project-wide defines for C/C++ files. However we still use “prefix files” for most defines. These are header files that are included ahead of line 1 of every file in the project. Now the prefix files are specified using #include in the same dialog that allows #defines. Different defines and files can be specified for each build target. One advantage of prefix files vs. adding defines directly into an IDE is that changes need to be made only to 1 prefix file and all projects (library and application) use the same setting without having to change it in every project. smx uses prefix files to define preprocessor symbols that specify the CPU and board details and which libraries and demos to link.

smx prefix files are stored in the CFG directory. They are of the form **cwcf*.h** and **cwcf*.inc**. For C/C++ files, each target in each project file #defines a few symbols related to the build target (Debug, Release, ROM) and #includes the proper prefix files. Assembly files still only support prefix files, so for them, each build target specifies one of the top-level prefix files, which defines a few symbols and then includes one or more of the main prefix files. Configuration is done only to the main prefix files, not to the shell files. The following is a summary:

Main Prefix Files (configuration is done to these)

cwcf.h, .inc	specifies global defines and pragmas
cwcfhdw.h, .inc	specifies board and processor
cwcflib.h, .inc	specifies which smx module libraries to link
cwcfdemo.h	specifies which demos to link to Protosystem

Top-Level (Shell) Prefix Files (only the .inc files are needed for CWCF v6 and later)

cwcfdbg.h, .inc	Debug target for Libraries
cwcfdbg.h	Debug target for Protosystem
cwcfrel.h, .inc	Release target for Libraries
cwcfrel.h	Release target for Protosystem
cwcfrom.h, .inc	ROM target for Libraries
cwcfrom.h	ROM target for Protosystem

Note: Only the **.inc** top-level prefix files are necessary starting with CWCF v6 because it adds a Prefix Text box in the C/C++ Preprocessor settings, so the contents of the top-level prefix files was moved there. This could not be done for the .inc files since they did not implement it for the assembler settings.

Note: Another set of .inc files suffixed “72” was created for CWCF v7.2 and later. The purpose is to define CC_REGISTER_PARS as 1, since it cannot be automatically defined as in C.

Originally, we had only 3 “shell” prefix files (_dbg, _rel, _rom) but since these included cwcfdemo.h, any time the user enabled a different demo in cwcfdemo.h, every library project (e.g. smx, smxNS) would show all files being out of date and needing to be recompiled, even though the libraries don’t use the demo settings. To avoid including the demo defines for library builds, more of these “shell” prefix files had to be created, so that library projects don’t use cwcfdemo.h. Comparing the 3 files in each group will make it clear what we have done doing.

Startup Sequence

_asm_startmeup -> __start -> main() -> …

_asm_startmeup is the BSP startup code. It does basic hardware init (e.g. PLL, SDRAM, chip selects, cache, etc.), and when using _ROM_CodeToRAM.lcf, it copies the application code from ROM to RAM.

__start is the routine in the CodeWarrior runtime library that clears BSS, copies initialized data from ROM to RAM, runs C++ initializers, etc and then branches to main().

Following main() is the standard sequence shown in the SMX Quick Start, in section SMX Startup and Scheduler Operation.

Startup Code

Freescale provides startup code for their evaluation boards on their web site, and CodeWarrior incorporates this code into its project stationery. We start with either the CodeWarrior stationery or the original Freescale BSP files.

We link and run this startup code for **all targets**, not just the ROM target, to ensure that the executable being debugged is as close as possible to what will run in ROM. This was not true for older CodeWarrior stationery projects. This is simpler for you, too, since you don't have to remember two different ways of running — the entry point is the same, and the same EVT is used (the one in vectors.s, not what is left by dBUG).

To debug the startup code, open the project settings and select the Debugger Settings panel. Select to stop at Program entry point. Do not specify the name of the assembly routine (e.g. _asm_startmeup) because for some reason CodeWarrior names the public assembly routines @DummyFn1, 2, etc. and it won't stop. Alternatively, you can set a breakpoint in it. (Just open the assembly startup file (e.g. mcf5282_lo.s) and click in the left margin next to the instruction you wish to break on. The entry point is usually _asm_startmeup.

ISRs

See the section ISRs at the beginning of the ColdFire section and in the Common Notes section at the beginning of this manual for general information about writing and hooking ISRs.

For CodeWarrior, write ISRs using the assembly shell technique. The body can be written in C. See below and the section ColdFire/ Architectural Notes/ ISRs above for more discussion.

It is not possible to write the whole ISR in C, for smx. CodeWarrior's *interrupt* pragma/keyword does not disable interrupts in the prolog. (If they ever change this, it will also be necessary that it does this in the first instruction.) CodeWarrior's asm{naked} directive doesn't work either since it actually generates a prolog and epilog, unless that has been changed in newer releases. Because of these limitations of CodeWarrior, you must use the assembly shell approach.

Calling Convention: Register Parameters

Prior to v7.2, CodeWarrior supported passing parameters on the stack or in registers. All smx project files were set for stack passing, and register passing was not supported. In order to support v7.2, we had to modify all assembly code to pass or expect parameters in registers. We added the CC_REGISTER_PARS define in the prefix files to select this. In the C header it can be set automatically, but for assembly, it has to be set by the user, so we created an alternate set of assembly preinclude files suffixed with "72" for use by v7.2 and later.

One complexity of register parameters on ColdFire is for **smx tasks and LSRs that take a parameter**, due to the fact that ColdFire has separate address and data registers. smx defines these parameters as integers, so when these functions are called by the scheduler, the compiler passes the parameters in data registers. This requires defining the parameter in the function prototype as an integer type (e.g. u32) regardless of its true type. If it is really a pointer, either typecast all uses of it in the task/LSR, or create a local variable of the proper pointer type and assign the parameter to it with the pointer typecast and use that variable in the code. Otherwise, if the function is defined to have a pointer parameter, the code generated by the compiler in the function will expect its value to be in an address register, but it was actually passed in a data register. **Note that the examples in the smx documentation do not reflect this.** They show tasks and LSRs with handles (e.g. MCB_PTR) as parameters. ColdFire users need to know to define the parameter as a u32 and then typecast it in the function (e.g. to MCB_PTR). A quick survey of popular embedded processors shows it is uncommon to have separate address and data registers, so we did not want to complicate our examples to show this, when it applies only to a minority of users.

It is also possible to use register parameters for pre-v7.2 CodeWarrior, but it is necessary to change the App project to use the RegABI version of the CodeWarrior libraries and to use our assembly prefix files suffixed “72”.

Console I/O

Functions such as puts() can be directed either to the UART or to a console window in the debugger via the BDM by linking CodeWarrior’s console_io_cf.c file (v7) or the MetroTRK C library (pre-v7). This can be done differently for each build target. For example, you may wish to output to the console window in the debugger for the Debug target and to the UART for the release and ROM targets. The following is an explanation of how this works, that is hopefully clearer than the CodeWarrior documentation.

v7.2: The discussion below applies, but the MetroTRK libraries have been eliminated, so instead add ColdFire_Support\ewl\EWL_C\src\coldfire\console_io_cf.c to your project if you want console i/o to go to a debugger window rather than the actual UART.

v7.1/0: The discussion below applies, but the MetroTRK libraries have been eliminated, so instead add ColdFire_Support\msl\MSL_C\MSL_ColdFire\srcs\console_io_cf.c to your project if you want console i/o to go to a debugger window rather than the actual UART.

C_4i_CF_MSL.a is the standard C library. When it is linked, puts() and similar functions call WriteUARTN() in uartmw.c, which calls out_char() in io.c. io.c interfaces to our UART driver. (These files are in the smx BSP for each ColdFire board.)

C_**TRK**_4i_CF_MSL.a is a special version of the library that directs puts() to the console window in the debugger, via the BDM.

Just click the dot on or off in the target column of the project for these 2 libraries to select one library and un-select the other.

The CodeWarrior Aux libraries are polled UART drivers. We do not link those. Instead, uartmw.c and io.c map onto our interrupt-driven UART driver, as explained above.

