Micro Digital

smxBase™ User's Guide

Foundation Definitions and Routines

Version 5.4.0
July 2025

by David Moore and Ralph Moore

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

1. Overview

smxBase™ provides the base for the SMX® RTOS. It contains common definitions and routines, board and processor support (BSP) code, and core app files.

smxBase includes the following parts:

1. **Configuration**: Defines the basic software and hardware environment you are using, such as Operating System, Processor Type, and Operation Size. See bcfg.h.
2. **Data Types**: Defines the basic data types that can be used by SMX modules, such as u8, u32, and bool. Also defines some keywords that are related to the compiler you are using, such as interrupt. See bdef.h.
3. **Base and Utility API**: These macros and functions used by SMX modules, such as swapping the endianness of data and writing unaligned data. See bapi.h, bbase.c. Defines console routines used to output debug and status information. See bapi.h.
4. **CPU API**. Defines some features that are specific to different types of processors, such as enable/disable interrupts and debug trap instructions. See barmm.h, etc.
5. **BSP APIs**: Defines the BSP functions that are used directly by SMX modules, such as intalling interrupt vectors and masking/unmasking particular interrupts. See BSP API section of bapi.h.
6. **Run Time Library**: Provides some basic C run time library functions in case the compiler does not provide them, such as stricmp() and ultoa(), or they are implemented incorrectly. See brtl.c,h.
7. **Error Management**: About 15 types of errors are detected and reported for smxBase calls using sb_EM() which sets sb_errno to the error number and increments sb_errctr. It is recommended to keep a watch on sb_errno (like smx_errno for smx). Also smxAware's Diagnostic group in the Text has a section for smxBase Errors that displays the last one that occurred. There is no error buffer for smxBase errors.
8. **App Core Files**: smxmods.* and smxaware.* were moved from the App dir so they can be shared by different variants of App.

smxBase is contained in the XBASE and BSP directories. bbase.h is the master include file, which is included by SMX modules and application code. **For applications using the smx kernel, include smx.h instead, which includes bbase.h.**

2. APIs

The following sections document the more significant APIs provided in smxBase. There are some simple APIs in the BSP for writing LEDs or writing strings to the LCD on the board if present, for example. These are easily understood from their interface header files.

smxBase services are bare functions and macros that are not protected from reentrancy, so use them with caution in a multitasking environment.

2.1 Dynamically Allocated Regions (DARs)

Dynamically Allocated Regions, DARs, are the most basic memory structure for smxBase. For each DAR, there is a statically defined DAR Control Block (DCB) which has the following format:

pi	pointer to first word of DAR
px	pointer to last word of DAR
pn	pointer to next free block
pl	pointer to last block allocated

and there is a statically defined pointer to the DAR. A DAR is initialized by sb_DARInit(). The address and size can be for a large buffer (array) defined in C, as done for smx, or it can be done with addresses defined in the linker command file or absolute addresses passed in this function call.

DARs can be located in RAM wherever desired. Blocks are allocated from a DAR with the sb_DARAlloc() function. Allocations are permanent, with the exception that the last block allocated can be freed using the sb_DARFreeLast() function.

DAR API

u8*	sb_DARAlloc(SB_DCB_PTR darp, u32 sz, u32 align)
bool	sb_DARFreeLast(SB_DCB_PTR darp)
bool	sb_DARInit(SB_DCB_PTR darp, u8* pi, u32 sz, bool fill, u32 fillval)

bool **sb_DARInit**(darp,*pi, sz, fill, fillval)

Initializes the DAR control block, DCB, pointed to by darp, starting at pi and of size, sz. DAR alignment is determined by pi. pi and sz must be word multiples. dcb.pi points to the first word of the DAR, and dcb.px points to the last word. If fill is TRUE, the block allocated for the DAR is filled with fillval. This helps visual recognition of DARs, during debug, and it shows wasted space between blocks due to alignment. The latter may be reduced by changing the order of DAR allocations. If pi or sz are not word multiples, they will be adjusted, and DAR size will be less than sz, accordingly.

u8* **sb_DARAlloc**(darp, sz, align)

Allocates a memory block of sz bytes from the DAR identified by its handle, darp. If the handle is invalid, SBE_INV_DAR is reported. sz must be at least 4 or SBE_INV_SIZE is reported. The block allocated is aligned on an align bytes boundary, where align is a power of two. For example align = 4 results in 4-byte (word) alignment; align = 32 results in 32-byte alignment. Align values < 4 result in abort with an SBD_INV_ALIGN error. The maximum align value is 0x4000, which specifies 16KB alignment. Additional 1's, in align, after the first 1 are ignored. Hence, alignments that are not powers of two are reduced to the closest, smaller powers of 2 (e.g. 5 would be reduced to 4). The next boundary matching the alignment is found and a block of sz bytes is allocated from the specified DAR.

Any space left between the previous block and the new block is wasted. Hence, it is advisable to allocate blocks in decreasing order of alignment, then size, if possible. An allocated block stops at the next word boundary above sz; odd bytes are wasted.

Allocation is permanent and cannot be returned to the DAR, except as described for sb_DARFreeLast(). sb_DARAlloc() returns a pointer to the block, if successful, or NULL, if not successful. Always test that a non-NULL pointer has been returned before using it, in case the DAR has run out of space or one or more parameters were invalid. SBE_INSUFF_DAR is reported if the DAR does not have enough space.

bool **sb_DARFreeLast**(darp)

Frees the last block allocated in the DAR selected by darp and refills it from the last word of the DAR, which is expected to be the DAR's fill pattern. This function allows reversing an allocation should a later step in a process fail. For example, if an error occurs when creating a block pool, the space obtained for the pool is returned to the DAR. A NULL darp or an invalid DCB will cause this function to fail.

If called a second time, without an allocation in between, DARFreeLast() does nothing and returns FALSE. That is, it cannot be used repeatedly to free blocks in reverse order. If you need to be able to do this, save allocated block pointers in an array, then load dar->pl each time prior to calling this.

Defining and Locating DARs

DARs can be defined, as shown in the example below for IAR EWARM. This approach utilizes the linker to avoid overlapping the DAR with any other memory area. However, other approaches can be used such as hard-coding the DAR starting and ending addresses in the DCB or allocating the DAR starting address and size in the linker command file. The approach used depends upon which works best in your application.

