µC/Probe®

Graphical Live Watch®

Target Manual
V4.0

Micriµm® Press

Weston, FL 33326

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

www.micrium.com

600-uC-Probe-Target-003

Table of Contents

Chapter

1

Introduction

µC/Probe is a Windows application designed to read and write the memory of any embedded target processor during run-time. Memory locations are mapped to a set of virtual controls and indicators placed on a dashboard. Figure 1-1 shows an overview of the system and data flow.

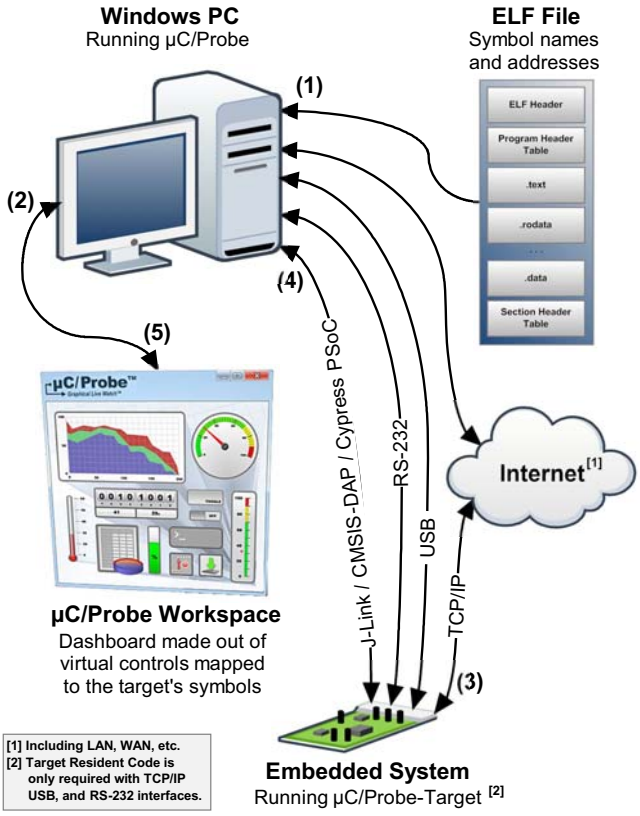


Figure 1-1 **µC/Probe Data Flow Diagram**

F1-1(1) 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) and creates a catalog known as *symbol browser*, which will be used by you during design-time to select the symbols you want to display on your dashboard. This document provides information on installing the µC/Probe Target C files and building the ELF file.

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.

F1-1(2) 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. Refer to the document µC/Probe User's Manual for more information on creating your own dashboard with µC/Probe.

F1-1(3) Before proceeding to the run-time stage, µC/Probe needs to be configured to use one of the four communication interfaces: JTAG, USB, 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., buttons and gauges respectively). At the same time, µC/Probe sends commands to write the memory locations associated with each virtual control (i.e., buttons on a click event).

F1-1(4) In the case of a reading request, the embedded target responds with the latest value. In the case of a write command, the embedded target responds with an acknowledgement. This document provides all the information you need in regards to the firmware that implements the communication interface that runs on the embedded target.

F1-1(5) µC/Probe parses the responses from the embedded target and updates the virtual controls and indicators.

Please refer to the document µC/Probe User's Manual for anything related to the Windows PC side of the system.

This document only provides information about the firmware that resides on the Embedded System which we will call *µC/Probe Target C files*.

The µC/Probe Target C files for communication purposes are only necessary in case you want to communicate through USB, TCP/IP or RS-232.

If you are unsure of which communication interface to use, try the communication options advisor in Figure 1-2:

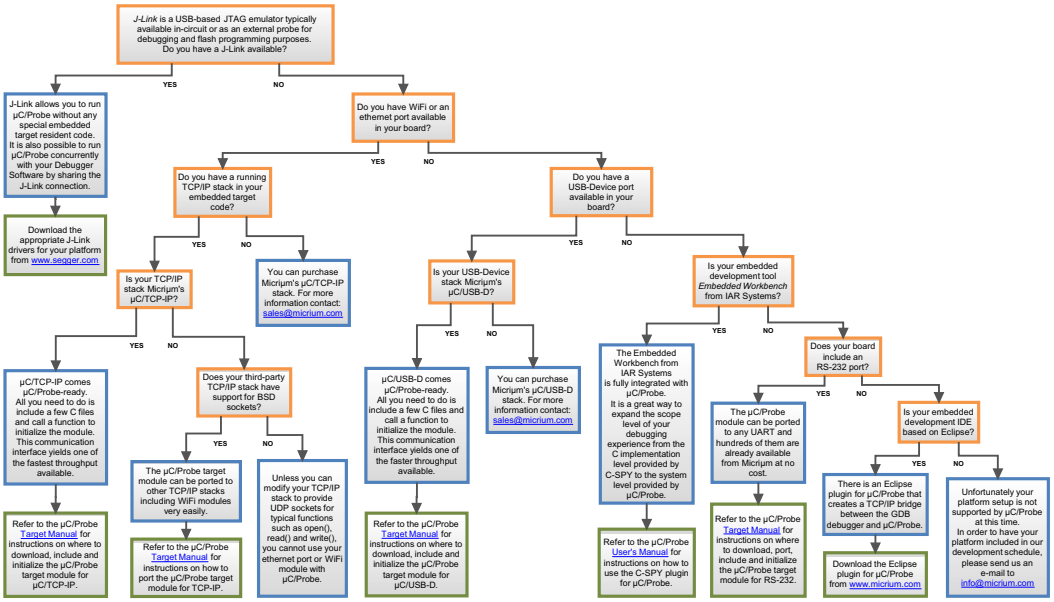


Figure 1-2 **Communication Options Advisor**

Chapter

2

µC/Probe-Target Modules

The µC/Probe-Target modules are the C files that reside on the embedded target and respond to the requests from µC/Probe running on the Windows PC.

This chapter aims at providing a brief introduction to these modules.

The latest µC/Probe-Target modules can be downloaded from the µC/Probe Home Page at `http://www.micrium.com/probe`.

The modules include the following groups of files:

- Application Module

  The Application Module includes your own embedded application C files. Some of them will have to be tweaked to include some configuration macros and initialization calls to the Target Communication Module.
  See Chapter 3, "Configuring µC/Probe-Target" on page 14 and Chapter 4, "Initializing µC/Probe-Target" on page 20 for more information on the tweaks you need to do to your application code.

- Target Communication Module

  The Target Communication Module consists of the files from Micriµm and includes two types of communication interfaces:

  - DCC (Debug Communication Channel) over JTAG

  - Generic (RS-232, TCP/IP, USB, etc.)

The embedded target can potentially implement all the communication interfaces at the same time and the Windows PC can have multiple instances of µC/Probe connected through each of the supported interfaces.

The µC/Probe-Target modules are illustrated in the form of a flow diagram in Figure 2-1:

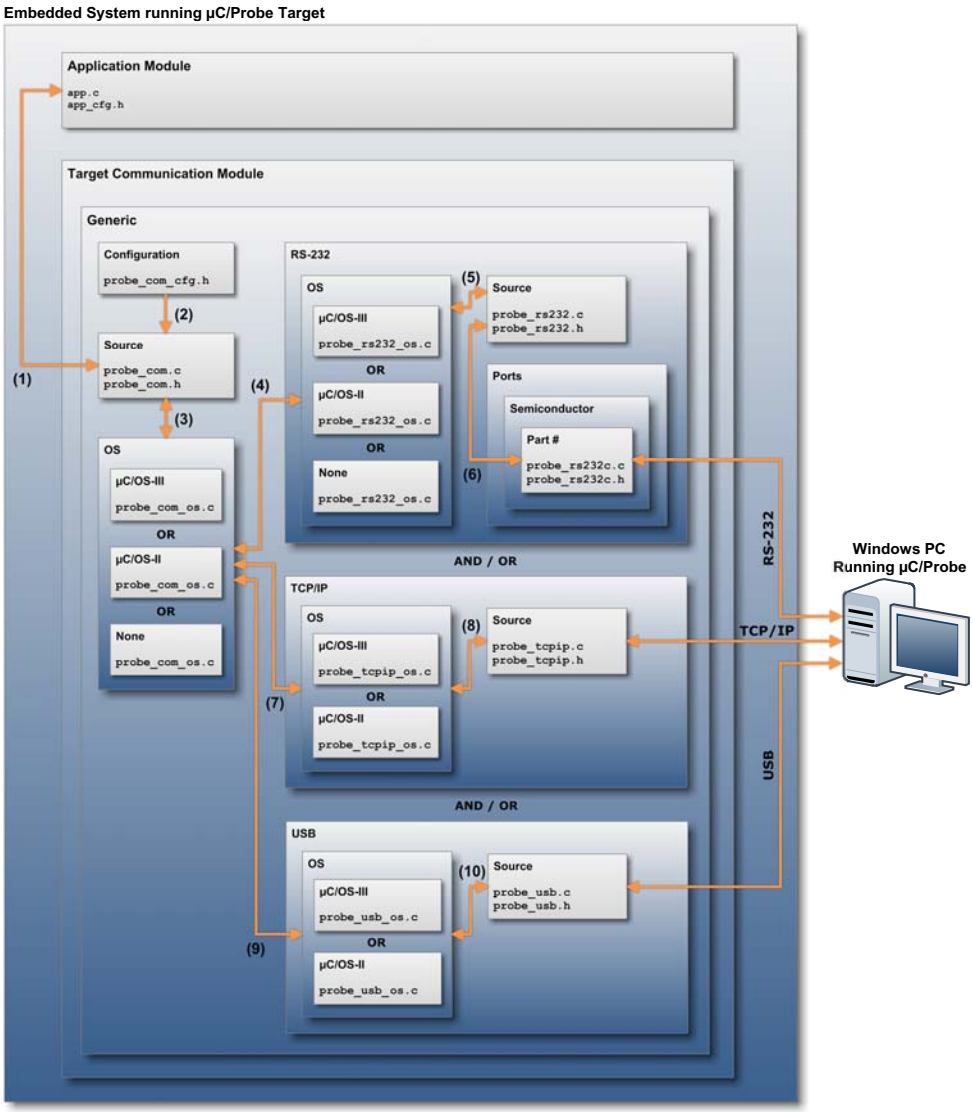


Figure 2-1 **µC/Probe-Target Data Flow Diagram**

F2-1(1) If you are planning to use one of the Generic interfaces such as USB, RS-232 or TCP/IP, then Micriµm provides code that is common for all.

F2-1(2) The generic communication module relies on a set of macros that specify its configuration. This configuration is specific to your application, therefore, we recommend to copy the file `probe_com_cfg.h` over to your application module or copy and paste the macros into your own application configuration header file (i.e., `app_cfg.h`). See Chapter 3, "Configuring µC/Probe-Target" on page 14 for more information on this configuration file.

F2-1(3) µC/Probe has been designed to work best with an RTOS, however, the Generic module source files are OS-independent and an abstraction layer allows you to select between µC/OS-III, µC/OS-II, the kernel of your choice or no OS at all.

F2-1(4) If the communication interface of your choice is RS-232 then the OS layers in this module implement a series of wrapper functions for two kernel objects:

- The RS-232 task that waits for packets to be received and formulates responses.
- A semaphore that provides the signaling mechanism for the reception of packets.

F2-1(5) The RS-232 module source files are OS-independent and implement the Reception and Transmission state machines responsible for the parsing of packets and the formulation of responses.

F2-1(6) The RS-232 module source files are also independent of the RS-232 driver.

The RS-232 driver is implemented in the Ports module and most of the time Micriµm has a port for almost every semiconductor's part number. In case a port is not available for your part #, you will have to implement nine functions in this ports module. See Appendix A, "Porting the RS-232 Communication Module" on page 38 for more information.

F2-1(7) The TCP/IP communication interface provides a faster throughput and longer connection distances. In a similar manner to the RS-232 module, this module is designed with a task model that makes use of semaphores. These kernel objects are abstracted by this OS layer.

F2-1(8) This interface requires TCP/IP sockets that support the UDP protocol. If you are using Micriµm's TCP/IP stack, then this module is ready to go. If you are using any other TCP/IP stack, then you need to tweak the TCP/IP module's source files, but chances are, since this module is written using µC/TCP-IP Berkley (BSD) sockets, that the changes would be minimum. See Appendix A, "Porting the TCP/IP Communication Module" on page 48 for more information.

F2-1(9) The USB communication interface provides a fast throughput. In a similar manner to the RS-232 and TCP/IP module, this module is designed with a task model that makes use of semaphores. These kernel objects are abstracted by this OS layer.

F2-1(10) This interface requires Micriµm's µC/USB-Device stack and its Vendor class, which are sold separately. Please contact us at sales@micrium.com for more details.

Chapter

3

Configuring µC/Probe-Target

This chapter aims at providing a detailed description of the tweaks you need to do to your application level code in order to run the µC/Probe-Target modules.

3-1 CONFIGURATION SETTINGS

The generic communication module that includes the USB, RS-232 and TCP/IP interfaces offers an abstraction layer for application level configuration. The configuration file, regardless of the communication interface you choose is the same.

