



smxAware™ User's Guide

Version 4.4.0
March 6, 2018

by Marty Cochran and David Moore

Introduction	1
Supported Debuggers.....	1
Installation	2
Changes to the Application.....	2
CodeWarrior for ColdFire Directions.....	3
CodeWarrior for PowerPC Directions.....	4
IAR EWARM Directions	4
SingleStep Directions	5
Using smxAware	7
smxAware Dialog Box	7
Kernel Displays	10
SMX Middleware Module Displays	18
Print Window.....	23
Modal vs Non-Modal Dialog.....	23
Suspended Task Information	24
Task-Specific Breakpoints.....	26
Configuration.....	27
smxAware GAT (Graphical Analysis Tools).....	28
Guides	28
Event Timelines	30
Profile	36
Stack Usage	38
Error Buffer	39
Event Buffer (text).....	40
Memory Usage.....	41
Memory Map Overview.....	42
Configuration.....	48
Downloading the Event Buffer.....	49
Application Preparation	49
smxAware Live	51
Installation	52
Using smxAware Live	52

Diagnostics	53
Text Display Error Messages.....	53
GAT Error Messages	54
Diagnostic Logging	54
Limitations	54
Tips	54
Troubleshooting.....	55

© Copyright 1996-2018

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(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

smxAware is a DLL that adds functionality to several popular 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. with the smxAware dialog box, accessible from the main menu.
- Display graphs of Event Timelines, Profiling, Stack Usage, Memory Usage, and Memory Map Overview using the Graphical Analysis Tool (GAT) feature. (CodeWarrior and IAR only).
- 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 (IAR and SingleStep only).
- Display information about suspended tasks: point where it will resume, call stack, and registers. A window allows selecting which task to view. (CodeWarrior only)

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

Supported Debuggers

<u>Debugger</u>	<u>CPU</u>	<u>Min Ver</u>
CodeWarrior [®] Professional	CF	4.1
CodeWarrior [®] Professional	PPC	5.0
IAR EWARM/C-SPY [®]	ARM	4.30A
SingleStep [®]	PPC	7.01

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. The Min Ver column above indicates the first version of the tools for which we supported smxAware; the current smxAware may not support such an old tool version.

Installation

Changes to the Application

First enable smxAware initialization in your application by enabling the define of **SMXWARE**, as instructed below, so smxaware_init() is called. This initialization routine 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.

Until smx initialization is done, smxAware will display a message in all views indicating it has not yet been initialized. To enable smxAware initialization:

1. Those using **CodeWarrior**: Uncomment **#define SMXWARE** in:
CFG\cwf.h (ColdFire)
CFG\ConfPpCw.h (PowerPC)

Those using **IAR**: Uncomment **#define SMXWARE** in:
CFG\iararm.h (ARM)

Those using **Diab** or **MetaWare** for PowerPC: Uncomment “**saware =**” in
CFG\ConfPpc.mki

Others: Uncomment the line “**saware =**” in your PRO.MAK.

2. Ensure **smxaware.c** (in the APP directory) is being compiled and linked into your application.

Also, it is necessary to compile the smx library with debug symbolics enabled for xglob.c. This should already be done in pre-built libraries and in the project file or make file we supply, but check this 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.

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, LSRs, messages, and others that cannot be named in a Create call, create a pseudohandle. See the section Pseudohandles later in this manual. Also note that handles should be defined as global variables.

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

CodeWarrior for ColdFire Directions

CodeWarrior Professional Edition is required to use smxAware (or any add-on DLL).

1. Put **CFrtos_smxCwCf.dll** in this dir:
Freescale\CodeWarrior for ColdFire V7.1\Bin\Plugins\Debugger\RTOS\
(if the RTOS directory does not exist then create it).
2. In project settings, select (in left pane): Debugger | CF Debugger Settings. In the right pane, select **Target OS = smxCwCf**.
If smxCwCf is not listed in the Target OS list, enable plugin diagnostics and see what is wrong: Edit | Preferences | General | Plugin Settings | Plugin Diagnostics Level: All Info. Note that the “CFrtos” prefix in the DLL name is required; do not change or delete it.
3. Press the Debug button to download the app. “smxAware” should appear to the right of Data on the main menu if it is working.
4. **Run at least until after the call to smxaware_init() in smx_Go() before attempting to use smxAware.**

Notes:

1. **Run to Cursor** (starting with CWCF v7): This now uses a task-specific breakpoint, which can cause unexpected behavior. You may be best to stop using Run to Cursor, or only use it to run to another point in the same function.
 - a. A task-specific breakpoint is one that only stops when hit in the context of the specified task. The debugger temporarily stops execution when it is hit, checks the task ID, and then continues execution if it is not the desired task ID.
 - b. The task ID is set to the ID of the thread window that you do Run to Cursor in. If done from the Symbolics window, the ID is set to 0, which never matches. Since CodeWarrior keeps opening new thread windows each time you stop in a new task, it is a very easy mistake to scroll down to another task’s code and run to cursor.
 - c. These breakpoints are listed in the Special section of the Breakpoints window. You can disable them by right clicking on one and selecting Disable Breakpoint. You can delete one by setting a normal breakpoint on the same line (by clicking in the left margin of the code window as usual, twice to appear and again to delete).
 - d. If the code is repeatedly hit, the debugger will keep pausing and continuing, and the stop button will enable and disable frequently, so that it won’t work. In this case, you can stop execution by disabling the breakpoint in the Breakpoints window.

- e. Before starting a debug session, you can disable smxAware to avoid this problem. (In project options, select CF Debugger Settings in the left pane and select BareBoard for Target OS in the right pane.)

CodeWarrior for PowerPC Directions

CodeWarrior Professional Edition is required to use smxAware (or any add-on DLL).

1. Put **EPPCrtos_Stub.dll** in this dir:
Metrowerks\CodeWarrior5\Bin\Plugins\Debugger\RTOS\
(if the RTOS directory does not exist then create it).
2. Set Edit | Proto Settings | Debugger | EPPC Target Settings | **Target OS = Stub**
3. Press the Debug button to download the app. “smxAware” should appear to the right of Data on the main menu if it is working.
4. **Run at least until after the call to smxaware_init() in smx_Go() before attempting to use smxAware.**

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.**

SingleStep Directions

Installation

1. Copy **smxsstep.dll** to directory `\sds70\cmd`

Enabling smxAware (SingleStep)

To enable this interface, select the SingleStep icon and open the Properties sheet. Open the Shortcut tab and specify the -k option as part of the command line in the Target field. For example:

```
C:\sds70\cmd\simppc.exe -k smx
```

(assuming SingleStep is on drive C:) and change Start In field to

```
C:\sds70\cmd
```

Verify proper installation by launching SingleStep. If menu item: Data | Kernel Objects brings up smx Kernel Objects window then installation was successful. If the Kernel Objects item is grayed out then SingleStep couldn't find smxsstep.dll.

Running smxAware (SingleStep)

1. Build the Debug version of the app.
2. Start SingleStep.
3. Open the debug window. Select File | Debug. Then, click on File and type in "C:\Smx\App\Diab.ppc\lv4\app.x". Then, click on Processor and select By Object

Type.

4. Then, click on Options and select Execute Until Main. Click on OK to load the debug session.
5. Select Run | ExceptionSimulation. Click on decrementer, fixed interval timer (FIT) and periodic interval timer (PIT). Click on OK to select these three interrupts required by app.x.
6. **Run at least until after the call to smxaware_init() in smx_Go() before attempting to use smxAware.**
7. Display the smxAware window: Select Data | KernelObjects. This dialog box displays most smx objects.

