µC/Trace™

The RTOS Event Analyzer

User's Manual

V1.0

Micriµm Press

Weston, FL 33326

Micriμm
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.micrium.com

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where Micriμm Press is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this manual are the property of their respective holders.

Copyright © 2013 by Micriμm except where noted otherwise. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

μC/Probe and the accompanying files are sold "as is". Micriμm makes and customer receives from Micriμm no express or implied warranties of any kind with respect to the software product, documentation, maintenance services, third party software, or other services. Micriμm specifically disclaims and excludes any and all implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Due to the variety of user expertise, hardware and software environments into which μC/Probe may be subjected, the user assumes all risk of using μC/Probe. The maximum liability of Micriμm will be limited exclusively to the purchase price.

600-uC-Trace-001

Micriμm Press

Table of Contents

Chapter

1

Introduction

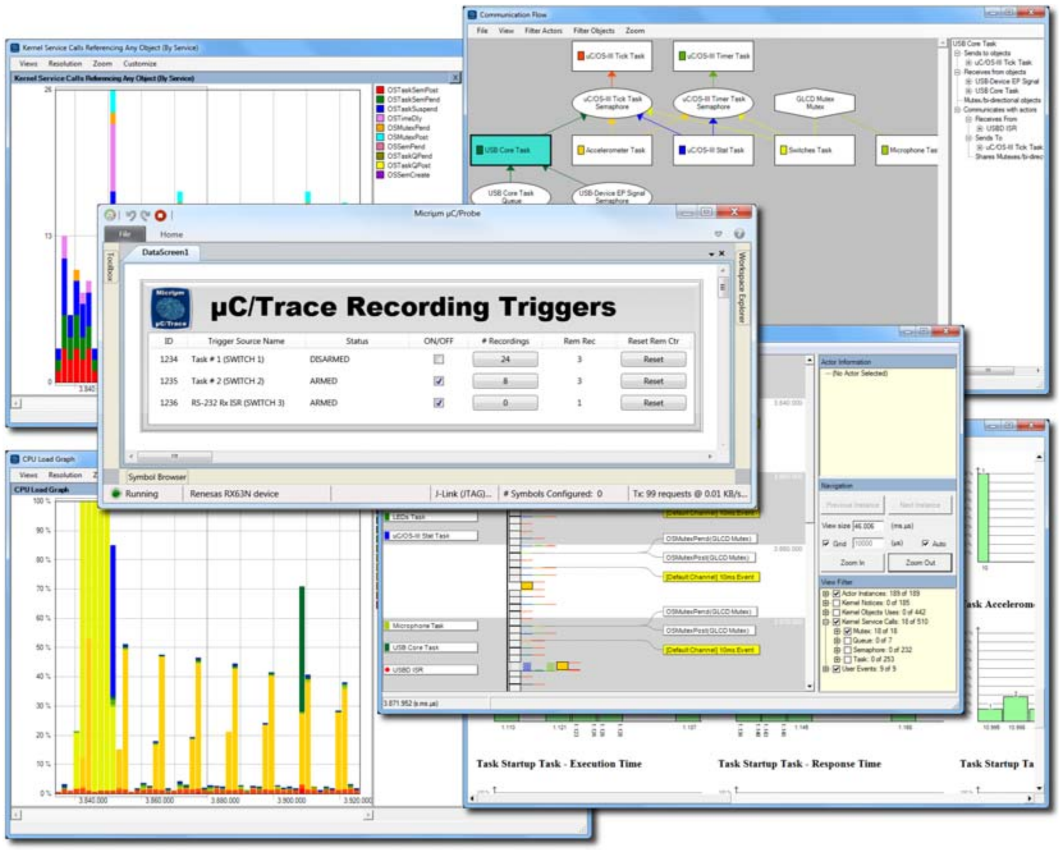
µC/Trace is a set of runtime diagnostics tools for embedded systems based on µC/OS-III. By simply including four C files into your project, µC/Trace gives you an unprecedented insight into the runtime behavior, which allows for reduced troubleshooting time and improved software quality, performance and reliability. Complex software problems which otherwise may require many hours or days to solve, can with µC/Trace be understood quickly, often in a tenth of the time otherwise required. This saves you many hours of troubleshooting time. Moreover, the increased software quality resulting from using µC/Trace can reduce the risk of defective software releases, causing damaged customer relations.

The insight provided by µC/Trace also allows you to find opportunities for optimizing your software. You might have unnecessary resource conflicts in your software, which are "low hanging fruit" for optimization and where a minor change can give a significant improvement in real-time responsiveness and user-perceived performance. By using µC/Trace, software developers can reduce their troubleshooting time and thereby get more time for developing new valuable features. This means a general increase in development efficiency and a better ability to deliver high-quality embedded software within budget.

µC/Trace provides more than 20 interconnected views of the runtime behavior, including task scheduling and timing, interrupts, interaction between tasks, as well as user events generated from your application as shown in Figure 1-1. µC/Trace can be used side-by-side with a traditional debugger and complements the debugger view with a higher level perspective, ideal for understanding the complex errors where a debugger's perspective is too narrow.

µC/Trace is more than just a viewer. It contains several advanced analyses developed since 2004, that helps you faster comprehend the trace data. For instance, it connects related events, which allows you to follow messages between tasks and to find the event that triggers a particular task instance. Moreover, it provides various higher level views such as the Communication Flow graph and the CPU load graph, which make it easier to find anomalies in a trace.

µC/Trace does not depend on additional trace hardware, which means that it can be used in deployed systems to capture rare errors which otherwise are hard to reproduce.


Kernel Service Calls Referencing Any Object (By Service)
Communication Flow
Micrium µC/Probe
DataScreen1
CPU Load Graph
Task Startup Task - Execution Time
Task Startup Task - Response Time

µC/Trace Recording Triggers

ID	Trigger Source Name	Status	ON/OFF	# Recordings	Rem Rec	Reset Rem Ctr
1234	Task # 1 (SWITCH 1)	DISARMED	☐	24	3	Reset
1235	Task # 2 (SWITCH 2)	ARMED	✓	8	3	Reset
1236	RS-232 Rx ISR (SWITCH 3)	ARMED	✓	0	1	Reset

Running | Renesas RX63N device | J-Link (JTAG)... | # Symbols Configured: 0 | Tx: 99 requests @ 0.01 KB/s...

Figure 1-1 **µC/Trace Analyzer Windows**

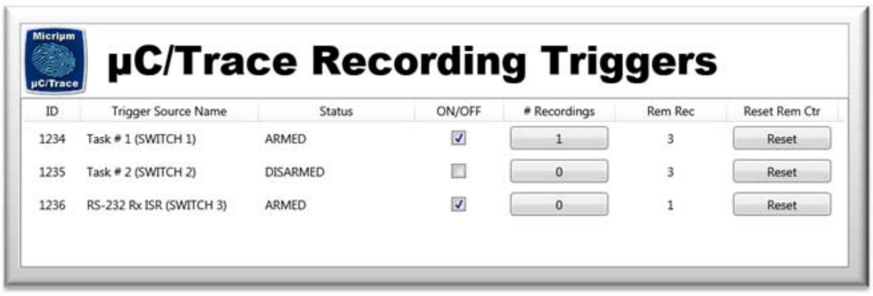
The µC/Trace solution consists of three parts:

- The PC application (*µC/Trace Analyzer*), used to analyze the recordings as shown in Figure 1-1.
- A trace recorder library (*µC/Trace Recorder*) that integrates with µC/OS-III, provided in C source code.
- Optionally, µC/Probe can be used to trigger and upload recordings from the target.

The µC/Trace Analyzer is a Windows application for the PC.

The µC/Trace Recorder stores the event data in a RAM buffer, which is uploaded on request to the host PC using your existing debugger connection or µC/Probe.

And finally, you can use µC/Probe and a special control designed for µC/Trace called *µC/Trace Trigger Control*, to trigger a recording and launch the µC/Trace Analyzer. The µC/Trace Trigger Control is shown in Figure 1-2:



µC/Trace Recording Triggers

ID	Trigger Source Name	Status	ON/OFF	# Recordings	Rem Rec	Reset Rem Ctr
1234	Task # 1 (SWITCH 1)	ARMED	✓	1	3	Reset
1235	Task # 2 (SWITCH 2)	DISARMED	☐	0	3	Reset
1236	RS-232 Rx ISR (SWITCH 3)	ARMED	✓	0	1	Reset

Figure 1-2 **µC/Trace Trigger Control**

µC/Probe is optional. If your platform is not supported by µC/Probe you can still use µC/Trace.

1-1 µC/TRACE SYSTEM OVERVIEW

µC/Trace is a set of tools designed to record and analyze trace data from an embedded system. Figure 1-3 shows an overview of the entire system and data flow when used in conjunction with µC/Probe.

