SMX® RTOS

Quick Start

Version 5.2

February 2024

by
David Moore

Micro Digital

Contents

Installation

SMX releases are posted online, and the access information is emailed. Tools and some third-party software are often shipped directly from the vendor. For MDI software, we create a release of exactly the modules you ordered, for the processor and tools you are using. We configure header files and demos, and we test it on a common evaluation or development board. This ensures you get a quick start.

SMX

Installation is as simple as downloading and unzipping the release file and adding \SMX\BIN to your path so our utilities can be found.

Compiler and Tools

See the SMX Target Guide for any extra steps and tips about installing your compiler and tools. Look for an Installation section at the beginning of the section for your compiler and tools.

Documentation

Manuals

Manuals are provided in PDF form at www.smxrtos.com/doc.

- **SMX Quick Start** — overview of SMX® Modular RTOS (this manual)
- **SMX Target Guide** — details about processors and tools
- smx kernel manuals
  - **smx User's Guide** — explains use of smx and multitasking
  - **smx Reference Manual** — kernel API and glossary of terms
  - **SecureSMX User's Guide** — explains use of secure features based on the MPU
- module manuals
  - **smxAware**
  - **smxFS**
  - **smxNS**
  - **smxUSBH**
  - **smxWiFi**
  - etc.

BSP Notes

These are PDF files that summarize important information about target boards. They show memory layout, peripherals supported, important notes, and other details and tips about the board. One of these is provided in the DOC directory for the BSP you ordered.

Release Notes and Text Files

ASCII .txt files in the DOC directory provide additional information about the modules (products) you licensed that is not covered in the manuals. The smx release notes (e.g. smx52.txt) contain important notes and changes from the previous version of smx.

Conventions

Since the SMX Target Guide is organized in a hierarchy several levels deep, we often use the following convention to refer to sections in it: Section1/ Section2/ Section3/ …. Section 3 is a sub-section of Section 2, which is a sub-section of Section 1. The space is put after each slash for readability and word-wrapping. We use backslashes in file paths.

Global Concepts

smx vs. SMX

smx means the smx multitasking kernel. SMX means the RTOS, which includes smx and all middleware such as smxFS, smxNS, and smxUSB.

Directory Structure

Main Directories

APP	Protosystem directory. See Protosystem section below. Contains demos too.
BIN	Utilities.
BSP	Board and processor support code.
CFG	Global configuration files; mainly preinclude files and similar files related to specifying modules to link and the target hardware to build for.
DOC	Documentation, including BSP notes and release notes.
ESMX	Example files that link with the Protosystem.
MPU	Memory Protection Unit support files, for SecureSMX.
XBASE	smxBase files. This is the foundation for all SMX modules. Contains general definitions, common/core Protosystem files, and OS porting layer used by middleware modules.
XPORT	Porting files to migrate from other RTOSes to SMX, such as FRPort for FreeRTOS and TXPort for ThreadX.
XSMX	smx kernel directory. Stores the smx source and header files.

Module Directories (may or may not be present in your release)

SA	smxAware. Copy to tool directory. See smxAware User's Guide.
XFD	Flash driver source files used by smxFS, smxFFS, and smxFLog.
XFFS2	smxFFS source files.
XFL	smxFLog source files.
XFS	smxFS source files.
XNS	smxNS source files.
XSMXPP	smx++ source files.
XUSBD	smxUSBD device stack source files.
XUSBH	smxUSBH host stack source files.
XUSBO	smxUSBO On-The-Go add-on source files.
XWIFI	smxWiFi source files.

"X" directories are module directories. Project files are in each.

Build Directories (XXX.YYY\ZZZ)

XXX.YYY is the **build directory**. The project files and other build files are stored here.

XXX designates the compiler:

IAR	IAR Embedded Workbench

YYY designates the processor:

AM	ARM-M (e.g. Cortex-M)
ARM	ARM (traditional, e.g. ARM9)

ZZZ is the **output directory**. The object files, library, executable, map, etc. are stored here after running a build. This directory is created automatically by the build. The name is usually Debug, Release, or ROM.

Benefits of SMX Directory Structure

- Files for each module (product) are separate. Header files for a module are kept with the module's source files. This makes it easy to see what files comprise each module; it is much cleaner than mixing hundreds of unrelated files in one directory.
- Allows keeping Debug, Release, ROM, and other versions built simultaneously; not necessary to "clean" between these different builds.
- Avoids mixing object files with source code files.
- Allows dual-build releases (two compilers and/or processors). This makes it easier to migrate to another, if the need should arise.

Protosystem

The Protosystem is the foundation for your application. It also serves as a sample application that runs demos for the different SMX modules. Demos are added by enabling macros in the appropriate configuration file. For IDE builds, this is set in the "preinclude" files in the CFG directory. See the section for the compiler you are using in the SMX Target Guide for more information about these. Similarly, SMX modules are enabled at the top of the preinclude files.

The Protosystem is stored in the APP directory, with some common, core files in XBASE.

You should build and run the Protosystem, as shipped, before making changes to it. Run the demos provided for the SMX modules you licensed. See the Getting Started section for your tools, below, for directions to do this. Keep the APP directory pure; create copies of it (with new names) for application development and experimentation.

