

## SeeCode: A New Approach to a Debugger

By implementing a Semantic Inspection Interface (SII) into its SeeCode debugger, MetaWare has made it possible for developers to have easily interpreted information during their debugging sessions. In addition, SeeCode's object-oriented architecture makes it easy to debug multiple threads, processors, and/or RTOSs within a single debugger session by using multiple instantiations of the appropriate debugger tool.

### INTRO

**D**ebuggers give programmers a view into their programs through the lens of the programming language used or the assembly language of the computer. Source and assembly listings, structured data display, and call stack traces are all useful in a debugger and are expressed in terms of either of these two languages.

But none of these facilities provides a deep or semantic understanding of the program being debugged. An array in the program might represent a waveform, but to the debugger it is only an array of integers or floating point numbers and that's how the debugger displays it. That the array represents a waveform is solely in the programmer's head.

To gain better insight into the state of a program, MetaWare has created the Semantic Inspection Interface (SII) API for its SeeCode debugger. Semantic inspection means that the user, possessing knowledge the debugger lacks about the program being debugged, can write code that inspects the state of the program and generates meaningful results. These results can take many forms; for example:

- summary data about the state of the program

- data sent to another process for graphical display
- analysis of the program in an attempt to find the reason for a bug

SII has been used for a wide variety of purposes, including RTOS awareness, signal visualization, profiling, high-level program design, Ethernet device simulation, and low-level simulation counters.

### THE SII API

SII affords a bi-directional communication mechanism with the debugger. The implementer of an SII instance codes a small set of functions. The key function is notification(). It is the method the debugger uses to communicate with the instance. notification() is called whenever an event of interest occurs, such as process or thread creation or destruction, a breakpoint, or a timer update. The notification method is declared as:

```
int notification(SI_notify_struct *)
```

The SI\_notify\_struct parameter contains the reason for the notification (for example, breakpoint), the process ID and thread ID, and other items relevant to the notification reason, such as the address of a breakpoint.

The debugger gives each SII instance a callback API object that implements access to the program being debugged. This access allows memory read/write, expression evaluation, symbol lookup, and so forth. Upon notification, the SII instance can invoke methods in the callback.

To avoid problems with multi-threaded programming, the debugger permits use of the callback API only when the SII instance has been called by the debugger. The SII instance cannot spawn another thread and invoke callback functions at any arbitrary time because that could corrupt debugger internal structures. Thus, the SII instance performs its job all in the context of a notification() call.

### SIGNAL VISUALIZATION TOOL

Perhaps the "whizziest" use of SII is a signal visualization tool (SVT). The SVT SII provides a connection to MATLAB®, a popular math workbench product that has extensive facilities for graphing (<http://www.math-works.com/products/matlab/>).

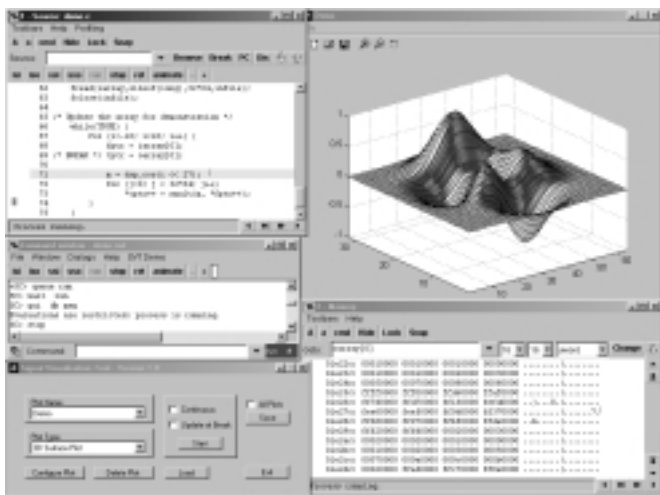


Figure 1. Illustrates how SeeCode can work with third-party programs to create additional data displays, including graphical displays. This example shows a PC MATLAB display.

