



# smxAware™ User's Guide

Version 5.2.0  
February 14, 2024

by Marty Cochran and David Moore

<b>Introduction .....</b>	<b>1</b>
<b>Supported Debuggers.....</b>	<b>1</b>
<b>Installation .....</b>	<b>1</b>
Changes to the Application.....	1
IAR EWARM Directions .....	2
<b>Configuration.....</b>	<b>3</b>
<b>Using smxAware .....</b>	<b>6</b>
smxAware Dialog Box .....	6
Kernel Displays .....	8
SMX Middleware Module Displays .....	17
Print Window.....	22
Modal vs Non-Modal Dialog.....	22
Suspended Task Information .....	23
Task-Specific Breakpoints.....	23
<b>Graphical Analysis Tools (GAT).....</b>	<b>25</b>
Guides .....	25
Event Timelines .....	27
Profile .....	33
Stack Usage .....	35
Error Buffer .....	37
Event Buffer (text).....	38
Resource Usage .....	39
Memory Map Overview.....	39
Downloading the Event Buffer .....	46
Application Preparation .....	47
<b>smxAware Live .....</b>	<b>48</b>
Installation .....	48
Using smxAware Live .....	49
<b>Diagnostics .....</b>	<b>50</b>
Text Display Error Messages.....	50
GAT Error Messages .....	51
Diagnostic Logging .....	51
<b>Tips .....</b>	<b>51</b>
<b>Troubleshooting.....</b>	<b>51</b>

© Copyright 1996-2024

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

All rights reserved.

smxAware is a Trademark of Micro Digital, Inc.  
smx is a Registered Trademark of Micro Digital, Inc.

## Introduction

*Note: This manual was thoroughly revised for SMX v5.20, and much information for older versions and tools has been deleted. It is recommended for users of older versions to get the corresponding manual from our support site or by emailing us to request it.*

smxAware is a DLL that adds functionality to embedded debuggers, to give them SMX awareness. While stopped at a breakpoint, you can:

- Display information about kernel specific objects such as ready queue, tasks, semaphores, messages, events, heaps, stacks etc.
- Display graphs of Event Timelines, Profiling, Stack Usage, Memory Usage, and Memory Map Overview using the Graphical Analysis Tool (GAT) feature.
- Display print statements generated by the application.
- Set task-specific breakpoints. The breakpoint will be triggered only if it is hit while the specified task is running.
- Display the address where a suspended task will resume.

smxAware Live is a remote monitoring version that is also documented in this manual.

## Supported Debuggers

smxAware supports IAR EWARM/C-SPY<sup>®</sup> for ARM processors. ColdFire and PowerPC users, please see the v4.4 manual at [www.smxrtos.com/support](http://www.smxrtos.com/support).

Keep in mind that tools change. The smxAware files you have should work well with the tools your release was built with. If you upgrade to newer tools and you have problems, ask for an update of smxAware.

## Installation

### Changes to the Application

Enable the define of **SMXWARE** in the main preinclude file, e.g. CFG\iararm.h so smxaware\_init() is called, and add **smxaware.c** to your application project.

Also, for some tools it is necessary to compile XSMX\xglob.c with debug symbolics enabled. This should already be done in the project file we supply, but check it if you have trouble. Newer versions of smxAware display a warning that smxVersion can't be read, if debug symbolics are not enabled for xglob.c.

smxaware\_init() does two things:

1. Determines which version of smx is being used. This is important because there are differences in internal data structures such as the smx\_cf structure and control blocks.
2. Initializes the print window feature. See the section Print Window below.

Pass names in smx\_XxxxCreate calls in your application to assign names to smx objects, such as tasks, semaphores, and exchanges. This allows smxAware to print the name of these objects in the corresponding displays. smxAware creates a table to correlate names and handles. If smxAware is unable to find a handle in the table, it simply prints the handle value (hex) in place of the name.

To name ISRs, messages, and others that cannot be named in a Create call, create a pseudohandle. See section Pseudohandles. Also note that handles should be defined as global variables.

Next follow the directions in the appropriate section for your debugger, below.

## IAR EWARM Directions

There are big and little-endian versions of the DLL. The big-endian version is suffixed "BE"; the little-endian version is not suffixed. Some ARMs are one or the other, and some support both. Also, there are DLLs for different versions of EWARM. You must use the proper version of the DLL for your target and IAR version. (Newer versions of EWARM come with smxAware installed, but the version in your release may be newer, or you may download a newer version from our support site, so you may need to replace these files in EWARM.)

1. Copy the smxAware files from SMX\SA to this EWARM directory:  
**arm\plugins\rtos\SMX\** (Create the SMX subdirectory if it doesn't exist).  
Specifically, copy:

**smxAwareGAT.exe** and one **.dll** and **.ewplugin** file, as indicated below. (It is ok to copy both the big and little-endian files, but you must only copy the files for one version of EWARM.)

### EWARM v8

**smxAwareIarArm8.dll, smxAwareIarArm8.ewplugin** (little-endian)            or  
**smxAwareIarArm8BE.dll, smxAwareIarArm8BE.ewplugin** (big-endian)

### EWARM v7

**smxAwareIarArm7.dll, smxAwareIarArm7.ewplugin** (little-endian) or  
**smxAwareIarArm7BE.dll, smxAwareIarArm7BE.ewplugin** (big-endian)

and similar for previous versions. For v4 there is no numeric suffix.

2. Exit and restart EWARM if you are already in it, and then open the app project.
3. In project settings, select (in left pane): **Debugger**. In the right pane, select the **Plugins** tab. Select **smxAware** (little endian) or **smxAwareBE** (big endian) from the list.
4. Press the Debug button to download the app. “smxAware” should appear in the main menu if it is working.
5. **Run at least until after the call to smxaware\_init() in smx\_Go() before attempting to use smxAware.**

## Configuration

Configuration information is stored in smxAware.ini. Most items are set in the GAT Options Dialog, but some must be set by editing smxAware.ini.

### **smxAware.ini**

This file stores smxAware state and configuration settings. It is automatically created with default values if it does not exist, in C:\Windows\Users\<username>\AppData\Roaming\SMX. It stores some values about your previous session, such as whether you had certain buttons enabled and window size, and it stores values set in the Options dialog. It also has a few additional values that can only be configured by editing this file manually:

**maxFilesToSave:** Number of past event traces to keep.

**dataPath:** Location to save trace files. Defaults to the location of the smxAware.ini.

**diagnostics:** Set to the desired diagnostic logging level. See section Diagnostic Logging.

**stackScanTimeoutSeconds:** Maximum number of seconds to scan questionable stacks after each debug step or run. Any stacks that have not been scanned will continue to show Usage and % with “?” following in the text stack display and light blue bars in the graphical display. Increasing this setting will make each step take longer in the debugger when the text stack display is expanded.

[**TOP\_LEVEL\_ITEM\_SORT**]: Specifies the order in which to display items in the Text window. Change the values to put them in the desired order. If smxaware.ini is deleted, they will revert to the default order.

## Options Dialog

The Options dialog is accessed by pushing the Options button in the GAT toolbar. The settings are stored in smxAware.ini which is created in the same directory the traces are saved in (see section Saved Traces). See the section smxAware.ini for information about settings in this file.

### Font Size

This controls the size of the font and thickness of the bars in the client area of the displays. Setting it to a lower number will allow more lines to fit on the screen. Default is 12.

### Enable Event Capture (Filtering Events)

These checkboxes allow you to control what classes of events are logged: Task, SSR, LSR, ISR, Error, and User events. For example, un-checking “Log ISR Events” will prevent ISR events from being added to the Event Buffer the next time you run. This will allow capturing a longer trace before the Event Buffer fills up. If ISR events are excluded, for example, even if the ISR button is pushed on the graphical display, no ISRs will appear because there are no entries for them in the buffer.

For SSRs, a finer level of control is given to filter SSRs in 8 groups. You can assign SSRs into groups by changing the SSR Group field (3<sup>rd</sup> byte) in the SMX\_ID constants at the end of XSMX\defs.h. Group is bit  $n + 1$ . For example, 0x--01---- is group 1, 0x--20---- is group 6, etc. Then the checkboxes for SSRs can be checked to enable logging of each group.

Note: To avoid logging particular ISRs, comment out the smx\_EVB\_LOG macros in them. To avoid logging LSRs, pass flag SMX\_FL\_NOLOG to smx\_LSRCreat(). It is currently not possible to control which specific tasks to log. SSRs can be selectively enabled by group — see the smx Users Guide for details.

Each checkbox corresponds to a flag in the smx global **smx\_evben**. This global can be changed in the code, by setting it to the desired SMX\_EVB\_EN\_ flags (see xevb.h). The checkboxes will reflect the value of smx\_evben, so if a checkbox changes from how you last set it, the code must have changed smx\_evben.

### Window Action

This controls what happens with open smxAware windows each time you step or run your application in the debugger.

**Update after each run:** The smxAware windows remain open and are updated after each step. This can slow down stepping depending upon how much data smxAware has to retrieve for whatever is currently being displayed.

**Close window on each run:** The smxAware windows automatically close each time you step or run. The user can manually open the GAT window any time the target is stopped.

### **Guidelines**

These checkboxes allow you to enable light gray guidelines in the Event Timelines display, to make it easier to see how things line up. This is an alternative to using the Crosshairs tool.

**Horizontal Guidelines:** Lines are added between each row to line up horizontal events.

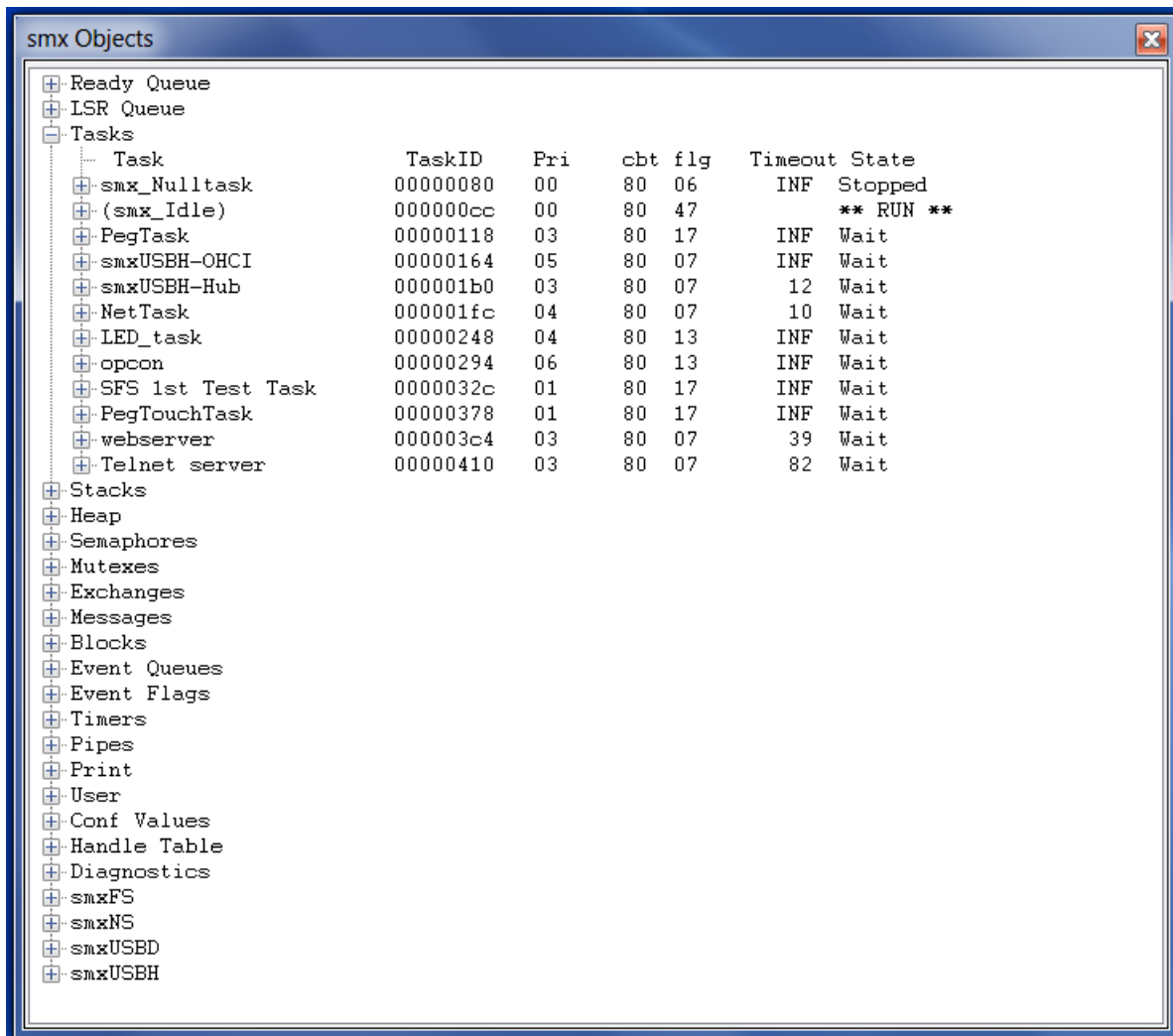
**Vertical Guidelines:** Vertical lines will be placed at each event to line up vertical events.

