User Trace Port

Table of Contents

1 Introduction

The iC5000 CPU interface supports on-chip debug (OCD) and on-chip trace (OCT) interfaces.

Some CPUs however provide no OCT port and no trace is possible there. To alleviate this limitation, a free port on the emulated CPU can be used to rebuild the trace port. If the CPU has no free ports, a bigger package device can be used either in a redesigned target system or on an emulation adapter.

The port must be manipulated by the target application – the code must be instrumented on appropriate positions. iSYSTEM provides instrumentation macros for this purpose.

2 Emulation Adapter

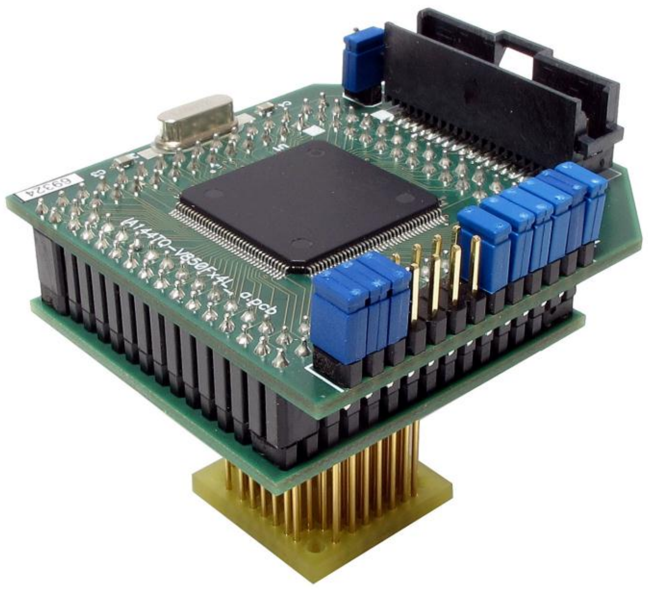
The emulation adapter is designed to connect directly to the iC5000 without any additional adaptation. The CPU port used for tracing is connected to the iC5000 connector.

If the bigger CPU package consumes more power than the original CPU and the target ECU cannot supply sufficient power, an optional external power supply (provided by iSYSTEM) can be attached to the emulation adapter.

Per default the power connection pins are populated with jumpers which bridge the target power supply to the CPU.

The J11 jumper can be removed to disconnect the target reset line.

The emulation adapter uses standard iSYSTEM target pinout, to which a target solder or wire adapter can be connected.



3 Application Instrumentation

To signal a value over a port, the application must write the data to the designated port and provide some kind of clock signal. iSYSTEM provides macros for different CPU architectures and emulation adapters, defined in the ***isystem_profile_HAL.h*** file.

Example

```
#define P25   *(volatile unsigned short *)0xFF400064
#define PM25  *(volatile unsigned short *)0xFF400364

// port initialization
#define isystem_profile_initialize() { PM25 = 0x0000; }
// set port value
#define isystem_profile_write(ID, VAL) { P25 = 0; P25 = 0x8000 | ((VAL)<< 2) | ID; P25 = 0; }
```

The ***isystem_profile_initialize*** must be called before any other profiling function – typically on ***main*** entry.

```
void main()
{
  isystem_profile_initialize();
  ...
```

The ***isystem_profile_write*** is used by other profiling macros.

3.1.1 Signal encoding

To allow signaling of different type of information (functions, tasks,…), the signaled value is split between data and ID fields:

DATA								ID	
DATA n	…	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0	ID1	ID0

A typical ID configuration would use two bits and use this encoding:

ID	DATA representation
00	Function execution
01	Task ID
10	IRQ ID
11	User data

On an 8-bit user port, this leaves 6 bits for data, which means that 64 different values can be signaled. This means:

- 63 functions (value 0 is reserved to signal exit from any function)
- 64 tasks
- 64 ISRs
- 64 user data values

Note: the iC5000 supports a maximum port width of 12 bits.

