Micro Digital

smxBase™ User's Guide

Foundation Definitions and Library

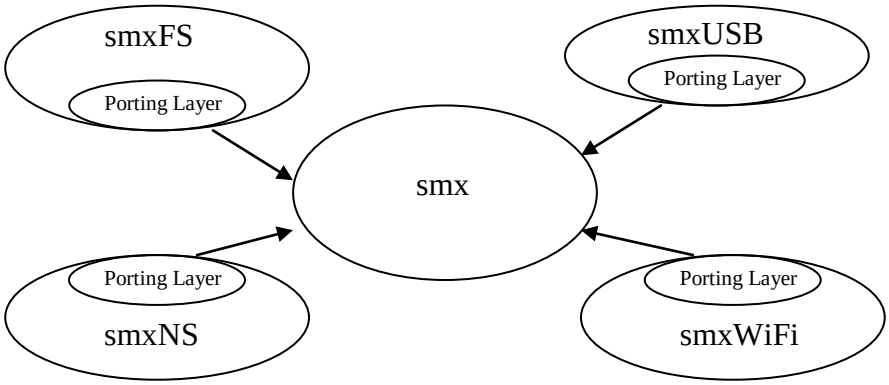
Version 5.1.0
December 18, 2021

by David Moore, Ralph Moore, and Yingbo Hu

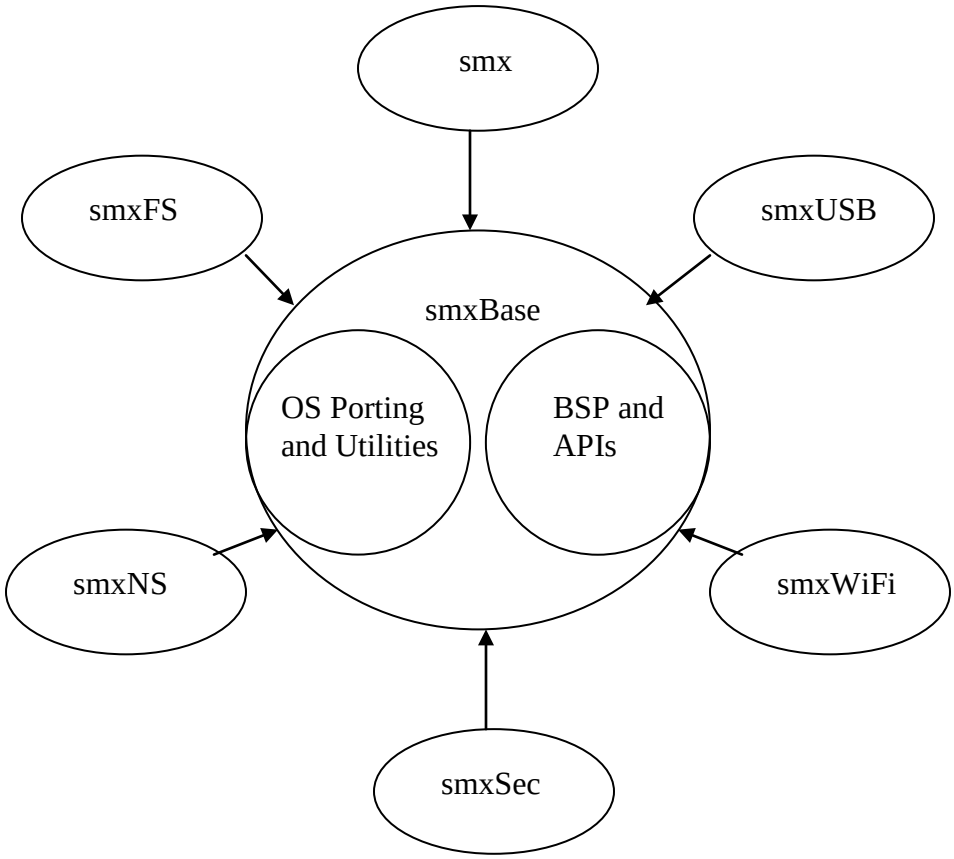
Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

1. Overview

smxBase™ provides the base for the SMX® RTOS, and it can support running middleware standalone. In the past the smx kernel was central to the SMX RTOS:



Now smx is a peripheral component like the others, with smxBase in the central role:



smxBase contains common definitions and routines, OS porting layer, and board and processor support (BSP) code. The OS porting layer is used by all SMX modules (middleware) such as

smxFS, smxNS, smxUSBH, and smxWiFi. In the past, each had its own porting layer. Now it is only necessary to implement this one porting layer and all SMX modules will work! This is a great simplification and reduction of work. (Each module still contains a small amount of porting code that is specific to it.) It is only necessary to port functions and definitions that are used by the modules you have. This can be determined by searching your release for SB_OS_ and sb_OS_.

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 BOOLEAN. 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, bcon.h, bkbd.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 barm.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 bbsp.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. **OS Porting Layer**: Defines a unique interface that is used by SMX modules so they can be ported to any OS or none. These avoid the need to modify the source code. See bos.c,h.

smxBase is contained in the XBASE and BSP directories. smxbase.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 smxbase.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)

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

Dynamically Allocated Regions, DARs, are the most basic memory structure for smxBase, in a non-MMU system. In general, there must be at least one DAR for each separate dynamic RAM area (e.g. SRAM and DRAM). 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, e.g. sb_sdar. 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.

At a minimum, sb_SDAR must be defined for system control blocks and it should not be used by application code. This is for safety, since application code is unproven and its blocks and stacks may exceed their boundaries. Overwriting system control blocks can cause system software to behave in mysterious ways, which is highly undesirable since the user is not privy to system software details.

DAR API

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

Initializes the DAR control block, DCB, pointed to by darp (e.g. sb_sdar) 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. Since DARAlloc() also creates SDAR, the first time, SBE_DAR_INIT_FAIL can occur due to failure to create it.

BOOLEAN **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

```
BOOLEAN     sb_BlockPoolCreate(u8 *p, PCB_PTR pool, u8 num, u16 size,  const char *name);
BOOLEAN     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_PARM par)
u8 *        sb_BlockGet(PCB_PTR pool, u16 clrsz)
BOOLEAN     sb_BlockRel(PCB_PTR pool, u8 *dp, u16 clrsz)
```

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

BOOLEAN **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.

BOOLEAN **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. The following is an example of creating an smx TCB pool in SDAR:

```
PCB     smx_tcbs;
sb_BlockPoolCreateDAR(sb_sdar, &smx_tcbs, NUM_TASKS, sizeof(TCB), SB_CACHE_LINE, "smx_tcbs")
```

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_PARM 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_PARM 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.

BOOLEAN **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 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 counter. 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_TM_EN = 1 (see bbsp.h), TM functions are inserted when compiled and if sb_TM_EN = 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 to the right panel (half) of the terminal. 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 Queue (OMQ) and Output Message Buffer (OMB) are implemented. These queue messages until they are output to the UART by the idle task or other low priority code. The OMQ contains simple records that point to constant messages, typically in flash memory. The OMB contains message strings for variable messages that were created in buffers, such as to print a strings with values. Sequence numbers are used in both to ensure messages are displayed in the proper order.

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. The OMB uses much more RAM than the OMQ, and extra run time is needed to copy the strings to it, so it is preferable to use the OMQ for constant messages. sb_MsgOutputConst() adds messages to the OMQ; sb_MsgOutputVar() adds them to the OMB.

Note that because OMQ and OMB have fixed sizes, they can overflow during times of peak activity. This is indicated by special characters printed, as discussed in sb_MsgDisplay(), below. bcfg.h has configuration options to set OMQ and OMB size (SB_CFG_OMQ_SIZE and SB_CFG_OMB_SIZE), but if it is not possible to make them large enough to avoid overflow, due to using a high debug level in one of the SMX libraries, for example, it may be necessary to temporarily enable the DIRECT option, which couples them directly to the UART. When the debug level is reduced after solving the problem, the DIRECT option should be disabled. See 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.

A couple of main configuration options are provided that are documented in the Configuration section of this manual. Here is some guidance about them:

- If you want the code to run at full speed (not wait on a polled UART) and you have plenty of RAM, set SB_CFG_MSGOUT_DIRECT to 0 and

SB_CFG_MSGOUT_VARMSG to 1. This is the default configuration and uses the OMQ and OMB to display messages. Set the OMQ and OMB sizes large enough to avoid losing messages (indicated by Q or B chars written to the ends of the lines).

- If you want the code to run at full speed (not wait on a polled UART) and you have limited RAM, set SB_CFG_MSGOUT_DIRECT to 0 and SB_CFG_MSGOUT_VARMSG to 0. This omits the OMB and only allows writing constant messages. Messages printed with sb_MsgOutVar() are lost. SMX middleware modules that print strings containing a constant part plus a value will only print the constant part.
- If you have very limited RAM and it is critical not to lose messages, temporarily set SB_CFG_MSGOUT_DIRECT to 1. This omits the OMQ and OMB and couples the output directly to the UART. If running with a polled UART driver, the code will be hindered by message output. You would use this configuration temporarily when setting the debug level high in one of the SMX middleware modules to determine why it is failing to initialize or run properly, and then return to the original settings once the problem is resolved.

void **sb_MsgDisplay**(void)

Displays all messages in the OMQ and OMB, starting with the oldest. Using the sequence number of each message, this routine alternates between the buffers to display them in order. Normally it is called from the idle task (only), if smx is present, or from low-priority code, if not. 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. 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 OMQ or OMB filled and messages were lost, a Q or B is displayed in the rightmost column to alert the viewer. (Note: in the case of error messages, the corresponding errors will have incremented error counters and their error records may be in 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 "\n" or 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 OMQ and OMB and is protected against reentry. (Attempted reentry results in a nop, but all messages in OMQ and 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 sb_omq 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_MsgOutConst**(u32 mtype, const char *mp)

Outputs a constant message to the terminal or other output device. The default implementation creates and stores a message record in the OMQ. The OMQ record format consists of message pointer, message type, overwrite flag, and sequence number. It is a circular queue. If it fills, a special character is displayed to the end of a line, as explained above. This function is written to be very fast. It can be used from multiple ISRs; it is interrupt-safe and protected against reentry. This function is for use in displaying constant messages (i.e. string literals, typically stored in ROM).