There is a template configuration file located at the path illustrated in Figure 3-1:

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        └──Cfg
                            └──Template
                                probe_com_cfg.h
```

Figure 3-1 **µC/Probe-Target Configuration Template**

Since the settings are specific to an application, you can either move this file to your application folder or copy and paste the contents to your application master configuration file (i.e., `app_cfg.h`).

Please keep in mind that future releases of µC/Probe-Target might include more configuration options and your `probe_com_cfg.h` may require modification should enhancements to the µC/Probe-Target code occur.

3-1-1 GENERAL CONFIGURATION SETTINGS

Listing 3-1 shows the macros that enable or disable the two generic communication interfaces. Use the definitions from the µC/LIB module DEF_ENABLED and DEF_DISABLED to control their availability:

```
/*
*********************************************************************************************
*                          COMMUNICATION METHOD CONFIGURATION
*********************************************************************************************
*/

#define  PROBE_COM_CFG_RS232_EN         DEF_ENABLED   /* RS-232 availability.            */
#define  PROBE_COM_CFG_TCPIP_EN         DEF_ENABLED   /* TCP/IP availability.            */
#define  PROBE_COM_CFG_USB_EN           DEF_ENABLED   /* USB    availability.            */
```

Listing 3-1 **probe_com_cfg.h: Enabling and disabling the generic communication interfaces**

Listing 3-2 shows the general communication settings that apply to RS-232, USB and TCP/IP:

```
#define  PROBE_COM_CFG_RX_MAX_SIZE            256     /* Config max receive  packet size  */ (1)
#define  PROBE_COM_CFG_TX_MAX_SIZE            256     /* Config max transmit packet size  */

#define  PROBE_COM_CFG_WR_REQ_EN         DEF_ENABLED  /* Config write req availability    */ (2)

#define  PROBE_COM_CFG_STR_REQ_EN        DEF_ENABLED  /* Config string req availability   */ (3)
#define  PROBE_COM_CFG_STR_IN_BUF_SIZE        128     /* Config size of string input  buf */ (4)
#define  PROBE_COM_CFG_STR_OUT_BUF_SIZE      2048     /* Config size of string output buf */ (5)

#define  PROBE_COM_CFG_TERMINAL_REQ_EN   DEF_ENABLED  /* Config terminal availability     */ (6)
#define  PROBE_COM_CFG_STAT_EN           DEF_ENABLED  /* Config statistics/counters       */ (7)
```

Listing 3-2 **probe_com_cfg.h: General communication configuration**

L3-2(1) These couple of settings control the maximum receive and transmit packet sizes, respectively. Larger maximum packet sizes will result in more efficient communication with better throughput. The largest receive and transmit packets will be of comparable size, typically.

L3-2(2) The setting `PROBE_COM_CFG_WR_REQ_EN` controls whether write requests are available. If disabled, the code to handle write requests (which allows the Windows application to write target memory locations) will not be included.

L3-2(3) The setting `PROBE_COM_CFG_STR_REQ_EN` controls whether string requests are available. If disabled, the code to handle string requests will not be included, and the string read/write interface functions will not be available. The string request functionality allows the target to write strings to the Windows application (perhaps indicating program status or event occurrence) or read strings from the Windows application (perhaps entered by the user to control the target).

L3-2(4) If string requests are enabled, the size of the input buffer, in bytes, must also be configured in `PROBE_COM_CFG_STR_IN_BUF_SIZE`.

L3-2(5) If string requests are enabled, the size of the output buffer, in bytes, must also be configured in `PROBE_COM_CFG_STR_OUT_BUF_SIZE`.

L3-2(6) The setting `PROBE_COM_CFG_TERMINAL_REQ_EN` controls whether terminal requests are available. If disabled, the code to handle terminal requests will not be included, and the terminal output interface function will not be available. The terminal request functionality allows the Windows application to execute terminal commands on the target (e.g., standard UNIX commands like cd or cat) and receive output generated by the execution.

L3-2(7) The setting `PROBE_COM_CFG_STAT_EN` controls whether statistics and counters will be maintained.

3-1-2 RS-232 CONFIGURATION SETTINGS

The same configuration file `probe_com_cfg.h` contains the settings that modify the behavior of the RS-232 module. Some of the code is shown in Listing 3-3:

```
#define  PROBE_RS232_CFG_RX_BUF_SIZE  PROBE_COM_CFG_RX_MAX_SIZE /* Config Rx buf size     */ (1)
#define  PROBE_RS232_CFG_TX_BUF_SIZE  PROBE_COM_CFG_TX_MAX_SIZE /* Config Tx buf size     */

#define  PROBE_RS232_CFG_PARSE_TASK_EN            DEF_ENABLED  /* Enable parsing task    */ (2)
#define  PROBE_RS232_CFG_TASK_PRIO                       12    /* Config task priority   */ (3)
#define  PROBE_RS232_CFG_TASK_STK_SIZE                  128    /* Config task stack      */

#define  PROBE_RS232_UART_0                               1
#define  PROBE_RS232_UART_1                               2
#define  PROBE_RS232_UART_2                               3
#define  PROBE_RS232_UART_3                               4
#define  PROBE_RS232_UART_4                               5
#define  PROBE_RS232_UART_5                               6
#define  PROBE_RS232_UART_6                               7
#define  PROBE_RS232_UART_6                               8
#define  PROBE_RS232_UART_DBG                            63

#define  PROBE_RS232_CFG_COMM_SEL         PROBE_RS232_UART_1    /* Config UART selection  */ (4)
```

Listing 3-3 **probe_com_cfg.h: RS-232 Communication Settings**

L3-3(1) These two settings determine the sizes of the buffers used for packets, one for receive and another for transmit. These are, effectively, the sizes of the maximum receivable and transmittable packets; consequently these should generally be configured to the maximum packet sizes.

L3-3(2) The setting `PROBE_RS232_CFG_PARSE_TASK_EN` determines whether an RS-232 task will be created to parse received packets and generate replies. Otherwise, packets will be parsed at interrupt-level.

L3-3(3) If the RS-232 parsing task is enabled, these two settings determine the priority and stack size of the task.

L3-3(4) The setting `PROBE_RS232_CFG_COMM_SEL` determines which UART or serial communication interface will be used for communication.

3-1-3 TCP/IP CONFIGURATION SETTINGS

The same configuration file `probe_com_cfg.h` contains the settings that modify the behavior of the TCP/IP module. The code related to TCP/IP is shown in Listing 3-4:

```
#define  PROBE_TCPIP_CFG_RX_BUF_SIZE  PROBE_COM_CFG_RX_MAX_SIZE /* Config RX buf size         */ (1)
#define  PROBE_TCPIP_CFG_TX_BUF_SIZE  PROBE_COM_CFG_TX_MAX_SIZE /* Config Tx buf size         */

#define  PROBE_TCPIP_CFG_TASK_PRIO               13             /* Config TCP/IP task priority*/ (2)
#define  PROBE_TCPIP_CFG_TASK_STK_SIZE          256             /* Config TCP/IP task stack   */

#define  PROBE_TCPIP_CFG_PORT                  9930             /* Config server port         */ (3)
```

Listing 3-4 **probe_com_cfg.h: TCP/IP Communication Settings**

L3-4(1) These two settings determine the sizes of the buffers used for packets, one for receive and another for transmit. These are, effectively, the sizes of the maximum receivable and transmittable packets; consequently these should generally be configured to the maximum packet sizes.

L3-4(2) The µC/Probe TCP/IP interface requires an RTOS and a task is created by default to parse received packets and generate replies. These two settings determine the priority and stack size of the task.

L3-4(3) The setting `PROBE_TCPIP_CFG_PORT` determines which port number the UDP server on the embedded target will be listening from.

3-1-4 USB CONFIGURATION SETTINGS

The same configuration file `probe_com_cfg.h` contains the settings that modify the behavior of the USB module. The code related to USB is shown in Listing 3-5:

```
                                                        /* Set Rx and Tx buffer sizes.   */ (1)
#define  PROBE_USB_CFG_RX_BUF_SIZE    PROBE_COM_CFG_RX_MAX_SIZE
#define  PROBE_USB_CFG_TX_BUF_SIZE    PROBE_COM_CFG_TX_MAX_SIZE

                                                        /* Set Rx and Tx timeouts.       */ (2)
#define  PROBE_USB_CFG_RX_TIMEOUT_MS                100u
#define  PROBE_USB_CFG_TX_TIMEOUT_MS                100u

#define  PROBE_USB_CFG_TASK_PRIO                     12     /* Set task priority.            */ (3)
#define  PROBE_USB_CFG_TASK_STK_SIZE                512     /* Set task stack size.          */

#define  PROBE_USB_CFG_INIT_STACK             DEF_TRUE     /* uC/Probe will init USB stack. */ (4)
```

Listing 3-5 **probe_com_cfg.h: USB Communication Settings**

L3-5(1) These two settings determine the sizes of the buffers used for packets, one for receive and another for transmit. These are, effectively, the sizes of the maximum receivable and transmittable packets; consequently these should generally be configured to the maximum packet sizes.

L3-5(2) These two settings determine the amount of time in milliseconds the USB device stack is willing to wait for a response after receiving and transmitting data respectively.

L3-5(3) The µC/Probe TCP/IP interface requires an RTOS and a task is created by default to parse received packets and generate replies. These two settings determine the priority and stack size of the task.

L3-5(4) This setting makes the µC/Probe USB communication module responsible for initializing the µC/USB-Device stack.

Chapter

4

Initializing µC/Probe-Target

If using a generic communication source such as RS-232, USB or TCP/IP, then first the Generic module and then the RS-232, or the USB and, or the TCP/IP interfaces must be initialized from your application level code prior to using µC/Probe as shown in Code Listing 4-1:

```
ProbeCom_Init();              /* Initialize the Generic module       */   (1)

ProbeRS232_Init(115200);      /* Initialize the RS-232 interface     */   (2)
ProbeRS232_RxIntEn();         /* Initialize the RS-232 Rx interrupts */   (3)

ProbeUSB_Init(dev_nbr,
              cfg_hs_nbr,
              cfg_fs_nbr);    /* Initialize the USB interface        */   (4)

ProbeTCPIP_Init();            /* Initialize the TCP/IP interface     */   (5)
```

Listing 4-1 **Initializing µC/Probe-Target**

L4-1(1) Initialize the generic communications module from your application code. This module must be initialized first.

L4-1(2) Initialize the RS-232 communication module from your application code. The argument of `ProbeRS232_Init()` should be the communication baud rate.

L4-1(3) Enable RS-232 receive interrupts. This step must be performed so that packets may be received.

L4-1(4) Initialize the USB communications module by calling the function `ProbeUSB_Init()` which in turn will initialize the Vendor class among other things.

As a reference, you can see the application template file at `$\Micrium\Software\uC-Probe\Target\Communication\Generic\USB\App\app_usbd.c`

L4-1(5) Initialize the TCP/IP communication module. This function call will create a UDP listening socket on the port configured in `probe_com_cfg.h`. A network connection between the target and the PC is required.

Chapter

5

Building µC/Probe-Target

One of the key elements of the system is the embedded target's output file known as ELF file. The ELF file contains a list of all the global variables in your embedded system including the name, data type and address location.

µC/Probe uses this ELF file to map each virtual control and indicator to the variable of your choice.

This chapter describes how to build a valid ELF file that is compatible with µC/Probe.

The process of building an ELF file involves the toolchain of your choice, your application C files, the µC/Probe-Target and other support C files from Micriµm and in some cases, any other third-party pre-compiled C objects or libraries as illustrated in Figure 5-1:

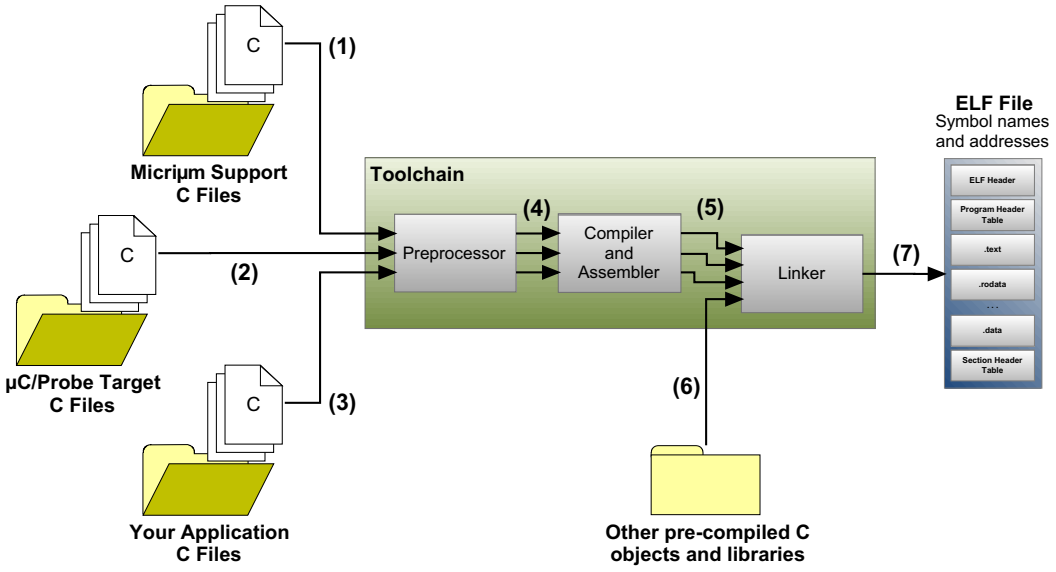


Figure 5-1 **Building an ELF file**

F5-1(1) The support C files from Micriµm include the µC/LIB and µC/CPU modules. These modules implement and define a series of functions, data types and macros required by the µC/Probe-Target C files. Regardless of the communication interface you choose, these files must be included.

F5-1(2) The µC/Probe-Target C files from Micriµm are the files that we discussed in the previous Chapter 2, "µC/Probe-Target Modules" on page 10. These files are the code that includes the communication state machine and drivers for all the supported communication interfaces.

F5-1(3) Your application code does not require major changes to make it work with µC/Probe. It does not require an RTOS either. All you need to do is a couple of simple things:

- Make sure that any variable that you want to display or control from a µC/Probe workspace is declared globally. Any data type goes; from booleans to data structures and everything in between, including arrays.

- Make some calls to initialize the µC/Probe-Target code. Chapter 4, "Initializing µC/Probe-Target" on page 20 shows an example of how to initialize the Target Communication module from the application level.

F5-1(4) Your toolchain's C preprocessor provides the ability to:

- Include the new header files from Micriµm's µC/Probe-Target C Files.

- Do conditional compilation depending on the communication interface of your choice.

- Expand macros to configure settings such as buffer sizes that affect the level of performance and footprint.

See Chapter 3, "Configuring µC/Probe-Target" on page 14 for more information on configuring your application code for the C preprocessor.

F5-1(5) Your toolchain's C compiler needs to be configured with the new include directories from Micriµm's µC/LIB, µC/CPU and µC/Probe-Target C Files. See Chapter 5, "Micrium's Support Files" on page 24 and Chapter 5, "µC/Probe-Target C Files" on page 28 to learn how to configure your C compiler.

F5-1(6) In case your embedded application links to other third-party pre-compiled C objects or libraries, know that as long as there is symbolic information in the DWARF-2 format for them, you will also be able to display and control them from µC/Probe. This group may include the OS of your choice. If you chose Micriµm's µC/OS-II or µC/OS-III then refer to the books from Micriµm Press for more information on building a µC/OS-based application. See Appendix H, "Bibliography" on page 94.

F5-1(7) Finally, the toolchain of your choice needs to be capable of generating an ELF file that includes symbolic information for debug purposes in the DWARF-2, -3 or -4 format. That is usually achieved by configuring your toolchain's linker to output debug information. See Appendix C, "ELF Files with DWARF Debug Information" on page 58 for examples of three different toolchains.

5-1 MICRIUM'S SUPPORT FILES

µC/Probe-Target is designed to work best with an OS and in the case of a TCP/IP interface, because of the nature of TCP/IP, an OS is required.

Describing the compilation process for µC/OS-II and µC/OS-III is beyond the scope of this document, but if you have a sample project for either of these, then you have all the support files from Micriµm that you need for µC/Probe-Target, including µC/CPU and µC/LIB.

In case you are planning to use a different OS or no OS at all, the additional support files from Micriµm that you will need to build the µC/Probe-Target are:

- µC/CPU
- µC/LIB

See Appendix H, "Bibliography" on page 94, for more information on µC/OS-II and µC/OS-III.

5-1-1 µC/CPU

The µC/CPU module defines portable data-types and critical section macros for specific processor architectures and compilers. The µC/Probe-Target modules make reference to the portable data types defined by the µC/CPU module, therefore, this module is required and the files you need to include in your compilation process are shown in Figure 5-2.