3.2 Functions

To profile functions, each such function must be instrumented immediately on entry and on exit from it like this:

Example

```
void UserFunction1()
{
  isystem_profile_func_entry(UserFunction1);
… // function body
  isystem_profile_func_exit();
}
```

3.2.1 Function definition file

The ***isystem_profile_func_entry*** macro generates an event on the trace port. It reports a value specified as parameter. To facilitate the usage of these values, the macro uses a convention where a constant using **ipf_** prefix ahead of function name must be defined.

All these constants should be defined in file ***isystem_profile_functions.h***. This way winIDEA will be able to resolve the values observed on the trace port to the real functions.

Example

```
#define ipf_UserFunction1 1
#define ipf_UserFunction77 2
#define ipf_IRQHandler 3
```

3.3 OS Events

A typical RTOS application will use a dozen tasks and IRQs. When a task is activated, the OS code will write the ID to a specific global variable. winIDEA would normally use data trace to observe these writes, but in this case the OS code must be modified manually to signal the task ID via the user trace port.

Some OSes will provide *hook* functions which are called on task activation and deactivation. These hooks can be used to signal the task ID.

Example

```
void PreTaskHook(char cTaskID)
{
  isystem_profile_task(cTaskID);
}
```

IRQs are best captured in the IRQ handler function and instrumented on entry and on exit.

Example

```
void IRQ_Handler(char cIRQ)
{
  static char s_cCurrentIRQ = 0;
  char cOld = s_cCurrentIRQ;
  s_cCurrentIRQ = cIRQ;
  isystem_profile_irq(cIRQ);
… // IRQ handler body
  isystem_profile_irq(cOld);
  s_cCurrentIRQ = cOld;
}
```

Note: in case an ORTI file is used to describe the OS, the type of signaling and the signaled values can be made a part of it. This way no further profiler configuration for the OS is required.

3.4 User data

Any additional data of interest to the user can be transmitted using the ***isystem_profile_user*** macro.

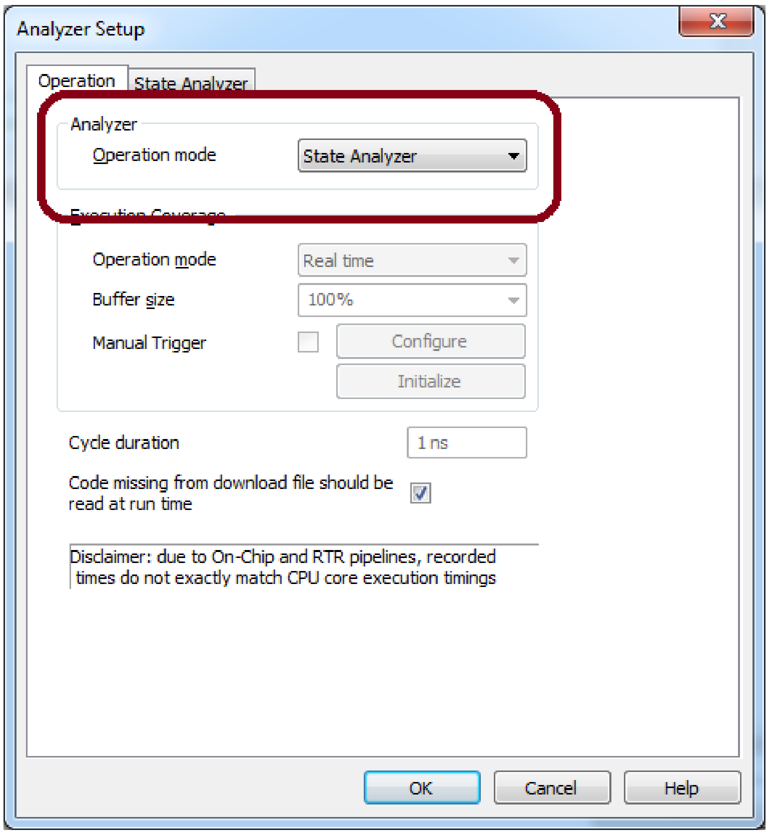
Example

```
char ReadADC()
{
  char cADCValue = ADC0_VAL;
  isystem_profile_user (cADCValue);
  return cADCValue;
}
```

4 winIDEA configuration

4.1 Trace Configuration

In ***Hardware/Analyzer Setup…*** select the **State Analyzer** as ***Operation mode***.



