SMX® RTOS

Target Guide

Version 5.2
February 2024

by
David Moore

Micro Digital

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.

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 multiple 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):
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). 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

PDF files in the DOC directory summarize important information about SMX support for various boards. 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 and XBASE directories. 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 **acfg.h**

acfg.h has application-related configuration of smx, such as numbers of objects and memory area sizes such as main heap and stack pool. (xcfg.h has smx kernel configuration.) 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.

2 **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.

3 **smxmain.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. aexit() is used to exit. It can be made to infinite loop or do whatever is appropriate for your system on exit.

4 **smxmain.h**

This file provides function prototypes and declarations for the Protosystem.

5 **smxmods.c**

This file contains init and exit 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().

6 **smxaware.c**

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

7 **XXX.YYY Subdirectory** (e.g. IAR.ARM)

Build directory. Project files, locator config files, debug macro files, etc are stored here.

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 smxmain.c. See the Protosystem section in the CPU section for a list of startup files for your CPU.

Target Defines

The project files pass several target-related defines to the compiler and assembler to control conditional compilation/assembly of the code. 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);
```

Instead of calling sb_IRQVectSet(), the ISR can be statically initialized in the vector table in the BSP, if there is one e.g. vectors.c (ARM-M) or vectors.s (ColdFire) or in the IRQ dispatcher irqdispatch.c (ARM). Some processors such as ARM that do hardware vectoring have registers to store the ISR addresses, so this cannot be done.

Alternatively, these functions can be used.