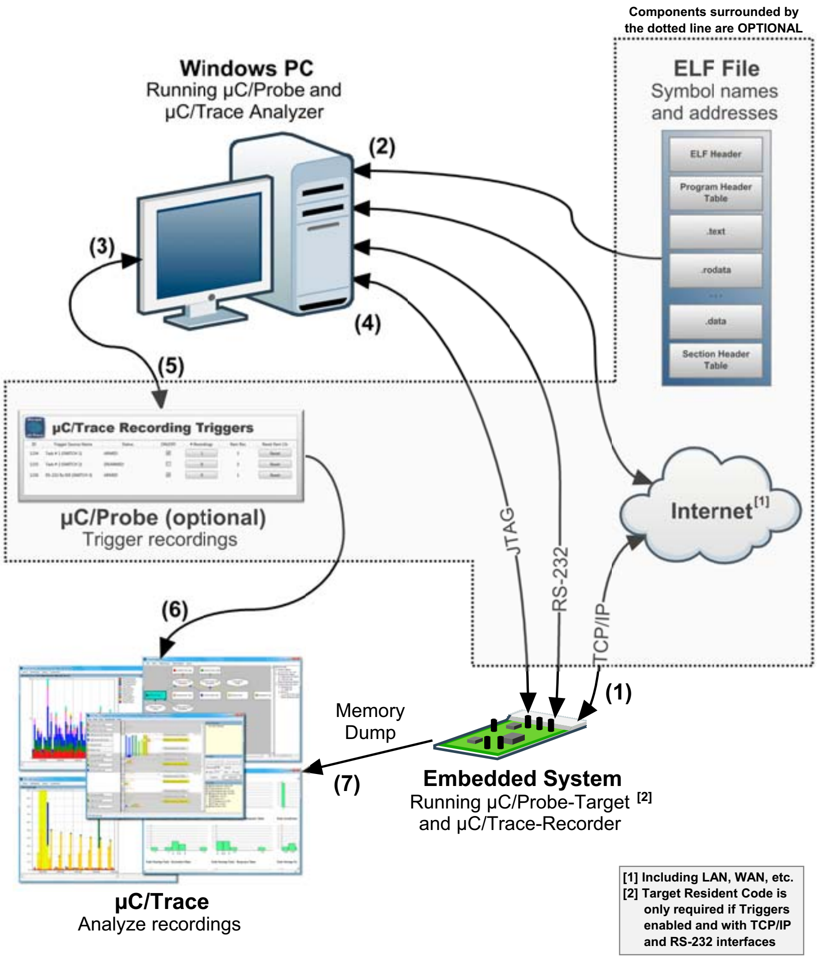

Components surrounded by the dotted line are OPTIONAL
Windows PC
Running µC/Probe and µC/Trace Analyzer
ELF File
Symbol names and addresses
ELF Header
Program Header Table
.text
.rodata
.data
Section Header Table
(2)
(3)
(4)
(5)
µC/Trace Recording Triggers
µC/Probe (optional)
Trigger recordings
Internet[1]
JTAG
RS-232
TCP/IP
(6)
(1)
Memory Dump
(7)
Embedded System
Running µC/Probe-Target [2] and µC/Trace-Recorder
µC/Trace
Analyze recordings
[1] Including LAN, WAN, etc.
[2] Target Resident Code is only required if Triggers enabled and with TCP/IP and RS-232 interfaces

Figure 1-3 **µC/Trace Data Flow Diagram**

F1-3(1) You start by creating trigger points in your embedded target application code. Each part of your code that you want to record needs to call a single line macro. µC/Probe is aware of all the trigger points you configured because you will create a configuration table that lists all the available trigger points.

F1-3(2) You have to provide µC/Probe with an ELF file with DWARF-2, -3 or -4 debugging information. The ELF file is generated by your toolchain's linker. µC/Probe parses the ELF file and reads the addresses of each of the embedded target's symbols (i.e., global variables) including the memory address of the buffer where the trace recording is stored. For more information on building the ELF file, see the µC/Probe Target Manual.

Alternatively, you can also provide a chip definition file that contains the chip's peripheral register addresses or provide your own custom XML based symbol file for those cases when your toolchain cannot generate one of the supported ELF formats.

F1-3(3) During design-time, you create a µC/Probe workspace using a Windows PC and µC/Probe. You design your own dashboard by dragging and dropping virtual controls and indicators onto a *data screen*. Each virtual control and indicator needs to be mapped to an embedded target's symbol by selecting it from the symbol browser. However, for µC/Trace purposes, all you have to do is drag and drop a µC/Trace Trigger control from the µC/Probe's toolbox. Refer to the document µC/Probe User's Manual for more information on using the µC/Trace Trigger control.

F1-3(4) Before proceeding to the run-time stage, µC/Probe needs to be configured to use one of the three communication interfaces: JTAG, RS232 or TCP/IP. In order to start the run-time stage, you click the *Run* button and µC/Probe starts making requests to read the value of all the memory locations associated with each virtual control and indicator (i.e., all the trace trigger points you programmed in the embedded target).

F1-3(5) µC/Probe will display all the available trigger points and you can arm or disarm each of them.

F1-3(6) When the embedded application executes the part of your code delimited by one of your armed trigger points, µC/Probe will take care of uploading the recording and launching the analysis tools.

F1-3(7) µC/Probe is optional. So, if your platform is not supported by µC/Probe, you can still record and analyze traces with µC/Trace by creating a memory dump file and loading it directly with the µC/Trace Analyzer.

If you are unsure of which communication interface to use with µC/Probe, try the communication options advisor in Figure 1-4:

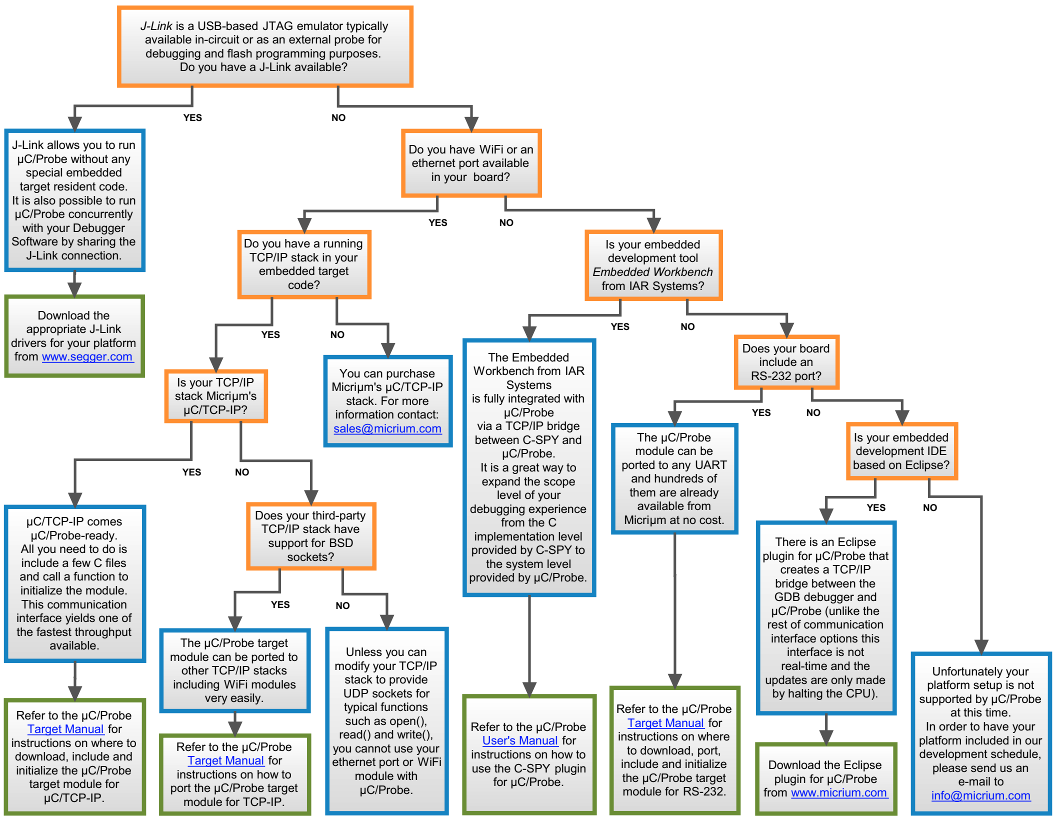

J-Link is a USB-based JTAG emulator typically available in-circuit or as an external probe for debugging and flash programming purposes. Do you have a J-Link available?
YES
NO
J-Link allows you to run µC/Probe without any special embedded target resident code. It is also possible to run µC/Probe concurrently with your Debugger Software by sharing the J-Link connection.
Download the appropriate J-Link drivers for your platform from www.segger.com
Do you have WiFi or an ethernet port available in your board?
YES
NO
Do you have a running TCP/IP stack in your embedded target code?
Is your embedded development tool Embedded Workbench from IAR Systems?
YES
NO
Is your TCP/IP stack Micriµm's µC/TCP-IP?
You can purchase Micriµm's µC/TCP-IP stack. For more information contact: sales@micrium.com
YES
NO
The Embedded Workbench from IAR Systems is fully integrated with µC/Probe via a TCP/IP bridge between C-SPY and µC/Probe. It is a great way to expand the scope level of your debugging experience from the C implementation level provided by C-SPY to the system level provided by µC/Probe.
Does your board include an RS-232 port?
YES
NO
µC/TCP-IP comes µC/Probe-ready. All you need to do is include a few C files and call a function to initialize the module. This communication interface yields one of the fastest throughput available.
Does your third-party TCP/IP stack have support for BSD sockets?
YES
NO
The µC/Probe module can be ported to any UART and hundreds of them are already available from Micriµm at no cost.
Is your embedded development IDE based on Eclipse?
YES
NO
There is an Eclipse plugin for µC/Probe that creates a TCP/IP bridge between the GDB debugger and µC/Probe (unlike the rest of communication interface options this interface is not real-time and the updates are only made by halting the CPU).
The µC/Probe target module can be ported to other TCP/IP stacks including WiFi modules very easily.
Unless you can modify your TCP/IP stack to provide UDP sockets for typical functions such as open(), read() and write(), you cannot use your ethernet port or WiFi module with µC/Probe.
Unfortunately your platform setup is not supported by µC/Probe at this time. In order to have your platform included in our development schedule, please send us an e-mail to info@micrium.com
Refer to the µC/Probe Target Manual for instructions on where to download, include and initialize the µC/Probe target module for µC/TCP-IP.
Refer to the µC/Probe Target Manual for instructions on how to port the µC/Probe target module for TCP-IP.
Refer to the µC/Probe User's Manual for instructions on how to use the C-SPY plugin for µC/Probe.
Refer to the µC/Probe Target Manual for instructions on where to download, port, include and initialize the µC/Probe target module for RS-232.
Download the Eclipse plugin for µC/Probe from www.micrium.com

Figure 1-4 **Communication Options Advisor**

This document will describe how to use µC/Trace by discussing the following topics:

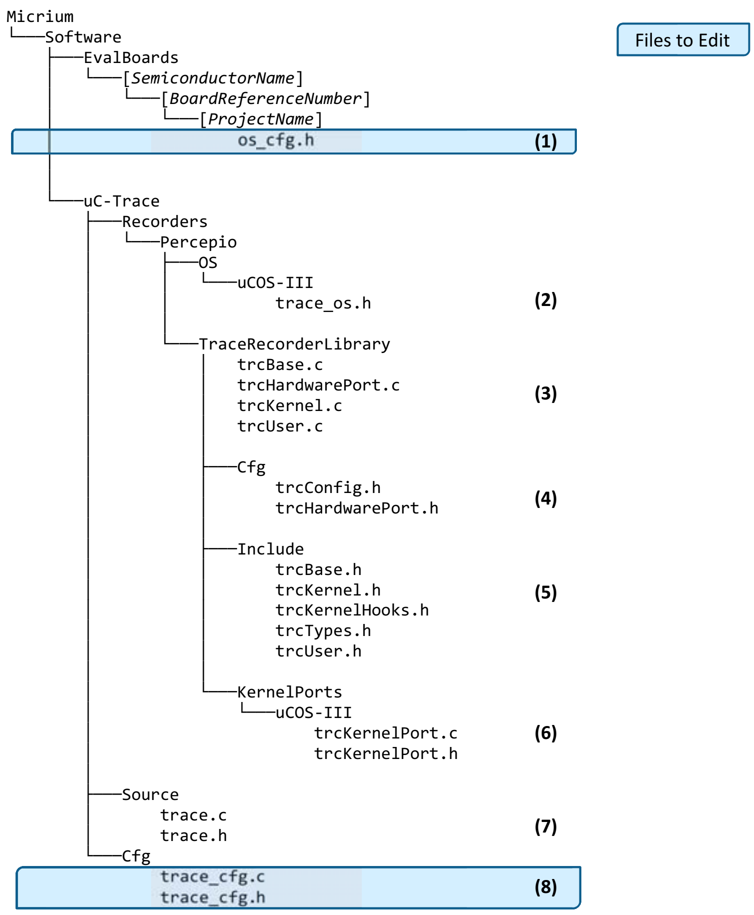
- Including the µC/Trace supporting code in your embedded target
- µC/Trace Triggers Functional Description that includes the following topics:
  - Configuring and Initializing µC/Trace in your embedded target
  - Instrumenting your embedded target code with µC/Trace Triggers
  - Triggering and analyzing recordings from a µC/Trace Trigger control for µC/Probe.

Chapter

2

µC/Trace Recorder Module

The µC/Trace Recorder module is the set of C files that reside on the embedded target and implement the events triggering and recording mechanism. The files are available at http://micrium.com/trace and this chapter aims at providing a brief introduction to the files.



```
Micrium
└───Software
    ├───EvalBoards
    │   └───[SemiconductorName]
    │       └───[BoardReferenceNumber]
    │           └───[ProjectName]
    │                   os_cfg.h                            (1)
    │
    └───uC-Trace
        ├───Recorders
        │   └───Percepio
        │       ├───OS
        │       │   └───uCOS-III
        │       │           trace_os.h                  (2)
        │       │
        │       └───TraceRecorderLibrary
        │           │   trcBase.c
        │           │   trcHardwarePort.c               (3)
        │           │   trcKernel.c
        │           │   trcUser.c
        │           │
        │           ├───Cfg
        │           │       trcConfig.h                 (4)
        │           │       trcHardwarePort.h
        │           │
        │           ├───Include
        │           │       trcBase.h
        │           │       trcKernel.h                 (5)
        │           │       trcKernelHooks.h
        │           │       trcTypes.h
        │           │       trcUser.h
        │           │
        │           └───KernelPorts
        │               └───uCOS-III
        │                       trcKernelPort.c         (6)
        │                       trcKernelPort.h
        │
        ├───Source
        │       trace.c                                 (7)
        │       trace.h
        └───Cfg
                trace_cfg.c                             (8)
                trace_cfg.h
```

Figure 2-1 **µC/Trace C Files**

F2-1(1) Somewhere in your application level code (typically `os_cfg.h`) you have to define the macro `TRACE_CFG_EN` as 1 to enable µC/Trace.

F2-1(2) µC/Trace is designed to work with third-party trace recorders such as the one developed by a Swedish company called Percepio AB. An OS layer defined in `trace_os.h` allows you to use any of the supported third-party recorders without any changes to your application code.

F2-1(3) The core source files for the trace recorder library by Percepio.

F2-1(4) A couple of files allow you to configure the size of the RAM buffer and the hardware clock among other settings as described in Chapter 3, on page 20.

F2-1(5) The core header files for the trace recorder library by Percepio.

F2-1(6) The kernel port that implements how to record the kernel events.

F2-1(7) The source files for µC/Trace and the optional µC/Trace Triggering mechanism.

F2-1(8) The configuration files for µC/Trace and the optional triggering mechanism. Typically you move these files to the same location where your application level code is, as described in Chapter 3, "trace_cfg.c/h" on page 17.

2-1 INCLUDING THE µC/TRACE FILES IN YOUR C PROJECT

µC/Trace requires µC/OS-III version 3.04.01 or newer and the files described in the previous section and shown in Figure 2-1.

In case you have a µC/OS-III based project and want to include µC/Trace in it, the next sections describe the steps involved.

2-1-1 COPY THE µC/TRACE FILES TO YOUR MICRIUM FOLDER

Download the µC/Trace files from http://micrium.com/trace and copy the entire `$/Micrium/Software/uC-Trace` folder shown in Figure 2-1 into your own `$/Micrium/Software` folder.

2-1-2 MOVE THE µC/TRACE CONFIGURATION TEMPLATE FILES

Move (*Cut+Paste*) the following files to the folder where you keep the rest of your application level configuration files (i.e. `app_cfg.h`):

`$\Micrium\Software\uC-Trace\Cfg\trace_cfg.c`

`$\Micrium\Software\uC-Trace\Cfg\trace_cfg.h`

2-1-3 INSERT THE µC/TRACE FILES INTO YOUR PROJECT

Some IDEs such as Eclipse do not require you to specify which files to compile by inserting each file in your project. But, other IDEs will require you to add the following files into your project:

`$\Micrium\Software\uC-Trace\Source\trace.c`

`$\Micrium\Software\uC-Trace\Source\trace.h`

`$\Micrium\Software\uC-Trace\Cfg\trace_cfg.c`

`$\Micrium\Software\uC-Trace\Cfg\trace_cfg.h`

2-1-4 CONFIGURE YOUR COMPILER'S INCLUDE PATHS

Configure your compiler's include paths with the following:

```
$\Micrium\Software\uC-Trace\Source
$\Micrium\Software\uC-Trace\Recorders\Percepio
$\Micrium\Software\uC-Trace\Recorders\Percepio\OS\uCOS-III
$\Micrium\Software\uC-Trace\Recorders\Percepio\TraceRecorderLibrary
$\Micrium\Software\uC-Trace\Recorders\Percepio\TraceRecorderLibrary\Cfg
$\Micrium\Software\uC-Trace\Recorders\Percepio\TraceRecorderLibrary\Include
$\Micrium\Software\uC-Trace\Recorders\Percepio\TraceRecorderLibrary\KernelPorts\uCOS-III
```

2-2 INCLUDING HEADER FILES

Additionally, each C file that makes use of µC/Trace, whether it is for Initialization or Instrumentation purposes (i.e. `app.c`), needs to include the following header files by using the preprocessing directive `#include`:

```
#include  "trace.h"
#include  "trace_cfg.h"
```

Listing 2-1 **Including µC/Trace header files**

2-3 µC/TRACE FUNCTIONAL DESCRIPTION

Figure 2-2 shows the block diagram of the entire system including the trace recording module in the target code and µC/Probe and µC/Trace in the host system. The entire operation can be described in 10 steps.

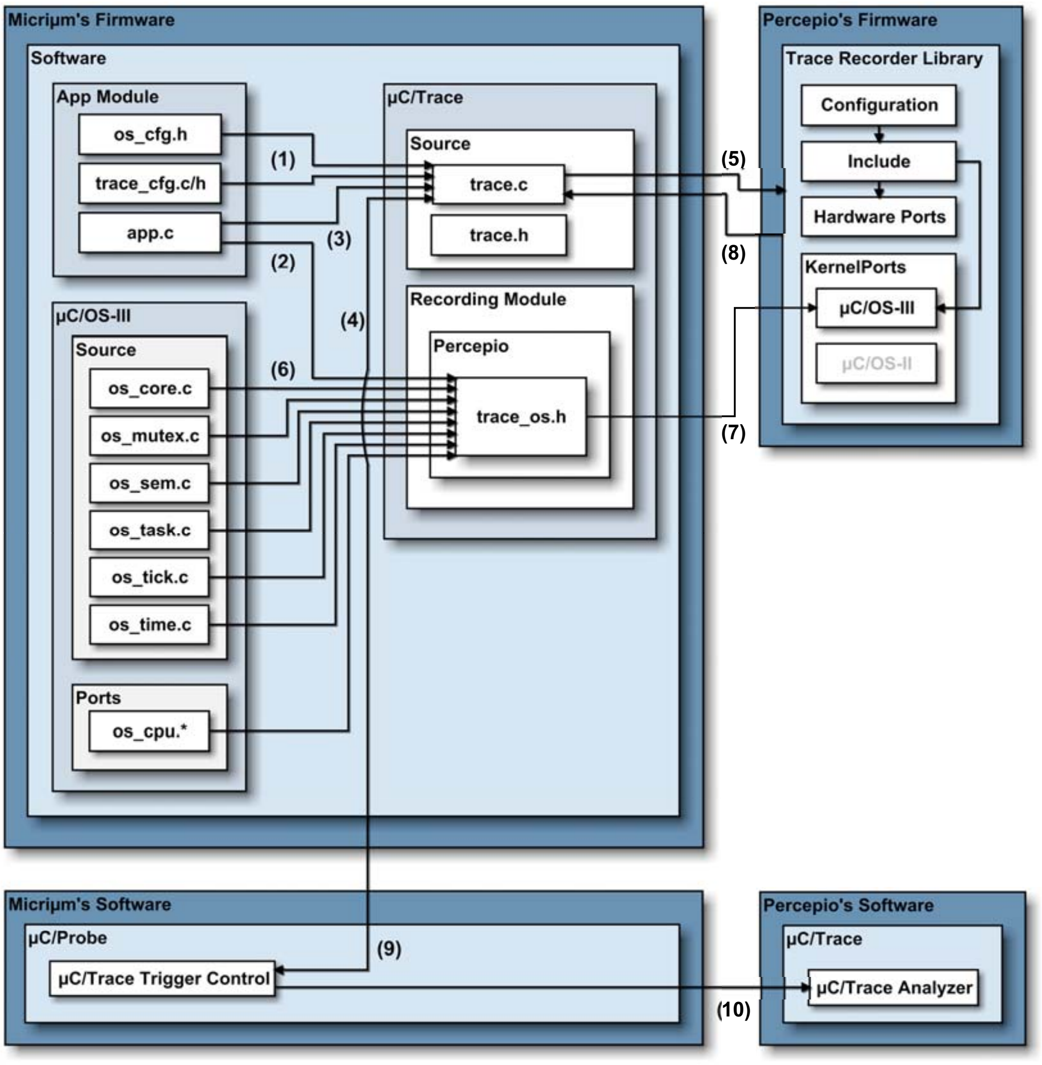


Figure 2-2 **µC/Trace Block Diagram**

F2-2(1) The first step to get started with µC/Trace is enabling the module in the configuration file for µC/OS-III (os_cfg.h) through the definition of the macro TRACE_CFG_EN set to 1.