1. Define the DAR, allocate memory, and initialize it (.c):

The following defines the size and an optional fill value for a DAR named "MDAR". A DCB is statically defined, memory for the DAR is allocated by using an array, and the

memory is assigned a section ID. The section ID is used by the linker to locate the DAR, as shown in step 2.

```
#define MDAR_SIZE = 0x100000;          /* 1 MB size */
#define MDAR_FILL =  0xA2A2A2A2;       /* unique fill value */

SB_DCB  app_MDAR;                                       /* DAR control block */
u8          mdar_mem[MDAR_SIZE] @ ".app_mdar";          /* allocate memory and section */

sb_DARInit(&app_MDAR, mdar_mem, MDAR_SIZE, TRUE, MDAR_FILL); /* initialize MDAR */
```

2. Allocate the DAR position in the linker command file (.icf):

This example places uninititialized MDAR into external RAM. It could be placed in any RAM area that is large enough to hold it.

```
do not initialize  { section .app_mdar };
Place in RAM_region  { section .app_mdar };
```

2.2 Base Block Pools

Base block pools are intended for high-speed operation, such as in ISRs and low-level device code. Each base pool is controlled by a statically-defined pool control block (PCB) as follows:

```
PCB poolA;
```

Block Pool API

```
bool      sb_BlockPoolCreate(u8* p, PCB_PTR pool, u8 num, u16 size,  const char* name);
bool      sb_BlockPoolCreateDAR(SB_DCB* dar, PCB_PTR pool, u8 num, u16 size,
                                                                u16 align, const char* name);
u8*       sb_BlockPoolDelete(PCB_PTR pool)
u32       sb_BlockPoolPeek(PCB_PTR pool, SB_PK_PAR par)
u8*       sb_BlockGet(PCB_PTR pool, u16 clrsz)
bool      sb_BlockRel(PCB_PTR pool, u8* dp, u16 clrsz)
```

bool **sb_BlockPoolCreate**(u8* p, PCB_PTR pool, u8 num, u16 size, const char* name)

Creates a block pool from a pointer (p) to a free memory area and from the address (pool) of the PCB to be used for it. The block pool will contain num blocks of sz bytes. It is the responsibility of the user to make sure that p is aligned, as desired, and that the memory area is of sufficient size for the pool. This service can be handy for creating a pool from a static area or from a block allocated from the heap:

```
u8  p[2000];
  -OR-
u8*  p = (u8*)malloc(2000);
sb_BlockPoolCreate(p, &poolA, 100, 20, "poolA")
```

creates a pool of 100 20-byte blocks, starting at p. The blocks are singly-linked into a free list starting at poolA.pn and the first word of the each block points to the next free block (not necessarily in order, by address.) p must point to a 4-byte boundary and sz must be a multiple of 4, otherwise pool creation is aborted and an error is reported. The reason for this is that free list pointers would not be word aligned, which can cause problems. Other fields in the poolA PCB specify the minimum and maximum block addresses, and block size, and number. This information is used to provide checks when blocks are released back to the pool.

bool **sb_BlockPoolCreateDAR**(SB_DCB* darp, PCB* pool, u8 num, u16 sz, u16 align,
const char* name)

Automatically allocates an aligned block pool from the specified DAR. If there is insufficient DAR, or sz or align is not a multiple of 4, pool creation is aborted and an error is reported. Calls sb_DARAlloc() if pool space has not already been allocated. If pool creation fails, sb_DARFreeLast() is called to return the pool space to the DAR.

Hence this service provides safer and more automatic operation than sb_BlockPoolCreate(), but it is more limited, since the block pool must come from a DAR.

u8* **sb_BlockPoolDelete**(PCB_PTR pool)

A pool created by either of the block pool create functions can be deleted by this function, which returns a pointer to the pool block. This pointer can be used to free the block back to the heap or to repurpose it, if it is a DAR or static block:

```
u8*  bp;
bp =  sb_BlockPoolDelete(&poolA);
```

sb_BlockPoolDelete() fails if poolA is invalid. Since PCBs are static and only heap data blocks are dynamic, sb_BlockPoolDelete() is not of great use. However, it could be useful to repurpose static or DAR blocks in order to reduce RAM requirements in scarce memory systems.

u32 **sb_BlockPoolPeek**(PCB_PTR pool, SB_PK_PAR par)

This service can be used to peek at a base block pool. Valid arguments are:

SB_PK_NUM	Number of blocks in pool.
SB_PK_FREE	Number of free blocks in pool.
SB_PK_FIRST	First free block in pool.
SB_PK_MIN	First physical block in pool.
SB_PK_MAX	Last physical block in pool.
SB_PK_NAME	Name of the pool.
SB_PK_SIZE	Size of the blocks in pool.

Returns 0 and reports SBE_INV_POOL if pool is invalid; returns 0 and reports SBE_INV_PAR if par is not recognized.

u8* **sb_BlockGet**(PCB_PTR pool, u16 clrsz)

is used to get a block from the specified pool and to clear its first clrsz bytes, up to the size of the block. Hence it will not clear beyond the end of the block. This function is interrupt-safe and can be used from ISRs. In the following example,

```
u8* bp;
bp = sb_BlockGet(poolA, 4);
```

A block is removed from poolA and its first 4 bytes are cleared (which is useful to get rid of the link address.) The address of the block is loaded into bp. bp would typically be used by application code (e.g. an ISR) to fill the block, before passing it on. sb_BlockGet() is aborted and NULL is returned if the pool is invalid (which could happen if it were not created) or if it is empty. For more reliable code, test bp before using it:

```
if (bp != NULL)
    /* fill block */
else
    /* correct problem */
```

Note that bp is aligned according to the alignment of poolA.

bool **sb_BlockRel**(PCB_PTR pool, u8* dp, u16 clrsz)

is used to release a block back to its pool, given its pointer, bp. It can fail and return FALSE if poolA is invalid or if bp is outside of the pool's memory range. bp can point anywhere within the block to be released. Hence if it were the working pointer used to unload the block, it need not be reset to the start of the block, in order to release it. In the following example,

```
sb_BlockRel(poolA, bp, 20);
```

bytes 4 thru 19 will be cleared (the first word of the block is used for the free list link). poolA.pn is set to point to the block that bp points to and it is set to point to the block that pn was pointing to. Blocks are typically not returned in the reverse order that they were obtained. Hence, over time, the free list will become scrambled and bear no relationship to the block order in memory. (This can be disconcerting when tracing a block free list via a debugger.) Note also that the next Get() will get the last released block, which improves cache performance.

sb_BlockGet() and sb_BlockRel() are interrupt-safe. This means that either can be used at the same time from different ISRs on the same pool. So, for example, ISR1 could get a block from poolA at the same time that ISR2 was returning a block to poolA. Note that other base pool services are not interrupt-safe. Pools should not be created, nor deleted from ISRs.

2.3 Time Measurement Functions

The smxBase Time Measurement functions permit precise time measurements. These functions use the tick timer counter so that an additional counter is not required. Times are reported in counts, and resolution is determined by the clock used for the tick timer. Hence, resolution may be as fine as one instruction clock or it may be many instruction clocks. The variable sb_ticktmr_cntpt can be used to determine resolution in usec. For example, if sb_ticktmr_cntpt = 500,000 and sb_ticks_per_sec = 100, then counts per second = 50,000,000, so the resolution is 0.02 usec. Delays up to one tick can be measured. The current value of the tick counter is referred to as *ptime*. If it is a down counter, instead of an up counter, ptime = sb_ticktmr_cntpt - counter.