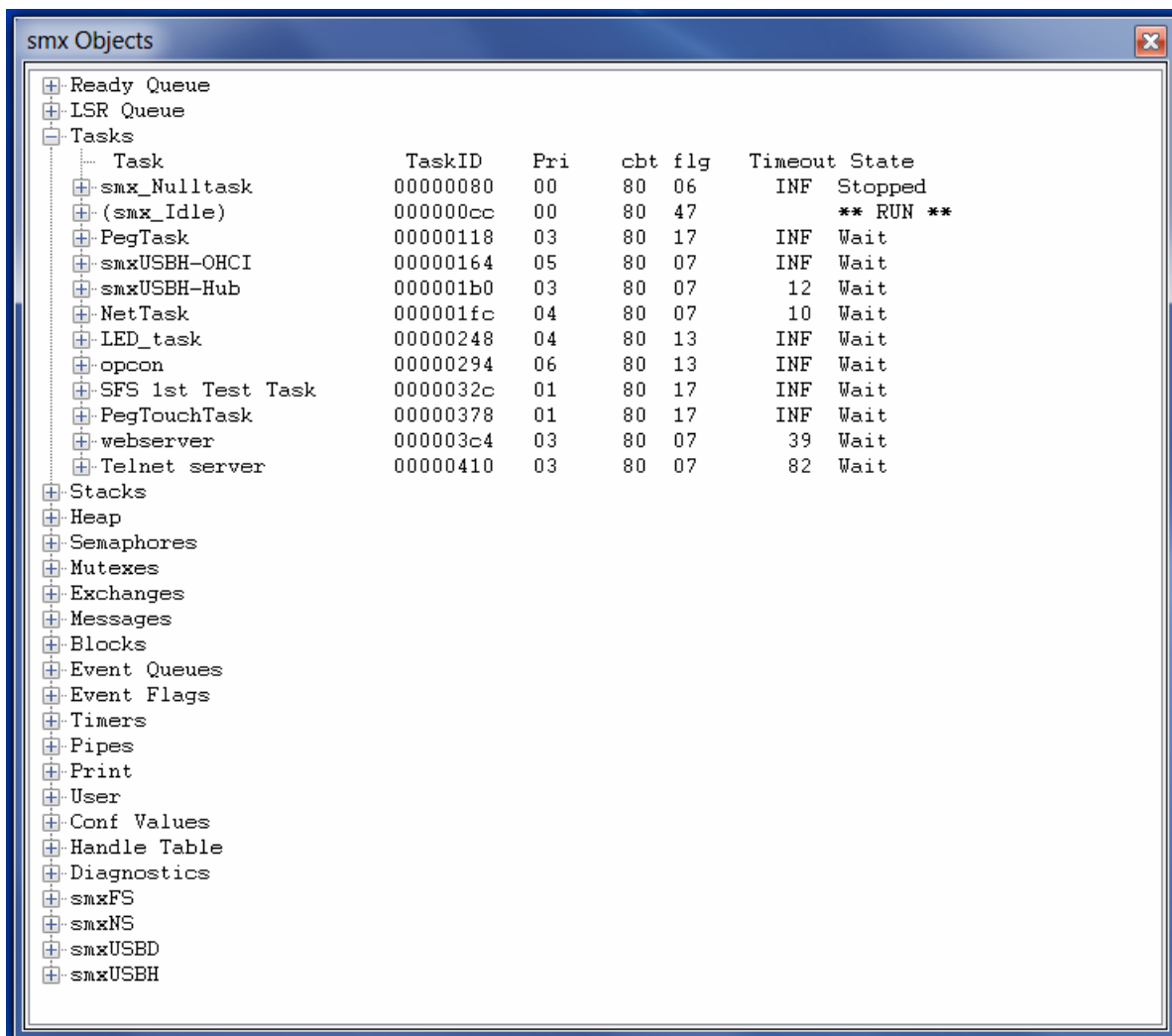
Using smxAware

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.

Copy to Clipboard: The CodeWarrior and IAR versions of smxAware have a copy to clipboard feature. Pressing “c” will copy all of the text in the current smxAware window to the clipboard. CodeWarrior copies the selected pane, so if you want to copy the right pane, click on it first. (“Ctrl-c” can also be used for IAR.)

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:

The screenshot shows a window titled "smx Objects" with a tree view on the left and a list of task details on the right. The "LED_task" is selected and expanded, showing its properties and flags.

Object Name	Address	Priority	PC	SP	Stack	State
smxUSBH-Hub	000001b0	03	80	07	12	Wait
NetTask	000001fc	04	80	07	10	Wait
LED_task	00000248	04	80	13	INF	Wait
opcon	00000294	06	80	13	INF	Wait

Property	Value
Task Name	LED_task
Queued In	00000b60 smx_TicksEQ
Forward Link	00000b60 smx_TicksEQ
Backward Link	00000294 opcon
cbtype	80
state	Wait
flags (see below)	00000013
Priority	04
Normal Priority	04
Error Number	00 No Error
Return Value	0
Suspend Value	f
Run Time Counter	00000635
Register Save Area	80244ea8
Stack Pointer	802451d8
this Pointer	00000000
Main Function	802bebac
Hooked Entry	00000000
Hooked Exit	00000000
Stack Top Pointer	80244ff8
Stack Bottom Pointer	802451f0
Stack Size	000001f8
Stack HWM	00000040 (valid)
Owned Mutex	none

flags:

Flag	Value
Event Queue	True
Start Locked	False
Stack Check	True
Perm Stack	False
Hooked Entry/Exit	False
Mutex Wait	False
Stack Overflow	False
Stack HWM Valid	True

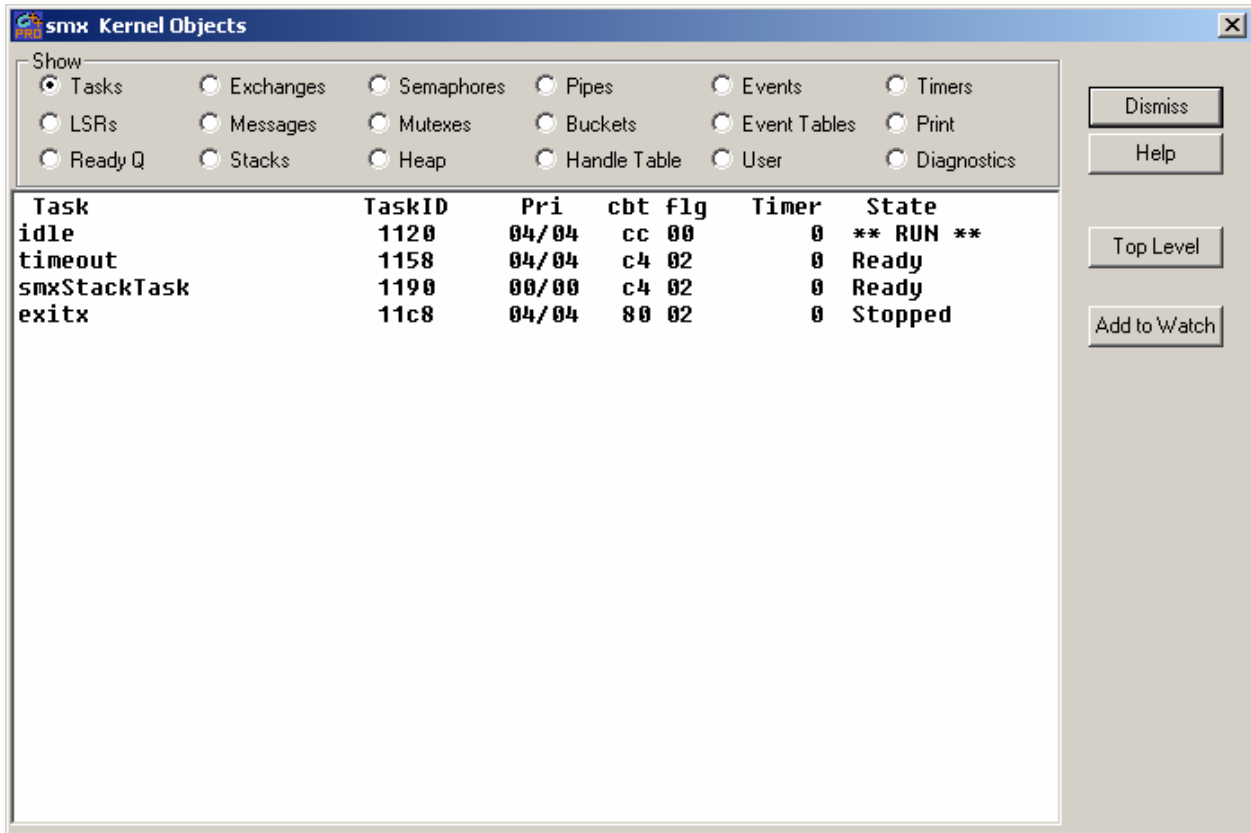
Other displays are similar.

For **CodeWarrior**, the dialog has 2 panes. The left pane lists the smx object types and the right pane shows the list of the object type selected in the left pane. The dialog is non-modal, meaning it can remain open as you step through the code. This allows you to watch things change. (However, there is a problem with some versions of the IDE, so we automatically close the window each time you step or run.) When the smxAware window is open, additional thread windows open when execution stops in a new task. See the Suspended Task Information section, below, which discusses the additional information CodeWarrior displays about suspended tasks.

The screenshot shows the smxAware window with a tree view on the left and a table of tasks on the right. The tree view includes categories like Tasks, LSRs, Exchanges, Messages, Semaphores, Mutexes, Event, Event Table, Timers, Pipes, Buckets, Print, User, Heap, Stacks, Handle Table, Ready Queue, and Diagnostics. The table on the right lists various tasks with their IDs, priorities, flags, timers, and states.

Task	TaskID	Pri	cbt	flg	Timer	State
idle	00268eb0	00/00	8c	00	0	** RUN **
timeout	00268efc	04/04	8c	03	0	WaitInf
smxStackTask	00268f48	00/00	8c	03	0	Ready
exitx	00268f94	04/04	80	02	0	Stopped
opcon	00268fe0	04/04	98	03	0	WaitInf
msg_send	0026902c	03/03	88	03	0	WaitInf
msg_receive	00269078	02/02	a8	03	0	WaitInf
preempter_task	002690c4	03/03	a8	03	0	WaitInf
master_task	00269110	02/02	88	03	b7	Sleep
start_hi_lo	0026915c	02/02	88	03	53	Sleep
sleeper_task	002691a8	03/03	88	03	53	Sleep
event_tbl_wait1	002691f4	02/02	88	03	0	WaitInf
event_tbl_wait2	00269240	02/02	88	03	0	WaitInf
event_tbl_wait3	0026928c	02/02	88	03	0	WaitInf
event_tbl_set_flag	002692d8	03/03	a8	03	0	WaitInf
timer_task	00269324	02/02	88	03	53	Sleep
pipe_put_task	00269370	02/02	a8	03	0	WaitInf
pipe_get_task	002693bc	01/01	88	03	0	WaitInf
LED_task	00269408	03/03	a8	03	0	WaitInf
display_task	00269454	03/03	a8	03	0	WaitInf
profile_task	002694a0	04/04	98	02	0	Stopped
slave0	002694ec	02/02	98	02	0	Stopped
slave1	00269538	02/02	98	02	0	Stopped
slave2	00269584	02/02	98	02	0	Stopped
slave3	002695d0	02/02	98	02	0	Stopped
slave4	0026961c	02/02	98	02	0	Stopped
lo_task	00269668	01/01	8c	02	0	Stopped
hi_task	002696b4	03/03	8c	02	0	Stopped

For other debuggers, the dialog has radio buttons at the top to select the smx object types and information for the selected object type is displayed in the white area below. Double-click on a line for details of that item. The dialog is modal meaning it must be dismissed before you can continue stepping through the code or otherwise using the debugger. The Add to Watch button adds the selected smx object to the debugger's watch window (if supported for the debugger).