More information about the Protosystem is given in the SMX Target Guide, such as a listing of the core files and those that are processor-dependent. The former are in the Common Notes section; the latter are in each of the processor sections.

Demos

Demo code is important because it serves as a confidence test you can immediately build and run to verify operation of SMX modules. It also serves as example code that can teach you the basics of using a module. All demos are stored in the APP\DEMOS directory. Only the appropriate demos are included for the modules you licensed. All of this code can be discarded.

Demos plug into the Protosystem. They are enabled by uncommenting lines in the main preinclude file, e.g. CFG\iararm.h (and adding or un-excluding the source files to the project for IDE builds). Some or all demos for the SMX modules you ordered are enabled, as shipped. We suggest you build and run first without changing the configuration. Some demos cannot be run together because of competition for the screen or keyboard. Enabling one demo at a time may be a good idea to see what each does. It will then be pretty clear which demos will run together. These limitations apply only to the demos; all SMX modules work together.

RTOS Porting Files

Files in XPORT map other RTOS services onto SMX, to make it easier to migrate to SMX. These files map other RTOS services onto SMX. Currently available are FRPort for FreeRTOS (in subdir FR) and TXPort for ThreadX (in subdir TX). Add the files for the RTOS you are using to your App project. First you can run the demos in the corresponding APPxx directory (e.g. APPFR, APPTX).

Version Numbers

1. **SMX_VERSION** (in xdef.h) indicates the current version of the smx kernel. It can be used by third party developers to condition their code to support different versions of smx. **smx_Version** is an smx global variable that is initialized to this value. It is used by smxAware so it can properly display control blocks and other structures that differ between smx versions.
2. **xxx_VERSION** constants in each SMX module (e.g. smxFS, smxNS, smxUSB) serve the same purpose.
3. The version number in the comment at the top of each file indicates the version when that particular file was last modified.

Build Information

Build Versions

Build target names are typically Debug, Release, and ROM, or similar. See Build Targets in the section for the compiler you are using, in the SMX Target Guide.

Debug	No or low optimization, debug symbolics enabled, located for RAM.
Release	Max or high optimization, no debug symbolics, located for RAM.
ROM	Same as Release but located for ROM/Flash.

SMX_BT_DEBUG, **SMX_BT_RELEASE**, and **SMX_BT_ROM** are defined in the project files for the respective build targets.

Module Defines

In order to enable a module in the Protosystem, it is necessary to define the following symbols. They are already defined in the main preinclude file (e.g. CFG\iararm.h) and just need to be uncommented.

Module	Defines
smxAware	SMXAWARE
smxFLog	SMXFLOG
smxFFS	SMXFFS2
smxFS	SMXFS
smxNS	SMXNS
smx++	SMXPP
smxUSBD	SMXUSBD
smxUSBH	SMXUSBH
smxUSBO	SMXUSBO
smxWiFi	SMXWIFI
CSL_USSL	uSSL

Demos are enabled by adding another define for each. These are the same as the module defines above, but with a "_DEMO" suffix (e.g. SMXUSBH_DEMO).

Optimization

By default, project files are set to optimize for speed rather than size (for build targets that enable optimization).

Conditionals

Although preprocessor conditionals can make code harder to read, they avoid the need for us to maintain multiple versions of each file. Having to remember to make every fix and improvement to multiple copies of the same file is error prone, no matter how careful one is. Having one file is safer. If the conditionals in a particular file are distracting while you are debugging it or making modifications, we recommend that you delete the conditional sections that do not apply to your release, such as sections for compilers and processors that you are not using. Some editors allow hiding conditional sections. Refer to Common Notes/ Target Defines in the SMX Target Guide for a list of the more important defines used in conditionals. If you are in doubt about one you encounter, please ask us.

Naming Convention

In SMX code, identifiers have a 2 or 3-letter prefix indicating the module (product) they are part of, such as smx_, sfs_, and sud_, so that each has its own namespace, to avoid conflicting with your code or third-party libraries. sb_ is used for smxBase and BSP. The prefix is lower case for functions, macros, and variables. It is capitalized for constants (#defines). The underscore is used to make it convenient to search application code for all calls made to a particular module such as the smx kernel and to visually separate the prefix from the name. Searching for smx without the underscore would produce many extraneous matches. Type names are generally not prefixed, to keep the names shorter and simpler.

There are relatively few data types used in a program compared to #defines, variables, and functions, so there is not as much of a namespace issue. Structure field names are purposely kept short, which is fine since each structure is its own namespace.

Getting Started

The directions in the following sections will help you get started with your tools. However, keep in mind that these tools are always changing; new versions are frequently released and the IDE can be a bit different for each target processor. If you encounter difficulty with our directions, please call us, and we will walk you through it.

The directions here are purposely terse. The other sections of this manual and the SMX Target Guide fill in the details. See the section for your compiler in the SMX Target Guide for more information.

ARM — IAR

Tool Setup

See **IAR Embedded Workbench ARM** in the ARM section of the SMX Target Guide.