The macros, shown with the functions below, provide a convenient method for inserting or not inserting TM function calls into the code. If SB_CFG_TM = 1 (bcfg.h), TM functions are inserted when compiled and if SB_CFG_TM = 0 they are omitted. In order to get precise time measurements, it is necessary to inhibit ISRs and LSRs from running. This can be done by calling sb_IRQsMask() before time measurements start, and calling sb_IRQsUnmask() after they have ended. Although the TickISR will not be running, the tick counter will operate normally.

The following functions permit multiple simultaneous time measurements. However if time measurements overlap, then included TM functions will add overhead to them. The added time, for each included TM function, is on the order of sb_TMCal and may not be significant.

void **sb_TMInit**(void) **sb_TM_INIT**()

Calls TMStart() immediately followed by TMEnd() in order to determine their overhead and loads that correction into sb_TMCal, which is used by subsequent TMEnd() calls. This function is called during initialization and need not be called again. It must be called before using the other TM functions.

void **sb_TMStart**(u32* ts) **sb_TM_START**(ts)

Called at the start of a time measurement. Stores ptime in ts.

void **sb_TMEnd**(u32 ts, u32* tm) **sb_TM_END**(ts, tm)

Called at the end of a time measurement. Reads ptime subtracts ts, corrects if negative, and corrects for overhead by subtracting sb_TMCal. Resulting count is stored in *tm.

TM_START()s and TM_END()s can be placed throughout the code. Use separate ts's for separate simultaneous time measurements. One ts count may be used by many ends, reflecting different paths through the code. Once the last end is passed, the ts count can be reused. Results are most conveniently stored in an array, which can be examined through the debugger or uploaded to a spreadsheet. For example:

```
u32   stma[] =  /* scheduler time measurements array */
    {
      0,    /* 0 stop */
      0,    /* 1 continue */
      0,    /* 2 suspend */
      0,    /* 3 resume */
      0,    /* 4 start */
      0,    /* 5 autostop */
      0,    /* 6 timeout overhead per pass */
    };
```

The accuracy of TM functions is directly related to the accuracy of the tick and it can be verified simply by counting ticks over many seconds and comparing the final count to a stopwatch. Normally, one's eye-to-thumb response adds about 0.7 second, so measure over 100 seconds to achieve 1% accuracy.

smx_etime can be used for longer time measurements. In that case, resolution will be one tick. If better accuracy is desired for measurements longer than a tick but less than 100 ticks, we recommend implementing the TM functions using another timer on the processor chip or re-purposing the tick counter for longer time measurements.

2.4 Message Display Functions

A message display manager is implemented in bmsg.c for use by all SMX modules. Functions are provided to allow outputting error, warning, and status messages simply by indicating the message type and string. There are no parameters to specify details of message formatting such as location, color, etc. By default, they are displayed in the right half of the terminal screen. They could be reimplemented to go to any type of device such as an LCD, disk, etc.

Because UARTs are slow, it was necessary to de-couple these functions from the UART driver. To achieve this, an output message buffer (OMB) is implemented at sb_omb. It queues messages until they are output to the UART by the idle task or other low priority code. Messages pending in the OMB, when stopped in the debugger, appear in the Print section of the smxAware text displays, following the messages that were output to the console.

The OMB is necessary for variable messages, because the temporary buffers in which they are created may be re-used to create new messages before the old messages have been sent out via the UART. Variable messages are copied into the OMB, when completed, so they are not lost.

Note that because the OMB has a fixed size, SB_SIZE_OMB in bcfg.h, it can lose messages during times of peak activity. This is indicated by display of a special character as discussed in sb_MsgDisplay(), below.

The main feature of the display manager is to store messages and decouple message output from the UART driver, to achieve the following objectives:

1. To support polling UART drivers that send messages to terminal emulators.

2. To defer message output to a low-priority task or code.
3. To be able to record events in ISRs and critical code for later display.
4. To prevent message conflicts without using semaphores or mutexes in drivers.

As a result, code that outputs error or trace messages is impacted minimally.

void **sb_MsgDisplay**(void)

Displays all messages in OMB, starting with the oldest. Normally it is called only from the idle task. Can also be called prior to a breakpoint or whenever else it is desirable to display messages. However, it is **not safe to call it from ISRs**, since it calls SSRs. Also, if called from a higher priority task than idle, it may abort and do nothing if idle is already in this function, and message display will continue when idle resumes. This is discussed more below.

Loads parameters and calls sb_ConWrite functions to output each message to a terminal emulator via a UART or to the local CRT if on a PC. If smx is present, messages are also loaded into the smxAware print ring and shown in the Print section of the text displays. Messages pending in OMB are also shown there. Note that the sb_ConWrite functions can also be directly used by the application, so if smx is present, they are protected by a mutex. The mutex is created with priority inheritance enabled to avoid priority inversion.

If the MLOST flag is set in the header of a message, a “B” is displayed in the rightmost column of that message to alert the viewer that one or more messages after this message were not loaded into OMB and thus have been lost. However, in the case of error messages, their error records can be retrieved from EB and EVB.

Currently this function displays messages in the right panel (half of the screen) and recognizes three types of messages, which are displayed in different colors, as follows:

Error	light red
Warning	yellow
Information	green

Each message is displayed on a new line in the right half of the terminal. Messages may be any length, but they must end in NUL. Long messages will be wrapped into as many lines as necessary. When the end of the panel is reached, display restarts at the top. A marker (*) in the column to the left of the message marks the newest message on the screen.

This function is interrupt-safe while accessing OMB and is protected against reentry. (Attempted reentry results in a nop, but all messages in OMB will be displayed anyway, so nothing is lost. To change this behavior so message display resumes immediately, the inuse flag could be replaced with a mutex that supports priority inheritance.) Interrupts are enabled the rest of the time to permit a polling UART driver to be used without impairing interrupt latency. However, in a non-multitasking system other operations will be blocked until display of all messages in

OMB is complete. This can be a long time (e.g. 2.8 msec, at 115,100 baud, per 40-character message. If that is problem, an interrupt-driven UART driver should be used.

void **sb_MsgOut**(u32 mtype, char* mp)

Loads the message at mp into OMB (sb_omb), if space permits. If there is not enough space for the message, it is not loaded, sb_ombfull is set, and the MLOST flag is set in the last message loaded into OMB. No further messages will be accepted by OMB until sb_MsgDisplay() has run and reset sb_ombfull.