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. -1 hides the item from the display. 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.

Tasks

Task	Task name obtained from the handle table, or handle if not found. () around the task name indicates the current task.
TaskID	Task handle.
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.
cbt	Control block type (cbtype).
flg	Task flags.
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 Using smxAware/ Suspended Task Information for details.

Double clicking on a task will display more task-specific information.

Task Name	Task name obtained from the handle table.
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
cbtype	Control block type
state	Task state
flags	Task flags
Priority	Current priority of task (0 to 127 and \geq Normal Priority)
Normal Priority	Normal priority of task (0 to 127)
Error Number	smx error number of last error caused by task. 00 No Error if none.
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.
exret	Low byte of exception return value, which indicates type of stack frame for FPU register autosave (ARM-M).
Run Time Counter	Counter for task profiling
Register Save Area	Pointer to part of stack where registers are saved (task context)
Stack Pointer	Stores stack pointer when task suspended

this Pointer	this pointer, for C++ tasks
Main Function	Address of task's main function (entry point)
Hooked Entry	Address of hooked entry routine
Hooked Exit	Address of hooked exit routine
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)
Stack Size	Usable bytes of stack (not including pad)
Stack HWM	Stack high water mark. Indicates stack usage, in bytes. Directly compares to stack size.
Owned Mutex	Mutex name or handle. One line for each.
Suspended Location	Address where task was suspended. See section Using smxAware/Suspended Task Information for details.

flags breakdown:

Event Queue	True/False	Task in event queue
Start Locked	True/False	Task starts locked
Stack Check	True/False	Stack checking enabled for task
Perm Stack	True/False	Task has permanently bound stack (not stack pool stack)
Hooked Entry/Exit	True/False	Routines hooked to save additional task state
In Priority Queue	True/False	Task is in a priority queue
Mutex Wait	True/False	Task is waiting to get a mutex.
AND/OR EG	True/False	Task is waiting on AND/OR of flags in Event Group
AND EG	True/False	Task is waiting on AND of flags in Event Group
Stack Overflow	True/False	Stack overflow detected, if true
Stack HWM Valid	True/False	Indicates Stack HWM (above) is valid; the stack has been scanned since the last time the task ran.

MPU regions are shown for the task for Cortex-M targets if MPU-Plus is present, and the task has set an MPA.

MPU (Cortex-M Targets if MPU-Plus is Present)

Details of the MPU are shown including:

Flags	Flags indicating whether MPU is on, background region is enabled, etc.
Caution	Warnings about overlaps
MPU[n]	Information about each slot, including start and end address, size, attributes, RBAR, and RASR.

Stacks (Task)

Task	Owner. Task name obtained from the handle table, or handle if not found.
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.

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).

Double clicking on a stack entry does nothing.

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.

Heap

The heap window shows the following main items:

Summary	Various statistics of heap usage and settings.
Allocated	List of allocated chunks (see below).
Bin nn	List of 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
BlockAddr	Block starting address. Address returned to the user where data will start.
Size	Block size of data part, excluding CCB and fences, 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 obtained from the handle table, or handle if not found.
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.

Double clicking on a semaphore will display the forward and backward links fields listed above.

Mutexes

Name	Mutex name obtained from the handle table, or handle if not found.
Owner Task	Owner task's name obtained from the handle table, or handle if not found.
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 \geq 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 $\neq 0$).
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.

Double clicking on a mutex will display all tasks waiting to get it, in priority order.

Exchanges

Name	Exchange name obtained from the handle table, or handle if not found.
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.

Double clicking on 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 address 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.

Double clicking on 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
Size	Message size (decimal, exact)
Reply Index	Index of the handle of the object to reply to among QCBs (typically an exchange handle)

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 address 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 obtained from the handle table, or handle if not found.
Handle	Event handle.

Double clicking on 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 obtained from the handle table, or handle if not found.
Handle	Event group handle.
flags	Hex image of flags set with <code>smx_EventFlagsSet()</code> .

Double clicking on 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 obtained from the handle table, or handle if not found.
OwnerTask	Name or handle of the task that owns the timer (the one that called smx_TimerStart()).
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	Hex address of the 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.

Double clicking a timer will display more timer-specific information.

Name	Timer name obtained from the handle table, or handle if not found.
Forward Link	Next timer in timer queue (smx_tq) 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.

Pipes

Name	Pipe name obtained from the handle table, or handle if not found.
Handle	Pipe handle.

Double clicking on a pipe will display more pipe specific information.

Name	Pipe name obtained from the handle table, or handle if not found.
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.)
Width	Pipe element width.
Flags	Flags
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 Print Window section 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.

Conf 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.

Handle Table

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). Double clicking on a handle table entry does nothing.

Diagnostics

Indicates the version of smx and the processor/memory model. Displays coarse profiling information (percent idle, work, and smx overhead). Also displays a list of smx kernel errors, 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 we have no idea who caused it. Some errors such as SMXE_OUT_OF_ and SMXE_INSUFF_ are encountered by a task but not necessarily caused by the task. It is not the task's fault that not enough control blocks or heap space was configured, for example. However, 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

smxUSB

smxUSBH

See SMX Middleware Module Displays below.

SMX Middleware Module Displays

This feature is currently available only for CodeWarrior and IAR ARM (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. For CodeWarrior it is necessary to compile the files that define these variables with debug symbolics enabled. In the project window, check that there is a dot in the bug column for the following files (in the corresponding library project):

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

If not, click in the bug column on the line for each file. (Clicking on a folder such as the top-level source folder adds the dot to everything in the folder.) If you have trouble, try building and linking the Debug version of the library instead. Contact Micro Digital for assistance if this doesn't solve it.

For IAR it is unnecessary to enable debug symbolics.

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 `sprintf()` requires an enormous amount of stack, so we do not recommend using it. For example, the PowerPC version allocates roughly 1700 bytes for local variables! Also note that you should protect any non-reentrant C library functions you use with the `in_clib` semaphore.

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.

Modal vs Non-Modal Dialog

The `smxAware` dialog for some debuggers is modal meaning it has to be closed before you can continue to use the debugger to step through the code, inspect variables, etc.

The dialog for CodeWarrior and 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 your code, since data is transferred after each step. (Note that there are problems with some versions of CodeWarrior when stepping with the `smxAware` window open. If your version has a problem, open the Options dialog and check "Close window on each run". This will automatically close it when you step or run.)

The lag depends on how much data is read from the target via the BDM/JTAG connection and the speed of that connection. The Task window causes the most delay since the TCB is the

biggest control block and every field of every TCB is read from the target after each step or run. If you are viewing semaphores or exchanges, much less information is transferred so stepping is faster, assuming you don't have a lot of these in use. Closing the smxAware window when you don't need to watch it will make stepping through the code faster. The Diagnostic and User windows transfer very little. In the Options dialog, you can check "Close window on each run" to make it automatically close each time you step or run, and then you only have to open it when you want to see it.

The same lag occurs when the CodeWarrior Process list is open (see next section). The debugger does not tell the DLL when this window is closed, so the DLL will continue to read information from the target after each step or run. Workaround: After closing the Process list, re-open and close the smxAware window. This will set smxAwareOpen to false, in the DLL, disabling smxAware until you open either window again.

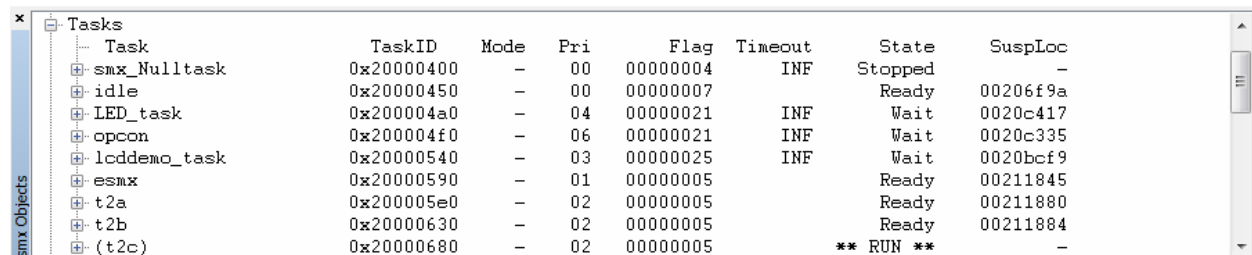
Suspended Task Information

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. 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 from there (and view call stack, variables, registers, etc.).

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. xarm_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 **	-

CodeWarrior

Note: This feature may or may not work for your version of smx, smxAware, and CodeWarrior, due to changes in CodeWarrior. It requires smxAware to know the stack layout for a suspended task. Due to changes in code generation for each new release of CodeWarrior the stack structure would change so this feature would break.

While the smxAware window or Process list is open, CodeWarrior opens a new Thread window each time execution stops in a new task, and the task name is shown in the title bar. Thread windows can also be opened from the Process list (View | System | [connection type]): Click once on “smx Process” in the left pane to get a list of tasks in the right pane, then double-click on the task in the right pane to open its thread window. Note that the Process list is limited to displaying only 2 states for the task, Running and Suspended. Unfortunately, it supports only 2 states, so this can be misleading because smx distinguishes between suspended and stopped states. For more specific task state information look at the State column in the Task display in the smxAware window.

Note: If you are connected to the target with something other than just the P&E parallel wiggler (e.g. the USB MultiLink, Lightning card, Abatron, etc.), you need to change the IDE preferences so the Process list works properly: Edit | Preferences, select Remote Connections in the left pane, select the interface you are using (e.g. Lightning), press Change, and then check the box “Show in process list”.

