Developing a LONWORKS Control Network

Emulation and Logic Analysis in Real-Time Debugging

Eric Kuzara
Hewlett Packard
Colorado Springs, Colo.

Eric Kuzara has worked for nine years at Hewlett Packard as an R&D engineer and R&D project manager. He has spent the past four years managing tools for embedded development, including real-time operating system connections, CASE tool development, software test tool development, and user interface development. He has a BS in computer engineering from the University of Michigan and an MBA in finance from the University of Colorado.
INTRODUCTION

Recent surveys have shown a strong trend in the embedded microprocessor applications market toward using real-time operating systems in embedded software applications. Operating systems help developers of real-time systems manage the complexity of their applications and provide a standard interface to their systems' resources. Additionally, when a commercial operating system (OS) is purchased, development time is frequently reduced due to the modular components and libraries supplied by the OS. What has not followed this trend toward real-time OS usage, however, is the availability of real-time development tools that allow a designer to debug a real-time OS as it interacts with the target application.

There are two debug options available to customers who are using real-time operating systems as part of their embedded applications. Option 1: Use a non-real-time debugger commonly based on a software monitor technology. Generally this monitor is linked in with the target application along with the OS kernel itself and provides debug access through which the OS data structures and communications mechanisms can be interrogated and controlled to some degree. Option 2: Use real-time emulation analysis or logic analysis.

Option 1

The first option of a non-real-time debug monitor has some advantages. Since most commercial OS vendors supply their own OS debug monitor, the developer has access to many internal resources of the OS that are not normally accessible. The user is able to browse through the internal OS resources to see how his application is affecting the system resources. Displays showing the current state of the application tasks (running, blocked on resource, blocked on time slice, etc...), the current contents of mailboxes (how many and what type of messages they contain), and the current state of event signals (event 1, 3, and 4 are pending, all others are cleared) are common views that debug monitors provide. Some monitors also provide high level run control such as setting a breakpoint when you enter task "A" or changing the priority of a given task at run time.

There are also some serious limitations, however, to using this OS debug monitor technology. The most obvious and severe limitation is the fact that debug monitors are static mode tools that must stop the application from executing in order to read the internal kernel resources and/or change the state of some task. This is unacceptable for many time-sensitive embedded applications where breaking the application execution can cause erroneous or at times dangerous results. And even when this intrusion is tolerable, the user only gets a snapshot of the state of the system at a single point in time and has no idea how the system arrived at that state or where it went from there. This debugging information is therefore not very helpful in tracking down problems related to real time flow of execution at the operating system task level.

Another problem with OS software monitors is the need for a dedicated I/O port to allow communications with the monitor. For many target applications no such I/O port exists. Therefore the port must be specially added to the target for debug purposes thereby increasing the cost and complexity of the target system.

A final OS debug monitor limitation concerns custom operating systems. Even though commercial operating systems are becoming increasingly popular, a large population of customers still use custom operating systems developed in-house to fill specific target requirements. Since OS monitors have been developed separately by each OS vendor, no OS monitor standard exists to help custom OS users who must create their own OS debug tool to get any debug information at all.

Option 2

The second debugging option available to users of embedded operating systems attempts to address the limitation that OS monitors are not real-time. This involves using emulation trace analyzers or logic analyzers to follow the flow of operating system activity. The major advantage of this scheme is the real-time nature of such analyzers that do not change the system flow even when measurements are being taken. It also has the advantage of not needing a special I/O port on the target for debug communications since all communication is handled through the emulator or analyzer probe connected to the microprocessor.

Until recently, however, this debugging method for OS activity was still a poor fit for the needs of the developer. One major limitation stemmed from the fact that these analyzers typically capture all bus cycle activity of a running application. Since operating system activity occurs at a much lower frequency, however, a trace of application activity quickly fills up a trace buffer and results in very few OS transactions being captured. Additionally, setting up such a measurement was very difficult if not impossible since a designer needed close knowledge of the OS in order to correctly set up the trace measurement. Lastly, analyzers do not talk in the native language of the OS kernel making interpretation of measurement results very difficult.

