Symbian OS Internals/14. Kernel-Side Debug

- Symbian OS Internals Table of Contents

by Morgan Henry

A computer lets you make more mistakes faster than any invention in human history-with the possible exceptions of handguns and tequila.
Mitch Ratcliffe

This chapter describes how the Symbian kernel architecture supports a range of debuggers and other useful development tools. It describes the types of tools available for development of kernel and application level software, and how the kernel implements and interacts with them. The reader should be familiar with EKA2's device driver model, memory model and scheduler.

Overview

The emulator provides the primary development environment for Symbian OS (see Figure 14.1). For most application and middleware development, the behavior of the emulator is sufficiently close to the behavior of Symbian OS on retail handsets to allow us to develop the majority of our software inside this environment.

As I described in Chapter 1, *Introducing EKA2*, we have made several improvements to the design of the emulator in EKA2. It now shares a significant amount of code with the kernel, nanokernel and scheduler. As a result, the emulator is a much more faithful representation of the behavior of the kernel on a target phone. This has made the EKA2 emulator suitable for the development of some types of kernel-side software, even including the development of device drivers that are not sensitive to the underlying hardware.

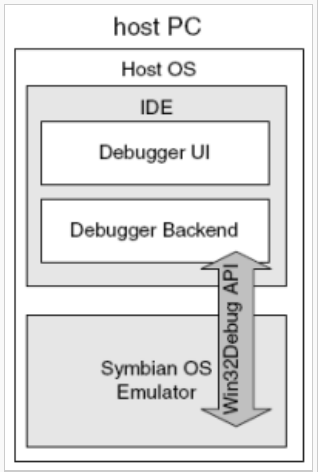

Figure 14.1 *Emulator debug architecture*

However, even with these improvements, there are occasions when application and kernel developers will need to move out of the emulated environment, and into a hardware environment. This is necessary for:

- Development of hardware-specific device drivers
- Diagnosis of defects that stubbornly appear only on a target mobile phone
- Software that is timing sensitive
- Software with dependencies on exotic peripherals only found on mobile phones.

Where the emulator can no longer assist you with your debugging tasks, EKA2 provides features to support debug on target hardware.

EKA2 is architected to support remote debuggers. The design aims were to provide direct kernel support for as much of the embedded tools market as possible, whilst remaining vendor independent. The new interface builds on experience with EKA1 phones, hopefully easing the integration task faced by those providing new tools.

The APIs described in this chapter provide operating system support for a number of debug tools:

- Emulator debugger for hardware agnostic application and middleware development
- Run-mode, target resident debuggers primarily focused on debugging applications, middleware, real-time and hardware-sensitive software.

- Hardware assisted stop-mode debuggers, primarily focused on debugging the kernel, device drivers and other kernel-mode software
- Post-mortem debuggers
- Trace output
- Profilers.

But first, let's look at how EKA2's architecture supports debugging.

Architecture

Debuggers need much more information, and much more control over the kernel than any other piece of software. This section describes how the Symbian OS kernel and each of the tools that I just listed interact with each other.

Emulator debuggers

In the emulator world, both the debugger IDE and Symbian OS are citizens of the host PC operating system. As I described in Chapter 1, *Introducing EKA2*, each Symbian OS thread maps onto a native Win32 thread - this allows the IDE to treat Symbian OS just like any other Win32 application. To observe and control Symbian OS threads, the debugger attaches to them by using the native Win32 debug APIs. While attached, the debugger will be notified of any events that affect the thread, such as breakpoints or termination. Both Symbian OS kernel and application threads can be debugged in this way. While it is attached, the emulator debugger has complete control over the execution of Symbian OS threads.

This method of debugging is extremely powerful, since existing state-of-the-art Windows development tools can be used to debug Symbian OS. This makes it an attractive development environment for much hardware-agnostic software.

Run-mode debuggers

The type of target debugger most familiar to application and middleware developers is the remote *run-mode* debugger (Figure 14.2). The debugger UI runs on a host PC, and controls threads running on a target operating system through a proxy running remotely on the target mobile phone. Debuggers with this *remote debug* capability are common in the embedded technology industry.

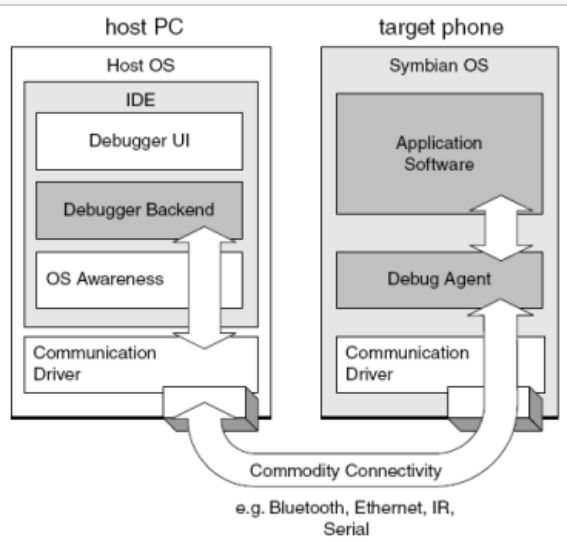

Figure 14.2 *Run-mode debug architecture*

In the Symbian environment, the host PC runs a debugger, which connects to the target phone running Symbian OS. The connection is over any common communications channel supported by Symbian OS, such as serial, Bluetooth, IR, USB or Ethernet.

The host debugger talks to a remote debug agent, running on Symbian OS, which performs actions on its behalf. It is the debug agent that directly observes and manipulates the application threads being debugged by the host. The debug agent will also report any information it believes is relevant for the debugging session back to the host. To reduce the number of information and event messages sent over the communications link, the host debugger will typically elect to observe and control only a few threads. The debug agent will report only those events and data that affect these attached threads.

The debugger remains mostly transparent to the rest of the system. This has the benefit that protocol stacks, timing-dependent software, and any software interacting with the real world environment will continue to function as normal during a debug session. This makes this type of debugger attractive to third-party application and middleware developers. However, these debuggers are not generally suitable for kernel or device driver development. In this architecture, the debug agent and the communication

channel are directly, and indirectly, clients of the Symbian OS kernel. Any attempt by the debugger to suspend parts of the kernel would result in a hung system, making the agent and communication channel inoperable.

The remainder of this section examines the components of the run-mode debug architecture in more detail. Figure 14.3 shows how the debugger communicates with its remote debug agent, and how the agent interacts with Symbian OS.

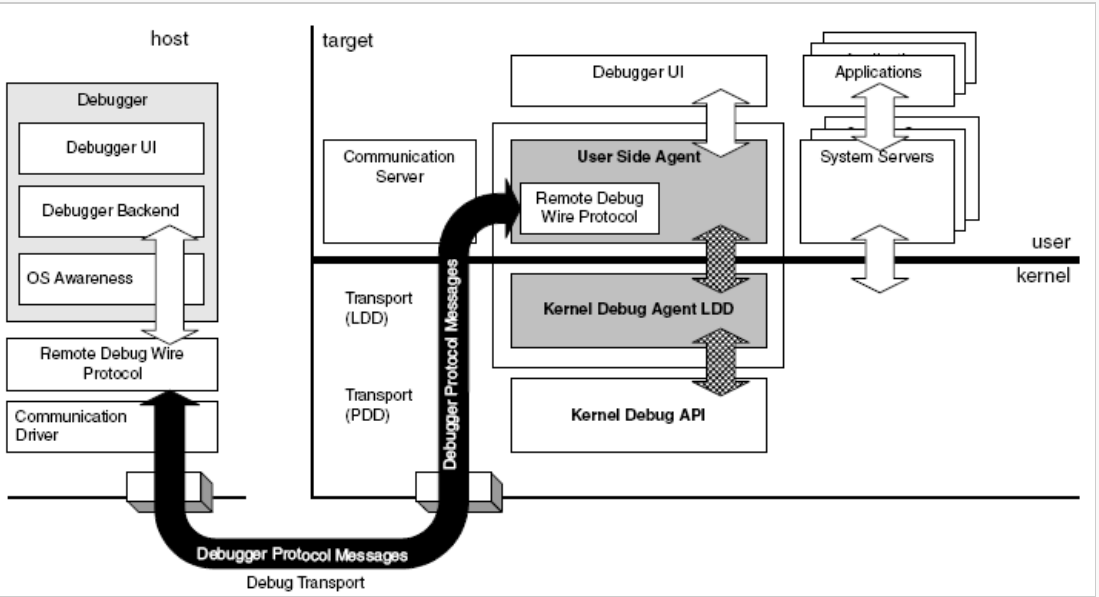

Figure 14.3 *Host debugger and debug agent on target platform*

Remote debug wire protocol

The backbone of this debugger environment is the remote debug wire protocol. This carries debug messages between the host debugger and the target, over the physical connection. The flow of these high-level protocol messages is shown in black on the diagram. The debugger communicates with the target to acquire information about threads and processes running on the target. It typically asks the target to perform thread or process-relative memory reads and writes, control the execution flow of threads, and set/remove breakpoints. The target will also notify the debugger of interesting events occurring on the mobile phone.

Debug agent

An essential component of this communication is the debug agent running on the target. This component translates the debugger's messages in the wire protocol into actions and requests for the Symbian OS kernel. (These are shown with cross-hatching.) The agent is a privileged client of the operating system, and encapsulates a significant amount of the *OS awareness* on behalf of the host debugger.

The debug agent comes in two parts. The kernel-side agent is tightly bound into the kernel via the kernel debugger API to give it the level of control and information it requires. The API (covered in Section 14.3) allows the agent to capture system events, suspend and resume threads, get and set thread context, shadow ROM, read and write to non-current processes, and discover information about code segments, libraries, and other kernel objects. The user-side agent implements the wire protocol and is responsible for setting up the communication channel. It can do this by talking to the comms server or by opening a channel to the communications driver directly. It is also a client of the file server for the upload and download of files into the file system. On phones that enforce platform security, it will also need to negotiate with the software install components to install uploaded executables. The debug agent may also use other APIs and servers.

The two parts communicate over the standard (RBusLogicalChannel) device driver interface - see Chapter 12, *Device Drivers and Extensions*, for more on this. Symbian does not define a common API here, because the functionality provided by the agent is largely dependent on the tool or wire protocol in use. (EKA1 does provide a debugger interface (RDebug), however, this proved not to be widely adopted, and is only really suitable for the GDB monitor, *GDBSTUB*.)