Building the Protosystem

1. Start Embedded Workbench (the IDE). (We only support building from the IDE; we do not provide makefiles to build from the command line.)
2. File | Open | Workspace. Browse to the command level subdirectory: \SMX\APP\IAR.ARM or IAR.AM. Go into the subdirectory for the board you are using and double-click on the **App_.eww** file there.
3. Edit \SMX\CFG\iararm.h to match your target (if not already set properly). This is a “preinclude” file included by the IDE ahead of each file. Changing it marks all files to be rebuilt.
4. Press the Make button.

Running and Debugging the Protosystem

1. Build the Protosystem (APP), as above.
2. Connect your JTAG unit to your target board and host. See ARM/ Tools/ JTAG Units in the SMX Target Guide for more information.
3. Connect a terminal or terminal emulator (115200-8-N-1) to the first COM port so you can see demo output from app.c.
4. Press the Debug button to download the app to the target. It should execute the startup code and stop at main().
5. Press the Go button. From there, you can step or run. If the board has LEDs, you should see them count up (in binary if it is a row of LEDs).
6. Press the Stop button to break execution.
7. When running, you can press Esc at the terminal to exit the application. This runs aexit() under the Idle task, at maximum priority. aexit() calls some exit functions, displays a message to the terminal indicating whether it is a normal exit or the error that caused the exit, then calls sb_Exit() which calls sb_Reboot(). These can be filled in with user code.
8. We recommend putting a breakpoint in **smx_EMHook**() in smxmain.c so that you will know immediately if an smx error occurs. The call stack shows how you got there.

Enabling smxAware

See the smxAware User's Guide for detailed setup information and instructions for use.

1 Copy the smxAware .dll, .ewplugin, and .exe files from \SMX\SA to arm\plugins\rtos\smx in the IAR EWARM directory. (You must create the smx subdirectory.)

2 Start Embedded Workbench. (Or exit and re-start so the DLL will be loaded.) smxAware should already be enabled in the project, but check it:
In the project Options, select Debugger in the left pane and the Plugins tab in the right pane. Put a checkmark next to smxAware in the list of plugins to load.

3 Start a debug session as usual (see previous section). A new "smxAware" entry should be added to the main menu.

Building, Running, and Debugging SMX Module Demos

1 Enable the demo(s) in CFG**iararm.h**.

2 Configure the demo(s). See the SMX Modules section of this manual.

3 Follow the same instructions as for Running and Debugging the Protosystem, above.

4 If you have difficulty, read the appropriate **.txt file in C:\SMX\DOC** for the module, if there is one. Otherwise, please ask!

What To Do Now

1 See the ARM section of the SMX Target Guide for more information about CPU and tool issues. See the IAR subsection for more information about using this compiler with SMX.

2 See the BSP notes PDF in the DOC directory for information about the board and processor.

3 Read the sections following these Getting Started sections, and begin application development.

Protosystem

The Protosystem is the foundation for your application. It builds several core application files plus BSP files, startup code, kernel, and middleware. It is stored in the APP directory.

More information about the Protosystem is given in the SMX Target Guide, such as a list of the core files and those that are CPU-dependent. The former are in the Common Notes section; the latter are in each of the CPU sections.

Project File

We intend that you use the Protosystem project file for your application. You should add your files to it and remove demo files.

Project files may have some files excluded from the build. IDEs commonly support this. It is easier to re-enable such a file than to browse to it and add it to the project if necessary in the future. You can delete files for options you did not license.

Configuration

Each SMX module (e.g. smx kernel, smxFS, etc.) has its own local configuration. There is also application configuration. This section summarizes where to find documentation about various configuration settings, and it documents smx kernel and application configuration settings.

Summary

1. Application configuration is done in **acfg.h** in the Protosystem (APP). Settings are documented below.
2. smx kernel configuration is done in **xcfg.h** (and assembly .inc) in XSMX. Settings are documented below.
3. smxBase configuration is done in **bcfg.h** in XBASE. Settings are documented in the smxBase User's Guide.
4. BSP configuration is done in **bsp.h** and **bsp.inc** in the subdirectory for your BSP. Many or most of the settings are probably already correct for your target, but check each to be sure. See the comments there and the information at the start of the BSP API section in the smxBase User's Guide.
5. SMX modules (e.g. smxFS, smxUSBH, etc.) have their own configuration files. See the SMX Modules section of this manual for the names of those.
6. Some places in the code are tagged for your attention. Search (grep) for "USER:" to find them.

Application Configuration (acfg.h)

Note that there are multiple versions of acfg.h (e.g. acfgmin.h and acfgmed.h for minimal and medium settings, respectively), and acfg.h simply selects which of these is used. To simplify, you should rename the one you use to acfg.h and delete the one(s) you are not using. References to acfg.h in the documentation mean the file you are using.

The values set here are mostly used to configure the smx kernel and are more application-dependent than those in the smx kernel configuration file, xcfg.h (see below). Most of these settings specify the number of control blocks to allocate and how big to make memory areas, such as the heap and stack pool. The smx error manager reports "OUT OF …" or "INSUFF …" if a setting is too small.

Sizes and Quantities

SMX_CFG_EB_SIZE

Number of error records in the error buffer. Error information is stored cyclically, so this determines how many errors it is possible to look back, when many have occurred.