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

Outputs a variable message to the terminal or other output device. The default implementation copies the message to the OMB, which holds messages until they can be output through the UART. Although this function can be used to display constant messages too, it is preferable to use sb_MsgOutConst() whenever possible, to save RAM and run time, since it only creates and initializes a record in OMQ; it does not copy the message string.

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

void **SFF_GET_LOCAL_TIME**(SFF_DATETIME * pDateTime)

Returns the local time via the parameter. You may need to read the date/time from the RTC of your system and fill out the member variable of structure DATETIME. For example, if your RTC returns January 27 2009, 1:29:18 PM then

```
pDateTime->wYear = 29; /* since 1980 */
pDateTime->wMonth = 1;
pDateTime->wDay = 27
pDateTime->wHour = 13; /* PM need to add 12 */
pDateTime->wMinute = 29;
pDateTime->wSecond= 18;
```

ALIGN macros: 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. barm.h, bcf.h, b86.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.

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**bbsp.h**. The BSP variables and functions are implemented in XBASE**bbsp.c** and bsp.c, which is located in the BSP directory. bbsp.c contains 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 today's 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 xbsp.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 xbsp.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: In the Notes at the top of bsp.c, document the IRQ numbering convention used by your bsp.c, as we do in the bsp.c provided. 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. Point to the relevant table/figure in the processor manual, in your comment.

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_DEBUGGER_IRQ_RX and **SB_DEBUGGER_IRQ_TX**: These are used only when a software debug monitor is used for debugging, not when using BDM, JTAG, etc. They are used during app init to unmask the IRQ used by the debugger for an asynchronous break (i.e. when the user presses the Stop button).

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.

SB_MIN_RAM: Set to 1 for targets with minimal RAM, such as SoCs that only support internal memories or boards with no external RAM. This is used to select smaller settings in acfg.h.

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

Configuration Data

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

cpu_periph_reg_base: The base address of the I/O space for the integrated peripherals in the CPU. Having this global permits other libraries to access the I/O space without having to include BSP header files. (The library routines would need to supply the offsets of course.)

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**bbsp.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()

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_DemoExit()
sb_DemoInit()
sb_EVBInit()
sb_Exit(retcode)
sb_PeripheralsInit()
sb_PtimeGet()
sb_Reboot()
sb_Restart()

Notes about the API reference below:

1. Functions that return BOOLEAN 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

BOOLEAN **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.

BOOLEAN **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=0) — 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.

BOOLEAN **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.

BOOLEAN **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.

BOOLEAN **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.

BOOLEAN **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.

BOOLEAN **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.

BOOLEAN **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()).

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=0) — 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.

BOOLEAN **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 typical sequence for hooking an interrupt:

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

Note that for some processor architectures, my_isr 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, if multiple interrupts share one IRQ in your system, you can use the OS porting function sb_OS_ISR_CFUN_INSTALL() to hook ISRs. You pass the pointer to the body of your ISR, which is a normal C function, and the dispatcher handles the rest. See section 4.4 Interrupt Service Routines (ISRs) for details. However, there is overhead and complexity in this approach, so normally the sequence of calls above should be used.

BOOLEAN **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.

BOOLEAN **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.

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.

BOOLEAN **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

BOOLEAN **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).

BOOLEAN **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 bbsp.c in \XBASE. 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().

BOOLEAN **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().

BOOLEAN **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

BOOLEAN **sb_ConsoleInInit**(void) — Optional

Initializes the console input device, if SB_CON_IN 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().

BOOLEAN **sb_ConsoleOutInit**(void) — Optional

Initializes the console output device, if SB_CON_OUT 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().

void **sb_DemoExit**(void) — Optional (stub required)

Does any necessary BSP demo shutdown or cleanup. Called by appl_exit().

void **sb_DemoInit**(void) — Optional

Starts BSP-specific demo code. In a multitasking environment, this should create and start any BSP demo tasks. Demo code should be put into a separate file and only called from this function. This function provides a hook for us to add demo code for special peripherals, such as text LCDs, TPUs, etc. that are only present on some processors or boards. Called by appl_init().

BOOLEAN **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().

BOOLEAN **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.

void **sb_Restart**(void) — Optional

Restarts the application from the entry point. Does not reboot the system. The default implementation in most BSPs is to infinite loop.

2.8 Console I/O

bcon.h and bkbd.h 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**bcon.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** and **crt.c** in the BSP directories. sb_ConInit() is called early in smx_Go() so that any smx initialization errors can be displayed to the screen.

The console input (keyboard) functions are implemented in **kbd.c** in the BSP directories.

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_IN_USES_INT

Set to 1 if the input port uses interrupts or 0 if polled.

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_CON_OUT_USES_INT

Set to 1 if the output 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

void **sb_KbdInit**()

Does any necessary keyboard initialization.

void **sb_KbdExit**()

Does any necessary keyboard de-initialization.

char **sb_KbdConvertKey**(u32 key32)

Converts the key code (e.g. scan code) passed by the driver into an ASCII character.

char **sb_KbdGetCharPoll**()

Gets a character by polling.

void **sb_KbdStoreKeyLSR**(u32 key32)

Passes key to a task via pipe or other method of intertask communication.

Output

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

Clears a line on the screen.

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

Clears the entire screen.

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

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.

int **sb_ConScroll**()

Scrolls the screen up a line.

void **sb_ConWriteChar**(int col, int row, int F_color, int B_color, int blink, char ch)
void **sb_ConWriteCharUnp**(int col, int row, int F_color, int B_color, int 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**(int col, int row, int Fcolor, int Bcolor, u32 ctr, int 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**(int col, int row, int F_color, int B_color, int blink, const char *in_string)
void **sb_ConWriteStringUnp**(int col, int row, int F_color, int B_color, int 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**(int col, int row, int F_color, int B_color, int blink,
const char *in_string, int num)

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

2.9 PCI

bpci.h defines the PCI functions. Use these functions to get basic configuration information of devices attached to the PCI bus. This is a standard API. For more information, please refer to a PCI reference such as PCI & PCI-X Hardware and Software Architecture & Design, Edward Solari and George Willse, ISBN 0929392639, **pages 1185-1202**. Also see **page 1022** and those that follow for the layout of the configuration register space.

Notes

1. Currently implemented in smx only for x86 and ColdFire. For ColdFire, it is likely you will have to modify the implementation for your hardware.

2. Defines and prototypes are in XBASE**bpci.h**.

   x86: Implemented in BSP\X86**pcix86.c** and **pcix86a.asm**.
   ColdFire: Implemented in BSP\CF**pcicf.c**.

3. Values are returned via parameters. The error code byte is returned by the function. See table below.

4. Three of the functions are marked "Implemented but untested". These functions are for special purposes ("PCI Support Extensions") and we have not needed them for our PCI drivers, so it is likely you won't either. If you do, they should work, or they should be close.

Return Codes

SB_PCI_FRET_SUCCESSFUL	0x00
SB_PCI_FRET_FUNCTION_NOT_SUPPORTED	0x81
SB_PCI_FRET_BAD_VENDOR_ID	0x83
SB_PCI_FRET_DEVICE_NOT_FOUND	0x86
SB_PCI_FRET_BAD_REGISTER_NUMBER	0x87
SB_PCI_FRET_SET_FAILED	0x88
SB_PCI_FRET_BUFFER_TOO_SMALL	0x89

Common Parameters

bus	Bus number where the device is located (0-255)
devfun	Device number [7::3] and Function Number [2::0]
reg	Configuration space register to read/write (0-255)

BOOLEAN **sb_PCIBiosPresent**(void)

Returns TRUE if a PCI BIOS is present.

u8 **sb_PCIFindClass**(u32 classcode, u16 index, u8 *bus, u8 *devfun)

Finds a PCI device by class code. See PCI_CC constants in xpci.h; for example, the class code for a VGA controller is PCI_CC_DISPLAY_VGA == 0x030000. *index* indicates which instance of this class of device to search for (i.e. the 1st, 2nd, etc device of this type). All devices of a particular class code can be found by calling this in a loop that increments *index*, each iteration. The bus, device, and function numbers are returned via the parameters.

u8 **sb_PCIFindDevice**(u16 vendor, u16 device, u16 index, u8 *bus, u8 *devfun)
Finds a PCI device by the vendor and device number. Each vendor is assigned a unique ID by the standards organization. Each vendor assigns IDs to their devices. You would need to know both of these to identify the device. *index* indicates which instance of this device to search for (i.e. the 1st, 2nd, etc device of this kind). The bus, device, and function numbers are returned via the parameters.

u8 **sb_PCIGenSpecialCycle**(u8 bus, u32 val)

Allows the caller to broadcast PCI Special Transaction data to a specified bus in the sytem. Implemented but untested.

u8 **sb_PCIGetIntRoutingOptions**(sb_PCIRoutingStruct *rt_struct, u16 *irqbits)
Returns the PCI interrupt routing options available on the platform. The PCI interrupt routing options define how the platform is able to route individual hardware interrupt lines to PCI devices and PCI slots. Returns a bitmap containing the current hardware interrupt line (IRQ) assignments that are exclusive to PCI devices. Implemented but untested.

u8 **sb_PCISetHardwareInt**(u8 bus, u8 devfun, u8 pin, u8 irq)

Allows the caller to request that a specific hardware interrupt line (IRQ) be connected to a specified interrupt pin of a PCI device. Implemented but untested.

u8 **sb_PCIReadConfigByte**(u8 bus, u8 devfun, u8 reg, u8 *val)
u8 **sb_PCIReadConfigWord**(u8 bus, u8 devfun, u8 reg, u16 *val)
u8 **sb_PCIReadConfigDword**(u8 bus, u8 devfun, u8 reg, u32 *val)

These read configuration data (u8, u16, or u32) for the specified device, from the Configuration Space Register indicated by *reg* and returns it in **val*. The layout of the configuration space is documented in the reference cited above, starting on **p. 1022**.

u8 **sb_PCIWriteConfigByte**(u8 bus, u8 devfun, u8 reg, u8 val)
u8 **sb_PCIWriteConfigWord**(u8 bus, u8 devfun, u8 reg, u16 val)
u8 **sb_PCIWriteConfigDword**(u8 bus, u8 devfun, u8 reg, u32 val)

These write configuration data (u8, u16, or u32) for the specified device, to the Configuration Space Register indicated by *reg*. The layout of the configuration space is documented in the reference cited above, starting on **p. 1022**.