Notes:

1. Be sure to end strings with \n\r. Otherwise, the console window in the debugger will not show the message until you print another one that does use \n\r.
2. When output is directed to our UART driver, don’t expect to see messages appear on a terminal immediately. Because the driver is interrupt-driven, the main-line code does not wait until the message goes out the UART before continuing on. Also, the UART is initialized and the terminal is cleared in ainit() (main.c), so messages won't appear until ainit() runs (or at least until sb_PeripheralsInit() and sb_ConsoleOutInit() run).

Linker Command Files (.lcf)

We created our linker files based on the examples provided by Freescale.

In smx v4.1, we changed the .lcf files to locate SDAR and ADAR (.smx_sdar and .smx_adar sections, respectively). When possible, we locate SDAR and the System Stack to internal SRAM for performance, and we locate ADAR to external SDRAM to separate it from SDAR. SDAR holds smx objects and queues; ADAR is intended for application use. In cases where all the SRAM is reserved for SMX middleware products (e.g. smxNS and smxUSB), we could not locate

SDAR to SRAM. In your case, space may be available due to not using one or more of those products, or you may have additional memory areas. You should change the .lcf file to achieve the best location for these for your system.

Note that we typically located the System Stack after some other data, such as SDAR, so it would not be at the start of a region of memory. A stack overflow into non-existent memory is likely to cause a processor fault, which would halt the system, while overflow into other data may not be catastrophic, especially if there is unused space at the end, as is likely to be the case with SDAR.

ROM Target and Copying Code to RAM

It is faster to run from RAM typically, so if speed is an issue and you have plenty of RAM you may want to use this approach to copy your code from ROM to RAM.

If you do not have the ROM_CodeToRAM version of the .lcf file, please contact us. It is necessary to use this .lcf file for the ROM target and to make a few changes to the startup code. We have only created this file and made these changes for some of our ColdFire BSPs, so we can make them for the one you are using if not already done. If your BSP includes this .lcf file, then the changes have already been made to your startup code, so it should just be a matter of deleting the ROM.lcf file from the project and adding the ROM_CodeToRam.lcf file and relinking.

A few notes about how it works: The hardware startup code (mcf5xxx_lo.s, sysinit.c/hwinit.c, and possibly another file or two) are located for ROM and the rest of the application is located for RAM (but stored in ROM). The ROM part must not call any functions in the RAM part until after it does the copy to RAM since otherwise, it would call whatever garbage is there causing an invalid instruction fault. (fault_handler at the end of mcf5xxx_lo.s will catch this in the debugger.)

For a ColdFire to boot properly, the value at offsets +0 and +4 in the ROM must have the initial stack pointer and the entry point, respectively. Also, in order to catch faults, these must be followed by at least the fault vectors of the EVT. To achieve this, mcf5xxx_lo.s is preceded by a mini EVT. This is only used during the startup code. Once we switch to RAM, the real EVT in vectors.s (in RAM) is used.

The code is copied from ROM to RAM near the end of the hardware startup code. The data is copied to RAM a little later, by _start in the CodeWarrior startup code that is called last by the hardware startup code. This is done the same as when using the normal ROM.lcf file.

The startup code works fine in all cases without the need for any conditionals, but you have to be careful about changes you make. For example, we call the routines to set the VBR, CACR, and ACRs after the code is copied to RAM because those functions are in RAM. If you don’t want to copy code to RAM you can delete the mini EVT and the copy loop.

A6 Stack Frames and Call Stack Display

In the scheduler porting macro for switching to a new stack for a task being started (not resumed), we clear a6 to terminate the chain of frame pointers, for the debugger to limit how far back it goes when displaying the call stack. We don’t know what algorithm the debugger uses for the call stack display, but we found that doing this does help. In old versions of CodeWarrior, before it supported .mem files to mark areas reserved, the debugger sometimes would go back too far and somehow end up in non-existent memory, which would cause a bus fault. This happened after running to a breakpoint or stepping sometimes, and starting each task with a cleared frame pointer

solved it. We find it also helps to enable the option A6 Stack Frames in the IDE for at least the smx library. If you have problems with the call stack window, try enabling it for all libraries (all build targets for each library).

Note that you can turn off A6 Stack Frames for improved efficiency, but the call stack window may not work as well. The .mem file will prevent crashing the debugger with a bus fault if a non-existent location is accessed. In this case, you can remove the line that clears a6 in the porting macro in XSMX\xcf.h. Keeping it probably won't cause a problem, but it would be unnecessary.

Using CodeWarrior

1. The Protosystem project files are located in the board directories under APP\CW.CF. For example, the project for M5282EVB is here:

   ```
   APP\CW.CF\M5282EVB\App_m5282evb_cw70.mcp
   ```

   Open the project file for the eval board you are using, do a make, and then press the green arrow button to download it to the board and debug. It should run as shipped. If not, contact us for help.

2. The Protosystem project file for each Freescale eval board was created starting with the CodeWarrior stationery for it or the Freescale BSP code. Modifications were made, and our Protosystem files were added to it. The project is based on the C++ stationery, since you may use C++ modules such as smx++ and PEG.

3. Files are added to a project with Project | Add Files or by right-clicking on a node and selecting Add Files…

4. The debugger configuration file (e.g. CF_M5282EVB_PnE.cfg) that the project targets point to is a copy of the file supplied with CodeWarrior that is in the same directory as the project (.mcp) file, executable (.elf), etc. For example, for the M5282EVB board, it is in APP\CW.CF\M5282EVB. This file is intended to initialize just the bare minimum to get debugging started. All hardware init should be done in the Freescale startup files in the project to ensure it is done when running free-standing from ROM too.

5. Manual Operations: Some operations must be done outside the IDE because CodeWarrior doesn't seem to have a way to run an external tool such as the smxNS NSBLDPG utility.

   a. Web Pages (smxNS Web Server): Run NSBLDPG.EXE from the command line prior to building the application after any web pages are changed or added. To do this, change to the directory that contains the web page source files and invoke nsbldpg, as in the following example:

      ```
      C:\SMX\APP\DEMO\WEBPAGE>..\..\..\bin\nsbldpg buildpg.cfg
      ```

   b. Web Pages (smxNet Web Server): Run REALTIME.EXE and BINTOC.EXE for any changed or new web pages in the WEBPAGES directory. The batch file do1htm.bat is convenient for doing this. (You must also add new pages to NETAPPS\virttbl.c and rebuild the application.) You can use MAKEWEB.BAT in WEBPAGES to build all of the pages (add any new ones to it).

6. File Organization: The organization of file nodes in an IDE project has no relation to their location on disk. This gives you the flexibility to add new groups and drag files into them in

the IDE without any worry about what directories they are in on the disk. (However, the IDE must be given a list of all paths that hold files in the project, on the Access Paths page.)

If you do want to move a file on disk, the IDE will not be able to find it. If the file is moved into one of the paths specified in the Access Paths, try Project | Reset Project Entry Paths, then Project | Re-search for Files. If this doesn't fix it, delete the node in the IDE and add it again. Note that newly added files are always put at the end of the link order list, so if its position in the link list matters, remember to go to the Link Order tab for each build target and drag it to the proper position.

7. File Names: Capitalization of file names in the project window has no relevance. For some reason, when using CodeWarrior on different versions of Windows, files get different cases when added to the project.

8. Linker: Our application projects for CodeWarrior may **suppress all linker warning messages** (not errors). Older versions of CodeWarrior did not have a way to disable warnings about application functions that replace those in a library, such as our heap functions replacing the C library heap functions. We changed the project files, but may have missed some.

Debugging with CodeWarrior

1. Path to .cfg file: The debugger runs a target configuration file to do some hardware initialization before it downloads code to the board. Originally, these come with CodeWarrior, but we supply them with smx with any changes we had to make. Each is stored in the same directory as the project file for the board. The project file points to the particular .cfg file to use, on the CF Debugger Settings page of the project settings.