SMX_CFG_EVB_SIZE

Size of the Event Buffer in bytes. The Event Buffer consists of variable-length records that range from 3 to 10 words (12 to 40 bytes). The larger records are for storing up to 6 parameters of SSRs or User events, which are uncommon. Increasing this value will give a longer trace in the smxAware event timelines graph and event buffer display, but consumes significant RAM and lengthens the time for upload via the debug connection.

SMX_CFG_HEAP_ADDRESS

Starting address of the main heap.

SMX_CFG_HEAP_SPACE

Size of the smx main heap. It is calculated based on the heap usage of various SMX modules such as smxFS, if present, plus additional space for your application heap requirement. Adjust that number as needed.

SMX_CFG_HEAP_ *

Other heap-related settings. See the eheap User's Guide for information about configuring the heap settings.

SMX_CFG_HT_SIZE

Number of handles in the handle table. The handle table used by smxAware to associate names with handles and pseudohandles so objects can be displayed by name or handles can be retrieved by name. It is needed only for objects with no name field in their control blocks or that have no control block, such as ISRs.

SMX_CFG_LQ_SIZE

Size of the LSR queue. This is the number of LSRs that can be enqueued in the LSR queue at one time. Each entry is a structure LQ_CELL (xtypes.h).

SMX_CFG_NUM_BLOCKS

Number of blocks of all sizes (BCBs). These are blocks associated with BCBs not raw blocks, heap blocks, stacks, or messages.

SMX_CFG_NUM_EQS

Number of event queues (ECBs).

SMX_CFG_NUM_EVGS

Number of event groups (EGCBs).

SMX_CFG_NUM_LSRS

Number of LSRs (LCBs).

SMX_CFG_NUM_MSGS

Number of messages of all sizes (MCBs).

SMX_CFG_NUM_MTXS

Number of mutexes (MUCBs).

SMX_CFG_NUM_SEMS

Number of semaphores (SCBs).

SMX_CFG_NUM_PIPES

Number of pipes (PXCBs).

SMX_CFG_NUM_POOLS

Number of block and message pools (PCBs).

SMX_CFG_NUM_STACKS

Number of stacks in the stack pool (not permanent stacks allocated from the heap). It is not necessary to allocate a stack per task. Instead, it is necessary only to allocate enough stacks for the maximum number of tasks without permanent stacks that could become ready at any time. See the One Shot Tasks chapter of the smx User's Guide for a way to write tasks to minimize this setting, by stopping rather than suspending on SSR calls, so that the stack can be given to another task while the first task is waiting.

SMX_CFG_NUM_TASKS

Number of tasks (TCBs and timeouts). This setting should be tuned close to the number of tasks, since the TCB is by far the largest control block, and smx_TimeoutLSR checks all timeouts. Tuning this setting to the number of tasks needed will reduce RAM usage and increase performance slightly.

SMX_CFG_NUM_TIMERS

Number of timer control blocks, TMRCBs.

SMX_CFG_NUM_XCHGS

Number of exchanges (XCBs).

SMX_CFG_PP_OBJ_NUM

Number of smx++ objects that can be allocated from the GlobalPool using the new() operator. Global objects (those defined at global scope or with the ::new operator) do not use blocks from GlobalPool. Only applies to smx++ users.

SMX_CFG_PP_OBJ_SIZE

Maximum size of smx++ objects that can be allocated from the GlobalPool. Only applies to smx++ users.

SMX_CFG_SA_PRINT_RING_SIZE

Size of the smxAware print ring, in bytes. This holds messages printed with sa_Print() functions, which are shown in the Print display in smxAware.

SMX_CFG_STACK_PAD_SIZE

Size of stack pads for all stacks. Must be a multiple of 4 bytes. Use during development and set to 0 for release. Stacks grow toward the pad, so the system will continue to run, as unless the stack overflows beyond the pad. If using the Cortex-M v8 MPU with SecureSMX, setting this to 0 may be preferred, to give an immediate fault at the point of overflow.

SMX_CFG_STACK_SIZE

Size of stacks in the stack pool. Must be a multiple of 4 bytes.

SMX_CFG_STACK_SIZE_IDLE

Size of idle task stack. It is also used for init and exit, which run in the context of the idle task.

Other Settings

SMX_CFG_TICKS_PER_SEC

Number of ticks per second. The BSP code initializes the hardware timer used for the tick based on this setting.

SMX_CFG_NULL_PTR_REF_CHECK

Enables code that runs in the idle task and at exit that checks to see if the word at address 0 has changed since ainit() at startup, which would indicate a null pointer was dereferenced. Set to 1 if you desire this extra checking, but only if your target has RAM at address 0. Note that RAM may be mapped to address 0 for Debug and Release build targets but for the ROM target, flash is mapped to 0, so in this case, a conditional should be used such as SMX_BT_ROM to set it to 0 or 1, as appropriate.

smx Kernel Configuration (xcfg.h)

SMX_CFG_DIAGS

If 1, extra diagnostic information is collected such as LSR queue high water mark. Setting to 0 removes this additional code and improves performance slightly. Typically this would be enabled during development and disabled for release.