The second step is optional and available only if your platform is supported by µC/Probe. If so, in this configuration stage you get to define the trigger points in the table declared in trace_cfg.c as shown in Listing 2-2:

```
/*
*********************************************************************************
*                            UC/TRACE TRIGGERS IDS
*********************************************************************************
*/

#define  TRACE_CFG_TRIG_ID_SW1                  1234u
#define  TRACE_CFG_TRIG_ID_SW2                  1235u
#define  TRACE_CFG_TRIG_ID_ISR_RS232_RX         1236u

/*
*********************************************************************************
*                      UC/TRACE TRIGGERS CONFIGURATION TABLE
*********************************************************************************
*/

const  TRACE_CFG_TRIG  TraceCfgTrigTbl[] =
{
    {TRACE_CFG_TRIG_ID_SW1,          "Task # 1 (SWITCH 1)",      3},
    {TRACE_CFG_TRIG_ID_SW2,          "Task # 2 (SWITCH 2)",      3},
    {TRACE_CFG_TRIG_ID_ISR_RS232_RX, "RS-232 Rx ISR (SWITCH 3)", 1}
};
```

Listing 2-2 **µC/Trace Triggers Configuration Table**

The first parameter is the trigger ID, the second parameter is the name of trigger and the third parameter is the number of recordings you want to capture before disarming the trigger automatically.

Once you have the trigger IDs configured, you can start instrumenting your code by simply calling the macro TRACE_TRIG() with the trigger ID as a parameter wherever you want to start recording.

F2-2(2) You initialize the µC/Trace module by calling the macro TRACE_INIT().

If you are not planning to use µC/Probe to trigger recordings, then you will also need to call the macro `TRACE_START()` to start recording.

F2-2(3) You initialize the optional µC/Trace Triggers module by calling the function `TraceTrigInit()` which is declared in `trace.c`.

F2-2(4) The user interface for µC/Trace Triggers is µC/Probe. Once you get µC/Probe communicating with your target as described in the µC/Probe documentation you can create a workspace that contains the µC/Trace Trigger control found in the µC/Probe toolbox. This control, allows you to not only arm and disarm the recording triggers in your target but also upload the recording and launch the µC/Trace Analyzer Windows application.

In this step, the user would arm one or more of the trigger points.

F2-2(5) As soon as the part of your target code reaches the point where the `TRACE_TRIG()` macro gets executed, the system will start recording.

F2-2(6) All the kernel events will be recorded into RAM.

F2-2(7) The events get recorded into RAM by using a special encoding that takes 4 bytes per event.

F2-2(8) As soon as the recording gets stopped either because your application calls the `TRACE_STOP()` macro or the RAM buffer gets full, the µC/Trace Triggers module gets notified by the recorder.

F2-2(9) In turn, the µC/Trace Triggers module notifies µC/Probe that the recording is finished.

F2-2(10) µC/Probe and its µC/Trace Triggers control in particular receive the notification from the target and start reading the recording off the target's RAM, dump the raw bytes to a file and launch the µC/Trace Windows application to analyze the trace.

Chapter

3

Configuring µC/Trace

This chapter aims at providing a more detailed description of the tweaks you need to do to your application level code and the recorder library to run µC/Trace.

3-1 ENABLING OR DISABLING µC/TRACE

3-1-1 OS_CFG.H

The first step to get started with µC/Trace is enabling the module in `os_cfg.h` through the definition of the macro `TRACE_CFG_EN` set to 1, which is useful to disable tracing once your embedded application gets deployed:

```
#define    TRACE_CFG_EN    1        /* Enable/Disable the uC/Trace Recorder.         */
```

Figure 3-1 **Enable/Disable µC/Trace**

3-2 RECORDER CONFIGURATION SETTINGS

The configuration files for the recorder library are usually placed along with your application level configuration files.

There is a configuration template in `$\Micrium\Software\uC-Trace\Cfg`.

You can make a copy of this template and place it along with the rest of your application level configuration files.

3-2-1 TRACE_CFG.C/H

These configuration files define a series of macros to select the hardware platform, customize your recordings and configure the optional µC/Trace triggering system.

SELECTING YOUR HARDWARE PLATFORM

The recorder needs to have knowledge of the hardware platform your are running, the timer/counter used for driving the system ticker in particular.

µC/Trace has been ported to the following hardware platforms:

- ARM Cortex-M
- ARM Cortex-A
- Renesas RX600

If your platform includes one of the above, then all you need to do is configure the macro `TRACE_CFG_HW_PORT` in `trace_cfg.h` as shown in the following code listing:

```
/*
*********************************************************************************************
*                                       HARDWARE PORT
*
* Note(s) : The recorder needs to have knowledge of the hardware platform your are running,
*           the timer/counter used for driving the system ticker in particular.
*
*           Select from one of the following supported platforms:
*
*                       - ARM CORTEX-M   :     4u
*                       - RENESAS RX600  :     5u
*                       - ARM CORTEX-A   :    14u
*********************************************************************************************
*/

#define  TRACE_CFG_HW_PORT                     5u
```

Listing 3-1 **Selecting your Hardware Platform**

If your hardware platform is not listed above, please contact Micriµm.

CUSTOMIZING YOUR RECORDINGS

Also in `trace_cfg.h`, the two settings you may want to edit are the maximum number of events to store in RAM and the maximum number of kernel objects to record as shown in the following code listing:

```
/*
*********************************************************************************************
*                                      RECORDING BUFFER
*
* Note(s) : This defines the capacity of the event buffer,
*           i.e., the number of records it may store.
*           Each recorded event typically uses one record (4 byte in RAM).
*           Adjust to your application needs and memory resources.
*********************************************************************************************
*/

#define  TRACE_CFG_MAX_EVENTS              2000u                                         (1)


/*
*********************************************************************************************
*                                   RECORDED KERNEL OBJECTS
*
* Note(s) : These define the maximum number of kernel object types to record.
*           Adjust to your application needs and memory resources.
*********************************************************************************************
*/

#define  TRACE_CFG_MAX_TASK                  16u                                         (2)
#define  TRACE_CFG_MAX_ISR                    8u
#define  TRACE_CFG_MAX_Q                     16u
#define  TRACE_CFG_MAX_SEM                   32u
#define  TRACE_CFG_MAX_MUTEX                  8u
#define  TRACE_CFG_MAX_FLAG                   2u
#define  TRACE_CFG_MAX_MEM                    2u
```

Listing 3-2 **Recorder Storage Settings**

L3-2(1) This defines the capacity of the event buffer, i.e., the number of records it may store. Each registered event typically uses one record (4 bytes).

L3-2(2) These define the maximum number of kernel object types to record. Adjust to your application needs.

CONFIGURING THE µC/TRACE TRIGGERING SYSTEM

The third step is optional and available only if your platform is supported by µC/Probe and you want to trigger recordings from µC/Probe.

If so, in this configuration stage you get to enable the system and define the trigger points as described in the following code listings.

```
/*
********************************************************************************************
*                                UC/TRACE TRIGGERS MODULE
*
* Note(s) : The uC/Trace Triggering mechanism requires the hardware platform to be
*           supported by uC/Probe.
*
*           If your platform is uC/Probe-ready, then enable this module by setting
*           the macro below to 1.
********************************************************************************************
*/

#define  TRACE_CFG_TRIG_EN                     1u                                            (1)

#if (defined(TRACE_CFG_TRIG_EN) && (TRACE_CFG_TRIG_EN > 0u))


/*
********************************************************************************************
*                               UC/TRACE TRIGGERS SETTINGS
********************************************************************************************
*/

#define  TRACE_CFG_TRIG_MAX_TRIGS             16u                                            (2)
#define  TRACE_CFG_TRIG_NAME_LEN              32u


/*
********************************************************************************************
*                             UC/TRACE TRIGGERS TASK SETTINGS
********************************************************************************************
*/

#define  TRACE_CFG_TRIG_TASK_STK_SIZE        128u                                            (3)
#define  TRACE_CFG_TRIG_TASK_PRIO             20u
```

```
/*
*********************************************************************************************
*                                UC/TRACE TRIGGERS IDS
*
* Note(s) : First you create a series of unique IDs for each of your trigger points.
*           In this example TRACE_CFG_TRIG_ID_SW1 is the trigger point executed when
*           you press the board's switch 1.
*********************************************************************************************
*/

#define  TRACE_CFG_TRIG_ID_SW1              1234u                                          (4)
#define  TRACE_CFG_TRIG_ID_SW2              1235u
#define  TRACE_CFG_TRIG_ID_SW3              1236u
```

Listing 3-3 **µC/Trace Triggers Configuration Table**

L3-3(1) As previously mentioned, the µC/Trace Triggering mechanism requires the hardware platform to be supported by µC/Probe. If your platform is µC/Probe-ready, then enable this module by setting the macro `TRACE_CFG_TRIG_EN` defined in `trace_cfg.h` to 1.

L3-3(2) The macro `TRACE_CFG_TRIG_MAX_TRIGS` declared in `trace_cfg.h`, defines the maximum number of trigger points throughout your code.

In the same file, the macro `TRACE_CFG_TRIG_NAME_LEN` allows you to define the maximum number of characters of each trigger point's name, in case you need to tweak the memory footprint.

L3-3(3) The triggering mechanism uses a µC/OS-III task to trigger the recordings and communicate with µC/Probe. The macros `TRACE_CFG_TRIG_TASK_STK_SIZE` and `TRACE_CFG_TRIG_TASK_PRIO` define the task's stack size and priority respectively.