2. Starting with v6.3, the debugger uses a .mem file for each board to determine maximum size of reads and writes to areas of memory and peripherals. If no .mem file is used, size defaults to 1, resulting in a slow debug download to the target. The .mem files supplied by CodeWarrior v6.3 and v6.4 specify 1 for ReadWrite size for SDRAM and SRAM, so we changed them to 4 to avoid slow downloads. The .mem file is specified in the project settings, on the same page where the debug .cfg file is specified (Debugger | CF Debugger Settings, in the left pane). We supply each .cfg file, with any necessary modifications, in the same Protosystem directory as the project file for the board.

   Starting with v7.0, they fixed them to set the size to 4 for the RAM areas (undoubtedly a common support problem). They also deleted all the lines that specified the peripheral register areas, so the files are much shorter. They moved that information into internal files somewhere.

3. By default, the debugger runs through the startup code automatically and stops at main(). If you want to debug the assembly startup code, open the project settings and select Debugger | Debugger Settings in the left pane. In the right pane, select "Program entry point". If this doesn't work, put a breakpoint in the startup code before starting a debug session. See the section ColdFire/ CodeWarrior/ Startup Code for more information about it.

4. smxAware, included with smx, is a DLL that plugs into CodeWarrior to display smx objects. It includes graphical display features to show event timelines and stack usages. See the smxAware User's Guide for full information.

5. In v5 through v6.2, the CodeWarrior debugger has an annoying tendency to stop at Timer0_ISR even though no breakpoint is set there. See Troubleshooting note #5 below for the solution.

6. Starting with v7, Run to Cursor is now a task-specific breakpoint, if smxAware is present. If you click in a function called by a different task and then do Run to Cursor, execution will pause there briefly and resume since it will check the thread ID and see it is not that of the current task at the time you clicked it. This is unlike a normal breakpoint, which is not task-specific. We have suggested to Freescale that they offer two versions of Run to Cursor or add a setting in Preferences to change this.

Debugging in Flash / ROM

Briefly, the technique is to build the application in CodeWarrior, as normal (set to generate an S-Record file), switch to your flash programmer and program it into flash, and then click the Debug button as normal.

The following are more discussion and tips about this.

1. The ROM target is already set to use the _ROM.lcf file. You can change the optimization level setting to make it easier to debug (assuming the problem happens in unoptimized code too).

2. CodeWarrior project settings:

   a. ColdFire Linker settings panel: Check the box Generate an S-Record File.

   b. Debugger Settings panel: Select to stop at Program entry point of you want to debug the startup code. Do not specify the name of the assembly routine (e.g. _asm_startmeup) because for some reason CodeWarrior names the public assembly routines @DummyFn1, 2, etc. and it won't stop. If you want to let it run to main(), select User specified and enter main on the text line.

   c. CF Debugger Settings panel: Uncheck all checkboxes in the Program Download Options (so it does not download anything to the board). Alternatively, you can leave it set to download the code and also check the Verify Memory Writes box to verify the latest image is there, but this operation takes a lot of time every time you press the Debug button. Having to remember to reprogram the flash after every rebuild is error-prone, and you are likely to find yourself debugging old code more than once. We have recommended to Freescale that they add a checkbox to allow doing a verify-only when debugging in ROM, so that this will be checked every time you press the debug button.

   d. CF Debugger Settings panel: You can uncheck the box Use Target Initialization File because the code is already in ROM. (The reason for using a Target Initialization file is to do just enough board init to be able to download code into RAM.)

3. Program the .s19 file into flash using your flash programmer. We recommend using CFFlasher for Freescale eval boards since it is so easy to use. See sections about various flash programmers in ColdFire/ Tools for more information.

   Tip: Leave your flash programmer running so it is quick to reprogram the image each time you rebuild.

Tip: CFFlasher automatically erases each block before programming the next block of your application, so it is unnecessary for you to do an erase operation first.

4. Press the Debug button (the green arrow with the bug), as when debugging in RAM. If you left the download and verify options checked, you will see the progress bar, and the debugger will seemingly download the program to the board, but it does not actually overwrite flash. The program is already there from the previous step. Turning on verification makes the download take much longer, so you ought to invest in a higher-speed BDM connection such as the P&E Lightning card, which is 10 times as fast as the parallel port.

5. The debugger should stop at whatever point you have it set to stop (e.g. main()).

   Tip: At least the first time, set it to start at the program entry point and then set the code pane to Mixed (source/disassembly) temporarily so you can ensure the correct code really is there in flash. Ensure that the first few disassembly instructions match the startup source code.

6. You can set breakpoints as usual. However, since they must be hardware breakpoints, you are limited in how many you can set. Different ColdFires support a different number. For example, the 5282 supports only 1, but the 5213 supports 4 (which is a good thing since any reasonable-sized app must be debugged in ROM, on it, due to having such minimal RAM). Typically the ColdFire manual tells you how many PC breakpoints can be set, in the bulleted features summary in the Overview chapter. CodeWarrior will not allow you to set more breakpoints than the number supported by the chip. It reports: “The maximum number of Hardware Breakpoints has been exceeded.” Note that stepping or running to cursor requires a breakpoint, so you will get an error if you try to do either operation when you already have the maximum number of breakpoints set.

Tips

1. CodeWarrior project files accumulate junk because it does not fully remove files you delete from them. To clean a project file, export it to XML and then re-import. These are options on the File menu.

Troubleshooting

1. Problem: Trouble getting the debugger to connect to the target.

   Cause: Maybe the PC BIOS is set for the wrong parallel port mode. Or, if you are using Windows XP, you need to make some changes to the registry to disable autoscanning that XP does on the parallel port.

   Solution: We provide a .reg file to modify the Windows XP registry, in MISC\WINXP\PEMICRO. Also see ColdFire/ Tools/ P&E Wiggler below for more information about both cases.

2. Problem: App doesn't run from flash.

   Cause: Possibly the board is not booting because of the BDM wiggler.

   Solution: Disconnect the BDM wiggler from the board (with power off) and try again. On some boards it is only necessary to unplug the parallel cable from the wiggler.

3. Problem: "Unknown link error".

   Cause: Might be that you have the .xMap file open in your editor.

   Solution: Ensure the .xMap file is not open in an editor since this prevents the linker from overwriting it, causing the link to fail.

4. Problem: "Bus Error" in debugger or bad task return value (if using the feature of smx in which the task can return a value to itself for the next run via its parameter).

   Cause: If using CodeWarrior ColdFire v4 or higher, this may be due to building the smx kernel library at optimization level 0. Starting with v4, CWCF changed level 0 to save data on the stack using the frame pointer (a6) and this causes trouble in a section of the scheduler where we clear a6. This is explained in more detail in the readme.txt file in XSMX\CW.CF.

   Solution: Rebuild the smx kernel library at optimization level 1 or higher.

5. Problem: Debugger stops on Timer0_ISR even though there is no breakpoint set there.

   Cause: This is a problem for the CodeWarrior v5 to v6.2 debugger (fixed in v6.3, but seems to still be a problem for 548x/7x). Our understanding is that they made some change to the debugger that caused this problem to occur, and they will have to make some substantial changes to handle the problem. So, for now, they have added a new setting panel to allow you to work around this.

   Solution: In the project settings, select "CF Interrupt Panel" in the left pane. In the right pane, check the checkbox and set the level to 7. For a warning and more information about this setting, search for "CF Interrupt Panel" in the CodeWarrior Targeting ColdFire manual. This is a new panel for v5.

6. Problem: Debug download is very slow for CWCF v6.3 and later.

   Cause: The size specified in the .mem file for ReadWrite size for the SDRAM or SRAM is set to 1 but should be 4.

   Solution: Change the setting to 4. See the tip about the .mem file in the section Debugging with CodeWarrior, above.

Diab (DC.CF)

Last updated for Diab v5.2.1.0.

Version

Use the version indicated by the readme.txt file in the root of your release or in XSMX\smxid.txt. It is often possible to use a newer version.

Build Targets

The standard build targets are supported (Debug and Release), as discussed in the Common Notes section at the beginning of this manual.

