
TECHNICAL DOCUMENT

Profiler Concepts

1 What is profiler?

Profiler is an instrument used to determine:

- How many times an event occurred
- How much time did the event last
- What were the minimum, maximum and average event durations
- What's the period (time) between consecutive occurrences of the event – minimum, maximum and average
- Sequence of various events

1.1 What is an Event?

An Event can be any of these:

- Writing a variable with a certain value – *data profiling*
- Execution of a program function – *execution profiling*
- Activation of an OS task – *task profiling*
- Auxiliary event measured by on an AUX line – *AUX profiling*

1.2 What kind of a tool is required for profiling?

To capture all events, the tool must provide:

- real-time trace capability
- high precision time measurement
- a large enough recording buffer to capture the required sequence of events

1.2.1 How long (time) can the profiler session be?

This depends on several factors:

1.2.1.1 Trace data from the CPU

can be generated at a high bandwidth. On an ICE system, this can exceed 400MB/s, on a modern, wide OCT even more. If this data is unfiltered, it will quickly fill any trace buffer.

On an ICE system, *qualifiers* (online address comparators) are used to record only events of interest and discard all other bus activity.

On an OCT system, the OCT qualifier is configured to report only program trace for example, or just data accesses to a certain variable.

1.2.1.2 Trace buffer size

If trace data bandwidth is filtered down to 50 MB/s, a 1GB trace buffer will record 20s.

1.2.1.3 Upload while sampling

If an optimal hardware configuration (including the PC) is used, over 40 MB/s can be streamed to the PC via high-speed USB2. If trace data bandwidth is filtered below this, a profiler session can last indefinitely.

1.2.1.4 OCT program flow reconstruction complexity

OCT program trace must be reconstructed by the PC. If strong compression is used, more processing is required for every trace message. If the PC can not keep up, profiler session will end. A faster PC might improve this.

1.3 Can profiler be used on all CPUs?

The CPU activity of interest must be visible to the tool. This is possible on CPUs where:

- CPU core bus is visible externally – typically older CPUs with no on-chip memory or newer 8 and 16-bit CPUs which can be put (by an ICE) into special emulation mode
- CPU's on-chip trace provides the required information – newer CPUs with on-chip trace port (ETM, Nexus, etc.)

1.3.1 On-Chip Trace (OCT)

On-chip trace is typically used on single-chip CPUs, which keep all required memory inside the CPU package, or the on-chip CPU pipeline and cache would obscure the real CPU activity.

CPU employs a compression technology to reduce the traffic over the OCT port. Where a classical ICE approach would use 70 CPU pins, the OCT port usually does fine with 16 or less. The OCT port does not show addresses and data in classical sense, but rather “send messages” which the trace tool then uses to reconstruct the activity in the CPU.

A typical program trace message is a branch message, which can say as much as “*a branch was executed after 4 sequential instructions*”. The tool must then reconstruct the program flow from its knowledge of the downloaded code and the CPU state from the previous message.

1.3.1.1 Program execution trace

can be highly compressed and therefore only a few package pins are required to stream it out. Many modern CPUs provide it.

1.3.1.2 Data trace

is less compressible and would require too many package pins to make all data accesses visible. A compromise approach is to configure the CPU's OCT module (at runtime) to show only select accesses, for example only write accesses to a certain variable. This is sufficient for data and task profiling, but unfortunately many silicon vendors choose not to implement data trace at all.

1.3.1.3 Instrumentation trace

is similar to data trace, but the trace message is explicitly generated by execution of a dedicated CPU op-code or by writing a special register. To generate instrumentation messages, the application must be modified (instrumented) to generate them at appropriate locations.

On ARM CPUs this is called **ITM**.

On CPUs with Nexus OCT port, this is called **OTM**. On most PowerPC processors this message is emitted when MMU configuration is modified and its usage is thus restricted to applicaitons who do not make use of MMU. Some newer Nexus CPUs also implement the **DQM** protocol which is, much like the ARM's ITM, dedicated to instrumentaiton trace.