```
└──Micrium
    └──Software
        └──uC-CPU
            │   cpu_core.c
            │   cpu_core.h                (1)
            │   cpu_def.h
            │
            ├──<Architecture>
            │   └──<Toolchain>
            │           cpu.h
            │           cpu_a.asm          (2)
            │           cpu_c.c
            │
            └──Cfg
                └──Template
                        cpu_cfg.h          (3)
```

Figure 5-2 **Micriµm Support Files: µC/CPU Files to Compile**

F5-2(1) The µC/CPU core files are architecture and toolchain independent. There is no need to edit these files, just need to include the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-CPU`.

F5-2(2) The `<Architecture>` folder is of course named according to the CPU architecture. For example, this folder could be named `ARM-Cortex-A9`. The same thing goes for the `<Toolchain>` folder. Example names for this folder are `GNU` and `IAR`. The files under these folders are of course CPU Architecture and Toolchain dependent. If the CPU architecture of your choice and the toolchain of your choice is listed in these folders, then you are not required to do anything to the files except for:

- Insert the following path into your toolchain's preprocessor include paths `$/Micrium/Software/uC-CPU/<Architecture>/<Toolchain>`

- Include the header file `cpu.h` using the preprocessing directive `#include` as shown in Listing 5-1:

```
#include <cpu.h>
```

Listing 5-1 **Including the µC/CPU module**

F5-2(3) The µC/CPU module refers to a series of configuration macros that modify the behavior of the module to fit a specific application. Therefore, a configuration template is provided for you to configure the module from the application level. Either copy the contents of this file into your application's configuration file (i.e., `app_cfg.h`), or place the file itself into your application folder.

5-1-2 µC/LIB

The µC/LIB module replaces some of the C standard library functions in order to simplify third-party certification. The µC/Probe-Target modules make reference to the functions and macros defined by the µC/LIB module, therefore, this module is required and the files you need to include in your compilation process are shown in Figure 5-3:

```
└──Micrium
    └──Software
        └──uC-LIB
            │   lib_ascii.c
            │   lib_ascii.h
            │   lib_def.h
            │   lib_math.c
            │   lib_math.h
            │   lib_mem.c                    (1)
            │   lib_mem.h
            │   lib_str.c
            │   lib_str.h
            │
            ├──Cfg
            │   └──Template
            │           lib_cfg.h            (2)
            │
            └──Ports
                └──<Architecture>
                    └──<Toolchain>
                            lib_mem_a.asm    (3)
```

Figure 5-3 **Micriµm Support Files: µC/LIB files to compile**

F5-3(1) The µC/LIB core files are architecture and toolchain independent. No need to edit the files, just need to do the following two things:

- Insert the path `$/Micrium/Software/uC-LIB` into your toolchain's preprocessor include paths.
- Include the header files `lib_def.h`, `lib_ascii.h`, `lib_math.h`, `lib_mem.h` and `lib_str.h` by using the preprocessing directive `#include` as shown in Code Listing 5-2.

```
#include <lib_def.h>
#include <lib_ascii.h>
#include <lib_math.h>
#include <lib_mem.h>
#include <lib_str.h>
```

Listing 5-2 **Including the µC/LIB module**

F5-3(2) Similar to the µC/CPU module, the µC/LIB module makes reference to a series of configuration macros that modify the behavior of the module to fit a specific application. Therefore, a configuration template is provided for you to configure the module from the application level. Either copy the contents of this file into your application's configuration file (i.e., `app_cfg.h`) or place the file itself into your application folder.

F5-3(3) The `Ports` folder contains an `<Architecture>` folder that is named according to the CPU architecture. For example, this folder could be named `ARM-Cortex-A9`. The same convention is used for the `<Toolchain>` folder. Example names for this folder are `GNU` and `IAR`. The files under these folders are of course CPU Architecture and Toolchain dependent. If the CPU architecture of your choice and the toolchain of your choice are listed in these folders, then you are not required to do anything to the files except for:

- Insert the following path into your toolchain's preprocessor include paths `$/Micrium/Software/uC-LIB/Ports/<Architecture>/<Toolchain>`

5-2 µC/PROBE-TARGET C FILES

5-2-1 RS-232 INTERFACE

If you choose to interface µC/Probe over a serial RS-232 interface, then your compilation process needs to include not only the required Micriµm support modules µC/CPU and µC/LIB but also the files illustrated in Figure 5-4 with a few caveats depending on the OS of your choice.

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        ├──Cfg
                        │   └──Template
                        │          probe_com_cfg.h                (1)
                        │
                        ├──OS
                        │   ├──None
                        │   │      probe_com_os.c
                        │   │
                        │   ├──uCOS-II
                        │   │      probe_com_os.c                 (2)
                        │   │
                        │   └──uCOS-III
                        │          probe_com_os.c
                        │
                        ├──RS-232
                        │   ├──OS
                        │   │   ├──None
                        │   │   │     probe_rs232_os.c
                        │   │   │
                        │   │   ├──uCOS-II
                        │   │   │     probe_rs232_os.c            (3)
                        │   │   │
                        │   │   └──uCOS-III
                        │   │         probe_rs232_os.c
                        │   │
                        │   ├──Ports
                        │   │   └──<Semiconductor>
                        │   │       └──<Part#>
                        │   │             probe_rs232c.c          (4)
                        │   │             probe_rs232c.h
                        │   │
                        │   └──Source
                        │          probe_rs232.c
                        │          probe_rs232.h                  (5)
                        │
                        └──Source
                             probe_com.c
                             probe_com.h                          (6)
```

Figure 5-4 **µC/Probe-Target Modules: RS-232 files to compile**

F5-4(1) The RS-232 module makes reference to a series of configuration macros such as buffer sizes, that modify the behavior of the module to fit a specific application. Therefore, a configuration template is provided for you to configure the module from the application level. Either copy the contents of this file into your application's configuration file (i.e., `app_cfg.h`) or place the file itself into your application folder.

F5-4(2) The OS abstraction layers in this module implement a series of wrapper functions for two kernel objects:

- The RS-232 task that waits for packets to be received and formulates responses.
- A semaphore that provides the signaling mechanism for the reception of packets.

Choose one of these 3 files depending on the OS of your choice.

F5-4(3) The OS layers in this module allow the RS-232 drivers to be OS-independent. Choose one of these 3 files depending on the OS of your choice.

F5-4(4) The RS-232 drivers are most likely available from Micriµm, otherwise see the Appendix A, "Porting the RS-232 Communication Module" on page 38 for more information on porting RS-232. If your semiconductor manufacturer and part number is available from Micriµm, all you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/RS-232/Ports/<Semiconductor>/<Part#>`
- Include the header file `probe_rs232c.h` by using the preprocessing directive `#include` as shown in Listing 5-3

```
#include <probe_rs232c.h>
```

Listing 5-3 **Including the RS-232 driver**

F5-4(5) The RS-232 module source files are OS-independent and implement the Reception and Transmission state machines responsible for the packet parsing and formulation of responses. No need to change these files. All you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/RS-232/Source`
- Include the header file `probe_rs232.h` by using the preprocessing directive `#include` as shown in Listing 5-4:

```
#include <probe_rs232.h>
```

Listing 5-4 **Including the RS-232 module**

F5-4(6) This module contains the code that is common not only for RS-232 but also for any other communication interface such as USB and TCP/IP. There is no need to make any changes to these files, all you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/Source`
- Include the header file `probe_com.h` by using the preprocessing directive `#include` as shown in Listing 5-5:

```
#include <probe_com.h>
```

Listing 5-5 **Including the Generic module**

5-2-2 TCP/IP INTERFACE

If you choose to interface your embedded target with µC/Probe over a TCP/IP interface, then your compilation process needs to include not only the required Micriµm support modules µC/CPU and µC/LIB but also an OS because of the nature of TCP/IP.

A TCP/IP stack that supports UDP sockets (such as Micriµm's µC/TCP-IP) is required. Without taking into account the OS and TCP/IP stack, the files you need to compile for the TCP/IP interface are illustrated in Figure 5-5, with a few caveats depending on the OS of your choice:

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        ├──Cfg
                        │   └──Template
                        │       probe_com_cfg.h                (1)
                        │
                        ├──OS
                        │   ├──None
                        │   │   probe_com_os.c
                        │   │
                        │   ├──uCOS-II
                        │   │   probe_com_os.c                 (2)
                        │   │
                        │   └──uCOS-III
                        │       probe_com_os.c
                        │
                        ├──Source
                        │   probe_com.c
                        │   probe_com.h                        (3)
                        │
                        └──TCPIP
                            ├──OS
                            │   ├──uCOS-II
                            │   │   probe_tcpip_os.c
                            │   │
                            │   └──uCOS-III                    (4)
                            │       probe_tcpip_os.c
                            │
                            └──Source
                                probe_tcpip.c
                                probe_tcpip.h                   (5)
```

Figure 5-5 **µC/Probe-Target Modules: TCP/IP files to compile**

F5-5(1) Similarly to other modules, the TCP/IP module makes reference to a series of configuration macros such as buffer sizes, that modify the behavior of the module to fit a specific application. Therefore, a configuration template is provided for you to configure the module from the application level. Either copy the contents of this file into your application's configuration file (i.e., `app_cfg.h`) or place the file itself into your application folder.

F5-5(2) The OS layers in this module implement a series of wrapper functions for some semaphores used by the generic communications module.

Choose one of these three files depending on the OS of your choice.

F5-5(3) The µC/Probe generic communications module implements the parsing of requests and formulation of responses that are common to not only the TCP/IP interface but also, all the rest of communication interfaces.

There is no need to make any changes to these files, all you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/Source`

- Include the header file `probe_com.h` by using the preprocessing directive `#include` as shown in Listing 5-6:

```
#include <probe_com.h>
```

Listing 5-6 **Including the Generic module**

F5-5(4) The OS layers in this module implement a series of wrapper functions for two kernel objects:

- The TCP/IP task that waits for packets to be received and formulates responses.

- A semaphore that provides the signaling mechanism for the reception of packets.

Choose one of these three files depending on the OS of your choice.

F5-5(5) The TCP/IP module source files are OS-independent and implement the Reception and Transmission state machines responsible for the packet parsing and formulation of responses. No need to change these files. All you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/TCPIP/Source`

- Include the header file `probe_tcpip.h` by using the preprocessing directive `#include` as shown in Listing 5-7:

```
#include <probe_tcpip.h>
```

Listing 5-7 **Including the TCP/IP module**

5-2-3 USB INTERFACE

If you choose to interface your embedded target with µC/Probe over a USB interface, then your compilation process needs to include not only the required Micriµm support modules µC/CPU and µC/LIB but also an OS because of the nature of USB and the µC/USB-Device stack.

A USB device stack that supports a Bulk In and Out endpoints (such as Micriµm's µC/USB-Device) is required. Without taking into account the OS and USB-Device stack, the files you need to compile for the USB interface are illustrated in Figure 5-5, with a few caveats depending on the OS of your choice:

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        ├──Cfg
                        │   └──Template
                        │       probe_com_cfg.h          (1)
                        │
                        ├──OS
                        │   ├──None
                        │   │   probe_com_os.c
                        │   │
                        │   ├──uCOS-II
                        │   │   probe_com_os.c           (2)
                        │   │
                        │   └──uCOS-III
                        │       probe_com_os.c
                        │
                        ├──Source
                        │   probe_com.c                   (3)
                        │   probe_com.h
                        │
                        └──USB
                            ├──App
                            │   app_usbd.c               (4)
                            │
                            ├──OS
                            │   ├──uCOS-II
                            │   │   probe_usb_os.c
                            │   │                         (5)
                            │   └──uCOS-III
                            │       probe_usb_os.c
                            │
                            └──Source
                                probe_usb.c               (6)
                                probe_usb.h
```

Figure 5-6 **µC/Probe-Target Modules: USB files to compile**

F5-6(1) Similarly to other modules, the USB module makes reference to a series of configuration macros such as buffer sizes, that modify the behavior of the module to fit a specific application. Therefore, a configuration template is provided for you to configure the module from the application level. Either copy the contents of this file into your application's configuration file (i.e., `app_cfg.h`) or place the file itself into your application folder.