Analyzer Setup

Operation | State Analyzer

Analyzer

Operation mode: State Analyzer

Execution Coverage

Operation mode: Real time

Buffer size: 100%

Manual Trigger: ☐ Configure Initialize

Cycle duration: 1 ns

Code missing from download file should be read at run time: ✓

Disclaimer: due to On-Chip and RTR pipelines, recorded times do not exactly match CPU core execution timings

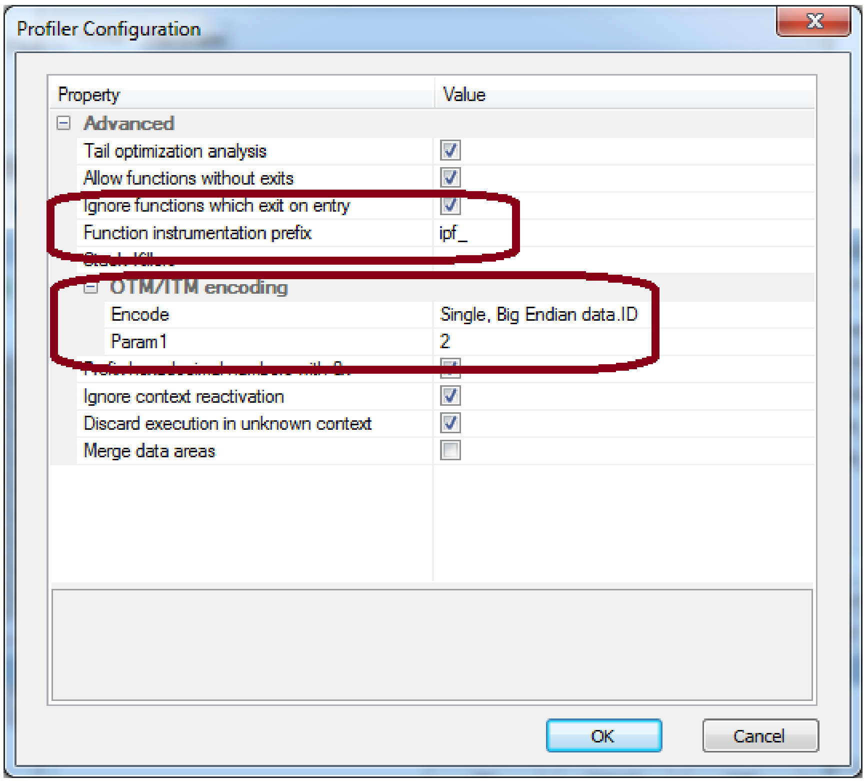
OK Cancel Help

4.2 Profiler configuration

4.2.1 Signaling specification

In ***Profiler/Advanced***, ensure that

- the ***Function instrumentation prefix*** is set to **ipf_** (default).
- The ***OTM/ITM encoding*** is set accordingly to the encoding used by the application.