# Using smxAware

## smxAware Dialog Box

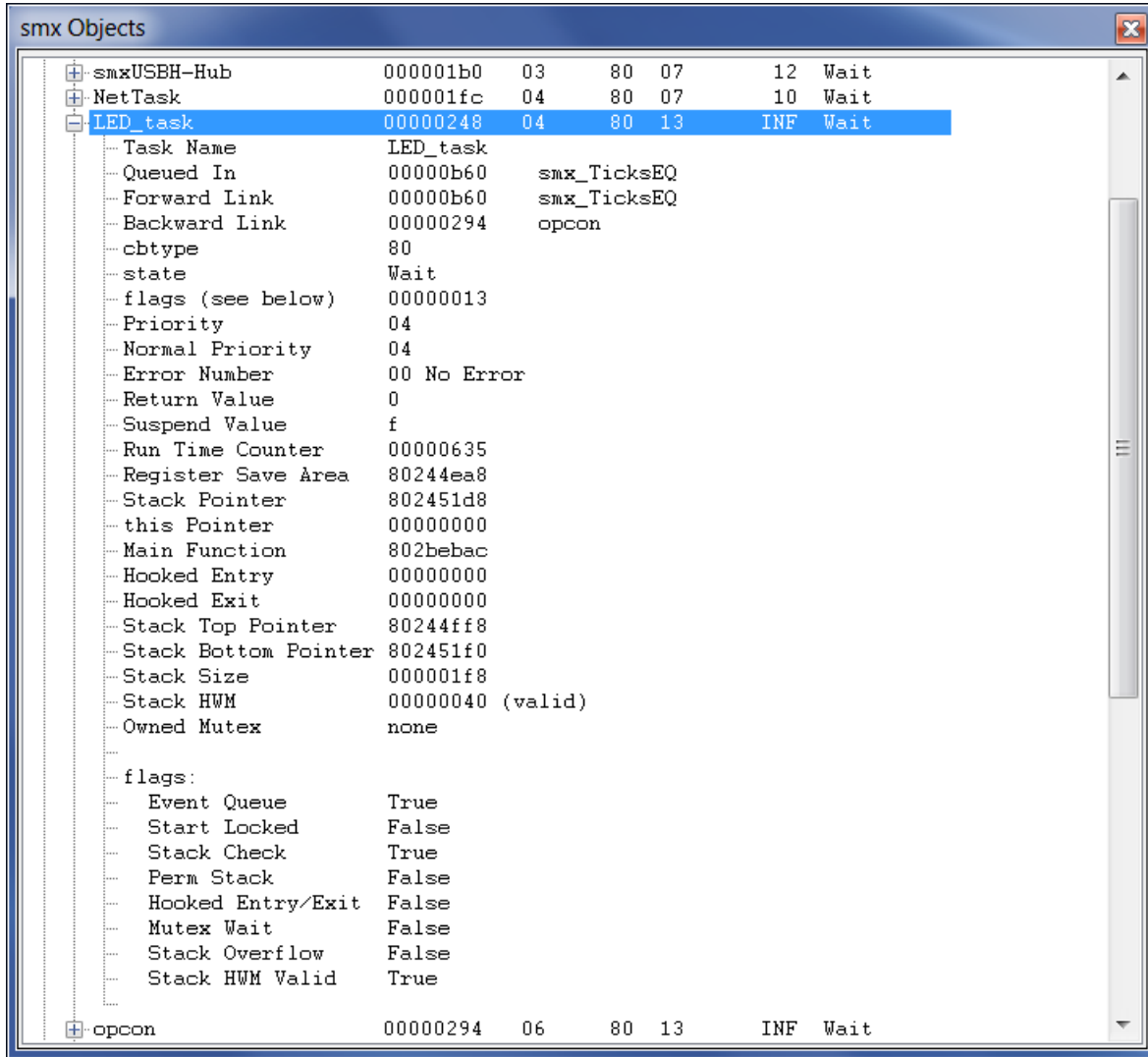
The smxAware dialog box lets you browse text displays of smx objects. It opens when selected from the main menu of the debugger/IDE. (The sections above describe how to start it from each debugger.) Use it while stopped at a breakpoint.

For **IAR EWARM**, the smxAware window is initially docked but can also be undocked. It is a single hierarchical tree. Here it is shown with Tasks expanded.





With a task expanded:



Other displays are similar.

### Copy to Clipboard:

IAR v9.40 and later: Pressing “c” will copy all of the text in the most recently expanded tree in the smxAware Text window to the clipboard. (To copy a tree already expanded, collapse and re-expand it. “Ctrl-c” is not supported.)

Other: Pressing “c” will copy all of the text in the current smxAware Text window to the clipboard. (“Ctrl-c” can also be used for IAR.)

## Kernel Displays

Below is a summary of the information displayed for each smx object type.

Note that the **order of these can be changed** by editing smxaware.ini. Change the values in the [TOP\_LEVEL\_ITEM\_SORT] section, to put them in the desired order. If smxaware.ini is deleted, they will revert to the default order.

### Ready Queue

Shows the tasks in each level of the ready queue, in order. Left-most is the first to be serviced. ( ) around the task name indicates ct. (Normally this will be the left-most task at the highest occupied level, but if it is locked and bumped with smx\_TaskBump(), it could appear at the end of the level, yet still be ct.)

### LSR Queue

#	Number of the LSR. 0 is next to run.
Address	LSR function address. Correlate to .map file.
Par	Parameter passed to LSR.

### LSRs

LSR	LSR name from LCB, or handle if not named.
Type	T for trusted; U for safe umode; P for safe pmode.
LsrID	LSR handle.
HostTask	Task LSR is associated with, if any.

Expanding an LSR shows:

Main Function	Main function address.
Flags	LSR flags (see below).
HostTask	Task LSR is associated with, if any.
Stack Top Pointer	Address of top of stack (end it grows to, not including pad).
Stack Bottom Pointer	Address of bottom of stack (end it starts from).
MPA Size	Number of slots in MPA for Safe LSR (SecureSMX)..
MPA Ptr	Address of MPA for Safe LSR (SecureSMX).
LSR Handle Ptr	Pointer to variable holding this LSRs handle.
Flags breakdown:	
Trust	1/0 Trusted LSR.
Umode	1/0 Safe LSR (umode/pmode).
Nolog	1/0 LSR not logged in EVB if True.

## Tasks

Note: Starting with smx v5, fields can be added to the end of the TCB, and smxAware will continue to work, and it will display the values as generic 32-bit values. This depends on having the following definition in smxaware.c: `u32 smx_tcb_size = sizeof(struct TCB)`. In the past we have advised against adding fields to the TCB because it will break smxAware. Now fields can be added, but only to the end, and there must be no other changes (which could break smx too).

Task	Task name from TCB, or handle if not named. ( ) around the task name indicates the current task.
TaskID	Task handle.
Mode	P is privileged, U is unprivileged. (SecureSMX)
Pri	Priority 0 to 127. 0 is the lowest priority. If two values are displayed, the first number is the current priority, and the second is the normal priority. See Own Pri field of Mutex display, below.
Flag	Task flags.
Err	* in this column means smx error occurred. For SecureSMX it may also mean there is an error in the Task's MPA. If this appears, expand the task and check the Error Number field and MPA.
Timeout	# of ticks (decimal) until the task will timeout. blank means task is not waiting or stopped (i.e. no timeout). INF means infinite timeout. Negative number means the timeout has happened but Timeout_LSR() has not yet moved the task to the ready queue.
State	Task state (Run, Ready, Sleep, Stopped, Wait (suspended), or WaitInf (suspended with infinite timeout)).
SuspLoc	Address where task was suspended. See section Suspended Task Information for details.

Expanding a task will display more task-specific information.

Task Name	Task name from TCB, or handle if not named.
Queued In	The queue it is in, if any. For the ready queue, it shows the ready queue level it is in (e.g. rq[3]).
Forward Link	Control block handle
Backward Link	Control block handle
Index	Index of task relative to first TCB.
Error	smx error number and name of last error caused by task. 00 No Error if none.
State	Task state
Flags	Task flags (see below)
Priority	Current priority of task (0 to 126 and >= Normal Priority)
Normal Priority	Normal priority of task (0 to 126)
Timeout Priority	Priority task is raised to if timeout occurs (0 to 126)
Return Value	Used by smx calls that cause the task to wait

Suspend Value	Used by some smx calls to save a value when they suspend, such as the differential count for tasks in an event queue.
Suspend Value 2 exret	Used by some smx calls to save a value when they suspend. Low byte of exception return value, which indicates type of stack frame for FPU register autosave (ARM-M).
Runtime Counter	Counter for runtime limiting or task profiling
Runtime Limit	Runtime limit (or pointer to top parent's rlim)
Runtime Limit Count	Runtime limit counter (or pointer to top parent's rlimctr)
Stack Pad Pointer	Pointer to pad above stack.
Stack Top Pointer	Address of top of stack (end it grows to, not including pad)
Stack Pointer	Stores stack pointer when task suspended
Stack Bottom Pointer	Address of bottom of stack (end it starts from)
Stack RSA Ptr	Pointer to part of stack where registers are saved (task context)
Stack HWM	Stack high water mark. Indicates stack usage, in bytes. Directly compares to Stack Size.
Stack Size	Usable bytes of stack (not including pad)
this Pointer	this pointer, for C++ tasks
Main Function	Address of task's main function (entry point)
Callback Function	Address of callback function for task exit, entry, init, etc.
Owned Mutex	Mutex name or handle. One line for each.
Suspended Location	Address where task was suspended. See section Suspended Task Information for details.
Task Handle Pointer srnest	Pointer to variable holding this task's handle. srnest when enter PendSVH/PreSched from SSRExitInt()

#### MPU / SecureSMX Fields:

Parent	Parent task.
Irq	Pointer to permitted IRQ table for task.
MPA Template Ptr	MPA template pointer for this task
MPA Pointer	MPA pointer for this task
MPA Size	Number of slots in MPA.
Dual Slot Number	Dual slot number (high nibble is aux slot).
Idle Counter	Number of idle passes per RTL frame.
Token Array Pointer	Pointer to token array for this task.

#### Flags breakdown:

Stack Check	1/0	Stack checking enabled for task
Stack HWM Valid	1/0	Indicates Stack HWM (above) is valid; the stack has been scanned since the last time the task ran.
Stack Overflow	1/0	Stack overflow detected, if true
Stack Perm	1/0	Task has permanently bound stack (not stack pool stack)
Stack Prealloc	1/0	Stack is preallocated block user passed.
Hooked	1/0	Callback routine hooked to save additional task state
In Priority Queue	1/0	Task is in a priority queue
In Event Queue	1/0	Task in event queue
Mutex Wait	1/0	Task is waiting to get a mutex.

Event Group AND/OR	1/0	Task is waiting on AND/OR of flags in Event Group
Event Group AND	1/0	Task is waiting on AND of flags in Event Group
Start Locked	1/0	Task starts locked.
Unpriv Mode	1/0	Unprivileged mode.
Pipe Front	1/0	Put packet to pipe front if 1, to back if 0.
Pipe Put	1/0	Task waiting to put packet to pipe.
Token Ok	1/0	Token test passed.
Rv_r0	1/0	Copy ct->rv to r0 in exframe on task stack.

## MPA (SecureSMX)

MPAs for all tasks are shown, using a format like the MPU display below. If the MPU has static slots, the regions show different indexes for MPA and MPU, and the static slots are displayed at the bottom of the MPA after a line.

## MPU (SecureSMX)

Details of the MPU are shown including:

Current Task	Current task name and handle.	
Flags	MPU ON/OFF	MPU is on/off
	BR ON/OFF	Background Region is enabled/disabled Remember that BR enabled for umode has no effect, but this is done for ISRs that interrupt utasks.
	(UN)PRIV MODE	Processor is currently in (un)privileged mode. Privileged if in Handler Mode, else is CONTROL.nPriv.
	MSP/PSP	Main Stack / Process Stack is in use
Caution	Warnings about overlaps	
MPU[n]	Information about each slot, including Start and End address, Size, Attributes, RBAR, RASR, and name of region.	
	Subreg Dis	Lists subregions that are disabled (0 to 7). Size indicates size of each subregion (1/8 of region size).
	Sub Start/End	Start and end address of enabled contiguous subregions. Size indicates their total size.

## Stacks (Task)

Task	Owner. Task name from TCB, or handle if not named.
StkTopAddr	Starting address of the memory block and top of stack. (Stack grows toward this end.)
Used	Amount of stack used (based on Stack HWM field in TCB). A “?” next to the value indicates that the value is questionable because the task has run since the last time its stack was scanned (task’s SHWM_VALID flag is 0), so it may have used more stack. See notes below.
Size	Size of memory block excluding padding.
%	Percent used (used/size * 100). “?” has the same meaning as for the Used column.

Type Bound, Shared, or None. Bound is a heap stack permanently allocated to the task; Shared is a stack from the stack pool that is released if the task stops (not if it suspends); None means the task currently does not have a stack (it stopped and released its shared stack).

The bottom of the window summarizes:

1. how many shared stacks are used out of the total number that exist.
2. how big of a pad is allocated at the logical top of each stack, if any.
3. how stack usage (HWM) is determined (i.e. by stack scanning or checking sp at task switches).

Shows entries for all stacks in use. It is done by listing all tasks, since this allows showing stack usage and HWM for tasks that currently don't have a stack. This information is independent of the particular stack assigned to the task; it reflects usage over the lifetime of the task.

The first line is the System Stack. This is used for ISRs, LSRs, scheduler, and error manager.

**The best way to view stack usage is graphically.** See the section Stack Usage for more information.

## Heap

The heap window shows the following main items for each heap:

Summary	Various statistics of heap usage and settings.
Allocated	List of allocated chunks (see below).
Donor Chunk	Size of donor chunk.
Top Chunk	Size of top chunk.
Small Bins	List of small bins. Summary line shows number of chunks and total space for all.
Upper Bins	List of upper bins. Summary line shows number of chunks and total space for all.

The Allocated and Bin items show these fields when expanded:

Type	Type of chunk: free, inuse, debug, start, end
Address	Block starting address. Address returned to the user where data will start.
Size	Block size of data part, excluding CCB and fences, if any.
Align	Alignment of data part (actual, which may be > requested).
SpSize	Size of spare space, if any.
ChunkAddr	Chunk starting address.
CSize	Chunk size.
bl	Backward link to previous chunk.
fl	Forward link to next chunk.
free bl	Backward link to previous free chunk in bin. (Only for free chunks.)
free fl	Forward link to next free chunk in bin. (Only for free chunks.)
Alloc Time	etime when chunk was allocated. (Only for debug chunks, i.e. CDCB.)
Owner	Task that allocated the chunk. (Only for debug chunks, i.e. CDCB.)

Fences            Ok or Broken (all fences should == SMX\_HEAP\_FENCE\_FILL.)  
 S/U/\*            Bin is sorted, unsorted, or being sorted. Applies only to upper bins.

## Semaphores

Name            Semaphore name from SCB, or handle if not named.  
 Handle          Semaphore handle.  
 count           Signal counter.  
 limit            The signal counter must reach this value before the top task(s) waiting at the semaphore will be resumed, for semaphore types that use a limit. See the Semaphores chapter of the smx User's Guide.  
 mode            Type of semaphore, e.g. binary resource.  
 Callback        Address of callback function.

Expanding a semaphore will display the forward and backward links and fields listed above and:

    Tasks Waiting            Names or handles of tasks waiting for the semaphore.

## Mutexes

Name            Mutex name from MUCB, or handle if not named.  
 Owner Task     Owner task's name from TCB, or handle if not named.  
 Own Pri        Owner task's priority. If two values are displayed, the first number is the current priority, and the second is the normal priority. The current priority is >= normal priority. Normal priority is the original priority of the task before promotion due to ceiling or priority inheritance.  
 nest            Nesting count.  
 pi              Priority inheritance enabled (if != 0).  
 ceil            Ceiling priority.  
 mtxp           Name or handle of next mutex in list of mutexes owned by a task. (The head of the list is pointed to by the task's TCB.mtxp.) If NULL, this mutex is either not owned or is the last mutex in the list.

Expanding a mutex will display all tasks waiting to get it, in priority order.

## Exchanges

Name            Exchange name from XCB, or handle if not named.  
 Handle          Exchange handle.  
 tq              1: one or more tasks is queued, 0: no tasks are queued.  
 mq              1: one or more messages are queued, 0: no messages are queued.  
 Status         Number of Messages Enqueued or Tasks Waiting.  
 Callback        Address of callback function.

Expanding an exchange will display the queue with any tasks or messages queued.

## Messages

Name	Name or handle of message. It is rare that messages are named so usually this will be the handle.
Owner	Message owner or “free” for a free message and “unused” for an unused MCB. (A free message is one with an allocated buffer but that is not owned.) Usually this will be a task handle. It can also be a pipe handle or LSR handle if an LSR received it. When the message is in an exchange, this field stores the handle of the exchange it is in, if any. However, in that case, the exchange name or handle is displayed in the Exchange field instead.
Pri	Message priority.
Exchange	The exchange name or handle that the message is in, if any.
Block	Pointer to the message buffer.
Pool	Pool the block is from.