The main use of the Thread window for a suspended task is to look through the call stack to see where the task has been and where it will execute next — the sequence it ran through before it was suspended.

The **Thread windows** are blank for stopped tasks. For other tasks (suspended, running), they show:

1. call stack
2. local variables
3. code

The **call stack** pane shows the sequence of nested calls that led to the last execution point. Clicking on each item (routine) in the call stack will change the view in the code pane to show the last point that executed in each routine. Lines that look like this: “VECTOR_TABLE <0x00000000>” are functions from libraries that were compiled with debug symbolics off, so the function name is unknown. In the case of a task suspended on an SSR (smx call), the lowest entry will be for smx_SSRExit() and the one above it is the SSR that the task called that caused the task to suspend. If you were to build the smx library with debug symbolics on, you would see these names there.

The **local variables** pane shows any local variables for the routine in which that the task was suspended. It will be empty if the code pane shows disassembly, since debug symbolics are not available. The Location column indicates the address or register that stores each variable. Do not

trust the values shown for variables stored in registers since the register value could have changed (several times) between the particular function in the call stack and the point where it is saved in the scheduler.

The **code** pane initially shows where the task will resume. This is the instruction following the last one to run before the task was suspended. If it shows address 0, that means the task stopped or was not yet started. Stopped tasks have no task state; only suspended tasks do. Most often, tasks are suspended as a result of an smx call. Since the smx library is provided in pre-built form without debug symbolics, the code window will show disassembly. If you have smx source code and rebuild the smx library with debug symbolics on, you will see source code in the code window. When a task is suspended due to an interrupt, the code window will show source code if the point of interruption is in a module compiled with debug symbolics enabled.

Registers window:

1. Each time a new Thread window is opened, a node is added to the Registers window for that task. Expanding the node shows its registers.
2. 0x77777777 indicates a register whose value is unknown. For better performance, smx saves the minimum number of registers on a task switch. More are saved for suspend on interrupt than suspend on call. For those not saved, we simply display 0x77777777.

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.

CodeWarrior

CodeWarrior does not yet support task-specific breakpoints.

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.
3. Right click the line with the breakpoint, 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): 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: The task does not have to be named or in the handle table to use the task-specific stepping toolbar features.

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.

SingleStep

The task-specific breakpoint will be triggered only if it is hit while the specified task is running. To use this option:

1. Run the application long enough to start the tasks.
2. Set a breakpoint.
3. Right click on the breakpoint marker to bring up the Modifying Breakpoint dialog box.
4. Select the task icon (the icon with a circle, triangle and square).
5. Select one or more tasks (expand the task list if necessary).

Configuration

smxAware.ini stores smxAware state and configuration settings. It is automatically created with default values if it does not exist. It stores some values about your previous session, such as whether you had certain buttons enabled, window size, etc. Currently, the only configuration options are for Graphical Analysis Tool, so it is documented in the Configuration section of the GAT section.

smxAware GAT (Graphical Analysis Tools)

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. Also, you can display the Event Buffer in text form with **smxAware | Event Buffer** from the menu and memory usage with **smxAware | Memory Usage**.

This feature is currently implemented only for CodeWarrior and 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 Vista/7 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

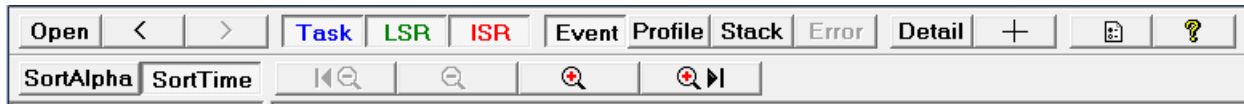
Color Key

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



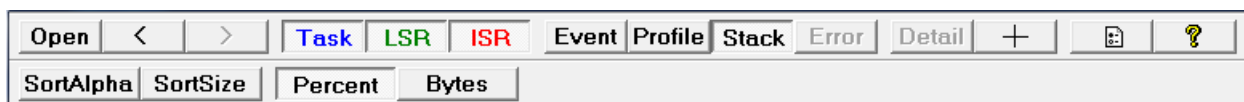
Open	Open saved trace files.
< >	Scan through saved trace files.
Task, LSR, ISR	See “Event Timelines/ Filters” below.
Event	Event Timelines display. See “Event Timelines” below.
Profile	Profile display. See “Profile” below.
Stack	Stack Usage display. See “Stack Usage” below.
Error	Error display. See “Error Buffer” below.
Detail	Details window. See “Event Timelines/ Details Button” below.
+	Crosshairs
Options	Configuration settings. See “Options Dialog” below.
?	Help
SortAlpha/Time	Re-orders lines. See “Event Timelines/ Sort Buttons” below.
-/+	Zoom. See “Event Timelines/ 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. Unlike other similar tools, the display is clean and un-cluttered, making it look deceptively simple. This section discusses the capabilities of this display, some of which might not be immediately evident.

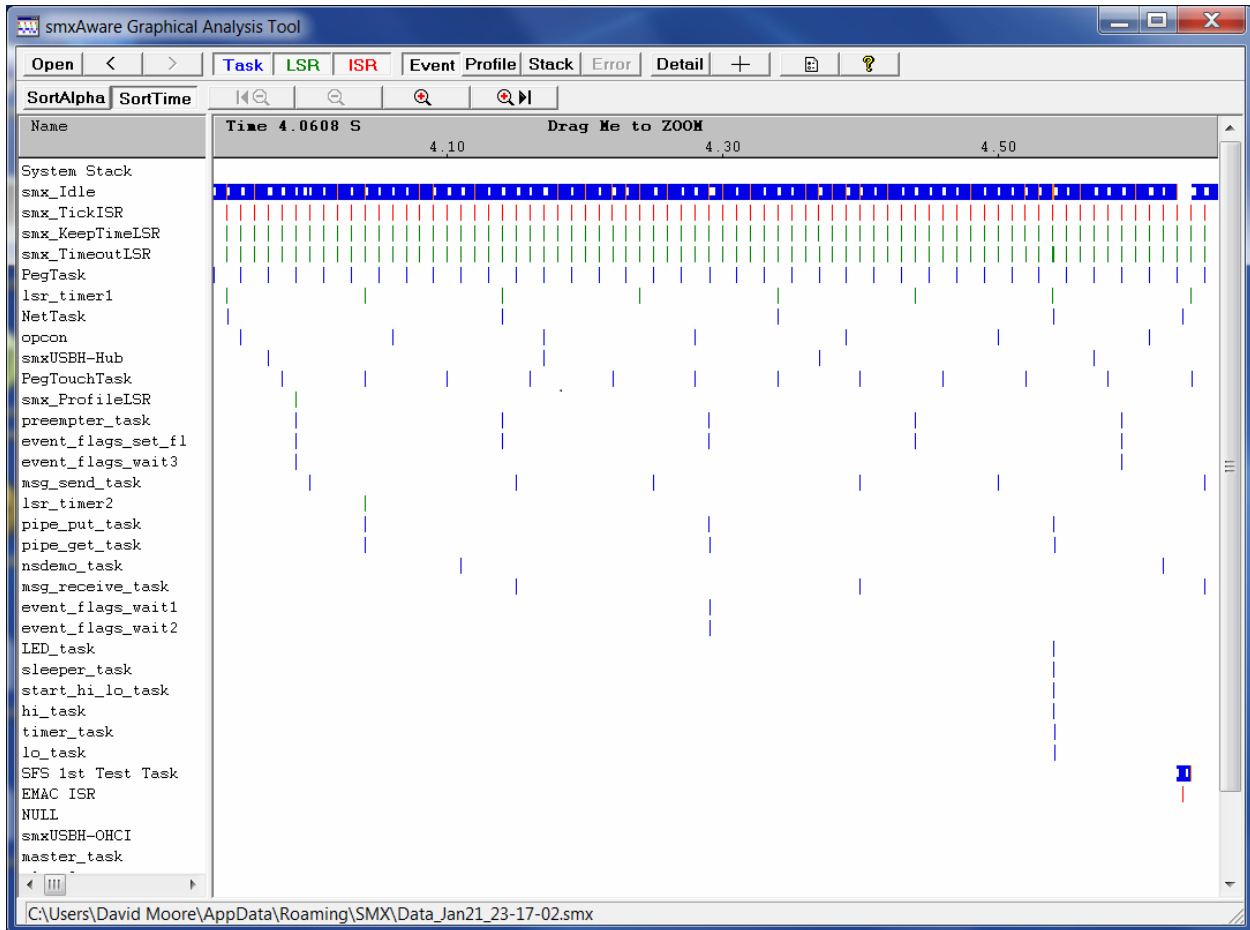
When the window is first opened it is zoomed all the way out, showing the entire trace. See “Zoom” below for discussion of zooming in and out.

Setup

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

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 the Event Timestamps section below. For more information about the Event Buffer, see the smx User’s Guide.