Profiler Configuration

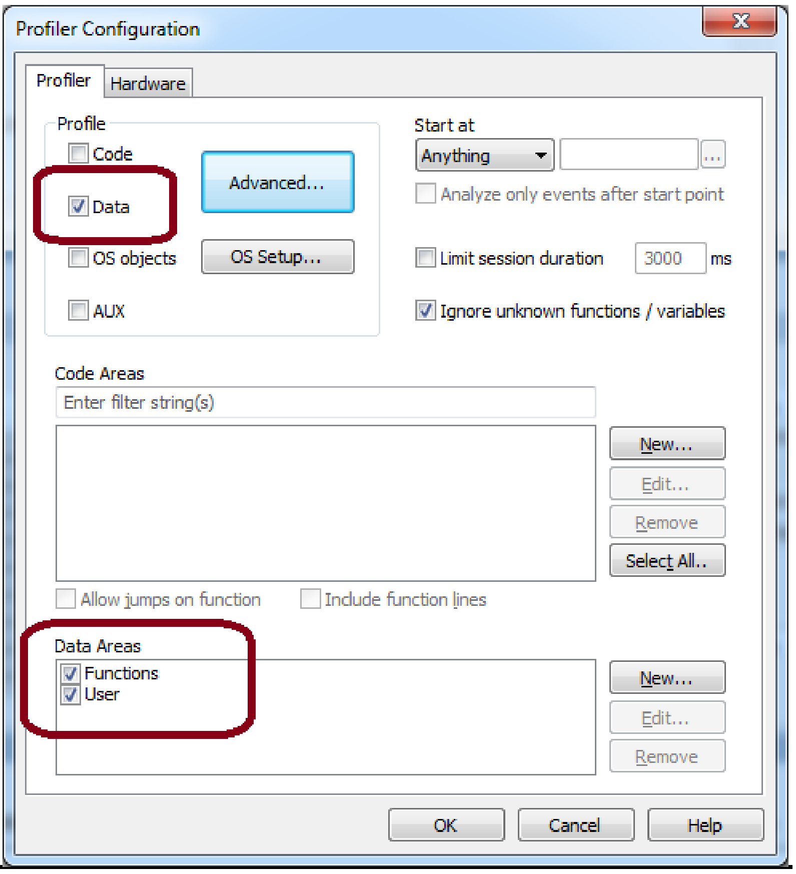
Property	Value
Advanced	
Tail optimization analysis	✓
Allow functions without exits	✓
Ignore functions which exit on entry	✓
Function instrumentation prefix	ipf_
Stack Killer	
OTM/ITM encoding	
Encode	Single, Big Endian data.ID
Param1	2
Prefix hexadecimal numbers with 0x	✓
Ignore context reactivation	✓
Discard execution in unknown context	✓
Merge data areas	☐

4.2.2 Area definition

In Profiler configuration define data areas for all the signaling types used.

Next, define each area accordingly.

If the ORTI file was adjusted accordingly, the OS variables do not need to be defined explicitly. Just enable the ***OS Objects*** option to profile them.



Profiler Configuration

Profiler | Hardware

Profile: ☐ Code; ☑ Data; ☐ OS objects; ☐ AUX

Start at: Anything

☐ Analyze only events after start point

☐ Limit session duration 3000 ms

☑ Ignore unknown functions / variables

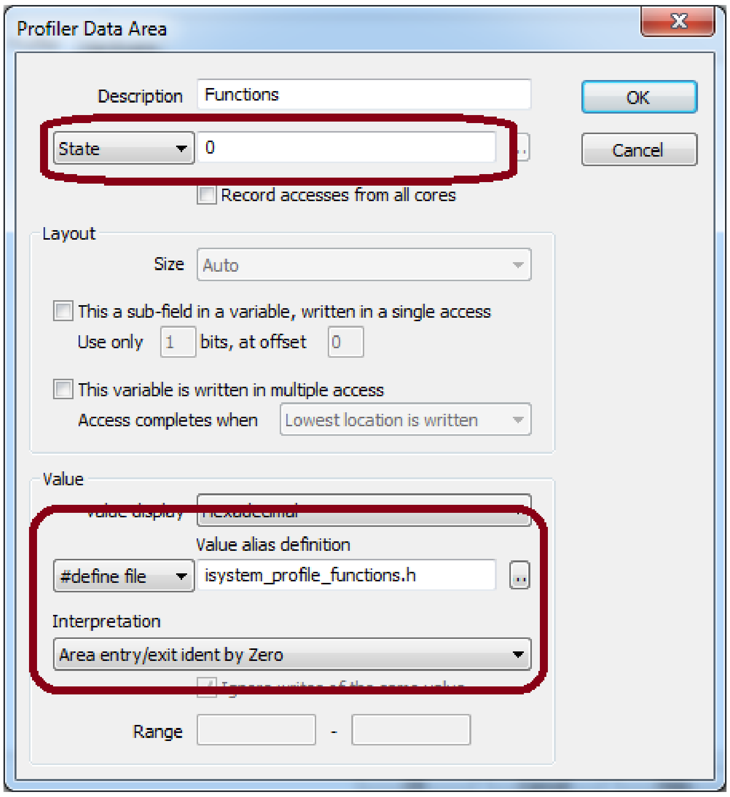
Code Areas: Enter filter string(s)