# DEBUGGING

When the SVT SII is loaded, it starts up MATLAB in another process and establishes a shared-memory connection to it. A GUI within MATLAB allows you to select arrays within the program being debugged and specify how you want to graph the array data. The SVT SII accesses the program data - the array contents - and transmits the data to MATLAB upon request. Using SVT you can get a real-time picture of your data as your program is computing it. For example, you can watch a waveform develop as it is being computed, or notify the SVT SII to transmit array data to PC MATLAB when the debugged program hits a breakpoint.

The MATLAB GUI is fully active regardless of what the debugger is doing, whether the debugged program is running or stopped. The MATLAB user can change the request to view a different array, or specify different parameters. How can the SVT SII service MATLAB requests at any time? The answer is that it cannot - again, it can do something only when it gets a notification() call. But the SVT SII can arrange to be called periodically no matter what: that is the purpose of the set\_timer\_update function, which requests a notification from the debugger every so often. Whether the program is running or not, the debugger will call the SII periodically. Setting the timer delay to, say, 100 milliseconds gives the MATLAB user the appearance that the debugger is an attentive "server" to MATLAB. Thus, no matter what is happening in the debugger (the user can be single-stepping, setting breakpoints, running, etc.), MATLAB can always get a "time slice" from the SVT SII, and therefore get the information it needs.

Because all debugger APIs are object oriented, there may be multiple instances of any API. Thus the user can ask the debugger to load the SVT SII more than once. This gives the user the ability to have two instances of PC MATLAB, each displaying different data.

## HIGH-LEVEL PROGRAM DESIGN

I-Logix ([www.ilogix.com](http://www.ilogix.com)) produces a tool called Rhapsody, an object-oriented analysis, design, and implementation environment. From a press release:

*Rhapsody allows the real-time embedded software engineer to design object-oriented software graphically, and to generate and test production-quality code, deployable in an embedded application. Graphical animation allows the real-time designer to visualize the application running in design form, either on the host or the target system. This enables applications to be tested before the target hardware is available.*

Graphical animation is achieved by instrumenting the C or C++ code generated by Rhapsody with calls to a C run-time library that sends information about the state of the running program to the Rhapsody animator display. The run time uses TCP/IP to send state information to the animator.

Using SII you can run a program in an embedded device or on a simulator, with or without TCP, and still get the benefit of the Rhapsody animator.

Rhapsody's instrumented code uses macros that you can program to call a specific function in your pro-

gram. Set a conditional breakpoint at that function. Then use an SII instance to capture the data given to that function when the breakpoint is hit; transmit the data to Rhapsody's animator using TCP/IP.

Furthermore, specify that the breakpoint has a condition that is never satisfied (such as  $1 == 0$ ) so that the debugger automatically restarts the program. When you run your program, SII automatically captures all the state transition data and supplies it to the animator, and the animator shows you the state transitions in your program as it runs. A MetaWare customer has done exactly this.

The Rhapsody SII instance isn't a server, as it only needs to send messages to Rhapsody. Unlike the SVT SII, it needn't use timed updates, and calls to notification() on breakpoint are sufficient.

## ETHERNET DEVICE SIMULATION

Wind River System's VxWorks® operating system relies on a UDP (a form of TCP) connection to the host to exchange messages with Wind River's Tornado workbench, which hosts tools such as debuggers and OS analyzers. The tools make calls into the workbench to discover the state of the OS and its tasks; those call requests are sent across UDP to an always-running server task in the OS and an answer is returned.

For the ARC processor the question arose of how to run VxWorks on an ARC simulator or an ARC board without an Ethernet device. SII came to the rescue.

Using timed notifications from the debugger running the VxWorks OS on a simulator or an ARC board, a VxWorks SII instance periodically intercepted UDP traffic from the workbench on the host and wrote the packets directly into the ARC's memory (whether the ARC was running or not; the ARC's memory can be read/written while the ARC is running). A polling task in the OS periodically checked the ARC memory for received packets and, having received the same, directed them to the appropriate task for processing, and put the return packet in a different place in memory. The VxWorks SII instance watched for that place in



Figure 2. Shows how, by using the RTOS API to increase RTOS-awareness, SeeCode can display RTOS-specific windows - in this case, Precise/MQX state window and threads window.