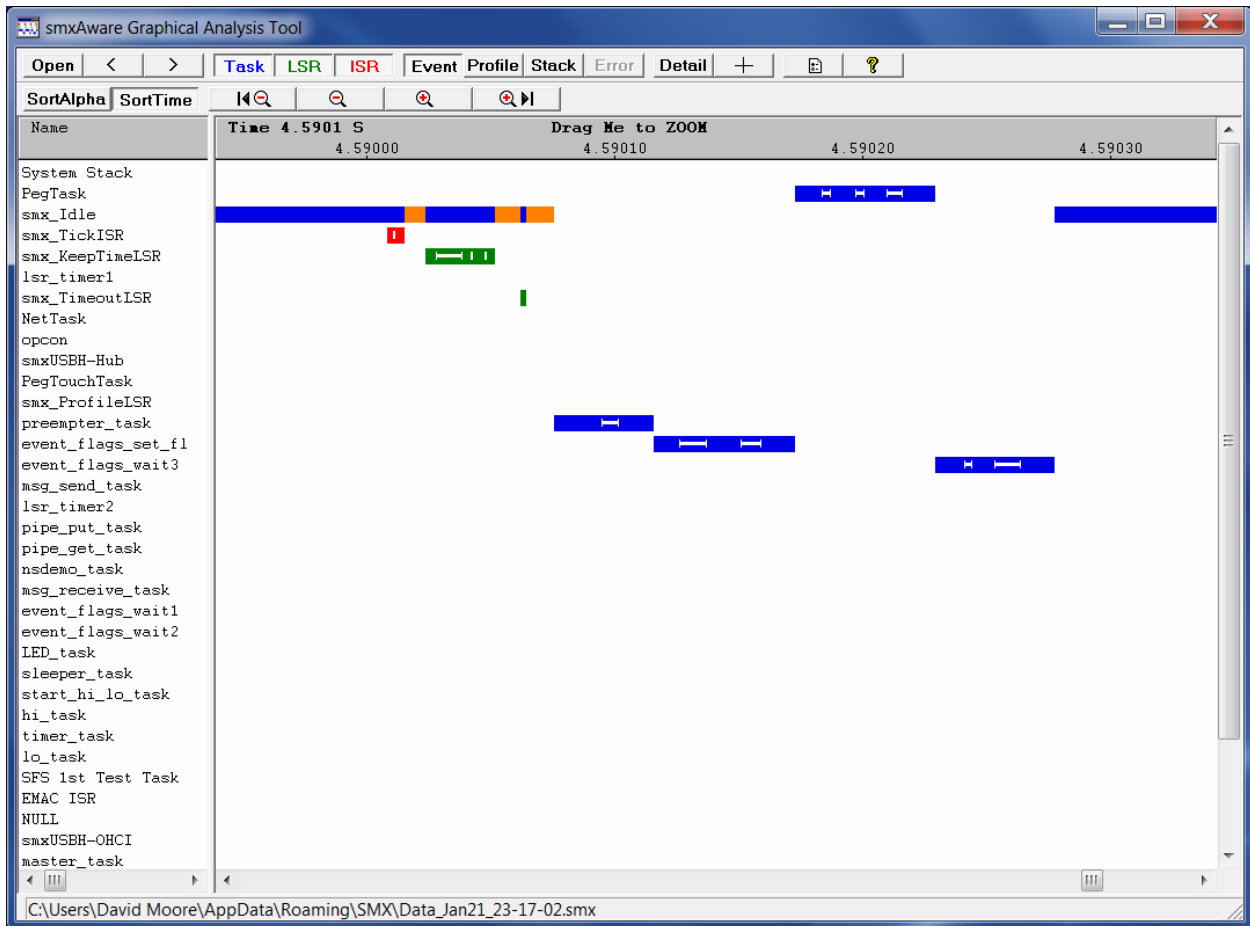
Event Timelines



Notes about this sample:

1. The window shows the whole trace initially. It can be zoomed in very finely. If you are reading this before you have smxAware, we recommend you try the standalone demo on our web site (<http://www.smxrtos.com/demo>) so you can see how much it can be zoomed in. Also see the next graphic below.
2. Notice the regularity of the smx_Tick_ISR and the smx_KeepTimeLSR. If you zoomed in, you would see that smx_KeepTime 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. In older versions of smxAware, they were stored in the same directory as the smxAware DLL (under your compiler directory). Now, due to security restrictions in newer versions of Windows, they are stored in C:\Users\\AppData\Roaming\SMX in Windows 7 or C:\Documents and Settings\\Application Data\SMX in Windows XP. (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 the section Options Dialog below.

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 time between events.

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. ISR invoke events appear as a single tick mark.

Error Dots

Errors appear as red dots on the bars 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. 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 LSR or ISR that is not tagged with smx_EVB_LOG_LSR() and smx_EVB_LOG_LSR_RET() (or ISR) 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

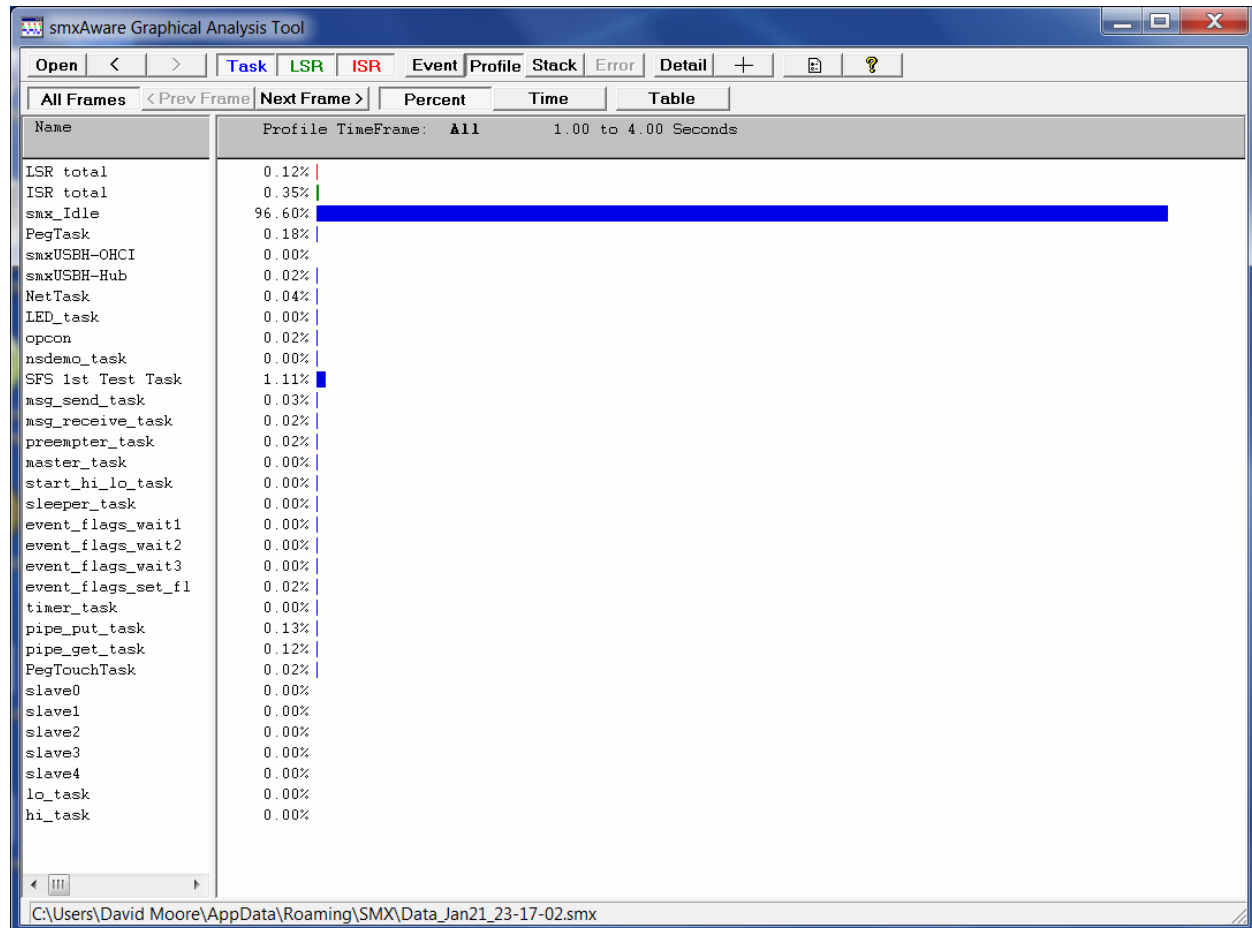
Set the reference line at the left edge of the event (right-click the mouse) and move the pointer to the right. The header shows the Delta time between the reference line and mouse pointer.

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().