```
sb_ISRInstall(IRQ_NUM, par, MyISR, "Name");
sb_IRQUnmask(IRQ_NUM);
```

sb_ISRInstall() is used by SMX middleware and is very convenient because it automatically assigns an ISR shell of the right type for the processor architecture, and it also allows passing a parameter to the ISR core function. This is usually used to pass an index to indicate which controller has interrupted, when there are multiple controllers of a type, such as USB host. This avoids having to duplicate the ISR function for each. It also automatically creates a pseudohandle for the ISR, giving it the specified name, so it is tracked in the SMX Event Buffer displayed by smxAware. It also calls sb_IRQConfig() to configure the IRQ. It saves the vector that was previously installed, so it can be later restored by sb_ISRRestore(), if desired. The ISR shell does smx_ISR_ENTER/EXIT() and calls the dispatcher, sb_ISR(), passing the parameter to it. sb_ISR() calls smx_EVB_LOG_ISR/RET(), sb_IRQClear(), sb_IRQEnd(), and the ISR function, passing the parameter to it. The ISR function is then just a normal C function, as all of the smx-related operations have been done by the shell and sb_ISR().

For more information about the foregoing functions, please see the API section of the smxBase User's Guide.

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 rather than inline 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 may use a different name for them. They are documented here 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 control blocks. The settings here are used in XSMX files.

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 or performance needs to be improved.

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

We recommend that you start with the project files we provide. If you prefer to create your own or create a makefile, 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 multiple setting 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.

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. That is, test it before the call and signal it after. If having only one 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. They can cause stack overflows that go undetected because the stack pointer jumps a large amount, possibly past the pad. Tasks that use such functions need larger stacks. smx stack usage checking and padding are a big help in catching stack overflows such as this. When possible, use simpler functions; in this case, use itoa() instead of sprintf(). Alternatively, you can create a simpler, 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 sized for the typical task in your system. Bound stacks are allocated by simply specifying a non-zero stack size 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 size

Heap size is controlled in acfg.h or the linker command file. See the Heap chapter 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 100 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.

Profiling

See the Precise Profiling chapter in the smx User's Guide.

SMX Utilities

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

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 option.

Syntax: mibtoc <infile> [outfile]

MpuMapper

Modifies map file to have symbols in address order, interleaved in placement summary section. See SecureSMX User's Guide.

MpuPacker

Helps order sections in linker command file to minimize memory waste. See SecureSMX User's Guide.

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}

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.

Tips

Debugging

1. If smx does not seem to be running correctly, set a breakpoint on **smx_EMHook()** in smxmain.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.
2. 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_errno, 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).

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 (maybe via compiler)
xarm_iar.s:	single ISR calling dispatcher; IAR assembler

Hardware Vectoring:

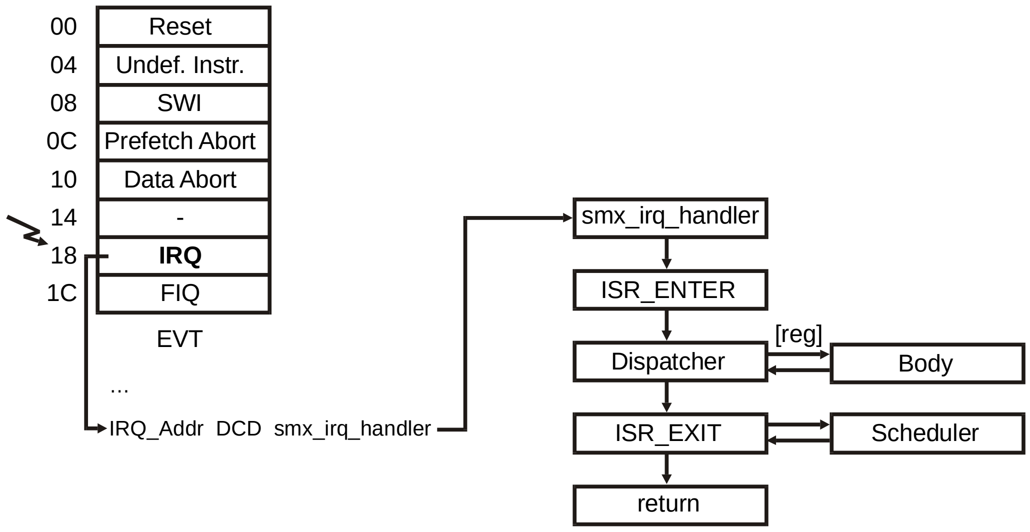
XSMX

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 (maybe via compiler)
xarm_iar.inc:	smx_ISR_ENTER/EXIT macros; IAR assembler

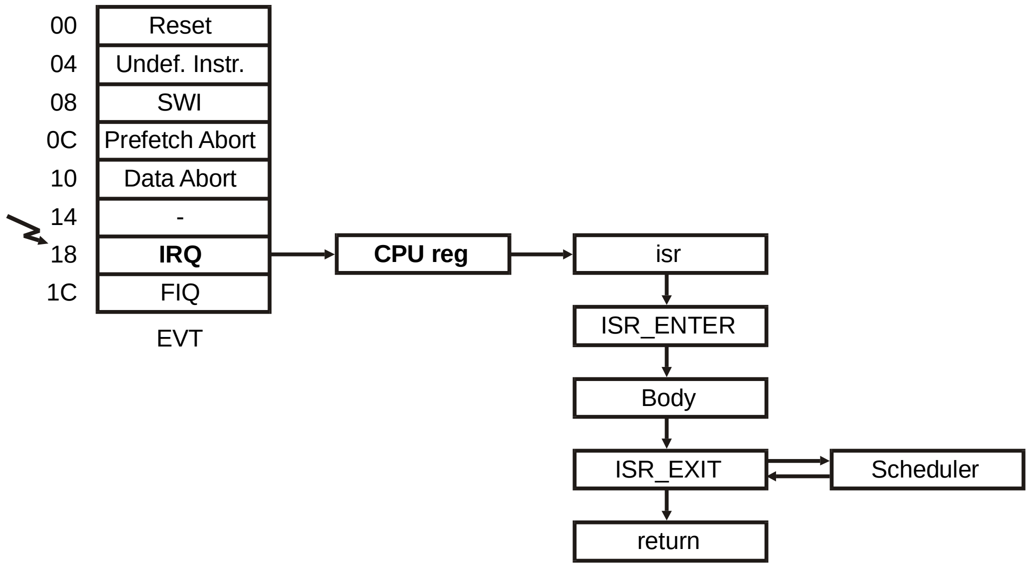
BSP\ARM

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 (maybe via compiler)