Expanding a message will display other MCB fields not shown in the summary (1-line) display:

Forward Link	Control block handle
Backward Link	Control block handle
Block Source	Heap or pool from which block came, or -1 if from free block.
Reply Index	Index of the handle of the object to reply to among QCBs (typically an exchange handle)

Other fields for ARMM7 and ARMM8: See SecureSMX manual.

## Blocks

Name	Name or handle of block. It is rare that blocks are named so usually this will be the handle.
Owner	Block owner or “free” for a free block. (A free block is one with an allocated buffer but that is not owned.) Usually this will be a task handle, but it can be an LSR handle if an LSR got it.
Pool	The name or handle of the block pool the block is in.
Block Pointer	Pointer to the data area of the block.
Size	Block size (decimal).

## Event Queues

Name	Event name from EQCB, or handle if not named.
Handle	Event handle.
Callback	Address of callback function.

Expanding an event will display the tasks queued along with priority and count. The counts are converted from differential count to absolute number of counts until each is resumed.

## Event Groups

Name	Event group name from EGCB, or handle if not named.
Handle	Event group handle.



flags           Hex image of flags set with `smx_EventFlagsSet()`.  
 Callback       Address of callback function.

Expanding an event group will display the tasks queued at each slot and the following information:

Flags	Flags currently set
TestMask	Test mask
ClearMask	Clear mask
AND, OR, or AND/OR	Indicates which type of condition the task is waiting for.

## Timers

Name	Timer name from TMCB, or handle if not named.
OwnerTask	Name or handle of the task that owns the timer (the one that called <code>smx_TimerStart()</code> ).
type	Type of timer: cyclic or one-shot
state	Pulse state LO/HI.
count left	The counts are converted from differential to absolute number of ticks remaining until the timer expires and LSR is invoked.
lsr	LSR to be called when the timer counts down to zero
opt	LSR parameter option. Indicates what will be passed to the LSR: par, pulse state (LO/HI), etime at timeout, or number of timeouts since start.

Expanding a timer will display more timer-specific information.

Name	Timer name from TMCB, or handle if not named.
Forward Link	Next timer in timer queue ( <code>smx_tq</code> ) or NULL if none.
Timeouts	Number of timeouts since last start.
Diff Count	Difference count from preceding timer.
Next Delay	When it will timeout again (etime).
Period	Period (ticks) for a cyclic timer.
Width	Pulse width.
Parameter	Parameter to LSR.
Owner	Task that started it.
Timer Pulse State	Low or High pulse state.

## Pipes

Name	Pipe name from PICB, or handle if not named.
Handle	Pipe handle.
Callback	Address of callback function.

Expanding a pipe will display more pipe specific information.

Name	Pipe name from PICB, or handle if not named.
Handle	Pipe handle.
Forward Link	Control block handle of waiting task. (Start of queue.)

Backward Link	Control block handle of waiting task. (End of queue.)
Flags	Flags
Width	Pipe element width.
Length	Number of elements pipe can hold.
Buffer Start	Address of the buffer.
Buffer End	Address of the end of the buffer.
Read Pointer	Buffer read pointer.
Write Pointer	Buffer write pointer.

## Print

See section Print Window below.

## User

This button activates a window that can be used to display custom user application objects that may be helpful in a debugging session. The user or Micro Digital can write Microsoft Visual C++ code to display any user application object, structure, variable, buffer, or memory value. Contact Micro Digital for more information.

## Config Values

Shows the configuration values for the smx kernel. These are stored in the smx\_cf structure and set in acfg.h in the application.

## Handles

Name	Object name.
Handle	Object handle.
Type	Type of handle (Task, Semaphore, Ready Queue, ...).

Displays all entries of the handle table and all objects named when created (which need not be added to the handle table starting with v4.2).

## Diagnostics

Indicates the version of smx and smxAware and the processor/memory model. Displays coarse profiling information (percent idle, work, and smx overhead). Also displays a list of smx kernel errors and the last smxBase error, if any. The column Reported/Caused By indicates who encountered or caused the error. This can be a task name or strings to indicate LSR, ISR, or general smx error. Some errors are clearly caused by the indicated task/LSR/ISR, such as SMXE\_INV\_PARM or SMXE\_STK\_OVFL, but others are not. For example SMXE\_RQ\_ERROR is a general system error, and it is not known what caused it. Some errors such as SMXE\_OUT\_OF\_ and SMXE\_INSUFF\_ are encountered by a task but not necessarily caused by the task. For example, it is not the fault of the task that not enough control blocks or heap space was configured, but it is possible that the task is trying to allocate more of something than it should, so showing the task name may be a helpful clue.

**smxFS**  
**smxNS**  
**smxUSBBD**  
**smxUSBH**

See section SMX Middleware Module Displays below.

## SMX Middleware Module Displays

*This feature is currently available only for IAR EWARM (little endian) versions of smxAware.*

The middleware sections display detailed information about each installed middleware product. They only appear in the list in the window if the corresponding modules are present in your application and you have the minimum version of each indicated below. This is because changes were made to some data structures in each product, such as field ordering and size.

If smxAware is unable to read some global variables it needs, it will display a message indicating this. These are the files with symbols that are referenced:

smxFS:	fapi
smxNS:	net.c, netconf.c, support.c, tcp.c
smxUSBBD:	uddcd.c, udfunc.c
smxUSBH:	udriver.c

Below is a sample of each display.

### smxFS

*Minimum Version: SFS\_VERSION >= 0x202 in \XFS\fpport.h*

```
Disk 0
DeviceID           00000000
Status             Device Mounted
FAT Type           FAT32
Sector Size        200
Cluster Size       200
Total Sectors      3c4a1
Reserved Sectors   46
First Data Sector  7f0
Free Clusters      1da6b
Cache Sizes
...
Open File 0 sfstest.bin
Handle             20069610
DeviceID           00000000
File Size          1ec000
```

File Pointer	1ec000
File Status	READWRITE
Update Status	File Updated File Cache Empty Cache Updated
Attributes	
Buffer Pointer	20069640
Path Cluster	2
Entry Cluster	b
Offset	e
First Cluster	15
FP Cluster	f75
Open File 1	Test1.bin
...	
Open File 2	Test2.bin
...	
Disk 1	
...	

## smxNS

*Minimum Version: SNS\_VERSION >= 0x0260 in \XNS\include\smxns.h*

Net Status  
Buffer Status  
ARP Status  
Route Status

These each expand to display a table. They are the same as the diagnostics smxNS reports via Telnet, and they are documented in the smxNS User's Guide in Appendix B: Debugging Techniques.

## smxUSB

*Minimum Version: SUD\_VERSION >= 0x0231 in \XUSB\udport.h*

Device Controller Name	AT91
Registered Function driver	Serial
Device Status	Configured
Device Address	2
Device Descriptor	
bLength	12
bDescriptorType	1
bcdUSB	110
bDeviceClass	2
bDeviceSubClass	0

bDeviceProtocol	0
bMaxPacketSize0	8
idVendor	4cc
idProduct	0
bcdDevice	232
iManufacturer	1
iProduct	2
bNumConfigurations	1
Total Configuration Number	1
Active Configuration Descriptor	
bLength	9
bDescriptorType	2
wTotalLength	27
bNumInterfaces	1
bConfiguration Value	1
iConfiguration	4
bmAttributes	c0
bMaxPower	a
Alternative for interface 0 is 0	
Interface Descriptor	
bLength	9
bDescriptorType	4
bInterfaceNumber	0
bAlternateSetting	0
bInterfaceClass	2
bInterfaceSubClass	2
bInterfaceProtocol	1
iInterface	5
Endpoint Descriptor	
bLength	7
bDescriptorType	5
bEndpointAddress	2
bmAttributes	Bulk
wMaxPacketSize	40
bInterval	0
Endpoint Descriptor	
...	

## smxUSBH

Minimum Version: *SU\_VERSION* >= 0x0224 in \XUSBH\uport.h

### Host Statistics

Host name OHCI

Registered class drivers

Device Name	hub
Device Name	usb-storage
Device Name	usb-mouse
Device Name	usb-keyboard

Plugged Device: hub

Address 1

Device Descriptor:

bLength	12
bDescriptorType	1
bcdUSB	110
bDeviceClass	9
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	8
idVendor	0
idProduct	0
bcdDevice	0
iManufacturer	0
iProduct	2
iSerialNumber	1
bNumConfigurations	1

Active Configuration Descriptor

bLength	9
bDescriptorType	2
wTotalLength	19
bNumInterfaces	1
bConfigurationValue	1
iConfiguration	0
bmAttributes	40
bMaxPower	0

Interface Descriptor 0

Alternate setting	0
bLength	9
bDescriptorType	4
bAlternateSetting	0
bInterfaceClass	9
bInterfaceSubClass	0
bInterfaceProtocol	0

Endpoint Descriptor for Endpoint 0

bLength	7
bDescriptorType	5
bEndpointAddress	81
bmAttributes	3
wMaxPacketSize	2
bInterval	ff

Plugged Device: usb-storage

Address	2
---------	---

Device Descriptor:

bLength	12
bDescriptorType	1
bcdUSB	110
bDeviceClass	0
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	40
idVendor	ea0
idProduct	6828
bcdDevice	110
iManufacturer	1
iProduct	2
iSerialNumber	3
bNumConfigurations	1

Active Configuration Descriptor

bLength	9
bDescriptorType	2
wTotalLength	27
bNumInterfaces	1
bConfigurationValue	1
iConfiguration	0
bmAttributes	80
bMaxPower	32

Interface Descriptor 0

Alternate setting	0
bLength	9
bDescriptorType	4
bAlternateSetting	0
bInterfaceClass	8
bInterfaceSubClass	6
bInterfaceProtocol	50

Endpoint Descriptor for Endpoint 0

bLength	7
bDescriptorType	5
bEndpointAddress	81
bmAttributes	2
wMaxPacketSize	40

```
bInterval          0
Endpoint Descriptor for Endpoint 1
...
```

## Print Window

The print feature is like using `printf()` to send info strings to the `smxAware` print window. The user calls `sa_Print()` or `sa_PrintVals()` with a null-terminated string. During execution, the strings are written to the print buffer in the order in which they are encountered. Examples:

```
sa_Print("looping");           /* display a string */
sa_PrintVals("i = %d j = %d", i, j); /* display 2 values */
sa_PrintVals("i = %d", i, 0);    /* display 1 value */
```

Caution: These functions are **not safe from ISRs**, since they call SSRs (semaphore).

Note: You may want to use C library functions to prepare the string for `sa_Print()`. However note that `printf()` requires a lot of stack, so we do not recommend using it. Also note that you should protect any non-reentrant C library functions you use with a mutex.

To use the print window:

1. Add calls to **sa\_Print()** from points of interest in your app, such as the examples above.
2. Build the Debug version of your app.
3. Run your app in the debugger. Open the `smxAware` window and select the **Print** display to view the contents.

`sb_MsgDisplay()` calls `sa_Print()` so those messages appear in this window too. Messages pending in the OMQ or OMB are also displayed.

## Modal vs Non-Modal Dialog

The dialog for IAR is non-modal, meaning you can continue to step through the code and use the debugger while the `smxAware` dialog is open. This is convenient to allow you to watch as `smx` objects change, but it can slow down responsiveness of the debugger. The delay is most noticeable when you are stepping through the code, since data is transferred after each step. Closing the `smxAware` window when not needed will make stepping faster. This can be done automatically; in project Options check “Close window on each run”.



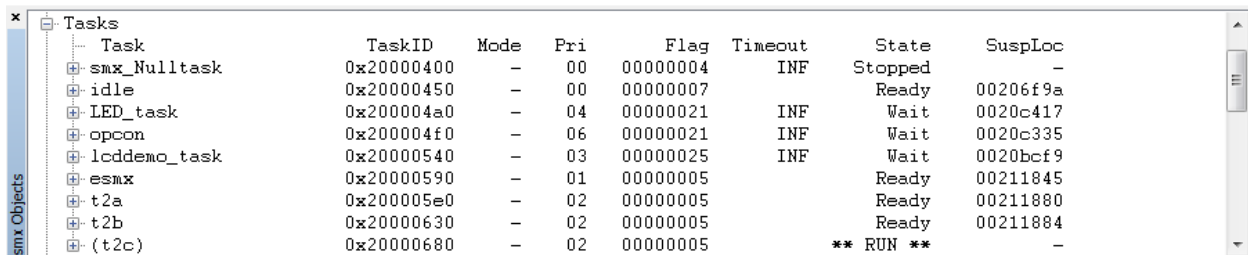
## Suspended Task Information

Some debuggers have the ability to display suspended task information, such as to show the location it was suspended in the code window, with call stack and even registers. However, this is tool-specific and can be difficult to implement, so in v4.4.0, we added a feature to smx to show where tasks were suspended. It can be easily retrofitted into existing v4.2, v4.3, and v4.4 releases. Currently it has been implemented only for ARM and ARM-M.

The **susploc** field was added to the TCB to store the location a task was suspended. It is the address of the next instruction that will execute when that task is resumed (and the one before it is the last one that ran before the suspension occurred). You can enter that address into the debugger's disassembly window to see the location, and you can set a breakpoint there to run to that point and then continue debugging that task.

smxAware displays this address in the SuspLoc column of the text Tasks display and in the Suspended Location field of an expanded Task. `SMX_CFG_SAVE_SUSPLOC` must be 1 in `xcfg.h` and the CPU architecture `.inc` file, e.g. `xarmm_iar.inc`, and `sa_susploc` must be `TRUE` in `smxaware.c`.

Note that the value in `TCB.susploc` should be ignored for stopped tasks (because they will restart from the beginning) and for `ct` (since it is currently running). smxAware displays a `-` for these cases.



Task	TaskID	Mode	Pri	Flag	Timeout	State	SuspLoc
smx_Nulltask	0x20000400	-	00	00000004	INF	Stopped	-
idle	0x20000450	-	00	00000007		Ready	00206f9a
LED_task	0x200004a0	-	04	00000021	INF	Wait	0020c417
opcon	0x200004f0	-	06	00000021	INF	Wait	0020c335
loddemo_task	0x20000540	-	03	00000025	INF	Wait	0020bcf9
esmx	0x20000590	-	01	00000005		Ready	00211845
t2a	0x200005e0	-	02	00000005		Ready	00211880
t2b	0x20000630	-	02	00000005		Ready	00211884
(t2c)	0x20000680	-	02	00000005		** RUN **	-

## Task-Specific Breakpoints

Task-specific breakpoints are breakpoints that only occur if the specified task is the active or running task. They are often used to break in a common routine that is called by multiple tasks.

### IAR

*Only available for IAR v6.10 and higher.*

To set a task-specific breakpoint:

1. Run until the tasks are created.
2. Set a breakpoint on the line of code.
3. Right click that line, and select "Edit Breakpoint". Click the Task button, and select a task from the drop-down list. Click the "Break only if selected task is active" checkbox.

The drop-down list only shows tasks that have been created at the time. If the task is not named (in the handle table) then the task handle will be displayed in the drop-down list. If the code at the breakpoint is only ever executed by one specific task, there is no need to make the breakpoint task-specific.

