User Trace Port –
Trace Port Emulation on Processors without on-chip Trace Hardware

How can I verify Functional/Timing Correctness on Processors without on-chip Trace?

Armin Stingl, iSYSTEM AG

1 Introduction

An important aspect during the development of software, especially for embedded systems, is the possibility to observe the program execution flow and timing behavior of the application. Such functionality, typically referred to as “Trace”, is the basis for analyzing and verifying the functional and timing correctness of the application. These kind of tests are often a requirement to meet Safety and/or Security standards.
Providing trace functionality in embedded systems requires, on one hand, the manufacturer of the processor to implement the on-chip hardware to generate, collect and output the relevant trace data. On the other hand, the tool manufacturer has to provide the hardware and software infrastructure for recording the trace output data and then re-constructing and displaying the program execution.

There are two conceptual types of program trace:

- Instruction Trace
  Allows reconstructing the program flow down to each individual (assembly) instruction.
- Execution Profiling
  Allows reconstructing how much time the CPU spends within a specific program area such as a function or OS task as well as the order of execution of each function or task.

Whereas Instruction Trace requires a relatively complex on-chip trace logic and a dedicated high-bandwidth (parallel) trace output port, function/task profiling has rather low bandwidth requirements and thus can be accomplished also via a serial communication and/or by the instrumentation of the program source code.

2 OS Task Profiling

A key benefit of using a Real-Time Operating System in an embedded application is the ability to extend functionality of the application by simply adding more OS tasks. Thus, an embedded application may comprise 50 or more tasks. But how can I make sure, the CPU utilization is well balanced among all the tasks and timing requirements can still be met?
OS task profiling addresses these issues and allows to visualize and analyze the application behavior on OS task level by means of two different visualization options:

- Profiling Statistics
- Profiling Timeline

2.1 Profiling Statistics

Statistical profiling analyzes how much CPU time (relative and absolute) was spent in the various OS tasks of the application. This information can be used to evaluate the CPU utilization.

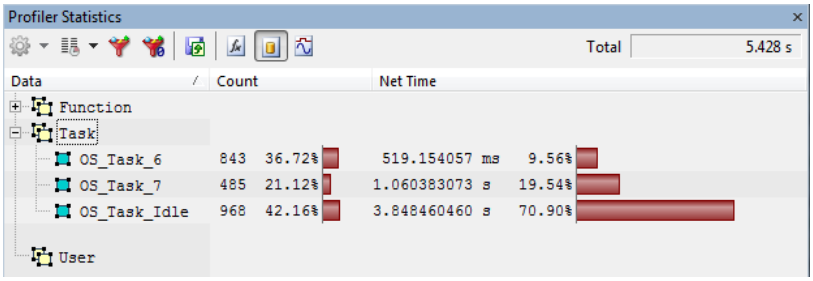

Figure 1 Profiler Statistics Example

In the example above the CPU spends about 71% in the Idle task of the OS, thus the CPU is quite under-utilized.

2.2 Profiling Timeline

Timeline profiling not only analyses the time, the CPU has spent within a certain task, but also tracks the execution order of all tasks.

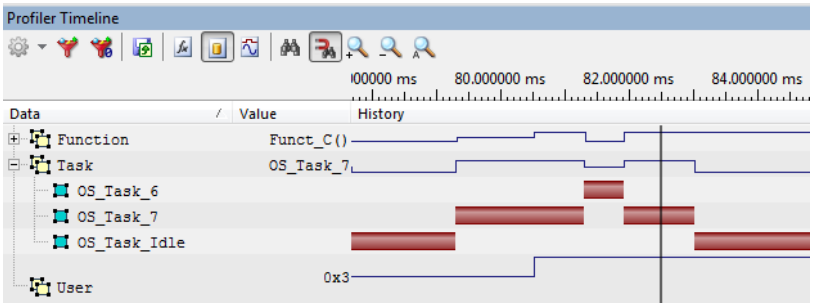

Figure 2 Profiler Timeline Example

In the example above, the OS Task 7 has been pre-empted by the higher-priority task 6 for about 0.5ms.

3 Trace Solutions for OS Task Profiling

3.1 Existing Solutions

For OS task profiling there are currently mainly two concepts used in the embedded processor market:

- Nexus Debug Standard (Nexus 5001 ™)
  Ownership Trace Message (OTM)
  CPU writes task ID into Ownership Trace register
  OTM including task ID output via parallel trace port (typically 6 or 10-pin port)

- ARM Cortex™ CoreSight / ITM
  Instrumentation Trace Macrocell (ITM)
  CPU writes task ID into ITM Stimulus register
  Stimulus register contents output via serial trace port (Serial Wire Viewer)

For both concepts the debug tool appends a time-stamp to each trace message when sampled from the device pins to allow for further timing analysis. All trace messages are stored in a trace file and subsequently analyzed and displayed in various formats.