2.5 Utility Macros and Functions

bapi.h and bbase.c provide some useful utility macros and functions that can be used in your code, such as endian conversion, min/max, and read/write unaligned data.

sb_BCD_BYTE_TO_DECIMAL(num)	convert BCD byte to decimal value
sb_DECIMAL_TO_BCD_BYTE(num)	convert decimal value to BCD byte
sb_INVERT_U16(v16)	swap the endianness of 16-bit data (reverses order of bytes)
sb_INVERT_U32(v32)	swap the endianness of 32-bit data (reverses order of bytes)
sb_MIN(a, b)	returns minimum of two values
sb_MAX(a, b)	returns maximum of two values
sb_LOU16(l)	returns low 16 bits of a 32-bit value
sb_HIU16(l)	returns high 16 bits of a 32-bit value
sb_MAKEU32(h, 1)	generate 32-bit value from high and low 16-bit data
sb_read32_unaligned(a)	read 32-bit data from unaligned address
sb_write32_unaligned(a)	write 32-bit data to unaligned address

ALIGN functions: Some processors, such as ARM, require a 32-bit value to be read/written on a 32-bit (4-byte) boundary. For example, attempting to write at an address ending in 0x5 will result in it writing to address 0x4. These macros and functions allow reading/writing the value from/to any boundary. The macro calls the function for processors that have this requirement. For other processors that allow writing to a 1-byte boundary, the macro does not call the function; it just returns val.

smxAware Print buffer function prototypes are in bapi.h. See the smxAware User's Guide.

2.6 CPU Macros

The following macros are related to the processor architecture (i.e. ARM, CF, etc). They are implemented in the CPU header files in the XBASE directory, e.g. barmm.h. They are simple macros usually implemented as a small number of inline assembly statements.

sb_DEBUGTRAP()

Halts the debugger using the opcode used for a software breakpoint. This is not available in all versions; check the CPU header file for your version. This can be used in error checks so that when debugging, the debugger will stop right in the place where the failure occurred.

sb_HALTEXEC()

This uses the CPU halt instruction. If an interrupt can bring the processor out of a halt, this macro does an infinite loop with the halt instruction. This is not available in all versions; check the CPU header file for your version.

sb_INT_DISABLE()

Disables interrupts at the processor by clearing (or setting) the processor's interrupt flag.

sb_INT_ENABLE()

Enables interrupts at the processor by setting (or clearing) the processor's interrupt flag.

sb_SVC()

Does an SVC() exception to call an API from umode (SecureSMX).

sb_IN_UMODE()

Returns TRUE if the processor is executing in umode (SecureSMX). Can be used in common code to check whether to do a direct API call or call via SVC exception.

2.7 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 unmasking interrupts. This API is common to all versions of smx. The key point is that the variables and function parameters and returns are the same for all platforms.

This section documents the BSP API and explains what you should do if you are creating a new port. This API is defined in XBASE**bapi.h**. The BSP variables and functions are implemented in **bsp.c** in the BSP directory (**bspm.c** for all ARMM) and XBASE**bbase.c**. bbasse.c contains a few functions that are the same for nearly all BSPs or at least for a processor architecture, to save repeating the code needlessly in each bsp.c. There is typically a separate bsp.c for each CPU, since the goal is to minimize use of conditionals in these files to keep them simple. The same bsp.c can typically support any board that has a particular CPU because with SoCs usually all of the peripherals that matter to the BSP are part of the CPU (e.g. interrupt controller and timers).

Platform-specific variables and functions are defined in the local bsp.h file, not in bapi.h. Such functions are implemented near the end of bsp.c. (Also at the end of bsp.c is a section for local

functions used by bsp.c itself.) Add any new variables and functions you need to bsp.h and bsp.c, not to bapi.h. Platform-specific extensions to this API are documented in BSP API Extensions in the previous sections for each CPU.

Processor vendors typically provide BSP code, so you should acquire that and then decide whether to map to their functions or put code inline in each function.

Configuration Constants

Configuration constants are in **bsp.h** and **bsp.inc**.

IRQ Numbering Convention: Notes at the top of bsp.c document the IRQ numbering convention used by the BSP. We use the term "IRQ" to designate hardware interrupts, which are a subset of the full interrupt space. The number of IRQs and numbering scheme vary per target. Number starting at 0, 1, or whatever makes sense for your target. Usually there is an interrupt mask register, and numbering the IRQs to match the bit positions in it is typically a good choice — that is what we have usually done in our own bsp.c files. The comment should point to the relevant table/figure in the processor manual.

SB_CPU_HZ, **SB_CPU_MHZ**: Set these to the speed the CPU runs at internally. This is the clock rate it uses for executing instructions, not the clock rate of the internal peripheral bus. Older BSPs have only the MHZ setting; newer ones have the HZ setting and MHZ is derived from it. Also, since the conversion to MHZ is done using division, different versions of the MHZ macro are provided that round differently (DOWN, RND, or UP).

SB_IRQ_MIN, **SB_IRQ_MAX**, etc: Set these appropriately for your target. See IRQ Numbering Convention above, for the distinction between IRQ and interrupt numbering.

Other settings vary for each BSP. See the comments next to each for discussion.

Configuration Data

Configuration data are in **bsp.c**.

sb_ticktmr_: These constants characterize the timer used for the smx tick, which is used for smx profiling, event buffer timestamps, time measurement routines, and polling delay routines.

> **sb_ticktmr_clkhz**: The frequency of the clock input to the timer, after any prescalers.
>
> **sb_ticktmr_cntpt**: The number of counts per rollover of the tick timer. Assuming the timer is 0-based, this would be 1 greater than the timer's maximum value (i.e. what is loaded into the timer "reference" or "modulus" register during initialization).

irq_table[]: An array that stores IRQ priority, interrupt vector number, and any other details related to configuring IRQs. Centralizing this information helps prevent double-assignment of interrupt priorities and vector numbers, and it simplifies the parameter lists of some API calls. Any BSP function that needs this information just references irq_table[]. On many targets, the IRQs map onto a contiguous range of interrupt numbers, starting at some base. In such a system, the mapping is simple, so you do not need a vector number field. Your IRQ_REC structure may have just one field, to indicate priority. If there are multiple interrupt controllers that are different, define a table for each (e.g. irq2_table[], etc). The following is an example of a simple irq_table[]:

```
typedef struct
{
  u8  pri;     /* interrupt priority (0) */
  /* no need for vector number since easy to calculate from IRQ */
} SB_IRQ_REC;

SB_IRQ_REC irq_table[SB_IRQ_NUM] =
{
  /* pri      IRQ  Summary */
  /* ---      ---  ------- */
  { 99 },  /*   0  Watchdog Interrupt (WDINT) */
  { 99 },  /*   1  Reserved for Software Interrupts only */
  { 99 },  /*   2  Embedded ICE, DbgCommRx */
  { 99 },  /*   3  Embedded ICE, DbgCommTx */
  {  0 },  /*   4  Timer 0 (Match 0-1 Capture 0-1) */ /* used for smx tick */
  {  2 },  /*   5  Timer 1 (Match 0-2 Capture 0-1) */
  {  3 },  /*   6  UART 0 (RLS, THRE, RDA, CTI) */
  {  3 },  /*   7  UART 1 (RLS, THRE, RDA, CTI, MSI) */
  { 99 },  /*   8  PWM 0 & 1 (Match 0-6 Capture 0-1) */
  { 99 },  /*   9  I2C 0 (SI) */
  …
}
```