The size of the data transmitted in an instrumentation message depends on the CPU. Sizes range from 8 to 32 bits.

Instrumentation trace requires less data to be transmitted over the OCT port (address is not given, just data value) and is in its nature less frequently used as a memory write. Most CPUs with program trace also implement instrumentation trace, without having to increase the number of OCT pins.

1.3.1.4 ARM Naming

- ETM = always program trace, sometimes also data trace
- DWT = data trace
- ITM = instrumentation trace
- HTM = AHB bus trace

The term ETM refers to an OCT 'Macrocell', but is for historical reasons often used to describe the entire OCT system as well as the OCT external port.

1.3.1.5 Nexus Naming

- level 1 = no trace, just debugging, usually IEEE1149 JTAG
- level 2 = program trace
- level 2+ = program trace and instrumentation trace (OTM)
- level 3 = program, instrumentation and data trace
- level 4 = level 3, plus capability to 'emulate' memory locations where the external tool supplies the data to the requested memory location.

2 Data profiling

2.1 How it works

The profiler is configured to record write accesses to a specific memory location. Whenever the location is written, the value and the time are recorded.

2.2 Requirements

The memory access (address and data value) must be visible. This is the case on a CPU with core bus visibility or a CPU with on-chip **Data Trace** or **Instrumentation Trace**.

2.3 When is it used

2.3.1 Monitoring state variables

A state variable is a (global) program variable which indicates the state of the application. It will typically assume only a small number of different values. Transition to any **state** is considered an Event and statistics is maintained for every state.

Example: state of a traffic light. For every state (Red, Green, flashing,...) the count/duration/period statistics are provided.

2.3.2 Monitoring regular variables

A regular variable can assume many different values. Every **change** is considered an Event. Only single statistic (count, period) is maintained for entire variable.

Example: readout of a temperature sensor. The A/D converter typically supplies thousands of different values, where statistics about every measurable temperature is not of interest. Profiler will show how the temperature changed over time, rate of change, etc.

2.3.3 Execution timing via instrumentation

When execution trace is not available, the application can be manually modified (instrumented) to **signal** events. This signal can be a write to global variable which is visible to the trace tool or an instrumentation message.

Example: measure duration of a function

```
char g_cSignal; // dedicated variable for signaling
void MyFunction()
{
    g_cSignal = 1; // manually added signalization that MyFunction entered
    ... function body ...
    g_cSignal = 0; // manually added signalization that MyFunction exited
}
```

Even when execution trace is available, the event of interest might not be just a function execution but two distinct points in an application, which by requirement must happen within a specified time.

Example: an external event causes an interrupt, the application must process it and take action. The duration of state '1' indicates the time between occurrence of interrupt and completion of processing.

```

char g_cSignal; // dedicated variable for signaling
void IRQ_Handler()
{
    g_cSignal = 1; // manually added signalization that IRQ occurred
    ... IRQ processing setup ...
    switch (IRQSRC)
    {
        case IRQSRC_MYEVENT:
            HandleMYEVENT();
            break;
    }
    g_cSignal = 0; // manually added signalization that IRQ exited
}
void HandleMYEVENT()
{
    ... MYEVENT processing ...
    g_cSignal = 2; // manually added signalization that the event has been processed
}

```

2.3.3.1 Advanced instrumentation

In case of possible multiple interrupt preemptions, or deeper level call stacks, more robust signaling can be used:

```

char g_cSignal = 0; // dedicated variable for signaling
#define ENTRY 0x01
#define FUNC_ID_MyFunction 0x02
#define FUNC_ID_IRQ_1 0x04
#define FUNC_ID_IRQ_2 0x06

void MyFunction()
{
    char cOldSignal = g_cSignal & ~ENTRY; // remember old value, without ENTRY flag
    g_cSignal = 1 | ENTRY; // manually added signalization that MyFunction entered
    ... function body ...
    g_cSignal = cOldSignal; // manually added signalization that MyFunction exited
}