New Measurements Now Available

Tools and techniques are now available which combine the real-time advantage of emulation and logic analysis with the high level OS specific view of OS monitors. This method provides a unique view of OS activity that greatly enhances the developers understanding of how his application is using OS resources and helps debug OS related problems in real-time. It allows a developer to view a historical trace of task switching and to view calls into the OS kernel (known as service calls) intermixed in the same display. Activity can be selectively traced to capture, for example, only OS activity in task "X" or only a specific service call. It also provides a means for setting up trigger conditions on complex high level OS activity to help pinpoint problems. Triggers can be set to occur when task "A" switches into task "B"
INTRODUCTION

Recent surveys have shown a strong trend in the embedded microprocessor applications market toward using real-time operating systems in embedded software applications. Operating systems help developers of real-time systems manage the complexity of their applications and provide a standard interface to their systems' resources. Additionally, when a commercial operating system (OS) is purchased, development time is frequently reduced due to the modular components and libraries supplied by the OS. What has not followed this trend toward real-time OS usage, however, is the availability of real-time development tools that allow a designer to debug a real-time OS as it interacts with the target application.

There are two debug options available to customers who are using real-time operating systems as part of their embedded applications. Option 1: Use a non-real-time debugger commonly based on a software monitor technology. Generally this monitor is linked in with the target application along with the OS kernel itself and provides debug access through which the OS data structures and communications mechanisms can be interrogated and controlled to some degree. Option 2: Use real-time emulation trace analysis or logic analysis.

Option 1

The first option of a non-real-time debug monitor has some advantages. Since most commercial OS vendors supply their own OS debug monitor, the developer has access to many internal resources of the OS that are not normally accessible. The user is able to browse through the internal OS resources to see how his application is affecting the system resources. Displays showing the current state of the application tasks (running, blocked on resource, blocked on time slice, etc.), the current contents of mailboxes (how many and what type of messages they contain), and the current state of event signals (event 1, 3, and 4 are pending, all others are cleared) are common views that debug monitors provide. Some monitors also provide high level run control such as setting a breakpoint when you enter task “A” or changing the priority of a given task at run time.

There are also some serious limitations, however, to using this OS debug monitor technology. The most obvious and severe limitation is the fact that debug monitors are static mode tools that must stop the application from executing in order to read the internal kernel resources and/or change the state of some task. This is unacceptable for many time-sensitive embedded applications where breaking the application execution can cause erroneous or at times dangerous results. And even when this intrusion is tolerable, the user only gets a snapshot of the state of the system at a single point in time and has no idea how the system arrived at that state or where it went from there. This debugging information is therefore not very helpful in tracking down problems related to real time flow of execution at the operating system task level.

Another problem with OS software monitors is the need for a dedicated I/O port to allow communications with the monitor. For many target applications no such I/O port exists. Therefore the port must be specially added to the target for debug purposes thereby increasing the cost and complexity of the target system.

A final OS debug monitor limitation concerns custom operating systems. Even though commercial operating systems are becoming increasingly popular, a large population of customers still use custom operating systems developed in-house to fill specific target requirements. Since OS monitors have been developed separately by each OS vendor, no OS monitor standard exists to help custom OS users who must create their own OS debug tool to get any debug information at all.

Option 2

The second debug option available to users of embedded operating systems attempts to address the limitation that OS monitors are not real-time. This involves using emulation trace analyzers or logic analyzers to follow the flow of operating system activity. The major advantage of this scheme is the real-time nature of such analyzers that do not change the system flow even when measurements are being taken. It also has the advantage of not needing a special I/O port on the target for debug communications since all communication is handled through the emulator or analyzer probe connected to the microprocessor.