F5-6(2) The OS layers in this module implement a series of wrapper functions for some semaphores used by the generic communications module.

Choose one of these three files depending on the OS of your choice.

F5-6(3) The µC/Probe generic communications module implements the parsing of requests and formulation of responses that are common to not only the USB interface but also, all the rest of communication interfaces.

There is no need to make any changes to these files, all you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/Source`

- Include the header file `probe_com.h` by using the preprocessing directive `#include` as shown in Listing 5-6:

```
#include <probe_com.h>
```

Listing 5-8 **Including the Generic module**

F5-6(4) The USB interface for µC/Probe requires Micriµm's USB device stack µC/USBD and its Vendor class. You can use this file as a reference on how to initialize the USB Device stack and its Vendor class.

F5-6(5) The OS layers in this module implement a series of wrapper functions for two kernel objects:

- The USB task that waits for packets to be received and formulates responses.

- A semaphore that provides the signaling mechanism for the reception of packets.

Choose one of these three files depending on the OS of your choice.

F5-6(6) The USB module source files are OS-independent and implement the Reception and Transmission state machines responsible for the packet parsing and formulation of responses. No need to change these files. All you need to do is two things:

- Insert the following path into your toolchain's preprocessor include paths: `$/Micrium/Software/uC-Probe/Target/Communication/Generic/USB/Source`

- Include the header file `probe_usb.h` by using the preprocessing directive `#include` as shown in Listing 5-7:

```
#include <probe_usb.h>
```

Listing 5-9 **Including the USB module**

Appendix

A

Porting µC/Probe-Target

Most of the time Micriµm will have a µC/Probe-Target sample application for your evaluation board, in which case you will not have to do anything. In other cases, you have to edit some of the module files to accommodate your particular processor, evaluation board and TCP/IP stack.

This appendix provides instructions for porting the RS-232 and TCP/IP communication interfaces to any other evaluation board, and in the case of the TCP/IP interface to other TCP/IP stack.

A-1 PORTING THE RS-232 COMMUNICATION MODULE

In order to port the RS-232 interface, you have to deal with the files illustrated in Figure A-1:

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        └──RS-232
                            └──Ports
                                └──<Semiconductor>
                                    └──<Part#>
                                        probe_rs232c.c
                                        probe_rs232c.h
```

Figure A-1 **µC/Probe-Target RS-232 port files**

The functions implemented by the RS-232 driver are shown in Table A-1:

Function Name	Description
`ProbeRS232_InitTarget()`	Initialize hardware.
`ProbeRS232_RxIntDis()`	Disable receive interrupts.
`ProbeRS232_RxIntEn()`	Enable receive interrupts.
`ProbeRS232_TxIntDis()`	Disable transmit interrupts.
`ProbeRS232_TxIntEn()`	Enable transmit interrupts.
`ProbeRS232_Tx1()`	Transmit one byte.
`ProbeRS232_RxISRHandler()`	Handle receive interrupts.
`ProbeRS232_TxISRHandler()`	Handle transmit interrupts.
`ProbeRS232_RxTxISRHandler()`	Handle receive and transmit interrupts.

Table A-1 **RS-232 communication module port functions**

Each of the port functions will be described in this section and presented in alphabetical order. The following information is provided for each entry:

- A brief description of the function
- The function prototype
- The filename of the source code
- The #define constant required to enable code for the function
- A description of the arguments passed to the function
- A description of any returned value(s)
- Specific notes and warnings regarding use of the function

A-1-1 ProbeRS232_InitTarget()

```
void ProbeRS232_InitTarget (CPU_INT32U  baud_rate);
```

File	Called from	Code enabled by
probe_rs232c.c	Application: ProbeRS232_Init()	N/A

This function is called to initialize the embedded target's UART or serial communication interface at the specified baud rate.

ARGUMENTS

`baud_rate` Serial baud rate in units of bits-per-second.

RETURNED VALUES

None.

NOTES/WARNINGS

Although the baud rate used may vary from application to application or target to target, other communication settings are constant. The hardware must always be configured for the following:

- No parity
- One stop bit
- Eight data bits

Neither receive nor transmit interrupts should be enabled by this function.

A-1-2 ProbeRS232_RxIntDis()

```
void ProbeRS232_RxIntDis (void);
```

File	Called from	Code enabled by
probe_rs232c.c	Application	N/A

This function disables the RS-232 receive interrupt source.

The application can start, pause and restart packet reception by enabling, disabling and re-enabling the UART or serial communication interface's receive interrupts.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

If supported and, or required by hardware, this function may also need to disable the receive overrun interrupt source.

A-1-3 ProbeRS232_RxIntEn()

```
void ProbeRS232_RxIntEn (void);
```

File	Called from	Code enabled by
probe_rs232c.c	Application	N/A

This function enables the RS-232 receive interrupt source.

The application can start, pause and restart packet reception by enabling, disabling and re-enabling the UART or serial communication interface's receive interrupts.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

If supported and, or required by hardware, this function may also need to enable the receive overrun interrupt source.

A-1-4 ProbeRS232_TxIntDis()

```
void ProbeRS232_TxIntDis (void);
```

File	Called from	Code enabled by
probe_rs232c.c	ProbeRS232_TxHandler() ProbeRS232_TxStart()	N/A

This function disables the RS-232 transmit complete interrupt source.

The RS-232 module relies on a transmit complete or transmit ready interrupt to know when a new byte can be safely transmitted. After transmitting the final byte of a packet, transmit interrupts are disabled.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

None.

A-1-5 ProbeRS232_TxIntEn()

```
void ProbeRS232_TxIntEn (void);
```

File	Called from	Code enabled by
probe_rs232c.c	ProbeRS232_TxHandler() ProbeRS232_TxStart()	N/A

This function enables the RS-232 transmit complete interrupt source.

The RS-232 module relies on a transmit complete or transmit ready interrupt to know when a new byte can be safely transmitted. Transmit complete interrupts are enabled after transmitting the first byte of a packet and remain enabled until the final byte of the packet has been transmitted.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

None.

A-1-6 ProbeRS232_Tx1()

```
void ProbeRS232_Tx1 (CPU_INT08U  c);
```

File	Called from	Code enabled by
probe_rs232c.c	ProbeRS232_TxHandler()	N/A

This function transmits one byte via RS-232.

ARGUMENTS

c The byte to transmit.

RETURNED VALUES

None.

NOTES/WARNINGS

None.

A-1-7 ProbeRS232_Rx/Tx/RxTxISRHandler()

```
void ProbeRS232_RxISRHandler (void);
void ProbeRS232_TxISRHandler (void);
void ProbeRS232_RxTxISRHandler (void);
```

File	Called from	Code enabled by
probe_rs232c.c	ProbeRS232_TxHandler() ProbeRS232_TxStart()	N/A

Three possible ISR handlers are prototyped in the RS-232 communication module header file, although one or more may be declared empty.

On a platform that locates receive and transmit interrupts on separate vectors the following two functions must be implemented:

- `ProbeRS232_RxISRHandler()`
- `ProbeRS232_TxISRHandler()`

Otherwise, the following function should handle the combined interrupts:

- `ProbeRS232_RxTxISRHandler()`

Upon a receive interrupt, the byte should be read from the hardware receive data register and passed to the function `ProbeRS232_RxHandler()`. Upon a transmit complete interrupt (or, on some MCUs/MPUs, transmit data register empty), the RS-232 communication core should be informed via a call to `ProbeRS232_TxHandler()`. Listing 3-2 shows an example of a combined receive and transmit handler.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

None.

EXAMPLE

```
void ProbeRS232_RxTxISRHandler (void)
{
    CPU_INT32U status;
    CPU_INT08U rx_data;

    status = UART0MIS;

    if ((status & UARTINT_TX) == UARTINT_TX) {     /* Chk if byte tx'd.         */
        ProbeRS232_TxHandler();
    }
    if ((status & UARTINT_RX) == UARTINT_RX) {     /* Chk if byte rx'd.         */
        rx_data = (UART0DR & 0xFF);                /* Rd rx'd byte.             */
        ProbeRS232_RxHandler(rx_data);
    }

    UART0ICR = status;                             /* Clr int's.                */
}
```

Listing A-1 **probe_rs232c.c: ProbeRS232_RxTxISRHandler()**

A-2 PORTING THE TCP/IP COMMUNICATION MODULE

The µC/Probe-Target TCP/IP module makes use of Micriµm's µC/TCP-IP Berkeley (BSD) sockets interface. In case you are using a TCP/IP stack different than µC/TCP-IP you are going to have to make changes to the files shown in Figure A-2:

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Communication
                    └──Generic
                        └──TCPIP
                            └──Source
                                    probe_tcpip.c
                                    probe_tcpip.h
```

Figure A-2 **µC/Probe-Target TCP/IP Port files**

The TCP/IP communication module basically consists of a single server. This server waits for a packet on a certain port (default, 9930); upon reception, a reply is generated and transmitted on the same port. By using the common BSD sockets interface, this module achieves near-universality, with some few tweaks (perhaps) necessary. First, it might be necessary to include the header file for your network protocol suite.

In the case of µC/TCP-IP the file `probe_tcpip.h` includes `net.h`, which is the primary header file for µC/TCP-IP, but this will probably need to be changed for a different stack.

If your stack does not allow blocking receives, then a delay may need to be inserted after transmitting and attempting to receive a packet (so that the server task does not monopolize the CPU). Even if your stack does allow blocking receives, the method for setting the timeout will be stack dependent, and `ProbeTCPIP_ServerInit()` should be modified to set this.

Minor identifier modifications may need to be made, depending on the stack's BSD implementation. Table A-1 shows the BSD socket functions that are used by the module. Find the analogous function in your TCP/IP stack and replace them in the file `probe_tcpip.c.`

BSD Sockets Function	Description	Called By
`socket()`	The function creates an endpoint for communication and returns a file descriptor for the socket.	`ProbeTCPIP_ServerInit()`
`bind()`	The function assigns a socket to an address.	`ProbeTCPIP_ServerInit()`
`recvfrom()`	Reads data from the remote host specified by fromAddr into buffer. The socket must be UDP.	`ProbeTCPIP_RxPkt()`
`sendto()`	Writes data to the remote host specified by fromAddr into buffer. The socket must be UDP.	`ProbeTCPIP_TxStart()`
`close()`	The function frees a socket's resources by disconnecting it from the remote host.	`ProbeTCPIP_ServerInit()`

Other things to review in the file `probe_tcpip.c` that may be different in the TCP/IP stack of your choice are the following:

- Socket data type
- Error codes
- Return values on function failure
- Byte ordering

Appendix

B

µC/Probe-Target API Functions & Macro's

Your application interfaces to µC/Probe using any of the functions or macros described in this appendix. Each of the user-accessible function is presented in alphabetical order. The following information is provided for each entry:

- A brief description of the function
- The function prototype
- The filename of the source code
- The #define constant required to enable code for the function
- A description of the arguments passed to the function
- A description of any returned value(s)
- Specific notes and warnings regarding use of the function

B-1 PROBECOM_STRRD()

```
CPU_SIZE_T ProbeCom_StrRd(CPU_CHAR  *pdest,
                          CPU_SIZE_T len);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

The function `ProbeCom_StrRd()` reads string data from the communication module's input buffer.

ARGUMENTS

`pdest` This is a pointer to the destination buffer.

`len` Length of the destination buffer, in octets/characters.

RETURNED VALUES

Number of octets/characters read.

NOTES/WARNINGS

This function implements a non-blocking read. It will read as much data as is already buffered, up to len bytes/characters. The calling application should monitor the return value to see if more data needs to be read. Since this function never blocks, it should not be called in a tight loop without a delay. This function *may* be called from an ISR.

B-2 PROBECOM_STRWR()

```
CPU_SIZE_T ProbeCom_StrWr(CPU_CHAR  *psrc,
                          CPU_SIZE_T len);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

The function `ProbeCom_StrWr()` writes string data to the communication module's output buffer.