There are a number of debugger wire protocols available. At the time of writing, the following implementations exist for Symbian OS:

- GDBSTUB implements the open GNU remote debug wire protocol on EKA1
- MetroTrk implements the Metrowerks proprietary protocol on EKA1
- MetroTrk implements the Metrowerks proprietary protocol on EKA2.

OS awareness

On the host side, the debugger's OS awareness module interprets any operating system specific information for the rest of the

debugger. This module is the host-side partner of the debug agent for Symbian OS - it encapsulates any methods and implementations that the debugger needs to communicate effectively with the target platform and that are not part of the core debugger. This could include items such as:

- Establishing a connection to the target
- Discovering which processor core is being debugged
- Discovering which version of the OS is available
- Defining which OS specific objects should be displayed as part of the OS object visualization.

The OS awareness module is an intrinsic part of the debugger-OS integration package and is usually supplied with the IDE.

Responsibilities

In this architecture the agent has the following responsibilities:

- Implementation of high-level wire protocol. It must provide an implementation of the protocol that matches the host debugger in use. For example, GDB uses the GNU remote debug wire protocol
- Configuring the debug connection. The agent must open and configure the communication channel to enable the connection to the host. For example, it should configure the serial port, or open a TCP/IP connection. For convenience, it may also provide a UI
- Uploading files and executables onto the target. Executables are built on the host PC and must be uploaded to the mobile phone for execution. Supplementary data files also need to be transferred. On a secure build of Symbian OS, the debug agent must use software install components to install the executable. The debugger may choose to use commercial PC connectivity software
- Implementation of CPU instruction stepping, and instruction decode. This is a fundamental capability of any debugger. The agent can implement instruction stepping of a suspended thread in a couple of ways: firstly, through software simulation of the instruction at the program counter, or secondly, by placing a temporary breakpoint at the next program-counter target address and resuming the thread. The agent must identify potential branches, which requires an instruction decode
- Capturing user-thread panics, and exceptions. Knowing that an application has died is essential for a developer. The debug agent registers with the kernel to receive notification when a thread panics, causes an exception, or otherwise terminates. When notified, the debugger can open the thread for diagnosis and relay the information to the developer
- Implementation of JIT debug triggers. *Just In Time* debugging catches a thread just *before* it executes a panic or exception routine. Capturing a thread early, before it is terminated, allows the developer to more closely inspect what went wrong, before the kernel removes the thread. In some cases, the developer can modify context, program counter, and variables to recover the thread
- Implementation of software breakpoint handler. Software breakpoints are implemented as undefined instructions. The agent must register to capture undefined exception events, and then handle any that are actually breakpoints. If the breakpoint is intended to be thread-specific, the handler must check the ID of a triggered thread against the breakpoint's intended target - if it does not match, then the thread should be resumed. If the breakpoint is placed in shared code, then there are further complications - the agent must implement an algorithm for efficiently handling multiple false triggers from untargeted threads. The agent must also be able to resume a thread that was incorrectly triggered without missing further valid triggers - that is, it must be able to execute the instruction under the breakpoint for the current thread without removing the breakpoint
- Breakpoint housekeeping. Add and remove breakpoints as libraries and processes are loaded and unloaded. For the developer's convenience, the debugger often makes it possible to set breakpoints in code that is not yet loaded. The debug agent defers the application of the breakpoint until the OS loads the target code. The agent can do this by registering to be notified of library and process load events. The agent is also responsible for shadowing ROM code to allow breakpoint instructions to be written
- Communicating addresses of code and process data to host. For executable to source-code association to work effectively, the host must relocate the executable's symbolic debug information to match the memory address of the corresponding code and data on the target. The debug agent must discover where the kernel has loaded each executable section and relay this to the debugger.

14.2.3 Hardware-assisted debuggers

Many debuggers in the embedded world provide support for ASICs equipped with JTAG ICE hardware.

The Joint Test Action Group defined and established the IEEE 1149.1 standard for boundary-scan hardware test and diagnostics. This standard is commonly referred to as JTAG. The interface has since been adopted as a popular access port for CPU control to support software debugging activities using the embedded In-Circuit Emulator (ICE) common on ARM hardware.

The ICE allows the target processor to be halted and subsequently controlled by the host at the instruction level. This provides features such as instruction level step, hardware breakpoint support, remote memory reads and writes, and CPU register

manipulation. The JTAG port can also provide an additional real-time data channel for a debugger protocol (Figure 14.4).

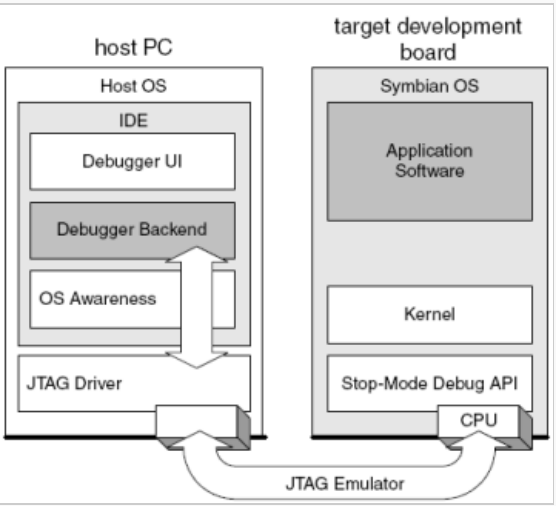

Figure 14.4 *Stop-mode debug architecture*

The host side runs a debugger with the ability to drive a connection to a JTAG emulator, such as ARM's RealView ICE, Lauterbach's ICD, and so on.

A hardware-assisted debugger will work adequately with Symbian OS with almost no additional support, just as it would with many other `OSes`. The debugger can immediately provide raw CPU/instruction level debugging and an unvarnished view of the current memory map.

This low-level view of the target can be improved when the debugger implements *OS aware* features. The host provides the necessary Symbian OS intelligence to interpret the simple memory and register read/write semantics of JTAG as high-level operating-system events, messages and objects. To assist the host in this task, the stop-mode debug API on the target provides metadata describing the layout of key kernel objects. Using this metadata, the host debugger can navigate the kernel's data structures to determine the current state of the kernel and all running applications.

The host then interprets this information and presents it to the developer in a meaningful manner to provide the following high-level debug functionality:

- Thread-relative memory read and writes
- Thread-relative breakpoints
- Multi-process and memory model awareness
- Kernel object display.

Using this approach, some tools vendors can support integrated stop-mode kernel and stop-mode application debugging.

The ability to halt the processor makes a hardware-assisted debugger an essential tool for debugging kernels, device drivers, and any other kernel-mode software that requires a significant amount of hardware *bit-twiddling*. However, while the debugger holds the CPU, it has limited usefulness for debugging real-time software and live communication stacks. Once the CPU has been halted, the real-time protocols that interact with the outside world will invariably fall over.

Figure 14.5 shows the interactions between the host debugger, the stop-mode debug API, and the kernel in more detail.

While the host debugger has control of the target CPU, the target is not running any kernel or application code; the CPU is frozen. Because of this, and in contrast to run-mode debuggers, there is little opportunity for a target-side debug agent to run, and no high-level wire protocol. The interface between the host debugger and target has only *simple* or *flat* register and memory read/write semantics (shown in black). Compared to the run-mode architecture, the host side debugger must encapsulate a far greater amount of knowledge about how to interact with Symbian OS. The host must understand:

- Scheduling and kernel locking strategy
- Per-process memory mapping
- How the kernel lays out its data structures.

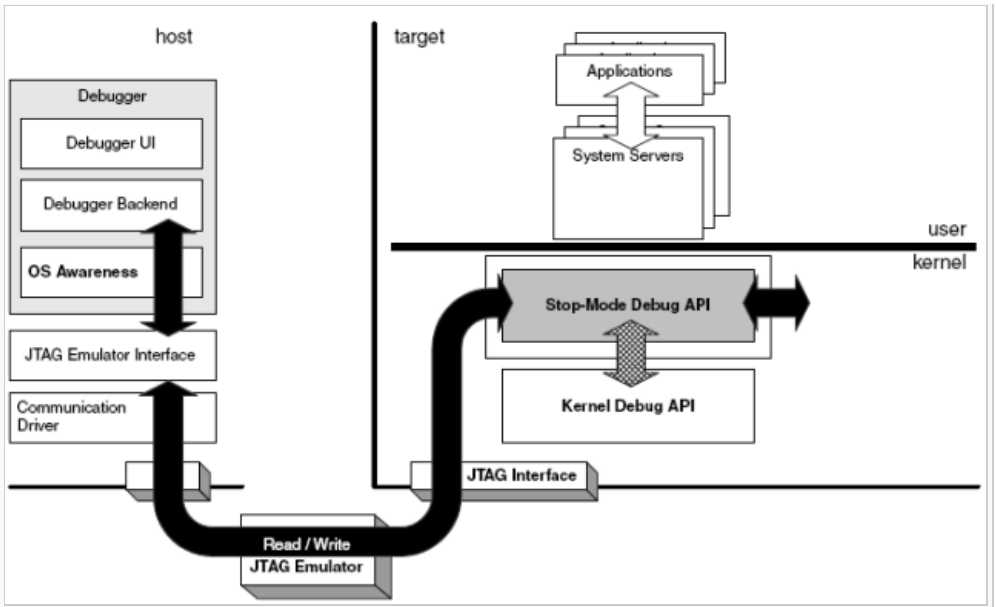

Figure 14.5 *Stop-mode debugger architecture*

Most modern stop-mode development tools have a generic OS abstraction and can support this host-side requirement.

The stop-mode debug API is responsible for ensuring all the information the debugger requires is available and valid. It presents metadata encapsulating the layout of the kernel's objects (such as processes, threads and memory chunks). The kernel continually updates the content while the target is running (shown with cross-hatching). This ensures that the metadata is consistent and valid whenever the data is required.

Hardware-assisted debuggers make effective post-mortem analysis tools. Even after a full system crash, much of the kernel data is intact, and can be inspected through the stop-mode debug API. Section 14.5 shows in detail how the Symbian OS debugger architecture supports hardware-assisted debuggers.