SMX_CFG_ERROR_MSGS

If 1, smx error messages are enabled (see xem.c). 0 saves const memory.

SMX_CFG_EVB

If 1, the Event Buffer is present; if 0 it is not. The Event Buffer is used by smxAware to display its event timelines graph and textual event buffer.

SMX_CFG_HT

If 1, the Handle Table is present; if 0 it is not. The Handle Table associates handles with names and is required by smxAware.

SMX_CFG_HT_SCAN_DUP

If 1, smx_HTAdd() checks if the name is already in the table and reports SMXE_HT_DUP if so. This is disabled by default because it is common in SMX modules to give things the same name (such as multiple USB class driver tasks since they are all created by a general function), and to avoid this by suffixing them takes extra code and it not meaningful. Also if there are many handles, doing this search will add significant overhead to all object creation. Enable this temporarily when you want to check for duplicates.

SMX_CFG_IDLE_RTLIM

Number of idle passes per runtime limit frame.

SMX_CFG_LOCK_NEST_LIMIT

Maximum lock nesting. Set as desired. SMXE_EXCESS_LOCKS is reported if this limit is exceeded.

SMX_CFG_MACROS

If 1, use macros not functions for EVB and SSR_ENTER/EXIT, else functions are used. Macros make the code slightly faster, but functions reduce code size and make debugging easier since they can be stepped over

SMX_CFG_MULT_SSTS

If 1, allow multiple system service tables, for API calls made via SVC exception. This allows limiting the set of calls available to each partition or task. See the SecureSMX User's Guide.

SMX_CFG_MPU

If 1, Cortex-M MPU support is enabled. See the SecureSMX User's Guide.

SMX_CFG_MPU_ENABLE

Set to 0 to temporarily disable use of the Cortex-M MPU and privilege/unprivileged operation, if it is interfering with development or debugging. In this case, the MPU is never enabled, all code runs privileged, and system calls are direct, not through SVC.

SMX_CFG_PORTAL

If 1 and SMX_CFG_MPU settings are 1, portals are enabled. See the SecureSMX User's Guide.

SMX_CFG_RTLIM

If 1, enables runtime limiting of tasks.

SMX_CFG_SSCTR

If 1, enables code that keeps counters of system calls (total and each). This adds some overhead to the SVC handler. Also, it is only implemented for the case of using a single system service table (i.e. SMX_CFG_MULT_SSTS is 0).

SMX_CFG_TOKENS

If 1, tokens are enabled to prevent unauthorized use of smx objects. See the SecureSMX User's Guide.

SMX_CFG_PROFILE

If 1, profiling is enabled. Set to 0 when bringing up a new port or making time measurements. See the smx User's Guide for information about smx profiling.

SMX_CFG_RTCB_SIZE

Number of run time counter samples in smx_rtcb.

SMX_CFG_RTC_FRAME

Determines rtc frame in ticks.

SMX_CFG_SAFETY_CHECKS

If 1, additional safety checks are enabled. Setting to 0 removes this additional code and improves performance slightly. Typically this would be enabled during development and disabled for release.

SMX_CFG_SAVE_SUSPLOC

If 1, the suspend location is saved in each task's TCB, for debugging. The address stored here is where a suspended task will resume (and the previous instruction is where it suspended). It is shown in the smxAware Task display summary.

SMX_CFG_STACK_SCAN

If 1, stack scanning and clearing code is present; if 0 it is not. Scanning is the best way to determine stack usage to enable stack size tuning. This information is stored in the TCB and is displayed in smxAware graphically and textually.

SMX_CFG_PRI_UP

If 1, priorities increase numerically, else they decrease. Setting to 1 is recommended, and support for decreasing priorities was added only to help those porting to SMX from another RTOS that numbers priorities this way.

SMX_ PRI_NUM

Specifies the number of priority levels. Must be <= 127. Note that 0xFF is reserved to mean no change for some API calls.

PRIORITIES enum

These are the predefined task priority levels. Although numbers could be passed for priorities, an enum allows using meaningful names. You can add new levels < 127 (0xFF). These are used in SMX modules and the application, so that is why this is located in xcfg.h rather than acfg.h.

SMX Startup and Scheduler Operation

startup code -> main() -> smx_Go() -> smx_SchedRunTasks() -> ainit() -> tasks

1 **startup code** is usually written in assembly language. Details of routines and files vary for each board and compiler. See the section Protosystem / BSP Files in the section for your CPU in the SMX Target Guide. This code calls main().

2 **main**() calls smx_Go(). Generally, you should not change main(). Instead add code to ainit(). Prior to calling smx_Go(), interrupts are masked. The interrupt mask that was in effect is later restored by ainit() (see below).

3 **smx_Go**() initializes smx. The smx_Idle task is created and started here. Finally smx_Go() calls smx_SchedRunTasks(), in the scheduler.

4 **smx_SchedRunTasks**() is the smx task scheduler. Since smx_Idle task was set to maximum priority, it is the first to run. ainit() is its code, initially (in smxmain.c).