```

The signal flow could then look like this:

Signal	Event	Function call stack
03	MyFunction entered	MyFunction
05	MyFunction preempted by IRQ_A	MyFunction - IRQ_A
07	IRQ_A preempted by IRQ_B	MyFunction - IRQ_A - IRQ_B
04	Execution returned to IRQ_A	MyFunction - IRQ_A
02	Execution returned to MyFunction	MyFunction
00	MyFunction returned	

3 Execution profiling

3.1 Requirements

Program execution must be visible. This is the case on a CPU with core bus visibility or a CPU with on-chip **Program Trace**.

3.2 When is it used

3.2.1 Identifying performance bottlenecks

In any application only a very small percentage of program functions will have a large impact on overall performance. Once these are located, optimization of their code or the algorithm can yield significant performance gain.

Profiler can be configured to profile only suspected or all functions. The results will show which functions execute frequently (and are perhaps candidates for inlining) and which take the large percentage of execution time (and should be manually optimized or revised).

3.2.2 Identify execution time deviations

Sometimes a function is expected to execute within a narrow time frame. The profiler provides minimum and maximum execution times for all invocations of the function – which can confirm this assumption or show which sequence of events lead the function to deviate from it.

Example: starting and completing an A/D conversion is expected to take no less than 5 us (which is the minimum time for CPU's ADC to stabilize) and 7 us (which is the total time we allow the function to perform all the necessary setup and cleanup actions).

3.2.3 Identify invocation period

A function might be required to execute a specified number of times every second. The profiler provides minimum and maximum times between any two consecutive invocations of the function – which can confirm this assumption or show which sequence of events caused the deviation.

Example: Checking a motion sensor is required at least once per second, but checking it more often than every 800 ms wastes energy.

3.3 Entry/Exit Mode

3.3.1 How it works

The profiler is configured to record executions of instructions at a specific function's entry and exit point. When any of these instructions is executed, the instruction address and the time are recorded.

Example: a function like this:

```
int min(int a, int b)
{
    return (a < b) ? a : b;
}
```

Yields code like this:

```

    min
    {
40002194  mr          r0,r3
40002198  mr          r3,r4
    return (a < b) ? a : b;
4000219C  cmp        7,0,r4,r0
400021A0  bclr      4,29
400021A4  mr          r3,r0
    min_EXIT_
    }
400021A8  blr
```

When this function is profiled, executions of instructions on addresses **40002194** and **400021A8** will be recorded.

3.3.1.1 Advantages

The information obtained in Entry/Exit mode is most accurate.

If the profiler hardware has access to real-time program flow (ICE bus access), hardware filtering can be used to reduce the amount of trace information considerably.

Only a few functions can be selected for profiling. If these follow the *Assumptions* below, the rest of the application can use optimizations and techniques which would break the Entry/Exit algorithm.

3.3.1.2 Disadvantages

This mode relies heavily on accurate Exit information. High compiler optimizations can obscure this and render this mode unusable.

3.3.2 Assumptions

Execution profiling analysis assumes regular function entry and exit sequence. This application:

```
void g()
{
}
void f()
{
    g();
}
```

is expected to yield this sequence:

```
f
g
g_EXIT_
f_EXIT_
```

A sequence like the one below (where exit from g() is not detected) is incorrect and the profiler will abort with an *incorrect entry/exit sequence* error:

```
f
g
f_EXIT_
```

3.3.2.1 Assembler routines

Routines written in assembly language can have an arbitrary number of return points and lack the symbolic information which would allow their automatic identification.

To be able to profile such routines, the exits can be configured manually in the profiler configuration dialog, or preferably, each exit from the routine is given a symbolic name following the exit naming convention.

Example:

```
MyRoutine:
    cmp R0,#3
    ble L1
MyRoutine_EXIT_:
    blr
L1:
    sub R0,#1
MyRoutine_EXIT_2:
    blr
```

Exits named in this manner are detected automatically.