☐ Allow jumps on function ☐ Include function lines

Data Areas: ☑ Functions ☑ User

Functions

- use the signaling ID **0**.
- The function constants are defined in file **isystem_profile_functions.h**
- This are signals entries and exits of functions.


Profiler Data Area

Description: Functions

State ▾ 0

☐ Record accesses from all cores

Layout

Size: Auto

☐ This a sub-field in a variable, written in a single access

Use only 1 bits, at offset 0

☐ This variable is written in multiple access

Access completes when: Lowest location is written

Value

Value display: Hexadecimal

Value alias definition

#define file ▾ isystem_profile_functions.h

Interpretation

Area entry/exit ident by Zero

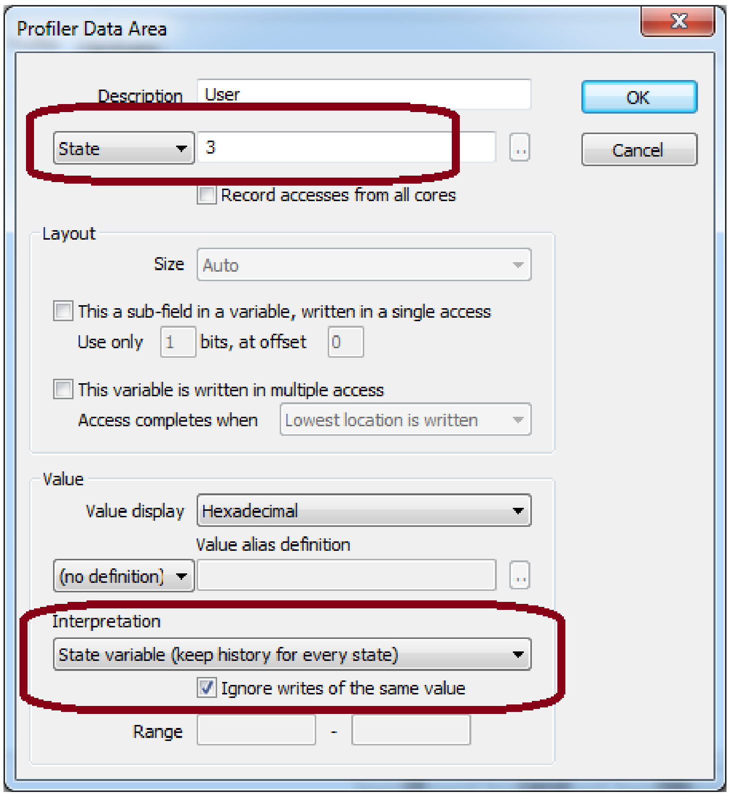
☑ Ignore writes of the same value

Range: -

OK
Cancel

User data

- Uses signaling ID **3**
- Is in this case a state variable


Profiler Data Area

Description: User

State ▾ 3

☐ Record accesses from all cores

Layout

Size: Auto

☐ This a sub-field in a variable, written in a single access

Use only 1 bits, at offset 0

☐ This variable is written in multiple access

Access completes when: Lowest location is written

Value

Value display: Hexadecimal

Value alias definition

(no definition) ▾

Interpretation

State variable (keep history for every state)