2.10 Block Device Interface

bbd.h provides a general interface for any file system or media utility to access different types of media. This hides details of the devices/media from the file system.

The following are the seven driver interface functions. They have been defined to support all types of fixed and removable media.

```
int     DriverInit(void);
int     DriverRelease(void);
int     DiskOpen(void);
int     DiskClose(void);
int     SectorRead(u8 * pRAMAddr, u32 dwStartSector, u16 wHowManySectors);
int     SectorWrite(u8 * pRAMAddr, u32 dwStartSector, u16 wHowManySectors);
int     IOCtl(uint dwCommand, void * pParameter);
```

Each driver prefixes these names so they are distinct (e.g. RAMDiskInit()). These functions are never called directly; they are called via the function pointers in the interface structure. The following is a more detailed discussion of them. When writing a new device driver, use the RAM disk driver as a guide (fdram.h, fdram.c).

Note: You may need to add mutex get/release to these functions if your file system re-enters them.

int **DriverInit**(void)

The file system should call this function exactly once when it first initializes the driver. Initialize your device hardware in this routine. Return SB_PASS if successful, or SB_FAIL if any error occurs. If SB_FAIL, the file system should not access this driver anymore.

int **DriverRelease**(void)

The file system should call this function exactly once when it shuts down usage of the driver. You can do some cleanup work for your device hardware such as disabling the controller and/or interrupt. Return SB_PASS if successful, or SB_FAIL if any error occurs.

int **DiskOpen**(void)

The file system should call this function once after the SBD_IOCTL_INSERTED IOCtrl returns SB_PASS. You can do some further hardware initialization work and allocate internal buffers and data structures in this function. You may also need to query the disk's physical information such as the total number of sectors and sector size. Return SB_PASS if successful, or SB_FAIL if any error occurs. If SB_FAIL, the file system may not continue to mount it again.

int **DiskClose**(void)

The file system should call this function once after the SBD_IOCTL_REMOVED IOCtl returns SB_PASS. You can do some further hardware cleanup work, such as freeing the internal buffers and data structures allocated by DiskOpen(). Return SB_PASS if successful, or SB_FAIL if any error occurs.

int **SectorRead**(u8 * pRAMAddr, u32 dwStartSector, u16 wHowManySectors)

The file system calls this function when it wants to read some data from the disk. The length of the read operation is specified in sectors.

pRAMAddr is the address of the RAM buffer for the data.

dwStartSector is the index of the starting sector you want to read. The index is the offset from the beginning of the disk.

wHowManySectors is the number of sectors you want to read. For example, if file system wants to read some data from sectors 35 to 36, it calls this with SectorRead(pBuf, 35, 2);

Return SBD_OK if successful. Any other value means an error occurred. Possible error codes include:

SBD_MEDIA_REMOVED
SBD_DEVICE_ERROR

int **SectorWrite**(u8 * pRAMAddr, u32 dwStartSector, u16 wHowManySectors)

The file system calls this function when it wants to write some data to the disk. The length of the write operation is specified in sectors.

pRAMAddr is the address of the RAM buffer for the data.

dwStartSector is the index of the starting sector you want to write. The index is the offset from the beginning of the disk.

wHowManySectors is the number of sectors you want to write. For example, if file system wants to write some data to sectors 41 to 44, then it calls this with SectorWrite(pBuf, 41, 4);

Return SBD_OK if successful. Any other value means an error occurred. Possible error codes include:

SBD_MEDIA_REMOVED
SBD_BAD_BLOCK
SBD_WRITE_PROTECT
SBD_DEVICE_ERROR

int **IOCtl**(uint dwCommand, void * pParameter)

The file system calls this function to get/set device-specific information.

dwCommand indicates which operation the file system needs the driver to do.

pParameter is a parameter that is specific to the command.

The pre-defined I/O commands are:

SBD_IOCTL_INSERTED
SBD_IOCTL_REMOVED
SBD_IOCTL_CHANGED
SBD_IOCTL_WRITEPROTECT
SBD_IOCTL_FLUSH
SBD_IOCTL_GETDEVINFO

New I/O commands can be added. Make sure the new commands are bigger than SBD_IOCTL_CUSTOM and add them to the header file of your driver.

SBD_IOCTL_INSERTED

The file system periodically passes this command to the IOCtl() function if it does not detect the media insertion event (the previous IOCtl (SBD_IOCTL_INSERTED) call returned SB_FAIL). This function returns the status to the caller via pParameter, which should be a pointer to int. Returns SB_PASS if media has been inserted, or SB_FAIL if no media is detected. If you are using a fixed media type such as NAND flash, just return SB_PASS. Only include code to detect if media is present, for best performance.

SBD_IOCTL_REMOVED

The file system may periodically pass this command to the IOCtl() function if it has already detected the media insertion (the previous IOCtl (SBD_IOCTL_INSERTED) call return SB_PASS). This function returns the status to the caller via pParameter, which should be a pointer to int. Returns SB_PASS if the media has been removed, or SB_FAIL if the media is still

inserted. If you are using a fixed media type such as NAND flash, just return SB_FAIL to pParameter. Only include code to detect if media is not present, for best performance.

Note: Some devices such as MMC/SD use different commands or methods to detect media insertion and removal, so the block device interface defines two separate commands (above) so the driver does not need to save the current status.

SBD_IOCTL_CHANGED

The file system may periodically pass this command to the IOCtl() function if it has already detected that media has been inserted (the previous IOCtl (SBD_IOCTL_INSERTED) call returned SB_PASS). This function returns the status to the caller via pParameter, which should be a pointer to int. Sometimes there is no file operation but the removable device may have been removed and inserted again, so the file system needs to refresh the device content.

SBD_IOCTL_WRITEPROTECT

The file system passes this command to the IOCtl() function to get the write protect status of the device. Return SB_PASS if it is currently write protected, and all write operation should fail.

SBD_IOCTL_FLUSH

The file system passes this command to the IOCtl() function to force flushing the data in the device driver's cache or buffer back to the device. If your device driver uses any buffer or cache, you should flush the cache when the driver receives this command.

SBD_IOCTL_GETDEVINFO

The file system passes this command to the IOCtl() function after it detects the card is inserted, and the file system uses the information returned. You must pass a pre-allocated SBD_DEVINFO structure pointer as the second parameter of IOCtl function. Then file system can retrive the information from the returned structure. You may need to retrieve the total number of sectors and sector size in the DiskOpen() function and save it in the SBD_DEVINFO structure. Then when the file system calls the IOCtl() function you do not need to query the hardware each time. This improves performance. Return SB_PASS if the value is ready, otherwise return SB_FAIL.

```
typedef struct
{
    u32 dwSectorsNum;
    uint dwSectorSize;
    uint wPartition;
    uint wAutoFormat;
    uint wFATNum;
    uint wRootDirNum;
    uint wRemovable;
    uint wFATCacheSize;
    uint wDirCacheSize;
    uint wDataCacheSize;
    uint wLogicalPartition;
} SBD_DEVINFO;
```

dwSectorsNum is the total sectors number of the registered device. For example, 65536.

dwSectorSize is the size of a sector in bytes. For example, 512 or 1024.

wPartition specifies which partition you want file system to handle. The valid values are 0, 1, 2, 3. This parameter is useful if your media has multiple partitions. Normally you should set it to 0. Set it to SBD_NO_PARTITION if your media should not have a partition table, such as a floppy disk.

wAutoFormat is a flag to tell the file system to automatically format the media when mounting it, if file system cannot detect a valid format on it. Setting this to 1 may be convenient but 0 is safest. Consider whether the user may insert media formatted on another OS (not the filesystem you are using), or whether the data is so critical that you would attempt to use a utility or service to salvage whatever you can from the media if it was corrupted.

wFATNum specifies how many FATs will be created. Normally it should be set to 2 for removable media, since this is most compatible with other OSs. This setting is used when formatting new media or media that was already corrupted before file system attempted to mount it. Otherwise, the value in the BPB of the current format is used again when reformatting. Ignore it if you are not using the FAT file system.

wRootDirNum specifies how many root directory entries will be created when file system formats this device. Normally you should just set it to 512. This setting is ignored if the format is FAT32 (since FAT32 has no root directory area; the root directory is a file like a subdirectory). This setting is used when formatting new media or media that was already corrupted before file system attempted to mount it. Otherwise, the value in the BPB of the current format is used again when reformatting. Ignore it if you are not using the FAT file system.

wRemovable specifies whether the media is removable (1) or fixed (0). For FAT file system, this is used to determine what value should be used for the MediaType byte in the BPB when formatting the media. The file system may overwrite this value.

wFATCacheSize specifies FAT cache size in bytes. You can use this field to overwrite the default file system cache settings. Otherwise, leave it alone. Used by FAT file system only.

wDirCacheSize specifies Directory cache size in bytes. You can use this field to overwrite the default file system cache settings. Otherwise, leave it alone. Used by FAT file system only.

wDataCacheSize specifies Data Cache size in bytes. You can use this field to overwrite the default file system cache settings. Otherwise, leave it alone. Used by FAT file system only.

wLogicalPartition specifies which logical partition in an extended partition you want the file system to handle. This parameter should only be checked if the partition indicated by **wPartition** is an extended partition. Used by FAT file system only.

SBD_IOCTL_DELSECTOR

This IOCtl is only used for some devices which need FTL such as NAND and NOR flash to tell the device driver some sectors are not necessary and the device driver can do garbage collection for those sectors in the future. Normally you do not need to implement this IOCtl.