Switches Used in Makefiles

1. Compiler

-c	compile only, don't link
-W :c++:.c	compile .c modules as C++
-Xnested-interrupts-off	disable ints on entry to an ISR
-@E+errs	list errors to file errs

2. Assembler

-o	output file name
-@E+errs	list errors to file errs

3. Linker

-lc	search for library libc.a
-ld	search for library libd.a (for C++)
-m2	generate link map, verbosity level 2
-@E+errs	list errors to file errs
-@O=$(proto).map	name of map file

Startup Sequence

_asm_startmeup -> __init_main() -> main() -> …

_asm_startmeup is the BSP startup code. It does basic hardware init (e.g. PLL, SDRAM, chip selects, cache, etc.).

__init_main() is the routine in the Diab runtime library that initializes data, clears BSS, copies initialized data from ROM to RAM, runs C++ initializers, etc. and then branches to main(). Source for this function can be reviewed at Diab\5.2.1.0\src\init.c. Once main() is called, execution continues with the standard sequence shown in the SMX Quick Start, in section SMX Startup and Scheduler Operation.

Startup Code

Freescale provides startup code for their evaluation boards on their web site. This code is provided in your release with some modifications for SMX and Diab.

ISRs

See the section ISRs at the beginning of the ColdFire section and in the Common Notes section at the beginning of this manual for general information about writing and hooking ISRs.

For Diab, you can write ISRs fully in C or using the assembly shell technique. See the section ColdFire/ Architectural Notes/ ISRs above for more discussion.

To write the ISR fully in C, use the switch -Xnested-interrupts-off to ensure the compiler disables all interrupts by generating move #$2700,sr as the first line of the prolog of the function. The function should start/end with the smx_ISR_ENTER()/smx_ISR_EXIT() macros, as is standard for smx ISRs. If you want enable nested interrupts, you should set the priority level in SR to the desired value after smx_ISR_ENTER().

An advantage of the assembly shell technique is that the interrupt level of the ISR that the processor set in SR will be preserved. When written in C, the ISR disables all interrupts.

ROM Target

The Diab ddump utility can be used to generate an S-record file from an object file. When run with a minimum of switches, this utility generates S-records for memory in RAM as well as ROM, which will cause some flash programmers, specifically CFflasher, to complain of verification mismatches.

The +dn switch, where n is a number such as 3, can be used to limit the type of S-records that are generated, but then you run the risk of not generating a needed section if the code is reworked in a way that generates another section. As released, the +d3 switch should generate the needed ROM S-records, but this isn't used.

Another option is to explicitly specify all the output sections that reside in ROM on the ddump command line.

Using Diab

1. Diab provides the -Xlint switch to generate a thorough set of warnings of suspicious or non-portable code. We reviewed these warnings, but the makefile we ship does not specify this switch.
2. If you would like to review the assembly language generated by the compiler, you can use the switches -Xkeep-assembly-file -Xpass-source to produce an assembly language file.

Tools

P&E Wiggler

A P&E Wiggler was supplied with older Freescale evaluation boards. It connects to the PC via parallel port. This is the connection type most of our CodeWarrior project files use by default.

If you have any trouble getting the debugger to connect, experiment with different parallel port modes in the BIOS. CodeWarrior documentation recommends setting the BIOS to bidirectional or PS/2 mode, but EPP seems to work fine for us.

If you are using **Windows XP**, some changes need to be made to the Registry to disable undocumented autodetect scanning it does on the parallel port. Run the **winxp2.reg** file provided in SMX\MISC\WINXP\PEMICRO. It was created by P&E Micro.

P&E Multilink

As laptops shed their parallel ports in favor of USB, this new USB type wiggler was introduced, and Freescale started supplying it with their evaluation boards. This unit is a little faster than the parallel wiggler, but we found that interchangeability among our different PCs was not good.

P&E Lightning

Lightning is a PCI card that plugs into your PC to dramatically increase the communication speed with the target. You continue to use the same wiggler and parallel cable, but it is connected to the Lightning card instead of the PC's parallel port. We highly recommend the Lightning card. It really is about 10 times faster than using the host's parallel port. This makes a big difference for downloading your application to the target, and it is especially important for using the smxAware graphical features, since it greatly shortens the time to download the smx event buffer from the target.

Plug the parallel cable into the DB25 connector on the board, and select it in the project settings in CodeWarrior: In the left pane, select Remote Debugging. In the right pane, select P&E Lightning from the Connection drop-down list. If you have trouble connecting with the debugger, try changing the Speed setting. Start with 20. On the same settings page, click Edit Connection to get to the Speed setting.

CF Flasher

CF Flasher is a free flash programming utility provided by Freescale for use with their eval boards (only). It has a simple interface, and it is very easy to use because it is pre-configured for each eval board. It supports S-Record and Binary file formats. Use this for programming the Protosystem ROM target onto your eval board. The latest version typically supports all of the Freescale ColdFire eval boards that are available. Sometimes, though, they have supplied a new .cfg file that could be added to support a new board until the next revision of the tool. Failing that, use another flash programmer, such as the one built into CodeWarrior or the P&E flash programmer. These are discussed in the following sections.

CFFlasher is available from the Freescale web site. Go to www.freescale.com/coldfire and enter "cfflasher" on the Keyword search line. It should come up first in the search results.

Warning: Flashing your program will overwrite dBUG or any firmware on the board. We recommend that you save its image to a file before you program flash the first time:

Saving dBUG

Press the Upload Flash button and specify the starting and ending address. The handling of the ending address changed from v2 to v3. For example, if the image ends at 0xFFE1FFFF, specify 0xFFE1FFFC for v3 or 0xFFE20000 for v1 or v2. For v1 and v2 it is important to get the last byte or else CF Flasher will have trouble programming it back later. The section for the Flash ROM in the board manual should indicate how big dBUG is, but double check it — use the Memory Window button to view memory at various places to see where it seems to end. The dBUG image for each eval board is probably available from the Freescale web site, but saving it is faster than finding it there, and this way you are ensured to have the same rev of it.

Flashing Your App

1. With the BDM cable connected and the board on, run CF Flasher.
2. Press the Target Configuration button and select the board. Press Ok.
3. Press the Program Flash button. Press the Filename button and browse to the S-Record file (e.g. AppRom.elf.S19). CFFlasher processes the file and determines the number of blocks (different regions of memory). If you have any problems, verify that the blocks are all in the address range for the flash and that they don't overlap.
4. Press the Program button. A bar graph shows progress. When done, the message area should say:

   Ready!
   0xXXXXXXXX Bytes Written
   0xXXXXXXXX Bytes Verified (if Verify checkbox was checked)

5. Power-off the board, **disconnect the wiggler from the board**, and power on. You should see it running on the terminal.

See the readme file included with CF Flasher for information about its other features.

CodeWarrior Flash Programmer v5 and Later

The flash programmer built into CodeWarrior has been rewritten in version 5.0 and is much better than the one in version 4. It works quite well, and you should try it before purchasing a separate tool for this. See Flashing Your App below, for directions.

Flashing Your App

There is nothing unusual about using the flash programmer with SMX, but here are some directions to get you started quickly:

1. Connect to the BDM, as when you are debugging, turn on the board, and start CodeWarrior.
2. Open the project you are working on and build the ROM target.
3. In the project settings for the ROM target, select Remote Debugging in the left pane. Set the connection type to what you are using (e.g. P&E Wiggler). Also, set it to use the same .cfg

file you use when debugging. (The flash programmer has to connect to the board and initialize it just like when debugging.)