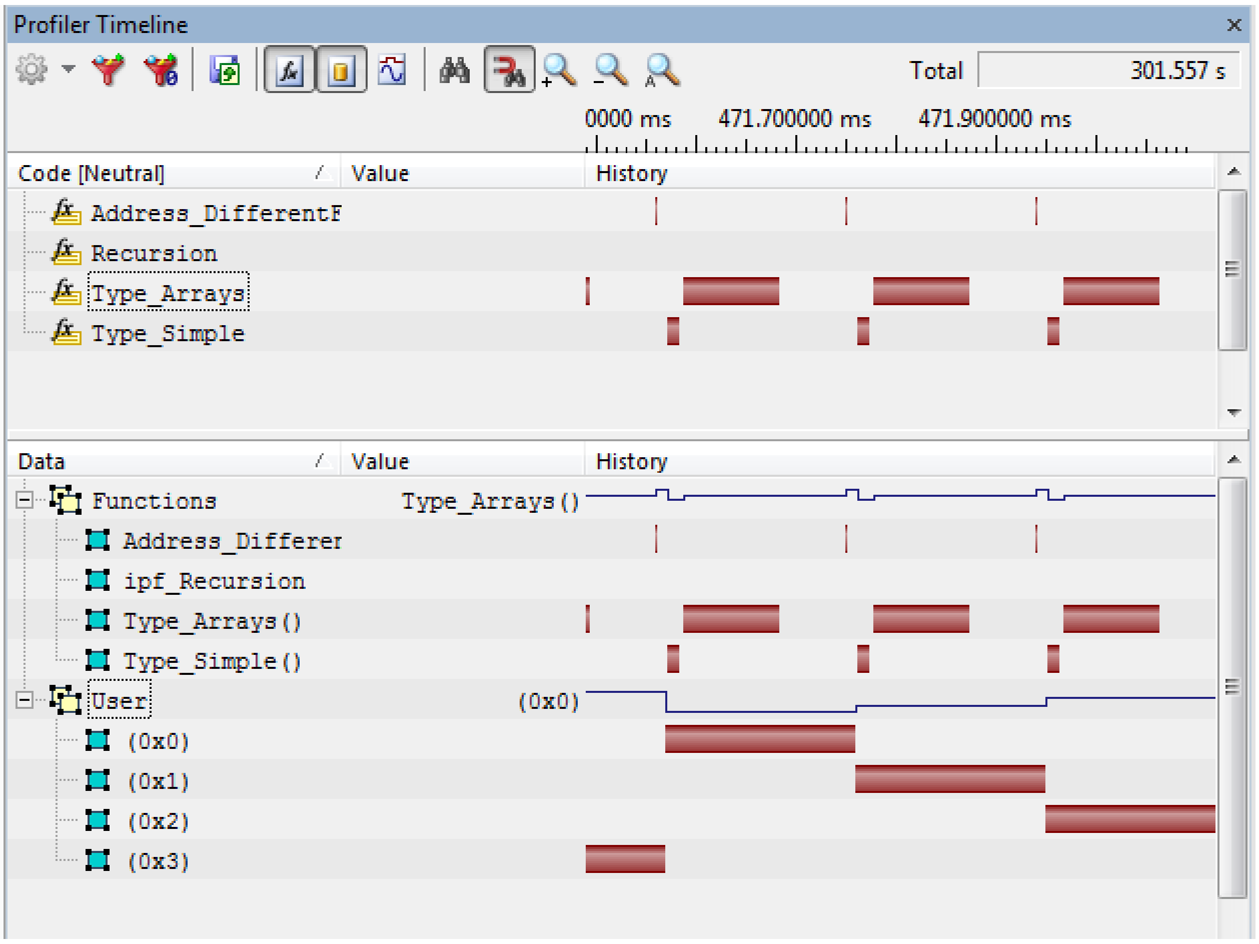
☑ Ignore writes of the same value

Range: -

OK
Cancel

5 Result display

Since all analyses are realized via instrumentation, the function execution is visible in the Data pane of the Analyzer window too. The profiler will additionally correlate these instrumentation events to and functions and display the results in the Function pane too.



All other information (Tasks, IRQs, User data) is displayed in the usual manner.

See ***Analyzer.pdf*** and ***ProfilerConcepts.pdf*** for more information.