L3-3(4) The first step to configure trigger points is to create a series of unique IDs for each of your trigger points. In this example `TRACE_CFG_TRIG_ID_SW1` is the trigger point executed when you press switch 1 on the board.

Once you have configured all the settings defined in `trace_cfg.h`, you switch to the file `trace_cfg.c` where you take each of your trigger IDs and create a configuration table as shown in the code listing below.

```
/*
*********************************************************************************************
*                          UC/TRACE TRIGGERS CONFIGURATION TABLE
*
* Example : { [Trigger ID], [Trigger Source Name], [Max # Files to Record Before Disarming] }
*
* Note(s) : Make sure the number of entries in this configuration table is
*           less than the value of TRACE_CFG_TRIG_MAX_TRIGS
*           otherwise, any other triggers in excess will be ignored.
*********************************************************************************************
*/

const  TRACE_CFG_TRIG  TraceCfgTrigTbl[] =
{
    {TRACE_CFG_TRIG_ID_SW1, "Signal Task # 1   (SWITCH 1)", 3},                        (1)
    {TRACE_CFG_TRIG_ID_SW2, "Post Q Task # 2   (SWITCH 2)", 3},
    {TRACE_CFG_TRIG_ID_SW3, "Other Kernel Objs (SWITCH 3)", 1}
};
```

Listing 3-4 **µC/Trace Triggers Configuration Table**

L3-4(1) The first parameter is the trigger ID, the second parameter is the trigger's name and the third parameter is the number of recordings you want to capture before disarming the trigger automatically.

Chapter

4

Initializing µC/Trace

Initializing the µC/Trace recorder module is easy, the first thing you need to do is call the macro `TRACE_INIT()` as shown in the following code listing:

```
int main(void)
{
    OS_ERR  err;


    CPU_IntDis();               /* Disable all interrupts.                */

#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_INIT();               /* Initialize the uC/Trace recorder.      */
#endif
.
.
.
```

Figure 4-1 **Initializing the µC/Trace Recorder**

The next step depends on whether or not your platform is supported by µC/Probe and you want to trigger recordings from µC/Probe. Otherwise, you can still use µC/Trace without µC/Probe.

If you are using µC/Probe to trigger recordings then you need to call the function to initialize the triggering module as shown in the following code listing:

```
static  void  AppTaskStart (void *p_arg)
{
.
.
.
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TraceTrigInit();             /* Initialize uC/Trace triggers.            */
#endif
.
.
.
```

Listing 4-1 **Initializing the µC/Trace Triggering Module**

Chapter

5

Instrumenting with µC/Trace

There are three groups of events you can record with µC/Trace; Kernel Service Calls, Interrupt Service Routines and User-Defined Events.

This chapter describes how to instrument your embedded application with each of those three types of events.

5-1 KERNEL SERVICE CALLS

Because µC/OS-III has already been instrumented, every Kernel Service Call gets automatically recorded as soon as you start recording.

5-2 INTERRUPT SERVICE ROUTINES

Unlike Kernel Service Calls, ISRs do not get automatically recorded. Instead, you need to place three marcos for each ISR you want to record.

To register an ISR for recording, you call the macro `TRACE_OS_ISR_REGISTER()` with three arguments: *unique ISR ID*, *Name* and *Priority* as shown in the code listing below:

```
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_OS_ISR_REGISTER(1, "RS-232 Tx ISR", 4);        /* Registering an ISR.           */
#endif
```

Listing 5-1 **Registering an ISR for Recording**

Additionally, you need to mark the beginning and end of the ISR by calling the macros TRACE_OS_ISR_BEGIN() and TRACE_OS_ISR_END() as shown in the code listing below:

```
void  BSP_Ser_ISR_Tx_Handler (void)
{
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_OS_ISR_BEGIN(1);                   /* Mark the beginning of an ISR.      */   (1)
#endif
    SCI2.SSR.BIT.TEND;
    SCI2.SSR.BIT.TEND = 0;
    BSP_OS_SemPost(&BSP_SerTxWait);          /* Post to the semaphore              */
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_OS_ISR_END();                      /* Mark the end of an ISR.            */   (2)
#endif
}
```

Listing 5-2 **Tracing an ISR**

L5-2(1) The macro TRACE_OS_ISR_BEGIN() takes the unique ID (i.e. vector number) of the ISR you want to record as an argument.

L5-2(2) The macro TRACE_OS_ISR_END() takes no arguments.

5-3 USER-DEFINED EVENTS

The last group of events you can record with µC/Trace is the user-defined events. Similar to ISRs, you first need to register a user-defined event by calling the macro TRACE_USR_EVT_CREATE() with the name of the event as shown in the code listing below:

```
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    CPU_INT16U   evt_hnd;

    evt_hnd = TRACE_USR_EVT_CREATE("50 ms User Event");     /* Create a user-defined event.  */
#endif
```

Listing 5-3 **Creating a User-Defined Event**

The macro to create the event returns an event handle and then, all you have to do is call the macro `TRACE_USR_EVT_LOG()` with the event handle to record it as shown in the code listing below:

```
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_USR_EVT_LOG(evt_hnd);           /* Recording a user-defined event every 50ms. */
#endif
```

Listing 5-4 **Tracing a User-Defined Event**

Chapter

6

Recording with µC/Trace

There are two ways to start recording with µC/Trace. If your platform is supported by µC/Probe, then you can use µC/Probe and a special control called µC/Trace Trigger control to trigger and analyze recordings. If not, then you can still use µC/Trace without µC/Probe.

This chapter describes the two methods.

6-1 START RECORDING WITH µC/PROBE

Once you have the trigger IDs configured as described in Chapter 3, on page 17, you can simply call the macro `TRACE_TRIG()` with the trigger ID as a parameter wherever you want to start recording.

When the part of your code that calls this macro gets executed, the system will check if the trigger has been armed from µC/Probe and will not only initiate recording but also notify µC/Probe that the recording is ready for upload and analysis.

The code listing below shows an example of start recording whenever the user presses a button:

```
AppSwitch1 = BSP_SwsRd(1);                  /* Read the status of switch #1.              */

if (AppSwitch1 == DEF_ON) {
#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_TRIG(TRACE_TRIG_ID_SW1);          /* Evaluate the recording trigger conditions.  */
#endif
.
.
.
```

Listing 6-1 **Start Recording with µC/Probe**

6-2 START RECORDING WITHOUT µC/PROBE

If your platform is not supported by µC/Probe, you can still use µC/Trace. All you have to do is call the macro `TRACE_START()` to start the recording and `TRACE_STOP()` to stop the recording. You can also let the recording fill the RAM buffer until it gets full and µC/Trace will stop the recording automatically.

It is not necessary to start µC/OS-III before starting the recording and you can even start recording right from the embedded application's main entry point as shown in the following code listing:

```
int main(void)
{
    OS_ERR  err;


    CPU_IntDis();                             /* Disable all interrupts.               */

#if (defined(TRACE_CFG_EN) && (TRACE_CFG_EN > 0u))
    TRACE_INIT();                             /* Initialize the µC/Trace recorder.      */
    TRACE_START();                            /* Start recording.                      */
#endif
```

Listing 6-2 **Start Recording without µC/Probe**

Chapter 7 Analyzing with µC/Trace

The analysis of the trace is performed with a windows application in your host PC.

µC/Trace Analyzer is available for sale at `http://percepio.com/tracealyzer/uctrace/`

There is also a 30-day evaluation version that includes a demo trace, so you can try it without even writing a single line of code for your embedded target.

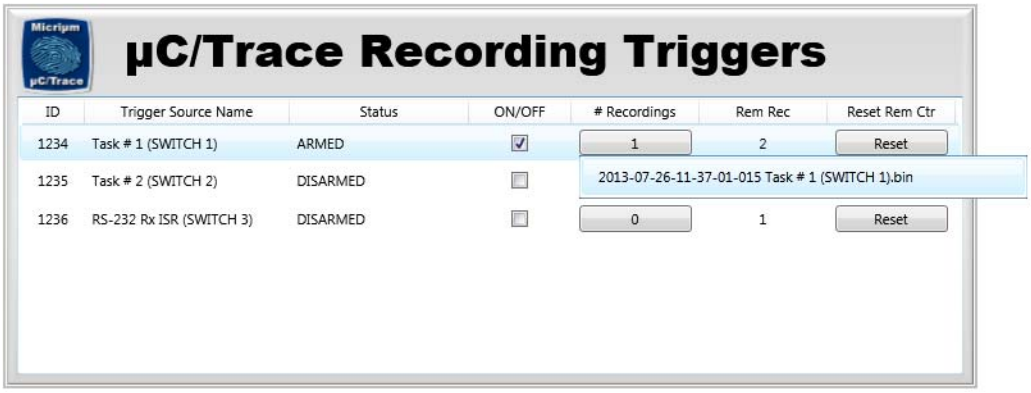
There are two ways to upload your recordings to your host PC for analysis. You can use µC/Probe or your debugging's software memory dump function.

This chapter will describe the two methods.

7-1 UPLOADING WITH µC/PROBE

When the part of your code instrumented with your trigger gets executed, the system will check if the trigger has been armed from µC/Probe and will not only initiate recording but also notify µC/Probe that the recording is ready for upload and analysis.