3.3.3 Possible issues

3.3.3.1 Identifying high-level function exit points

Function entry and exit information is obtained from the download file which contains symbolic information. While function entry points are always well defined, only few compilers generate information on where a function exits (note that this is not necessarily the last byte in the function space).

To identify the exit points, the function code is analyzed at download time. Locations where exits have been identified are given an artificial symbol named <function name>_EXIT_[<index>], for example:

```
min_EXIT_
min_EXIT_2
```

If an exit is incorrectly identified, the profiler results can be incorrect, but usually the profiler session will fail due to incorrect function entry/exit sequence.

Correctness of function exit identification can be checked in the disassembly window. If a discrepancy is detected, technical support should be notified. Until a solution is provided, the exit can be specified manually in the profiler configuration dialog for the function.

3.3.3.2 Inaccurate trace

Ideally the CPU would provide accurate execution information, but this is almost never the case. One of the most common deviations is an op-code refetch after an interrupt service. In this case the CPU fetches an op-code and indicates that it is being executed, but a pending interrupt causes this instruction to be cancelled. After the interrupt returns, the instruction is fetched and executed again. This can give a false impression that the op-code was executed twice.

Example:

```
min
{
40002194 mr r0,r3 <- this op-code is interrupted by an IRQ and refetched later
40002198 mr r3,r4
```

The resulting trace then appears like this:

```
min
IRQHandler
IRQHandler_EXIT_
min
```

this for the profiler is indistinguishable from a function which would look like this:

```
void min()
{
  IRQHandler()
  min(); // recursive call
}
```

Through code analysis and understanding of the trace protocol such situations are filtered out, but locating all such deviations of the CPU is a trial and error process.

3.4 Range Analyzed Mode

3.4.1 How it works

The profiler monitors continuous stream of program flow. When PC moves from a body of one function to another, profiler considers this a call of a function or return from the current function (depending on the type of instruction which caused the change of flow).

3.4.1.1 Advantages

The information obtained in Range Analyzed mode is nearly as accurate as in Entry/Exit mode.

Quality of debug information is not as critical as in Entry/Exit mode.

3.4.1.2 Disadvantages

Full program flow access is required, which is unpractical for bus trace systems.

Line level profiling is not available.

3.4.2 Assumptions

Debug information must provide accurate enough information about function location and size.

The application must maintain ANSI stack concept. Some operating systems dismiss the stack of a task once the task is terminated, without letting it unwind. This will cause the profiler to report an error.

3.5 Range Mode

3.5.1 How it works

The profiler monitors continuous stream of program flow. When PC moves from a body of one function to another, profiler considers this a call of a function or return from the current function (depending on the type of instruction which caused the change of flow).

3.5.1.1 Advantages

Quality of debug information is not as critical as in Entry/Exit mode.

Compiler and OS optimizations do not affect the profiler.

Task switches do not need to be recorded. If data trace is not available, Range mode profiler still works.

3.5.1.2 Disadvantages

Full program flow access is required, which is unpractical for bus trace systems.

Line level profiling is not available.

If compiler tail-merge optimizations are used, the shared code is not attributed to the optimized function.

3.5.2 Assumptions

Debug information must provide accurate enough information about function location and size.

3.6 General Considerations

3.6.1 On-Chip Trace FIFO Overflows

If the application code uses dense indirect branches (return from function, call function via pointer, frequent IRQs), the OCT FIFO cannot keep up with the amount of generated messages and on-chip trace overflows are reported.

Possible solutions:

- Check if the CPU can use a wider OCT port. This is configurable on some CPUs
- Check if the OCT clock can be increased or double data rate / half rate clocking can be used. This is configurable on some CPUs.
- Check if the Profiler configuration dialog provides ‘Stall CPU to avoid overflows’ option (or an option with equivalent meaning). If the option is available and checked, the internal on-chip trace logic will stall the CPU until there is free space available for a new message in the on-chip trace FIFO.