Post-mortem analysis tools

Under Symbian OS, we use post-mortem tools to analyze the cause of crashes and hangs after software has failed. The analysis can apply to specific executables, or to whole-system crashes. The tool is dormant on the target, waiting for a trigger from the operating system indicating abnormal termination. In response to this trigger, the tool will gather information from the kernel and system servers about the crash to present to the debugger. Some tools (for example, D_EXC and the crash logger) save a human-readable context dump to a file that can be analyzed off the phone. Others (for example, the crash debugger) are interactive, and allow the developer to interrogate the host over a communications link.

The debug API provides triggers and hooks for various system events, on which to hang the post-mortem analysis tools. Code can be installed and run on events such as: hardware exceptions, thread death, kernel death and trace output.

For more details, see the documentation for `TEventCode` in the Symbian Developer Library's C++ component reference section.

Usage information for the previously mentioned crash debugger post-mortem tool can be found in the Symbian Developer Library Device Driver Guide for EKA2 versions of Symbian OS.

Figure 14.6 shows the details of the post-mortem tool on the target. You can see that this architecture is similar to the run-mode architecture, with the host PC side omitted. In fact, the set of kernel interfaces used by both tool-types are largely the same. In the post-mortem case, the kernel-side agent uses the kernel debug API to register with kernel events associated with thread termination. The user-side agent is responsible for outputting the information over a communications channel or to disk.

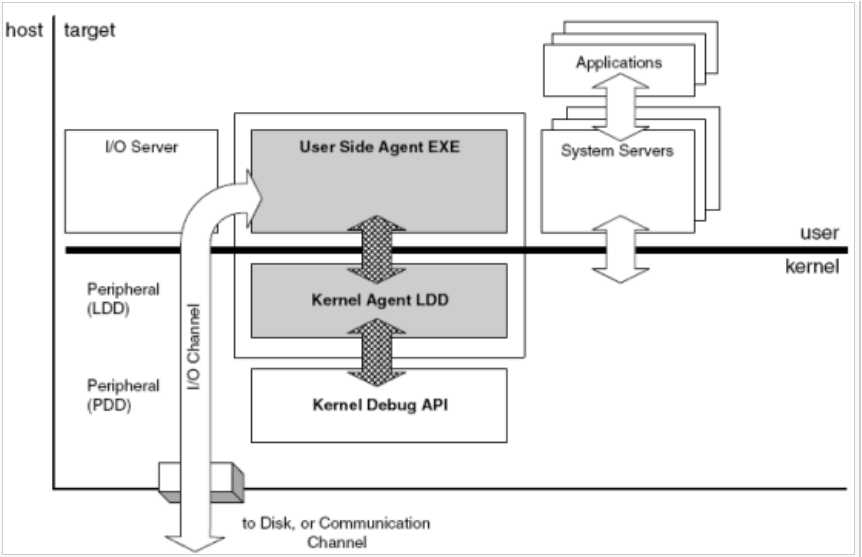

Figure 14.6 *Target post-mortem tools architecture*

A good example of this architecture is the D_EXC tool. This implements a minimal kernel debug agent (MINKDA) and a user-side agent (D_EXC). Source code for these tools is available at these locations:

```
\base\e32utils\d_exc\d_exc.cpp
\base\e32utils\d_exc\minkda.cpp
```

Analyzing post-mortem dump

To save ROM, the Symbian OS build tools strip symbolic debug information from all executables before they reach the mobile phone. This means that the post-mortem tools on the target have very little contextual information to annotate the crash dump. This can be a particular problem when attempting to read stack traces. To solve this, Symbian provides tools for the host that match up addresses found in the post-mortem dump and the symbolic debug information on the host.

MAKSYM is a command-line tool that cross-references the log file generated when building a ROM image with the verbose linker output that contains the executable symbols. The output is a text file that lists the address of every global and exported function in the ROM. This file can be used to interpret addresses found in post-mortem dumps. MAKSYM symbolic output is essential for diagnosing problems when using the crash logger or crash debugger. Similarly, two other tools - PRINTSYM and PRINTSTK- use the MAKSYM symbolic output to interpret D_EXC output. More information on using these tools can be found in the Symbian Developer Library.

The kernel debug interface

Debug tools will use many of the general purpose APIs provided by the kernel. In addition, we provide the kernel debug interface to specifically support the remaining requirements of all these tools. In this section, I will describe the features of the kernel debug interface, namely:

- Kernel event notification
- Thread context API
- Code and data section information
- Shadowing ROM pages.

Other features typically used by debug tools are:

- Kernel resource tracking (kernel object API)
- Thread-relative memory read and write.

Kernel objects and the kernel object containers are covered in Section 5.1, and I describe thread-relative memory access for the moving and multiple memory models in Chapter 7, *Memory Models.*

The kernel debug functionality is optional, since you will not always want to allow intrusive debug tools onto production mobile phones. The _DEBUGGER_SUPPORT__ macro enables the kernel features required specifically for a debugger implementation. You'll find this defined (or not) in the variant MMH file. (The project definition files for the variant are typically named \base\<variant>\variant.mmh. See \base\lubbock\variant.mmh for an example.) You can check at run-time if your build of the kernel includes these features by calling.

```
TBool DKernelEventHandler::DebugSupportEnabled();
```

By default, software and ROM images supplied on the Symbian OS DevKits are built with debug support enabled. However, phone manufacturers may choose to switch it off.

Kernel event notification

The kernel exposes an API to allow a debug agent to track kernel activity, and if necessary respond to it. We chose the events exposed through this API to allow the most important features of a debug agent to be implemented. Of course, other kernel-side software can use them too.

The following events can be captured:

- User-side software exceptions, occurring when the user thread calls User::Panic or `RThread::RaiseException()`. Typically on ARM processors, the latter will be caused by an integer divide by zero exception. Debuggers will report the exception or panic details to the developer for diagnosis. Post-mortem tools can also trigger on these events to save context for later analysis
- Hardware exceptions: for example, code aborts, data aborts, invalid instruction, memory access violation and floating-point co-processor divide by zero. These are obviously very important in aiding the developer with her diagnosis. It is also often possible for the developer to use the debugger to alter thread context and variables to retry the instruction. Importantly, software breakpoints are commonly implemented as undefined instructions - the debug agent uses the hardware exception event to implement the breakpoint handler
- Thread scheduled for the first time. The run-mode debug agent uses this event to apply any uncommitted thread-specific breakpoints that apply. Deferring the breakpoint commit is especially useful if the breakpoint is in shared code where false triggers may cause noticeable performance degradation
- Kernel object updates. When a process, thread or library is created or destroyed, or a process's attributes are updated, an event is generated. (Process attribute changes can be caused by a name change, or a chunk being added or removed from the process.) Debuggers, more than most other software, care about what code is running on the mobile phone. The debug agent often uses these notifications for housekeeping tasks: process and library creation events are indications that new code has been loaded to the phone, which the debug tool may like to annotate or commit breakpoints to
- Thread object updates. These happen when a thread terminates, or thread attributes are updated (always a thread name change). These events are usually reported to the developer for information
- Chunk object updates. These happen when a chunk is created or destroyed, or memory is committed or de-committed from a chunk, or a chunk's size is adjusted: for example, creating a thread or a heap, or calling `RChunk::Create()`. A stop-mode debugger closely monitors the memory mappings created by the kernel. To avoid crashing the target while it is halted, it must be certain that addresses it is attempting to access are actually mapped in. Other debug tools may also present this information to the developer
- Code objects. These events happen when a user-side library (DLL) is loaded, closed or unloaded, or code segments are loaded or unloaded: for example, using `RLibrary::Load()`, `RProcess::Create()` or `RLibrary::Close()`. As I mentioned earlier, library events are indications that new code has been loaded or unloaded to the phone. The debug tool may like to take this opportunity to annotate or commit breakpoints in the new code
- Device drivers are loaded or unloaded, using APIs such as `User::LoadLogicalDevice()`
- User-side trace output - this indicates that `RDebug::Print()` has been called. Debuggers and tracing tools may want to capture calls to the trace port to allow the string to be redirected to an alternative output channel. (See Section 14.6 for more on this.)

Kernel events reference documentation

A detailed description of each event can be found in the Symbian Developer Library's C++ component reference section, and in the source for `TKernelEvent` inkernel\kernel.h. A summary is provided here:

`TKernelEvent`	Meaning
`EEventSwExc`	The current user-side thread has taken a software exception, `(User::RaiseException())`. The exception type is provided as the first argument to the handler. `NKern::UnlockSystem()` has been called by the kernel. The current thread can be discovered from `Kern::CurrentThread()`. (See Chapter 6, *Interrupts and Exceptions,* for more on exceptions.)
`EEventHwExc`	The current thread has taken a hardware exception. A pointer to the structure on the stack containing the thread context is the passed as the first argument. This structure is CPU specific. For the ARM processor the structure is `TArmExcInfo`. This pointer has the same value as returned by `DThread::Context()`.