3.2 What if the Processor does not have any on-chip Trace Hardware?

In order to save chip size and/or pins, many processor manufacturers do not implement any on-chip trace hardware. Nevertheless, these processors may still be used in conjunction with Real-Time Operating Systems and application development may still require a reliable timing analysis by means of OS task profiling.
This is a situation where flexible and powerful debugging tools can make the difference and can offer a user-friendly solution to overcome such hardware restrictions.

The company iSYSTEM has recently developed a flexible and user-friendly method for adding profiling capabilities to processors which do not offer on-chip trace hardware such as Nexus or ITM. As this solution is highly adjustable by the user according to the individual needs and the processor capabilities, this feature is called “User Trace Port”.

The concept is fairly straight-forward:
On the application or development board the target device is replaced by a special adaptor, which expands the original device to a package option with higher pin-count (e.g. 100-pin QFP to 144-pin QFP). This introduces an additional General-Purpose I/O (GPIO) port to the system, which is not used by the application software (as not existent on the original device). This spare GPIO port can now be utilized to output trace messages. Instrumentation code is inserted into the original source code, which writes trace messages to this User Trace GPIO port.

Figure 3 shows the User Trace adaptor for a VH850Vx4L device, expanding the original package to a 144-pin package.

Figure 3 User Trace Adaptor for VH850Vx4L

The User Trace port is also routed to the debug connector, and thus allows the debugger hardware (iSYSTEM “Bluebox”) to record the User Trace messages. iSYSTEM’s Integrated Development Environment & Debugger software (winIDEA GUI) can be configured to recognize various formats of User Trace messages. The Analyzer/Profiler window within winIDEA finally displays the program execution profile in the same fashion as with Nexus or Cortex ITM trace hardware.

Figure 4 shows a complete iSYSTEM User Trace system consisting of target application board, User Trace adaptor and iC5000 BlueBox.

Figure 4 Complete iSYSTEM User Trace System

3.3 Why do we use a GPIO Port?

The following criteria have been taken into consideration for the selection of a GPIO port as trace output port:

- Often all on-chip serial communication interfaces, such as UART, are all in use by the application.
- An unbuffered serial communication interface (e.g. UART) may lead to trace data overflow conditions.
- Unbuffered trace output yields a better Time-Stamp Accuracy.

Thus, a spare GPIO port seems to be the most suitable solution for the User Trace port.

4 Code Instrumentation

As the presented profiling concept is based on sending out trace messages via a GPIO port, the application source code needs to be instrumented. Code must be inserted that extracts the relevant information (e.g. ID of OS task to be executed next) and writes this data to the trace port in the proper format. iSYSTEM provides all necessary instrumentation code as C macros in dedicated C header files.

The macro

```
#define isystem_profile_task(ID) isystem_profile_write(ipfid_Task, ID)
```

writes an OS Task message to the trace port.

The following two code examples illustrate the instrumentation of the application source code. Typically, OS ports provide so-called hooks to add user-specific functions to OS services, such as the context switch (i.e. switch to the next OS task). The two widely used Real-Time Operating Systems, uC/OS-II and OSEK-OS are given as examples.

uC/OS II

```
#include <isystem_profile.h>

/*============================================================
 Send out Task Message containing the ID of task, which is ready
 and has highest priority.
 Message is sent out just before context switch to task.
 ============================================================*/
void App_TaskSwHook(void)
{
   isystem_profile_task(OSTCBHighRdy->OSTCBId);
}
```

OSEK / AUTOSAR OS

```
#include <isystem_profile.h>

/*============================================================
 Send out Task Message containing the ID of task, which is ready
 and has highest priority.
 Message is sent out just before context switch to task.
 ============================================================*/
void PreTaskHook(void)
{
   isystem_profile_task(GetTaskID());
}
```

To enable the winIDEA Analyzer to display the task names instead of the raw task ID numbers, macros need to be defined in a dedicated header file that assign a task name to each ID.

```
      // ================================
      // Sample Task ID Macro Definition:
      //      OS Task Name        Priority
      // ================================
      #define OS_Task_Idle          0
      #define OS_Task_6             6
      #define OS_Task_7             7
```

If an ORTI file for the OSEK OS is available, it can be used to provide the task ID definitions to winIDEA. Thus, the above profiler configurations are not needed.

The macro `isystem_profile_task` uses the following macros for the actual write accesses to the trace port.

```
// ========================================================
// Target Device: V850Fx4L 144-pin (16-bit trace port P25)
// --------------------------------------------------------
// 1. Clear Trace Port to 0x0000.
// 2. Rising edge on P25[15], i.e. MSB => Sample Clock
//    Set Message ID, i.e. this is a task message.
//    Set Message Value, i.e. OS Task ID.
// 3. Clear Trace Port back to 0x0000.
// ========================================================
#define P25   *(volatile unsigned short *)0xFF400064
#define isystem_profile_write(ID, VAL)       \
        { P25 = 0;                           \
          P25 = (1<<16) | ((VAL) << 2) | ID; \
          P25 = 0; }
```

The trace port „Clear – Set – Clear“ approach is used for making the sequence interrupt-safe. Even if the current sequence is interrupted, the new sequence (of the interrupt handler) will also start with clearing the trace port to 0 before writing a new trace output value.