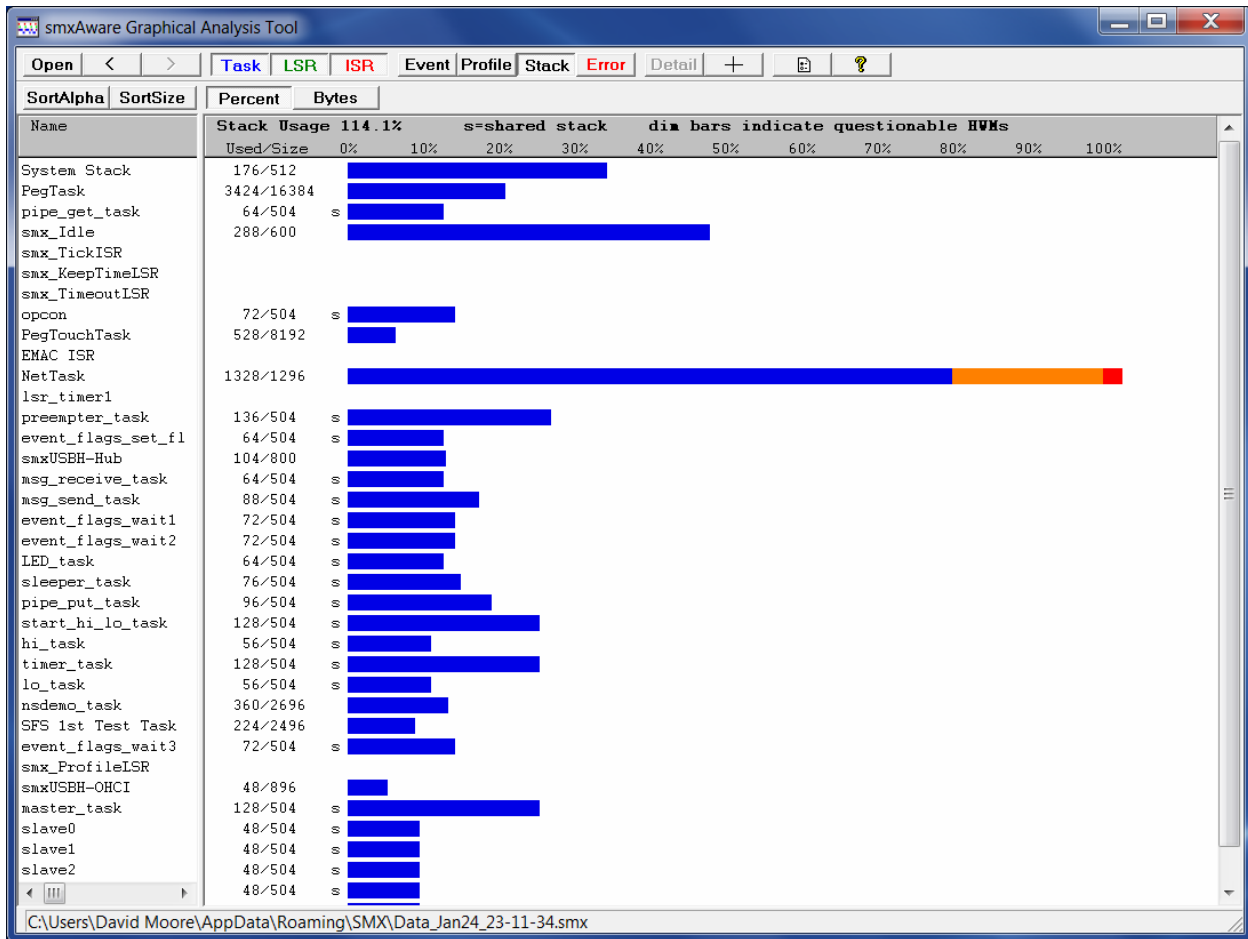
Clicking the Table button shows the data in tabular form:

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_fl	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 tasks as a percent of each stack's size, so you can see how big to make your stacks and 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 stack sizes. 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. Most stack scanning is done by the idle task.

- 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_StackStack()` (`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 (`STACK_PAD_SIZE` in `acfg.h`) so the system will continue to run after overflow (if it overflows 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.

Error Buffer

For convenience, the Error button was added to allow you to inspect the error buffer from the Graphical Analysis Tool, so you don’t have to switch back to the `smxAware` text window to look at it. The error buffer 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 the section about the `smxAware` Diagnostic window earlier in this manual 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.

```

smxAware Event Buffer
Open Save Copy Task LSR ISR SSR Error Invok User All ?
Find: Next Prev Match case Color
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
C:\Users\David Moore\AppData\Roaming\SMX\Data_Jan24_20-51-27.smx
  
```

Buttons

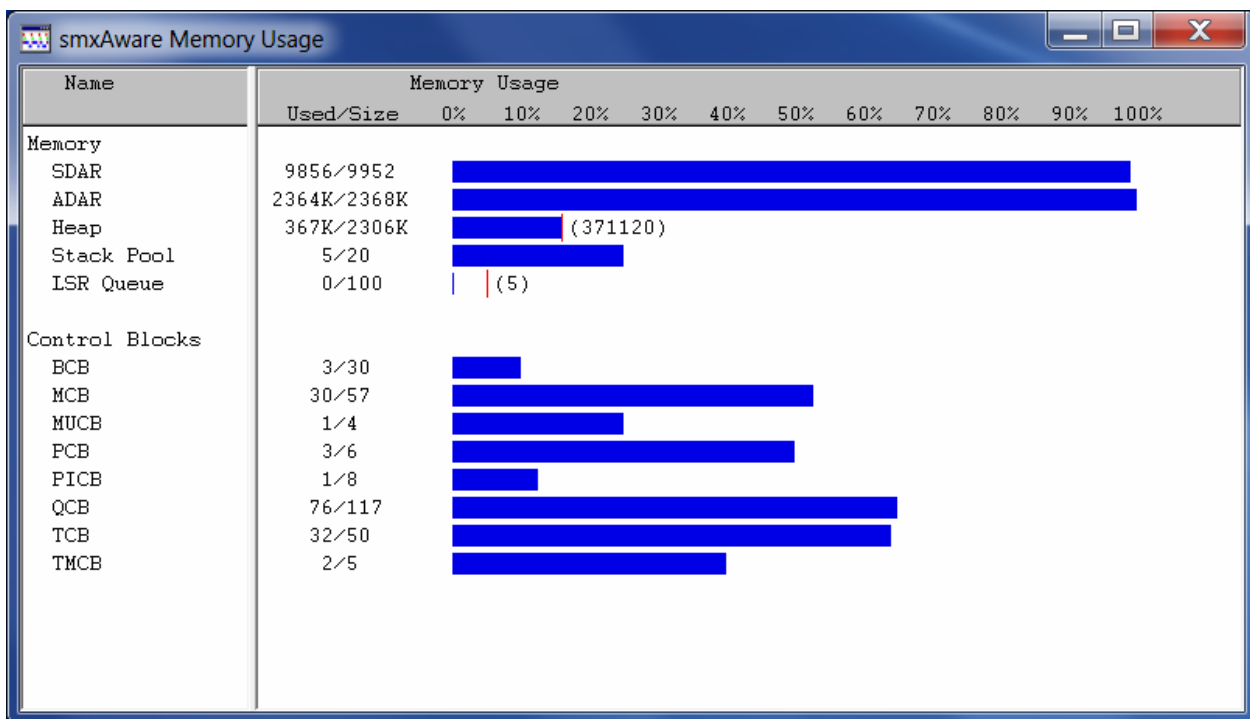
- Open Open saved trace file.
- Save Save text to a file. Saves filtered lines only (i.e. what appears in the window). To save all lines, ensure all filters are depressed (press the All button).
- Copy Copies to clipboard. Same note as Save.

- 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.

Memory Usage

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



Notes:

- By default the heap and stack pool are allocated from ADAR, so ADAR usage reflects the memory occupied by these.
- 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.

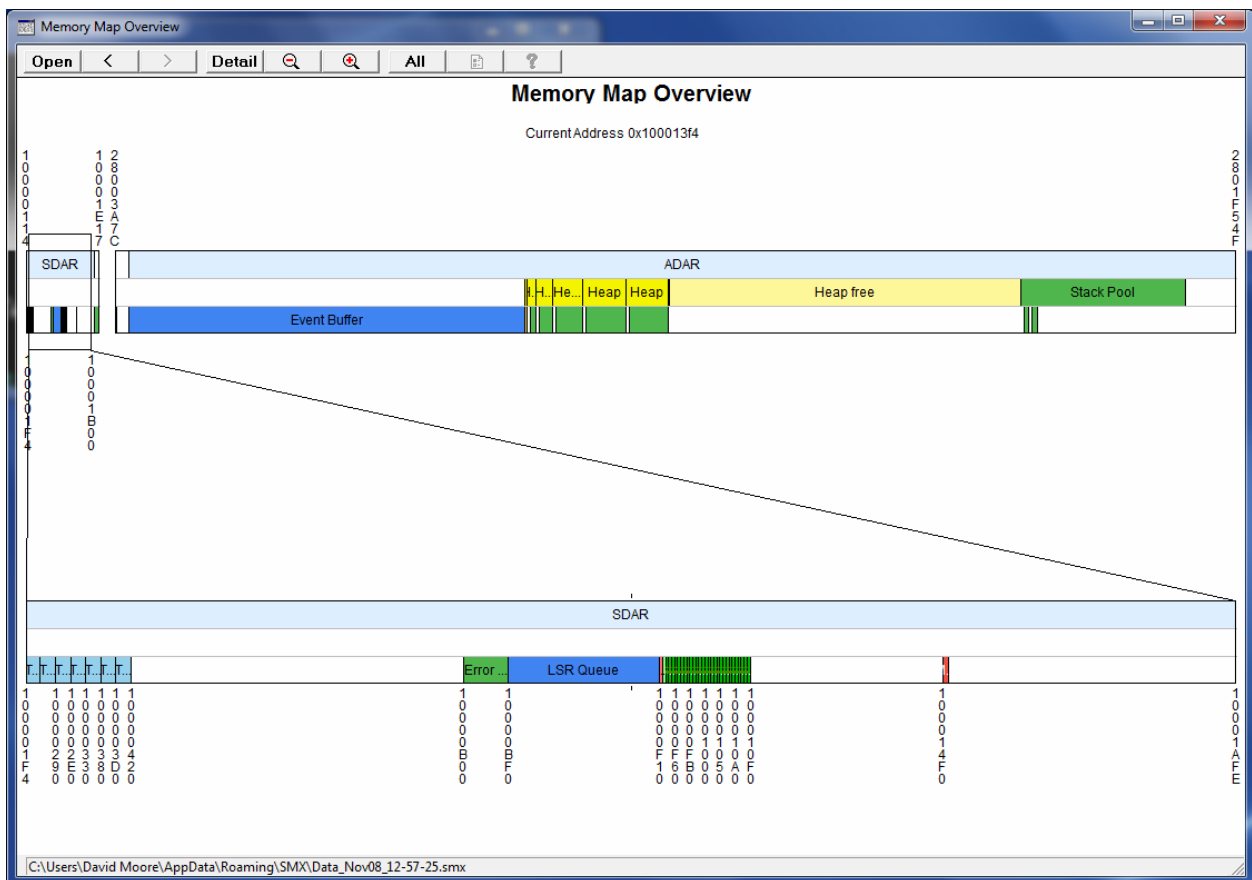
Memory Map Overview

This feature requires **SMX v4.1 or later** and is currently only supported for **IAR EWARM**.