Stepping (using the green task-specific stepping toolbar, in some IAR versions): If more than one task can execute the same code, there is a need for both task-specific breakpoints and task-specific stepping. For example, consider some utility function, called by several different tasks. Stepping through such a function to verify its correctness can be quite confusing without task-specific stepping. Standard stepping usually works as follows (slightly simplified): When you invoke a step command, the debugger computes one or more locations where that step will end, sets corresponding temporary breakpoints, and simply starts execution. When execution hits one of the breakpoints, they are all removed, and the step is finished. During that brief (or not so brief) execution, basically anything can happen in an application with multiple tasks. In particular, a task switch may occur, and another task may hit one of the breakpoints before the original task does. It may appear that you have performed a normal step, but now you are watching another task. The other task could have called the function with another argument or be in another iteration of a loop, so the values of local variables could be totally different. Hence, there is a need for task-specific stepping. The step commands on the green stepping toolbar behave just like the normal stepping commands, but they will make sure that the step does not finish until the original task reaches the step destination (unless a different breakpoint is executed first).

Note: In the standard debugger menu, there are no Instruction Step Over and Instruction Step commands. This is because the standard Step Over and Step Into commands are context sensitive, stepping by statement and function call when a source window is active, and stepping by instruction when the Disassembly window is active. The RTOS stepping commands are unfortunately not context sensitive; you must choose which kind of step to perform.

# Graphical Analysis Tools (GAT)

smxAware with GAT includes graphical displays that are very useful. To access them select **smxAware | Graph** from the menu. There are 3 graphical displays here plus an error buffer display. These are selected by the buttons “Event”, “Profile”, “Stack”, and “Error”. These are discussed below, in turn. There are also **Event Buffer**, **Resource Usage**, and **Memory Map** displays in the smxAware menu.

*This feature is currently implemented only for IAR.*

**If these displays don’t work, check that EVB\_SIZE is non-zero in APP\acfg.h and that SMX\_CFG\_EVB is set to 1 in xsmx\xcfg.h.**

For IAR, GAT runs as a standalone executable that is launched automatically by the IDE, rather than as a window within the IDE. It is launched when you select smxAware | Graph or Event Buffer from the menu. If the GAT window does not open, see the Troubleshooting section at the end of this manual.

The standalone **smxAwareGAT.exe** can also be run from Windows to look at past traces off-line. The Event Trace Buffer that is downloaded from the target is saved in the directory indicated in Saved Traces below, or where the smxAware.ini file there specifies in its dataPath setting. (In the past, they were stored in the same directory as the smxAware DLL, but this is not allowed by Windows User Access Control.) The file name of each saved trace indicates the date and time it was saved. If the GAT window does not open, see the Troubleshooting section at the end of this manual.

## Guides

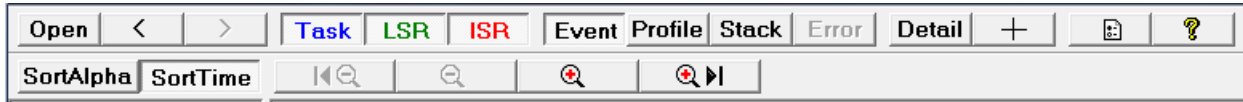
### Color Key

blue	tasks (light blue is used for tasks with no events during the sample period)
green	LSRs
red	ISRs
orange	scheduler
white	SSR calls in blue and green bars; ISR invokes in red bars
red dots	errors detected

## Toolbar

The toolbar was changed to be 2 lines. View-specific buttons were moved/added to the second line.

### Event Timelines



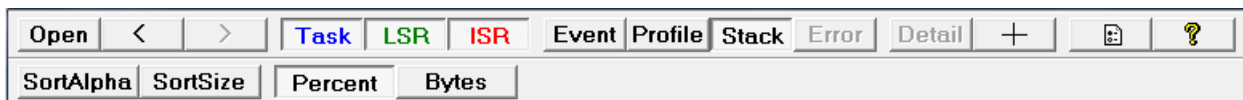
Open	Open saved trace files.
< >	Scan through saved trace files.
Task, LSR, ISR	See section Filters below.
Event	Event Timelines display. See Event Timelines below.
Profile	Profile display. See Profile below.
Stack	Stack Usage display. See Stack Usage below.
Error	Error display. See Error Buffer below.
Detail	Details window. See Details Button below.
+	Crosshairs
Options	Configuration settings. See section Options Dialog.
?	Help
SortAlpha/Time	Re-orders lines. See section Sort Buttons below.
-/+	Zoom. See section Zoom below.

### Profile



All Frames	Shows the average for all frames.
Prev Frame	Moves to the previous frame.
Next Frame	Moves to the next frame.
Percent	Shows profile information as a percent of total time.
Time	Shows actual run time.
Table	Shows data in tabular form.

### Stack



SortAlpha/Size	Re-orders lines in alphabetically or by stack size.
Percent	Shows stack usage as a percentage of each stack's size.
Bytes	Shows the number of bytes used.

## Event Timelines

This is the premier feature of GAT. This display lets you visualize system operation with bars that indicate when tasks, LSRs, and ISRs ran, and it indicates events that occurred in them, such as smx calls. This gives you a good view of system execution over a short sample period. As the system runs, smx logs entries in its Event Buffer, and smxAware displays this information with graphical bars. The display is clean and un-cluttered, making it appear deceptively simple. This section discusses its capabilities, some of which might not be immediately evident.

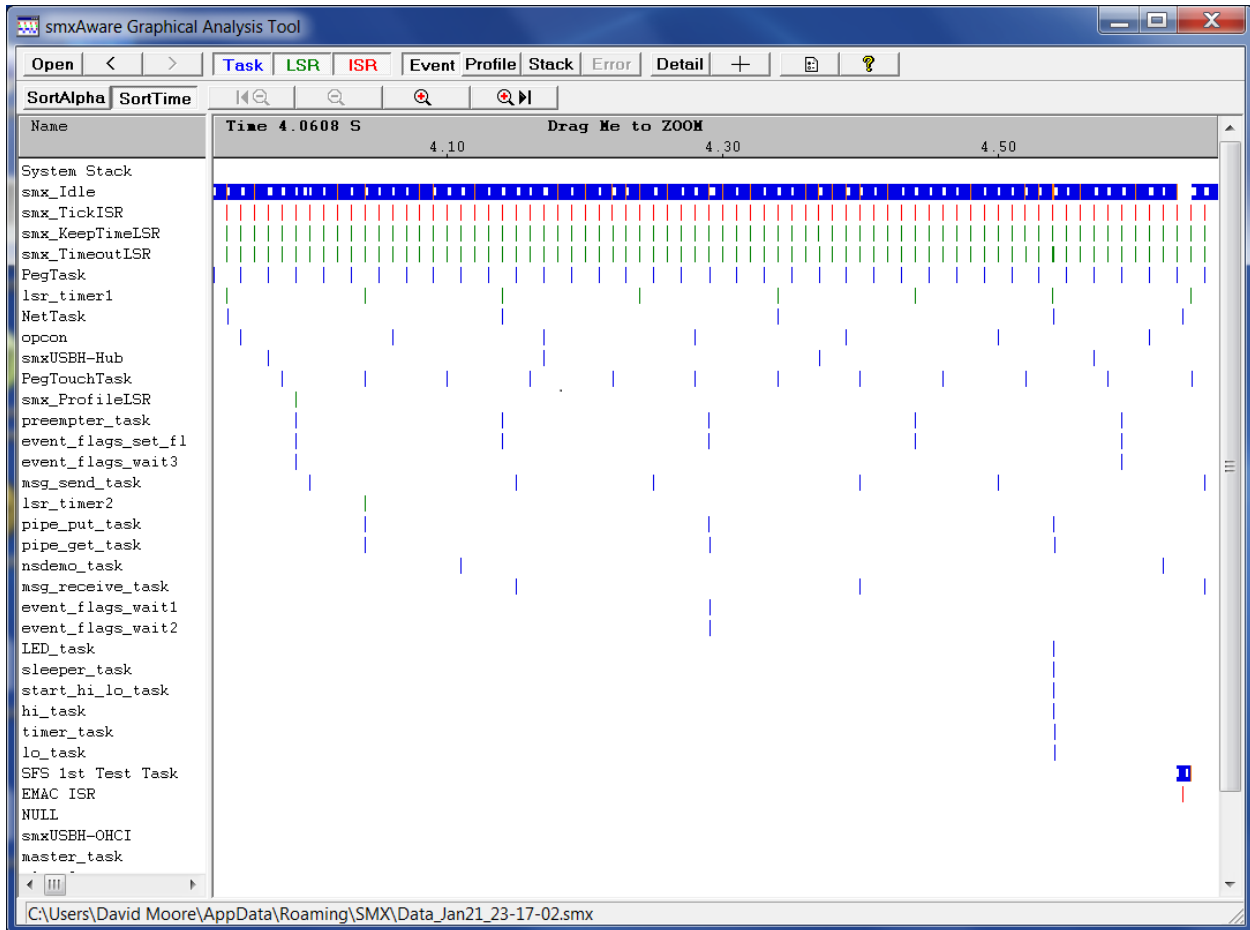
When the window is first opened it is zoomed all the way out, showing the entire trace. See section [Zoom](#) below for discussion of zooming in and out.

### Setup

To use this feature, the smx Event Buffer must be enabled by setting `SMX_CFG_EVB_SIZE` to a non-zero number of bytes in `acfg.h` and by setting `SMX_CFG_EVB` to 1 in `xcfg.h`. Ensure you have enough heap space to accommodate it.

Also, the `sb_ticktmr_` variables in the BSP must be set appropriately, to tell smxAware the characteristics of the timer used for event record timestamps. smxAware uses this information to convert the timestamp into a meaningful time (fractional seconds). See section [Event Timestamps](#). For more information about the Event Buffer, see the [smx User's Guide](#).

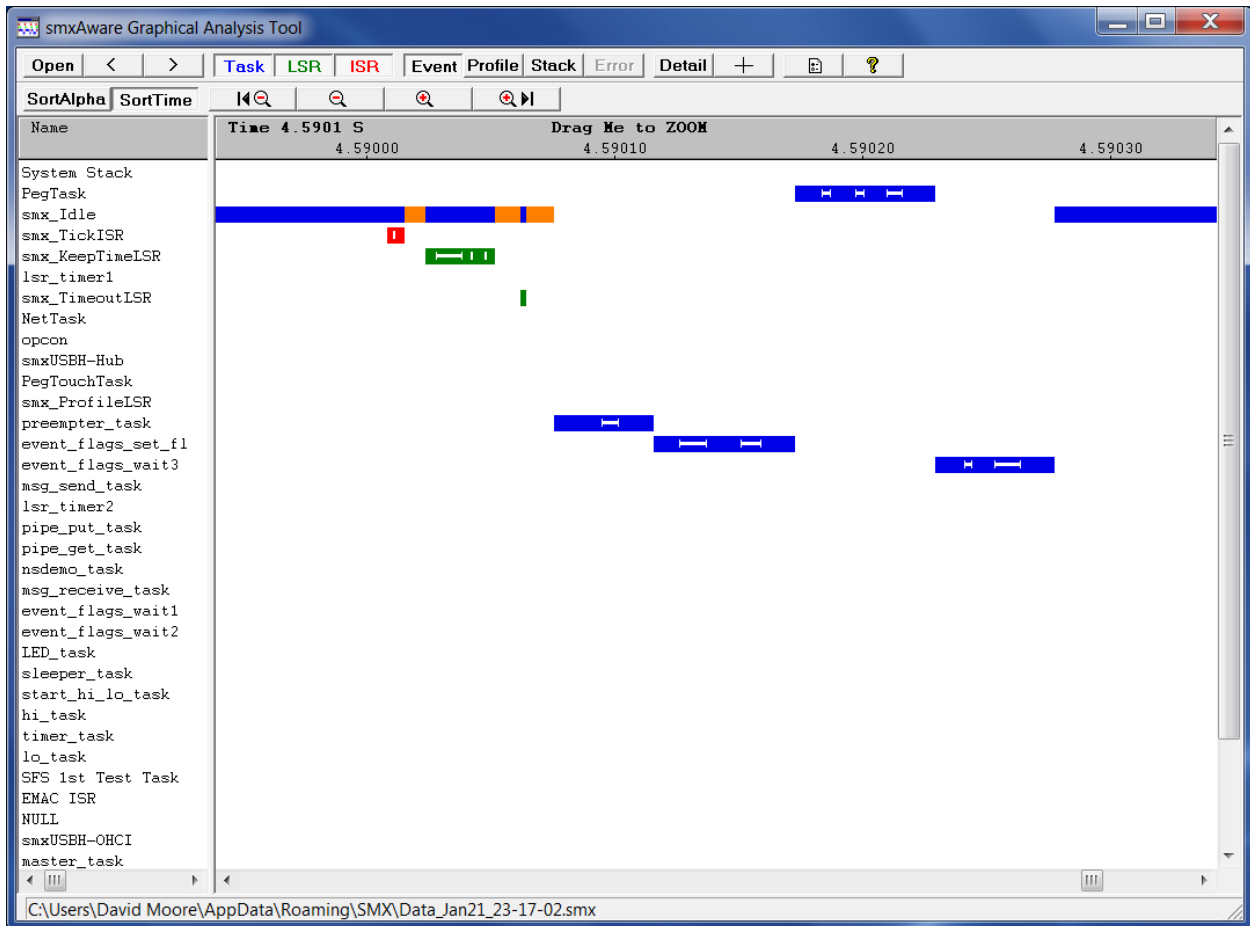
## Event Timelines



Notes about this sample:

1. The window shows the whole trace initially. It can be zoomed in very finely, as shown in the next graphic below.
2. Notice the regularity of smx\_TickISR and smx\_KeepTimeLSR. If you zoomed in, you would see that smx\_KeepTimeLSR runs right after smx\_TickISR.
3. It is also interesting that although there are many tasks in the demo doing various things, the system is mostly idle. This gives a clear picture of how little time it takes for smx operations such as sending and receiving messages.

The next sample shows the same trace zoomed fairly finely, to show about 400 usec of the trace, near the end of the trace (see scale). When zoomed, it is possible to see the system call events (white bars) within the colored bars.



## Saved Traces

The Open button allows you to open past traces. They are stored in C:\Windows\Users\\AppData\Roaming\SMX. (AppData and Application Data are hidden directories.) The date and time are encoded in the file name of each. To increase the number of saved traces, increase the setting maxFilesToSave in smxAware.ini. If you want to permanently keep the trace simply rename it to a descriptive name. Note that saved traces can be opened offline by running smxAwareGAT.exe.

## Filters

The Task, ISR, and LSR buttons control which bars are enabled in the display. The buttons are toggles that are visibly in or out. This does not affect what events are logged in the Event Buffer. To control that, see section Options Dialog.

## Reference Line

Right-click anywhere in the client area of the timelines display to set a vertical reference line. To clear it, right-click in the header or in the left pane. As you move the mouse, the time delta is displayed in the header area. This is useful for measuring times.

## Zoom

In addition to the 4 zoom buttons, the view can be zoomed by dragging the mouse in the gray header above the timelines display right or left with the left button pressed.

If a reference line is set, the line stays fixed, so you can zoom in to a specific area.

