SMX® RTOS

Target Guide

Version 5.4
July 2025

by
David Moore

Micro Digital Associates, Inc.
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

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 processor architecture, compilers, and tools. These are continually changing, so please consult the release notes in the DOC directory for additional and corrected information.

Be sure to use the version of this manual that matches your version of SMX, as information has been removed here for architectures and tools no longer supported.

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

4. **main.h**

   This file provides function prototypes and declarations for the Protosystem.

5. **mwmods.c**

   This file contains init and exit code needed for some middleware modules. It is divided into sections for each module. It has 2 top-level routines, mw_modules_init() and mw_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. **sys.c**

   Contains system-related routines such as idle and opcon tasks and heap management routines.

8. **XXX.YYY Subdirectory** (e.g. IAR.AM)

   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 shared by all, 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 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).

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.

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.

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.

Tips

Debugging

1. 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.
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-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 to be 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.

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 and ARMv8-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 other architectures 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**

   Simple API for writing to LCDs. Used by lcddemo_task_main().

6. **lcddemo.c**

   LCD 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. **uart.c, uarti.c**

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

10. **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.

11 **STM32, LPCxx…** (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.

IAR Embedded Workbench ARM (IAR.AM)

Last updated for IAR v9.10.

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:

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

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 main 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.

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_iar910.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.

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.

Drivers

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_CFG_CON 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.

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.

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.

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