4. Select Tools | Flash Programmer from the menu.
5. In the left pane of the Flash Programmer window, select Flash Configuration. In the right pane, select the Device and set the Flash Memory Base Address to the proper address if not already.
6. In the left pane, select Erase / Blank Check. In the right pane, select sectors with shift-click or check the All Sectors box. You need to clear at least from the first one down as far as your program image goes (consult the map). Press the Erase button. When done, press Blank Check to verify it worked.
7. In the left pane, select Program / Verify. Check the box Use Selected File and browse to your .s19 file. Clear the checkbox Restrict Address Range if the .s19 file has only addresses in the flash area (in the Protosystem project, this is true). Otherwise, set the address range to that of your flash device. Press the Program button. When done, press Verify. If the verify fails, you may not have cleared enough flash sectors in the previous step. Check the map file or simply clear the whole flash.
8. After a successful verify, close the flash programmer window, turn off the board, disconnect the BDM wiggler from the board, and turn on the board. On some boards it is sufficient to disconnect the parallel cable from the wiggler, but on others you must disconnect the wiggler from the board.

P&E Flash Programmer

We have not used it but we heard it works well.

Drivers

Disk

See smxFS documentation.

Ethernet

See smxNS or smxNet documentation.

LED

Simple LED routines are provided in **led.c** in the board directory in the BSP (e.g. BSP\CF\5282\M5282EVB\led.c). See section APIs/ LED API at the end of this manual.

Timers

The Protosystem includes code for the timers. On ColdFires prior to the 5282, there is only 1 type of timer. Starting with the 5282, there may be several different types of timers. The timers supported by this driver are called “DMA Timers”.

timer0.s and timer1.s are provided in BSP\CF to initialize the first two timers. The first timer is used for the smx tick. The timer ISR calls smx_TickISR() in timer0.s, which does what is required of the smx tick handler: It invokes smx_KeepTimeLSR() and it updates the profile counters (profiling is optional). It initializes the timer to 100Hz by default. If you change the rate, you must also change the value of SB_TICKS_PER_SEC in bsp.h and the sb_ticktmr globals in bsp.c.

The second timer is only initialized as an example; it is not used for any purpose. You can edit timer1.s to make it do whatever you want. By default it is initialized to 50Hz.

The values used to configure the timers are defined in bsp.inc.

The only API function is for initialization (i.e. Timer0_Init() and Timer1_Init()), which is called automatically by sb_PeripheralsInit(), so there is nothing you need to call. The call to Timer1_Init() is probably commented out, but could be enabled.

UART and Terminal

We provide interrupt driven UART drivers for all ports, in uart0i.s, uart1i.s, etc. in BSP\CF. (Polled versions are provided in uart0p.s, uart1p.s, etc., if you prefer not to use interrupts.) The functions are intended for use in sending and receiving ASCII data rather than binary data. In particular a 0 return value for the In routines means no data. As written, they are useful to send output messages to a terminal and to get user input from the terminal. You will have to modify the driver or create a new one using ours as a guide, to do something different such as send binary, packetized data.

The following is a summary of the API. The functions for the second, third, and fourth drivers are the same, but replace “0” with “1”, “2”, or “3”.

Main API Routines

void **Uart0_Init**(void)

> Initializes the UART based on settings in bsp.inc. **Note that this routine does not take the configuration parameters as arguments.**

char **Uart0_InChar**(void)

> Gets and returns one character from the UART. Does not wait; returns the character if available, or 0 if not.

char * **Uart0_InStr**(void)

> Gets and returns a string from the UART. The string is NUL-terminated when a carriage return (ASCII 0x0D) is received.

void **Uart0_OutChar**(char c)

Sends one character out the UART.

void **Uart0_OutStr**(const char *s)

Sends a string out the UART.

void **Uart0_RingInit**(void)

Can be called to re-initialize the ring buffers. Clears the buffers and counters, and resets the pointers. By default the buffers are 400 bytes each.

Diagnostic Routines

u32 **Uart0_In_Ring_GetCt**(void)

Returns the number of characters in the In ring buffer.

u32 **Uart0_In_Ring_GetTotalCt**(void)

Returns the total number of characters that have been put into the In ring buffer since the buffer was last initialized (i.e. startup or when Uart0_RingInit() is called).

u32 **Uart0_Out_Ring_GetCt**(void)

Returns the number of characters in the Out ring buffer.

u32 **Uart0_Out_Ring_GetTotalCt**(void)

Returns the total number of characters that have been put into the Out ring buffer since the buffer was last initialized (i.e. startup or when Uart0_RingInit() is called).

u32 **Uart0_ISR_GetTxCt**(void)

Returns the number of times the TX ISR has run.

u32 **Uart0_ISR_GetRxCt**(void)

Returns the number of times the RX ISR has run.

u32 **Uart0_GetOECt**(void)

Returns the number of Overrun errors that have occurred.

u32 **Uart0_GetFECt**(void)

Returns the number of Framing errors that have occurred.

u32 **Uart0_GetPECt**(void)

Returns the number of Parity errors that have occurred.

Terminal I/O

If you wish to connect a terminal to one of these for input and output, ensure XBASE\bcfg.h is set so that:

```
#define SB_CON_IN 1
#define SB_CON_OUT 1
```

Specify the port for each in bsp.h as follows:

```
#define SB_CON_IN_PORT 0       /* 0, 1 (2, 3 for some CFs) */
#define SB_CON_OUT_PORT 0   /* 0, 1 (2, 3 for some CFs) */
```

These settings are independent. Input or output can be individually enabled and the port can be different for each.

When SB_CON_IN is set for a port, the InChar and InStr functions in the driver for that UART are disabled and the ISR puts the characters into op_pipe for retrieval by the opcon task (via the store_key() LSR). Otherwise, characters are put into a ring buffer and the Uartn_InChar and Uartn_InStr routines retrieve the character(s) from it.

By default, the drivers are configured for 115200-8-N-1. Turn off flow control in your terminal or terminal emulator.

Video (Graphics)

See PEG or C/PEG documentation.

Video (Terminal)

sb_ConWriteString(), and other functions in XBASE\bcon.c are mapped onto the UART driver API so text output goes out the serial port to a terminal. See the section APIs/ Video API at the end of this manual.

Other Notes

—

Tips

—

PowerPC

Architectural Notes

Tick

smx requires a periodic interrupt, called *tick*, to pace the system; all smx timing is based on this interrupt. smx uses the Programmable Interval Timer (PIT) to generate a periodic tick interrupt. Each processor family uses a different clock source to increment the PIT. The tick interrupt rate should be in the range of 100 ms to 1 ms.

For the MPC8xx and MPC5xx families, smx will automatically set up the PIT timer based on the value of SB_TICKS_PER_SEC in bsp.h.

- 8xx users see start_tick_8xx() in conf8xx.c
- 5xx users see start_tick_5xx() in conf5xx.c

For the IBM 400 Family modify #define ClksPerPitInt in ppc_bios.s to change the default tick rate and set SB_TICKS_PER_SEC in bsp.h to match.

See the Timing chapter of the smx User's Guide for more details about smx timing. Also see "Tick info" in ppc_bios.s.

Critical Exceptions for the IBM 400 Family

An external source requests a critical interrupt by driving the critical interrupt pin (CritInt). The critical exception is enabled by the MSR[CE] bit. MSR[EE] does **not** affect the critical exception. Micro Digital recommends not calling SSRs from the critical exception. In order to call SSRs from the critical exception, code must be added to the sb_INT_ENABLE() and sb_INT_DISABLE() macro to set/clear the MSR[CE] bit.

PIT Exception for the IBM 400 Family

The Protosystem application for the IBM 400 family uses the PIT interrupt for the time base (smx tick). See ba_pit_except_handler() in ppc_bios.s.

smx Library Default Processor

The smx library is built for the 600 family by default, so it will work on all PowerPCs. Setting the p macro for your particular processor will result in better optimization by the compiler and more efficient versions of the smx porting macros.

Porting to a New PowerPC or Board

If you are using a PowerPC that we do not support, please follow this guide to adapt one of our existing BSPs to your particular PowerPC. Also refer to the Protosystem section, which follows. Only refer to the smx Porting Guide if you are porting to a new compiler or CPU family that is not yet supported by smx. See the section Common Notes/ Porting in this manual for an overview of porting.