The behavior of the Zoom buttons is as you would expect, except for Zoom in Max. Pressing this button zooms in to show the 4 most recent events (those at the right edge of the window) or, if a reference line is set, it zooms in to the 4 events nearest it. You may be able to zoom in a little more since it zooms until the 4 events fill the screen, but this may not be maximum zoom.

## Panning / Scrolling

In addition to the horizontal scroll bar, the view can be panned by dragging the client area left or right with the left mouse button pressed.

## Event Lines in Bars

Some events, such as SSR calls and LSR invokes appear as white lines inside the blue, green, and red bars. You may need to zoom in a bit to see them. For example, this task line shows many events:



The vertical tick at the left of each event indicates the entry event and the tick at the right indicates the exit event. For example, they indicate the entry and exit of an SSR (smx call). The bar connecting them shows the duration of the SSR call. LSR invoke events appear as a single tick mark.

## Error Dots

Errors appear as red dots in the bars of the routines that caused them. The dots always appear the same size regardless of zoom level so that they are noticeable. Moving the mouse over a dot with the Details window open shows information about the error. Zooming in will show where the error occurred relative to other events on the bar. Here is an example that shows what the dot looks like when zoomed out.



When zoomed in, you can see where the error occurred relative to other events:





## Details Button

With the Details button pressed, mouse-over the tick marks at the ends of the white event lines to see details of the event in the small window that pops up. For an SSR, the left tick shows the parameters passed and the right tick shows the return value (which is the initial FALSE return because the call has to wait). Example:

```
Time    1.3196755
Name    PegTask
smx_EventQueueCount(
    smx_TicksEQ,
    2,
    INF);
```

Left Tick (Start of Event)

```
Time    1.3196833
Name    PegTask
smx_EventQueueCount(
    return FALSE;
```

Right Tick (End of Event)

This is what is displayed when the mouse is moved over the left and right edges of an SSR bar. It shows the parameters and return value of the call.

The ends of the colored bars are also events (the start and end of a task, LSR, or ISR). Mouse-over them for details. You can also mouse-over error dots to get details about error events.

The Details button and its associated window allow you to get detailed information about events without cluttering the display with a lot of icons.

Note: The **Name** field shows the name of the task/LSR/ISR bar. In the case of an error dot for the smx STK\_OVFL error, if it is reported by smx\_IdleTask, the dot appears in the task line for smx\_IdleTask, so that is the name that is shown in this window, not the name of the task whose stack overflowed. For this error, please press the Error button to see the error list, which shows the name of the task whose stack overflowed. Also note that if an error is reported in an ISR that has no smx\_EVB\_LOG\_ISR() and smx\_EVB\_LOG\_ISR\_RET() macros, there will be no timeline for it, so the dot will be drawn in the timeline of the task that was running when it was reported.

## Re-Ordering Event Lines

You can drag and drop lines in the left pane up or down to change the ordering. The event bars move along with the text lines. This is useful to better visualize sequences of events. Also, when you switch to the Profile and Stack Usage displays, the same ordering is used.

## Sort Buttons

These re-order the lines so that the lines are sorted alphabetically or by time. For time, the topmost line is the one with the first event, the second line has the next event, etc. If the

reference line is set (by right-clicking the mouse), it is the events after the reference line that determine the sorting. Otherwise, it is relative to the left edge of the window.

### **Overlaps**

It is correct that ISRs and LSRs overlap task bars, since they run in the context of the current task. Also, there is a period between the ISR event and LSR event when the LSR scheduler runs that shows up as a gap between the ISR and LSR events. We mark this region orange in the blue task bar as a reminder that the LSR scheduler ran during this time.

### **Duration of an Event**

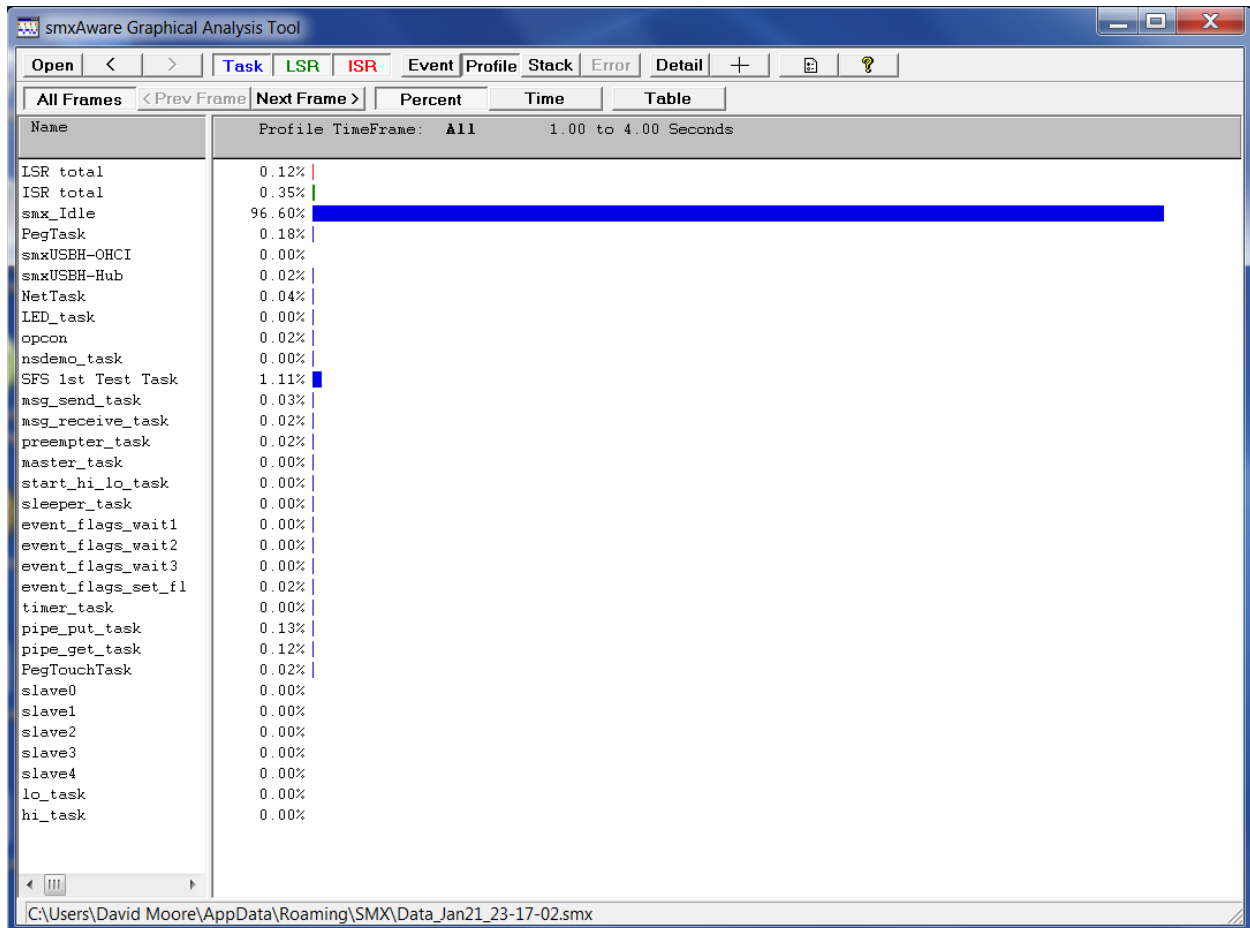
Set the reference line at the left edge of the event (right-click the mouse) and move the pointer to the right. The header shows the Delta time between the reference line and mouse pointer.

### **Guidelines**

To enable horizontal and vertical guidelines, open the Options dialog and check the Guidelines checkboxes.

## Profile

This display shows profiling information gathered by smx. It allows stepping through the profile frames. The first graph shown is the average of all frames. See the profiling sections of the Diagnostics chapter of the smx User's Guide for information about smx profiling.



Notice the All Frames button is pressed which shows the average of all samples. Using the Next Frame / Prev Frame buttons, you can step through each sample.

Overhead is shown on the first line (not pictured here), which indicates the time for scheduling and other system overhead and profiling overhead. It is calculated as the remaining time, as explained in the smx User's Guide. It may differ from the value shown for Ovh on the terminal display, because the latter is smoothed by the code in smx\_ProfileDisplay().

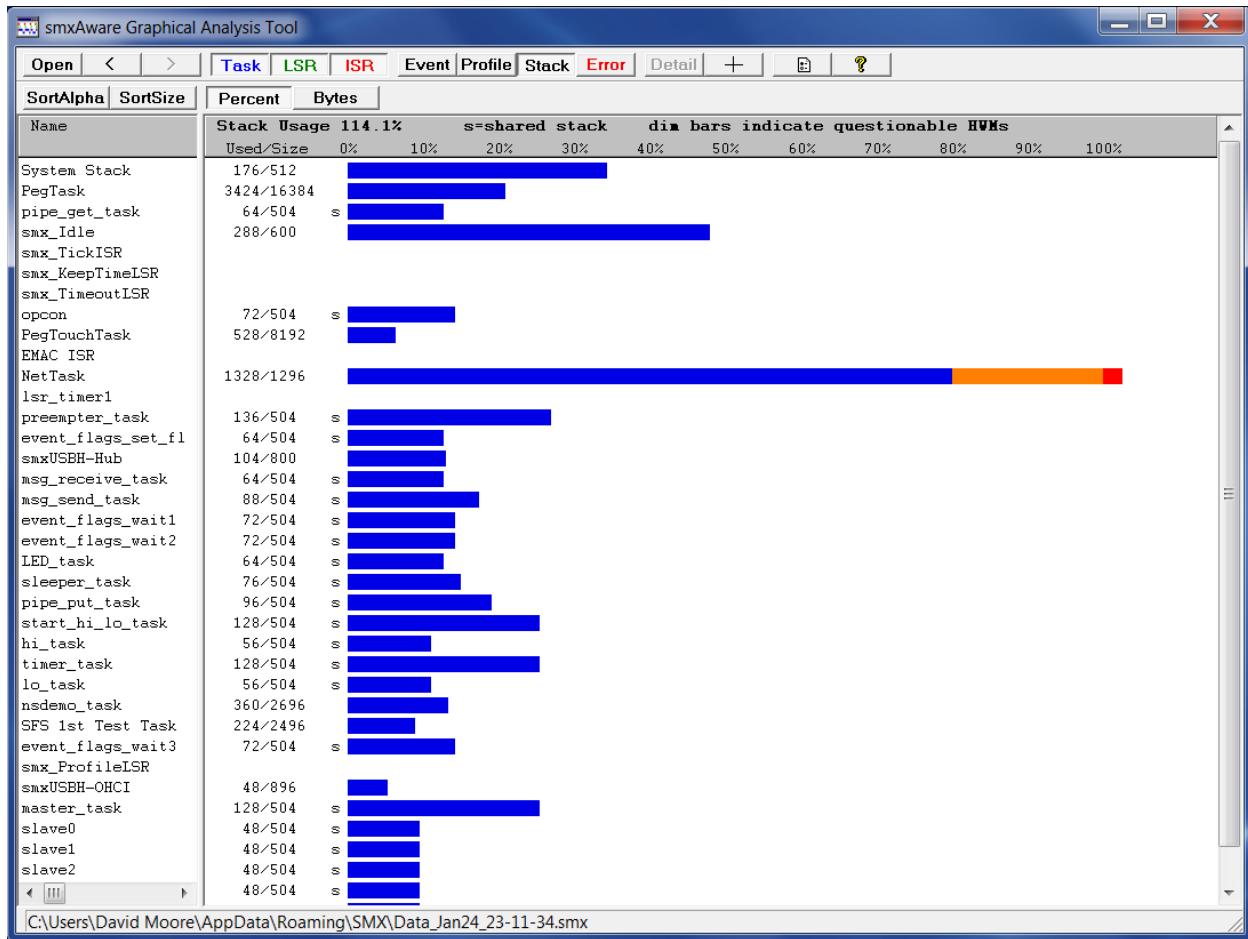
Clicking the Table button shows the data in tabular form:

Name	Profile TimeFrame: All 1.00 to 4.00 Seconds		
ISR total	0.12%	0.12%	0.12%
ISR total	0.34%	0.35%	0.36%
smx_Idle	97.16%	96.74%	95.89%
PegTask	0.18%	0.18%	0.18%
smxUSBH-OHCI	0.00%	0.00%	0.00%
smxUSBH-Hub	0.02%	0.02%	0.02%
NetTask	0.02%	0.05%	0.05%
LED_task	0.00%	0.00%	0.00%
opcon	0.02%	0.02%	0.01%
nsdemo_task	0.00%	0.00%	0.00%
SFS 1st Test Task	1.11%	1.11%	1.10%
msg_send_task	0.03%	0.03%	0.03%
msg_receive_task	0.02%	0.02%	0.02%
preempter_task	0.02%	0.02%	0.01%
master_task	0.00%	0.00%	0.01%
start_hi_lo_task	0.00%	0.01%	0.00%
sleeper_task	0.00%	0.00%	0.00%
event_flags_wait1	0.01%	0.00%	0.01%
event_flags_wait2	0.00%	0.00%	0.01%
event_flags_wait3	0.01%	0.01%	0.00%
event_flags_set_f1	0.02%	0.02%	0.02%
timer_task	0.00%	0.00%	0.00%
pipe_put_task	0.13%	0.13%	0.13%
pipe_get_task	0.12%	0.12%	0.12%
PegTouchTask	0.02%	0.02%	0.03%
slave0	0.00%	0.00%	0.00%
slave1	0.00%	0.00%	0.00%
slave2	0.00%	0.00%	0.00%
slave3	0.00%	0.00%	0.00%
slave4	0.00%	0.00%	0.00%
lo_task	0.00%	0.00%	0.00%
hi_task	0.00%	0.00%	0.00%

All three samples are shown. The number of profile samples is configured in smx.

## Stack Usage

This display shows stack usages for all task stacks as a percent of each stack's size, to help size stacks and see whether overflow has occurred in any of them. In a multitasking system, stacks account for a large portion of the system's RAM requirement, so it is very helpful to have this display to be able to tune them. Also, stack overflow is a common and difficult problem to detect without a tool such as this. Note that if stacks are put into SRAM to boost performance, fine-tuning stack sizes is even more important.



### Key:

- Blue bars indicate stacks that are ok.
- Orange bars indicate stacks that are close to overflow.
- Red bars indicate that overflow has occurred.
- Dim blue bars (not shown) mean that stack usage may not be accurate (actual usage may be higher) because the stack has not been scanned since the last time the task ran. These usually appear briefly, if at all, because after smxAware draws the graph, it scans those stacks via the debug connection and then redraws them. Most stack scanning is done by the idle task.

- White mark in bars indicates the current stack pointer. If it is far from the right end of the bar, investigate why so much more stack is needed. Possibly it is only needed during initialization, a large buffer in the stack, or something else that can be changed to reduce stack size.
- The numbers at the left of each bar indicate the number of bytes used vs. the stack size. Note that the red bars only go to about 110% regardless of how severe the overflow is, so consult these numbers to see the actual usage.
- An “s” next to a bar indicates a shared stack (one from the stack pool). Note that all shared stacks have the same size (e.g. 504 bytes, in the diagram above). Keep in mind that stack usage is independent of whether there is currently a stack assigned to the task or not. It reflects the maximum amount of stack used by the task throughout its existence. A task whose stack is marked “s” may not currently have a stack assigned to it (because it is stopped, not suspended). If a task is deleted and re-created, the usage cannot be retained because the TCB is freed and reallocated each time.