Until recently, however, this debugging method for OS activity was still a poor fit for the needs of the developer. One major limitation stemmed from the fact that these analyzers typically capture all bus cycle activity of a running application. Since operating system activity occurs at a much lower frequency, however, a trace of application activity quickly fills up a trace buffer and results in very few OS transactions being captured. Additionally, setting up such a measurement was very difficult if not impossible since a designer needed close knowledge of the OS in order to correctly set up the trace measurement. Lastly, analyzers do not talk in the native language of the OS kernel making interpretation of measurement results very difficult.

New Measurements Now Available

Tools and techniques are now available which combine the real-time advantage of emulation and logic analysis with the high level OS specific view of OS monitors. This method provides a unique view of OS activity that greatly enhances the developers understanding of how his application is using OS resources and helps debug OS related problems in real-time. It allows a developer to view a historical trace of task switching and to view calls into the OS kernel (known as service calls) intermixed in the same display. Activity can be selectively traced to capture, for example, only OS activity in task “X” or only a specific service call. It also provides a means for setting up trigger conditions on complex high level OS activity to help pinpoint problems. Triggers can be set to occur when task “A” switches into task “B”
or when a certain message is sent to a specific mailbox.

This new technique is available now in products offered by Hewlett-Packard. However, with some engineering resources applied, it could be implemented by anyone familiar with emulation or logic analysis. The technique consists of three parts described below. The three parts work together in the following manner: In part 1, the interface library into the OS kernel is modified such that at run time, whenever an OS service call is invoked or a task switch occurs, some encoded data is written out to a predefined data area. In part 2, the analyzer is set up to capture only the special encoded data written to this predefined data area. In part 3, the captured trace data is decoded into the high level OS service call mnemonics and task identifiers to provide a useful and easily readable listing of high level OS activity.

PART I: INSTRUMENTING THE APPLICATION CODE

Section 1

First, one must create a small data area used exclusively for the OS activity measurements. Then, for each OS service call that is to be tracked, a section of this data area is defined that is large enough to hold the incoming parameters and the outgoing return values. For example, if a service call to create a task looked like:

```
CREATE_TASK(u int32 priority, id; u int32 ctl_block)
```

the corresponding assembly language data section would look something like:

```
Start_of_OS_data_block
Enter CREATE_TASK DS.L 3
Exit CREATE_TASK DS.L 1

The three long words declared by the DS.L 3 are intended to hold the three incoming parameters of "priority", "id", and "ctl_block". The one long word declared by DS.L 1 will hold the return value when this service call returns.

A section of this type is defined in the data area for each OS service call that is to be tracked. In addition, another global data section unrelated to a specific service call is defined which will allow task switch tracking. This data section consists of two data words, one for the task being exited and the other for the task being entered. For example, the data declarations might look like:

```
Enter_TASK DS.L 1
Exit_TASK DS.L 1

End_of_OS_data_block
```

This whole OS activity data area should be surrounded by a pair of labels that delineate this special OS area such as those shown in the above examples with "Start_of_OS_data_block" through "End_of_OS_data_block". These labels will be used later.

Section 2

This subsection involves modifying the application code to write out the incoming parameters and outgoing return values whenever the application code executes the OS service call. The modification required to accomplish this depends on some characteristics of the OS kernel and whether source code or binary OS libraries are available from the OS vendor.

For kernels written in assembly language where the assembly language service routines are accessible to the user, tracking of OS service calls takes advantage of the interface library which allows a high level language such as "C" to call the assembly language based operating system service call. This library is a set of functions that correspond directly to each routine available from the OS. Each function in the library is responsible for taking parameters off the stack and placing these values into proper registers. A software TRAP instruction is then executed to pass control to the kernel which interrogates the registers and performs some action. Using this mechanism, an OS kernel can easily be made to work with various compilers by just modifying the interface library.

For example, a Motorola 68000 based assembly language interface to a task create kernel service routine may look like:

```
MOVE.L #CREATE_TASK,D0 ;Set up desired service call id
MOVE.L (SP) ,D1 ;Put parameter "priority" here
MOVE.L (SP+4),D2 ;Put parameter "id" here
MOVE.L (SP+8),D3 ;Put parameter "ctl_block" here
TRAP #0 ;Trap into the OS kernel
RTS ;Return to the application program
```

There is an interface routine like this one in the library for each OS service call in the kernel. To add the capability to track calls to these OS service routines, the following additional assembly instructions are needed:

```
MOVE.L #CREATE_TASK,D0 ;Set up desired service call id
MOVE.L (SP),D1 ;Put parameter "priority" here
MOVE.L (SP+4),D2 ;Put parameter "id" here
MOVE.L (SP+8),D3 ;Put parameter "ctl_block" here

MOVEL D1-D3, Enter_CREATE_TASK ;***NEW INSTRUCTION***

TRAP #0 ;Trap into the OS kernel

MOVE.L D0, Exit_CREATE_TASK ;***NEW INSTRUCTION***

RTS ;Return to the application program
```
or when a certain message is sent to a specific mailbox.

This new technique is available now in products offered by Hewlett-Packard. However, with some engineering resources applied, it could be implemented by anyone familiar with emulation or logic analysis. The technique consists of three parts described below. The three parts work together in the following manner: In part 1, the interface library into the OS kernel is modified such that at run time, whenever an OS service call is invoked or a task switch occurs, some encoded data is written out to a predefined data area. In part 2, the analyzer is set up to capture only the special encoded data written to this predefined data area. In part 3, the captured trace data is decoded into the high level OS service call mnemonics and task identifiers to provide a useful and easily readable listing of high level OS activity.

**PART I: INSTRUMENTING THE APPLICATION CODE**

**Section 1**

First, one must create a small data area used exclusively for the OS activity measurements. Then, for each OS service call that is to be tracked, a section of this data area is defined that is large enough to hold the incoming parameters and the outgoing return values. For example, if a service call to create a task looked like:

```
CREATE_TASK(u_int32 priority, id; u_int32 ctl_block)
```

the corresponding assembly language data section would look something like:

```
Start_of_OS_data_block
Enter CREATE_TASK DS.L 3
Exit CREATE_TASK DS.L 1
```

The three long words declared by the DS.L 3 are intended to hold the three incoming parameters of “priority”, “id”, and “ctl_block”. The one long word declared by DS.L 1 will hold the return value when this service call returns.

A section of this type is defined in the data area for each OS service call that is to be tracked. In addition, another global data section unrelated to a specific service call is defined which will allow task switch tracking. This data section consists of two data words, one for the task being exited and the other for the task being entered. For example, the data declarations might look like:

```
Enter_TASK DS.L 1
Exit_TASK DS.L 1
End_of_OS_data_block
```

This whole OS activity data area should be surrounded by a pair of labels that delineate this special OS area such as those shown in the above examples with “Start_of_OS_data_block” through “End_of_OS_data_block”. These labels will be used later.

**Section 2**

This subsection involves modifying the application code to write out the incoming parameters and outgoing return values whenever the application code executes the OS service call. The modification required to accomplish this depends on some characteristics of the OS kernel and whether source code or binary OS libraries are available from the OS vendor.

For kernels written in assembly language where the assembly language service routines are accessible to the user, tracking of OS service calls takes advantage of the interface library which allows a high level language such as “C” to call the assembly language based operating system service call. This library is a set of functions that correspond directly to each routine available from the OS. Each function in the library is responsible for taking parameters off the stack and placing these values into proper registers. A software TRAP instruction is then executed to pass control to the kernel which interrogates the registers and performs some action. Using this mechanism, an OS kernel can easily be made to work with various compilers by just modifying the interface library.

For example, a Motorola 68000 based assembly language interface to a task create kernel service routine may look like:

```
MOVE.L #CREATE_TASK,DO iSet up desired service call id
MOVE.L (SP), D1 iPut parameter “priority” here
MOVE.L (SP+4), D2 iPut parameter “id” here
MOVE.L (SP+8), D3 iPut parameter “ctl_block” here
TRAP #0 iTrap into the OS kernel
RTS ;Return to the application program
```

There is an interface routine like this one in the library for each OS service call in the kernel. To add the capability to track calls to these OS service routines, the following additional assembly instructions are needed:

```
MOVE.L #CREATE_TASK, DO iSet up desired service call id
MOVE.L (SP), D1 iPut parameter “priority” here
MOVE.L (SP+4), D2 iPut parameter “id” here
MOVE.L (SP+8), D3 iPut parameter “ctl_block” here
```

```
MOVEL D1-D3, EnterCREATE TASK ;***NEW INSTRUCTION***
TRAP #0 ;Trap into the OS kernel
```

```
MOVE.L D0, ExitCREATE TASK ;***NEW INSTRUCTION***
RTS ;Return to the application program
```

Kuzara
RTOS Debugging
From this it can be seen that whenever any OS service call is invoked at run time, the area to which the data is written identifies which OS service call has been invoked and the contents of the data specify the parameters and return value of that particular call.

If a kernel is written in a high level language or the interface library is available only in binary form, the above tracking scheme must be implemented differently. This situation requires the creation of interfaces written in the kernel's high level language that write out this same tracking data. For the previous task create service call example:

```c
CREATE_TASK(u int32 priority, id; *u int32 ctl_block)
```

a new function that included the tracking modifications would be written which would look like:

```c
u int32 Enter>Create_TASK[3]
```

```c
u int32 Exit_CREATE_TASK;
```

```c
u int32 /* Same return value as old function */
```

```c
New CREATE TASK(u int32 priority, id; *u int32 ctl_block)
```

```c
{
  Enter_CREATE_TASK = priority;
  Enter_CREATE_TASK+1 = id;
  Enter_CREATE_TASK+2 = ctl_block;
  Exit_CREATE_TASK = CREATE_TASK(u_int32 priority, id; *u int32 ctl_block);
}
```

Now, the application code would need to be changed so that all calls to CREATE_TASK are made through New CREATE_TASK. As before, each service call would have one of these new interface routines. This scheme is slightly more intrusive than the previous assembly language interface library method but provides the same level of OS service call tracking.

Section 3

This final subsection provides tracking when the operating system switches out of one task and into another. For this feature, a special hook into the operating system is needed so that whenever any task switch occurs, a user defined routine can be executed. Passed into this routine are parameters indicating which task is being exited and which is being entered. At run time when this routine gets called, the parameters are simply written to the global data area defined earlier. This “task switch callout” routine might look like:

```c
void
Task_switch_routine(u int32 exit_task_id, enter_task_id)
```

```c
{|-
| Enter_TASK = exit_task_id;
| Enter_TASK = enter_task_id;
```
From this it can be seen that whenever any OS service call is invoked at run time, the area to which the data is written identifies which OS service call has been invoked and the contents of the data specify the parameters and return value of that particular call.

If a kernel is written in a high level language or the interface library is available only in binary form, the above tracking scheme must be implemented differently. This situation requires the creation of interfaces written in the kernel's high level language that write out this same tracking data. For the previous task create service call example:

```c
CREATE_TASK(u int32 priority, id; *u int32 ctl_block)
```

a new function that included the tracking modifications would be written which would look like:

```c
c void Enter_CREATE_TASK(u_int32 priority, id; *u int32 ctl_block)
```

New CREATE TASK(u int32 priority, id; *u int32 ctl_block) {
  Enter_CREATE_TASK = priority;
  Enter_CREATE_TASK+1 = id;
  Enter_CREATE_TASK+2 = ctl block;

  Exit_CREATE_TASK = CREATE_TASK(u_int32 priority, id; *u int32 ctl_block);
}
```

Now, the application code would need to be changed so that all calls to CREATE_TASK are made through New_CREATE_TASK. As before, each service call would have one of these new interface routines. This scheme is slightly more intrusive than the previous assembly language interface library method but provides the same level of OS service call tracking.