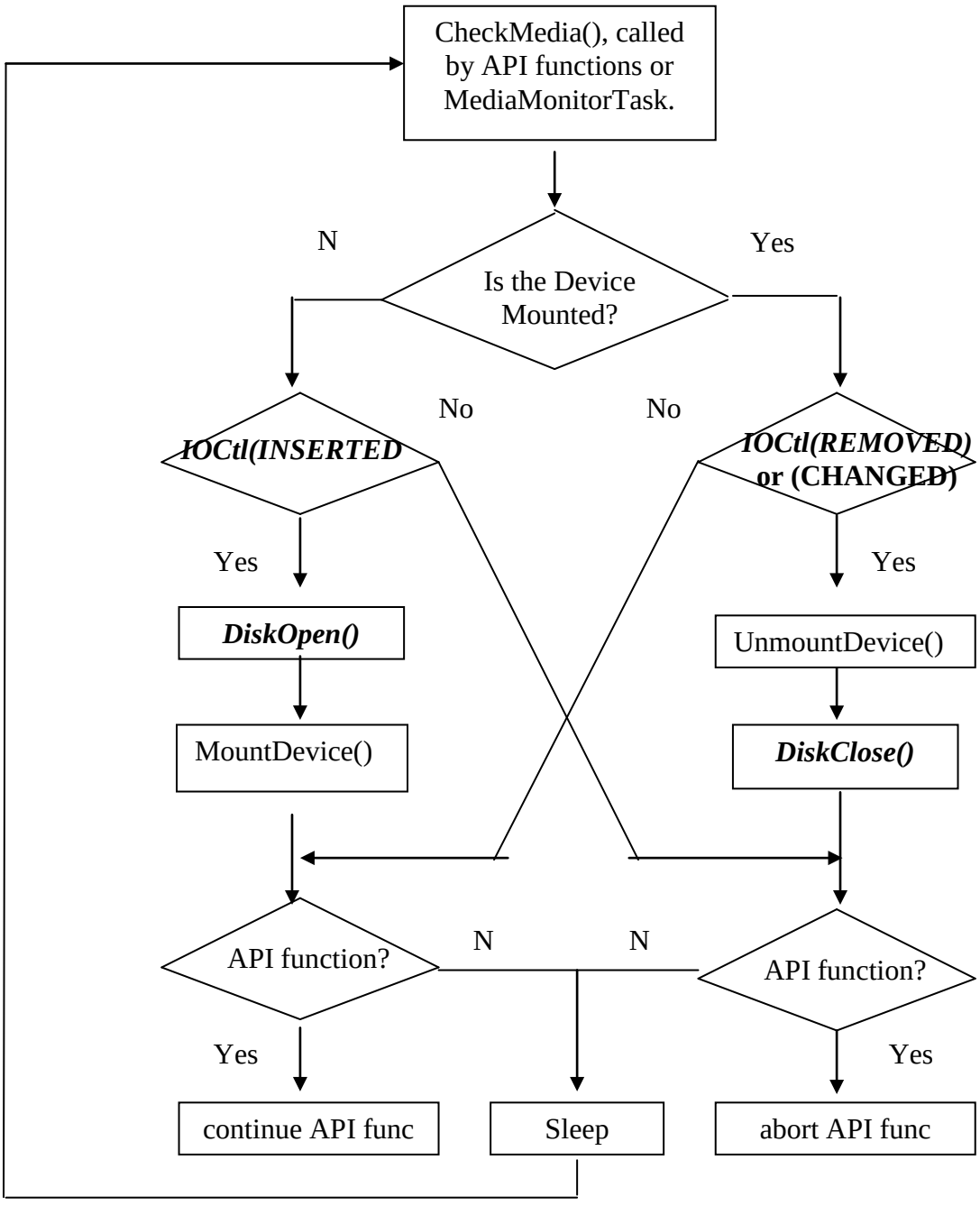
Sample IOCtl() code (from RAM disk driver)

```
static int  RAMIOCtl(u32 dwCommand, void * pParameter)
{
   int result = SB_PASS;
   switch(dwCommand)
   {
   case SBD_IOCTL_INSERTED:
      *((int *)pParameter) = SB_PASS;
      break;
   case SBD_IOCTL_REMOVED:
      *((int *)pParameter) = SB_FAIL;
      break;
   case SBD_IOCTL_CHANGED:
      *((int *)pParameter) = SB_FAIL;
      break;
   case SBD_IOCTL_WRITEPROTECT:
      *((int *)pParameter) = SB_FAIL;
      break;
   case SBD_IOCTL_GETDEVINFO:
      {
         SBD_DEVINFO * pDeviceInfo = (SBD_DEVINFO *)pParameter;
         pDeviceInfo->dwSectorsNum = RAMDISK_SIZE/RAMDISK_SECTOR;
         pDeviceInfo->dwSectorSize = RAMDISK_SECTOR;
         pDeviceInfo->wPartition = 0;
         pDeviceInfo->wAutoFormat = 1;
         pDeviceInfo->wRootDirNum = 256;
         pDeviceInfo->wFATNum = 1;
         pDeviceInfo->wRemovable = 0;
         break;
      }

   case SBD_IOCTL_FLUSH:
      break;
   }
   return result;
}
```

Media Change and Mounting and IOCTL Commands

The following diagram shows how smxFS checks for media change and how the media is mounted by either the monitor task or calls from the API functions.



Media Change Checking and Mounting Procedure

2.11 UART

For some targets, smxBase provides simple interrupt-driven UART driver which can be used for terminal output or a communication channel

int **sur_Open**(uint port, u32 baudrate, uint parity, uint dbit, uint sbit, uint ibs, uint obs, uint fc)

Summary Open the UART port.

Details This function should be called first before using other UART API functions. It initializes the UART port according to the parameters.

Pars

port	UART port number
baudrate	baud rate of the UART port
parity	parity of the UART port, should be one of the following: SUR_PARITY_NONE SUR_PARITY_ODD SUR_PARITY_EVEN SUR_PARITY_MARK SUR_PARITY_SPACE Not all parity options may be supported by your UART peripheral.
dbit	Number of data bits of the UART port
sbit	Number of stop bits of the UART port. Should be one of the following: SUR_STOP_BITS_1 SUR_STOP_BITS_1_5 SUR_STOP_BITS_2 Not all the stop bit option is supported by your UART peripheral.
ibs	Incoming data buffer size. Must be greater than 0.
obs	Outgoing data buffer size. Must be greater than 0.
fc	Flow control of the UART port. Should be one of the following: SUR_FC_NONE SUR_FC_HARDWARE SUR_FC_XONXOFF Not all the flow control option may be supported by your UART peripheral.

Returns

0	UART port opened.
-1	Could not open the UART port. May have run out of some resource required by the driver.

See Also sur_Close()

Example

```
if(sur_Open(115200, SUR_PARITY_NONE, 8, SUR_STOP_BITS_1, 64, 64,
                                                    SUR_FC_NONE) == 0)
{
}
```

int **sur_Close**(uint port)

Summary Close the UART port.

Details This API will release all the resource allocated by sur_Open(). No other UART API functions should be called after this AP is called.

Pars

	port	UART port number

Returns

0	UART port closed.
-1	UART port has not been opened yet.

See Also sur_Open()

Example

```
sur_Close(1);
```

int **sur_InByte**(uint port, u8 *pb, SUR_CALLBACK cb, uint tmo)

Summary Get one byte of data from the UART port.

Details Try to get one byte from the UART port. You may indicate timeout value and callback function if the data is not available.
Comments about timeout and callback function:

- This API will return immediately if the required data is already in the internal buffer. Callback function and timeout value are ignored for this case.
- If cb is NULL and tmo is not 0, this API will block the current calling task until it gets the required data or times out. When timeout occurs, this API will return any data available into the user's buffer.
- If cb is not NULL then tmo is ignored and this API will return immediately if the data is not ready yet. Callback function will be called only when the data is ready. Callback function may not be called at all under some conditions such as the cable is disconnected.
- If cb is NULL and tmo is 0, and data is not ready yet, this API will return any available data in the internal incoming buffer to the user's buffer.

Pars

	port	UART port number
	pb	Pointer to the byte buffer
	cb	Callback function if the data is not ready at present and you want the UART driver to call it when the data is ready.
	tmo	Timeout value (milliseconds) you want to wait if the data is not ready yet or SB_TMO_INF for infinite wait (not SB_OS_TMO_INF).

Returns

>=0	Data length returned in the buffer.
-1	This UART port is not open.

See Also sur_InData()

Example

```
sur_InByte(1, &c, NULL, 100);
```

int **sur_InData**(uint port, u8 * pdst, uint len, u8 term, SUR_CALLBACK cb, uint tmo)

Summary Get some data from the UART port.

Details Try to get some bytes from the UART port. You may indicate timeout value and callback function if the data is not available.

Comments about timeout and callback function:

- This API will return immediately if the required data is already in the internal buffer. Callback function and timeout value are ignored for this case.
- If cb is NULL and tmo is not 0, this API will block the current calling task until it gets the required data or times out. When timeout occurs, this API will return any data available into the user's buffer.
- If cb is not NULL then tmo is ignored and this API will return immediately if the data is not ready yet. Callback function will be called only when the data is ready. Callback function may not be called at all under some conditions, such as the cable is disconnected.
- If cb is NULL and tmo is 0, and data is not ready yet, this API will return any available data in the internal incoming buffer into the user's buffer.

Comments about terminator byte:

- This API allows you to indicate a terminator byte for a packet. For example, SLIP uses 0xCC as a packet terminator. Terminator byte is only checked when len is 0.
- You need to make sure the internal incoming buffer is big enough to hold the whole packet.
- You need to make sure the data buffer is big enough to hold the whole packet.

Pars

port	UART port number
pdst	Pointer to the data buffer. Buffer needs to be at least the same size as the incoming internal buffer, when a terminator byte is indicated (len is 0).
len	Data buffer len
term	Terminator byte. Ignored if len is not 0.
cb	Callback function to call when the data is ready, if it is not ready immediately
tmo	Timeout value (milliseconds) you want to wait if the data is not ready yet or SB_TMO_INF for infinite wait (not SB_OS_TMO_INF).

Returns

>=0	Data length returned in the buffer.
-1	This UART port is not open.

See Also sur_InData()

Example

```
sur_InData(1, pBuf, 10, 0, NULL, 100);
```

int **sur_OutByte**(uint port, u8 b, SUR_CALLBACK cb, uint tmo)

Summary Send one byte of data to the UART port.