The recording gets uploaded by µC/Probe using one of the many communication interfaces and is made available as shown in the figure below:



µC/Trace Recording Triggers

ID	Trigger Source Name	Status	ON/OFF	# Recordings	Rem Rec	Reset Rem Ctr
1234	Task # 1 (SWITCH 1)	ARMED	✓	1	2	Reset
1235	Task # 2 (SWITCH 2)	DISARMED	☐			
1236	RS-232 Rx ISR (SWITCH 3)	DISARMED	☐	0	1	Reset

2013-07-26-11-37-01-015 Task # 1 (SWITCH 1).bin

Figure 7-1 **Uploading with µC/Probe**

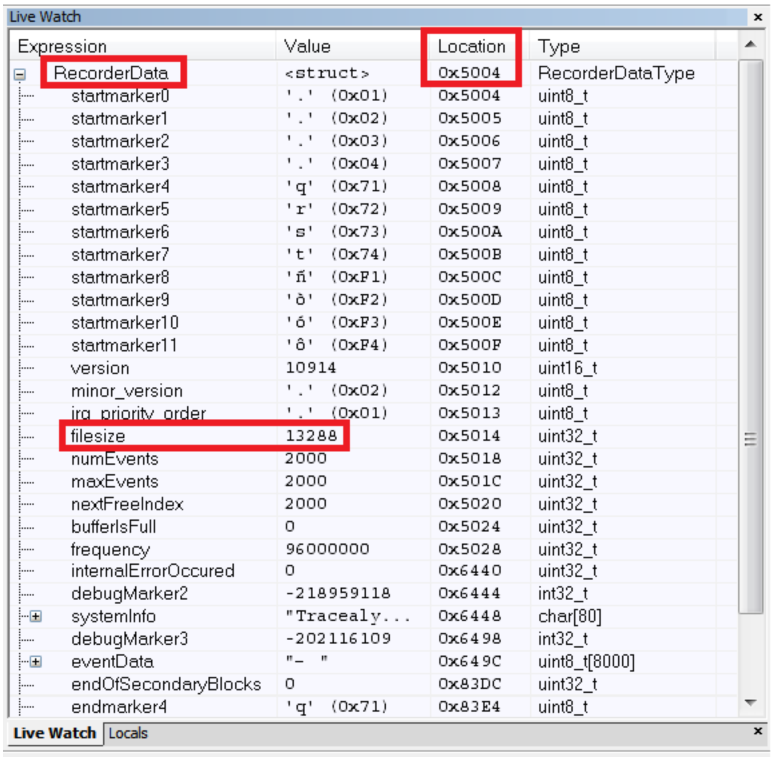
7-2 UPLOADING WITH A DEBUGGING TOOL

If your debugging tools include a memory dump utility, then you can use it to save the µC/Trace recording to a binary file.

This type of utility typically asks you for the memory start address and the number of bytes to save.

The µC/Trace recording buffer is stored in RAM and in the form of a data structure called `RecorderData`.

Find out what is the address of `RecorderData` and calculate the number of bytes according to the number of events. Or, if your debugger tools have a utility to watch expressions, then add a watch for `RecorderData` and write down the address location and the number of bytes from the `filesize` field as shown below:



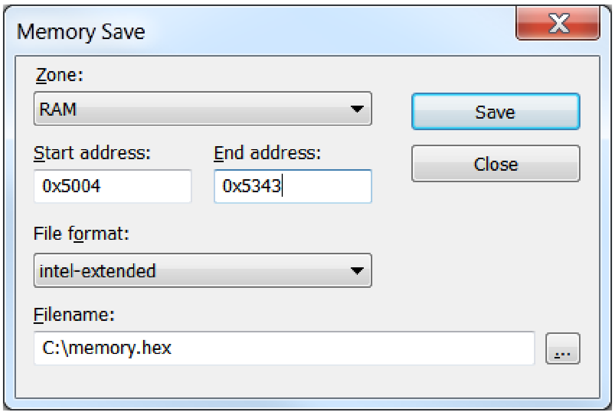
Live Watch

Expression	Value	Location	Type
RecorderData	<struct>	0x5004	RecorderDataType
startmarker0	'.' (0x01)	0x5004	uint8_t
startmarker1	'.' (0x02)	0x5005	uint8_t
startmarker2	'.' (0x03)	0x5006	uint8_t
startmarker3	'.' (0x04)	0x5007	uint8_t
startmarker4	'q' (0x71)	0x5008	uint8_t
startmarker5	'r' (0x72)	0x5009	uint8_t
startmarker6	's' (0x73)	0x500A	uint8_t
startmarker7	't' (0x74)	0x500B	uint8_t
startmarker8	'ñ' (0xF1)	0x500C	uint8_t
startmarker9	'ò' (0xF2)	0x500D	uint8_t
startmarker10	'ó' (0xF3)	0x500E	uint8_t
startmarker11	'ô' (0xF4)	0x500F	uint8_t
version	10914	0x5010	uint16_t
minor_version	'.' (0x02)	0x5012	uint8_t
irq_priority_order	'.' (0x01)	0x5013	uint8_t
filesize	13288	0x5014	uint32_t
numEvents	2000	0x5018	uint32_t
maxEvents	2000	0x501C	uint32_t
nextFreeIndex	2000	0x5020	uint32_t
bufferIsFull	0	0x5024	uint32_t
frequency	96000000	0x5028	uint32_t
internalErrorOccured	0	0x6440	uint32_t
debugMarker2	-218959118	0x6444	int32_t
systemInfo	"Tracealy...	0x6448	char[80]
debugMarker3	-202116109	0x6498	int32_t
eventData	"- "	0x649C	uint8_t[8000]
endOfSecondaryBlocks	0	0x83DC	uint32_t
endmarker4	'q' (0x71)	0x83E4	uint8_t

Live Watch | Locals

Figure 7-2 **Recording Buffer**

Once you have the address and number of bytes, you can use a memory dump utility to save the contents of `RecorderData` to a binary file as shown below:


Memory Save

Zone: RAM

Start address: 0x5004

End address: 0x5343

File format: intel-extended

Filename: C:\memory.hex

Figure 7-3 **Memory Dump Utility**

Once you have the file saved in your host PC, all you have to do is open the µC/Trace Analyzer application for Windows and open the file from the *File* -> *Open* menu.

Appendix

A

μC/Trace Analysis

The μC/Trace Analyzer application gets installed with its own HTML based documentation that describes each analysis tool in detail. This appendix is designed to be a quick reference to analyze some of the most representative μC/OS-III kernel events.

A-1 TASK SCHEDULING EVENTS

The main trace view provides all recorded information on a vertical time line. This view is complemented by over a dozen additional views providing high level overviews or focused views from different perspectives. The task scheduling is presented using color coded rectangles, where the color helps to identify the task or ISR.

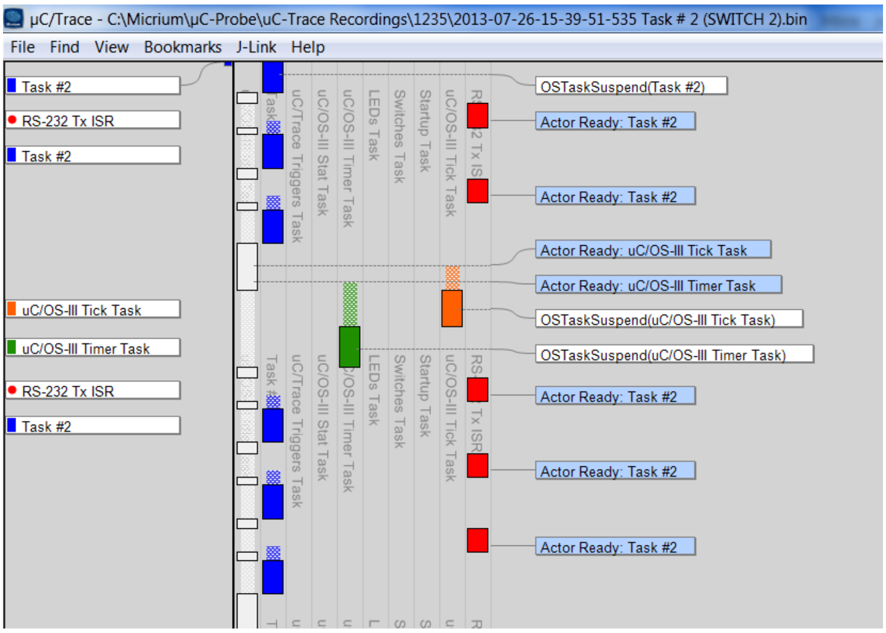
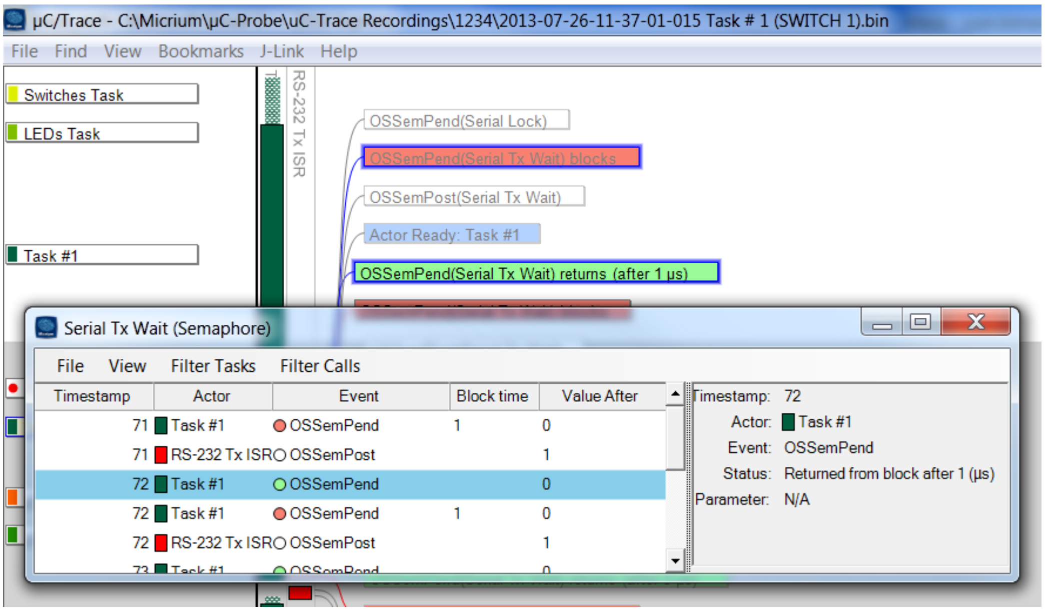

μC/Trace - C:\Micrium\μC-Probe\uC-Trace Recordings\1235\2013-07-26-15-39-51-535 Task # 2 (SWITCH 2).bin

Figure A-1 **Task Scheduling Events**

Notice in Figure A-1 that the ISR represented by the red rectangle does not have that dotted pattern above its rectangle like the other tasks do. That is because µC/OS-III context switches to the ISR immediately while the other tasks have to wait for µC/OS-III to schedule their execution. In other words, that dotted pattern gives you an idea of how long your tasks are waiting for their turn at the CPU.