Section 3

This final subsection provides tracking when the operating system switches out of one task and into another. For this feature, a special hook into the operating system is needed so that whenever any task switch occurs, a user defined routine can be executed. Passed into this routine are parameters indicating which task is being exited and which is being entered. At run time when this routine gets called, the parameters are simply written to the global data area defined earlier. This “task switch callout” routine might look like:

```c
c void Task_switch_routine(u_int32 exit_task_id, enter_task_id)
```

Most commercial embedded operating systems have this special hook already. If source code is available for the OS kernel, this hook can be easily added if it doesn’t already exist.

Part II: Capturing the OS tracking data

The second part of this OS tracking technology involves setting up the real-time analyzer so that it captures only encoded OS tracking data and not all low level bus activity. This is the mechanism that provides the high level OS trace data view since analysis information is captured whenever an OS service call is invoked or a task is switched but it is not captured when low level application bus cycle activity is occurring.

To get this high level view, one need only set up the analyzer to capture all writes to the address range specified by the labels “Start_of_OS_data_block” and “End_of_OS_data_block”. This measurement uses the range resource found on most emulation or logic analyzers.

By ranging on different labels in the OS data table, various views of OS activity can be captured. For example, a user may wish to follow only task switch activity without looking at the interspersed OS service calls. For this the analyzer would capture the address range between “Enter_TASK” and “Exit_TASK” only. Likewise, if the user desired to follow OS service call activity only for a specific service call such as watching all messages being sent through the SENDMESSAGE service call, the analyzer would capture only data writes to the “Enter_SENDMESSAGE” through “Exit_SENDMESSAGE” data areas.

If an analyzer has the ability to trigger on boolean conditions, more measurements can be made to help identify problems. For example, if a user wished to find out if any task ever sends either of two messages, the analyzer could be set up to trigger as follows:

```c
c store all writes to the OS data table and
stop storing when Enter_SEND_MESSAGE+4 gets written a
<message 1>
 OR Enter_SEND_MESSAGE+4 gets written a
<message 2>
```

(assuming that Enter_SEND_MESSAGE+4 is the parameter containing the message contents).

Analyzers that have the ability to trigger on a sequence have the ability to set up very specific trigger conditions to fully isolate OS related problems. For example, if a user suspects that something is wrong with their task priority scheme, they might want to determine if task “1” ever switches directly into task “3” (which they know should never occur). An analysis sequence measurement can be set up to trace for this condition as follows:
store all writes to the OS data table
and stop storing when Enter_TASK gets written a 1
followed immediately by Enter_TASK getting
written a 3

With many analyzers, valuable measurements that cross between high level OS activity
and low level source code activity can also be taken. One very common user problem is
capturing an analysis trace of shared code. Since multiple OS tasks may be using the same
code module, it is sometimes impossible to trace into a desired function at the proper time,
say only when it is called by task “I”. This is now possible with the following set up:

when Enter_TASK gets written a 1
store 1 low level bus cycles after address <desired
function>

Many combinations of the above measurements can be made depending on the power of the
real-time analyzer. As can be seen, however, these measurements can become very complex
and cumbersome to set up and use. When using an emulation system, these measurement set
ups may be made much easier by entering the analyzer set ups in system “command files” or
by making these measurements accessible through a single measurement keystroke.

Many command file systems allow parameters to be given on the same command line after
the command file name. This allows the majority of the set up to remain the same while
specific parameters are input to get the specific measurement desired. For example, to put
the latest example in a command file, the command file would be called
"trace_shared_function" and it would accept two parameters: 1) the desired task id to trigger
after and 2) the function name to trace. Invoking this measurement now becomes much
easier as in:

trace_shared_function 1 calculate_percent

A library of these command files can be created to allow easy repeated access to useful
measurements.

PART III: DISPLAYING THE CAPTURED OS DATA

Now that the analyzer has captured the OS tracking data, the data needs to be displayed in a
useful form. The captured analysis data is actually stored as a series of memory hex writes.
This, however, can be very hard to follow since translation is required to interpret, for
example, that a memory write to a specific address in the data table indicates that a particular
OS service call was invoked.