5 **ainit**() restores the interrupt mask that had been in effect in main() before they were masked. Normally, the startup code should have had all interrupts already masked, so they still remain masked, but if there had been a need to enable an interrupt prior to main(), this would re-enable it. (As a general rule, interrupts should be unmasked individually right after each ISR is hooked.) Then ainit() creates some tasks and calls smx_modules_init(), which performs some additional initialization of SMX modules, such as smxNS and smxUSBH. Then it calls appl_init(), which creates application tasks. These tasks do not run yet, since smx_Idle is maximum priority and it does not suspend itself (see note 4 below). The last step of ainit() is to call smx_TaskStartNew(), which sets smx_Idle's code to smx_IdleTaskMain() and lowers its priority to 0.

Important: ainit() and all routines it calls must not call SSRs that suspend, or other tasks will start running before initialization is complete. See note 4 below.

6 **tasks** Once smx_TaskStartNew() completes, the system is multitasking! The highest priority task in the ready queue is dispatched. (If there is more than one, the first task that was started is the first to run.) From this point on, the highest priority task will run. Every interrupt and every smx call designated as an SSR in the Reference Manual is an entry into the scheduler. The scheduler first runs any LSRs. If the current task is locked, execution returns to it. Otherwise, the scheduler looks to see if a higher priority task has become ready. If so, the current task is immediately suspended and the higher priority task is resumed or started.

Notes:

1. The smx scheduler (xsched.c) consists of:
   a. LSR scheduler
   b. Task scheduler
   c. smx_SSR_ENTER() and smx_SSR_EXIT() routines (begin and end all system services (SSRs))

Note that SSR and ISR exit call the prescheduler, smx_PreSched() which is written in assembly for each processor architecture to call the LSR and Task schedulers.

2. ISRs branch to the scheduler only if LSRs are waiting to run, for efficiency. (See the check of smx_lqctr in smx_ISR_EXIT.) Also, nested ISRs do not enter the scheduler, and instead return to the point of interrupt.

3. Locking is accomplished by the smx_DO_CTTEST() macro, which is used by SSRs (see xsmx.h). If the current task is locked, *smx_sched* is not set, so after the scheduler runs any waiting LSRs, the task scheduler is not entered, and instead the scheduler returns to the current task.

4. ainit() actually runs in the multitasking environment, as the idle task. It completes before any other tasks run, because it is set to the maximum priority level. However, this would not be true if idle were to suspend or stop itself by calling an SSR with a timeout. Then some other task could run before the system was fully initialized, thus causing an error. (Note that locking idle is not a solution because that does not prevent it from suspending or stopping itself.) Note that your application init in appl_init() is called by ainit(), so it also must not call SSRs that suspend or stop.

5. Setting smx_Idle task's code to ainit() and then later switching it to smx_IdleMain() demonstrates how a task's main function can be changed at any time.

SMX Modules

This section gives an overview of the various modules of the SMX RTOS, such as the kernel, file system, TCP/IP stack, USB stacks, and GUI. It summarizes the directory, demo, and configuration information, and it gives a few key tips about some modules.

Notes

1. Documentation: In addition to the PDF manual(s) for each module, check the DOC directory for a .txt file with supplemental information.

2. Demo files are stored in the APP\DEMO directory. All of these can be discarded. Demos are provided only for the modules you licensed. Select the ones to use in the “prefix” or “preinclude” file in the CFG directory. Each demo uses a different region of the terminal screen and some use the keyboard, so it may not be possible to run all demos simultaneously. Some demos are described in more detail in their manual or release notes, but a few points are mentioned below.

3. Demo configuration is documented below and in the comments at the top of the demo files. Check the comments in the demo files, in case there are new settings added since this manual was updated.

Modules

smx kernel

Directory: **XSMX**
Demo Files: **app.c** in Protosystem
Configuration: **acfg.h** in Protosystem (main cfg), **xcfg.h**

smx++ (C++ Kernel API)

Directory: **XSMXPP**
Demo Files: **sppdemo.cpp, dprocess.cpp, dprocess.hpp**
Configuration: —

smxFLog

Directory: **XFL**
Demo Files: **fldemo.c**
Configuration: **flcfg.h**

smxFFS

Directory: **XFFS2**
Demo File: **ffs2demo.c, ffs2test.c, ffs2tst1.c**

Configuration: **ffcfg.h**

smxFS

Directory: **XFS**
Demo Files: **fsdemo.c**, **fstest.c**
Configuration: **fcfg.h**

If you are using the USB disk driver, also define SMXUSBH and add/enable the smxUSBH files in the project.

fsdemo creates a subdirectory and creates many files in it. It creates files of random sizes and does random operations on them (choosing from a list), such as read, write, and truncate. It is meant to be a fairly simple example to study. It also tests performance by writing a large test file to the disk (e.g. 20 MB), and then reads it back and reports the average read and write speeds.

fstest is similar to fsdemo but tests more operations.

smxNS

Directory: **XNS**
Demo Files: **nsdemo.c**, **nstels.c**
Configuration: \XNS\include\nscfg.h.