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.



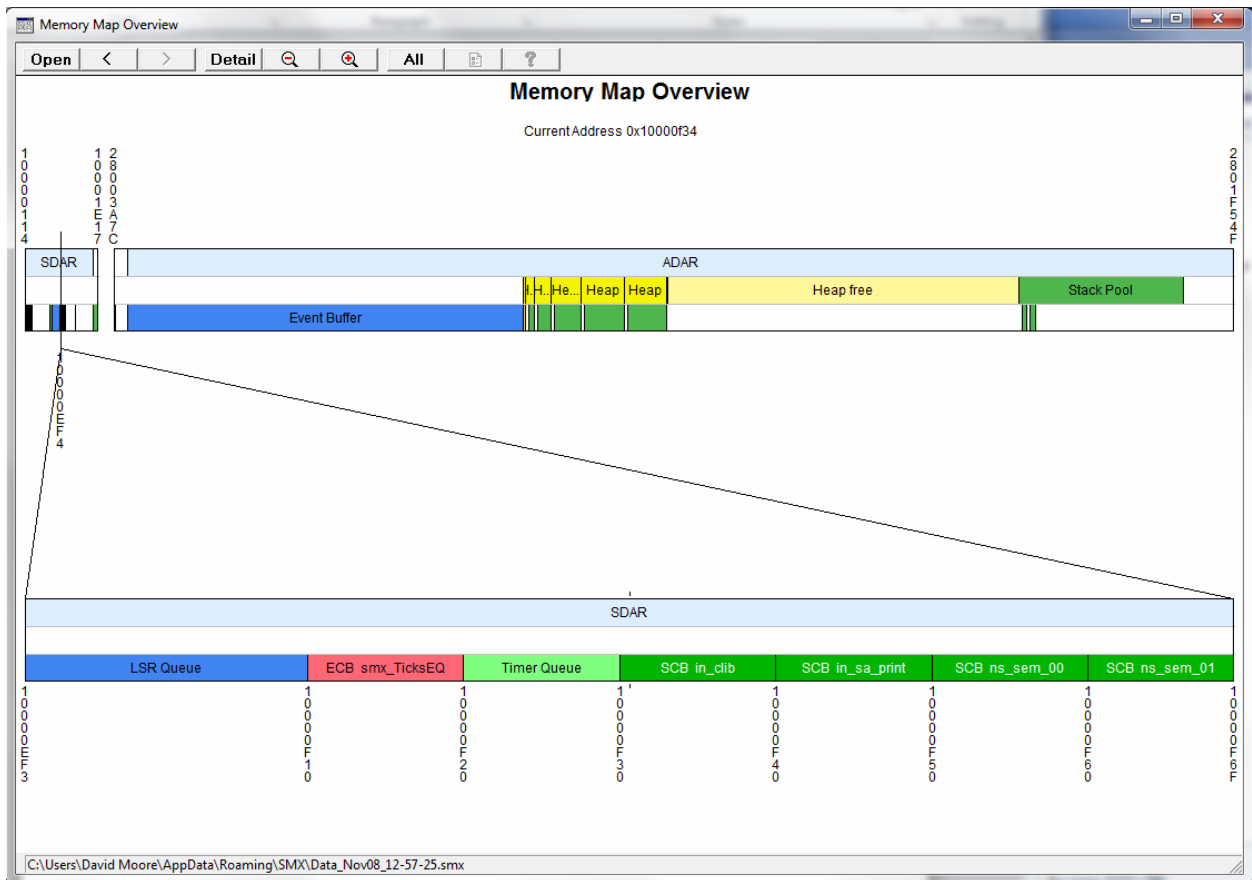
The top memory band has a Magnifier Rectangle that the user can drag, stretch, and shrink to display an area in the lower band.

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:

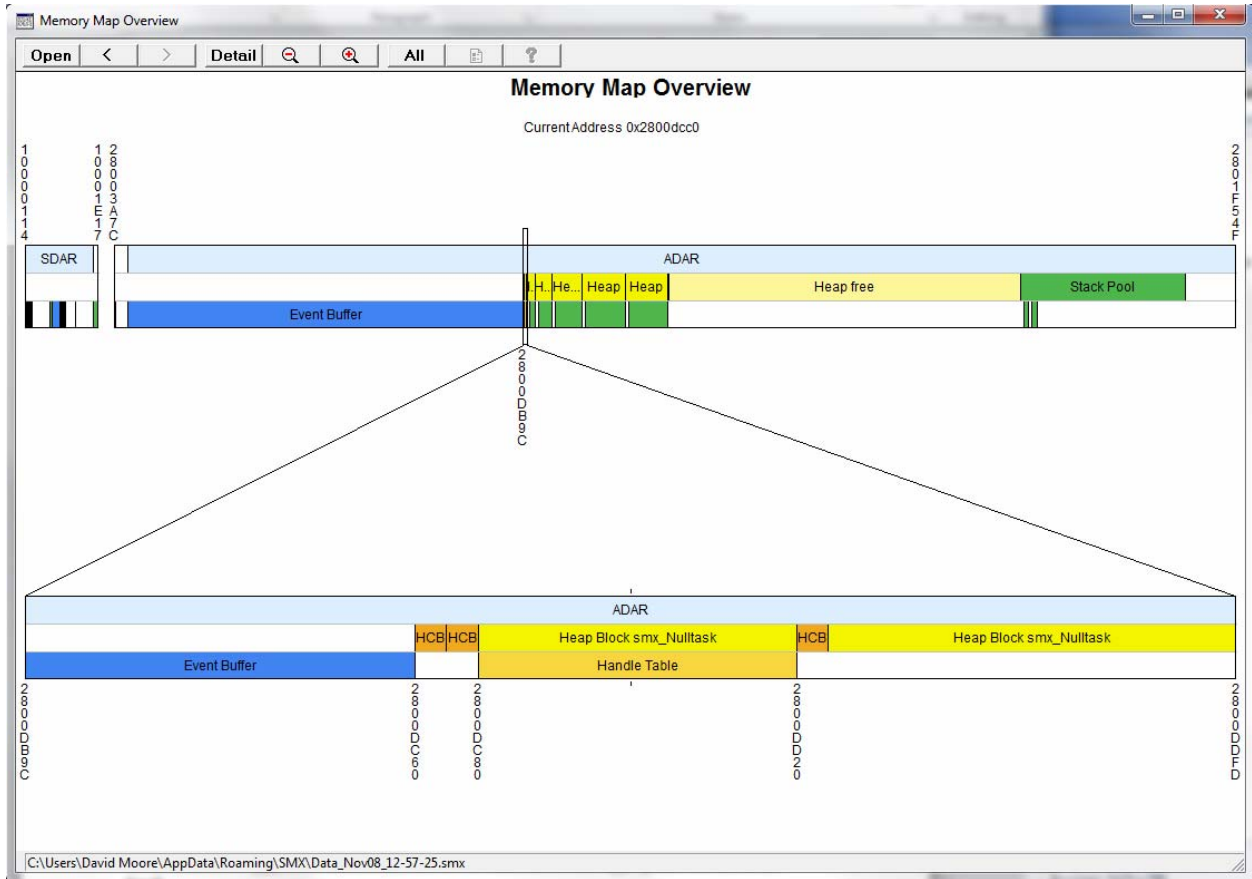
- The top bar is for DARs.
- The middle bar is for the Heap, Stack Pool, and Control Block Pools.
- The bottom bar is for Stacks, Control Blocks, and all other memory types.

The idea of the bars 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. 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.

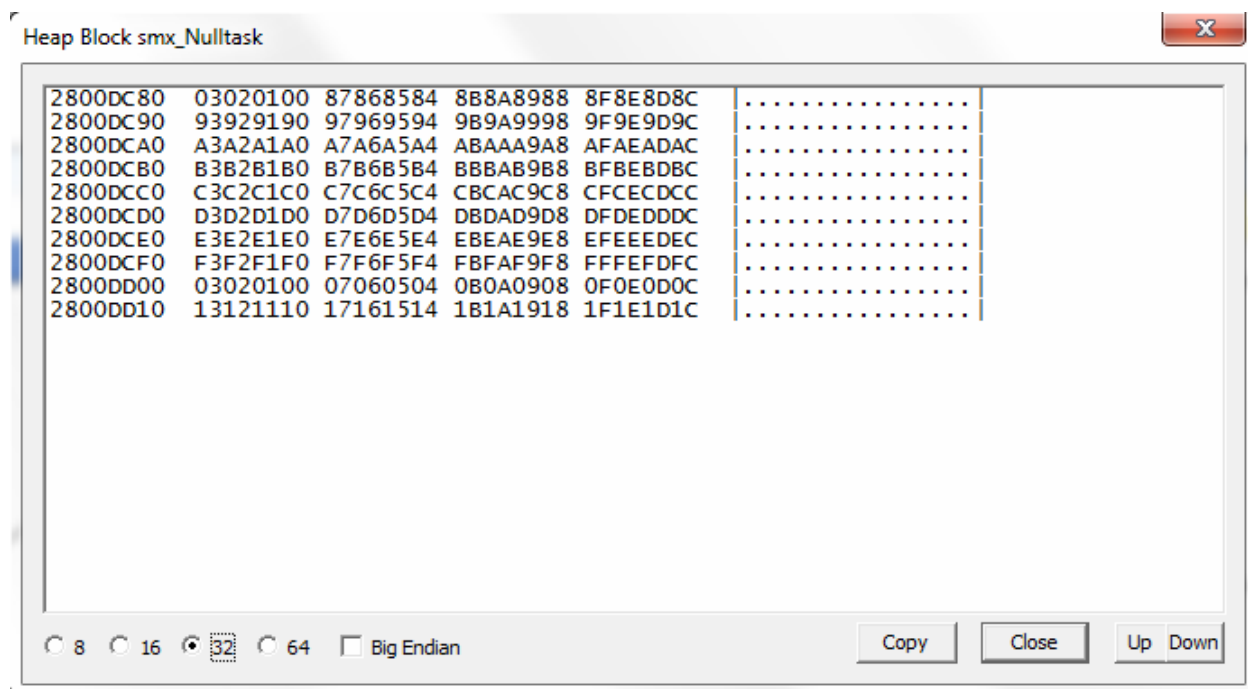
Toolbar Buttons



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.
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.