The answer to this translation problem is to write a translation program that does this data
conversion automatically and produces displays of the translated OS specific data results.
This program would convert any write to service call data areas to their equivalent OS
service call mnemonic and to their OS specific parameter and return values. For example,
writes to Enter_CREATE_TASK through Enter_CREATE_TASK+8 would be converted and
a trace data message similar to the following may be displayed:

Entering Service call: CREATE_TASK
parameter 1 <task id>
parameter 2 <priority>
parameter 3 <control block>

When the OS returned from this service call and the return value was written to
Exit_CREATE_TASK, a different message would be displayed:

Returning from Service call: CREATE_TASK
return value <return value>

Likewise, the program would translate task switch activity in a similar way. Writes to
Enter_TASK and Exit_TASK would convert into the following messages:

Exiting Task: <exit task id>
Entering Task: <enter task id>

A longer example trace might therefore look like the following:

<table>
<thead>
<tr>
<th>State</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Entering Task: 5</td>
</tr>
</tbody>
</table>
| 002   | Entering Service call:SEND MESSAGE
|       | mailbox id 1234        |
|       | message 55             |
| 003   | Returning from Service call:SEND MESSAGE
|       | return value 0         |
| 004   | Entering Service call:SEND_EVENT
|       | event number 4         |
| 005   | Exiting Task: 5       |
| 006   | Entering Task: 2       |
| 007   | Returning from Service call:RECEIVE_EVENT
|       | return value 0         |

In reviewing this OS trace, we can easily interpret the results and relate them to our actual
system. First, we see that our task “5” was entered. It then invoked OS service call
"SEND_MESSAGE" to send a message “5” to mailbox “1234”. It then returned with a
successful return value. Next we see that the service call “SEND_EVENT” was called to
send event “4”. At this point the trace shows us that the current task “5” now blocks and the
OS switches over to task “2”. The trace then clearly shows us that task “2” successfully
returns from a previous call to “RECEIVE_EVENT”. We now know that we must have
been blocked at this service call waiting for the previous event “4” to be sent.
store all writes to the OS data table
and stop storing when Enter_TASK gets written a 1
followed immediately by Enter_TASK getting
written a 3

With many analyzers, valuable measurements that cross between high level OS activity and
low level source code activity can also be taken. One very common user problem is
capturing an analysis trace of shared code. Since multiple OS tasks may be using the same
code module, it is sometimes impossible to trace into a desired function at the proper time,
say only when it is called by task “I”. This is now possible with the following set up:

\[ \text{when Enter_TASK gets written a 1} \]
\[ \text{store 1 low level bus cycles after address <desired function>} \]

Many combinations of the above measurements can be made depending on the power of the
real-time analyzer. As can be seen, however, these measurements can become very complex
and cumbersome to set up and use. When using an emulation system, these measurement set
ups may be made much easier by entering the analyzer set ups in system “command files” or
by making these measurements accessible through a single measurement keystroke.

Many command file systems allow parameters to be given on the same command line after
the command file name. This allows the majority of the set up to remain the same while
specific parameters are input to get the specific measurement desired. For example, to put
the latest example in a command file, the command file would be called
“trace_shared_function” and it would accept two parameters: 1) the desired task id to trigger
after and 2) the function name to trace. Invoking this measurement now becomes much
easier as in:

\[ \text{trace_shared_function 1 calculate_percent} \]

A library of these command files can be created to allow easy repeated access to useful
measurements.

PART III: DISPLAYING THE CAPTURED OS DATA

Now that the analyzer has captured the OS tracking data, the data needs to be displayed in a
useful form. The captured analysis data is actually stored as a series of memory hex writes.
This, however, can be very hard to follow since translation is required to interpret, for
example, what a memory write to a specific address in the data table indicates that a particular
OS service call was invoked.