A-2 SEMAPHORE EVENTS

µC/OS-III semaphores make their mark on the trace by logging the kernel service calls `OSSemPend()` and `OSSemPost()` as shown in Figure A-2:


µC/Trace - C:\Micrium\µC-Probe\uC-Trace Recordings\1234\2013-07-26-11-37-01-015 Task # 1 (SWITCH 1).bin
File Find View Bookmarks J-Link Help
Switches Task
LEDs Task
Task #1
RS-232 Tx ISR
Serial Tx Wait (Semaphore)
File View Filter Tasks Filter Calls

OSSemPend(Serial Lock)
OSSemPend(Serial Tx Wait) blocks
OSSemPost(Serial Tx Wait)
Actor Ready: Task #1
OSSemPend(Serial Tx Wait) returns (after 1 µs)

Timestamp	Actor	Event	Block time	Value After
71	🟩 Task #1	● OSSemPend	1	0
71	🟥 RS-232 Tx ISR	○ OSSemPost		1
72	🟩 Task #1	○ OSSemPend		0
72	🟩 Task #1	● OSSemPend	1	0
72	🟥 RS-232 Tx ISR	○ OSSemPost		1
73	🟩 Task #1	○ OSSemPend		0

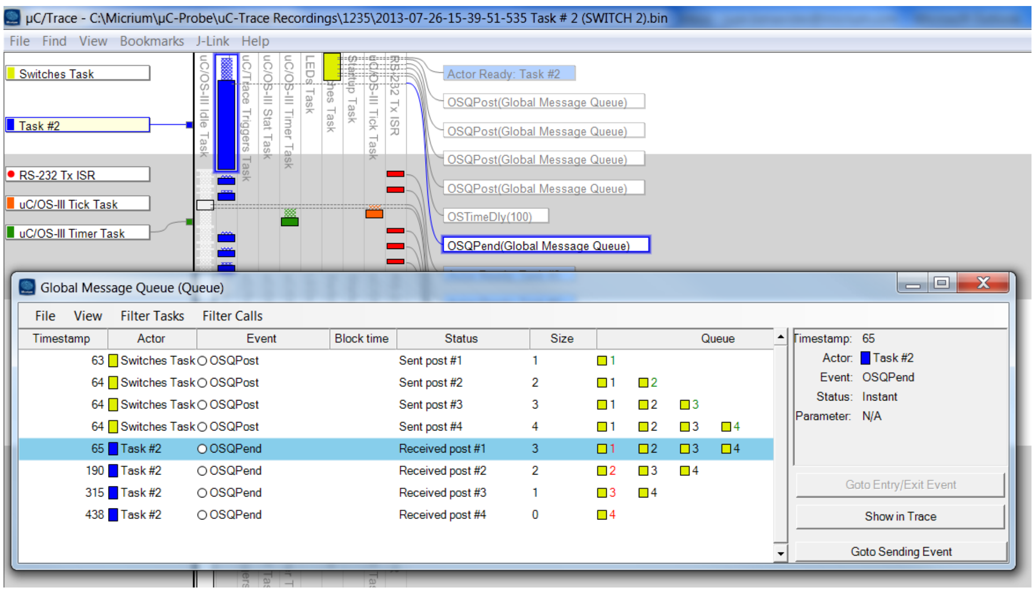
Timestamp:	72
Actor:	🟩 Task #1
Event:	OSSemPend
Status:	Returned from block after 1 (µs)
Parameter:	N/A

Figure A-2 **µC/OS-III Semaphore Events**

If the call to `OSSemPend()` results in a blocking call because the semaphore has not being signaled yet, the viewer will display the event with a red rectangle and when the semaphore finally gets signaled it will be displayed again as a green rectangle along with the blocking time between parentheses as shown in the image above.

A-3 MESSAGE QUEUE EVENTS

µC/Trace records µC/OS-III message queue events such as `OSQPend()` and `OSQPost()` as shown in Figure A-3:


µC/Trace - C:\Micrium\µC-Probe\uC-Trace Recordings\1235\2013-07-26-15-39-51-535 Task # 2 (SWITCH 2).bin

Actor Ready: Task #2
OSQPost(Global Message Queue)
OSQPost(Global Message Queue)
OSQPost(Global Message Queue)
OSQPost(Global Message Queue)
OSTimeDly(100)
OSQPend(Global Message Queue)

Global Message Queue (Queue)

Timestamp	Actor	Event	Block time	Status	Size	Queue
63	Switches Task	OSQPost		Sent post #1	1	1
64	Switches Task	OSQPost		Sent post #2	2	1 2
64	Switches Task	OSQPost		Sent post #3	3	1 2 3
64	Switches Task	OSQPost		Sent post #4	4	1 2 3 4
65	Task #2	OSQPend		Received post #1	3	1 2 3 4
190	Task #2	OSQPend		Received post #2	2	2 3 4
315	Task #2	OSQPend		Received post #3	1	3 4
438	Task #2	OSQPend		Received post #4	0	4

Timestamp: 65
Actor: Task #2
Event: OSQPend
Status: Instant
Parameter: N/A

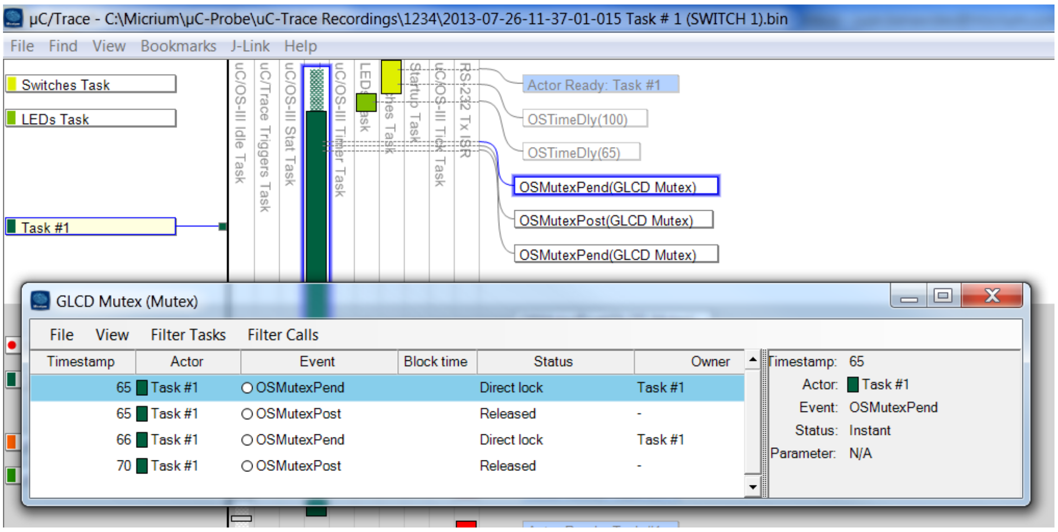
Figure A-3 **µC/OS-III Message Queue Events**

Notice in the example illustrated above how the *Switches Task* posts 4 messages into the *Global Message Queue* and notice how in turn *Task #2* consumes each message.

The diagram helps you keep track of each message by making sure no message is being dropped or the queue is overflowing.

A-4 MUTEX EVENTS

µC/Trace records µC/OS-III mutex events such as `OSMutexPend()` and `OSMutexPost()` as shown in Figure A-4:



Timestamp	Actor	Event	Block time	Status	Owner
65	🟩 Task #1	○ OSMutexPend		Direct lock	Task #1
65	🟩 Task #1	○ OSMutexPost		Released	-
66	🟩 Task #1	○ OSMutexPend		Direct lock	Task #1
70	🟩 Task #1	○ OSMutexPost		Released	-

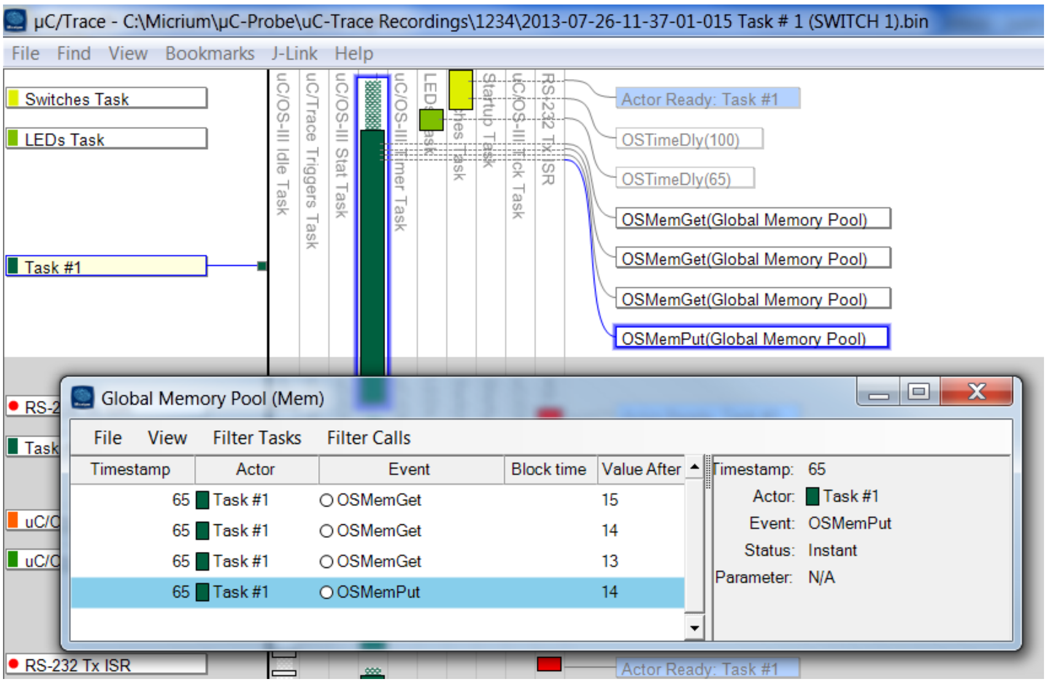
Timestamp:	65
Actor:	🟩 Task #1
Event:	OSMutexPend
Status:	Instant
Parameter:	N/A

Figure A-4 **µC/OS-III Mutex Events**

Notice how the tool shows you which task owns the mutex and similar to semaphores, if pending on a mutex results in a blocking call it will show you for how long it had to wait for the mutex to be released. This is a great tool to identify possible resource starvation cases.