Details Try to send one byte to the UART port. You may indicate a timeout value and callback function if you need to know the data has been fully sent out.
Comments about timeout and callback function:

- If cb is NULL and tmo is not 0, this API will block the current calling task until the data has been fully sent out by the UART or the operation timed out.
- If cb is not NULL then tmo is ignored, and this API will return immediately. Callback function will be called when all the data in the internal outgoing buffer is sent out. Callback function normally will be called even if the cable is disconnected, unless you closed the port.
- If cb is NULL and tmo is 0, this API will copy the data into the internal outgoing buffer and return immediately.

Pars

port	UART port number
b	Byte to be sent
cb	Callback function if you want to know when the data is fully sent out.
tmo	Timeout value (milliseconds) you want to wait for the data to be fully sent out or SB_TMO_INF for infinite wait (not SB_OS_TMO_INF).

Returns

>=0	The data length copied to the internal outgoing buffer. 0 if timeout occurs or buffer is full.
-1	This UART port is not open.

See Also sur_OutData()

Example

```
sur_OutByte(1, 'H', NULL, 100);
```

int **sur_OutData**(uint port, u8 * psrc, uint len, u8 term, SUR_CALLBACK cb, uint tmo)

Summary Send some data to the UART port.

Details Try to send some bytes to the UART port. You may indicate a timeout value and callback function if you need to know the data has been fully sent out.

Comments about timeout and callback function:

- If cb is NULL and tmo is not 0, this API will block the current calling task until the data is fully sent out by the UART or the operation timed out.
- If cb is not NULL then tmo is ignored and this API will return immediate. Callback function will be called when all the data in the internal outgoing buffer is sent out. Callback function normally will be called even if the cable is dicsonnected, unless you closed the port.
- If cb is NULL and tmo is 0, this API will copy the data into the internal outgoing buffer and return immediately.

Comments about terminate byte:

- This API allows you to indicate terminate byte of one packet, for example string will use 0 as terminate character. Terminate byte is only checked when len is 0.

Pars

port	UART port number
psrc	Pointer to the data buffer
len	Data buffer length
term	Terminator byte. Ignored if len is not 0.
cb	Callback function if you want to know when the data is fully sent out.
tmo	Timeout value (milliseconds) you want to wait for the data to be fully sent out or SB_TMO_INF for infinite wait (not SB_OS_TMO_INF).

Returns

>=0	The data length copied to the internal outgoing buffer. 0 if timeout occurs or buffer is full.
-1	This UART port is not open.

See Also sur_InData()

Example

```
sur_OutData(1, pBuf, 10, 0, NULL, 100);
sur_OutData(1, (u8 *)“Hellow World”, 0, 0, NULL, 100);
```

2.12 Run Time Library

brtl.h and brtl.c handle issues related to the compiler's C run time library. In particular, it includes some C library header files that must be included before smx include files to avoid problems, as well as a few commonly used header files, for convenience. It also implements some functions that are not supplied by some RTLs, such as ultoa(). Add any functions that are missing for your compiler.

Possible unsupported C functions include:

```
char *  _itoa(int val, char *str, int radix);
char *  _ultoa(unsigned long val, char *str, int radix);
char *  _strupr(char *str);
int     _stricmp(const char *__s1, const char *__s2);
int     _strnicmp(const char *__s1, const char *__s2, size_t len);
```

3. Common Definitions

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

3.1 Configuration

bcfg.h defines the basic information about your system. We like to make things automatic when possible, so these settings are wrapped with conditionals. We recommend that you simplify our configuration files by removing these conditionals; just leave the one line for each setting that is correct for your system.

Operating System Selection

For RTOSes other than SMX, we have done only limited testing, so please consider our port to be a good start but not necessarily a drop-in solution. This is because these are competitors' products which we either do not have or have only a limited or old version. In some cases, we have done the implementation based only on printed documentation. We will help you resolve any problems.

SB_OS_SMX

Define this macro if you are using the SMX® RTOS.

SB_OS_CMX

Define this macro if you are using CMX.

SB_OS_ECOS

Define this macro if you are using eCos.

SB_OS_EMBOS

Define this macro if you are using Segger embOS.

SB_OS_FREERTOS

Define this macro if you are using FreeRTOS.

SB_OS_ITRON

Define this macro if you are using ITRON.

SB_OS_MQX

Define this macro if you are using Freescale MQX.

SB_OS_NUCLEUS_PLUS

Define this macro if you are using Nucleus Plus.

SB_OS_POWERPAC

Define this macro if you are using IAR PowerPac.

SB_OS_QUADROS

Define this macro if you are using Quadros.

SB_OS_RTX

Define this macro if you are using Keil RTX.

SB_OS_THREADX

Define this macro if you are using ThreadX.

SB_OS_UCOS_II

Define this macro if you are using uC/OS II.

SB_OS_UCOS_III

Define this macro if you are using uC/OS III.

SB_OS_VRTX

Define this macro if you are using VRTX.

SB_OS_VXWORKS

Define this macro if you are using VxWorks. **The porting layer implementation has not been tested or even compiled. It is based on the online documents of VxWorks. You must check and fix any problems in the implementation.**

SB_OS_NORTOS

Define this macro if you are not using any RTOS.

SB_MULTITASKING

Set to 1 to run smxBase under a multitasking environment, such as SMX or ECOS.
Set to 0 to run smxBase under a non-multitasking environment, such as DOS.

We tested all the OS porting macros for the above cases using simple test code and evaluation packages provided by the vendors of those RTOS on some boards we have (except VxWorks). Interrupt-related BSP functions have not been tested because those are processor- and board-specific. Unless you are using SMX, you must check that the implementations are correct for your environment.

CPU Architecture

Define one of the following CPU architectures. You can add your own CPU type and add the corresponding section in bdef.h.

SB_CPU_ARM
SB_CPU_BLACKFIN
SB_CPU_COLDFIRE
SB_CPU_POWERPC
SB_CPU_RX
SB_CPU_SUPERH

CPU Operation Size

Define one of the following CPU operation sizes. 64-bit has not been tested.

SB_CPU_16BIT
SB_CPU_32BIT
SB_CPU_64BIT

CPU Memory Addressing Granularity

SB_CPU_MEM_ADDR_8BIT

Most processors can address 8-bit data, in which case this should be set to 1. However, some TI DSPs, for example, only allow accessing 16-bit values. This is a big problem for protocol data structures that have 8-bit field and are packed. In this case, set this to 0 so code is enabled to handle the problem. Not all middleware products support this feature. Check the User's Guide for each product.

CPU I/O Type

SB_CPU_MEM_MAPPED_IO

Set to 1 if you can access the peripheral's registers as addresses in memory space. This is the case for most processors. Set to 0 for x86 which has a separate I/O address space.

Interrupt Settings

SB_CFG_IRQ_MAX_NUM

Maximum IRQ number that will be used in your system. It is unlikely you will use all the IRQs of your processor so just set it to the maximum number used by the drivers used by middleware and your application.

SB_CFG_IRQ_MAX_SHARED

Maximum IRQ which will share the same IRQ number. If you are using an x86 processor and PCI bus, it is likely multiple devices will be assigned the same IRQ number. For example a PCI Ethernet and USB controller may both use IRQ 12. Set to 1 if none are shared.

Data Types

SB_CFG_EXACT_WIDTH_TYPES

C99 compilers provide new integer data types which have exact widths, to allow writing more portable code. If your compiler supports this feature, set it to 1.

SB_CFG_STDBOOL

C99 compilers provide a built-in _Bool data type and a new header stdbool.h that maps bool onto it for C compiles. (bool is a C++ data type.) If your compiler supports this feature, set it to 1.

SB_CFG_INT64_TYPE

Set to 1 if your compiler supports 64-bit integers. Some use a special data type for this, so if you get a syntax error in bdef.h where u64, etc. are defined, check if your compiler has an alternate syntax. Otherwise, set this to 0, and you will not be able to use this data type or any of our code that uses it.

Console I/O

See the section APIs/ Base and Utility/ Message Display Functions for information about OMB and OMQ and for more guidance in setting the SB_CFG_MSGOUT constants.

SB_CON_IN
SB_CON_OUT

Set to 1 to enable console input and output routines. On most targets this is via RS232 connection to a terminal or terminal emulator, such as a PC running TeraTerm.

SB_CFG_OMB_SIZE

Set to the Output Message Buffer size (in bytes). OMB holds variable messages that were built in buffers in RAM until they are sent out the UART. They are copied here because the code that created them may create a new message in those buffers.

SB_CFG_OMQ_SIZE

Set to the Output Message Queue size (in records). OMQ points to constant messages, typically in ROM, until they are sent out the UART.

SB_CFG_MSGOUT_DIRECT

Set to 1 to output messages directly to the UART, without buffering in OMB and OMQ. This slows down operation of the code that displays messages when using a polled UART driver, but it ensures no messages will be lost, and it saves RAM since there is no OMB or OMQ.

SB_CFG_MSGOUT_VARMSG

Set to 0 to permit displaying only constant messages. This omits the OMB, saving RAM. This option could be used on severely RAM-constrained systems. Set to 1 to also allow

outputing variable messages. (This setting is ignored if SB_CFG_MSGOUT_DIRECT == 1.)

SB_CFG_MSGOUT_DELAY

Set to the number of milliseconds that sb_MsgOutConst() and sb_MsgOutVar() should wait after writing a message. This is useful when using an interrupt-driven serial driver for terminal output and debugging, so that the message will finish printing on the terminal before the next statement, when stepping through the code.

Other Configuration

SB_CFG_OSPORT_USE_MUTEX

For some OS ports such as SMX, it is possible to use a mutex or a counting semaphore to protect critical sections. Mutexes may offer advanced capabilities such as avoidance of priority inversion, but they may add more complexity. A counting semaphore is adequate in many systems. Set to 1 to use mutexes; 0 to use semaphores.

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	may be int or unsigned char
booli	always int type
false	
true	

BOOLEAN
FALSE
TRUE
NULL
OFF
ON

If SB_CFG_EXACT_WIDTH_TYPES is set to 1 in bcfg.h, u32 and similar are defined using C99 built-in exact-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 BOOLEAN (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.

