

Debugging and Optimising OSEK Kernels in Real-Time

An OSEK kernel provides a framework for building a real-time application as a series of tasks and for allocating resources among, and communicating between, those tasks. Thus, the operation of the kernel has a profound effect on the efficiency and effectiveness of the application. And in turn, debugging, testing, Verification and Validation of the application requires accurate measurement of the behaviour and performance of the kernel. This article describes a "System-Level Debugger" that displays the kernel and task activity based on a defined interface between the debugger software and the kernel, together with the appropriate hardware instruments for real-time performance measurement and tracing.

INTRODUCTION

Embedded systems require the ability to multi-task or perform several tasks 'at once'. In addition certain of these tasks must be able to respond and operate within a defined time-period. These requirements have led to the development of Real-Time Operating Systems (RTOSs). An RTOS provides a framework (or API) that allows an application to be built as a series of tasks, additionally, the RTOS allows tasks to share and allocate resources and to intercommunicate. The OSEK/VDX operating system is a particular RTOS implementation that is gaining wide acceptance in the automotive industry (see Ref. 5). As the application is built on top of the OSEK kernel, the operation of the kernel has a profound effect on the efficiency and effectiveness of the application. And in turn, debugging, testing, Verification and Validation of the application requires accurate measurement of the behaviour and performance of the kernel. In order to allow successful deployment of systems based on the OSEK kernel it is vital that the tools used to debug, test and analyse the system are OSEK kernel-aware, that is, they present the user with 'high-level' OSEK kernel of exactly what their application is doing and when it is doing it.

Ashling Microsystems Ltd., an Irish company with world-wide headquarters at the National Technological Park in Limerick, designs and manufactures Microprocessor Development Systems, Source Level Debuggers, Development Software, and Software Quality Assurance tools for the international market. This article describes how Ashling have adopted their real-time debugging tools to provide full support for debug, test and analysis of automotive software systems that use an OSEK kernel.

RTOS INTERFACE TO THE SOURCE-LEVEL DEBUGGER

In the past, establishing communication between a debugger and an RTOS required a unique implementation for each debugger and each RTOS; thus, substantial re-development work was required for each new RTOS and each new debugger. To resolve this problem, Ashling has adopted a published standard for the API (Application Programming Interface) by which the debugger software communicates to the RTOS.

The Ashling Source-level Debugger (PathFinder) supports the Kernel Debug Interface (KDI) standard (see Ref. 1 and 4). This allows the RTOS vendor to write a Kernel Support Module (KSM, typically implemented as a Dynamic Link Library or DLL on Windows based hosts) which uses the KDI API to provide 'RTOS aware' debug features in the source-level debugger. The KDI API provides the following features:

- **Target access:** The caller (KSM) may use this API to access the memory and registers of the target processor.
- **Symbolic access:** The KSM may use this API to evaluate expressions. For example, if the current program contains a variable called NumTasks this API can be called to evaluate NumTasks.
- **Windows:** KDI allows creation of Windows and writing to those windows. These windows appear in the debugger as normal child windows.
- **Menus:** KDI allows additional RTOS specific menus to be created in the debugger.
- **Breakpoints:** KDI provides a mechanism for setting 'task aware' breakpoints.

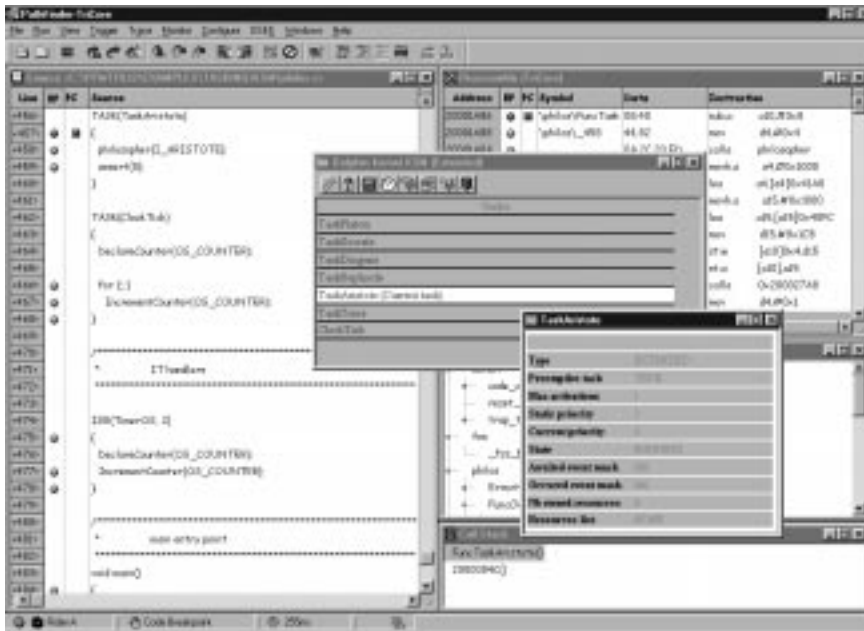


Figure 1. Ashling's "PathFinder" Source-Level Debugger displays the status of the OSEK kernel (from TECSI) by communicating using the KDI Standard

The user can load the RTOS vendor's KSM via the RTOS Debug menu in PathFinder. The KSM will then create specific RTOS windows within the debugger showing the current RTOS status, for example, the number of tasks, their current status etc. This is illustrated in Figure 1.

This mechanism provides a good solution for 'static' based debugging, however, the method has limitations, notably, the application must be halted to allow the RTOS windows to update. To debug real-time execution flow problems at a task level we need a mechanism that allows us to see what our application is doing without halting or intruding on the application, we need Task Aware Performance Analysis.