	(Again, see Chapter 6, *Interrupts and Exceptions*, for more on exceptions.)
EEventAddProcess	Event delivered when a process is created (that is, during a call to `RProcess::Create` or Kern::ProcessCreate). Argument 1 points to the process being created. Argument 2 points to the creator thread (which may not be the current thread). In some cases, the creator thread cannot be reliably determined and this will be set to NULL. The process being created is partly constructed (and has no threads and no chunks). The event is triggered just after creation.
EEventUpdateProcess	Event delivered after a process attribute change. Currently this applies only to process renaming and a change to the address space through chunk addition/removal, though we may extend it in the future. Argument 1 points to the process being modified. The process lock may be held. The event is triggered just after the name change, just after chunk is added, or just before a chunk removal.
EEventRemoveProcess	Event delivered when a process terminates. The first argument points to the process (DProcess) being terminated. The current thread is the kernel server thread. The process is partly destructed, so its resources should be accessed only after checking they still exist.
EEventLoadedProcess	Event delivered immediately after a process is created (that is, during a call to `RProcess::Create` or Kern::ProcessCreate). Argument 1 points to the process. The process being created is fully constructed.
EEventUnloadingProcess	Event delivered when a process is being released, but before its code segment, stack chunk and so on are unmapped. Argument 1 points to the process. The process being released is fully constructed.
EEventAddThread	Event delivered when a user or kernel thread is created (that is, during a call to `RProcess::Create`, `RThread::Create` or Kern::ThreadCreate). The thread being created is fully constructed but has not executed any code yet. Argument 1 points to the thread being created. Argument 2 points to the creator thread (which may not be the current thread).
EEventStartThread	Event delivered when a user or kernel thread is scheduled for the first time. The thread has not executed any code yet. The current thread is the thread being scheduled. Argument 1 points to the thread being scheduled.
EEventUpdateThread	Event delivered after a thread attribute change. Currently this applies only to thread renaming but we may extend it in the future. Argument 1 points to the thread being modified.
EEventKillThread	Event delivered when a user or kernel thread terminates. The current thread and argument 1 is the thread being terminated. This is in the ECSExitInProgress state, and so cannot be suspended. The thread's address space can be inspected.
EEventRemoveThread	Event delivered when a user or kernel thread is about to be closed. The current thread is the kernel thread. Argument 1 points to the thread being terminated. The thread is partly destructed so its resources should be accessed only after checking if they still exist.
EEventNewChunk	Event delivered when a chunk is created. Argument 1 points to the chunk being created.
EEventUpdateChunk	Event delivered when physical memory is committed to or released from a chunk. Argument 1 points to the chunk being modified.
EEventDeleteChunk	Event delivered when a chunk is deleted. Pointer to the chunk is provided as an argument.
EEventAddLibrary	Event delivered when a user-side DLL is explicitly loaded. Argument 1 points to the `DLibrary` instance being loaded. Argument 2 points to the creator thread. DLibrary::iMapCount is equal to 1 if the DLL is loaded for the first time into the creator thread's address space. If the DLL is being loaded for the first time, any global constructors haven't been called yet. The DLL and all its dependencies have been mapped. The system-wide mutex `DCodeSeg::CodeSegLock` is held.
EEventRemoveLibrary	Event delivered when a previously explicitly loaded user-side DLL is closed or unloaded (that is, a call to `RLibrary::Close`). Argument 1 points to the `DLibrary` instance being unloaded. `DLibrary::iMapCount` is equal to 0 if the DLL is about to be unloaded. If the DLL is about to be unloaded, its global destructors have been called but it is still mapped (and so are its dependencies).

	The system-wide mutex `DCodeSeg::CodeSegLock` is held when this event is triggered.
`EEventAddCodeSeg`	Event delivered when a code segment is mapped into a process. Argument 1 points to the code segment, and argument 2 points to the owning process. The system-wide mutex `DCodeSeg::CodeSegLock` is held.
`EEventRemoveCodeSeg`	Event delivered when a code segment is unmapped from a process. Argument 1 points to the code segment. Argument 2 points to the owning process. The system-wide mutex `DCodeSeg::CodeSegLock` is held.
`EEventLoadLdd`	Event delivered when an LDD is loaded. Argument 1 points to the LDD's code segment (which is an instance of `DCodeSeg`). The current thread will always be the loader thread. The event is triggered before the LDD factory function is called.
`EEventUnloadLdd`	A LDD is being unloaded. The current thread is always the loader thread. The LDD's code segment (DCodeSeg instance) is passed as argument 1.
`EEventLoadPdd`	A PDD has been loaded. The current thread is always the loader thread. The first argument is the PDD's code segment (DCodeSeg instance). The PDD factory function has not been called yet.
`EEventUnloadPdd`	Event delivered when a PDD is unloaded. The current thread is always the loader thread. The first argument points to the PDD's code segment (DCodeSeg instance).
`EEventUserTrace`	Event delivered when `RDebug::Print` has been called in user-side code. The current thread is the user-side caller. Argument 1 points to the user-side buffer containing the Unicode string for printing. The characters cannot be accessed directly, because they are in user-space, so they string must copied using kumemget(). The event is delivered in a thread-critical section, so the call to kumemget() must be protected with XTRAP. Argument 2 holds the length of the string in characters. The size of the buffer is twice the length. On exit from the event handler use `DKernelEventHandler::ETraceHandled` to prevent further processing of the trace request by the kernel.

Kernel event dispatch

To issue an event, the kernel calls `DKernelEventHandler::Dispatch()` at the appropriate place in the code. Some wrapper macros are provided for this function, to conditionally compile the event dispatch, including it only when debugger support is enabled (that is, when _DEBUGGER_SUPPORT_ is defined, and `DKernelEventHandler::DebugSupportEnabled()` is true).

```
// Dispatch kernel event aEvent
#define __DEBUG_EVENT(aEvent, a1)
#define __DEBUG_EVENT2(aEvent, a1, a2)

// Dispatch kernel event aEvent if condition aCond is true
#define __COND_DEBUG_EVENT(aCond, aEvent, a1)
```

The previous table shows the guarantees that are made for each event about the current state of the kernel and the object passed in.

Kernel event capture

When implementing a debug agent, you will need to provide event handlers for the events you wish to capture. In this section, I will discuss how this is done.

To capture events, you simply create an instance of `DKernelEventHandler` and add this to the kernel's event handler queue. During construction, you provide a pointer to your event handler function, and some private data. The next time any event is issued, each event handler in the queue will be called in order.

```
// Derive an event handler class
class DMyEventHandler : public DKernelEventHandler
    {
public:
    DMyEventHandler();
private:
```

```
    static TUint EventHandler(TKernelEvent aEvent, TAny* a1, TAny* a2, TAny* aThis);
    };

DMyEventHandler::DMyEventHandler() : DKernelEventHandler(EventHandler, this) {}
```

The kernel will maintain an access count on the handler so, when the time comes, it can correctly ascertain when it is safe to destruct the object. The kernel won't destroy the object if there are any threads currently executing the handler. When cleaning up, you should use the `Close()` method to remove the object rather than deleting it.

You can now implement the event handler function. The first parameter of the hander indicates the type of the event. The event type determines the semantics of the next two (void *) parameters. The function is always called in the thread-critical section.

The following simple code snippet shows a handler that counts the total number of processes started by the kernel since the handler was installed:

```
TUint gAllProcessesCount = 0;

TUint DMyEventHandler::EventHandler(TKernelEvent aEvent,
TAny* a1, TAny* a2, TAny* aThis)
    {
    switch (aType)
        {
        case EEventAddProcess:
            // increment the process counter
            gAllProcessesCount++;
        default:
            break;
        }
    return DKernelEventHandler::ERunNext;
    }
```

Inside the hander you can use the following functionality:

- Reading/writing of thread memory
- Getting/setting of the thread's context information. If the context is changed, the remaining handlers will not have access to the original context
- Signaling threads, mutexes and other synchronization objects
- Waiting on mutexes and other synchronization objects
- Suspending the thread.

Your handler's return value, a bit-field, determines what happens next:

- If bit `ERunNext` is not set, the kernel will not run any more handlers
- for this event
- If the event is a user trace, setting bit `ETraceHandled` will stop any further processing of the trace command by the kernel. This is useful if you want to intercept the trace for processing yourself, and prevent the kernel outputting it to the usual debug channel
- If `EExcHandled` is set, the kernel will not perform the usual cleanup code for the thread that generated the exception. (The kernel won't generate a *KERN-EXEC 3* and won't make an attempt to destroy the thread object.)

It is worth noting that we may choose to extend the set of events that the kernel generates in the future. The handler should take care to respond to any unknown `TKernelEvent` values by returning `ERunNext`.

Example code for testing the capturing of hardware exceptions and panic events can be found here:

```
\base\e32utils\d_exc\d_exc.mmp
```

Context switch event

To complete the set of notifications available to a debug agent, the kernel provides a context switch event. This event is triggered on every change to the currently scheduled thread. This notification is particularly useful for the implementation of software profilers - for example, to record the length of time spent in each thread.

In this API, we wanted to allow a device driver or extension to provide a callback function that would be called by the scheduler after every context switch. During the implementation, we were conscious that this was impacting a critical part of the scheduler and we were not prepared to compromise its performance.

The cost for a typical implementation of the callback mechanism on an ARM processor would be three instructions:

1. Load the function pointer for the callback
2. Compare it to NULL
3. Execute the callback if non-NULL.

This three-instruction cost is paid at every call to the scheduler, even if the callback function is not provided.

To work around this performance impact, we devised an alternative mechanism for installing the callback function: the kernel provides two implementations of the scheduler code segment affected by the callback. The kernel also provides functions to replace the fast version (with no callback) with the slower version that supports the callback hook.

The kernel publishes the following functions:

```
NKern::SchedulerHooks(TLinAddr &start, TLinAddr &end)
```

This returns the address in the scheduler where the callback trampoline should be placed. (This is located towards the end of the scheduler, after it has selected the new thread to run.)

```
NKern::InsertSchedulerHooks()
```

This is the function for patching-in the callback trampoline. It constructs a branch-and-link instruction to the callback trampoline and inserts it at the address returned by `NKern::SchedulerHooks`. It performs a write to an instruction in the scheduler code, which is usually in ROM - so you must shadow the ROM first.

In the general case in which no software is using the event, this implementation has zero speed overhead. Because of this, the API is a little more awkward to use, but this is clearly a compromise worth making for an API that has only a few specialized clients, and affects such a performance critical area of the nanokernel.

Tools that wish to use the context switch event are responsible for shadowing the appropriate area of scheduler code, and calling the function to patch the scheduler. Let's look at some example code.