The answer to this translation problem is to write a translation program that does this data
conversion automatically and produces displays of the translated OS specific data results.
This program would convert any write to service call data areas to their equivalent OS
service call mnemonic and to their OS specific parameter and return values. For example,
writes to Enter_CREATE_TASK through Enter_CREATE_TASK+8 would be converted and
a trace data message similar to the following may be displayed:

Entering Service call: CREATE_TASK
parameter 1 <task id>
parameter 2 <priority>
parameter 3 <control block>

When the OS returned from this service call and the return value was written to
Exit_CREATE_TASK, a different message would be displayed:

Returning from Service call: CREATE_TASK
return value <return value>

Likewise, the program would translate task switch activity in a similar way. Writes to
Enter_TASK and Exit_TASK would convert into the following messages:

Exiting Task: <exit task id>
Entering Task: <enter task id>

A longer example trace might therefore look like the following:

\[
\begin{array}{ll}
001 & \text{Entering Task: 5} \\
002 & \text{Entering Service call: SEND_MESSAGE} \\
& \quad \text{mailbox id 1234} \\
& \quad \text{message 55} \\
003 & \text{Returning from Service call: SEND_MESSAGE} \\
& \quad \text{return value 0} \\
004 & \text{Entering Service call: RECEIVE_EVENT} \\
& \quad \text{event number 4} \\
005 & \text{Exiting Task: 5} \\
006 & \text{Entering Task: 2} \\
007 & \text{Returning from Service call: RECEIVE_EVENT} \\
& \quad \text{return value 0}
\end{array}
\]

In reviewing this OS trace, we can easily interpret the results and relate them to our actual
system. First, we see that our task “5” was entered. It then invoked OS service call
“SEND_MESSAGE” to send a message “55” to mailbox “1234”. It then returned with a
successful return value. Next we see that the service call “SEND_EVENT” was called to
send event “4”. At this point the trace shows us that the current task “5” now blocks and the
OS switches over to task “2”. The trace then clearly shows us that task “2” successfully
returns from a previous call to “RECEIVE_EVENT”. We now know that we must have
been blocked at this service call waiting for the previous event “4” to be sent.
There are two potential mechanisms available to perform this automatic data conversion. First and most useful is performing the conversion in the analyzer software itself so that the direct output of the analyzer screen is the translated OS trace. Many emulation and logic analyzer systems have the ability of allowing users to install their own conversion programs. Hewlett-Packard’s real-time operating system debug products provide this conversion program as an integrated piece of the measurement product software.

If not available, however, analyzer data can always be converted to ASCII and transferred to a computer where it can then be translated in a post-processing manner. In either case, the conversion program is relatively straightforward, consisting mainly of a large CASE statement of the following form:

```c
switch (current address)
{
    case (Enter_SEND_MESSAGE)
        print "Entering OS Service call: SEND MESSAGE"
        print "  mailbox id: ", current data value
        print "  message: ", next data value
    case (Exit_SEND_MESSAGE)
        print "Exiting OS Service call: SEND MESSAGE"
        print "  return value: ", current data value
    case (Enter_CHANGE_PRIORITY)
        print "Entering OS Service call: CHANGE_PRIORITY"
        print "  task id: ", current data value
        print "  priority: ", next data value
    case (Enter_TASK)
        print "Entering Task: ", current data value
    case (Exit_TASK)
        print "Exiting Task: ", current data value
}
```

CONCLUSION

As described here, the process of instrumenting operating system application code, setting up powerful analyzer measurements to capture OS specific data and displaying the captured OS data in an easy-to-understand, OS specific format provides the capability for powerful analysis measurements tailored especially for users of embedded real-time operating systems. Without the methodology described here, the user is powerless to track the flow of their operating system code in real-time or to analyze the communication between their OS tasks in real-time. The methodology described here provides real-time measurements that, together with OS specific debugger monitor technology, give OS users the power to debug their OS application in two dimensions: non-real-time and real-time.