The following is a more complex example:

```
typedef struct
{
  u8  il;          /* interrupt level (0-7, 0 means no interrupt) */
  u8  ip;          /* interrupt priority (0-3, within interrupt level) */
  u8  avec;        /* 1 is autovector; 0 is not */
  u8  vecnum; /* vector number in EVT for IRQ (no simple mapping from IRQ to vector) */
} SB_IRQ_REC;

SB_IRQ_REC irq_table[SB_IRQ_NUM+1] =
{
  /* il, ip, avec, vecnum        IRQ  Summary */
  /* --  --  ----  ------        ---  ------- */
  {  99,  0,   0,     0   },  /*  0   -- */
  {  99,  0,   0,     0   },  /*  1   External Priority Level 1 / External IRQ1 */
  {  99,  0,   0,     0   },  /*  2   External Priority Level 2 */
  {  99,  0,   0,     0   },  /*  3   External Priority Level 3 */
  {  99,  0,   0,     0   },  /*  4   External Priority Level 4 / External IRQ4 */
  {  99,  0,   0,     0   },  /*  5   External Priority Level 5 */
  {  99,  0,   0,     0   },  /*  6   External Priority Level 6 */
  {  99,  0,   0,     0   },  /*  7   External Priority Level 7 / External IRQ7 */
  {  99,  0,   0,     0   },  /*  8   Software Watchdog Timer Timeout */
  {   6,  3,   1,    30   },  /*  9   Timer 1 */
```

```
  {  5,  3,   1,    29   },  /* 10   Timer 2 */
  {  99,  0,   1,    0   },  /* 11   MBUS (I2C) */
  {  4,  3,   0,    50   },  /* 12   UART 1 */
  …
};
```

99 means an unused row. Different values are used in different BSPs, depending on the interrupt controller. The value chosen just has to be out of the range of possible priority values.

Functions

Below is a summary of the smx BSP API functions, as defined in XBASE**bapi.h**. Most are required.

Interrupt

sb_IntCtrlInit()
sb_IntStateRestore(prev_state)
sb_IntStateSaveDisable()
sb_IntTrapVectSet(int_num, isr_ptr)
sb_IntVectGet(int_num, extra_info)
sb_IntVectSet(int_num, isr_ptr)
sb_IRQClear(irq_num)
sb_IRQConfig(irq_num)
sb_IRQEnd(irq_num)
sb_IRQMask(irq_num)
sb_IRQToInt(irq_num)
sb_IRQUnmask(irq_num)
sb_IRQVectGet(irq_num, extra_info)
sb_IRQVectSet(irq_num, isr_ptr)
sb_IRQsMask()
sb_IRQsUnmask()
sb_ISRInstall(irq_num, par, fun, name)
sb_ISRRestore(irq_num)

Memory

sb_DMABufferAlloc(num_bytes)
sb_DMABufferFree(buf)

Time

sb_ClocksInit()
sb_DelayMsec(num)
sb_DelayUsec(num)
sb_StimeSet()
sb_TickInit()
sb_TickIntEnable()

Misc

sb_ConsoleInInit()
sb_ConsoleOutInit()
sb_EVBInit()
sb_Exit(retcode)
sb_PeripheralsInit()
sb_PtimeGet()
sb_Reboot()

Notes about the API reference below:

1. Functions that return bool return TRUE for success; FALSE for fail. Other return values are explained in the descriptions.
2. See “IRQ Numbering Convention” above for the meaning of “IRQ”.

Interrupt Handling Functions

bool **sb_IntCtrlInit**(void) — Required for some targets.

Initializes interrupt controller/dispatcher, if necessary. For example, on many ARM processors, all interrupts go to one software dispatcher routine which calls the appropriate user ISR. For those ARM processors, this routine sets up some data structures needed by the dispatcher and hooks the dispatcher. This is not where to hook vectors; do that in sb_PeripheralsInit() or in other initialization code. This routine must be called before hooking any interrupt vectors. Called by ainit().

void **sb_IntStateRestore**(CPU_FL prev_state) — Required

This restores the processor interrupt state (flag) to what it had been before sb_IntStateSaveDisable() was called. The interrupt flag bit(s) is usually in a processor Flags register, so the typical operation of this is to restore the Flags register to the value passed in. Note that this also restores the other flags to the state they had before.
Alias: sb_INT_ENABLE_R(s) so it stands out and is found in searches for sb_INT_ENABLE.

CPU_FL **sb_IntStateSaveDisable**(void) — Required

This function is used to disable interrupts before a short critical section of code. It saves the processor interrupt state, disables interrupts, and returns the saved value. This differs from calling sb_INT_DISABLE() and sb_INT_ENABLE() (see CPU Macro API) because it takes into consideration whether interrupts were already disabled before sb_INT_DISABLE() was called. In that case, it is not desirable to re-enable interrupts at the end of the critical section. The caller must save the value returned to pass to sb_IntStateRestore(). Returning the value to the caller rather than saving it in a global variable is done to permit nesting of calls to this routine. For example, the code in the critical section may call a function that also calls this routine, and so on. Each will save the previous state separately rather than overwriting a single global variable. As the call chain is unwound each restores the state to the value it saved.
Alias: sb_INT_DISABLE_S(s) so it stands out and is found in searches for sb_INT_DISABLE.

bool **sb_IntTrapVectSet**(int int_num, ISR_PTR isr_ptr) — Required for some targets.

Does the same as sb_IntVectSet() but for trap vectors, assuming your target requires special handling for trap vectors. For example, on x86 protected mode systems, it is necessary to set the descriptor gate type differently for a trap than an interrupt. If your target does not distinguish between trap and interrupt vectors, just map this function onto sb_IntVectSet(), or don't implement it. int_num is the interrupt number not IRQ number; see IRQ Numbering Convention near the start of the BSP API section. isr_ptr is the address of the ISR to hook to this trap vector.

ISR_PTR **sb_IntVectGet**(int int_num, u32* extra_info) — Required for most targets.

Returns the address of the ISR hooked to a particular interrupt level, i.e. the address stored in the interrupt vector table for the specified interrupt number. It should work for all interrupts (software and hardware). It is required for targets that support software

interrupts. int_num is the interrupt number not IRQ number; see IRQ Numbering Convention near the start of the BSP API section. The extra_info parameter is a means for the routine to return additional information. For example, in x86 32-bit protected mode, it is used to return the segment selector of the ISR address. This parameter is a pointer to a u32. If 0 is passed it is not used.

bool **sb_IntVectSet**(int int_num, ISR_PTR isr_ptr) — Required for most targets.