Processor Architecture

SB_CPU_ARCH should be one of the following:

SB_UNKNOWN
SB_X86
SB_POWERPC
SB_COLDFIRE
SB_SUPERH
SB_ARM
SB_68K
SB_ARMM
SB_BLACKFIN
SB_RX

SB_PROCESSOR is combination the processor architecture and endian information.

SB_BIG_ENDIAN indicates that the processor's memory addressing puts the most significant byte first. Some processors allow selecting big- or little-endian mode.

SB_LITTLE_ENDIAN indicates that the processor's memory addressing puts the least significant first. Some processors allow selecting big- or little-endian mode.

4. Porting Layer

4.1 Processor Architecture

barm.h, barmm.h, bcf.h, etc. 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_DATA_ALIGN_32
SB_STACK_ALIGN
sb_HALTEXEC()
sb_DEBUGTRAP()
sb_INT_ENABLE()
sb_INT_DISABLE()
__LITTLE_ENDIAN__
CPU_FL
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. See 4.4 Interrupt Service Routines (ISRs) for additional discussion about interrupt shells. __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.
__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 Operating System

The OS porting macros and functions (sb_OS_) are intended for use by SMX middleware modules such as smxFS, smxNS, smxUSB, etc. not by your application code. Your code should call kernel and BSP services directly.

bos.h and bos.c are the operating system porting files. Below, we show the function prototype of each API, but **most are implemented as macros** that directly map to the native OS APIs. This is done to show the expected parameter and return types and how to define them as functions when necessary.

Keep in mind that the OS porting layer implements the superset of macros and functions needed by all modules of SMX. Since your particular release is a subset, you can save time by **only implementing what is needed**. For example, smxFS, smxUSBD, and smxUSBH do not use block pools, and smxNS does not use mutexes. We suggest you search your release to see what is used before implementing anything. For example, search for sb_OS_MUTEX to see if any of the mutex macros are used.

A. Task-related functions:

BOOLEAN **sb_OS_TASK_CREATE_PREEMPTIBLE**(SB_OS_TASK_HANDLE *pTaskHandle, SB_OS_PTASKFUNCPAR mainFunc, uint parameter, uint priority, uint stack, const char * name);

Summary Creates and start a preemptible task.

Details This function combines task create and start into a single operation because that is what many OSes do (not SMX). Also it takes a parameter, and the task main function has a pararameter, because many OSes require all task main functions to have a parameter.

The task handle is a structure that stores the task's main function and stack addresses. In many OSes, such as SMX, the TCB is such a structure and is pre-allocated by the system, so the handle is a pointer to the TCB. However, some OSes do not provide such a structure so we define it in bos.h.

Pars

pTaskHandle	The handle of the task. See discussion above.
mainFunc	The main function of this task.
parameter	The parameter you want to pass to the task.
priority	The task's priority. The value is OS-specific.
stack	The stack size of this task.
name	The name of this task.

Returns

TRUE	This task is created and started.
FALSE	Create or start task failed.

BOOLEAN **sb_OS_TASK_CREATE_NONPREEMPTIBLE** (SB_OS_TASK_HANDLE *pTaskHandle, SB_OS_PTASKFUNCPAR mainFunc, uint parameter, uint priority, uint stack, const char * name);

Summary Creates and start a non-preemptible task. This function is not used by any smx middleware

Details This function combine task create and start into a single operation because that is what many OSes do (not smx). Also they take a parameter and the task main function has a pararameter because many OSes require all task main functions to have a parameter.

The task handle is a structure that stores the task's main function and stack addresses. In many OSes, such as SMX, the TCB is such a structure and is pre-allocated by the system, so the handle is a pointer to the TCB. However, some OSes do not provide such a structure so we define it in bos.h.

Pars

pTaskHandle	Task handle. See discussion above.
mainFunc	Task main function.
parameter	Parameter to pass to the task.
priority	Priority of the task. The value is OS-specific.
stack	Stack size for the task.
name	Name of the task.

Returns

TRUE	Task created and started.
FALSE	Create or start failed.

void **sb_OS_TASK_DELETE(SB_OS_TASK_HANDLE** *pTaskHandle);

Summary Deletes a task.

Details This function deletes a task. For example, the system exit task can call this function to delete all the application tasks. Some OSes may require the application to let the task exit (return from their main function) first and then delete that task.

Pars pTaskHandle Task handle.

Returns none

void **sb_OS_TASK_FINISHED**(void);

Summary Does any needed task termination code.

Details Some OSes may need to call a special system API before a task exits its main function, such as to delete itself. VRTX does not even allow a task to return from its main function. This function can be used to workaround this case. If your system does not have this requirement, define it as an empty macro.

Pars none

Returns none

SB_OS_TASK_ID **sb_OS_TASK_GET_CURRENT**(void);

Summary Returns the ID of the current task.

Details Only smxFS needs this function, to setup the current working directory of each task. A task ID is some kind of unique identifier of the task. In many OSes such as SMX, the handle and ID are the same. The OSes that do not have a task handle (in the sense of our porting layer) have only a task ID.

Pars none

Returns The current task ID.

SB_OS_TASK_ID **sb_OS_TASK_HANDLE_TO_ID**(SB_OS_TASK_HANDLE *pTaskHandle);

Summary Converts a task handle to a task ID.

Details Only smxNS uses this function, to delete the current task.

Pars pTaskHandle Task handle from create task.

Returns The task ID of the pTaskHandle.

BOOLEAN **sb_OS_TASK_IS_CURRENT**(SB_OS_TASK_ID pTaskId);

Summary Checks if pTaskId is the current task ID.

Details Only smxFS and smxNS use this function, to check the current task.

Pars pTaskId Task ID to check.

Returns TRUE TaskId is current task ID.
FALSE TaskId is not current task ID

void **sb_OS_TASK_PREEMPT_ALLOW**(void);

Summary Makes the current task preemptible.

Details In some cases, the best way to implement this function is by enabling interrupts. If your OS is non-preemptible, you may need to implement this function as an empty macro.

Pars none

Returns none

.

void **sb_OS_TASK_PREEMPT_BLOCK**(void);

Summary Makes the current task non-preemptible.

Details In some cases, the best way to implement this function is by disabling interrupts. If your OS is non-preemptible, you may need to implement this as an empty macro.

Pars none

Returns none

void **sb_OS_TASK_START_PREEMPTIBLE**(void);

Summary Used at the start of tasks to make them preemptible.

Details Makes the task preemptible for OSes that start a task non-preemptible (such as smx pre-v4.) If your OS starts tasks preemptible, implement this as an empty macro.

Pars none

Returns none

void **sb_OS_TASK_YIELD**(void);

Summary Yields the current task to let other tasks run.

Details Only smxNS uses this function. If your OS is non-preemptible, you may need to implement this as an empty macro.

Pars none

Returns none

B. ISR-related functions:

void **sb_OS_INT_DISABLE**(void)

Summary Disables interrupts using the processor's interrupt flag.

Details Disable/enable interrupts can be used to protect global variables that are modified both in ISR/LSR or task. It can also be used during the initialization of the system to protect unexpected interrupts.

Pars none

Returns none

void **sb_OS_INT_ENABLE** (void)

Summary Enables interrupts using the processor's interrupt flag.

Details Disable/enable interrupts can be used to protect global variables that are modified both in ISR/LSR or task.

Pars none

Returns none

void **sb_OS_IRQ_CLEAR**(int irq)

Summary Clears (acknowledge) the specified IRQ.

Details You may need to acknowledge the interrupt in the ISR for that interrupt so the interrupt controller won't keep generating the same interrupt. Some processors and/or interrupt controllers may not need the software to clear/acknowledge interrupt at all. For this case, implement it as an empty macro.

Pars irq IRQ number

Returns none

void **sb_OS_IRQ_END**(int irq)

Summary Signals end of interrupt for the specified IRQ.

Details Some interrupt controllers may need to write a register to signal end of interrupt, so it can generate this interrupt again. If your interrupt controller does not need EOI, implement it as empty macro.

Pars irq IRQ number

Returns none

void **sb_OS_IRQ_VECT_SET**(int irq, ISR_PTR func);

Summary Hooks the ISR to the interrupt vector table.

Details This API hooks an ISR to the interrupt vector table of your processor.

Pars irq IRQ number
func ISR function

Returns none

ISR_PTR **sb_OS_IRQ_VECT_GET**(int irq);

Summary Gets the ISR for the specified IRQ from the interrupt vector table.

Details This API gets the address of the ISR currently hooked for the specified IRQ in the interrupt vector table.

Pars irq IRQ number

Returns The interrupt vector associated with that IRQ.

void **sb_OS_IRQ_MASK**(int irq);

Summary Masks (disables) the specified IRQ.

Details This API only disables one IRQ, not all.

Pars irq IRQ number

Returns none

void **sb_OS_IRQ_UNMASK**(int irq);

Summary Unmasks (enables) the specified IRQ.

Details This API only enables one IRQ, not all.

Pars irq IRQ number

Returns none

void **sb_OS_ISR_ENTER**(void);

Summary Code required by your OS to be at the start of an ISR.

Details This API is OS-specific. Most OSes that require this operation provide a system call or macro for this. You may define it as a macro to map this API to that system call.

Pars none

Returns none

void **sb_OS_ISR_EXIT**(void);

Summary Code required by your OS to be at the end of an ISR.

Details This API is OS-specific. Most OSes that require this operation provide a system call or macro for this. You may define it as a macro to map this API to that system call.

Pars none

Returns none

BOOLEAN **sb_OS_ISR_CFUN_INSTALL**(int irq, uint param, SB_OS_PISRFUNC func, SB_OS_PLSRFUNC lsr, const char *name);

Summary Installs a C function as an ISR into the smxBase built-in dispatcher.

Details This function will not install that function into the vector table. The C function is called through the smxBase built-in ISR dispatcher. That dispatcher is implemented in BSP\isrshells.c or isrshells.s. Please see section 4.4 Interrupt Service Routines (ISRs).

Pars	irq	IRQ number.
	param	The parameter you want to pass to the C ISR. You cannot pass a parameter to the real ISR in the interrupt vector table, but the smxBase built-in dispatcher will allow you to pass a parameter to your C ISR. This feature is useful especially when you need to use one interrupt handler code to handle identical perapheral interrupts, such as two identical USB/Ethernet controllers.
	lsr	LSR linked with the ISR.
	name	Name of the ISR. For debug purposes only.