Trace Port Signaling:

The trace port can be 8, 16 or 32 bits wide, depending on the individual processors in use. The V850Fx4L processor for instance offers 16-bit ports. For User Trace, these 16 pins are allocated as follows:

15	14 2	1 0
CLK	VAL[12:0]	ID[1:0]

CLK: Trace Message Sampling Clock (on rising edge)
VAL: Trace Message Data, e.g. OS Task ID
ID: Trace Message Identifier, e.g. OS Task Message

5 Other supported Profiling Options

So far, the User Trace Port concept has been presented for OS task profiling, however, it can also be used for function profiling and for sending out user-defined data.

5.1 Function Profiling

Similar to OS task profiling, function profiling allows for analyzing the timing behavior of individual functions of the application. Each function to be included into the profiling needs to be instrumented, i.e. the function entry and the exit point(s) are marked with macros that generate a special trace message.

Function Profiling Instrumentation:

```
#include <isystem_profile.h>

void Funct_A(void)
{
    isystem_profile_func_entry(Funct_A);

    // function body …

    isystem_profile_func_exit();

}
```

5.2 User Data Profiling

Finally, a dedicated User Trace message is available that allows sending out any arbitrary data that is of interest to the user.

Data Profiling Instrumentation:

```
#include <isystem_profile.h>
// ========================================
// Function returns ADC Result and
// sends out ADC result via User Trace port.
// ========================================
uint16_t ReadADC(void)
{
    unsigned short AdcData = ADC0->RESULT_REG;
    isystem_profile_user(AdcData);
    return AdcData;
}
```

The sample Profiler Timeline below illustrates the combined display of OS Task, Function and User Data Profiling.

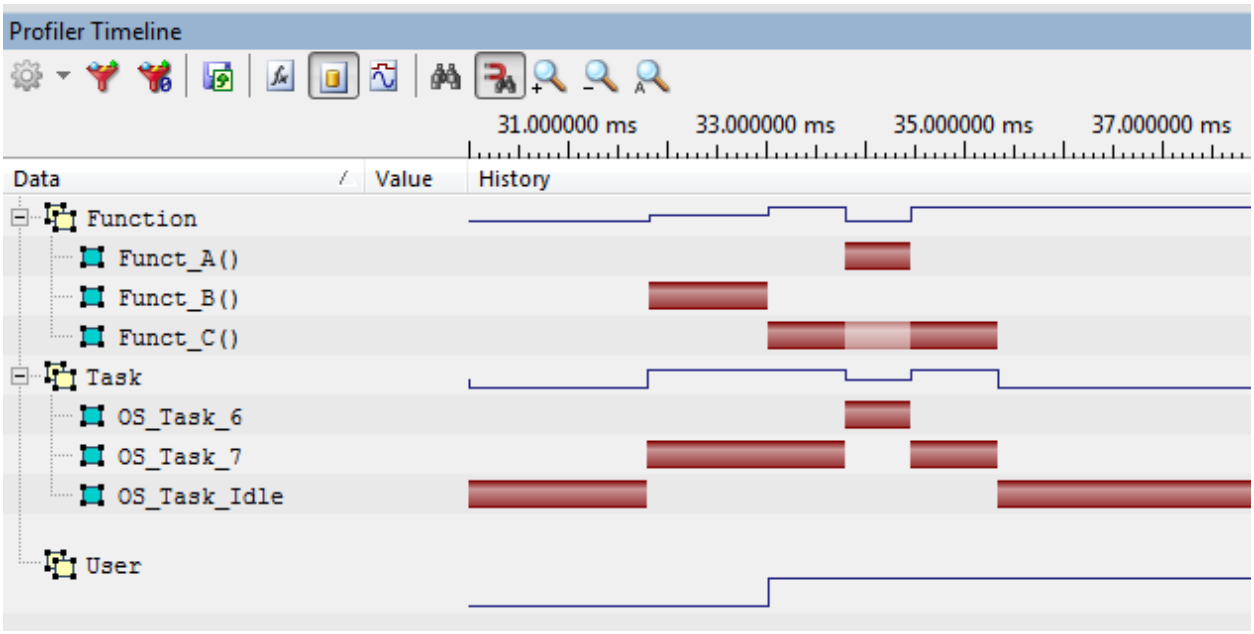

Figure 5 Profiler Timeline Example with OS Task, Function and User Data Profiling

OS_Task_7 calls Funct_B() and Funct_C().
OS_Task_6 pre_empts OS_Task_7, thus Funct_C() is interrupted to execute Funct_A() within OS_Task_6.
User Data is updated by Funct_C().

6 Supported Processors

User Trace adaptors are currently available for the following processor families:

- V850Fx4L
- RH850

In addition, the User Trace port feature may also be relevant for:

- PowerPC Pictus (MPC560xP)
- Cortex-M0 and M0+ based micro controllers