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 `smx\app\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);
```

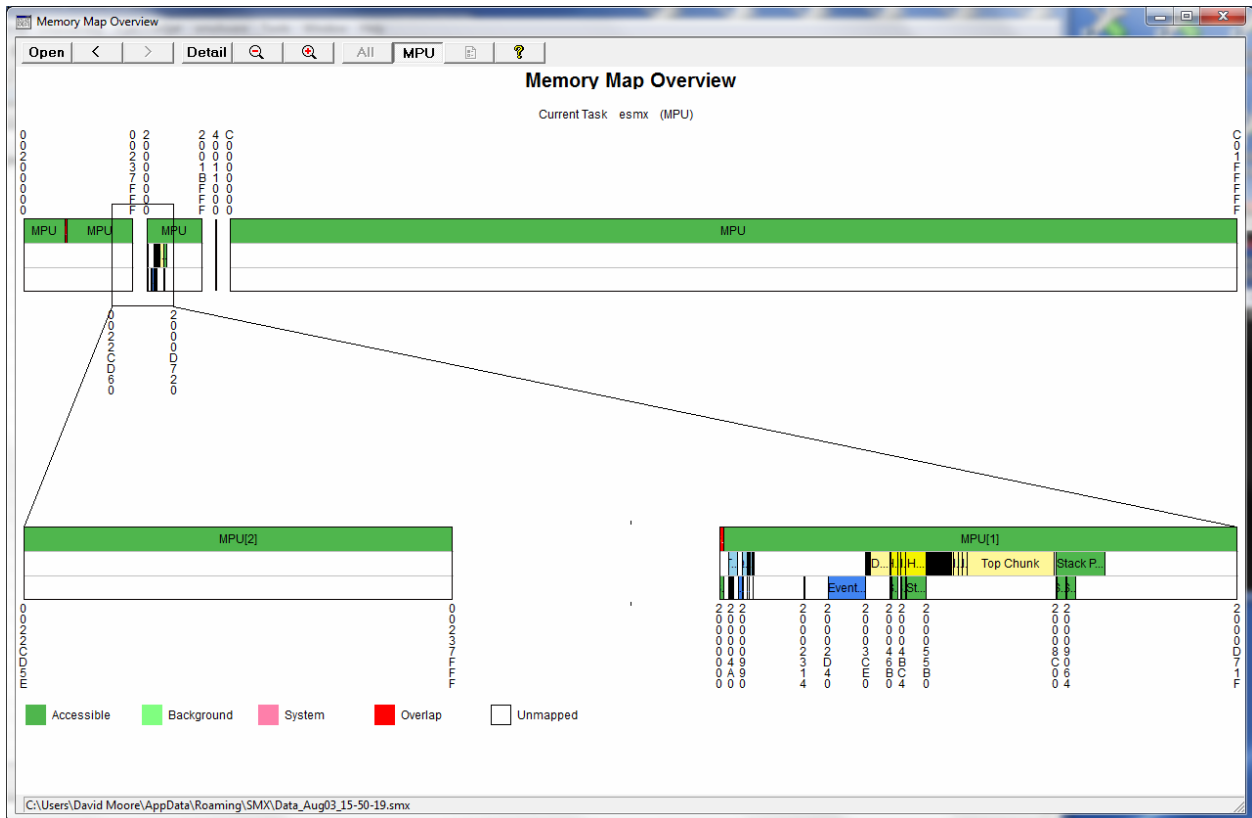
where RAM and SRAM are regions defined above in the file.

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 Cortex-M targets, if MPU-Plus is present, 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:



Configuration

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 `smxAware GAT/ Event Timelines/ Saved Traces`). See the section `smxAware.ini`, below, for information about other 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

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.

Note: To avoid logging particular ISRs and LSRs, comment out the `smx_EVB_LOG` macros in them. 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. Also, some versions of CodeWarrior ColdFire have a problem that causes `smxAware` to hang if this option is enabled. If you have this problem, select the next option.

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.

smxAware.ini

This file stores smxAware state and configuration settings. It is automatically created with default values if it does not exist, in C:\Users\\AppData\Roaming\SMX in Windows 7 or C:\Documents and Settings\\Application Data\SMX in Windows XP. 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.

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. Example times for a 1500-entry buffer:

ARM JTAGjet USB 2.0 (CW)	4 sec
ARM JTAGjet USB 2.0 (IAR)	0.5 sec
ColdFire P&E Wiggler:	22 sec
ColdFire P&E Lightning card:	3 sec
PowerPC Macraigor Raven:	2 sec

Application Preparation

smx events are automatically logged in the Event Buffer. However, it is necessary for you to add macros to your ISRs and LSRs 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 and LSRs 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_PTR 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, smx_KeepTimeLSR, 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 xesr.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_EVB_LOG_LSR() and **smx_EVB_LOG_LSR_RET()**

Add these to LSRs that you want to log in the Event Buffer. Put smx_EVB_LOG_LSR at the beginning of the LSR, and put smx_EVB_LOG_LSR_RET at the end.

Example (assumes isr1_handle defined as in “Pseudohandles” section above):

```
void isr1(void)
{
    smx_ISR_ENTER();
    smx_EVB_LOG_ISR(isr1_handle)
    // ISR code
    smx_EVB_LOG_ISR_RET(isr1_handle)
    smx_ISR_EXIT();
}
```

Assembly language macros are not provided, but shell functions are available in xesr.c that can be called from assembly LSRs. 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_EVBlogInvoke() 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.

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 "Using smxAware/ Print Window" earlier in this manual 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.

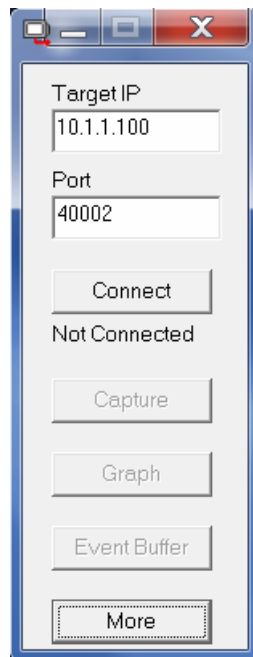
Installation

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. On Windows 7 and later, most of the title bar is covered with buttons; 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, some 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 APP\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(), which is called by main(), so run past that point before opening the smxAware window. In older versions of smxAware, this message could also be caused by inability to read certain important smx globals from the application (often because they were deadstripped by the linker). Other error messages in this section were added to report those problems, so this message is less likely to be seen, if you have run past smxaware_init().

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 library 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.

Limitations

Tips

1. If your smx application doesn't execute properly, put a breakpoint in function smx_EMBreak() in main.c (or smx_EM() in xesr.c). If smx runs out of resources or has another error, it will call this function.
2. If stepping is slow when the smxAware (non-modal) dialog is open, either close it or change it to a different view that shows less data, such as the Diagnostics window. Or open the Options dialog and check "Close window on each run" to close it automatically.

Troubleshooting

Note: Starting with the May 2013 release, smxAware links the static version of the C RTL, so the notes below about DLL load problems (e.g. MFC42.DLL) no longer apply. The notes are preserved temporarily for those using an older version.

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).

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 and any 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 DLL does not load (not in IDE menu) or GAT EXE does not run.

Cause: Possibly we sent a Debug version of either, which requires MFC42.DLL and/or MSVCRT.DLL, but these are old or missing from your system. These are the Microsoft Foundation Classes DLL and C Run-Time Library DLL, respectively. It could also be that we sent you the Debug version of the DLL instead of the Release version, by mistake. The Debug version needs Debug versions of these libraries (MFC42D.DLL and MSVCRTD.DLL). These are not provided with Windows.

Solution: Request the smxAware Release version DLL from us, or copy these DLLs from Visual C++ v6 (or later) or another source to your Windows SYSTEM32 directory. They should be dated 6/17/98 or later. If you have these DLLs, the problem may be that you need the Debug DLLs as explained above. If you have Microsoft's dumpbin utility, you can run this command to see what DLLs it needs:

```
dumpbin /dependents <dllname>.dll
```

This utility is provided with Visual C++, as are the D versions of the libraries. If you don't have it, ask us to check this for you.

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 library) or certain files in other SMX libraries (see section SMX Middleware Module Displays) with debug symbolics on. If you do not have smx source code, contact Micro Digital to rebuild the smx library for you.

Solution: Ensure the smx library project file or makefile is set to compile xglob.c with debug symbolics.

Problem: GAT window does not open and instead a file open dialog appears. Occurs on Windows Vista and newer versions of Windows.

Cause: This is likely caused by User Access Control (UAC) of Windows Vista and newer versions of Windows. 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 and running on Windows Vista or 7, 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: Error: Internal error kc_getval : ovl = ??

Cause: SingleStep can't find one of the required symbols in the symbol table.

Solution: Make sure you are building and running the Debug version of the application.

Problem: SMXE_INV_QCB 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_QCBS was reported, increase NUM_QLEVELS in APP\acfg.h. Ensure smxaware_init() is being called from smx_Go().