See the smxNS User’s Manual for information about getting started and running demos.

smxUSBD

Directory: **XUSBD**
Demo File: **usbddemo.c**
Configuration: **udcfg.h**, **udport.c,h**

See the Demo Configuration section at the top of usbddemo.c to enable different demo features.

smxUSBH

Directory: **XUSBH**
Demo File: **usbhdemo.c**
Configuration: **ucfg.h**, **uport.c,h**

See the Demo Configuration section at the top of usbhdemo.c to enable different demo features.

smxUSBO

Directory: **XUSBO**
Demo File: **usbodemo.c**
Configuration: **uocfg.h**, **uoport.c,h**

smxWiFi

Directory: **XWIFI**
Demo File: **wifidemo.c**
Configuration: **wfcfg.h**, **wfport.c,h**

Third Party Modules

uSSH

Summary: SSH from Cypherbridge
Directory: **XCSL\uSSH**
Demo Files: **APP\DEMO\nsdemo.c**
Configuration: \XCSL\uSSH\options.h

uSSL

Summary: SSL/TLS from Cypherbridge
Directory: **XCSL\uSSL**
Demo Files: **APP\DEMO\nsdemo.c**
Configuration: \XCSL\uSSL\options.h, ussltask.h

Support

Support Site

Check **www.smxrtos.com/support** regularly for fixes, enhancements, and technical information. To access it, you must supply a password. You will be notified whenever it changes, if you have given us your email address and you are current on your maintenance and support contract. To get the password, email **support@smxrtos.com**. Indicate the company you work for that licensed our software and the serial number of your license / release.

Bug Fixes

As fixes are made, we post entries on the Product Fixes page of the support site. These are categorized by product, and dates are marked next to each entry to make it easy to see which are new since you last checked. Each entry is a link to more information about the fix and how to apply it. Sometimes fixed source files are provided. Contact support@smxrtos.com if you need help applying fixes. If many are needed, it might be better to request an update.

Application Development

Before you begin work on your application, please build and run the Protosystem, as shipped, as a confidence test. The project file set to build and link some or all of the SMX modules you licensed. Please follow the instructions in the Getting Started section of this manual, for your processor and tools.

Main Steps

1. Make a copy of the APP (Protosystem) directory, naming it for your application. (Keep the original, pure directory so you can do confidence tests or experimentation, in a copy of it.)
2. Replace app.c with one or more application files.
3. Configure.
4. Remove any unnecessary code and conditionals (optional).

Guidelines

1. **To allow you to easily integrate future updates of smx we suggest that you minimize modification to the Protosystem files.** Of course, you may remove any irrelevant code from them, but you should not add application code to them. **Put your code into new files.** You should tag all changes you make to SMX files.
2. We recommend putting application initialization routines into each application file. These should be called from appl_init() which, in turn, is called by ainit() in smxmain.c. Each initialization routine creates smx objects, starts tasks, etc. as needed by the code in its file. Similarly, there should be exit routines in each application file, if the application exits. These should be called from appl_exit(), which in turn, is called by aexit().

app.c

To start your application, create a new app.c like this:

```
/* app.c */

#include "smx.h"
#include "smxmain.h"

void appl_init(void)
{
}

void appl_exit(void)
{
}
```

These are the hooks for you to initialize and exit your application. Add code to appl_init() to create your main smx task(s) and other objects. You do not need to create everything here.

You can create smx objects (tasks, semaphores, exchanges, etc.) from any task, at any time, so typically, you just add code here to create the main objects, to get the system started.

Create any other files and include smx.h and smxmain.h in them. That's it!

Simplification

The Protosystem is purposely kept minimal, and demo code is separated into the DEMO directory and app.c. There is not much code in the Protosystem files, so there is not a lot to strip out. However, here are some things you can do:

- Demos should be disabled and not linked, of course.
- Replace app.c with your own (see the section app.c, above).
- Strip out conditionals for other compilers and modules (products) you aren't using. However, since you may want to update to a new version of SMX (which means moving your app to the new Protosystem), you ought to minimize this.

Coding

Refer to the example code in ESMX, and copy useful sections into your code. Start by linking esmx to the Protosystem and following the Debug Tour document there, to get familiar.

Debugging

The topic of debugging and diagnostics could easily fill a whole manual, and someday maybe it will. Until then, these are a few helpful notes:

1. **smx Errors** are listed alphabetically in the **Glossary** section of the Reference Manual, at SMXE_xxx. If an smx error occurs, look there for information about possible causes and things to try. These are kernel errors, only.
2. The Protosystem opcon task recognizes a couple keys that change the terminal display:

   Ctrl-D changes the output mode to suppress ANSI Esc sequences for cursor positioning and color, and it displays messages sequentially at the first column of the terminal. This allows capturing a clean log from the terminal program. In TeraTerm, for example, use File | Log… to set the output file name. Then terminal output will also be saved to the file. This is helpful to review and to send us for technical support.

   Ctrl-E clears the screen and displays the contents of the error buffer. Errors are displayed in red, inline with other messages in the right half of the screen, normally, but this is a way to look at the smx errors condensed. Note that the error buffer is cyclic and also may be bigger than the number of lines on the terminal, so a * marks the most recent error.

3. If you suspect an smx error is occurring but cannot tell because you have no terminal or display or it has been switched to graphics mode, you can put a breakpoint at **smx_EMHook()** in **smxmain.c**. While there you can inspect **errnum** to find out which error occurred and **smx_ct** to see which task caused it (or LSR, if **smx_clsr** is set). The call stack shows how you got there.