Note that depending on the CPU and the application, the run time performance can be affected by this option significantly. If absolutely no impact on real-time execution is required, then alternative solutions must be used.

This option is available depending on the microcontroller architecture.

- Check if compiler optimizations are the cause. Especially when optimizing for size, the compiler can move typical function prolog and epilog code in a separate routine – but this triples the number of direct and indirect messages for a simple function.
- Use code instrumentation as explained in section *Execution timing via instrumentation*.

4 Task profiling

4.1 Multi-tasking concepts

In a multitasking environment, usually provided by an operating system, apparently multiple parallel operations are executed. Every such operation is called a *Task*. In reality only one task is active at a time. The operating system decides which task should be run based on task priorities, states, synchronizations etc.

4.1.1 Task control block (TCB)

For a task to be independent of other tasks, it keeps its own set of registers. When the task is executing, these are the regular CPU core registers.

When the OS *deactivates* a task, all registers are saved to a structure called *task control block* which is kept in global memory of the OS. The registers include the program counter and the stack pointer.

When the OS *activates* the task again, the registers previously saved in the task control block are restored to CPU registers and execution resumes at the point where it was previously interrupted.

4.1.2 Task creation and termination

An application still starts with a single task - at function 'main', but can then *create* more tasks. To create a task, it calls into the OS and specifies the *task control function* and usually the stack size for the task. This function behaves just like function 'main'. When it exits, the task *terminates*.

```
void main()
{
    OSCreateTask(Task1ControlFunc, 100); // create a task, specify control func and stack size

    // perform main task activities
    while (WorkToBeDoneInMain)
    {
        DoMainWork();
    }
}

void Task1ControlFunc() // this function is called first after task is created
{
    while (WorkToBeDoneInATask)
    {
        DoTaskWork();
    }
}
```

The *OSCreateTask* will:

- Create a new task control block
- Set TCB's initial value of the program counter to address of *Task1ControlFunc*
- Allocate 100 bytes of stack space
- Set TCB's initial value of the stack pointer to the address of allocated stack
- Put the (pointer to) TCB in the list of task for the *scheduler* to process.

When *Task1ControlFunc* ultimately exits, the OS will perform a cleanup:

- Remove the TCB from the scheduler list
- Free the allocated stack space
- Free the TCB structure

Even though the real task *lifetime* exists between a call to `OSCreateTask` and final cleanup, it is usually considered to be the progress from the control function entry and exit. This function is entered and exited **only once**.

4.1.3 Task activation and deactivation

To execute multiple tasks in ‘parallel’, the operating systems *switches* between tasks. Typically every task will only be given a few milliseconds to run in order to achieve an illusion of parallelism.

When a task is allowed to run, it is *activated* and considered active. When the OS scheduler decides that a different task should run, this task is *deactivated*.

In the *lifetime* of a task, it can be activated and deactivated **many times**.

4.2 Detecting task events

4.2.1 Detecting task creation and termination

In a static embedded OS (number of tasks known at compile time), each task usually has a dedicated control function. In this case the creation and termination can be observed by *execution profiling* the task control function.

The gross time will match the total time spent executing the task.

The count will match the number of times the task has been created. It will usually be just one.

4.2.2 Detecting task activation and deactivation

Determining the active task is the key point to profiling in a multitasking environment. Besides giving information about task activation frequency, run time etc. it is mandatory to allow execution profiling.

The OS scheduler always keeps track of the active task in a global OS variable, usually a pointer to the active task’s control block.

By *data profiling* this variable, both the task profiling and the execution profiling are made possible.

If the CPU does not provide *data trace*, the value of this variable (or the task ID) must be *signaled* using *instrumentation trace*.

4.2.3 Detecting other OS events

If the OS provides signaling for other events, like the service it is currently executing, regular data profiling is used to capture these signals.