TBD

BSP Files

1. **crt0.s**, **crt1.s**, **crti.s**, **crtn.s, cwcrt0.c, cwcrte.c**

   These files contain start-up code for an embedded environment. crt0.s is provided by Diab Data to initialize the stack pointer and environmental registers before calling main(). crt1.s, crti.s, and crtn.s are provided by MetaWare to do the same. cwcrt0.c and cwcrte.c are provided by CodeWarrior to do the same.

2. **initppc.c**

   PowerPC board level configurations and initializations.

3. **ppc_bios.s**

   This file contains routines to handle interrupts. See *fit_handler* for an example of how to write an interrupt routine.

4. **ppc_link.lnk**

   Contains link map commands. Modify this file to match your memory and interrupt configuration.

5. **term.c**

   Terminal I/O routines for message output and keyboard input. Interfaces to UART driver.

CFG directory

1. **confppc.mki, confppcw.h**

   Contains configuration defines common to all makefiles, including:
   - Defines for board level support
   - Defines to enable Protosystem demos
   - Defines to build the smxFS, smxNS, and other module libraries

BSP API Extensions

None. smxPPC has its own BSP code that was written before the smx BSP API was defined. A future release will implement the smx BSP API.

Compiler Notes

Reentrancy of C Run-Time Library

Most of the functions (except the file i/o related functions) in the Diab Data and MetaWare C Libraries are reentrant. The Diab Data reentrant functions are marked with “REENT” or “REERR” symbol in the function listing. The MetaWare reentrant functions are marked with a “Reentrant” symbol in the function listing. To handle the functions that are not reentrant, use the *in_clib* semaphore or macros as discussed in section Common Notes/ Misc Notes/ C Run-Time Library at the beginning of this manual.

CodeWarrior (CW.PPC)

Last updated for CodeWarrior v8.1.

Version

Use the version indicated by the readme.txt file in the root of your release or in XSMX\smxid.txt. It is often possible to use a newer version. Use the EABI compiler.

Build Targets

Most of the project files include 2 targets: Debug and Flash. However, the Flash target is not fully complete for many boards. It is necessary to write the standalone startup code for the board and to make the CodeWarrior linker command file initializes the DATA section.

IDE

1. If you get a weird compiler error such as: “preprocessor settings don’t support hardware floating point”, it may be that the IDE has the wrong path to the source file. To check the path, select the file and press the Project Inspector button. If it is wrong, delete the file from the project and then add it again.

Compiler

1. ppc_eabi.dll dated BEFORE 12-8-97 has a bit field bug. ppc_eabi.dll dated 12-8-97 has fixed this bug.

Assembler

1. Parentheses are required around .if directives ie .if (CW == 1)
2. ppc_eabi_asm.dll dated 10-10-97 has the following bug: .if can NOT have a .include in it ie, the following will cause the error: “IF directive not balanced by ENDIF or ENDC”. Example of proper syntax:

```
.if (CW == 1)
.include "file1.inc"
.include "file2.inc"
.endif
```

3. ppc_eabi_asm.dll dated 10-10-97 does not allow a constant to be loaded into a register. The following will cause the error: "Invalid argument 1000@ha"

```
addis   r15, r15, 1000@ha
```

Linker

1. CodeWarrior 3.0 changes the names of all of the data/code sections. smx has been updated to work with the new names. If you wish to use CodeWarrior 2.0 contact Micro Digital for 2.0 version of CW.C.

Libraries

1. Summary

```
runtime.ppceabi(N).a = no floating point support
runtime.ppceabi(S).a = software emulation floating point support
runtime.ppceabi(H).a = hardware emulation floating point support
```

   This is documented in "Targeting PowerPC for Embedded Systems". You can find it in the directory: CodeWarrior Documentation/Embedded [doc].

2. Starting with IDE 6.0, building a library with the library option causes unresolved link errors. Use partial link instead.

Debugger

We assume you will use the CodeWarrior debugger, but it is also possible to use SingleStep, if you have it. See the section PowerPC/ Tools/ SingleStep.

Diab (DC.PPC)

Last updated for Diab v4.3g.

Version

Use the version indicated by the readme.txt file in the root of your release or in XSMX\smxid.txt. It is often possible to use a newer version.

Installation

1. If install asks for default target select EABI backend: **PPC403ES**. This will create the file diab\4.3g\conf\default.conf which can be modified directly. Do not be concerned with the

target family in this file because it will be superceded by the target specified in the mak command.

2. Add to your path in autoexec.bat: **c:\diab\4.3g\win32\bin**.

3. Change the "libdir" line in \smx\app\dc.ppc**pro.mak** to reflect the directory that your Diab C library is in.

Build Targets

The standard build targets are supported (Debug and Release), as discussed in the Common Notes section at the beginning of this manual.

Compiler

1. "illegal char 134(octal)"

   Could be caused by a #define line with spaces or tabs after continuation character.

2. "bad #pragma"

   Could be caused by comments on #pragma line.

3. "illegal storage class for xxx"
   "redeclaration of xxx"

   Could be caused by using:

   ```
   #pragma global_register xxx=r14
   extern int xxx;
   ```

   xxx must be declared global in every file (don't use extern).

4. If a program executes correctly when compiling without optimizations but fails when optimized, it could be one of the following:

   a. If a global is modified in one task and read in another task then make the global volatile so the evaluation doesn't get optimized out. For example:

      ```
      volatile int busy;
      ```

   b. Use of memory references mapped to external hardware. Add the volatile keyword.

   c. Use of uninitialized variables exposed by the optimizer.

   d. Use of expressions with undefined order of evaluation.

5. Complier error: "(1000) Flexlm error: No such error exists."

   Temporary license has expired. Contact Diab.

Assembler

1. assignment must start in the first column (e.g. nine .equ 9)

2. .endm must **not** be on the 1st line. May cause "error: end of file in macro".

3. .extern symbol directive doesn't seem to do anything.
4. asm macros in a C file: _asm and { and } must be on line one.

Linker

1. The Diab Data compiler/linker does not guarantee to keep adjacent variables together in memory. For example:

```
int   padding[20];
u8    bstack[10000];
int   estack;
int   postpadding[20];
```

These may not be kept together in memory.

2. Linker may combine interrupt vectors if there is enough space left over in the previous interrupt vector. You can get around this by restricting the amount of space in the vector to the amount required by that vector. For example, in file ppc_lnk.lnk:

```
reset_excpt:    org = 0x0000100, len = 0x00032
dec_excpt:      org = 0x0000900, len = 0x00004
pit_excpt:      org = 0x0001000, len = 0x00004
fit_excpt:      org = 0x0001010, len = 0x00004
```

Debugger

See section PowerPC/ Tools/ SingleStep.

MetaWare High C/C++ (HC.PPC)

Last updated for High C/C++ compiler v4.5.

Version

Use the version indicated by the readme.txt file in the root of your release or in XSMX\smxid.txt. It is often possible to use a newer version.

Installation

1. Follow the MetaWare installation instructions. Select big endian (be) if given a choice.
2. Add the path to the MetaWare BIN directory to your path. If other compilers are installed, you should create a separate working environment for each. See the section Common Notes/ Misc Notes/ Command Line Environment in Windows.
3. In \smx\cfg\confppc.mki, change the libdir line to reflect the directory that your MetaWare C library is in. Change the fp line if you want to enable hardware floating point.

4. Verify that you are using the following versions (or later):

compiler	4.5
assembler	3.10
linker	5.1f

Build Targets

The standard build targets are supported (Debug and Release), as discussed in the Common Notes section at the beginning of this manual.

Compiler

1. Inline assembly must use %r15 instead of r15 for registers. See the note about Register-Naming Schemes in section PowerPC/ MetaWare/ Assembler.
2. If a program executes correctly when compiling without optimizations but fails when optimized, it could be one of the following:
   a. If a global is modified in one task and read in another task then make the global volatile so the evaluation doesn't get optimized out. For example:

      ```
      volatile int busy;
      ```

   b. Use of memory references mapped to external hardware. Add the volatile keyword.
   c. Use of uninitialized variables exposed by the optimizer.
   d. Use of expressions with undefined order of evaluation.