4. Debugging a multitasking application is more challenging than debugging sequential code. When you step over an instruction, it is possible an interrupt will occur, causing a task switch and then a return to the current task, without you being aware. It looks to you like the debugger ran only the instruction you stepped over, when, in fact, a considerable amount of other code may have run. It is easy to be misled into thinking that if something went wrong during that step, such as an smx error being flagged or a watched variable being corrupted, that the instruction you stepped over was the culprit. However, it could have been caused by an entirely different task that ran during that instant. Keep this in mind. Debugging can be further complicated if multiple tasks share the same code, since it may become necessary to determine which task is currently running. Adding smx_ct->name to the debugger Watch window is recommended.

5. **smxAware** is a big help. This is a DLL and EXE that adds smx-awareness to the debuggers we support. It allows viewing smx objects by name and setting task-aware breakpoints for some debuggers. It shows stack usages, which is a big help for catching stack overflows. Versions with GAT (Graphical Analysis Tool) allow you to view event timelines, profiling, stack usage, and memory layout. smxAware Live is a remote monitoring version.

6. **Stack Overflow** can be a difficult bug to track because the symptoms usually arise long after the corruption — often not until the task with the corrupted stack is resumed. smx helps greatly by doing automatic stack scanning and stores the number of bytes used, in the TCB (in the shwm field, meaning stack high-water mark). This information is displayed textually in the Stack window in smxAware and graphically by smxAware GAT. Stack checking is configured in **acfg.h**. Set STACK_SCAN to 1. Also, we recommend you enable stack padding during development (set STACK_PAD_SIZE) so the system will continue running if a stack only overflows into its pad.

7. Stepping over the **smx_TaskStartNew()** call at the end of ainit() causes the Protosystem to free run. This is because smx_TaskStartNew() assigns a new function to the task and restarts it using that code, so execution never returns following the call. This is true when stepping over any call to this function.

BSP API

The Board Support Package (BSP) API is a set of low-level functions that interface to the hardware, for use by SMX and the application. Primarily the API contains routines for hooking, masking, and configuring interrupts. The API is defined in XBASE\bbsp.h and implemented in bsp.c in each BSP (and XBASE\bbsp.c for some common routines). There is one bsp.c file for each board/platform supported. We intend for you to add any additional hardware initialization code to sb_PeripheralsInit(). See the BSP API section of the SMX Target Guide for detailed information.

Utilities

These are utilities that are exceptionally useful for software development. We highly recommend that you use them.

Diff

BeyondCompare (www.scootersoftware.com) is a very good utility for differencing source files. It has 2 panes that show the directory tree and allows easily navigating and opening files for side-by-side comparison, with differences highlighted. It is inexpensive and has a free evaluation period. It is easy to see which files are different and to transfer changes from one to the other, incrementally or all at once. A good use of this tool is to copy changes to your main-line code after experimenting. Rather than experiment in your working directory, make a copy of it. When you have it working, compare the two trees. You can review and transfer the changes to your main-line code individually. This is great for catching temporary changes you should have reversed.

Another use is to find the change(s) that broke the application. Restore a backup of your development directory and verify it builds and runs ok. Then open your latest working directory in one pane and the restored version in the other and transfer changes a little at a time, re-testing each time, to isolate what caused the problem.

Grep

Grep is an invaluable tool for finding things in unfamiliar code. It allows searching for a text string in all files in a directory (and even in nested subdirectories). This is especially helpful when trying to find where a function or variable is defined. The one supplied with Borland C++ is simple and works well. Dig up an old version of this compiler to get it, if you don't already have a grep utility. There are only 4 switches you need to know:

-d+	search subdirectories too
-i+	ignore case
-l+	list file names only (don't show matching lines from files)
-w+	whole-word search

Put quotes around multi-word search strings.

Shell

For command line users, we recommend you use a shell utility such as **FAR** (www.rarlab.com) rather than using the Windows command line for your build environment, since SMX has nested subdirectories, and you will quickly tire of typing the cd command to get down to the build directories. Shell utilities show what is in each directory much more cleanly than the dir command, and they are very efficient for copying and moving files and whole directories. They are far superior to Windows Explorer for this purpose, although they may not look as pretty. FAR is a clone of the venerable Norton Commander that supports long file names.

Terminal Emulator

The Protosystem assumes a terminal is connected to display messages and to take user input. (Assume 115200-8-N-1, unless told otherwise.) You can connect your target board's serial port to a spare serial port on your host system and run a terminal emulator. We recommend **Tera Term Pro**, which is easy to use, small, and free.

Tips

1. smx terminology and error messages are documented in the Glossary section of the smx Reference Manual.
2. Grep the code for “USER:” to find places where you may want to make changes. This is a convenient way for us to tag things for your attention.
3. smx kernel errors are recorded in the error buffer, and they are displayed on the terminal. Error handling code is in xem.c. You can modify it to do what you want. Put a breakpoint on smx_EMHook() in smxmain.c or smx_EM() in xem.c, and if hit, look at the call stack in the debugger to see how you got there.

Index