Sets an interrupt vector to the address of the ISR specified. It should work for all interrupts (software and hardware). It is required for targets that support software interrupts. int_num is the interrupt number not IRQ number; see IRQ Numbering Convention near the start of the BSP API section.

bool **sb_IRQClear**(int irq_num) — Recommended

Clears/Acknowledges the interrupt in the device and/or interrupt controller(s), to shut off generation of a particular interrupt. For some processors, this is handled automatically by the hardware in the process of dispatching the interrupt. For others, this routine must be called to prevent the same interrupt from occurring repeatedly. This should be called near the top of the ISR, following smx_ISR_ENTER(), before interrupts are enabled.

bool **sb_IRQConfig**(int irq_num) — Required for targets that need irq_table[]

Configures an IRQ to the settings in irq_table[]. Each target has different fields in each entry of this table — whatever makes sense for that particular target. For example, some ColdFires have interrupt level, interrupt priority, autovector set/unset. This function writes the appropriate hardware register(s) to put the settings into effect. This routine is only implemented for targets that allow configuring the interrupt priority.

bool **sb_IRQEnd**(int irq_num) — Recommended

Signals End Of Interrupt (EOI) to the device and/or interrupt controller(s), to reenable generation of a particular interrupt. For many targets, it is necessary only to reenable the interrupt in the appropriate device register; it is not necessary to also reenable it in the interrupt controller. PCs require both. Note that some drivers may send the EOI to the device, in which case this routine needs only to send an EOI to the controller, if that is required for the target. Using a switch() statement, each IRQ can be handled as appropriate. This should be called in your ISR only at a point where it is safe for another of the same interrupt to be generated, typically near the bottom of the ISR.

bool **sb_IRQMask**(int irq_num) — Required

Masks the specified IRQ in the hardware mask register, to disable a particular interrupt source. This is a simple operation if the IRQ numbering convention is based on the numbering of interrupt sources in the mask register.

bool **sb_IRQUnmask**(int irq_num) — Required

Unmasks the specified IRQ in the hardware mask register, to enable a particular interrupt source. For some targets, such as the ColdFire 5272, the mask actually has a few bits per IRQ, and these indicate a priority. For targets like it, this function should get the priority

from irq_table[]. Call this after hooking and configuring the IRQ (with sb_IRQVectSet() and sb_IRQConfigVect()).

bool **sb_IRQsMask**(void) — Required

Masks all hardware interrupts (IRQs), but first saves the mask in one or more global variables in bsp.c. This variable is used by sb_IRQsUnmask() to restore the mask. Saving the mask in global variables is non-reentrant, but that is ok since this routine masks all interrupts, so no task switch will occur due to an interrupt. A task switch could occur due to an smx call, but this function is intended only to be for short critical sections, not across operations that could cause a task switch.

Typically, masking all interrupts is done by setting all bits in the mask register to 1. A nice feature on some processors is the use of a single bit in this register to mask all interrupts. In the best implementations, this bit does not change the mask bits, so it is not necessary to save and restore the mask value; only the global mask bit needs to be cleared to re-enable them. When implementing this routine for a processor with a mask all bit, check whether the mask bits are changed or not. This function is called by main() before it calls smx_Go(). You may call it in your code too, but note that calls to it cannot be nested.

bool **sb_IRQsUnmask**(void) — Required

Unmasks all IRQs that had been unmasked prior to the call to sb_IRQsMask(). Disables interrupts with sb_INT_DISABLE(), restores the mask to the value that was saved by sb_IRQsMask(), and reenables interrupts with sb_INT_ENABLE(). This is called in ainit() to restore the interrupt mask to the state it was in before smx_Go() was called. You can use it too, but note that calls to it cannot be nested.

int **sb_IRQToInt**(int irq_num) — Required

Converts an IRQ number to the corresponding interrupt number. For many targets, this is determined by a simple calculation. This is the case if IRQs generate a contiguous range (or even a few ranges) within the overall interrupt space. Otherwise, if the interrupt number for some or all IRQs can be individually set, irq_table[] structures should have a vecnum (vector number) field, and irq_table[irq_num].vecnum should be returned. Called by other BSP functions.

ISR_PTR **sb_IRQVectGet**(int irq_num, u32* extra_info) — Required

Returns the address of the ISR hooked to a particular IRQ. Generally, this simply calls sb_IntVectGet() with the IRQ number converted to the interrupt number. The extra_info parameter is a means for the routine to return additional information. For example, in x86 32-bit protected mode, it is used to return the segment selector of the ISR address. This parameter is a pointer to a u32. If 0 is passed it is not used.

bool **sb_IRQVectSet**(int irq_num, ISR_PTR isr_ptr) — Required

Sets an IRQ vector to the address of the ISR. Generally, this simply calls sb_IntVectSet() with the IRQ number converted to the interrupt number. After calling this, it is also necessary to configure and unmask the IRQ. Here is the sequence to hook an interrupt:

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

Note that for some processor architectures, MyISR in the call above must be an assembly shell that does smx_ISR_ENTER/EXIT() and calls the body of the ISR written in C. See the Architectural Notes subsection of the section for your processor in the SMX Target Guide for details.

Alternatively, sb_ISRInstall() can be used instead of the first two calls. See the description of it below and section 4.4 Interrupt Service Routines (ISRs) for details.

bool **sb_ISRInstall**(int irq_num, uint par, ISRC_PTR fun, const char* name)
— Already implemented

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

```
sb_ISRInstall(IRQ_NUM, par, MyISR, “Name”);
sb_IRQUnmask(IRQ_NUM);
```

bool **sb_ISRRestore**(int irq_num) — Already implemented

Restores an interrupt vector to what was present before sb_ISRInstall() was called.

Memory Functions

void* **sb_DMABufferAlloc**(uint num_bytes) — Required

Allocates a buffer that can be used for DMA operations. A DMA buffer must be contiguous physical memory, and on some targets, it must be within a certain address range. Since smx does not support virtual memory, the first requirement is met by the smx heap, so for most targets, this function can be implemented with just a call to smx_HeapMalloc(). On a PC, a DMA buffer must be below the 16 MB point in memory, so an additional check is needed. Returns a pointer to the buffer allocated.

bool **sb_DMABufferFree**(void* buffer) — Required

Frees a buffer allocated by sb_DMABufferAlloc(). For most targets, this function can be implemented with just a call to smx_HeapFree(). buffer is the address returned by sb_DMABufferAlloc().

Time Functions

bool **sb_ClocksInit**(void) — May be Required

Initializes PLL(s) and/or other multipliers and dividers used to control CPU core and bus clock frequencies. This is not intended to initialize a real-time clock, watchdog, or other peripherals. This should be called early in the assembly startup code. It is not necessary to implement this routine if the startup code already handles this.

void **sb_DelayMsec**(u32 num) — Already implemented

Macro implemented simply as sb_DelayUsec(1000 * num).

Caution: Avoid overflow. This is only intended for short delays (< 1 sec). If used for longer delays, look at the implementation of sb_DelayUsec() to ensure the multiplication does not overflow. Also remember that this is not a precise delay and the **imprecision is magnified** for longer delays.

void **sb_DelayUsec**(u32 num) — Often Required

Simple delay function that waits at least as many microseconds as the number specified. It is intended for use during hardware initialization, such as in sb_PeripheralsInit(), to wait some specified time before checking a bit or continuing. In our BSPs, it is implemented to read a hardware timer counter to do the delay. It uses the same timer used for the smx tick, to maximize the number of timers available to your application. It should not require interrupts to be enabled. In an smx system, after initialization, you should normally use smx services to do delays, so that other tasks can run while one is waiting. This function can be tested by doing a long delay (e.g. 10 sec == 10,000,000). It should produce a delay that is slightly longer. This function cannot be used until after sb_TickInit() has been called, except if 0 is passed for num.