Assembler

1. A semicolon on a .macro line causes the assembler to lock up, use # for a comment.
2. The assembler sometimes causes Win95 to display the message "MS DOS may not run well unless it is run in MS DOS mode. Would you like to create a shortcut?". Answer NO. Check the log, the file probably assembled correctly.
3. Register-Naming Schemes: The ELF assembler now supports the following register naming formats:
   a. %f<n> and %r<n>
   b. f<n> and r<n>
   c. <n>

   where <n> is the register number.

   The first and third register-name formats are unambiguous (that is, there is no conflict between these names and identifier names) because C/C++ does not permit identifier names starting with a digit or a '%' character.

To make the second register-name format (“[f|r]<n>”) unambiguous, specify assembler command line option “-%reg” or “-percent_reg”, or set one of the following in the assembly file:

```
.option %reg
.option percent_reg
```

If you specify option “-%reg” (or “-percent_reg”) on the assembler command line, or set “.option %reg” (or “.option percent_reg”) in the assembly source, the assembler constrains itself to accept register names only in the format “%[f|r]<n>” or “<n>”. This constraint permits you to define and use identifiers with names like r0, r1, ..., f0, f1, ..., and so on.

If you do not constrain the assembler, it accepts all three formats as valid (in which case r<n> and f<n> must be register names and cannot be identifiers).

By default, the compiler generates assembly sources with the “%[fr]<n>” format for registers. However, the hcppc.cnf file passes option “-%reg” to the assembler to ensure that identifier names like r<n> and f<n> are not rejected by the assembler as invalid identifiers.

Linker

1. The linker uses -m to create a memory map, but it is only sent to the standard output. We capture it along with compiler and linker messages to a file named “log”.

2. You may see this error:

```
Error:  library not found -lcc
        library not found -lc
        library not found –lsds
```

Some versions of the linker can’t find the MetaWare libc.a and libcc.a library if the library is on a different drive than smx. Linker version 5.1 fixes this problem.

Debugger

We assume you will use the SeeCode debugger, but it is also possible to use SingleStep, if you have it. See the section PowerPC/ Tools/ SingleStep.

Setup

The first time SeeCode is run a number of options need to be set.

1. Create a Windows icon, right-click it and select Properties. Then:

   a. Set the Working (or Start in) directory to C:\SMX\APP\HC.PPC\Debug

   b. Set the Command Line (Target) to scppc.exe -OKN

2. Run SingleStep. In the “Debug a process” window in the Command Line dialog box enter “app.x” This is the elf file to be debugged. Click Ok.

3. When the “Command window” appears, select Menu item “Dialogs >> Options” to bring up the “Single-process Options” window. Set Program name to app.x. In the “Program Options 1” window set RTOS selection to None and uncheck (disable) Auto execute to main.

4. Select Machine-specific options from the “Single-process Options” window.
    a. Set Target interface to Macraigor JTAG or Abatron JTAG.
    b. Set the Target Chip.
    c. Enable the “Use a chip initialization file” check box. For the Ibm405 chip use the chip initialization file: C:\hcppc4.5\sc\chipinit\ibm405
5. In the “Machine-specific options” window select Macraigor setup OR Abatron setup. Click the Save button.
6. These setting are saved in files C:\SMX\APP\HC.PPC\Debug\.sc.args and .sc.properties
7. Close SeeCode by closing the “Command window”. The next time you run SeeCode it should load app.x to your target instead of the simulator.

Tips

1. SeeCode uses a lot of resources, so close down the apps you don’t need and run the Microsoft Resource Meter to make sure you do not run low.
2. If you have trouble getting SeeCode to run, cycle the power to the JTAG pod and the board.
3. When SeeCode starts, the command and source windows will be displayed. We recommend opening the Source file window (Windows >> Display... >> Source file). Use this window for easy access to all smx files compiled with the debug option (-g). Size and position these windows the way you like them and save the setup (Window >> Save windows to current directory).
4. If SeeCode does not expand structures, add -g to compile flags. See how “dbg_sym1” is used in the makefile (pro.mak).

Tools

SingleStep

SingleStep is typically used with Diab C/C++, but the information is presented here because it is a stand-alone tool that can be used with any PowerPC compiler.

Simulator and Debugger

SingleStep is offered as both a simulator and debugger. The simulator can be used until target hardware is available, using simulated peripherals. Both the simulator and debugger have the same user interface, so switching between them is easy. The debugger can interface to an emulator, BDM connector, or target monitor (software).

Installation and Setup

1. Leave the C++ Activation Key field blank.

2. Modify sstep.ini (in directory SMX\MISC\SSTEP\PPC) to match your configuration. Uncomment the **memorytype** statement that most closely matches the target board you are using. This is used to select an alias _config statement to load a config file for your board. Copy .cfg files from SMX\MISC\SSTEP\PPC to SINGLESTEP\CMD.

3. Modify srcpath in sstep.ini to match where you installed SMX (drive and path). For example:

   ```
   set srcpath = ( c:\smx\xsmx c:\smx\bsp\ppc c:\smx\app )
   ```

   Note the space after "(" and before ")".

Tips

1. If SingleStep takes an excessive amount of time to start (more than 1 or 2 minutes), abort the load and try again. It always seems to load the second time.

2. We recommend you purchase smxAware (from Micro Digital). This DLL adds to SingleStep the ability to display and navigate tasks and other smx objects symbolically.

3. To view and walk the heap, open the watch window from the menu. In the name box type: smx_cf.heap_min. In the change type box, type: struct HCB* and click OK. Double click on nbp to expand the next heap block.

4. To view the handle table, open the watch window from the menu. In the name box type: hti. In the Display Limit tab, select Treat ptr as ptr to array of size 50 and click OK.

6. If loading the debug file causes this error: "memory overlap down", it is caused by the ROM length too large in file ppc_lnk.lnk

7. To watch registers that may not be in the Register windows, bring up the Add To Watch dialog box. Type in $reg:

   ```
   $DEC or $MSR
   ```

8. If you get a SingleStep "illegal instruction" or app.x never leaves the idle task the cause could be: Processor object type is 600 family. If your target is the 400, 500, or 800 family set the File >> Debug >> Processor to your processor.

9. The following warnings do not affect debugger operation

   ```
   fromelf: warning: unexpected fundamental type: 0x8008
   fromelf: warning: unexpected fundamental type: 0x1A
   ```

   These error messages mean you are using something that is not supported in SingleStep such as long long or boolean expression.

10. If SingleStep aborts the download with the error: "down alias", scroll the debug window's dialog box to find the cause of the error. The error is usually caused by interrupt vectors overlapping the code section. SingleStep 7.03 file fromelf.exe has a bug in it that can cause this error. Contact SDS to download a new fromelf.exe with a date of 1/15/97 or later, then delete files .db, .db2, .dbk, and .ou1 and try the download again.

Drivers

Disk

See smxFS documentation.

Ethernet

See smxNS or smxNet documentation.

LED

—

UART and Terminal

TBD

Video (Graphics)

See PEG or C/PEG documentation.

Video (Terminal)

sb_ConWriteString(), and other functions in XBASE\bcon.c are mapped onto the UART driver API so text output goes out the serial port to a terminal. See the section APIs/ Video API at the end of this manual.

Other Notes

—

Tips

—

Appendix A: Makefile Structure

This appendix shows a sample Protosystem makefile and explains the sections step-by-step. The Protosystem makefile handles a lot of complexity so it is complicated by macros. It may look intimidating if you are not familiar with makefiles, but it is pretty simple once you learn a few concepts. The library makefiles are all much simpler but resemble the Protosystem makefile, so learning about the Protosystem makefile teaches you about all SMX makefiles.

Makefile syntax varies for each make utility. The makefiles we supply for a particular compiler suite are written in the syntax for the make utility supplied with the compiler. For compilers that do not include a make utility, we have standardized on the Microsoft NMAKE syntax, since this is a good, commonly available make utility, and it is emulated by third-party make utilities such as Opus Make. The following makefile example is for NMAKE and Microsoft tools. The syntax for other make utilities is quite similar so it should not be hard for you to relate the following discussion to your makefiles.