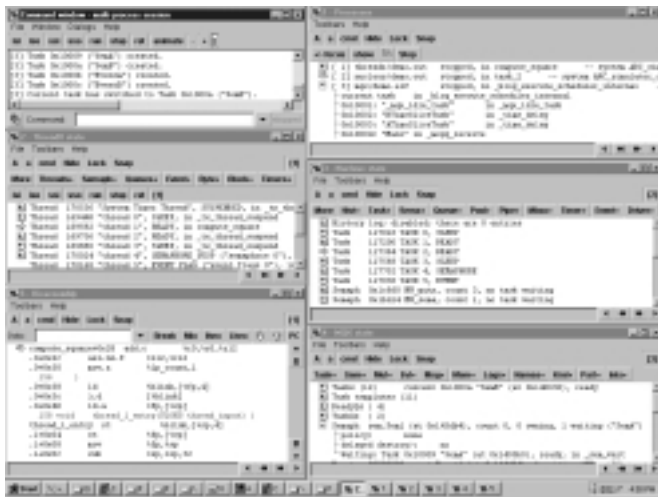


Figure 3. Shows a debugging session with three threaded RTOSs being run on two different architectures: ThreadX and Nucleus on ARC, and Precise/MQX on ARM.

ARC memory to change; when it did, the packet was extracted from the ARC and transmitted back to the waiting workbench. This was the first time that VxWorks had been run on a simulator.

This is an interesting situation because the debugger is being used as the tool to run VxWorks and shuttle UDP traffic to it. At the same time, a debugger is hosted on the workbench, debugging specific tasks in the OS. SeeCode is at the time of this writing being ported to the workbench protocols, so in the end there will be two versions of SeeCode running: one to run the OS and the other to debug the tasks, with one debugger indirectly using the other.

## RTOS AWARENESS

RTOS awareness is a critically important facility when debugging programs on a multi-thread RTOS. The debugger supports multi-threaded RTOSs and can display the state of not only the current task but those that are suspended. To do so, it must know a list of all tasks and the saved register sets for each suspended task. The debugger defines an RTOS-awareness API that a user can program to tell the debugger just this information every time the OS stops.

For Precise's MQX RTOS ([www.psti.com](http://www.psti.com)), an SII instance is used for two purposes.

First, it injects the RTOS API instance into the debugger, using the function `set_RTOS_awareness` in the SII callback. This allows the debugger to get access to non-current tasks, display disassembly or source windows that are thread-specific, and set thread-specific breakpoints and watchpoints. The SII instance is also notified on program creation and termination, and can perform any initialization or cleanup the RTOS API may need.

Second, the MQX SII instance also uses an SII initialization function to add MQX menus in SeeCode's command window. Selection of menu items triggers calls to `notification()`, and the MQX SII instance in response brings up various MQX windows that display

the internal state of MQX or its components. From the `notification()` call, the MQX SII instance can determine which menu item was selected and display the appropriate window.

## SII IS OBJECT-ORIENTED

APIs are typically either a collection of functions in a DLL or a function table. Either approach has serious limitations in expressing complex situations such as those illustrated in this paper.

The MetaWare SeeCode debugger's APIs are all object-oriented. This allows multiple instances of each API, indispensable in debugging multi-CPU applications, simulating multiple peripherals of the same type on a single chip (for example, two UARTs), or in adding more than one copy of a feature (such as signal visualization) to the debugger. The Coordinated Multi-Processor Debugging facility for SeeCode allows debugging multi-processor applications of up to 256 processors in a single debugging session.

The ability of a programmer to get to market quickly may depend on being able to get a semantically relevant picture of what his application is doing. The SeeCode debugger's advanced Semantic Inspection Interface was designed exactly for that purpose, and to date has been used in startling ways unanticipated when SII was first conceived ■

---

*Tom Pennello did his graduate work under Franklin L. DeRemer at UCSC. Together they developed fully-automatic error recovery and efficient LALR lookahead set computation. Thomas's industry experience includes creating C, C++, and Pascal compiler front ends, and both RISC and CISC optimizing code generators. Thomas received his Ph.D. degree from UCSC in transformational grammars.*