First we install the scheduler hooks by shadowing the appropriate area of ROM, and then we call the `NKern::InsertSchedulerHooks()` function to patch the scheduler:

```
TInt InsertSchedulerHooks()
    {
    // Get range of memory used by hooks
    TLinAddr start,end;
    NKern::SchedulerHooks(start,end);

    // Create shadow pages for hooks
    TUint32 pageSize=Kern::RoundToPageSize(1);
    for(TLinAddr a=start; a<end; a+=pageSize)
        {
        NKern::ThreadEnterCS();
        TInt r=Epoc::AllocShadowPage(a);
        NKern::ThreadLeaveCS();
        if(r!=KErrNone && r!=KErrAlreadyExists)
            {
            RemoveSchedulerHooks();
            return r;
            }
```

```
        }
    // Put hooks in
    NKern::InsertSchedulerHooks();

    // Make I and D caches consistent for hook region
    Cache::IMB_Range(start,end-start);
    return KErrNone;
    }
```

Now the hooks are installed, we can ask to be called back on every context switch:

```
// Ask for callback
NKern::SetRescheduleCallback(MyRescheduleCallback);
```

The callback function that you provide must have the following prototype:

```
void MyRescheduleCallback(NThread* aNThread);
```

You can temporarily disable context switch callbacks by passing a NULL function pointer to `NKern::SetRescheduleCallback()`. To completely remove the rescheduling hooks, we do the following:

```
void RemoveSchedulerHooks()
    {
    // Prevent rescheduling whilst we
    // disable the callback
    NKern::Lock();

    // Disable Callback
    NKern::SetRescheduleCallback(NULL);

    // Invalidate CurrentThread
    CurrentThread() = NULL;

    // Callback now disabled...
    NKern::Unlock();

    // Get range of memory used by hooks
    TLinAddr start,end;
    NKern::SchedulerHooks(start,end);

    // Free shadow pages which cover hooks
    TUint32 pageSize=Kern::RoundToPageSize(1);
    NKern::ThreadEnterCS();
    for(TLinAddr a=start; a<end; a+=pageSize)
        Epoc::FreeShadowPage(a);
    NKern::ThreadLeaveCS();
    }
```

Your callback function will be called with the kernel preemption lock held, during a reschedule. (For more information on rescheduling see Section 3.6.) As I've said, this area of code is performance sensitive, and your callback function should therefore use as few processor cycles as possible.

The callback function you provide should be of the type `TRescheduleCallback` (see INCLUDE\NKERN\NKERN.H). The kernel passes a pointer to the newly scheduled `NThread` as a parameter to your function. In some cases you will need to find the Symbian

OS thread that corresponds to this nanokernel thread. You can construct it as follows:

```
DThread* pT = _LOFF(aNThread, DThread, iNThread);
```

Before doing this, you should first check that the thread really is a Symbian OS thread, and not a thread belonging to the personality layer, as shown in Section 3.3.3.1.

Example code for testing the scheduling hooks and event capture can be found here:

```
\base\e32\kernel\kdebug.cpp
\base\e32test\debug\d_schedhook.cpp
\base\e32test\debug\d_eventtracker.cpp
```

Thread context API

The kernel exposes an API that allows debug agents to get and set user-side thread context. The API allows the thread context to be retrieved and altered for any non-current, user-side thread. This allows the debugger to display register state for any thread in the system, access any thread's stack to display a stack trace and modify program flow.

The nanokernel does not systematically save all registers in the supervisor stack on entry into privileged mode, and the exact subset that is saved depends on why the switch to privileged mode occurred. So, in general, only a subset of the register set is available, and the volatile registers may be corrupted.

Function	Description
e32\include\kernel\kernel.h `DThread::Context(TDes8 &)`	Retrieve the user thread context to the descriptor provided. This is a virtual method, implemented by the memory model, `(DArmPlatThread::Context()` in e32\KERNEL\ARM\CKERNEL.CPP).
INCLUDE\NKERN\NKERN.H NKern::ThreadGetUserContext	Get (subset of) user context of specified thread. See nkern\arm\ncthrd.cpp for more information.
INCLUDE\NKERN\NKERN.H NKern::ThreadSetUserContext	Set (subset of) user context of specified thread. See nkern\arm\ncthrd.cpp for more information.

The current thread context can also be read from user-side code by using the method `RThread::Context()`.

Example test code that exercises thread context can be found in:

```
base\e32test\debug\d_context.h
base\e32test\debug\d_context.cpp
base\e32test\debug\t_context.cpp
```

Context type

We've seen that, on entry to privileged mode, the nanokernel does not systematically save all registers in the supervisor stack. To improve performance, it only pushes registers when it needs to.

To retrieve and modify the user context for any non-current threads, we need to discover exactly where on its stack the thread's context has been pushed. To do this, we must work out what caused the thread to be switched from user mode. The set of possibilities is enumerated in the type `NThread::TUserContextType` (include\nkern\arm\ nk_plat.h)

```
// Value indicating what event caused thread to enter
// privileged mode.

enum TUserContextType
    {
    EContextNone=0, // Thread has no user context
    EContextException=1, // Hardware exception
    EContextUndefined,
    EContextUserInterrupt, // Preempted by interrupt
    // Killed while preempted by int taken in user mode
```

```
    EContextUserInterruptDied,
    // Preempted by int taken in executive call handler
    EContextSvsrInterrupt1,
    // Killed preempted by int taken in exec call hdler
    EContextSvsrInterrupt1Died,
    // Preempted by int taken in executive call handler
    EContextSvsrInterrupt2,
    // Killed preempted by int taken in exec call handler
    EContextSvsrInterrupt2Died,
    EContextWFAR, // Blocked on User::WaitForAnyRequest()
    // Killed while blocked on User::WaitForAnyRequest()
    EContextWFARDied,
    EContextExec, // Slow executive call
    // Kernel-side context (for kernel threads)
    EContextKernel,
    };
```

The kernel does not keep a record of this type, since it is not usually required. The value is calculated only on demand, by calling:

```
IMPORT_C TUserContextType NThread::UserContextType();
```

Apart from the previous context functions, the stop-mode debug API is the primary client for this API (see Section 14.5.2).

Code and data section information

When an executable is loaded from disk, the loader dynamically allocates code and data chunks for it, as I discussed in Chapter 10, *The Loader*. The linear address of these chunks is unlikely to match the link address in the executable's symbolic debug information, so the debugger has to determine the address allocated by the loader, which allows it to relocate the symbolic debug information to match the run address.

The kernel exposes APIs to get the base addresses of code, data and BSS sections for any library or process:

```
\generic\base\e32\include\e32cmn.h
class TModuleMemoryInfo;

\generic\base\e32\include\e32std.h
TInt RProcess::GetMemoryInfo(TModuleMemoryInfo& aInfo);
TInt RLibrary::GetRamSizes(TInt& aCodeSize, TInt& aConstDataSize)
TInt RProcess::GetRamSizes(TInt& aCodeSize, TInt& aConstDataSize,
TInt& anInitialisedDataSize, TInt& anUninitialisedDataSize)

generic\base\e32\include\kernel\kern_priv.h
TInt DCodeSeg::GetMemoryInfo(TModuleMemoryInfo& aInfo, DProcess* aProcess);
```

TModuleMemoryInfo returns

- The base address of the code section (.text)
- The size of the code section
- The base address of the constant data section (.rdata)
- The size of the constant data section
- The base address of the initialized data section (.data)
- The base address of the uninitialized data section (.bss)
- The size of the initialized data section.

The D_EXC logger shows these APIs in use:

```
\base\e32utils\d_exc\minkda.cpp
```

ROM shadow API

The kernel publishes APIs that allow debug agents to shadow pages of ROM in RAM. This allows the debugger to set and clear breakpoints in the ROM address range.

Function defined in memmodel\epoc\platform.h	Description
Epoc::AllocShadowPage	Allocates a shadow page for the given address. Returns `KErrAlreadyExists` if the ROM page has already been shadowed.
Epoc::FreeShadowPage	Frees a shadow page for the given address.
Epoc::FreezeShadowPage	Freezes a shadow page for the given address, that is, the page is made read-only.

The thread must be in its critical section when these calls are made. The thread can enter its critical section with `NKern::ThreadEnterCS()`, and exit with `NKern::ThreadLeaveCS()`.

The implementations of these functions can be found in memmodel\epoc\mmubase\mmubase.cpp. They are memory model-dependent.

Example code demonstrating the ROM shadow API can be found here:

```
\generic\base\e32test\mmu\d_shadow.h
\generic\base\e32test\mmu\d_shadow.cpp
\generic\base\e32test\mmu\t_shadow.cpp
```

Target debugger agents

The kernel debug interface provides the foundation for building and integrating tools into the Symbian OS platform. In this and subsequent sections of this chapter, I will describe how the different types of tools are implemented on top of this API.

Debug agent

A debug agent for a run-mode debugger must translate the debug protocol requests received from its host debugger into actions for the kernel and other Symbian OS servers.

The responsibilities for the debug agent were outlined in Section 14.3.2. However, the division of responsibilities between the agent and host is flexible: for example, it is possible to implement CPU instruction step on the host or the target, but it is almost always the case that the host performs the relocation of symbolic debug information for source association.

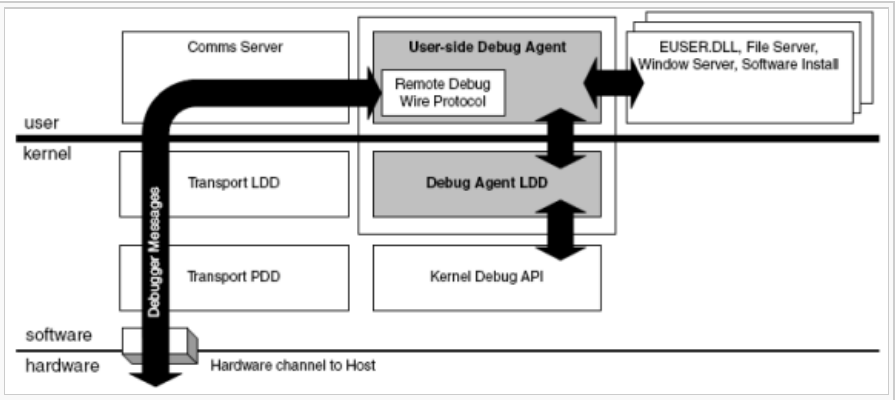

Figure 14.7 *User-side and kernel-side debug agent for a run-mode debugger*

Figure 14.7 shows the architecture of the debug agent. You can see from the diagram that the debug agent has dependencies on other components in the system, such as the user library, file server and software install. Because of this, it is not possible to place breakpoints in any of those components. These components are essential to the correct operation of the debug agent - suspending any threads serving the debug agent will make the agent inoperable. For example, if the debugger has configured the serial port as the debug channel, you cannot place a breakpoint in the communication server's serial protocol module.

In EKA1, it was not possible to place breakpoints anywhere in the user library because the kernel was linked to this library too. This is no longer a problem in EKA2, since kernel-side software now has its own utility library.

Similarly, if the user-side agent depends on functionality in any other server, then it will not be able to debug that server.

The debugger implementation should seek to minimize these restrictions. This usually involves providing a duplicate implementation for the private use of the debug agent.