4.3 Execution profiling in a multitasking environment

Because the OS scheduler can interrupt a regular program flow, the *execution profiling assumption* no longer stands. This code:

```
void main()
{
    OSCreateTask(Task1ControlFunc, 100); // create a task, specify control func and stack size
    // perform main task activities
    while (WorkToBeDoneInMain)
    {
        DoMainWork();
    }
}
void Task1ControlFunc() // this function is called first after task is created
{
    while (WorkToBeDoneInATask)
    {
        DoTaskWork();
    }
}
```

Can very well generate this event sequence (leading number is the timestamp of the event):

```
10 DoMainWork
20 DoTaskWork
30 DoMainWork_EXIT_
40 DoMainWork
50 DoTaskWork_EXIT
```

this will cause a multitasking unaware profiler to abort with invalid function entry/exit sequence error.

If however the active task is traced along, the same sequence now looks like this:

```
5 TASK: 0 // ID of the main task
10 DoMainWork
15 TASK: 1 // ID of Task1ControlFunc's task as it got activated
20 DoTaskWork
25 TASK: 0 // ID of the main task
30 DoMainWork_EXIT_
40 DoMainWork
45 TASK: 1 // ID of Task1ControlFunc
50 DoTaskWork_EXIT
```

The execution profiler can now look at each task separately and the regular entry/exit sequence is used again:

```
TASK: 0
10 DoMainWork
Suspend from 15-25
30 DoMainWork_EXIT_
40 DoMainWork
Suspend from 45-

TASK: 1
20 DoTaskWork
Suspend from 25-45
DoTaskWork_EXIT
```

This 'rearrangement' also affects the timestamps and considers the suspended time.

4.4 Possible issues

Virtually the only issue is identification of task activation.

If *data trace* is available, it's just a matter of identifying the global OS variable which keeps the active task identification.

If data trace is not available, the OS itself must provide active task signaling using *instrumentation trace*, which can be realized only by the OS vendor.

Some operating systems provide a mechanism called *pre/post task hook*, where the application registers a function with the OS, which the OS will call whenever a task switch occurs:

```
void PreTaskHook(TaskType TaskID)
{
    GenerateInstrumentationMessage(TaskID); // generate an instrumentation
                                           // message with the value given in the parameter
}
```

4.5 OSEK/ORTI

The internal layout of an OSEK operating system is described by an ORTI file. This file describes amongst other things how many tasks there are (all tasks are known at compile time) and how to determine which task is active.

Per ORTI standard, a **RUNNINGTASK** ‘object’ defines the active/running task. This example defines two tasks named **Task_1** and **Task_2**, plus a state where no task is running – **NO_TASK**.

```
IMPLEMENTATION OS_XY {
  OS {
    ENUM [
      "NO_TASK" = 0xFFFF,
      "Task_1" = 0,
      "Task_2" = 1,
    ] RUNNINGTASK, "Running Task Identification";
    ...
  }
}
```

Later in the file (in the *information section*), the location of the RUNNINGTASK object is defined.

```
OS xx {
  RUNNINGTASK = "osActiveTaskIndex";
}
```

These two definitions mean:

- To find out which the active task is, evaluate "osActiveTaskIndex"
- If the evaluation yields a value of **0**, **Task_1** is active, if it's a **1**, **Task_2** is active, if it's **0xFFFF**, **NO_TASK** is active.

When CPU is stopped, the active task is obtained by reading the `osActiveTaskIndex` variable.

When CPU is running (during profiling session), value of the `osActiveTaskIndex` is obtained by tracing write accesses to it.

4.5.1 Possible issues

In the above example, the variable `osActiveTaskIndex` keeps the ID of the active task. As far as the OS is concerned, this variable is just overhead. The optimal concept for it is to:

- Have a TCB for every task
- Have a global pointer to the active TCB

Task ID is then just a member in the TCB structure:

```
struct TCB
{
  unsigned short ID; // ID of the task
  unsigned long StackBase;
  unsigned long PC;
  unsigned long SP;
  ...
};
TCB tcbTask_1, tcbTask_2; // TCBS for the two tasks
```

```
TCB * pActiveTCB; // global variable pointing to the active task's control block
```

Switching to a different task is now just a matter of setting the pointer to point to a different TCB. However the following problems can occur:

4.5.1.1 Untraceable ID problem

To get the task ID, some OSes report this in the ORTI file:

```
OS xx {
    RUNNINGTASK = "pActiveTCB->ID";
    ...
}
```

This can be evaluated when CPU is stopped, but when the `pActiveTCB` is changed at runtime, the trace will only see the new value of `pActiveTCB` but not the value of the ID.