We recommend you enable stack scanning in your application, since that is the most reliable method of determining stack usage (`SMX_CFG_STACK_SCAN` in `xcfg.h` and `STACK_SCAN` in `acfg.h`). The alternative is to rely on `smx` periodically checking the value of the stack pointer, but this will likely miss times when the stack pointer is at an extreme. If scanning is off, all bars will be dim, since determining stack usage this way is unreliable. Whether the bar is dim or not depends on the state of the `stk_hwmv` flag in the TCB. This flag is set after the stack is scanned. It is cleared when the task is started or resumed, since it may use more stack as it runs. Stack scanning is done by `smx_StackScan()` (`XSMX\xsched.c`) which is called by the idle task. If many of the bars are initially dim, the system is heavily loaded and the idle task is not running very often.

Also enable stack padding (`SMX_CFG_STACK_PAD_SIZE` in `acfg.h`) so the system will continue to run after overflow (if only into the pad), and you can determine the amount of stack needed for each task after letting the system run a while. Note that stack sizes next to the bars do not include the pad size.

This display uses the `shwm` and `ssz` fields of the TCB. `shwm` is the “high-water mark,” an indication of the number of bytes of stack that have been used. `ssz` is the size of the stack (the usable area, not including any padding at the top, the Register Save Area (RSA), or loss due to alignment).

The System Stack usage is also shown. This stack is used by ISRs, LSRs, the scheduler, and error handling.

`smxAware` scans any questionable task stacks (those for tasks that have run since the last time they were scanned by `smx`) before it displays this stack information, when the Stack button is pushed in GAT or the Stack tree is expanded in the Text displays. By default this is limited to 3 seconds, in case there are many large stacks to scan. If the time limit is reached, any remaining stacks that weren’t scanned will show Usage and % with “?” following, as indicated above for these fields. The time limit can be increased to allow more stack scanning to be done if you are seeing “?” often, or it can be reduced to shorten the delay to make debugging more responsive,

since every run or step with the Stack tree expanded will be slowed down for stack scanning. (However, when stepping, it is unlikely many other tasks will run, so few if any tasks should need to be scanned on successive steps.) Edit **stackScanTimeoutSeconds** in **smxAware.ini** to change the time limit, if desired. See the Configuration section for more information.

## **Error Buffer**

For convenience, the Error button allows inspecting the error buffer from the Graphical Analysis Tool, to avoid needing to switch back to the smxAware text window to see it. It shows all errors in textual form, in the order in which they occurred. The Error button is disabled if no errors have occurred. The Reported/Caused By column indicates who encountered or caused the error. See section Diagnostics for more discussion.

## Event Buffer (text)

This shows the information contained in the smx Event Buffer, in textual form. Each line represents one event in the buffer. This is an alternate way to view the data shown by the Event Timelines bar graph. This window has the ability to filter which events are displayed and save to a file or copy to the clipboard. It allows searching for any string and stepping next or previous.

The screenshot shows the 'smxAware Event Buffer' window. It has a menu bar with 'Open', 'Save', 'Copy', 'Task', 'LSR', 'ISR', 'SSR', 'Error', 'Invok', 'User', and 'All'. Below the menu bar is a search area with 'Find:', 'Next', 'Prev', 'Match case', and 'Color' (checked). The main area contains a list of events with columns for time, name, type, and details. The events are as follows:

Time	Name	Type	Details
145.6395390	smx_Idle	SSR	return=00000001
145.6396448	smx_Idle	SSR	smx_TaskLockClear()
145.6396474	smx_Idle	SSR	smx_TaskLockClear() return=TRUE
145.6493819	smx_TickISR	ISR	enter
145.6493842	smx_Invok	Invk	802c08f8 p1=0x00000000
145.6493889	smx_TickISR	ISR	exit
145.6493969	smx_KeepTimeLSR	LSR	enter
145.6494008	802c08f8	SSR	smx_EventQueueSignal() p1=smx_TicksEQ
145.6494065	802c08f8	SSR	smx_EventQueueSignal() return=TRUE
145.6494103	smx_Invok	Invk	802c5b8c p1=0x00000002
145.6494154	smx_Invok	Invk	802c4378 p1=0x00000000
145.6494206	smx_Invok	Invk	802c0bbc p1=0x00000000
145.6494245	smx_KeepTimeLSR	LSR	exit
145.6494314	lsr_timer1	LSR	enter
145.6494390	lsr_timer1	LSR	exit
145.6494494	smx_TimeoutLSR	LSR	enter
145.6494515	smx_TimeoutLSR	LSR	exit
145.6494621	PegTask	Task	<resume>
145.6494724	PegTask	SSR	smx_SemTest() p1=PegPresentationSem p2=INF
145.6494757	PegTask	SSR	smx_SemTest() return=TRUE
145.6494857	PegTask	SSR	smx_SemSignal() p1=PegPresentationSem
145.6494892	PegTask	SSR	smx_SemSignal() return=TRUE
145.6494982	PegTask	SSR	smx_EventQueueCount() p1=smx_TicksEQ p2=2 p3=INF
145.6495066	PegTask	SSR	smx_EventQueueCount() return=FALSE
145.6495203	smx_Idle	Task	<resume>
145.6496799	smx_Idle	SSR	smx_TaskLockClear()
145.6496826	smx_Idle	SSR	smx_TaskLockClear() return=TRUE
145.6497507	smx_Idle	SSR	smx_TaskLockClear()
145.6497535	smx_Idle	SSR	smx_TaskLockClear() return=TRUE
145.6593829	smx_TickISR	ISR	enter
145.6593852	smx_Invok	Invk	802c08f8 p1=0x00000000
145.6593898	smx_TickISR	ISR	exit
145.6593979	smx_KeepTimeLSR	LSR	enter
145.6594021	802c08f8	SSR	smx_EventQueueSignal() p1=smx_TicksEQ
145.6594100	802c08f8	SSR	smx_EventQueueSignal() return=TRUE
145.6594139	smx_Invok	Invk	802c0bbc p1=0x00000000
145.6594178	smx_KeepTimeLSR	LSR	exit
145.6594239	smx_TimeoutLSR	LSR	enter
145.6594261	smx_TimeoutLSR	LSR	exit
145.6594369	msg_receive_task	Task	<resume>
145.6594463	msg_receive_task	SSR	smx_MsgReceive() p1=mailXchgA p2=INF
145.6594513	msg_receive_task	SSR	smx_MsgReceive() return=msg
145.6594604	msg_receive_task	SSR	smx_MsgSend() p1=msg p2=mailXchgB p3=NULL
145.6594681	msg_receive_task	SSR	smx_MsgSend() return=TRUE
145.6594820	msg_send_task	Task	<resume>
145.6595170	msg_send_task	SSR	smx_EventQueueCount() p1=smx_TicksEQ p2=10 p3=INF
145.6595245	msg_send_task	SSR	smx_EventQueueCount() return=FALSE
145.6595384	msg_receive_task	Task	<resume>
145.6595479	msg_receive_task	SSR	smx_EventQueueCount() p1=smx_TicksEQ p2=25 p3=INF

The status bar at the bottom shows the file path: C:\Users\David Moore\AppData\Roaming\SMX\Data\_Jan24\_20-51-27.smx

### Buttons

- Open      Open saved trace file.
- Save      Save text to a file. Saves filtered lines only (i.e. what appears in the window). To save all lines, ensure all filters are depressed (press the All button).
- Copy      Copies to clipboard. Same note as Save.

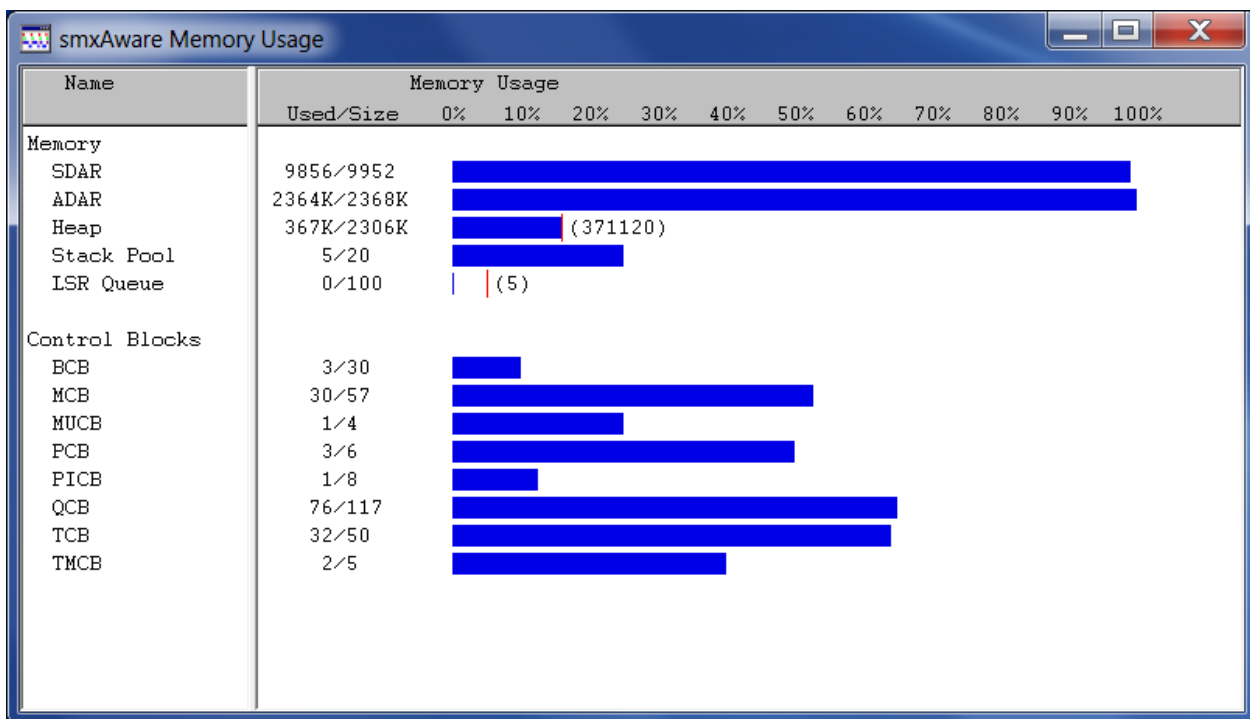


- Task, etc. Filter the display to show only selected types of events. Toggle.
- All Toggles all filters on/off.
- Next Finds the next occurrence of the string entered on the Find line.
- Prev Finds the previous occurrence of the string entered on the Find line.

You can search for any text displayed. For example, to quickly get to events at a certain time, you could search for the first digits of the time. For example, searching for 4.66 in the trace shown will find the first matching entry. Also, for convenience, the view moves as you type.

## Resource Usage

This shows a summary of memory usage by main system objects.



Notes:

- Stack Pool, LSR Queue, and Control Block sizes are indicated in number of units; others are in bytes.
- Thin red lines indicate high water marks. They indicate the maximum usage at any time during execution since startup.

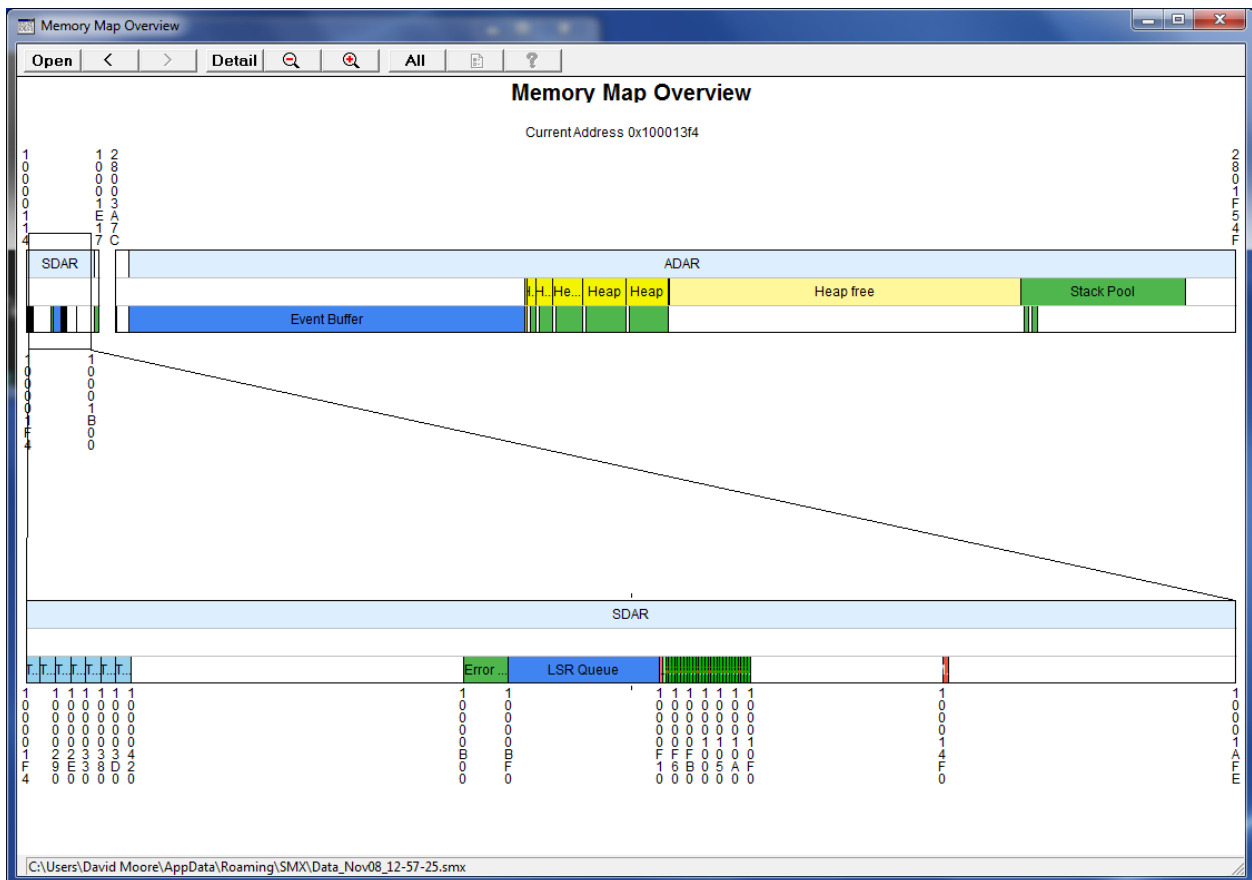
## Memory Map Overview

This shows an overview of the memory layout of the system, with the ability to zoom in for increasing detail, much like Google Earth. Areas are colored and labeled, and double-clicking one will open a hex dump of the data there. Finally, you can visualize the memory layout of your

system! Seeing the proximity of one object or region to another may give clues about the cause of a problem, especially a suspected overflow.