A-5 MEMORY PARTITION EVENTS

µC/Trace records µC/OS-III memory partition events such as `OSMemGet()` and `OSMemPut()` as shown in Figure A-4:



Timestamp	Actor	Event	Block time	Value After
65	🟩 Task #1	○ OSMemGet		15
65	🟩 Task #1	○ OSMemGet		14
65	🟩 Task #1	○ OSMemGet		13
65	🟩 Task #1	○ OSMemPut		14

Timestamp:	65
Actor:	🟩 Task #1
Event:	OSMemPut
Status:	Instant
Parameter:	N/A

Figure A-5 **µC/OS-III Memory Partition Events**

The image above shows the example of a memory partition of 16 blocks of memory. Because the tool keeps track of how many blocks of memory are left in the memory partition, notice how *Task #1* first acquires three blocks of memory and then releases just one of them.

Appendix A

Frequently Asked Questions (FAQs)

- **What is the difference between the Evaluation Edition and Professional Edition of µC/Trace?**

  The Evaluation Edition of µC/Trace gives you the full functionality of the Professional Edition but for a limited period of time of 30 days. The best way to experience the differences is to download and try for yourself. You can easily switch between the two editions using the “Demo” option on the Welcome screen and under the File menu.

- **Where do I find the Evaluation Edition?**

  Both editions are included in our single installer. The Evaluation Edition does not require a license key. Just use the “Evaluate” option, and take the opportunity to explore the premium features. The evaluation period ends after 30 days.

- **Is µC/Probe required to use µC/Trace?**

  No, µC/Probe is optional.

- **How difficult it is to include µC/Trace into my µC/OS-III embedded project?**

  It is very easy. If your embedded project is running µC/OS-III v3.04.01 or newer, all you need to do is download the µC/Trace Recorder and include 4 C files into your project.

- **How do I get started with µC/Trace?**

  There is a comprehensive User’s Manual in the µC/Trace Downloads section of our website. If you have any questions after reading this, do not hesitate to contact Micriµm at info@micrium.com.

- **What chips are supported by µC/Trace?**

µC/Trace takes advantage of the generic, hardware-independent µC/OS-III code. However, to get high resolution timestamping, it is necessary to read a hardware timer. We provide official pre-configured solutions for some common chips, including the ARM Cortex-M and Renesas RX600 architectures. However, if your hardware platform is not listed, then the port is an easy task to accomplish yourself; it is a simple matter of defining a set of four macros to match the timer features of your chip. Several corporate users have done this themselves, to perform a proper evaluation.

■ **How much RAM does the µC/Trace recorder library need?**

The recorder RAM buffer can typically be adjusted to fit your system. A small buffer of 5-10 KB can often give a trace of 50-200 ms, depending on the application.

For example, in one demo project on a ARM Cortex-M4 MCU, we used 70 KB of RAM, which allowed for 17,500 events. That gave about 7 seconds of trace history at 2500 events/second, which is a fairly normal rate (10 KB/s).

A buffer of 32KB may last for over ten seconds in a system with low average activity. Note that you can always record continuously, for hours, days or weeks, even with a small buffer, since the ring-buffer mode allows for keeping only the most recent events. For example, when stopped at a breakpoint, the buffer would then contain the trace leading up to the current state. You may also choose to stop the recording when the buffer is full.

■ **Which IDEs/debuggers are supported?**

Most debuggers will work, as long as it can save the RAM buffer in ".bin", ".hex" or ".mch" (MPLAB) formats. In your debugger IDE, open the RAM memory view and use the "save" option (there is usually such an option). Then just open the resulting file – it will locate the trace data in the RAM dump automatically.

■ **I included the µC/Trace code into my µC/OS-III project, but it does not compile. The error message is: "Struct OS_TCB has no field TaskID". What am I doing wrong?**

Make sure your project is running µC/OS-III version 3.04.01 or newer.

- **I included the µC/Trace code into my µC/OS-III project, but it does not compile. The error message is: "Struct OS_TCB has no field NamePtr". What am I doing wrong?**

  Make sure your `os_cfg.h` enables debugging by setting the macro `OS_CFG_DBG_EN` to 1.

- **I included the µC/Trace code into my µC/OS-III project, but it does not compile. The error message is: "Duplicate symbol "_vTraceTaskInstanceIsFinished" in trace.o". What am I doing wrong?**

  In order to make it easy to include µC/Trace, we have a C file that includes other C files. This way you only have to insert 4 files into your project. To fix this issue you can either make sure you only inserted 4 files into your project (`trace.c/h` and `trace_cfg.c/h`) or go ahead and remove the appropriate `#include`s in `trace.c`.

- **What does the label "(startup)" mean?**

  This is displayed as a placeholder "task", representing the initial time interval before any task activation has been recorded.

- **My trace does not load, or is showing only a single task named "(startup)". What's wrong?**

  Check the following things:

  - Is `TRACE_INIT()` called? This must be called early, before calling any kernel or recorder function.
  - Is `TRACE_START()` called?
  - Double-check that you have enabled µC/Trace in `os_cfg.h`:

    ```
    #define TRACE_CFG_EN 1
    ```

  - Are there any error messages from the recorder? This is displayed when opening a trace file with the µC/Trace Windows application. The most common error message is from insufficient allocation of object handles, i.e., for tasks, queues and other kernel objects. This is configured in trace_cfg.h in macros named `TRACE_CFG_MAX_TASK`, `TRACE_CFG_MAX_MUTEX`, etc. Using the debugger, add a watch for `RecorderDataPtr` and check if the field `internalErrorOccured` is 0 or 1. If 1, then there has been an error and the `systemInfo` field then holds the error message. Such error messages are stored by calls to `vTraceError()`, so placing a break-point there is often a good idea.

- Are you reading the right memory area when saving the RAM contents in your debugger? Make sure that trace data block (`RecorderDataPtr`) is fully included.
- Is your debugger output format supported? Currently, we support binary files (use file suffix ".bin" or ".dump"), which can be generated by the GCC debugger (GDB), by J-Link Commander and by Renesas HEW. It also supports the Intel Hex format, which can be generated using IAR Embedded Workbench (use file suffix ".hex"). We also support ".mch" files from MPLAB. If your debugger can provide RAM dumps but only in other formats, please contact info@micrium.com.

If you still have problems, please contact Micriµm support (info@micrium.com).

- **I get an error message "Object table lookup with invalid object handle or object class!" when opening the trace. What's wrong?**

  This error usually means the trace recording functions are been called before µC/OS-III has been initialized.

  Make sure to call `TRACE_INIT()` and `TRACE_START()` before `OSInit()`.

- **I get an error message "Warning, Recorder reported error: ..." when opening the trace. What's wrong?**

  Check the following things:

  - The most common problem: double-check that you have added the following line in the very end of your os_cfg.h:

    ```
    #define TRACE_CFG_EN 1
    ```

- In case the message is “Not enough TASK handles…”, “Not enough QUEUE handles …” or similar, you need to increase the value some of the constants in `trace_cfg.h` (`TRACE_CFG_MAX_TASK`, `TRACE_CFG_MAX_Q`, etc.) to better reflect the maximum number of tasks or queues used.
- In other cases, you can get more details using your debugger. Put a break-point inside `vTraceError` (where the error message is stored), and check the context in which it is generated.

How do I enable tracing of interrupt handlers?

Interrupt handlers (or Interrupt Service Routines, ISRs) are not recorded by default. ISRs using µC/OS-III functions, such as `OSTaskQPost()`, are however recorded automatically but without the ISR identity, e.g., “ISR using Queue #1?. To record ISRs properly, you need to add two calls in the interrupt handlers you wish to record. See `TRACE_OS_ISR_BEGIN()` and `TRACE_OS_ISR_END()` for further information. Use `TRACE_OS_ISR_REGISTER()` to set the name of the ISR.

What does the label “ISR #1? mean?

If you have not set a name for the ISR using `TRACE_OS_ISR_REGISTER()`, the ID stored using `TRACE_OS_ISR_BEGIN()` is displayed.

What does the label “ISR using …” mean?

If you have interrupt handlers using µC/OS-III functions, such as `OSTaskQPost()`, but are not calling `TRACE_OS_ISR_BEGIN()` and `TRACE_OS_ISR_END()`, the recorder assumes that the µC/OS-III call was made from an interrupt handler, but does not know the identity of the interrupt handler. In this case, the name is set to “ISR using “, with the queue or semaphore used by the µC/OS-III call. To fix this, add `TRACE_OS_ISR_BEGIN()` and `TRACE_OS_ISR_END()` calls to your interrupt handlers.

I’m not using µC/OS-III. Is there a way to use µC/Trace with RTOS XYZ?

No. µC/Trace is designed to work with µC/OS-III only. However, Percepio has versions of the analyzer for several other RTOS. Please contact support@percepio.com for more information.

What is the relationship between µC/Trace and µC/Probe?

µC/Trace and µC/Probe are part of the embedded systems tools offered by Micriµm.

You can use µC/Probe and a special control designed for µC/Trace called µC/Trace Trigger Control, to trigger a recording from your PC, upload the recording to your PC and launch the µC/Trace Analyzer, which is much nicer than using your debugger's memory dump utility.

- **What is the relationship between Micriµm and Percepio?**

Percepio is a member of the Micriµm partner program, and has created a special version of their recorder and analyzer for µC/OS-III under the name µC/Trace. The name µC/Trace is a trademark of Micriµm.

- **What is the background of µC/Trace?**

The first version of the tool was created by Percepio in 2004, during Dr. Johan Kraft's PhD work at Mälardalen University. A simple trace visualization tool was needed to verify scheduling simulations, but when ABB Robotics began using the tool systematically in 2005 (on VxWorks), the project started its evolution into something much larger. Between 2005 and 2008, the tool evolved to version 1.31 and Percepio learned a lot by implementing trace recorders for various real-time operating systems, such as VxWorks, OSE, RTXC Quadros and FreeRTOS, on various industrial systems spanning from small microcontrollers to large Intel-based systems. The second generation of the tool was initiated in 2009 and released in 2011.

In 2013 Percepio and Micriµm entered into an agreement to develop and commercialize a version of the product built into µC/OS-III under the name µC/Trace.

Appendix B

Bibliography

- Labrosse Jean. *µC/OS-III The Real-Time Kernel*. Micriµm Press, ISBN 978-0-98223375-3-0, 2009.