JIT debugging

Symbian OS supports JIT debugging directly for the emulator only. However, the debug agent can implement JIT on target hardware, by placing JIT debug traps (implemented as breakpoints) on the common termination points for user-side threads. The address of these functions in EUSER.DLL can be discovered by using the `RLibrary::Lookup()` method on the ordinal for each. When a JIT breakpoint is triggered, the *reason* information can be discovered from the thread's context.

Function	Ordinal (gcc build)	
`RThread::Panic()`	812	The category is in r1, the panic number is in r2.
`RProcess::Panic()`	813	The category is in r0, the panic number is in r1.
`RThread::RaiseException()`	868	R0 holds the exception number.
`User::RaiseException()`	1910	R1 holds the exception number.

14.4.3 Breakpoints

The handling of hardware exceptions is critical for the functioning of a run-mode debugger on ARM platforms. Software breakpoints are implemented by replacing code at the break location with an *undefined instruction*, or the *BKPT* opcode on ARM v5 processors. When the thread executes the undefined instruction, the CPU will generate an exception.

You can write software breakpoints to RAM-loaded code by modifying the code chunk's permissions. You can write them to code in ROM by first shadowing the target page of ROM in RAM (as I discussed in Section 14.3.4). When writing breakpoints into a code area, you should make certain that the cache is coherent with the modified code in RAM. This will ensure that the breakpoint instruction is committed to main RAM before the code is executed. The cache operations required to maintain coherence are dependent on the mobile phone's memory architecture, but a call to the instruction memory barrier function (Cache::IMB_Range) specifying the modified address range will perform the necessary operations. (I discussed caches and memory architectures in more detail in Chapter 2, *Hardware for Symbian OS*.)

Once a breakpoint is set, the kernel debug agent can capture its breakpoint exceptions by installing an event handler for `EHwEvent`.

When the exception handler is run, the debug agent can determine what to do next. If the exception was not due to a breakpoint set by the debugger, then the agent can pass the exception onto the next handler (and ultimately the kernel). If the exception was a breakpoint intended for the current thread, then the agent can suspend the thread with `DThread::Suspend()`, then replace the instruction removed by the breakpoint and notify the host debugger via the user-side agent.

Stop-mode debug API

The vast majority of hardware platforms supported by Symbian OS are ICE-enabled. Kernel developers and those porting the operating system to new hardware often have access to development boards exposing the JTAG interface, and allowing the use of CPU-level debuggers.

The main problem with OS support in a stop-mode debugger is that there is little or no opportunity for the operating system to run code on behalf of the debugger to enable it to perform the tasks it needs. There is no debug agent, and no high-level wire protocol between host and target; communication is through remote memory reads and writes initiated over JTAG. While the CPU is halted, the debugger must do all the data discovery and object manipulation for itself by rummaging around inside the kernel - the *OS awareness* is pushed onto the host.

However, Symbian OS publishes an API to make this a little easier. It is a table-driven interface onto the data and kernel objects: a virtual *map* of the kernel. We implement the map as a memory structure that can be easily navigated with memory reads initiated over the JTAG interface.

Thread and process awareness is common in the run-mode debuggers used for application development, but far less common in JTAG stop-mode debuggers. Using the API provided, it is possible to integrate the following features into a stop-mode debugger:

- Thread and process awareness
- Thread-specific breakpoints
- Memory management awareness
- Code segment management.

The EKA2 stop-mode debug API is similar in principle to the API provided in EKA1. However, EKA2 has one significant design

change. We made this change with the intention of improving performance and reducing interference in timing characteristics of the target when a debugger is attached.

The EKA1 solution copied key information from the kernel into a *debug log* for convenient access by the debugger. EKA2 does not copy information - instead, the debugger must locate the information in-place. This design improves the performance of Symbian OS while the debugger is attached, at the cost of a slightly more complex client interface (see Figure 14.8).

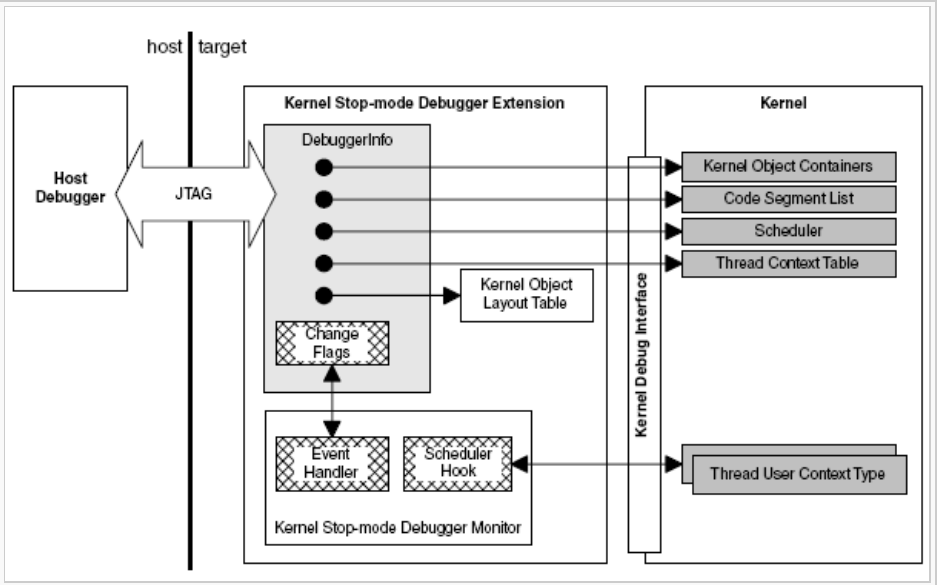

Figure 14.8 *Stop-mode debugger interface*

The stop-mode debugger API is built on top of the kernel's core debug APIs. It is implemented as a kernel extension (KDEBUG.DLL), which is enabled simply by including it in a ROM. For most variants, you can do this simply by defining the STOP_MODE_DEBUGGING macro. Alternatively, you can add the following line to <variant>rom\kernel.iby:

```
extension[VARID]=
\Epoc32\Release\<assp>\urel\KDEBUG.DLL \System\Bin\kdebug.dll
```

The `DDebuggerInfo` interface implemented by the KDEBUG extension is the debugger's gateway into the kernel. The host initially locates the gateway from a pointer stored at a constant address in the superpage:

```
DDebuggerInfo* TSuperPage::iDebuggerInfo
```

(The superpage is used for communication between the bootstrap and the kernel, and is described in Section 16.2.2.1.)

The gateway contains:

- The object layout table, which provides a virtual *map* of the kernel's data types, to allow them to be navigated. The table also isolates the debugger from minor changes in layout of kernel objects
- Pointers to the kernel's object containers for threads, processes, libraries, memory chunks, semaphores, and so on
- Access to the scheduler for information about the current thread, and current address space
- Access to the kernel's locks that indicate the validity and state of various kernel objects
- A mechanism for retrieving the context of any non-current thread.

The KDEBUG extension installs an event handler, which it uses to update *change flags*. These flags can be read by the host debugger to determine if any new kernel objects have been removed (for example, thread death) or created (for example, library loaded).

The extension has a minimal impact on performance when installed. Furthermore, there is no overhead on the phone software since the interface can be entirely removed on phones that don't expose JTAG hardware, without re-compiling the kernel.

The implementation can be found here:

```
e32\include\kernel\kdebug.h
e32\kernel\kdebug.cpp
```

Kernel object layout table

The kernel object layout table provides the host debugger with a virtual map of the kernel's data structures. The table is an array of offsets of member data from the start of the owning object.

Given a pointer to a kernel object, the address of any object of a known type, and any of its members can be found by looking up the offset in the table and adding it to the object pointer.

Using this method, and starting from the `DDebuggerInfo` object, the host debugger can walk the kernel's data structures by issuing the appropriate memory reads over JTAG.

The table layout (host interface) is defined in this header file:

```
e32\include\kernel\debug.h
```

Here is a small section of this file so that you can see what it looks like:

```
e32\include\kernel\debug.h

enum TOffsetTableEntry
    {
    // thread info
    EThread_Name,
    EThread_Id,
    EThread_OwningProcess,
    EThread_NThread,
    EThread_SupervisorStack,

    // scheduler info
    EScheduler_KernCSLocked,
    EScheduler_LockWaiting,
    EScheduler_CurrentThread,
    EScheduler_AddressSpace,

    // and so on ...
    }
```

The constants published in this file correspond to indices in the object table defined by the stop-mode debug API.

```
const TInt Debugger::ObjectOffsetTable[]=
    {
    // thread info
    _FOFF(DThread, iName),
    _FOFF(DThread, iId),
    _FOFF(DThread, iOwningProcess),
    _FOFF(DThread, iNThread),
    _FOFF(DThread, iSupervisorStack),

    // scheduler info
    _FOFF(TScheduler, iKernCSLocked),
    _FOFF(TScheduler, iLock.iWaiting),
    _FOFF(TScheduler, iCurrentThread),
    _FOFF(TScheduler, iAddressSpace),

    // and so on ...
    }
```

Symbian builds and delivered this table with every release of the kernel. Indirection through the table provides a level of binary compatibility for the host debugger - the indices will not change between releases of the OS, even if the actual layout of kernel objects does change.

Thread context

Acquiring the thread context of any non-current thread presents a challenge for a stop-mode debugger. It is worth examining the solution in a little more detail.

The context for the current thread is always available directly from the processor. The context for any non-current thread is stored in its supervisor stack. However, as I mentioned in Section 14.3.2, it is not always straightforward to determine where the registers are placed in the stack frame - or, indeed, which register subset has been saved, and in which order the registers were pushed. This will depend on the reason the switch to privileged mode occurred: the thread's user context type. (I list the `TUserContext` types in Section 14.3.2.1.)

In a multi-session debugger, where non-current threads may be visible, the host debugger needs to be able to identify the context of any thread at any time - it must always be able to determine the user context type for a thread.