## Introduction

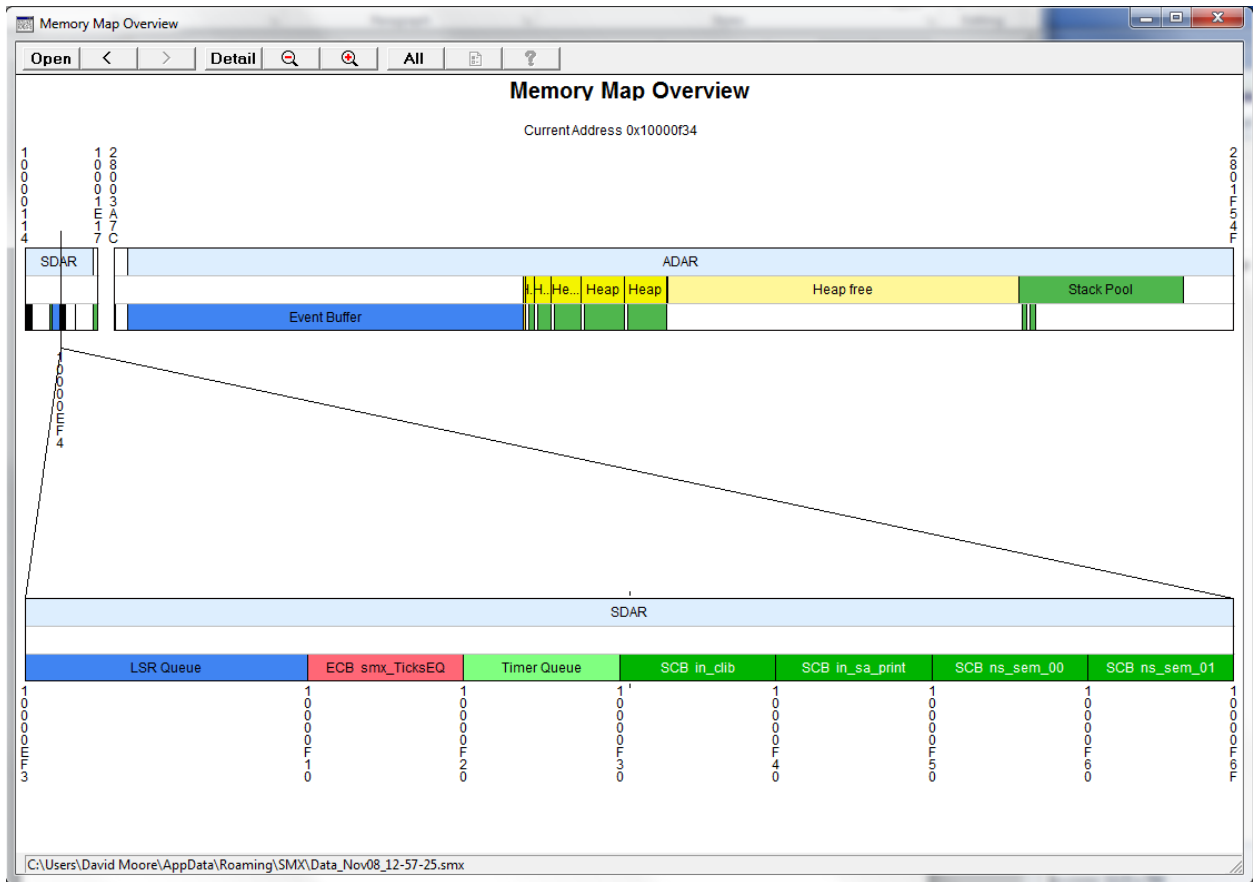
The Memory Map Overview window has 2 horizontal memory bands. The top band is static and displays only the memory in the target that can be discovered by smxAware. It will typically have one or more gaps between memory areas.



The top memory band has a Magnifier Rectangle that the user can drag, stretch, and shrink to display an area in the lower band.

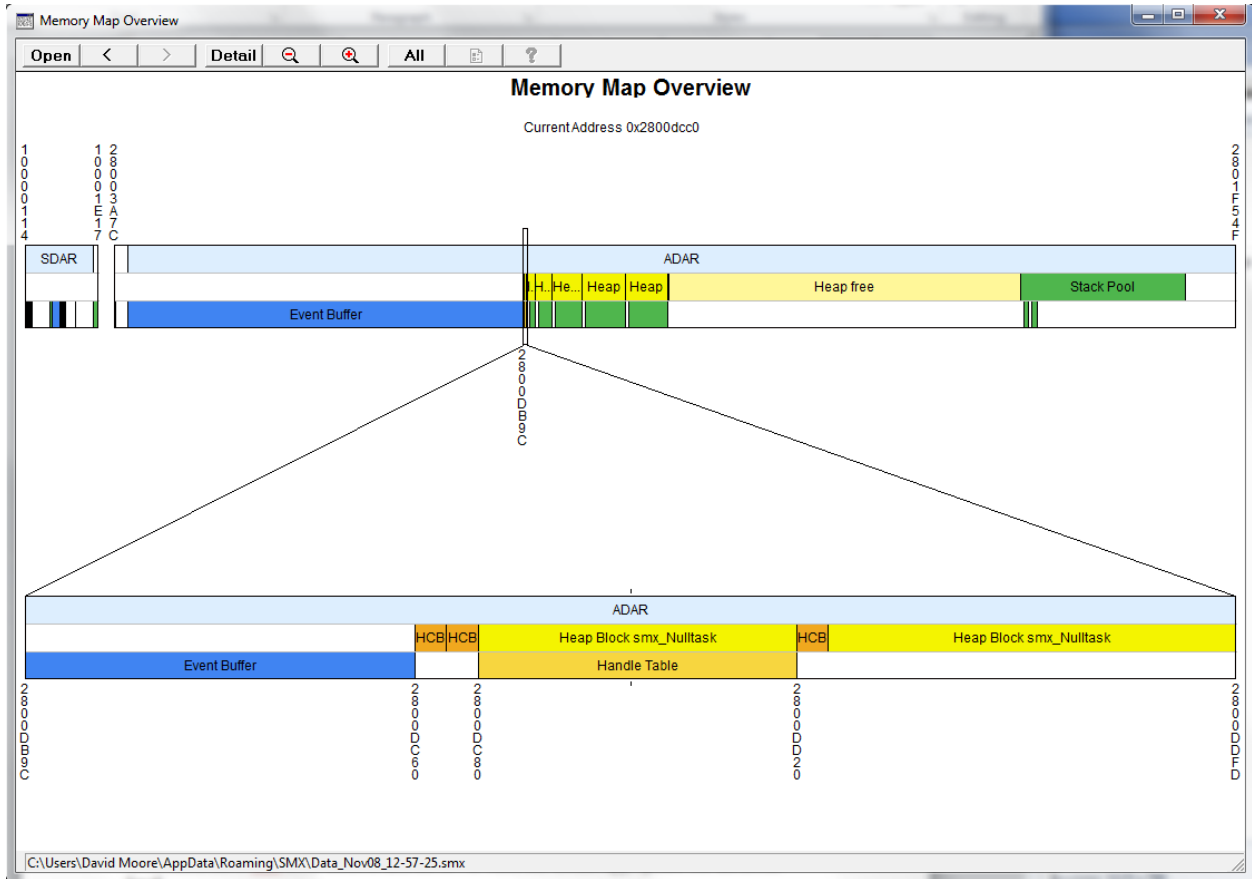
Both bands have smx areas colored and the lower band has details appropriate for the zoom level.

Both bands are sliced horizontally into three bars. The idea of them is to show containment. For example the top bar shows ADAR, and the middle bar shows Heap and Stack Pool because they are contained within ADAR (in the release this image was captured from). Similarly, the bottom bar shows stacks which are contained in the heap and stack pool. The map above was zoomed to show these details in the middle of the bottom band:



Notice the event and semaphore control blocks and other objects are named.

In the following display, the heap has been expanded to show even the CCBs which precede each block.



Double-clicking any colored region opens a data window to show its contents. See Data Window below.

### Magnifier Rectangle Navigation

Grab anywhere inside the Magnifier Rectangle and slide it to the desired location. If the Magnifier Rectangle is too narrow to grab inside, you can also grab above or below it. Watch for the cursor to change to a hand when you are in the proper location to drag.

Zoom by grabbing the left or right side of the Magnifier Rectangle. Watch for the cursor to change to a two-headed arrow.

Notice in the screenshots above, the Magnifier Rectangle in the top bar has shrunk to a narrow line due to the high level of zoom.

## Lower Band Navigation

**Pan:** Grab and drag left or right anywhere inside the lower band to scroll horizontally.

**Zoom:** Grab and drag a little above or below the lower band to zoom. Or spin the mouse wheel or use the + - buttons on the toolbar.

Set a reference line by right clicking anywhere in the upper or lower bands. The reference line will cause the zoom function to center around it, like in the Event Timelines display. Remove the reference line by right clicking outside the upper or lower bands.

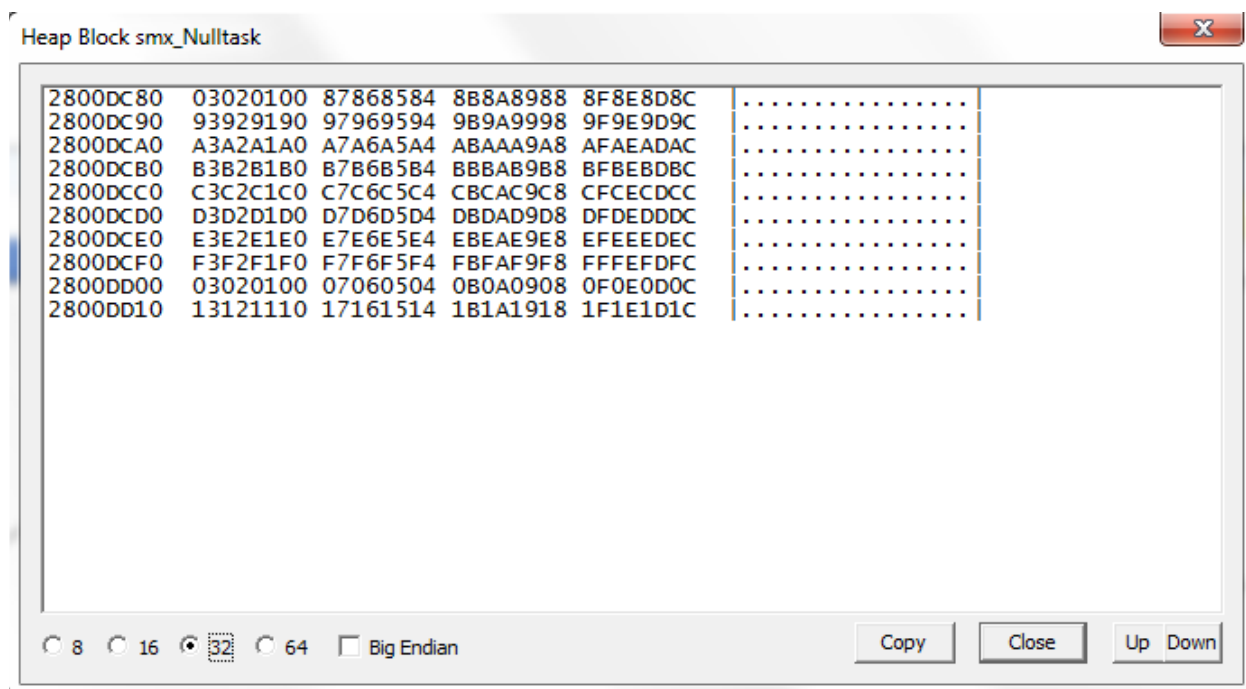
## Toolbar Buttons



Open	Open one of the data files that is automatically uploaded from the target and saved on the host PC each time the target is stopped by the debugger and smxAware is opened. The left side of the status line at the bottom left of the window has the path to the current data file.
<	Open the previous (by date) data file that was automatically saved.
>	Open the next (by date) data file that was automatically saved.
Detail	Open the detail window. This window displays details of the region that is under the cursor, such as name, start/end addresses, and size.
-	Zoom out a little with each click.
+	Zoom in a little with each click.
All	Top band displays all used and unused RAM. See section All Button below for more information.
MPU	Shows MPU regions (SecureSMX)
Options	Change display options.
?	Help.

## Data Window

Double-clicking any colored region in the upper or lower band opens a Data Window that shows a memory dump of the bytes in that region. Buttons allow selecting 8, 16, 32, and 64-bit display, as well as changing endianness. The dump shows the exact range of bytes occupied by the region (e.g. a single TCB, task stack, heap block, etc.). It can be extended with Up/Down buttons, and the original region is delimited by lines to make its boundaries clear. The following shows a small heap block. Notice the title bar indicates the name of the block, which is the same as what is shown in the colored bar that was clicked. (Data is simulated in this capture.)



Double-clicking another region opens a second window so you can compare two regions. Two windows are the maximum that can be opened, and attempting to open more will toggle between them. When you double click to open a region, only that region will be displayed, up to a maximum of 5000 bytes.

To view data before or after the selected region, click the Up or Down button to read from the target 1000 more bytes above or below the start or end of the range displayed. Each button press adds 1000 more bytes. Start and end region delimiters (lines) are placed in the data to show the boundaries of the original object that was double-clicked, as a visual reference.

Stack blocks are shown in single-column format, by default. The radio buttons at the bottom of the window can be used to change the format. Stack top and bottom are indicated with separator lines. For the current task's stack, "<-sp" marks the current top of stack (based on the CPU SP register). For a suspended task, "<-tcb.sp" marks the saved stack pointer.