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:



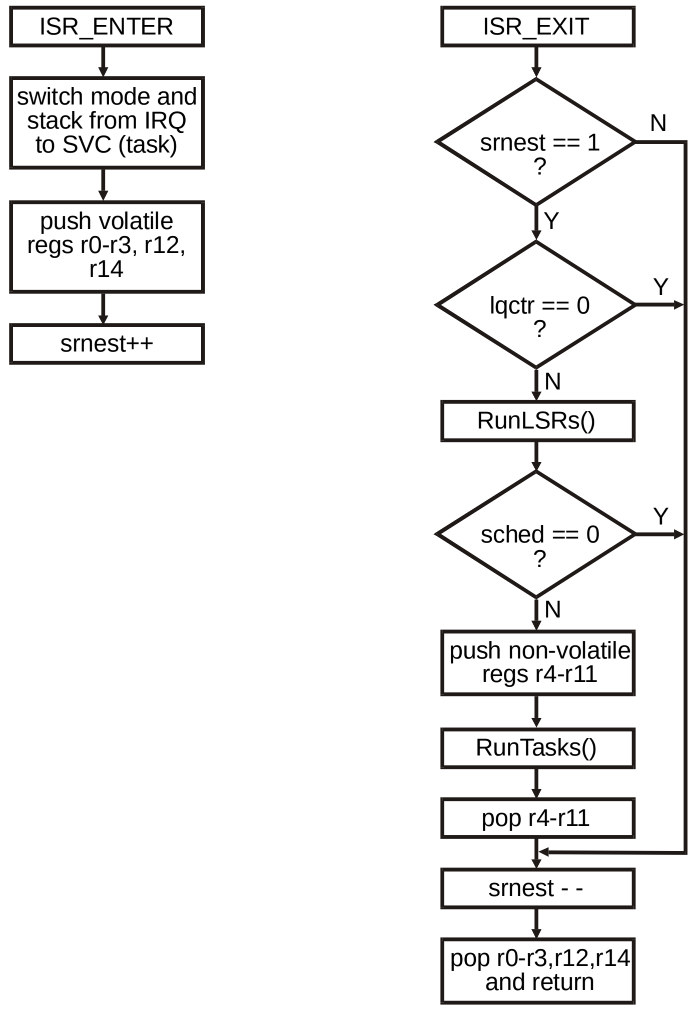
Software Vectoring



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.


ISR_ENTER
switch mode and stack from IRQ to SVC (task)
push volatile regs r0-r3, r12, r14
srnest++
ISR_EXIT
srnest == 1 ?
N
Y
lqctr == 0 ?
Y
N
RunLSRs()
sched == 0 ?
Y
N
push non-volatile regs r4-r11
RunTasks()
pop r4-r11
srnest - -
pop r0-r3,r12,r14 and return

Thumb Code

We have seen no interest in Thumb support for traditional ARM, so we have not worked with it in many years. In the past, we 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 a customer spent a couple days to find out why their regularly occurring interrupt would sometimes occur much later than expected. 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 processor 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 architecture 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. IAR Embedded Workbench: Create a new board preinclude file similar to the .h file provided (e.g. stm32746geval.h). Modify iararm.h to include it.

    b. CrossWorks: Create a new board preinclude file similar to the .h file provided. Modify gcwarm.h to include it.

4. Create a new build directory for your board, under APP\IAR.ARM, or similar. Copy the project files and other build files to the new directory and rename them for your target. Then modify the project in the IDE 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, …** (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 often have copied only the files we needed to subdirectories under BSP\ARM, and we have made any necessary changes to them. Add any others that are useful or their whole BSP, and transfer our changes to the new files. They are tagged with MDI: comments.

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 sb_irq_table[]. Generally, sb_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 sb_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.

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

Last updated for IAR v9.10.This information applies to ARM and ARM-M, and is not repeated in the ARM-M architecture section.

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, _iar910 in the name of the project files means v9.10. 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 directory 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, which opens the project file(s) it contains.

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. Selection of SMX modules 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. 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 disk and directory, 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.

Note that we typically located the System Stack after some other data 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.

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. SecureSMX includes our MpuMapper utility which makes some modifications to the map file to make it more helpful for partitioning code for security. See the SecureSMX User’s Guide about it.

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. We successfully use and recommend IAR I-jet or J-Link.

See the Embedded Workbench Debugging Guide, Part 4: Additional Reference Information/ Reference Information on the C-SPY Hardware Debugger Drivers for directions to set up your debug hardware.

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: I-jet | Breakpoint Usage (or in place of I-jet, the debug probe you are using). 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 don’t know how to disable it.

Flash Loader

EWARM has a built-in flash loader interface and includes pre-made flash loaders for many 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 is 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.AM, or similar. For example, the workspace file for STMicro STM32H753I-EVAL is here:

   ```
   APP\IAR.AM\STM32\App_stm32h753i_iar830.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 C++ in the Compiler Language setting to support features we use such as default parameters.
3. Files are added to a project with by right-clicking the top node or a group/folder 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.