Tip: For a very short delay, 0 can be passed for num, which will delay briefly. In this case, the delay will just be the time for the call, prolog, a few instructions, epilog, and return. This technique can be used even before sb_TickInit() has been called.

u32 **sb_PtimeGet**(void) — Required for uses listed below.

Returns precise time, the counter of the hardware timer used to generate the smx tick. This is used by time measurement functions in smxBase, smx profiling, event buffer timestamps, and polling delays. The value returned is 32 bits and should start at bit 0 with no extraneous bits set. Shift and mask it if necessary. It must count up, so for hardware timers that count down, return (sb_ticktmr_cntpt-1 - timer_value).

bool **sb_StimeSet**(void) — Required

Sets smx_stime, the smx global system time variable, in UNIX/ANSI format. smx_stime will be the number of seconds elapsed since Jan 1, 1970 00:00:00. This format is used by all ANSI C Time and Date routines. See the example in XBASE\bbase.c. smx_stime can be set to 0 if calendar time is not needed. Then, the timeout for smx_TaskSleep() and smx_TaskSleepStop() will be relative to when the application started running. See the smx User's Guide for more information about stime. Called by ainit().

bool **sb_TickInit**(void) — Required

Initializes a hardware counter or timer to generate a tick interrupt, from which all smx timing is derived. The tick rate is specified by SB_TICKS_PER_SEC in bsp.h. Also hooks the timer's interrupt vector to smx_TickISR(). Other timers should be initialized in sb_PeripheralsInit(). Also, the counter is used by sb_PtimeGet() which is used by sb_DelayUsec(), smx_EVB (event buffer), smx_RTC (profiling), and smx_TM (time measurement) routines. Called by main() so these things that use the counter can be used early. If needed earlier, the call could be moved to the startup code, but mask the interrupt or don't enable here but in sb_TickIntEnable().

bool **sb_TickIntEnable**(void) — Required only in special cases

Enables generation of tick interrupt. Normally this is done by sb_TickInit(). This function is only needed in the case that the tick cannot be masked by sb_IRQMask() or sb_IRQsMask() because it is not in the range of IRQs or for some other reason. This is true for ARM-M, for example. For other cases it does nothing. Called by ainit().

Misc Functions

bool **sb_ConsoleInInit**(void) — Optional

Initializes the console input device, if SB_CFG_CON is 1 in XBASE\bcfg.h. For a terminal, this would initialize the UART for input. Initialization may be done in sb_PeripheralsInit(), in which case this function is unneeded. This is separate from sb_ConsoleOutInit() and sb_PeripheralsInit() in case these operations must be called at different points. For example, a keyboard may require an ISR to be hooked and unmasked but direct screen writes for output do not and can be enabled earlier. Called by ainit().

bool **sb_ConsoleOutInit**(void) — Optional

Initializes the console output device, if SB_CFG_CON is 1 in XBASE\bcfg.h. For a terminal, this would initialize the UART for output. Initialization may be done in sb_PeripheralsInit(), in which case this function is unneeded. This is separate from sb_ConsoleInInit() and sb_PeripheralsInit() in case these operations must be called at different points. For example, a keyboard may require an ISR to be hooked and unmasked but direct screen writes for output do not and can be enabled earlier. Called by ainit().

bool **sb_EVBInit**(void) — Required if Event Buffer enabled.

Initializes global sb_ticktmr variables, if this must be done dynamically for a particular target. Otherwise, the BSP should statically initialize these variables and make this a null function that returns TRUE. See the smxAware User's Guide for discussion of these variables, in section Graphical Analysis Tool/ Application Preparation/ Event Timestamps. Called during initialization by smx_EVBInit() which is called by smx_Go().

void **sb_Exit**(int retcode) — Optional (stub required)

Exits as appropriate for your system. Typically goes into an infinite loop or can be made to restart the application by calling sb_Restart(). Called by smx_ExitTaskMain().

bool **sb_PeripheralsInit**(void) — Required

Initializes peripherals such as Timers, UARTs, and LEDs. Peripherals such as Ethernet controllers, USB, etc. that are supported by other SMX modules (smxNS, smxUSB, etc.) are initialized in the drivers in their respective libraries, so nothing is needed here. You can add code to initialize your peripherals here. Called by ainit().

void **sb_Reboot**(void) — Optional

Reboots the system by resetting the processor. The default implementation in most BSPs is to infinite loop.

2.8 Console I/O

bapi.h and opcon_main() in app.c define the console I/O functions, which usually are implemented for a terminal via a UART.

The API and color constants are defined in XBASE**bdef.h**. F_color and B_color are foreground and background color, respectively. The supported colors are:

BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, WHITE

The underlined colors are the only ones available on a terminal; the others map onto the closest of these. All are prefixed with SB_CLR_.

The console output functions are implemented in XBASE**bcon.c**. sb_ConInit() is called early in smx_Go() so that any smx initialization errors can be displayed to the screen. Console input is implemented in opcon_main() in app.c

Unp versions are unprotected from reentrancy. They do not use kernel services so they may be used from ISRs and LSRs, but screen output may be corrupted if one is interrupted. The normal versions of the functions use smx mutex or task lock SSRs, so they must not be used from ISRs. They also cannot be used from LSRs since the mutex calls wait. Note that the mutexes are created with priority inheritance enabled, to avoid priority inversion.

Configuration

These are defined in bsp.h in each BSP.

SB_CON_IN_PORT

Select the serial port number for input, starting at 0. Set to -1 if a local keyboard is used.

SB_CON_OUT_PORT

Select the serial port number for output, starting at 0. Set to -1 for if a local video display is used.

SB_CFG_UARTI

Set to 1 if the UART port uses interrupts. An example of this is if our CRT functions are mapped to a serial port and the serial driver uses interrupts. When this value is set to 1, the screen is cleared a little later than if it were set to 0, since smx must be past the point in initialization that creates the LSR queue. Also, when this is set to 1, it is necessary to unmask the interrupt used. When there actually is a display connected to the target and it is written to by writing to video memory, no interrupt is required, so set this to 0.

Input

Input is handled by opcon_main() in app.c.