## All Button

When not enabled (default), the top band only includes stacks, heaps, and smx objects. When enabled, the top band displays all used and unused RAM. All is only useful to compare the amount of space stacks and heaps use vs. the total amount of target memory. smxAware can't get an accurate accounting of the target's memory without help from the target. In XBASE\smxaware.c you will find variables such as sa\_RAM\_S and sa\_RAM\_E that contain starting and ending addresses, typically set from symbols defined in the linker command file. smxAware will read these values and use them to display different memory areas. In the linker command file for IAR, it may be necessary to make minor changes:

For the old-style .icf file that defines symbols with `__ICFEDIT__` in the name, it is necessary to add the exported keyword if not already there, as shown:

```
define exported symbol __ICFEDIT_region_RAM_start__ = 0x10000000;
```

For the new-style .icf file, it may be necessary to add symbols like this:

```
define exported symbol RAM_S = start(RAM);  
define exported symbol RAM_E = end(RAM);  
define exported symbol SRAM_S = start(SRAM);  
define exported symbol SRAM_E = end(SRAM);
```

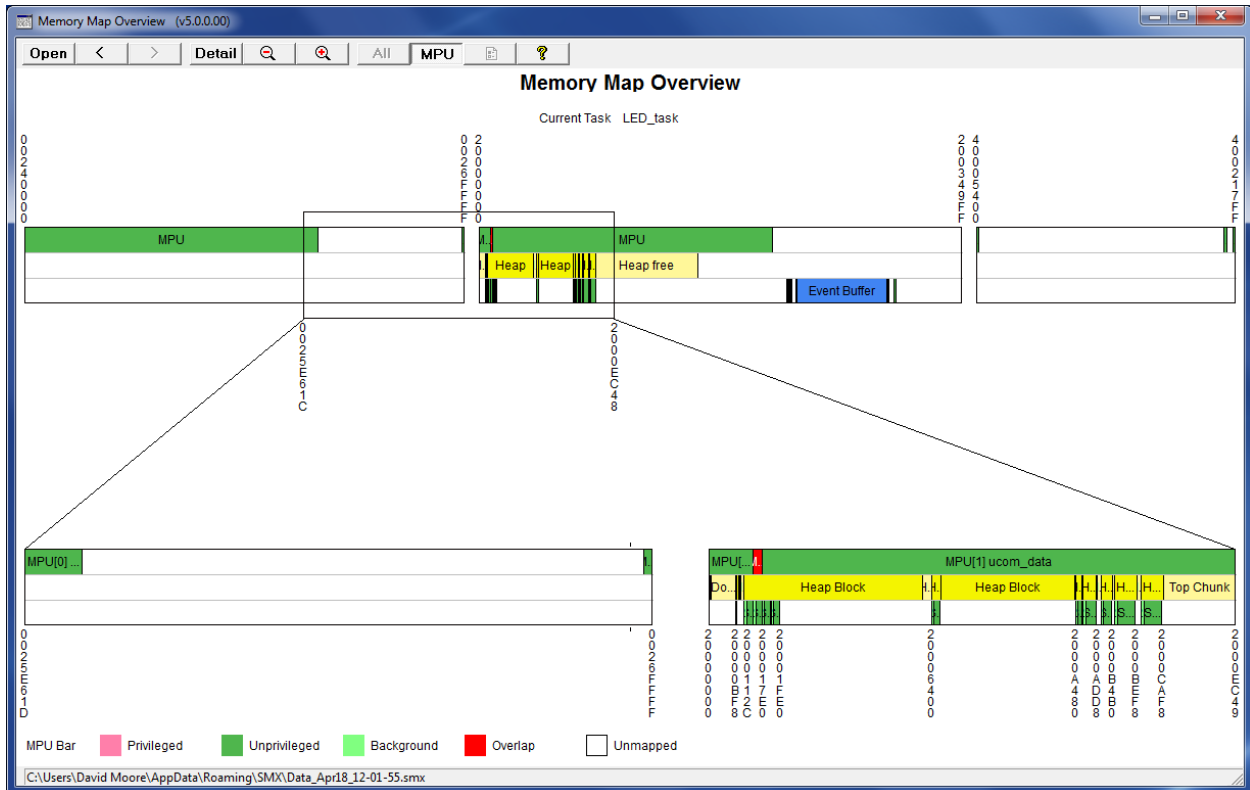
where RAM and SRAM are regions defined above in the file.

Note that the RAM symbols defined in `smxaware.c` are the superset of all RAM symbols that appear in our .icf files. They are all treated the same by `smxAware`, and the only reason they were named this way rather than `RAM0` to `RAM9` (or whatever) is to make it easier to match them up with the names in the .icf files.

The All button is a minor feature that does not provide much additional information, so you may prefer to just comment out these lines in `smxaware.c`.

## Cortex-M MPU Support

For SecureSMX, an MPU button is enabled that allows showing MPU regions in the top bar of each band. Different colors indicate regions that are accessible to the current task, system regions, and overlapping regions:



As the key in the lower left shows, privileged regions in the MPA (regardless of current mode) will show pink. This alerts you that these regions are only accessible if in privileged mode, which would be the case in an ISR that interrupts a utask that has privileged regions. Most regions will show dark green (unprivileged). Unmapped areas (white above) will show light green if you stop in an ISR that interrupted a utask, reminding you that the ISR has access to everything, not just what is mapped by the current MPU regions.

## Downloading the Event Buffer

Whenever the Graph or Event Buffer items are chosen from the menu after running or stepping through the application, the full Event Buffer must be read from the target via the debug connection. This can take awhile on a system with a slow connection. Typically, evaluation boards come with a low-cost, slower connection device. It is worth buying a JTAG unit such as IAR I-jet.



## Application Preparation

smx events are automatically logged in the Event Buffer. However, it is necessary for you to add macros to your ISRs to log them. Also, you can add user macros to your tasks to put timestamps and store data (e.g. variable values) in the Event Buffer. It is also necessary to set a few global variables to indicate to smxAware the nature of the clock used for timestamps in event records. The following sections explain what you need to do in your application.

### Pseudohandles

All of the smx\_EVB\_LOG macros require that you pass a handle to identify what is being recorded. ISRs do not have handles, so pseudohandles must be created to identify them. This is true for user events too. Also, the pseudohandles should be added to the smx Handle Table so smxAware can print the name. For example:

```
void* isr1_handle; /* defined at global scope */
...
void appl_init(void)
{
    ...
    isr1_handle = smx_SysPseudoHandleCreate();
    smx_HT_ADD(isr1_handle, "isr1");
    ...
}
```

Pseudohandles are pre-defined for smx\_TickISR and smx\_LSR\_INVOKE() events (in xglob.c and xht.c).

### Event Macros for Use in the Application

Most of the macros in xeVB.h are used internally (in the scheduler and elsewhere). Some are provided for use in your application code. This section documents the macros for your use. These macros each add an event to the Event Buffer, and it appears as a white mark within the bar of the Task, LSR, or ISR whose handle is passed.

#### **smx\_EVB\_LOG\_ISR() and smx\_EVB\_LOG\_ISR\_RET()**

Add these to ISRs that you want to log in the Event Buffer. Put smx\_EVB\_LOG\_ISR at the beginning of the ISR, right after smx\_ISR\_ENTER(), and put smx\_EVB\_LOG\_ISR\_RET at the end, right before smx\_ISR\_EXIT(). Assembly language macros are not provided, but shell functions are available in xeVB.c that can be called from assembly ISRs. If better performance is required, create assembly versions via the compiler, then optimize them, and convert them to assembly macros.

#### **smx\_EVBLogInvoke() (smx\_EVB\_LOG\_INVOKE)**

smx\_EVB\_LOG\_INVOKE() is not a user macro since it is automatically used in smx\_LSR\_INVOKE() macro and smx\_LSRInvoke() SSR. However, if you write an assembly ISR that invokes an LSR, and you want to log the invoke event, call smx\_EVBLogInvoke() (xeVB.c).

### **smx\_EVB\_LOG\_USERn()**

This macro can be used anywhere in your code to add a user record to the Event Buffer. It stores the timestamp and up to n 32-bit values that you pass as parameters. The handle par is typically a pseudo handle but could be another identifier. See the logging user events section of UG Event Logging for discussion.

### **sa\_Print()** (calls smx\_EVB\_LOG\_USER\_PRINT)

sa\_Print() is a function that prints a string to the print ring buffer and also calls the macro smx\_EVB\_LOG\_USER\_PRINT() to log this event in the Event Buffer. The macro is only for use by this function; don't use it in your code. See section Print Window for examples of using this function.

## **Event Timestamps**

sb\_PtimeGet() is called by each smx\_EVB\_LOG macro to get the timestamp for each event. It returns the counter of the timer used to generate the smx tick. See the documentation for this function and the sb\_ticktmr\_ variables in the BSP API section of the smxBase User's Guide for more information.

Using the tick timer ensures there is at least one event for every rollover of the timer. This is required for smxAware to display timelines correctly. The smx\_EVB macros used by the tick ISR and/or smx\_KeepTimeLSR() ensure there is at least one event per rollover.

## **smxAware Live**

smxAware Live is a version of smxAware for remote monitoring of the application, without a debugger. It allows viewing the GAT displays, such as timelines and event buffer. It communicates with the target via TCP/IP, using smxNS. It is designed to be minimally intrusive. When the Capture button is pressed, the application stops adding new records to the Event Buffer while a low priority task sends the data to smxAware Live. Then event logging resumes automatically.

Additional target monitoring features will be added in future releases. Note that smxAware GAT is included with smx, but smxAware Live is an extra cost option.

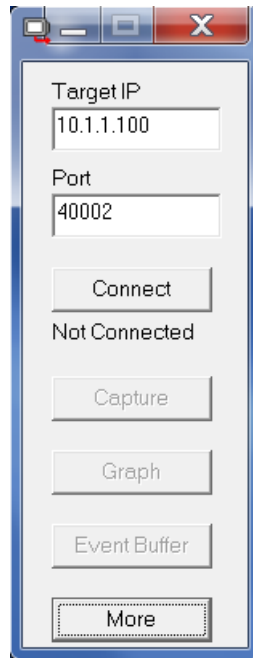
## **Installation**

No installation is needed. The smxAware Live executable can be run from the SMX\SA directory, or you may copy it to another directory. Note that there are big endian (BE) and little endian (LE) versions. Use the one that matches your target. If you run the wrong one, an error dialog will display telling you to run the other.

Enable the define `SMXAWARE_LIVE` in the prefix file in the `SMX\CFG` directory (e.g. `iararm.h`) and recompile your application. This enables sections of code in `smxaware.c` that are needed by `smxAware Live`.

## Using `smxAware Live`

When you run `smxAware Live`, its control panel displays:



Enter the IP address of your target and click `Connect`. The message below the input box will change to `Connected` if it succeeds, and the `Capture` button will un-dim. Clicking `Capture` will cause it to read the event buffer from the target and then immediately open the event timelines window. The `Graph` and `Event Buffer` buttons open the event timelines and text event windows, respectively. These look the same as `smxAware GAT`. Each time `Capture` is clicked, the windows are updated.

If you get a “Bad address” error when you try to connect, check that `SMXAWARE_LIVE` is defined in the master preinclude file (in `SMX\CFG`, e.g. `iararm.h`), that this conditional appears in `smxaware.c`, and that you rebuilt your application.

If you have other connection problems, click the `More` button to open a lower pane that shows diagnostic information. The window can be widened by dragging the corner.

The control panel has a vertical format so it uses minimal space on typical monitors, which have a wide aspect ratio. Note that it can be moved anywhere on the screen by dragging the title bar. Most of the title bar is covered with buttons, so grab it under the `Min/Max/Close` buttons.

# Diagnostics

## Text Display Error Messages

The following are the errors you may see in the text display, with more explanation about each. Only the first part of the error message is shown.

### **Apparently your processor is Big/Little Endian but you are using the Little/Big...**

You are probably using the wrong endian version of the smxAware DLL. For example, most ARM processors are little endian, some are big endian, and some allow choosing. Two versions of the smxAware DLL are provided. You must use the one that matches the endianness of your target. This is tested if sa\_ready has an invalid value. In that case, the bytes are reversed, and if the value is then valid, this message is displayed.

### **Could not read sa\_ready from target.**

Check the link map to ensure it is listed and wasn't deadstripped by the linker. Assuming it is there, something else is wrong. Maybe there is something wrong with the debugger or the connection. If you have this problem we may have to add more diagnostics to the DLL (or debug it with your app on your hardware) to help determine why it is failing.

### **sa\_ready has an invalid value.**

This global has only a few possible values. If it does not have one of those values, then it probably was corrupted. This may indicate a memory corruption in your application due to a bad pointer, for example. In any case, if it does not have a valid value, smxAware won't work. This global is initialized in smxaware\_init() in XBASE\smxaware.c. The code there makes it clear what values it can have.

### **smxAware has not been initialized.**

This error usually occurs because the target has not run long enough to initialize smxAware. The function smxaware\_init() must be called by the application before the smx objects are visible in smxAware. This function is called in smx\_Go() (or ainit() in older versions), so run past that point before opening the smxAware window.

### **smx\_Version (and probably other smx globals) could not be read from the target because the debugger could not locate them.**

Be sure the smx kernel is compiled with debug symbolics enabled for xglob.c. Also verify smx\_Version appears in the link map, to ensure it wasn't deadstripped.

## GAT Error Messages

When running the standalone smxAwareGAT.exe, if any of the events in the event buffer data file (data\_\*.smx) are corrupt then GAT will touch up the bad data point so the graph can be displayed. The following error message will appear below the window title bar.

InternalError=0x10: Timestamp of an event <= previous event.

InternalError=0x20: Timestamp of an event >= next event.

InternalError=0x40: Someone wrote into an area of the event buffer that should be zero.

## Diagnostic Logging

Additional diagnostic information can be enabled by setting “diagnostics = n” in the [CONFIGS] section of smxaware.ini, where n is one of the following:

- 1 Log information about stack scanning.
- 2 Log information about communication via debug connection, such as transfer times.

Data is written to LogFile.txt in the same directory as smxaware.ini. The data is intended for use by MDI support personnel and is not documented here.

## Tips

1. If your smx application doesn't execute properly, put a breakpoint in function smx\_EMHook() in smxmain.c (or smx\_EM() in xem.c). If smx runs out of resources or has another error, it will call this function.
2. If stepping is slow when the smxAware dialog is open, either close it or set project Options to close it automatically by checking “Close window on each run”.

## Troubleshooting

Note: The version of smxAware in your release is likely to be newer than the one included in the IAR EWARM release, so first try replacing that. The latest version is available from the Enhancements section of our support site ([www.smxrtos.com/support](http://www.smxrtos.com/support)).

**Problem:** smxAware does not load (not in IDE menu) or window does not open.

**Cause:** If you upgraded to a new version of the compiler suite and installed it to a new directory, you must copy the smxAware DLL to the new directory.

**Solution:** Copy the smxAware .dll, .exe, and related files to the new directory and restart the IDE. The installation directions at the beginning of this manual specify the directory

to copy it to. If this doesn't fix it, maybe the tools changed so that you need an updated DLL from Micro Digital.

**Problem:** smxAware window displays message that it can't read smx\_Version or another specified global variable.

**Cause:** smxAware can't determine the address of the variable. This is most likely caused by not compiling xglob.c in the smx kernel or certain files in other SMX modules with debug symbolics enabled. See section SMX Middleware Module Displays.

**Solution:** Ensure the project is set to compile xglob.c and key middleware files with debug symbolics enabled.

**Problem:** GAT window does not open and instead a file open dialog appears.

**Cause:** This is likely caused by Windows User Access Control (UAC). It should only be an issue for older smxAware DLLs, since v4.1.0 was changed to save the .smx files under the Documents and Settings or Users directory, as newer versions of Windows require. Older versions of smxAware stored the trace files in the EWARM Plugins dir, under Program Files, but only Administrators are permitted to write files there. If you are using smxAware pre-v4.1.0, change EWARM to run as Administrator. Also, it is probably necessary to take ownership of the plugins directory the DLL is in.

**Solution:** Right click on the EWARM icon or entry in the Start menu, and select Run as..., select Administrator. In Windows Explorer, right click on the plugins directory where the DLL resides and select Take Ownership.

**Problem:** SMXE\_INV\_SCB errors caused by sa\_Print() calls.

**Cause:** Semaphores used by smxAware tracing were not initialized. If SMXE\_OUT\_OF\_QCBS is reported, then maybe the semaphores couldn't be created. Otherwise, maybe smxaware\_init() hasn't been called.

**Solution:** If SMXE\_OUT\_OF\_SCBS was reported, increase SMX\_CFG\_NUM\_SEMS in APP\acfg.h. Ensure smxaware\_init() is being called from smx\_Go().