Solution 1

Have the OS report the ID via global variable as per usual practice. This can sometimes be forced by OS configuration option, or by a fix provided by the OS vendor.

Solution 2

The ORTI file must be modified (manually or by OS vendor fix), where the value of the pointer identifies the task:

```
IMPLEMENTATION xx {
    OS {
        TOTRACE ENUM [
            "NO_TASK" = 0,
            "Task_1" = "&tcBTask_1",
            "Task_2" = "&tcBTask_2",
        ]
    }
    OS xx {
        RUNNINGTASK = "pActiveTCB";
        ...
    }
}
```

In this case a change of the value of `pActiveTCB` is traced and the value of the pointer can be matched directly to the task.

This can lead to the next problem:

4.5.1.2 Task identification requires more bits than data/instrumentation trace supports

If the active task is identified by value of a pointer, full pointer value must be visible. Problems arise when:

- Tracing a 16-bit pointer on 8-bit CPU, where the value is transmitted in two (not necessarily consecutive) writes.
- Data trace is not available, and instrumentation trace is used for signaling, but the instrumentation trace is only 8-bits wide.

Solution 1

Have the OS report the ID via global variable as per usual practice. This can sometimes be forced by OS configuration option, or by a fix provided by the OS vendor.

Solution 2

Implement further instrumentation to transmit all 32-bits through several instrumentation messages. Since this incurs much more time and code size overhead than the recommended solution, it should be avoided if possible.

An extension to OSEK and ORTI has been proposed which would allow the OS to inform the profiling tool via ORTI file on how the task switches are signaled. Contact your OS vendor to verify if these extensions have been implemented in your version.

4.6 Custom operating system

If your operating system is not explicitly supported by winIDEA, support can be added by manually generating an ORTI file, which describes the OS – especially the active task identification.

This example assumes that there are 2 tasks in the OS and that a global variable *g_byActiveTaskID* always holds the value of the active task (0 for no task, 1 for "Task1" and 2 for "Task2")

```
IMPLEMENTATION MyORTI {
  OS {
    ENUM UINT8 [
      "NO_TASK" = "0",
      "Task1" = "1",
      "Task2" = "2"
    ] RUNNINGTASK, "Running task";
  };
};
OS MyOS {
  RUNNINGTASK = "g_byActiveTaskID";
};
```

Detailed ORTI specification is available at: <http://portal.osek-vdx.org/files/pdf/specs/>

5 Common CPU trace capabilities

CPU	Program Trace	Data Trace	Instrumentation trace
HC08	ICE	ICE	
S12(X)	ICE	ICE	
e200z6 / MPC555x/556x e200z3 / MPC553x	Nexus	Nexus	OTM 8-bit
e200z1 / MPC551x	Nexus		OTM 8-bit
e200z0 / MPC560x	Nexus		OTM 8-bit
e200z3z335 / MPC563x	Nexus		OTM 8-bit
e200z4 / MPC564x e200z7 / MPC567x	Nexus	Nexus	OTM 8-bit DQM 32-bit
MCF5xxx	OCT		WDDATA 8/16/32 bit
PPC4xx	RiscWatch		
ARM7/9	ETM	ETM	
Cortex M	ETM	DWT	ITM 32-bit
Cortex R/A	ETM	ETM	
78k	ICE	ICE	
V850 Fx3	OCT	OCT	
V850 Fx4	Nexus	Nexus	OTM
R8C	ICE	ICE	