Output

void **sb_ConClearEndOfLine**(u32 col, u32 row, u32 len)
void **sb_ConClearEndOfLineUnp**(u32 col, u32 row, u32 len)

Clears to the end of a line on the screen. Clears only to the end of the left panel or right panel, depending on which panel the string starts in. This way status messages in the right panel are not cleared by messages written in the left panel. col, row, and len are the starting column, row, and length of the message that was just written, i.e. the one whose end of line should be cleared.

void **sb_ConClearLine**(u32 row)

Clears a line on the screen.

void **sb_ConClearScreen**(void)
void **sb_ConClearScreenUnp**(void)

Clears the entire screen.

void **sb_ConCursorOff**(void)
void **sb_ConCursorOn**(void)

Turns the cursor off or on. Not implemented in some versions.

void **sb_ConInit**(void)

Does any necessary screen/terminal initialization. Sets the global video pointer to the starting address of the video memory or sends the ANSI terminal reset command to a terminal. (Does not initialize the UART; that is done in sb_PeripheralsInit().)

int **sb_ConPutString**(const char* in_string)

Similar to puts(), it writes a string to the screen and scrolls up a line when it reaches the bottom.

void **sb_ConScroll**(void)

Scrolls the screen up a line.

void **sb_ConWriteChar**(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink, char ch)
void **sb_ConWriteCharUnp**(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink, char ch)

Writes the specified character to the screen at the column (x) and row (y) specified, with the specified foreground and background colors. If *blink* is non-zero, the text blinks on and off.

void **sbConWriteCounter**(u32 col, u32 row, u32 Fcolor, u32 Bcolor, u32 ctr, u32 radix)

Writes a numeric value to the screen at the specified column (x) and row (y), with the specified foreground and background colors. If radix == 10, the number is converted in decimal; if 16, it is hexadecimal.

void **sb_ConWriteString**(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink, const char* in_string)
void **sb_ConWriteStringUnp**(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink, const char* in_string)

Writes the specified string to the screen starting at the column (x) and row (y) specified, with the specified foreground and background colors. If *blink* is non-zero, the text blinks on and off.

void **sb_ConWriteStringNum**(u32 col, u32 row, u32 F_color, u32 B_color, u32 blink, const char* in_string, u32 num)

Like sb_ConWriteString() but it writes the indicated number of characters. Does not look for a NUL character.

3. Common Definitions

This section documents smxBase configuration and definitions such as basic data types.

3.1 Configuration

bcfg.h defines some basic configuration settings.

Data Types

SB_CFG_FIXED_WIDTH_TYPES

C99 compilers provide integer data types which have exact widths, to allow writing more portable code for variables that must be the same size on any platform. If your compiler supports this feature, set it to 1.

UART Driver

SB_CFG_UARTI

Set to 1 to use the interrupt-driven UART driver; 0 to use the polling vendor provided code. Only supported for some targets, as indicated in the conditional in bcfg.h.

Console I/O

See the section APIs/ Base and Utility/ Message Display Functions for information about OMB.

SB_CFG_CON

Set to 1 to enable console input and output with a terminal or with a terminal emulator, such as TeraTerm.

SB_SIZE_OMB

Controls sb_omb size in bytes. OMB holds messages until they are sent to the UART.

Other Configuration

SB_CFG_TM

Set to 1 to enable time measurement routines.

3.2 Data Types and Defines

bdef.h defines the basic data types and keywords used in SMX. The keywords have been ported for several compilers. Add a section for your compiler and implement it, if necessary.

Data Types

s8	8-bit signed
s16	16-bit signed

s32	32-bit signed
s64	64-bit signed
u8	8-bit unsigned
u16	16-bit unsigned
u32	32-bit unsigned
u64	64-bit unsigned
uint	unsigned 16-bit or 32-bit integer depending on CPU word size
vs8	volatile 8-bit signed
vs16	volatile 16-bit signed
vs32	volatile 32-bit signed
vs64	volatile 64-bit signed
vu8	volatile 8-bit unsigned
vu16	volatile 16-bit unsigned
vu32	volatile 32-bit unsigned
vu64	volatile 64-bit unsigned
f32	floating point
f64	double precision floating point
bool	boolean
booli	boolean int-size
vbool	volatile boolean
false	0
true	>0
NULL	0
OFF	0
ON	>0

If SB_CFG_FIXED_WIDTH_TYPES is set to 1 in bcfg.h, u32 and similar are defined using C99 built-in fixed-width datatypes.

bool is a pre-defined type for C++ and its size is controlled by the compiler. For C we have to define it. For C99 we use stdbool.h to map it to _Bool. For Windows, we have to define it as u8 since Windows .h files define it as a byte. Otherwise, we define it as int for efficiency, but you can change it if this causes a problem or if you prefer to use another type. One problem is it may conflict with other code in your project. booli is always int-sized, for use in structs where alignment matters. Since bool may be a smaller type, a (bool) typecast is needed to assign a booli to a bool. For bool, the compiler adds a little code where bool variables are assigned to ensure they can only be set to true or false. SMX code uses bool (our data type), which does not add this checking, which is unnecessary since SMX always sets such variables to TRUE or FALSE.

volatile is a keyword that tells the compiler that the variable or field could be changed externally, such as by an ISR, so the compiler reads it each time before using it. Variables and fields that map onto peripheral registers should be defined with a volatile data type.

4. Porting Layer

Note: The OS porting layer has been removed in v5.2.0. Middleware now supports only SMX.

4.1 Processor Architecture

barmm.h and bsp.h define processor architecture-specific features such as endianness, enable/disable interrupt instructions, instruction to do a trap, register size, etc. The most common settings and macros are:

SB_STACK_ALIGN
sb_HALTEXEC()
sb_DEBUGTRAP()
sb_INT_ENABLE()
sb_INT_DISABLE()
SB_CPU_BIG_ENDIAN
ISR_PTR

4.2 Compiler

Compiler Keywords

bdef.h defines the following keywords as appropriate for each compiler.

__inline__	Keyword for inline functions.
__interdecl	Keyword that the compiler requires to declare a C language interrupt service routine (ISR) function that is activated directly by the interrupt controller, not through an ISR shell. For systems, using assembly ISR shells, __interdecl should be defined as a null macro. See 4.4 Interrupt Service Routines (ISRs) for additional discussion about interrupt shells. __interdecl precedes the *void* keyword in an ISR declaration.
__interrupt	Keyword that the compiler requires to declare a C language interrupt service routine (ISR) function that is activated directly by the interrupt controller, not through an ISR shell. For systems, using assembly ISR shells, __interrupt should be defined as a null macro. __interrupt follows the *void* keyword in an ISR declaration.
__packed	Keyword for packed structures, if the compiler has one.
__packed_gnu	Same as __packed, but for the GNU compiler.
__packed_pragma	Set to 0 if the compiler has a *packed* keyword, and it is used. Otherwise, set to 1 to enable #pragma pack(1) in the code. If this is not the syntax your compiler uses for this pragma, change it everywhere it is used.
__short_enum_attr	Keyword to make an enum use the smallest data type to hold all values.
__unaligned	For processors, such as ARM, that require reading/writing data on a same-sized boundary (e.g. 4-byte boundary to read 4-byte data), this keyword tells the compiler the data may not be properly aligned, generating extra code to read/write each byte separately.

4.3 Interrupt Service Routines (ISRs)

The SMX Target Guide discusses how to write ISRs for different architectures. See the section Architectural Notes/ ISRs for the processor architecture you are using.