ARGUMENTS

`psrc` This is a pointer to the source buffer.

`len` Length of the source buffer, in octets/characters.

RETURNED VALUES

Number of octets/characters written.

NOTES/WARNINGS

This function implements a non-blocking write. It will write as much data as fits into the buffer, up to len bytes/characters. The calling application should monitor the return value to see if more data from the buffer needs to be written. Since this function never blocks, it should not be called in a tight loop without a delay. This function *may* be called from an ISR.

B-3 PROBECOM_TERMINALOUT()

```
CPU_SIZE_T ProbeCom_TerminalOut(CPU_CHAR  *pdest,
                                CPU_SIZE_T len);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

This function outputs data over the terminal.

ARGUMENTS

`pdest` This is a pointer to the destination buffer.

`len` Length of the destination buffer, in octets/characters.

RETURNED VALUES

Number of octets/characters written.

NOTES/WARNINGS

This function implements a blocking write. It will queue the request and wait until all of the data has been buffered or transmitted before returning. Terminal data may ONLY be output while a command is being executed. Generic read/write functionality is provided by the string read/write functions (see Appendix B, "ProbeCom_StrRd()" on page 51 and Appendix B, "ProbeCom_StrWr()" on page 52). This function *must not* be called from an ISR.

B-4 PROBECOM_TERMINALEXECCOMPLETE()

```
void ProbeCom_TerminalExecComplete(void);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

This function outputs data over the terminal.

ARGUMENTS

None.

RETURNED VALUES

None.

NOTES/WARNINGS

This function should be called by the application from the terminal I/O call back function when the current terminal command and associated output processing have been completed.

B-5 PROBECOM_TERMINALEXECSET()

```
CPU_SIZE_T ProbeCom_TerminalExecSet(PROBE_COM_TERMINAL_EXEC_FNCT);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

This function sets that handler that will be invoked to process a terminal command. The handler should be a function with the following prototype:

```
void App_TerminalExecFnct(CPU_CHAR  *pstr,
                          CPU_SIZE_T len);
```

where `pstr` is a pointer to the command string, and `len` is the length of the command string (in characters) excluding the final NULL byte. The command string will *not* include a terminating new line or line feed.

ARGUMENTS

`handler` The function handler that will be invoked.

RETURNED VALUES

None.

NOTES/WARNINGS

The application must call this function in order to specify a terminal command callback function if terminal I/O processing is to be performed.

B-6 PROBECOM_TERMINALINSET()

```
CPU_SIZE_T ProbeCom_TerminalExecSet(PROBE_COM_TERMINAL_EXEC_FNCT);
```

File	Called from	Code enabled by
probe_com.c	Application	N/A

This function sets that handler that will be invoked to process terminal input. The handler should be a function with the following prototype:

```
void App_TerminalInFnct(CPU_CHAR  *pstr,
                        CPU_SIZE_T len);
```

where `pstr` is a pointer to the input string, and `len` is the length of the input string (in characters) excluding the final NULL byte. The input string will *not* include a terminating new line or line feed.

ARGUMENTS

`handler` The function handler that will be invoked.

RETURNED VALUES

None.

NOTES/WARNINGS

The application must call this function in order to specify a terminal input callback function if terminal I/O processing is to be performed.

Appendix

C

ELF Files with DWARF Debug Information

This appendix shows examples of how to configure your toolchain to generate an ELF file with debugging information in the DWARF-2, -3 or -4 format.

These examples are provided as reference, please consult your own toolchain's manual if your IDE is not listed in these examples.

C-1 IAR EWARM

C-1-1 ASSEMBLER OPTIONS

Open the options for your embedded project, select **General Options -> Assembler -> Output** and select the check box **Generate Debug Information** as shown in Figure C-1:

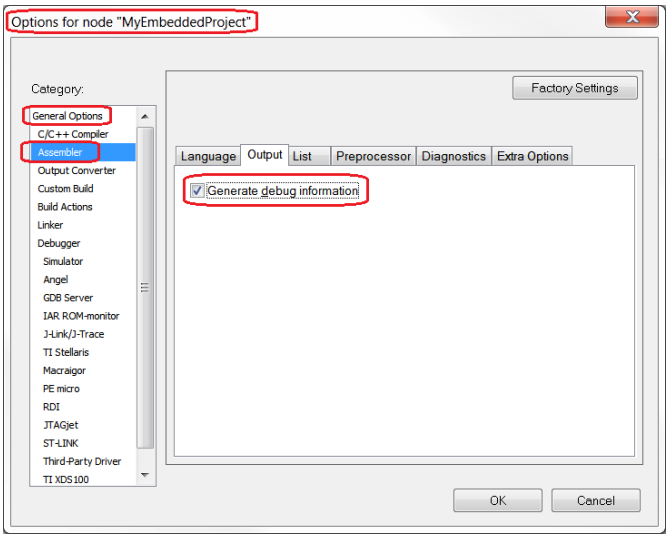


Figure C-1 **IAR EWARM Assembler Debug Information**

C-1-2 C COMPILER OPTIONS

Open the options for your project, select **General Options -> C/C++ Compiler -> Output** and select the check box **Generate Debug Information** as shown in Figure C-2:

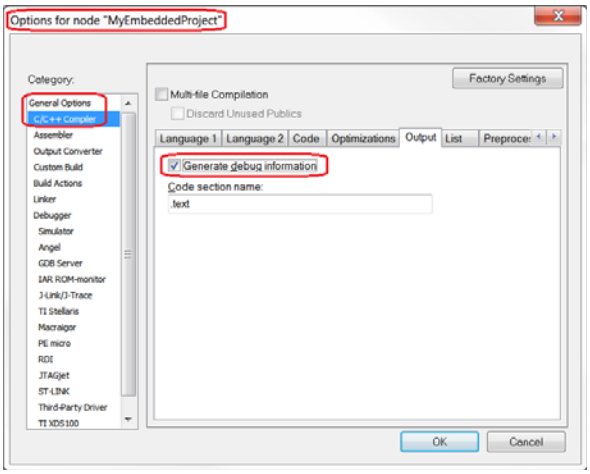


Figure C-2 **IAR EWARM C Compiler Debug Information**

C-1-3 LINKER OPTIONS

Open the options for your project, select **General Options -> Linker -> Output** and select the check box **Include Debug Information in output** as shown in Figure C-3:

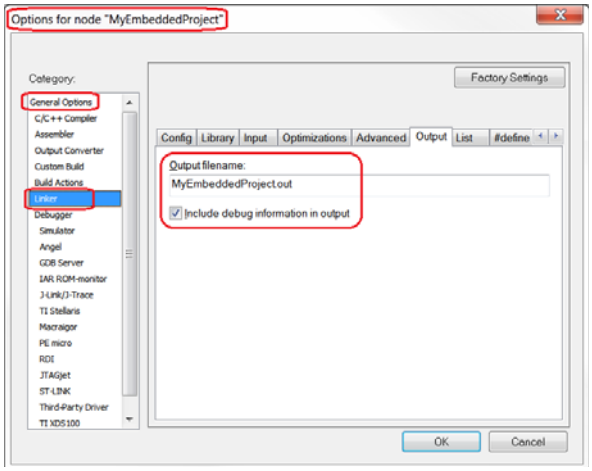


Figure C-3 **IAR EWARM Linker Debug Information**

C-2 KEIL µVISION 4

Open the options for your project, select the **Output** tab and select the check box **Debug Information** as shown in Figure C-4:

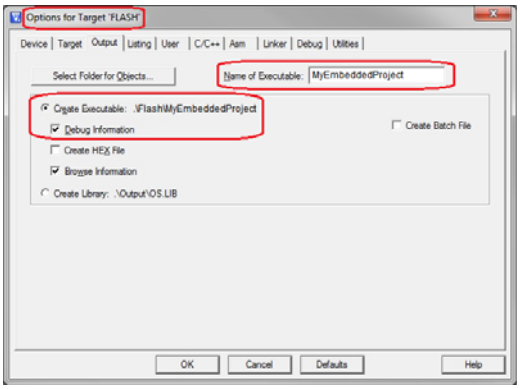


Figure C-4 **Keil µVision 4 Debug Information**

C-3 RENESAS E2STUDIO

Open the properties for your project, select **C/C++ Build -> Renesas Settings -> Compiler -> Object** and select from the drop downs the options shown in Figure C-5:

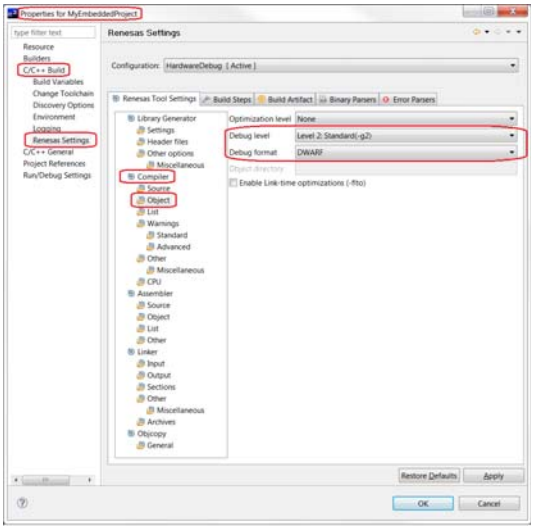


Figure C-5 **Renesas e2Studio C Compiler Debug Information**

Appendix

D

XML-based Custom Symbol Files (CSF)

µC/Probe is capable of parsing XML-based Custom Symbol Files (CSF), which is very useful for those cases where your toolchain is incapable of generating one of the ELF file formats supported by µC/Probe. The best way to create a CSF file is by modifying the template located in your µC/Probe installation directory at:

```
$\Micrium\µC-Probe\Help\uC-Probe-CustomSymbolFile-Template.csf
```

The template is associated with an XSD document that defines the XML schema for CSF files supported by µC/Probe. We recommend using an XML editor capable of providing *IntelliSense* features such as Visual Studio, shown in Figure D-1:.

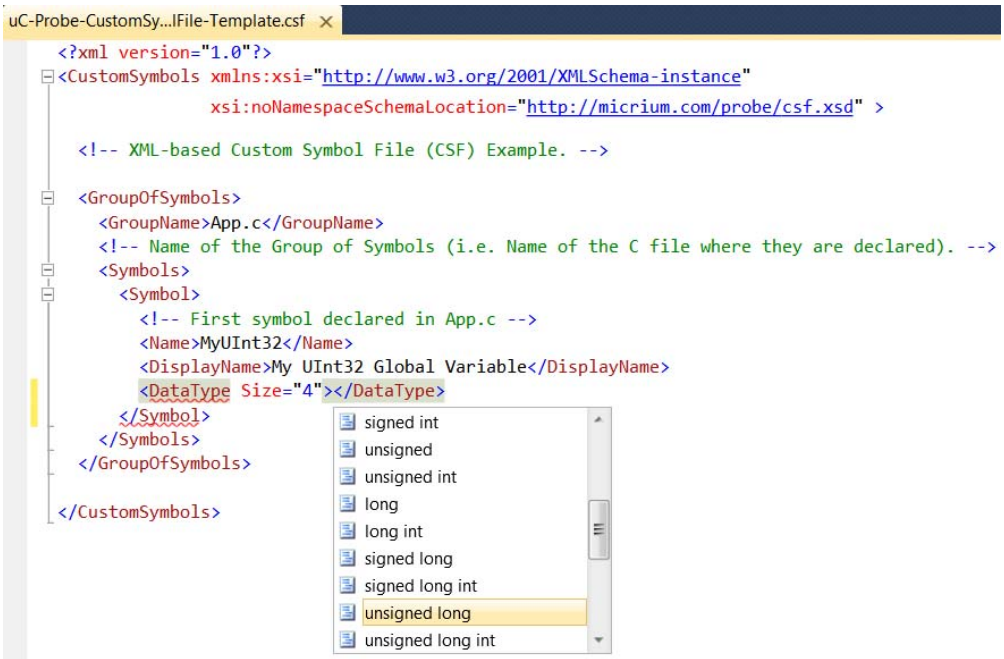


Figure D-1 **Creating a CSF in Visual Studio: Drop-Down Lists**

Visual Studio makes editing your CSF file easier by filling required XML syntax for you. For example, after a schema is associated with your CSF, you get a drop-down list of expected elements any time you type "<".

When you type *SPACE* from inside a start tag, you also get a drop-down list showing all attributes that can be added to the current element.

Likewise, when you type "=" for an attribute value, or the opening quote for the value, you also get a list of possible values for that attribute.

Moreover, *ToolTips* appear on these IntelliSense lists giving you a description of each element as illustrated in Figure D-2:

```
uC-Probe-CustomSy...lFile-Template.csf
<?xml version="1.0"?>
<CustomSymbols xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="http://micrium.com/probe/csf.xsd"

  <!-- XML-based Custom Symbol File (CSF) Example. -->

  <GroupOfSymbols>
    <GroupName>App.c</GroupName>
    <!-- Name of the Group of Symbols (i.e. Name of the C file where they are declared). -->
    <Symbols>
      <Symbol>
        <!-- First symbol declared in App.c -->
        <Name>MyUInt32</Name>
        <DisplayName>My UInt32 Global Variable</DisplayName>
        <DataType Size="4">unsigned long</DataType>
        <MemoryAddress>0x10F4</MemoryAddress>
      </S  Memory address in either decimal or hexadecimal format (0x1234).
    </Sym
  </GroupOfSymbols>

</CustomSymbols>
```

Figure D-2 **Creating a CSF in Visual Studio: Tool Tips**

Custom Symbol Files need to have the extension `.csf` for µC/Probe to recognize them as such. Listing D-1 shows an example of a CSF file that declares one integer, one array and one data structure.