   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 and Groups: If a file or group in the project has a gray icon next to it, it is excluded from the build. To change this, right click it and select Options, and 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. Alternatively, set a breakpoint in the startup code.
3. smxAware, included with smx, is a DLL that plugs into C-SPY to display smx objects and graphs. The graphical displays 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 kernel, 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

—

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 have done occasional projects with chip vendor tools, but the GNU tool we've spent the most time with is Rowley CrossWorks. Information for it is is below. If you are using different tools, you will need to create the project files or makefiles, linker config files, 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.

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).

The nicest thing about CrossWorks is that the Windows version of the tools is natively hosted on Windows and does not require Cygwin. It has a nice IDE and debugger built in. These are proprietary, not Eclipse-based.

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.

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. Mostly, they define preprocessor symbols to indicate the processor and board, and they have defines to indicate which SMX modules and demos to link. The following is a summary:

C/C++

gcwarm.h	Master preinclude file. Selection of SMX modules 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. 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 ours, except for the smx scheduler. 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.

By default, our project files set the optimization level of Debug targets to None and of Release and ROM targets to Optimize For Size. 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 we just compile everything for C++.

Exception Handling

To enable exception handling, select the Common target in the Properties Window. 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: If the smx kernel is being built as a separate library, in the Release target, ensure that debug symbolics are enabled for xglob.c or the whole library.

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.

IAR 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.

Drivers

Disk

See smxFS documentation.

Ethernet

See smxNS documentation.

LED

Simple LED routines are provided in **led.c** in the board directory in the BSP (e.g. BSP\ARM\STM32\STM32H7xx\STM32H753I-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 different processor architecture.

“Cortex” does not mean this newer 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, M1, M3, M4, M7, M23, M33

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 and Cortex-M4 Processors,” Joseph Yiu, ISBN 978-0124080829, 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, users 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.

Flash Locking

Some processos have the ability to lock the flash for security. Unfortunately this sometimes happens by accident and 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.

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 it may be shared by several related BSPs.

4 **irqtable.c**

Contains just sb_irq_table[] which defines the priority and any other properties for all interrupt vectors. In other BSPs (e.g. traditional ARM), 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, 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 sb_irq_table[]. Generally, sb_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 sb_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 may have forgotten to enable a peripheral you are using, by enabling its clock. This is especially likely for ARM-M processors.

   Solution: Use the chip vendor BSP routine to enable the peripheral.

2. 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 sb_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. See the section ARM-M/ Architectural Notes/ ISR Priority Level for more discussion about BASEPRI and PRIMASK.

3. 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.

Index

Index