RTOS INTERFACE TO THE PERFORMANCE ANALYSER

Ashling's STARS (Software Test, Analysis and Reporting System) Performance Analyser is a real-time, non-intrusive measurement system that can perform the following types of analysis:

- **Accumulated time:** The total time spent in a function or range since the start of measurement.
- **Accumulated count:** The number of times a function or range has been entered and exited since the start of measurement.
- **Average time:** (Accumulated time/Accumulated count).
- **Maximum and minimum times:** The maximum and minimum execution times for the function or range.
- **Function trace:** A high-level trace showing the program flow as function entry and exits.

Time-based analysis may be either Inclusive or Exclusive. Inclusive means that the measurement includes the time spent in the function and all

functions called by that function. Exclusive means that the measurement only includes the time spent within that function; time spent in functions called by that function is not included in that function's displayed execution-time.

ANALYSING OSEK/VDX PERFORMANCE

The Performance Analyser works by analysing every executed instruction at a hardware bus-level. As function or range entry or exit points are executed, the analyser stores their address and the time of execution. This

data is continuously acquired and analysed as the program executes. The Analysis results are transferred to the PathFinder debugger for on-the-fly display without halting emulation.

In a RTOS environment, it is important that the Performance Analyser is able to perform analysis at a task level; that is, to measure:

- **Accumulated time:** The total time spent in a task since the start of measurement. For Inclusive mode this includes the time spent in all functions called by the task. For Exclusive mode, it only includes time spent in the task, excluding any functions called.
- **Accumulated count:** The number of times a particular task has been executed. The count is incremented each time the task is switched to by the scheduler.
- **Average time in a task:** (Accumulated time/Accumulated count).
- **Maximum and minimum times.** The maximum and minimum execution times for a task.
- **Task trace:** A high-level trace showing the program flow between tasks and functions.

In order to perform the above analysis it is vital that the Performance Analyser knows when a task is entered and exited. The traditional method of 'watching' the start and end addresses of a function will not work, because once a task is started it can be exited and re-entered at any point.

The C/C++ language has no specific keywords for defining tasks; in an OSEK/VDX RTOS, tasks are normal C functions. These are normally configured as tasks by an entry in the OSEK Implementation Language (OIL) configuration file. The OIL file allows the end user to describe their OSEK application, this file is normally processed to create header files (.H) by

a separate OSEK system configuration tool provided by the OSEK vendor. These header files contain the OSEK related definitions and are included in the application source (C) files.

Most tasks tend to be a continuous loop that perform some action and then sleep until awoken, or until a specified event has happened. For example, the following pseudo-code illustrates a common task structure:

```
AnExampleTask()
{
    for (;;)
    {
        ...
        do something;
        ...
        ...
        wait for something to happen;
    }
}
```

Therefore, unlike traditional functions, most tasks will never execute their end or exit address. To allow the Performance Analyser to know when a task is entered or exited, OSEK provides PreTaskHook and PostTaskHook hooks. These hook functions are called by the OSEK scheduler before and after a task executes. The hook routines are called with the current Task ID as a parameter, so that there are only two hook functions regardless of the number of tasks. With some additional code in these hook routines we can cause a unique code address to be executed for each task's entry and exit. The Performance Analyser can then monitor these unique code addresses and determine when individual tasks are entered and exited. The following code illustrates how this can be done:

```
void PreTaskHook(TaskType TaskID)
{
    switch (TaskID)
    {
        case Task0:
TASK0Entry:
            break;
            // repeat for all tasks
        default:
            break;
    }
}

void PostTaskHook(TaskType TaskID)
{
    switch (TaskID)
    {
        case Task0:
```

```
TASK0Exit:
            break;
            // repeat for all Tasks
        default:
            break;
    }
}
```

Therefore, whenever Task0 is entered the label TASK0Entry is executed; when Task0 is exited the label TASK0Exit is executed. As can be seen from the above code-fragment, the intrusion on the context switch time is minimal, and the overhead (in terms of additional code and slower system execution) is very modest. Within the Pathfinder debugger, the user tells the Performance Analyser that Task0 has an entry point at TASK0Entry and an exit point at TASK0Exit.

The hook code can be hand written or generated by the OSEK system configuration tool and placed in one of its output source files (these are compiled and linked in with the rest of the user's project). The configuration tool knows the number of tasks and thus has all the knowledge required to generate the hook code. Alternatively, a separate utility can be used to process the OIL file and generate the hook code.

Part of the work described in this article was undertaken within the ESPIRIT Projects 23177 "STEEPCAM" and 24026 "DOLPHIN". ■

REFERENCES

- [1] Tasking Kernel Debug Interface (KDI) specification. Revision 1.1. July 1997
- [2] ESPIRIT Project 23177 "STEEPCAM"
<http://www.ashling.com>
- [3] ESPIRIT Project 24026 "DOLPHIN"
http://www.omimo.be/system/templates/OMIProjects_Detail.cfm?ID=87&Project=24026
- [4] ESPIRIT Project 7325 "OMI/DEBUG"
http://www.omimo.be/system/templates/OMIProjects_Detail.cfm?ID=20&Project=7325
- [5] OSEK/VDX Operating System Specification 2.0 revision 1

Hugh O'Keeffe is the R&D Manager with Ashling Microsystems Ltd. Hugh has a B. Eng. in Electronic Engineering from the University of Limerick. Hugh has extensive experience in both embedded systems design and the development of debug tools for embedded system designers. Hugh's team recently completed work on the first phase of the Ashling TriCore Development Toolset, this resulted in the worlds first In-Circuit Emulator and Source-level C/C++ debugger for the Infineon TriCore Architecture. His current interests are on-chip debug techniques to allow non-intrusive Performance Analysis on high-end microprocessor architectures.