```
<?xml version="1.0"?>
<CustomSymbols xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="http://micrium.com/probe/csf.xsd" >          (1)
  <GroupOfSymbols>                                                                          (2)
    <GroupName>App.c</GroupName>                                                            (3)
    <Symbols>
      <Symbol>
        <Name>MyUInt32</Name>                                                               (4)
        <DisplayName>My UInt32 Global Variable</DisplayName>                                (5)
        <DataType Size="4">unsigned long</DataType>                                         (6)
        <MemoryAddress>0x10F4</MemoryAddress>                                               (7)
      </Symbol>
      <Symbol>
        <Name>MyStruct</Name>
        <DataType Size="50">struct</DataType>
        <MemoryAddress>0x50AC</MemoryAddress>
        <DataMembers>                                                                       (8)
          <Symbol>
            <Name>MyArray</Name>
            <DataType Size="16" IsArray="true" ArrayLength="4">int</DataType>               (9)
            <MemoryAddress>0x50AC</MemoryAddress>
          </Symbol>
          <Symbol>
            <Name>MyCharPointer</Name>
            <DataType Size="4" IsPointer="true">char</DataType>                            (10)
            <MemoryAddress>1025</MemoryAddress>
          </Symbol>
        </DataMembers>
      </Symbol>
    </Symbols>
  </GroupOfSymbols>
</CustomSymbols>
```

Listing D-1 **XML-based Custom Symbol File Example**

LD-1(1) The root element is called `<CustomSymbols>`. It includes the XSD schema reference to help you editing in an XML editor such as Visual Studio.

LD-1(2) Each symbol or group of symbols are within the element tag `<GroupOfSymbols>`. In this case it is used with the purpose of creating a group of symbols declared in one single C file.

LD-1(3) The name of the group of symbols is `App.c`.

LD-1(4) The name of the first symbol in the group as declared in C.

LD-1(5) You can also specify an alias for the group name for display purposes.

LD-1(6) The element tag `<DataType>` is the ANSI C data type of the variable including the size in bytes as an attribute.

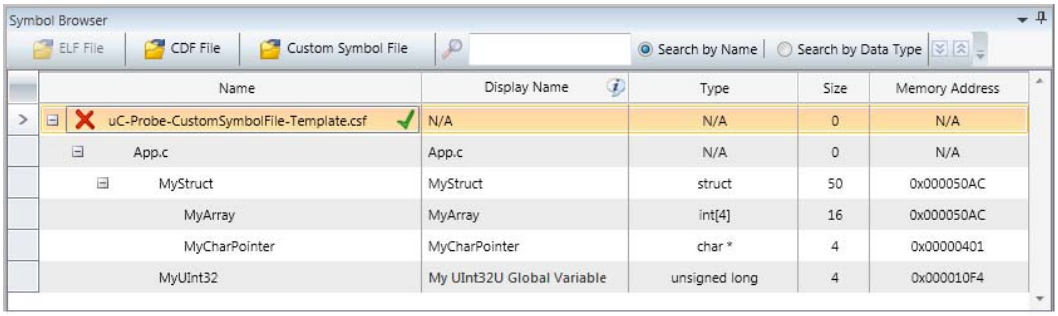
LD-1(7) The element tag `<MemoryAddress>` is the variable's memory address in either decimal or hexadecimal format (0x1234).

LD-1(8) For more complex symbols such as data structures, there is a tag called `<DataMembers>` that allows you to specify a group of symbols that make part of a data structure.

LD-1(9) In order to declare an array, you need to specify three attributes: A boolean flag that indicates that the symbol is an array, the total number of bytes and the number of elements in the array.

LD-1(10) Finally, any data type can be declared as a pointer by using the data type boolean attribute `IsPointer.` In this case, it is the intention to specify a symbol declared as `char *.`

In order to verify your CSF file, you can use the Symbol Browser from within µC/Probe as shown in Figure D-3. Notice the relationship between the XML tags and the tree nodes in the Symbol Browser.:



Name	Display Name	Type	Size	Memory Address
uC-Probe-CustomSymbolFile-Template.csf	N/A	N/A	0	N/A
App.c	App.c	N/A	0	N/A
MyStruct	MyStruct	struct	50	0x000050AC
MyArray	MyArray	int[4]	16	0x000050AC
MyCharPointer	MyCharPointer	char *	4	0x00000401
MyUInt32	My UInt32U Global Variable	unsigned long	4	0x000010F4

Figure D-3 **XML-based Custom Symbol File Example as seen from µC/Probe's Symbol Browser**

Appendix

E

Terminal Window Control

µC/Probe provides an option to enable debug traces to output the embedded target's activity via any of the communication interfaces supported by µC/Probe. A trace message is displayed in a terminal window control in µC/Probe, by calling a function `ProbeTermTrcPrint()` from your embedded application as illustrated in Figure E-1. Additionally, you can prefix the messages with special tags that µC/Probe will replace with icons that you get to choose.

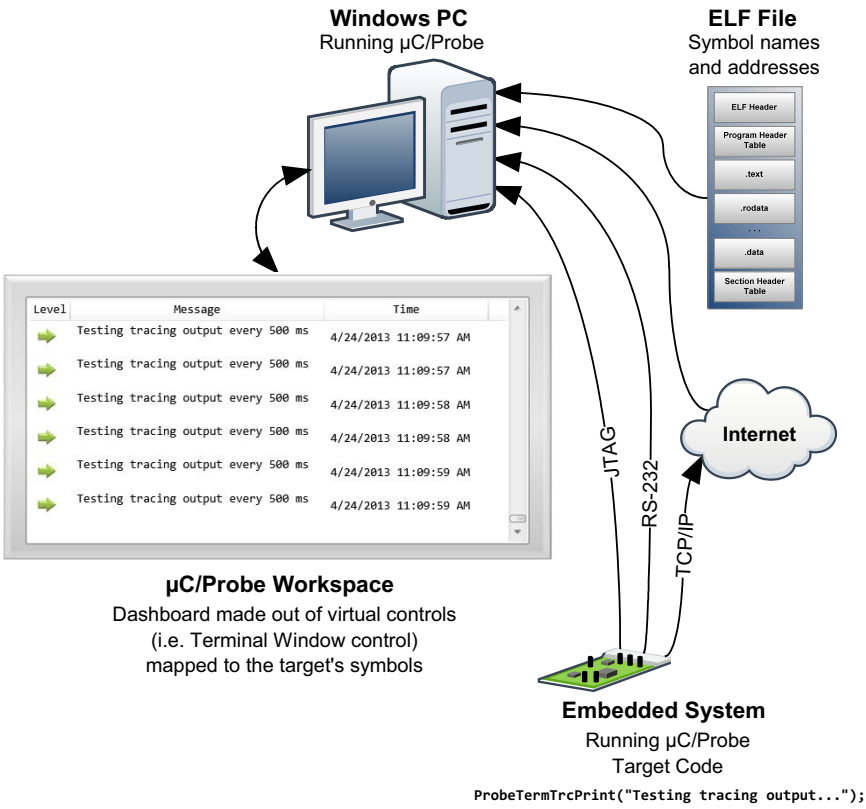


Figure E-1 **Terminal Window Control - Trace Interface**

At the same time, µC/Probe provides the option to enable a command-line interface to the embedded target. A command-line interface allows the user to issue a command to the target from a terminal window control in µC/Probe. Examples of command lines are `ipconfig`, `dir` or whatever command the programmer wants to implement in the embedded target.

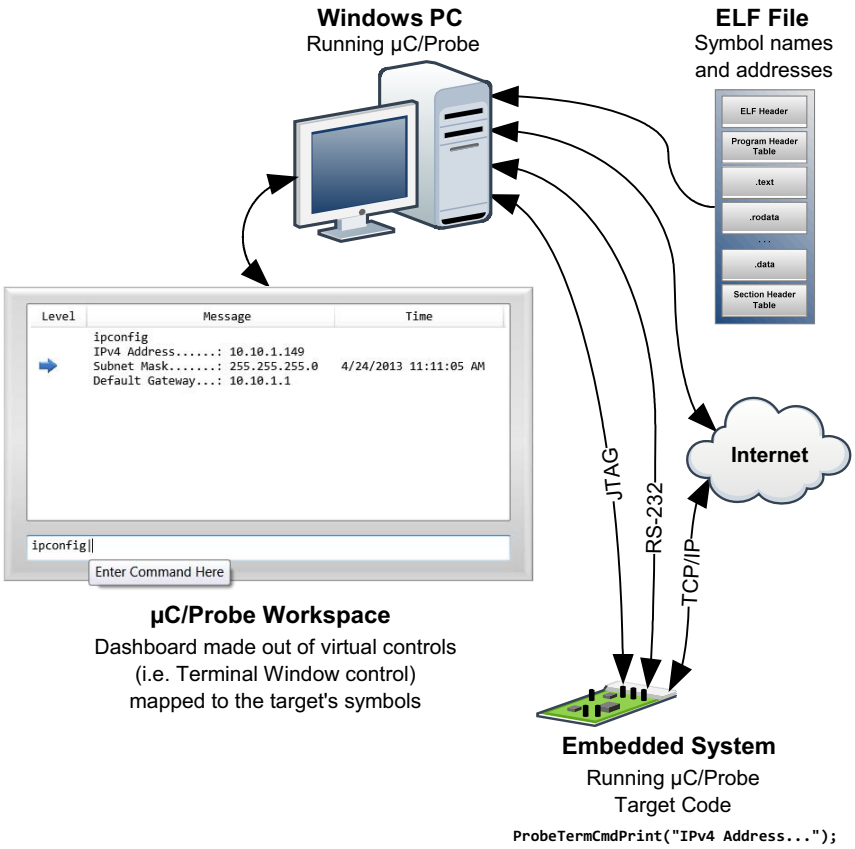


Figure E-2 **Terminal Window Control - Command Line Interface**

This appendix will introduce you to the debug trace and command-line tools available in µC/Probe. It will show you how to include them in your embedded target code and make use of them by covering the following topics:

- Prerequisites
- Downloading the necessary code for your embedded target

- Including the code in your embedded target project
- Configuring the code in your embedded target project
- Initializing the command-line and tracing interfaces
- Using the tracing function
- Using the command-line interface

E-1 PREREQUISITES

The only requirement is to have a running copy of µC/Probe communicating with your embedded target. Whether it is interfaced via J-Link, RS-232, TCP/IP or any of our certified third-party plugins such as the one distributed with IAR Systems Embedded Workbench, you can add trace and command-line functionality to your embedded product very easily.

E-2 DOWNLOADING THE NECESSARY CODE FOR YOUR EMBEDDED TARGET

The target code that supports the µC/Probe terminal window control is available for free from our download center at:

`http://micrium.com/downloadcenter/download-results?searchterm=mp-uc-probe&supported=true`

Look for the download link labeled µC/Probe 3.0 Target Code for the Terminal Window Control.

The download includes the files illustrated in Figure E-3

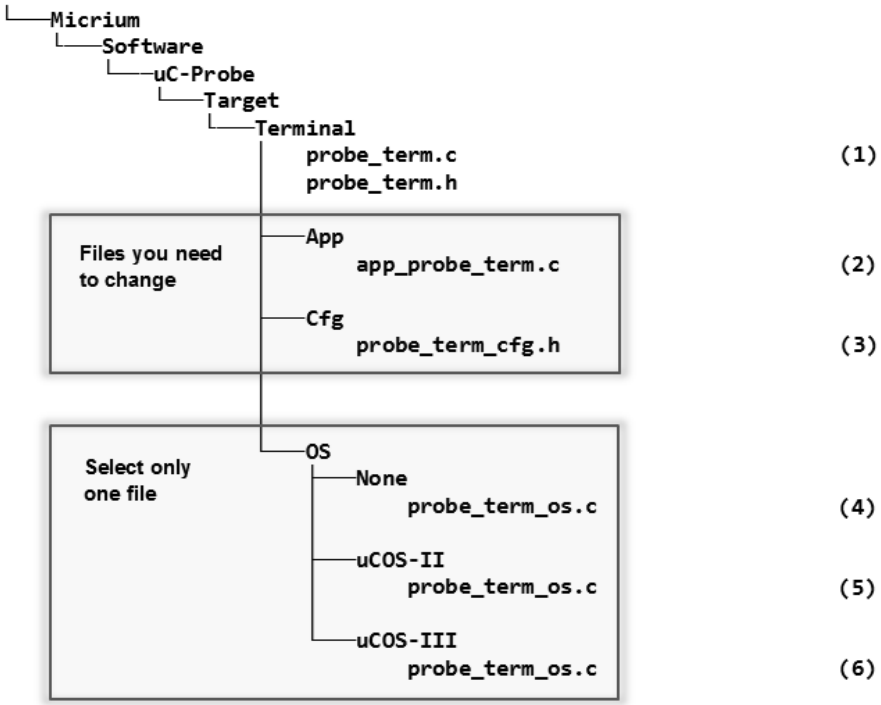


Figure E-3 **Terminal Window Target Code Files**

FE-3(1) The C source files `probe_term.c` and `probe_term.h` implement the core of the terminal window interface, including the protocol handshaking, state machines and the mechanism for message queueing. It is generic code and it does not depend on the kernel you are running; therefore, no changes to this code are necessary.

FE-3(2) The C source file `app_probe_term.c` sits at the application level and allows you to register a callback function that is invoked every time the user enters a command in the terminal window from µC/Probe. This callback function allows you to parse the command and respond appropriately. See section E-8 “Using the Command Line Interface” on page 79 for more details.

FE-3(3) The C header file `probe_term_cfg.h` allows you to configure the terminal window interface to satisfy your application's footprint and performance requirements. See section E-4 “Configuring the Code in your Embedded Target Project” on page 74 for more details.

FE-3(4) The C source file `probe_term_os.c` implements the OS layer for the case your embedded application does not have a kernel or it has a kernel different than the ones made by Micriµm.

FE-3(5) The C source file `probe_term_os.c` implements the OS layer that supports embedded applications based on Micriµm's µC/OS-II real-time kernel. No changes are required.

FE-3(6) The C source file `probe_term_os.c` implements the OS layer that supports embedded applications based on Micriµm's µC/OS-III real-time kernel. No changes are required.

E-3 INCLUDING THE CODE IN YOUR EMBEDDED TARGET PROJECT

The following sections describe which files to include in your embedded project depending on whether you are running µC/OS-II, µC/OS-III or no kernel at all.

Follow the instructions from one of the next three options.

E-3-1 INCLUDING THE CODE IN YOUR µC/OS-II EMBEDDED TARGET PROJECT

Configure your C project to compile all the C files shown in Figure E-3 except for the files indicated by the annotations FE-3(4) and FE-3(6).

Then, you need to add to your application code the following directive:

```
#include <probe_term.h>
```

Finally, you need to configure your C project's compiler settings to include the two new directory paths where the terminal window target code is located:

```
$\Micrium\Software\uC-Probe\Target\Terminal
```

```
$\Micrium\Software\uC-Probe\Target\Terminal\OS\uCOS-II
```

E-3-2 INCLUDING THE CODE IN YOUR µC/OS-III EMBEDDED TARGET PROJECT

Configure your C project to compile all the C files shown in Figure E-3 except for the files indicated by the annotations FE-3(4) and FE-3(5).

Then, you need to add to your application code the following directive:

```
#include <probe_term.h>
```

Finally, you need to configure your C project's compiler settings to include the two new directory paths where the terminal window target code is located:

```
$\Micrium\Software\uC-Probe\Target\Terminal

$\Micrium\Software\uC-Probe\Target\Terminal\OS\uCOS-III
```

E-3-3 INCLUDING THE CODE IN ANY EMBEDDED TARGET PROJECT

Configure your C project to compile all the files shown in Figure E-3 except for the files indicated by the annotations FE-3(5) and FE-3(6).

Then, you need to add to your application code the following directive:

```
#include <probe_term.h>
```

Finally, you need to configure your C project's compiler settings to include the two new directory paths where the terminal window target code is located:

```
$\Micrium\Software\uC-Probe\Target\Terminal

$\Micrium\Software\uC-Probe\Target\Terminal\OS\None
```

E-4 CONFIGURING THE CODE IN YOUR EMBEDDED TARGET PROJECT

The C header file `probe_term_cfg.h` allows you to configure the terminal window interface to satisfy your application's footprint and performance requirements through a series of pre-processor macros as shown in the code listing below:

```
/*
*********************************************************************************************
*                          UC/PROBE TERMINAL WINDOW GENERIC CONFIGURATION
*********************************************************************************************
*/
                                                        /* ----- COMMAND LINE INTERFACE ---- */
#define PROBE_TERM_CFG_CMD_EN                    1  /* Interface enable.                 */  (1)
#define PROBE_TERM_OS_CFG_CMD_RX_TASK_STK_SIZE 128  /* OS task stack size.               */  (2)
#define PROBE_TERM_OS_CFG_CMD_RX_TASK_PRIO      10  /* OS task priority.                 */  (3)
#define PROBE_TERM_OS_CFG_CMD_TX_TASK_STK_SIZE 128  /* OS task stack size.               */  (4)
#define PROBE_TERM_OS_CFG_CMD_TX_TASK_PRIO      11  /* OS task priority.                 */  (5)
                                                        /* ---------- TRACE INTERFACE -------*/
#define PROBE_TERM_CFG_TRC_EN                    1  /* Interface enable.                 */  (6)
#define PROBE_TERM_OS_CFG_TRC_TASK_STK_SIZE    128  /* OS task stack size.               */  (7)
#define PROBE_TERM_OS_CFG_TRC_TASK_PRIO         12  /* OS task priority.                 */  (8)

#define PROBE_TERM_CFG_BUF_SIZE                 64  /* Max size of the Rx/Tx msg arrays. */  (9)
#define PROBE_TERM_CFG_Q_SIZE                    8  /* Max nbr of msg arrays in the q.   */ (10)

#define PROBE_TERM_OS_CFG_TASK_DLY_MSEC        100  /* OS task dly in msecs to yield CPU.*/ (11)
```

Listing E-1 **Terminal Window Configuration**

LE-1(1) The C pre-processor macro `PROBE_TERM_CFG_CMD_EN` allows you to enable or disable all the C code related to the command line interface. The command line interface core includes two state machines; One for the reception of commands and the other for the transmission of responses. If your embedded application runs on top of a kernel, then each state machine is executed by a task.

LE-1(2) The C pre-processor macro `PROBE_TERM_OS_CFG_CMD_RX_TASK_STK_SIZE` allows you to specify the size of the command-line reception task's stack assuming your embedded application is running on top of a kernel.

LE-1(3) The C pre-processor macro `PROBE_TERM_OS_CFG_CMD_RX_TASK_PRIO` allows you to specify the priority of the command-line reception task assuming your embedded application is running on top of a kernel.

LE-1(4) The C pre-processor macro `PROBE_TERM_OS_CFG_CMD_TX_TASK_STK_SIZE` allows you to specify the size of the command-line transmission task's stack assuming your embedded application is running on top of a kernel.

LE-1(5) The C pre-processor macro `PROBE_TERM_OS_CFG_CMD_TX_TASK_PRIO` allows you to specify the priority of the command-line transmission task assuming your embedded application is running on top of a kernel.

LE-1(6) The C pre-processor macro `PROBE_TERM_CFG_TRC_EN` allows you to enable or disable all the C code related to the tracing interface. The tracing interface core includes one single state machine. If your embedded application runs on top of a kernel, then this state machine is executed by a task.

LE-1(7) The C pre-processor macro `PROBE_TERM_OS_CFG_TRC_TASK_STK_SIZE` allows you to specify the size of the trace task's stack assuming your embedded application is running on top of a kernel.

LE-1(8) The C pre-processor macro `PROBE_TERM_OS_CFG_TRC_TASK_PRIO` allows you to specify the priority of the trace task assuming your embedded application is running on top of a kernel.

LE-1(9) The C pre-processor macro `PROBE_TERM_CFG_BUF_SIZE` allows you to specify the size of the data buffer. In the case of the command-line interface, configure it to hold the longest command including its arguments that you expect to receive from the user.

LE-1(10) The C pre-processor macro `PROBE_TERM_CFG_Q_SIZE` allows you to specify the amount of messages of length `PROBE_TERM_CFG_BUF_SIZE` that you can hold in queue.

LE-1(11) The C pre-processor macro `PROBE_TERM_OS_CFG_TASK_DLY_MSEC` allows you to specify the amount of time in milliseconds that the command-line and tracing interfaces are willing to yield the CPU in case your embedded application is running on top of a kernel.

E-5 INITIALIZING THE COMMAND LINE AND TRACING INTERFACES

In order to initialize the command-line and tracing interfaces, you need to add to your application code the 3 lines shown in the code listing below:

```
int  main (void)
{
    PROBE_TERM_ERR probe_term_err;                                                    (1)

    ...

    AppProbeTermSetAllHooks();             /* Sets all the application hooks.          */    (2)
    ProbeTermInit(&probe_term_err);        /* Initialize the uC/Probe terminal window.  */    (3)

    ...

    return (0);
}
```

Listing E-2 **Initialization**

LE-2(1) Declare a local variable to receive the error code from the terminal window module.

LE-2(2) The function `AppProbeTermSetAllHooks()` is declared in `app_probe_term.c`. You need to call this function to register the callback function that ties the terminal window module with your application code for command-line purposes. If you are only going to use the tracing functionality, then you do not need to call this function.

LE-2(3) Finally, you need to call the function `ProbeTermInit()` in order to initialize the state machines and OS layer.

E-6 USING THE TRACING FUNCTION

In order to output a trace message from your embedded application to the terminal window control in µC/Probe, you need to call the non-blocking function `ProbeTermTrcPrint()` as shown in the following code listings.

The first example in code Listing E-3 shows you that you can call the `ProbeTermTrcPrint()` function multiple times one after the other for up to a maximum of the value configured in `PROBE_TERM_CFG_Q_SIZE`, which represents the maximum number of entries in the queue.

Because of the nature of the µC/Probe communication interfaces, every time you call `ProbeTermTrcPrint()` the system is queueing up the message; Therefore, consider configuring `PROBE_TERM_CFG_Q_SIZE` according to your application needs. The example below shows that if `PROBE_TERM_CFG_Q_SIZE` is set to 8, then you can call `ProbeTermTrcPrint()` for up to 8 consecutive times. Keep in mind that depending on the communication interface of your choice, each call to `ProbeTermTrcPrint()` takes a total time of up to 100ms to display the message.

```
ProbeTermTrcPrint("Testing a trace line #1\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #2\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #3\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #4\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #5\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #6\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #7\n");   /* Prints a trace msg on the term window. */
ProbeTermTrcPrint("Testing a trace line #8\n");   /* Prints a trace msg on the term window. */
```

Listing E-3 **Tracing by Multiple Consecutive Calls**

In case your embedded application does not have enough memory to fit queues, you can also choose to trace messages by concatenating them into one single call as shown in code Listing E-4:

```
ProbeTermTrcPrint("Testing a trace line #1\nTesting a trace line #2\n"
                  "Testing a trace line #3\nTesting a trace line #4\n"
                  "Testing a trace line #5\nTesting a trace line #6\n"
                  "Testing a trace line #7\nTesting a trace line #8\n");
```

Listing E-4 **Tracing by a Single Call**

Also, notice that the maximum length of your tracing message depends on both, the maximum data buffer size configured in `PROBE_TERM_CFG_BUF_SIZE` and the queue size configured in `PROBE_TERM_CFG_Q_SIZE`, because the system splits the tracing message in chunks of `PROBE_TERM_CFG_BUF_SIZE` characters in length and puts them in the queue.

E-7 USING THE TRACING MESSAGE ICON TAGS

µC/Probe allows you to display icons on the terminal window for each of the tracing messages. All you have to do is include in your message a tag for the icon you want to display next to the message. For example, if your tracing message is meant to be an error, you can prefix your message with the tag `[Error]` and µC/Probe will include an error icon next to the message.