The kernel does not routinely store this information, so the stop-mode debug API installs a scheduler callback to update the thread's context type on every reschedule. The result is stored in the `NThread` object:

```
inline TInt NThread::SetUserContextType()
```

The context type value can be used as an index into the user context tables. This will yield a structure that describes the layout of the thread's stack, as shown in Figure 14.8.

```
static const TArmContextElement* const*
NThread::UserContextTables();

const TArmContextElement* const ThreadUserContextTables[] =
    {
    ContextTableUndefined, // EContextNone
    ContextTableException,
    ContextTableUndefined,
    ContextTableUserInterrupt,
    ContextTableUserInterruptDied,
    ContextTableSvsrInterrupt1,
    ContextTableSvsrInterrupt1Died,
    ContextTableSvsrInterrupt2,
    ContextTableSvsrInterrupt2Died,
    ContextTableWFAR,
    ContextTableWFARDied,
    ContextTableExec,
    ContextTableKernel,
    0 // Null terminated
    };
```

This structure holds 18 pointers to tables (one for each thread context type). Each item is an array of `TArmContextElement` objects, one per ARM CPU register, in the order defined in `TArmRegisters`:

```
// Structure storing where a given
// register is saved on the supervisor stack.
class TArmContextElement
    {
public:
    enum TType
        {
```

```
        // register is not available
    EUndefined,
    // iValue is offset from stack pointer
    EOffsetFromSp,
    // iValue is offset from stack top
    EOffsetFromStackTop,
    // value = SP + offset
    ESpPlusOffset,
    };

public:
    TUint8 iType;
    TUint8 iValue;
    };

enum TArmRegisters
    {
    EArmR0 = 0,
    EArmR1 = 1,
    EArmR2 = 2,
    EArmR3 = 3,
    EArmR4 = 4,
    EArmR5 = 5,
    EArmR6 = 6,
    EArmR7 = 7,
    EArmR8 = 8,
    EArmR9 = 9,
    EArmR10 = 10,
    EArmR11 = 11,
    EArmR12 = 12,
    EArmSp = 13,
    EArmLr = 14,
    EArmPc = 15,
    EArmFlags = 16,
    EArmDacr = 17,
    };
```

The `TArmContextElement::iType` determines how the register location should be calculated. Figure 14.9 shows an example of a thread context table and its associated *thread context state*.

The algorithm for obtaining a thread context is intended to be run on the host by a stop-mode debugger. Here it is:

```
reg_value GetSavedThreadRegister(<thread>,<reg-id>)
    {
    type = READ(thread, EThread_UserContextType)
    IF ( KernelLock != 0 )
        RETURN “Kernel Locked”
    IF (thread == CurrentThread)
        RETURN “Register not saved – current thread”

    // Select the appropriate context table
    ContextTable = READ(DDebuggerInfo::iThreadContextTable)
    ContextTable = READ(ContextTable[Type])

    // Get stack pointer and stack top
    SP = READ(<thread>,EThread_SavedSP)
```

```
    StackTop = READ(<thread>, EThread_SupervisorStack)

    + READ(<thread>,EThread_SupervisorStackSize)
 
    // Get the iType and iValue fields for this
    // register from the thread context table
    iType = READ( &ContextTable[<reg-id>].iType )
    iValue = READ( &ContextTable[<reg-id>].iValue )

    // Now handle the cases and calculate the
    IF ( iType == OFFSET_FROM_SP )
        RETURN READ( SP[iValue] );
    ELSE
        IF( iType == OFFSET_FROM_STACK_TOP)
            RETURN READ( StackTop[-iValue] );
    ELSE
        IF( iType == SP_PLUS_OFFSET)
            RETURN SP[iValue];
    ELSE
        // Other case currently not used by OS
        RETURN “Register Not Valid”
    }

    // Read field <offset-tag> from <object>
    val READ(<object>, <offset-tag>)
        {
        offset = OffsetTable[<offset-tag>]
        pointer = <object>
        RETURN READ( pointer[offset] )
        }
```

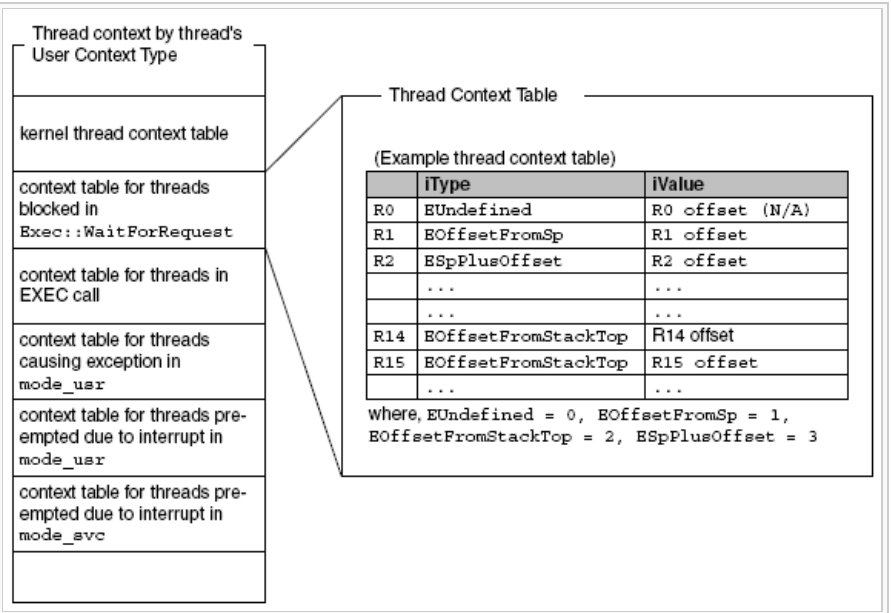

Figure 14.9 *Thread context table and state*

Note that in some states, the Symbian OS kernel exploits the conventions in the ARM-Thumb procedure call standard and thus not all registers in the thread's context will be saved. Typically, the argument registers (that is, R0-R3) are not saved because the caller throws those registers away - so reading the thread context for these registers will return garbage values. This is harmless for user code, and it should be harmless for the debugger too. The debugger, if it has the capability, may choose not to show these registers at all.

Memory of non-current threads

Reading and writing to the current process's memory is straightforward - the process's data will be mapped into its run address, and the host can follow all pointers relative to the current process directly.

However, if the host wants to read and write to an address in a process that is not currently scheduled, it must take into account that the memory may not be available in the current address space, or may appear at a different logical address. The debugger must move to an address space in which the memory is available, and translate the process-relative pointer from its run address into an equivalent pointer in the new address space.

The implementation is memory-model specific. It is an equivalent operation to the `DThread::RawRead()` and `DThread::RawWrite()` methods that the kernel uses when transferring data between processes.

To perform this operation, the debugger must understand how the kernel's memory model works. I will give a short description of the method for each memory model, and you can find more detail in the stop-mode debugger integration guide. I describe the kernel's memory maps for the moving and multiple memory models in Chapter 7, *Memory Models*.

Accessing memory of non-current threads under the multiple memory model

The multiple memory model maintains a memory mapping for each process, and swaps between these address spaces at a reschedule. When the processor is halted, the debugger will have access to the memory of the current process and the kernel address space, but this will not contain mappings for any other process's memory.

To access memory belonging to a non-current process, the host debugger has two options:

- Create a new temporary memory mapping exposing the memory from the non-current process, then translate the process-relative pointer to the new mapping
- Temporarily move to an address-space that already contains a mapping for the target memory.

The former is the method used by the `DThread::RawRead()`}and `DThread::RawWrite()` methods. For the host debugger, it is likely to be more practical to simply re-use the address space for the target process that is provided by the kernel.

The host can change the address space by modifying the appropriate MMU registers. (For ARMv6 processors the debugger programs the ASID and TTBR with the values provided in the target `DProcess` object.) Once in the appropriate address space, the process-relative pointer can be used to access the memory.

Accessing memory of non-current threads under the moving memory model

The moving memory model maps the current process's memory into the run section. All non-current processes are mapped into the kernel's home section, and are only visible to the kernel. When the processor is halted, the debugger can grant itself access to the kernel's memory by modifying the MMU access control register, DACR.

To use a process-relative pointer to access memory belonging to a non-current process, the host debugger must obviously take into account where in the home section the kernel has mapped the process's memory chunks. It is most efficient to use the same calculation that is implemented by the kernel's `DThread::RawRead()` and `DThread::RawWrite()` methods, which I will now briefly describe.

To perform the pointer translation, the debugger iterates through the process's list of chunks (DMemModelProcess::iChunks) until it finds the chunk with an address range that covers the target address. The home address can now be calculated as:

```
home_address = chunk.iChunk.iHomeBase + target_address – chunk.iDataSectionBase
```

Accessing memory of non-current threads under the direct memory model

The direct memory model maintains only a single memory mapping that is valid for current and non-current processes. It is sufficient for the debugger to ensure that it has read/write access permissions before using a process-relative pointer.

Kernel state

The host debugger may halt the target processor at any time. This means that the debugger might find kernel data structures in an indeterminate state - for example if the kernel was interrupted in the middle of updating them. Before walking the kernel data structures, the debugger must ensure that the kernel is self-consistent.

The debugger can determine this by examining the kernel and system locks that are exposed through the debug API. The kernel and system locks show when it is safe to access kernel objects, see Section 3.6. If any of the locks are non-zero, the debugger cannot assume that either the thread list or the MMU mappings are in a consistent state. This means that it is unsafe to walk the kernel's data structures. Most debuggers relay this information to their user via the IDE's UI, by graying out the OS visualizations.

The debugger could also repeatedly step the processor, until the locks are cleared.

Kernel trace channel

The kernel provides tracing support as the lowest common denominator debugging tool. The trace port is available for all software, from the bootstrap and device drivers, right up to C++ applications. Software can output trace strings through the trace port to assist with development and diagnosis.

By default, most hardware platforms will configure a serial port as a debug channel to allow the ASCII strings to be picked up by a host PC with a standard terminal program.

The trace support is extended on some base ports to allow the debug strings to be redirected to another port. Usually this would be another serial port, but it can also be a dedicated hardware debug channel. For example, the kernel implements debug trace over the JTAG data channel for ARM CPUs.

Being able to redirect the kernel trace is invaluable during system integration, where conflicts may arise between high-level software and the trace port.

Figure 14.10 shows the program flow from the two clients (in light gray) to the output to hardware (in dark gray).

Redirect user trace

Any application-trace strings that are passed into the kernel through the `RDebug::Print()` functions can be captured and redirected by a device driver. The capture facility is part of the kernel event notifier described in Section 14.3.1. The driver can handle the trace string and terminate the trace, or it can pass it back to the kernel for processing as usual.