Returns	TRUE	C ISR has been installed into the dispatcher.
	FALSE	Could not install the C ISR.

BOOLEAN **sb_OS_ISR_RESTORE**(int irq);

Summary Restores the ISR replaced by a previous call to sb_OS_ISR_CFUN_INSTALL().

Details This function can be used to restore the old ISR installed to the interrupt vector table.

Pars	irq	IRQ number.

Returns	TRUE	ISR has been restored.
	FALSE	Could not restore the ISR.

void **sb_OS_LSR_INVOKE**(SB_OS_PLSRFUNC lsr, uint par);

Summary Invokes a link service routine (LSR) that allows you call OS APIs before the ISR returns to the current task.

Details If your sytem does not need a link service routine, map this function to a macro to call the LSR directly with the parameter.

Pars	lsr	Link service routine.
	par	Parameter to pass to the link service routine.

Returns none

C. Pipe, Mutex, and Semaphore-related functions:

Pipes are used to transfer data between tasks or between an ISR and a task (via LSR). The OS may not have a pipe, but you can implement it with a mailbox, message, or queue. Pipes are only needed by smxUSBH in the symmetric multi-processing (SMP) case and smxNS.

Mutex macros are used to protect access to data structures or critical sections of code. They can be implemented with a semaphore, if desired or if the OS does not have a mutex. Mutexes should be available to the first caller. smxNS does not use any mutexes.

Semaphore macros are used to signal completion of events. Some OSs may call this type of object an event or a signal. Semaphores should not test true until signaled. A binary semaphore can only have a count of 0 or 1. This type is useful when it does not matter how many times it was signaled, just that it was signaled.

For the mutex and semaphore wait macros, a return of 1 means the task got the mutex or semaphore within the timeout duration; 0 means it timed out. Be careful when implementing these, since some OSs return 0 for success.

SB_OS_PIPE_HANDLE **sb_Pipe_Create**(uint width, uint number, const char *name);

#define **sb_OS_PIPE_CREATE**(p, w, n, name) p = sb_Pipe_Create(w, n, name)

Summary Creates a pipe.

Details Some OSes may not support naming a pipe. For that case, ignore the name parameter; it is only for debug purposes.

Pars

w	Width (in bytes) of each element in the pipe.
number	Maxmium number of elements in the pipe,
name	Name of the pipe.

Returns Handle of the pipe.

void **sb_OS_PIPE_DELETE**(SB_OS_PIPE_HANDLE *p);

Summary Deletes a pipe.

Details Deletes a pipe created by sb_OS_PIPE_CREATE(). After you delete it, you cannot send/receive data to/from the pipe.

Pars

p	Pointer to pipe handle.

Returns none

BOOLEAN **sb_OS_PIPE_GET_INF**(SB_OS_PIPE_HANDLE *p, void *dp);

Summary Tries to get an element from the pipe, waiting forever.

Details If the data is not ready, the current task will be suspended forever. This function copies the element to the user memory pointed to by dp.

Pars

p	Pointer to the pipe handle.
dp	Pointer to the data (one element).

Returns

TRUE	Got data.
FALSE	Error occurred when retrieving the data, such as pipe has been deleted.

BOOLEAN **sb_OS_PIPE_GET_TMO**(SB_OS_PIPE_HANDLE *p, void *dp, uint tmo);

Summary Tries to get an element from the pipe, waiting only until timeout.

Details If the data is not ready, the current task will be suspended for the specified timeout. This function copies the element to the user memory pointed to by dp.

Pars

p	Pointer to the pipe handle.
dp	Pointer to the data (one element).
tmo	Timeout value in milliseconds. **0 means no wait.** If your OS uses 0 for INF or other meaning, you must check for 0 and handle specially.

Returns

TRUE	Got data.
FALSE	Timeout or error occurred when retrieve the data, such as pipe has been deleted.

void **sb_OS_PIPE_PUT**(SB_OS_PIPE_HANDLE *p, void *dp);

Summary Puts data into the pipe.

Details This operation wakes up a task that is waiting on the pipe. It copies the element in user memory pointed by dp to the pipe buffer.

Pars

p	Pointer to the pipe handle.
dp	Pointer to the data (one element).

Returns none

SB_OS_MUTEX_HANDLE **sb_Mutex_Create**(const char *name);

#define **sb_OS_MUTEX_CREATE**(m, name) m = sb_Mutex_Create(name)

Summary Creates a mutex.

Details Some OSes may not support naming a mutex. For that case, ignore the name parameter; it is only for debug purposes.

Pars name Name of the mutex.

Returns Handle of the mutex.

void **sb_OS_MUTEX_DELETE**(SB_OS_MUTEX_HANDLE *m);

Summary Deletes a mutex.

Details Deletes the mutex created by sb_OS_MUTEX_CREATE(). First releases the mutex if that is not done by the OS's mutex delete function.

Pars m Pointer to the mutex handle.

Returns Handle of mutex.

BOOLEAN **sb_OS_MUTEX_GET_INF**(SB_OS_MUTEX_HANDLE *m);

Summary Tries to get a mutex, waiting forever.

Details Before accessing a critical section for a shared resource, this is called to get the mutex. The current task will be blocked until this mutex is available (infinite timeout).

Pars m Pointer to the mutex handle.

Returns TRUE Got the mutex
FALSE Error occurred while getting the mutex.

BOOLEAN **sb_OS_MUTEX_GET_TMO**(SB_OS_MUTEX_HANDLE *m , uint tmo);

Summary Tries to get a mutex, waiting up to timeout.

Details Before accessing a critical section for a shared resource, this is called to get the mutex. The current task will be blocked until this mutex is available or it times out.

Pars m Pointer to the mutex handle.
tmo Timeout value in milliseconds. **0 means no wait.** If your OS uses 0 for INF or other meaning, you must check for 0 and handle specially.

Returns TRUE Got the mutex
FALSE Error occurred while getting the mutex or timed out.

void **sb_OS_MUTEX_RELEASE**(SB_OS_MUTEX_HANDLE *m);

Summary Releases a mutex.

Details Releases a mutex when leaving the critical section for a shared resouce.

Pars m Pointer to the mutex handle.

Returns none

SB_OS_SEM_HANDLE **sb_Sem_Create**(uint thres, const char *name);

#define **sb_OS_SEM_CREATE**(s, thres, name) s = sb_Sem_Create(thres, name)

Summary Creates a counting semaphore.

Details Some OSes may not support naming a semaphore. For that case, ignore the name parameter; it is only for debug purposes.

Pars thres Threshold. The number of signals required before a test will pass. For event notifications, thres is normally 1.
name Name of the semaphore.

Returns Handle of the semaphore.

void **sb_OS_SEM_DELETE**(SB_OS_SEM_HANDLE *s);

Summary Deletes a counting semaphore.

Details Deletes the semaphore created by sb_OS_SEM_CREATE().

Pars s Pointer to the semaphore handle.

Returns none

BOOLEAN **sb_OS_SEM_WAIT_INF**(SB_OS_SEM_HANDLE *s);

Summary Waits for a counting semaphore, forever.

Details The current task will be blocked until this semaphore is signaled (infinite timeout).

Pars s The pointer of semaphore handle.

Returns TRUE semaphore was signaled
FALSE Error occurred while waiting the semaphore.

BOOLEAN **sb_OS_SEM_WAIT_TMO**(SB_OS_SEM_HANDLE *s, uint tmo);

Summary Waits for a counting semaphore, up to the timeout.

Details The current task will be blocked until this semaphore is signaled or the wait times out.

Pars s Pointer to the semaphore handle.
tmo Timeout value in milliseconds. **0 means no wait.** If your OS uses 0 for INF or other meaning, you must check for 0 and handle specially.

Returns TRUE Semaphore was signaled.
FALSE Error occurred or the wait timed out.

void **sb_OS_SEM_SIGNAL**(SB_OS_SEM_HANDLE *s);

Summary Signals a counting semaphore.

Details This API wakes up the task that is waiting on the semaphore.

Pars s Pointer to the semaphore handle.

Returns none

void **sb_OS_SEM_SIGNAL_ISR**(SB_OS_SEM_HANDLE *s);

Summary Signals a counting semaphore from an ISR.

Details This API wakes up the task that is waiting on that semaphore. It can be called in the ISR. If the OS does not support signalling a semaphore in an ISR, define this as a

macro that maps to sb_OS_SEM_SIGNAL(), and smx middleware will call it in an LSR instead of the ISR.

Pars s Pointer to the semaphore handle.

Returns none

SB_OS_SEMB_HANDLE **sb_Semb_Create**(const char *name);

#define **sb_OS_SEMB_CREATE**(s, thres, name) s = sb_Semb_Create(thres, name)

Summary Creates a binary semaphore.

Details Some OSes may not support naming a semaphore. For that case, ignore the name parameter; it is only for debug purposes.

Pars name Name of the semaphore.

Returns Handle of the semaphore.

void **sb_OS_SEMB_DELETE**(SB_OS_SEMB_HANDLE *s);

Summary Deletes a binary semaphore.

Details Deletes the semaphore created by sb_OS_SEMB_CREATE().

Pars s Pointer to the semaphore handle.

Returns none

BOOLEAN **sb_OS_SEMB_WAIT_INF**(SB_OS_SEMB_HANDLE *s);

Summary Waits for a binary semaphore, forever.

Details The current task is blocked until this semaphore is signaled (infinite timeout).

Pars s Pointer to the semaphore handle.

Returns TRUE Semaphore was signaled
FALSE Error occurred while waiting for the semaphore.

BOOLEAN **sb_OS_SEMB_WAIT_TMO**(SB_OS_SEMB_HANDLE *s, uint tmo);

Summary Waits for a binary semaphore, for the specified timeout.

Details The current task is blocked until this semaphore is signaled or the wait times out.

Pars

s	Pointer to the semaphore handle.
tmo	Timeout value in milliseconds. **0 means no wait.** If your OS uses 0 for INF or other meaning, you must check for 0 and handle specially.