See the Tips at the end of this Appendix for more information.

<1> Configuration: This section has various macros for configuring the makefile.

<2> Module Library Selection section. These macros select which of the SMX module libraries are linked. This section is pre-configured by Micro Digital as part of shipping your order, to include all modules purchased. Demos are now selected in a separate file that is included by the makefile, called DEMODEFS.MKI. In order to link a demo, the corresponding module library must be enabled or the makefile will issue an error message.

<3> Paths section: These are macros that specify the paths to the files used during the build. First are listed paths to our directories, then paths to third party libraries. If you are using any of these, ensure the path in the makefile matches the directory where you installed those libraries. The *smxroot* macro allows moving the Protosystem directory anywhere.

<4> Debug, and Release settings. These sections specify the assembler, compiler, and linker switches that differ between these versions. The version built is whichever is specified when you run **mak.bat**.

This section also specifies which library to link for each module. Note that some have a second line commented out to link the Debug ("d") library. Linking a Debug library should be done only when you are debugging files in the library itself. Having both lines in the makefile, with one commented out, makes it easy to switch libraries.

<5> This line includes the file DEMODEFS.MKI inline into the makefile. This include file is where you configure which demos are enabled. When enabling a demo, the corresponding library must also be enabled, near the top of pro.mak.

<6> This defines the *mk* macro to be the makefile. Files with $(mk) dependencies are rebuilt if the makefile is changed. This is necessary since you may select another demo to run (in demodefs.mki), which causes new conditionals to be defined, so the files using any of these conditionals must be rebuilt.

<7> Module Libraries: These macros are set to the appropriate libraries from the Debug or Release section above, depending on which module libraries are enabled. Also, they specify any additional object files to link for the module.

<8> This is the main rule in the makefile, specifying the ultimate target, which is the Protosystem executable. The files and libraries listed to the right of the colon are the *dependencies*. $(pro).exe is the *target*. The way a make utility works is that the target is rebuilt if any of the dependencies has a later timestamp than the target. That is, the Protosystem is rebuilt if it is out of date, with respect to any of the files from which it is built.

Following the rule is the sequence of commands to be issued if the Protosystem must be built.

```
# PROTOSYSTEM MAKEFILE
        ...
<1>     #***************************** CONFIGURATION *******************************
        …
<2>     # Module Library Selection
        #saware  = /DSMXAWARE
        #sfs      = /DSMXFS
        ...
<3>     #************************************ PATHS ************************************
        bf      = ..            # build files (makefiles, locator scripts, etc)
        i       = ..\..         # Protosystem/application include files
        ...
        # 3rd-party library/include paths
        pegd = \peg\examples\pegdemo
        pegi = \peg\include
        …

<4>     #[d]***************** DEBUG VERSION SETTINGS *******************
        !IF ("$(v)" == "d") || ("$(v)" == "D")
        ax = /DSMX_BT_DEBUG /Zi
        cx = /DSMX_BT_DEBUG /Odi /Z7
        lx = /DEBUG /PDB:NONE
        pro = app
        l           = $(smxroot)\xsmx\mc.p3\release\smxr.lib
        ...
        #[r]***************** RELEASE VERSION SETTINGS ***************
        !ELSEIF ("$(v)" == "r") || ("$(v)" == "R")
        ...
        #*********************************************************************

<5>     !include $(bf)\demodefs.mki

<6>     mk = $(bf)\pro.mak $(DEMO_MAKS)

<7>     # Module Libraries
        !IFDEF saware
        smxaware = smxaware.obj
        !ENDIF
        !IFDEF sfs
        fsl = $(xfsl)
        !ENDIF
        ...

<8>     #******************************** BUILD RULES ********************************

        $(proto)   : app.obj bios.obj bsp.obj main.obj \
        ...

         LINK @<<
        $(lx) /BASE:0 /FIXED:NO /MAP /NOD /OUT:$(pro).exe …
        segf startf bios bsp main app ...
        ...
        <<
        !IF ("$(v)" == "d") || ("$(v)" == "D")
              echo Preparing Debug Files > con
        ...
              echo    *** All Done *** > con
        !ENDIF
```

<9> The Common Switches section shows the switches that are common to all or virtually all files. Later macros in sections for building particular groups of files use the co and ao macros in their definitions. Notice that the $(cx) and $(ax) macros are used, which specifies additional switches that depend on the version being built (e.g. Debug, Release) — see <A3>.

<10a> The switches on this line are primarily defines that describe the target environment. Defines we use are documented below. The switches encapsulated by the cx and ax macros are both defines and compiler switches that vary for the version being built (i.e. Debug vs. Release).

<10b> These macros equate to defines such as /DSMXFS, which enables conditional code for particular modules (smxFS, in this case). The DEMO_DEFS macro is defined in demodefs.mki and adds all of the defines for to enable particular demos (e.g. SMXFS_DEMO).

<10c> These are include paths to search for header files.

<10d> These are compiler switches.

<11> The Protosystem Core Files section lists the rules for building the files that are part of the Protosystem. Just like the rule for $(pro).exe, the object file (e.g. main.obj) is built if any of the dependencies (i.e. source files, include files) have a newer timestamp. The command to build the object file runs the compiler. Notice that a new macro, cp, is defined for all files in this section. Similar macros are defined for subsequent sections. What differs is the path to the source file.

<12> This section builds the assembly files for the Protosystem. The ap macro is used to build all files in this section.

<13> Include of DEMORULE.MKI: This file is included inline into the makefile. It contains the rules for building demo files.

Other sections may be added as support for additional modules and third party libraries are supported. Also, the ordering may change, but this gives the general idea. If you have read through this section, you should feel quite comfortable with them now.

```
<9>     #*************************** Common Switches ********************************

<10a>   co =    $(cx) /DPME32 $(dpmi) $(embedded) $(ht) \
<10b>           $(mw) $(msys) $(peg) $(saware) $(scd) $(scom) $(sd) ... \
<10c>           /I$(i) /I$(pme) /I$(x) /I$(cfg) /I$(ppi) /I$(fi) /I$(ni) /I$(pegi) ... \
<10d>           /c /G3 /Gs /nologo /W3 /Zp /Fo$@

        ao =    $(ax) /DPME32 /DMICROSOFTC /DMASM … \
                $(mw) $(msys) $(peg) $(saware) $(scd) $(scom) $(sd) ... \
                /I$(i) /I$(pme) /I$(x) /I$(cfg) /I$(fi) /I$(mwi) /I$(di) … \
                /c /coff /Cp /nologo /W2 /Zm

<11>    #*********************** Protosystem Core Files ***************************

        cp = $(co) /Tp$(s)\$*.c
        h = $(i)\acfg.h $(i)\main.h
        ...
        main.obj:                       $(s)\main.c $(h) $(mk)
                echo COMPILING $*.c > con
                set CL= $(cp)
                CL
        ...

<12>    ap = $(ao) /Ta$(s)\$*.asm
        a  =

        bios.obj:                       $(s)\bios.asm $(a) $(mk)
                echo ASSEMBLING $*.asm > con
                set ML=$(ap)
                ML
        ...

<14>    !include $(bf)\demorule.mki
```

Tips

1. Compiler, assembler, and linker switches are documented in this manual, in the section for your compiler.

2. SMX makefiles make extensive use of macros and this is what may make them appear complicated. Macros are simply text strings defined like this:

   m = something

   Wherever the make utility encounters $(m), it textually replaces it with “something”. If the makefile has this text in it: abc$(m)123, it will be replaced with abcsomething123. Macros are case-sensitive. $(M) would be a different macro. If M is undefined, $(M) expands to nothing, so abc$(M)123 expands to abc123. Also, macros can be nested. For example:

   M = abc$(m)123

   Macros are very useful for allowing options in a makefile.

3. You can simplify your makefile by using your editor’s Replace command to replace a given macro with the text for that macro. Use your editor to do the replace rather than doing it manually to ensure accuracy. (You would do this only for the settings that do not vary for your particular environment.)

Index