This method can be used to capture trace strings for redirection to an alternative output channel, or for analysis. For example:

- Capture the trace strings and package them for sending over a wire protocol to a host debugger. The host debugger can then display the string in a console or output window
- Redirection to a file, on target, for later download and analysis
- Redirection to RAM, for performance
- Redirection to dedicated trace hardware
- Redirection to an analysis tool.

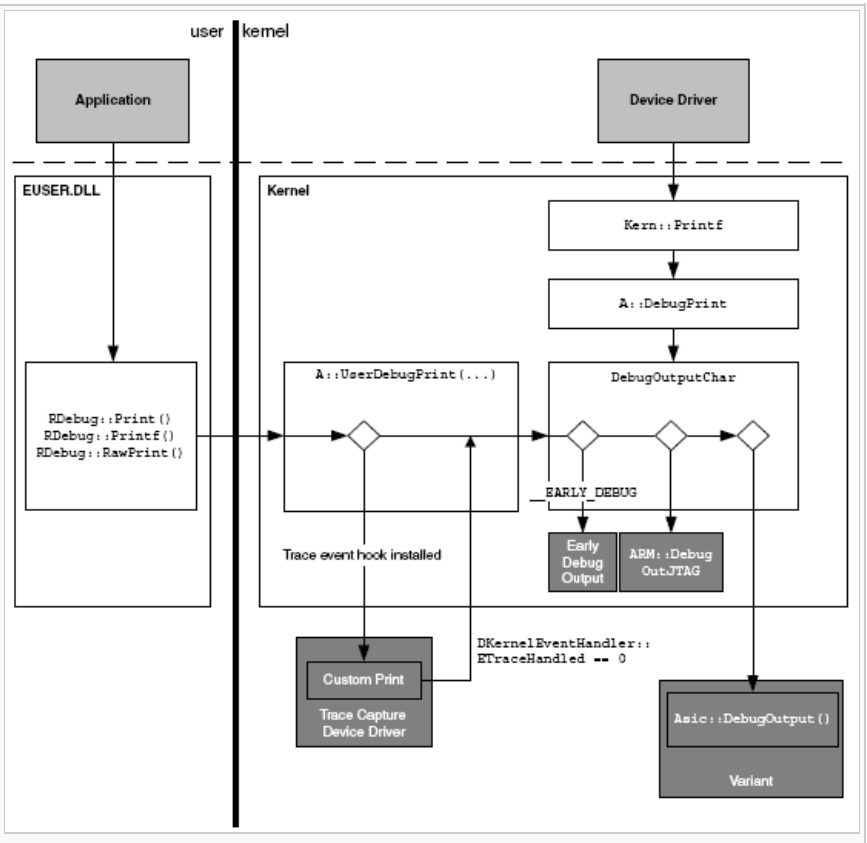

Figure 14.10 *Kernel trace*

Symbian provides example code that demonstrates the use of this API to capture user-side trace and display it in a console window on the mobile phone:

```
\base\e32test\debug\t_traceredirect.cpp
\base\e32test\debug\d_traceredirect.cpp
\base\e32test\debug\d_traceredirect.h
```

Debug output

The channel used to output the trace is determined by the `DebugOutputChar()` function:

```
void DebugOutputChar(TUint aChar)
// Transmit one character to the trace port
// (for debugging only)
```

The default implementation for ARM platforms is defined in \base\e32\kernel\arm\cutils.cpp. If the __EARLY_DEBUG macro is defined, then this function can be replaced by linking-in a custom implementation. Custom *early debug* implementations are provided as a convenience to developers porting Symbian OS to new hardware, for use during the early stages of porting, when other communications channels are unreliable or not available. If the debug port, `TheSuperPage()`.iDebugPort, is set to Arm::EDebugPortJTAG then the kernel outputs the string to the JTAG co-processor data channel (CP14). If the variant DLL is available (that is, Arm::TheAsic is defined), then the string is handed to the variant for output.

The kernel doesn't make any attempt to arbitrate access to the destination port. You will find that if the kernel trace and some other communications protocol, such as PPP, are being directed to the same port, then the two serial streams will be interleaved. In such cases, the communications protocol is likely to fail.

Kernel-side software can use the trace channel by calling

```
Kern::Printf();
```

User-side software uses a slow exec call, Exec::DebugPrint, which is available through the following function wrappers provided in EUSER.DLL:

```
RDebug::Print();
RDebug::Printf();
RDebug::RawPrint();
```

The trace will appear on the port specified in

```
TheSuperPage().iDebugPort
```

This value can be set in a number of ways:

- Call `Hal::Set(EDebugPort, <port>)`
- The DEBUGPORT <port> ESHELL command
- The DEBUGPORT <port> ROMBUILD keyword.

The kernel defines the following values for port:

Constant	Port number	Header file	Meaning
Arm::EDebugPortJTAG	42	kernel\arm\arm.h	Send trace strings to ARM co-processor 14.
`TRomHeader::KDefaultDebugPort`	-1	e32rom.h	Send trace strings to the default port.
`KNullDebugPort`	-2	e32const.h	Don't output trace strings.
	other	variant header	Send the trace

The semantics of other port values are defined by the ASIC. Some ASICs implement these as port numbers (for example, 0, 1, 2, 3, and so on) and others use hardware port addresses.

For example, on the Lubbock platform, when the port is set to 3, the trace appears on the serial port labeled *BTUART*. Any other

value means the trace appears on the port labeled *FFUART*.

On the *H2* reference platform, when port is set to 2, trace output goes to COM3, and when it is set to 0, the trace goes to COM0.

Caveats

There are some problems with this style of tracing which are worth noting. Tracing alters timing. For most user-side software, this will not be a problem. However, with time-critical software such as kernel code, peripheral code and communication stacks, the addition of trace output may cause it to fail. Also, if the defect you are diagnosing is timing-dependent, such as a race condition, then adding trace output can make the problem move somewhere else, making it harder to track down.

Timing problems are compounded if a slow output channel, such as a serial UART is used. You can mitigate the problem by outputting less trace information, or by switching to a faster trace channel. The throughput of the JTAG data channel is typically greater than a serial UART. For some tasks, you may need to use an even faster channel, such as dedicated trace hardware or logging to RAM. You can implement the latter by using the user trace capture API provided by the kernel, see Section 14.3.1 for more details.

Another side effect of compiling trace strings into your code is that its *shape* changes - functions and data move address and also move relative to each other. As a consequence the code may change too (for example, the size of relative branches). Again, this can occasionally affect the reproducibility of your defect.

The kernel also has some tracing blind spots - if you are tracing during power down, you will find that you tend to lose whatever was in the FIFO when the ASIC was moved to a low-power mode. This makes the quality of the trace unpredictable.

Kernel trace in practice

To aid development, we have liberally placed debug trace strings throughout the kernel code, so the activities of the system can be observed.

The kernel trace strings are wrapped in a macro (__KTRACE_OPT) that is expanded only in debug builds. The strings are categorized into 30 functional areas (e32\include\nkern\nk_trace.h), so you can choose to get trace output from only the areas you are interested in. F32 follows a similar model.

Macro	Bit number	Trace strings relating to
KHARDWARE	0	Hardware abstraction layer
KSERVER	2	`DServer`
KMMU	3	Memory model
KSEMAPHORE	4	Semaphores
KSCHED	5	Scheduler
KPROC	6	`DProcess`
KDUBUGGER	8	Kernel-side debug agents
KTHREAD	9	`DThread`
KDLL	10	DLLs
KIPC	11	IPC v1 and v2
KPBUS1	12	Peripheral bus controller
KPBUS2	13	Peripheral bus controller
KPBUSDRV	14	Peripheral bus driver
KPOWER	15	Power management
KTIMING	16	`DTimer`
KEVENT	17	Kernel events
KOBJECT	18	`DObject`
KDFC	19	DFCs
KEXTENSION	20	Kernel extensions
KSCHED2	21	Scheduler
KLOCDRV	22	`TLocalDrive`
KTHREAD2	24	`DThread`
KDEVICE	25	Logical and physical device drivers
KMEMTRACE	26	Memory allocations
KDMA	27	DMA framework
KMMU2	28	Memory model

KNKERN	29	Nanokernel

Tracing can be enabled using any of these methods:

- ROMBUILD kerneltrace keyword
- `User::DebugMask()`
- ESHELL trace command.

We changed the syntax of the trace command between v9.0 and v9.1 of Symbian OS to allow more trace bits to be allocated. More information is available from the Symbian Developer Library in the Base Porting Guide for EKA2 versions of Symbian OS and the C++ component reference section.

Summary

Symbian's primary development environment for applications and middleware is the EKA2 emulator. Symbian also supports development tools for target hardware, for the development of kernel-side and hardware dependent software.

The core building block of the debug architecture is the kernel debug interface. This interface is designed to support many of the development tools that are common to the embedded technology tools industry - that is, remote software on-target debuggers, hardware assisted on-target debuggers, post-mortem analysis tools, system trace and profilers.

The kernel debug architecture delivers the high level of information and execution control required to build powerful debugging and analysis tools.

Stop-mode kernel debugging and stop-mode application debugging is supported on mobile phones with JTAG ICE hardware. The stop-mode debug interface provides a method for the hardware assisted debugger to fully explore the operating system and extract information about kernel objects even while the target CPU is halted. The debugger implements Symbian OS awareness and kernel object visualizations using this interface. This improves the model of the operating system available to developer through the debugger.

The architecture supports software run-mode debuggers that are suitable for application and middleware development on mobile phones. These debuggers running on the PC talk to a proxy debug agent on the target, which is a privileged client of the kernel. This style of debugger allows system services to continue running during a debug session. However, they are not generally suitable for development of kernel-side software. EKA2 provides the necessary primitives required to implement the debug agent. The functionality provided includes the kernel event notification API to notify the debug agent of significant events, an interface to control thread execution and retrieve context, information about executable code and data sections for the relocation of symbolic debug information, and functionality to support setting breakpoints in ROM.

The architecture supports both interactive and non-interactive post-mortem analysis tools for the examination of kernel and application state at the point of thread or system death. In addition, the kernel provides primitive kernel tracing for defect diagnosis during development of kernel-side software.

In the next chapter, I will look at how Symbian OS manages a phone's power resources.