Returns

TRUE	Semaphore was signaled.
FALSE	Error occurred while waiting for the semaphore or the wait timed out.

void **sb_OS_SEMB_SIGNAL**(SB_OS_SEMB_HANDLE *s);

Summary Signals a binary semaphore.

Details This API wakes up the task that is waiting on the semaphore.

Pars

s	Pointer to the semaphore handle.

Returns none

void **sb_OS_SEMB_SIGNAL_ISR**(SB_OS_SEMB_HANDLE *s);

Summary Signals a binary semaphore from an ISR.

Details This API wakes up the task that is waiting on the semaphore. It can be called in the ISR. If the OS does not support signalling a semaphore in an ISR, implement/define this as a macro that maps to sb_OS_SEMB_SIGNAL(), and smx middleware will call it in an LSR instead of the ISR.

Pars

s	The pointer of semaphore handle.

Returns none

D. Block and Memory Management

Blocks and Block Pools are only used by smxNS for the web server.

void **sb_OS_BLOCK_POOL_CREATE**(SB_OS_POOL_HANDLE p, uint num, uint size, void *ptr, const char *name);

Summary Creates a block pool.

Details This API creates a block pool. Each block in the pool has the same size.

Pars

p	Handle of the block pool.
num	Total number of blocks in the pool.
size	Size of each block.
ptr	Starting address of the block pool memory.
name	Name of the block pool, for debug purposes only.

Returns none

void **sb_OS_BLOCK_POOL_DELETE**(SB_OS_POOL_HANDLE p);

Summary Deletes a block pool.

Details This API deletes the block pool created by sb_OS_BLOCK_POOL_CREATE().

Pars

p	Handle of the block pool.

Returns none

void **sb_OS_BLOCK_GET**(SB_OS_POOL_HANDLE p, SB_OS_BLOCK_HANDLE b);

Summary Gets a free block from the block pool.

Details This API gets a free block from the block pool.

Pars

p	Handle of the block pool.
b	Handle of the block.

Returns none

void **sb_OS_BLOCK_REL**(SB_OS_POOL_HANDLE p, SB_OS_BLOCK_HANDLE b);

Summary Releases a block back to the block pool.

Details This API puts a block back to the block pool so other task can use it again later.

Pars p Handle of the block pool.
b Handle of the block.

Returns none

void ***sb_OS_BLOCK_GET_PTR**(SB_OS_BLOCK_HANDLE b);

Summary Gets the memory pointer associated with a block handle.

Details This API gets the user memory pointer of a block.

Pars b Handle of the block.

Returns Pointer to the memory.

void ***sb_OS_MEM_ALLOC**(uint size)

Summary Allocates some memory from the heap.

Details This API allocates some memory from the heap. You can use the C heap (provided by the compiler) or your system's heap.

Pars size Size of the required memory.

Returns Pointer to the heap block.

void **sb_OS_MEM_FREE**(void *ptr);

Summary Frees some memory to the heap.

Details This API frees the memory allocated by sb_OS_MEM_ALLOC().

Pars ptr Pointer to the heap block to be freed.

Returns none

void ***sb_OS_MEM_REALLOC**(void *ptr, uint size)

Summary Re-allocates a heap block to change its size.

Details This API re-allocates the heap block to change its size. **It is not used by any smx middleware**.

Pars

ptr	Pointer to the heap block to re-allocate.
size	Size of the new heap block.

Returns Pointer to the new heap block.

E. Time and Delays

u32 **sb_OS_MSEC_GET**(void)

Summary Gets the number of milliseconds since the system started.

Details This API gets the number of milliseconds since the system started. It rolls over every 49 days.

Pars none

Returns Milliseconds since the system started.

u32 **sb_OS_TICKS_GET**(void)

Summary Gets the number of ticks since the system started.

Details Tick time is depends on OS configuration.

Pars none

Returns Ticks since the system started.

u32 **sb_OS_STIME_GET**(void)

Summary Gets the system time in seconds since January 1, 1970

Details Get the system time in seconds since January 1, 1970.

Pars none

Returns Seconds since January 1, 1970.

uint **sb_OS_TICKS_PER_SEC**(void);

Summary Gets the number of ticks per second.

Details Gets the number of ticks per second. Used to calculate the length of each tick.

Pars none

Returns Number of ticks per second.

void **sb_OS_WAIT_MSEC_MT**(u32 ms);

Summary Suspends the current task for the specified number of milliseconds.

Details Suspends the current task for at least the specified number of milliseconds, allowing other tasks to run..

Pars Time to suspend, in milliseconds.

Returns none

void **sb_OS_WAIT_MSEC_POLL**(u32 ms);

Summary Delays a few milliseconds.

Details Delays for at least the specified number of milliseconds using polling (i.e. a busy loop) rather than suspending the current task. Intended for short delays.

Pars Time to poll, in milliseconds.

Returns none

void **sb_OS_WAIT_USEC_POLL**(u32 ms);

Summary Delays a few microseconds.

Details Delays for at least the specified number of microseconds using polling (i.e. a busy loop) rather than suspending the current task. Intended for short delays.

Pars Time to poll, in macroseconds.

Returns none

F. Misc Macros

sb_OS_LINEAR_TO_POINTER(lin)	Converts a linear address to a pointer.
sb_OS_POINTER_TO_LINEAR(ptr)	Converts a pointer to a linear address.
sb_OS_OBJECT_HANDLE_CREATE(h, name)	Creates a pseudohandle for an object that has no handle. Used by smx to add names of ISRs, LSRs, etc. to the handle table for debugging.

G. Object Handles

The type definitions for handles vary according to the OS implementation. Normally, handles are defined as OS structures for Tasks, Mutexes, etc. These structures may need to be allocated using OS-specific CREATE() macros and freed with OS-specific DELETE() macros. Some OSs, such as SMX, provide pools of these data structures so they need not be dynamically allocated.

SB_OS_TASK_HANDLE	Task (thread) handle
SB_OS_PIPE_HANDLE	Pipe handle
SB_OS_MUTEX_HANDLE	Mutex handle
SB_OS_SEM_HANDLE	Counting Semaphore handle
SB_OS_SEMB_HANDLE	Binary Semaphore handle
SB_OS_TIMER_HANDLE	Timer handle
SB_OS_POOL_HANDLE	Pool handle
SB_OS_BLOCK_HANDLE	Block handle

H. ISR, LSR, and Task Function Data Types

SB_OS_PISRFUNC	ISR function with a (uint) parameter.
SB_OS_PLSRFUNC	LSR function with a (uint) parameter.
SB_OS_PTASKFUNCPAR	Task procedure function with a (uint) parameter, pointing to a task-specific parameter.

I. Task Priorities

SB_OS_TASK_PRI_MAX
SB_OS_TASK_PRI_HIGH
SB_OS_TASK_PRI_NORM
SB_OS_TASK_PRI_LOW

SB_OS_TASK_PRI_MAX is used for tasks that process real interrupt requests for the device, so it should be a very high priority to ensure the interrupt is processed immediately.

SB_OS_TASK_PRI_HIGH is used by some server tasks to process the received data (request), such as user input tasks.

SB_OS_TASK_PRI_NORM is used by some normal, non-time-critical tasks.

SB_OS_TASK_PRI_LOW is used for low priority tasks such as some demo tasks.

J. Test of OS Porting Layer

bostest.c can be used to test some of the porting layer, such as task, block pool, pipe, semaphore, mutex, and time functions. Interrupt-related macros are not included in this test, because it would be necessary for us to have a competitor's OS here running on a board, and even so, some OSes do not have a standard BSP layer, so interrupt handling varies for different targets.

4.4 Interrupt Service Routines (ISRs)

Normally in application code, you will use **sb_IRQVectSet()** to hook interrupts (and **sb_IRQConfig()** and **sb_IRQUnmask()** to configure and unmask them). The OS porting layer has macros that are used to hook interrupts, for use by SMX middleware modules, to avoid many conditionals in the code. These map onto appropriate functions for each OS. **It is recommended to use the sb_IRQ macros for hooking interrupts, not the sb_OS porting macros.** This is because the porting macro API is more likely to change than the smxBase API. (This can happen when we try to support a new OS in the porting layer that works differently from others we've supported so far.) In particular, you should not use sb_OS_ISR_CFUN_INSTALL(), except to support shared interrupts, because it adds extra overhead to ISR handling. This is explained in more detail in the following paragraphs.

Handling ISRs in portable middleware is a challenge. Different processors and OSs have different reqirements for ISRs, and different compilers use different keywords to declare ISR functions. Putting all the variations in the middleware core source code is unwise, so smxBase implements a built-in software ISR dispatcher to handle the details of the actual ISR. The middleware's C ISR function can focus on just the details related to that device and protocol.

In addition to providing a portable interface for ISR handling, the smxBase software interrupt dispatcher can support shared IRQs (i.e. used by multiple peripherals). The macro sb_OS_ISR_CFUN_INSTALL() registers a C function with the smxBase ISR dispatcher. The actual ISR hooked to the processor's interrupt vector table is one of the built-in ISR shell functions named sb_OS_ISRx(). sb_OS_ISR_CFUN_INSTALL() hides the details about shared interrupts, and from the driver's point of view, it's not even known that the IRQ is shared. This feature is especially important for PCI bus based devices. For example, if the USB controller and Ethernet Controller are both using IRQ 15, then the USB and Ethernet controller drivers do not need to know about each other in their own C function ISR. This makes the drivers more portable. If you need to support shared interrupts, you should use sb_OS_ISR_CFUN_INSTALL() to hook them.

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, e.g. ARM. For processors that require software vectoring rather than supporting hardware vectoring, the BSP has a simple dispatcher function. This dispatcher, sb_IRQDispatcher (irqdispatch.c), is hooked to the vector (or called from smx_irq_handler() in the case of smx), and it calls the ISR shell in the smxBase IRQ dispatcher, which calls the actual ISR function(s) in the driver(s).

5. Building the Library

After configuring bcfg.h and preinclude files in the CFG directory as appropriate for your system, build the library with the project file or makefile supplied in the build directory (e.g. IAR.ARM). It is built like other SMX module libraries, as documented in the SMX Quick Start. If a makefile is provided, run the mak.bat file to invoke it. Run mak.bat without arguments for syntax help.