```
ProbeTermTrcPrint("[Error] Testing error.\n");      /* Prints an error   msg on the term win. */
ProbeTermTrcPrint("[Warning] Testing warning.\n");  /* Prints a  warning msg on the term win. */
ProbeTermTrcPrint("[Info] Testing information.\n"); /* Prints an info    msg on the term win. */
```

Listing E-5 **Tracing Message Icon Tags**

E-8 USING THE COMMAND LINE INTERFACE

In order to be able to process user's commands from your embedded application, you need to implement the callback function `AppProbeTermHookRx()` declared in the C file `app_probe_term.c` as shown in the example code below:

```
void  AppProbeTermHookRx (CPU_CHAR  *p_str)
{
    CPU_CHAR  buf[PROBE_TERM_CFG_BUF_SIZE];


    if (Str_CmpIgnoreCase_N(p_str, "dir", 3) == 0) {             /* Process "dir".          */
        ProbeTermCmdPrint("test.txt\ntest.jpg\ntest.wav\n");
    } else if (Str_CmpIgnoreCase_N(p_str, "ipconfig", 8) == 0) { /* Process "ipconfig".     */
        ProbeTermCmdPrint("IPv4 Address......: 10.10.1.149\n");
        ProbeTermCmdPrint("Subnet Mask.......: 255.255.255.0\n");
        ProbeTermCmdPrint("Default Gateway...: 10.10.1.1\n");
    } else if (Str_CmpIgnoreCase_N(p_str, "echo", 4) == 0) {     /* Process "echo" for test */
        Str_Copy_N(&buf[0], &p_str[5], PROBE_TERM_CFG_BUF_SIZE - 2);
        Str_Cat_N(&buf[Str_Len(buf)], "\n\0", 2);
        ProbeTermCmdPrint(&buf[0]);
    } else {
        ProbeTermCmdPrint("[ERROR] Invalid/Unknown Command\n");
    }
}
```

Listing E-6 **Application Hook**

The function `AppProbeTermHookRx()` gets called every time the user sends a command line from µC/Probe. The pointer to the command line string is `p_str`. All you have to do is parse the command line and respond to the command by calling the function `ProbeTermCmdPrint()`.

Alternatively, if the number of commands to process is more than just a couple, you can always tie this callback function to a Micriµm module that is specifically designed to provide a shell interface to embedded systems, called µC/Shell.

µC/Shell is distributed for free and is a stand-alone module that allows you to parse and execute a string that contains a command and its arguments. For more information, please contact our sales department to get a free copy of µC/Shell.

Appendix

F

µC/Trace Triggers Control

µC/Trace is a runtime diagnostics tool for embedded software systems based on µC/OS-III. µC/Trace gives developers 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 F-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.

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

The µC/Trace solution consists of three parts:

- The PC application (µC/Trace), used to analyze the recordings as shown in Figure F-1.
- A trace recorder library that integrates with µC/OS-III, provided in C source code.
- Optionally, µC/Probe can be used for the target system connection.

The PC application µC/Trace has been developed for Microsoft Windows.

The trace recorder library 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 F-2:

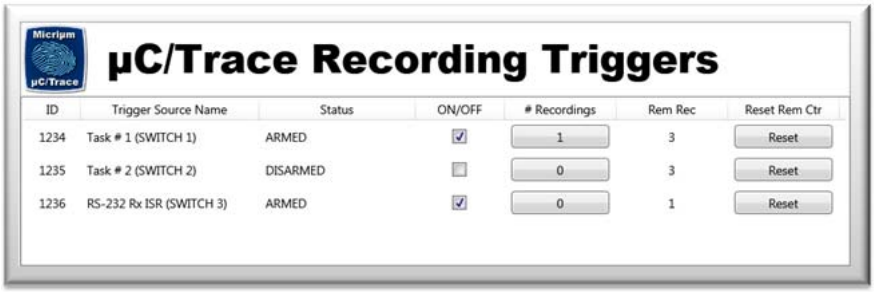


Figure F-2 **µC/Trace Trigger Control**

This appendix will describe how to use the µC/Trace Trigger control by discussing the following topics:

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

F-1 INCLUDING THE µC/TRACE SUPPORTING CODE IN YOUR TARGET

The target code to support the µC/Trace Triggering functionality is available at the Micriµm website at:

http://www.micrium.com

The files are illustrated in the figure below:

```
Micrium
└──Software
    └──uC-Trace
        ├──Recorders
        │   └──Percepio
        │       ├──OS
        │       │   └──uCOS-III
        │       │           trace_os.h                (1)
        │       │
        │       └──TraceRecorderLibrary
        │           │   trcBase.c
        │           │   trcHardwarePort.c
        │           │   trcKernel.c                   (2)
        │           │   trcUser.c
        │           │
        │           ├──Cfg
        │           │       trcConfig.h               (3)
        │           │       trcHardwarePort.h
        │           │
        │           ├──Include
        │           │       trcBase.h
        │           │       trcKernel.h
        │           │       trcKernelHooks.h          (4)
        │           │       trcTypes.h
        │           │       trcUser.h
        │           │
        │           └──KernelPorts                    (5)
        │               └──uCOS-III
        │                       trcKernelPort.c
        │                       trcKernelPort.h
        │
        └──Source
            trace_trig.c                              (6)
            trace_trig.h
```

Figure F-3 **µC/Trace Target Code Files**

FF-3(1) 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.

FF-3(2) The core source files for the trace recorder library by Percepio.

FF-3(3) A couple of files allow you to configure the size of the RAM buffer and the hardware clock among other settings.

FF-3(4) The core header files for the trace recorder library by Percepio.

FF-3(5) The kernel port that implements how to record the kernel events.

FF-3(6) The µC/Trace Triggering mechanism that allows you to use µC/Trace in conjunction with µC/Probe to arm/disarm recording triggers and upload recordings to the host PC.

F-2 µC/TRACE TRIGGERS FUNCTIONAL DESCRIPTION

Figure F-4 shows the block diagram of the entire system including the trace recording facility in the target code and µC/Probe and µC/Trace in the host system. The entire operation can be described in 10 steps.

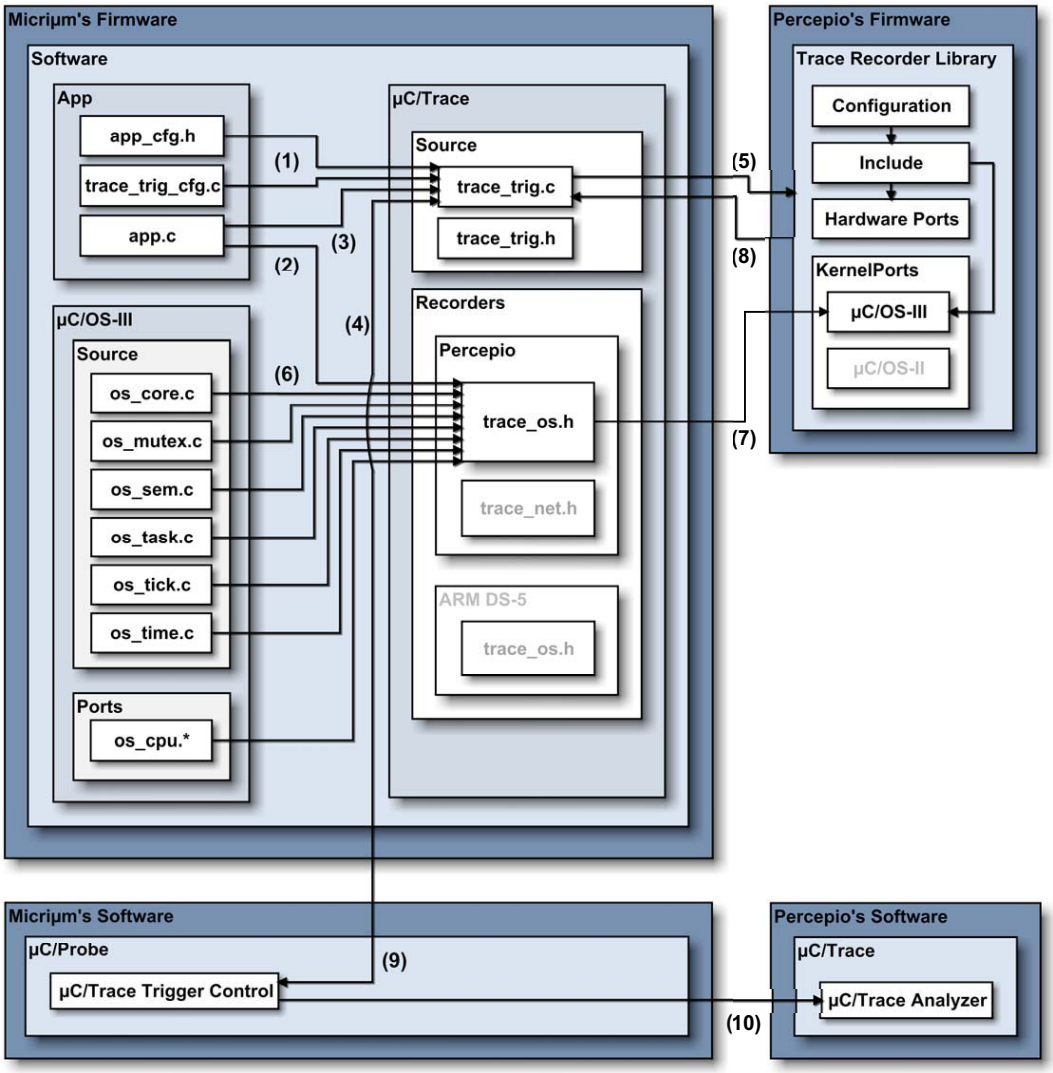


Figure F-4 **µC/Trace Block Diagram**

FF-4(1) The first step to get started with µC/Trace is enabling the module in `app_cfg.h` through the definition of the macro `TRACE_CFG_EN` set to 1. The second step in this configuration stage is to define the trigger points in the table declared in `trace_trig_cfg.c` as shown in Listing F-1:

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

#define  TRACE_TRIG_ID_SW1                  1234u
#define  TRACE_TRIG_ID_SW2                  1235u
#define  TRACE_TRIG_ID_ISR_RS232_RX         1236u

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

const  TRACE_TRIG_CFG  TraceTrigCfgTbl[] =
{
    {TRACE_TRIG_ID_SW1,          "Task # 1 (SWITCH 1)",      3},
    {TRACE_TRIG_ID_SW2,          "Task # 2 (SWITCH 2)",      3},
    {TRACE_TRIG_ID_ISR_RS232_RX, "RS-232 Rx ISR (SWITCH 3)", 1}
};
```

Listing F-1 **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.

FF-4(2) You initialize the µC/Trace module by calling the macro `TRACE_INIT()`.

FF-4(3) You initialize the µC/Trace Triggers module by calling the function `TraceTrigInit()`.

FF-4(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 shown Figure F-2, 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.

FF-4(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.

FF-4(6) All the kernel events will be recorded into RAM.

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

FF-4(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.

FF-4(9) In turn, the µC/Trace Triggers module notifies µC/Probe that the recording is finished.

FF-4(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.

Appendix

G

Oscilloscope Control

µC/Probe allows you to analyze data in real-time by showing the value of multiple memory addresses in a screen akin to an oscilloscope. Similar to other controls in µC/Probe, you simply select the variables you want to plot from the **Symbol Browser**. The oscilloscope control can display up to 8 channels in either a single vertical scale or multiple scales.

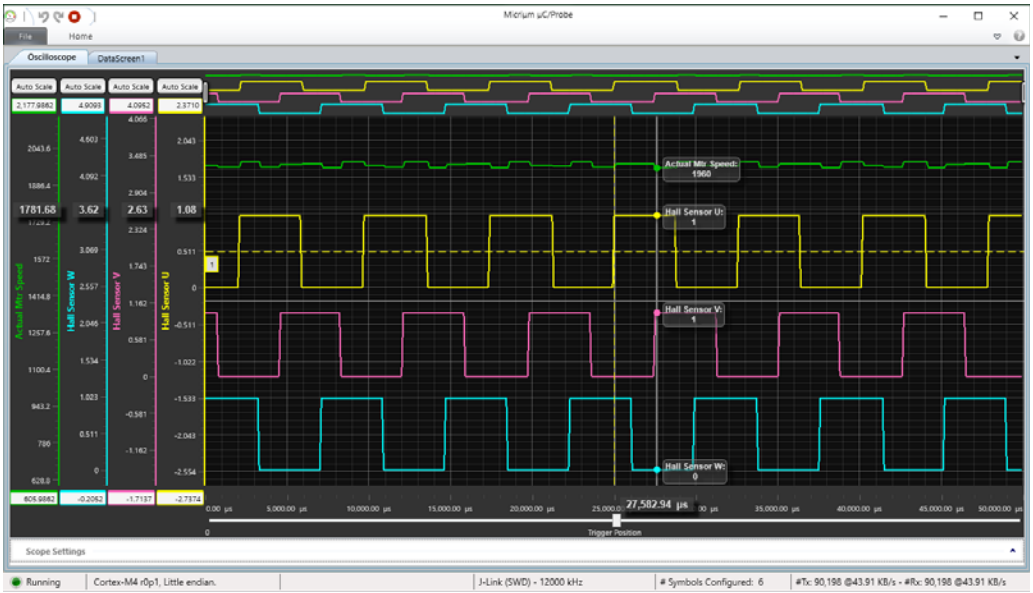


Figure G-1 **µC/Probe Oscilloscope Control**

This appendix gives you step-by-step instructions on how to include and configure the embedded target resident code to support the oscilloscope control in µC/Probe. The final section provides a summary in the form of block diagrams.

For more information on how to use the oscilloscope control from within µC/Probe (i.e. the Windows PC), refer to the document **µC/Probe User's Manual**.

G-1 DOWNLOADING THE NECESSARY CODE FOR YOUR EMBEDDED TARGET

The target code that supports the µC/Probe Oscilloscope Control is available for free from our website at:

`https://www.micrium.com/tools/ucprobe/software-and-docs/`

Look for the download link labeled µC/Probe Embedded Target Code.

The download includes the files illustrated in Figure G-2

```
└──Micrium
    └──Software
        └──uC-Probe
            └──Target
                └──Scope
                    │   probe_scope.c          (1)
                    │   probe_scope.h
                    │
                    └──cfg
                        probe_scope_cfg.h      (2)
```

Figure G-2 **Oscilloscope Target Code Files**

FG-2(1) The C source files `probe_scope.c` and `probe_scope.h` implement the core of the oscilloscope control, including the state machine and the mechanism for triggering. It is generic code and it does not depend on the kernel you are running; therefore, no changes to this code are necessary.

FG-2(2) The C header file `probe_scope_cfg.h` allows you to configure the oscilloscope control to satisfy your application's footprint and performance requirements. See section G-3 “Configuring the Code in your Embedded Target Project” on page 90 for more details.

G-2 INCLUDING THE CODE IN YOUR EMBEDDED TARGET PROJECT

Configure your C project to compile all the C files shown in Figure G-2.

Then, you need to add to your application code the following directive:

```
#include <probe_scope.h>
```

Finally, you need to configure your C project's compiler settings to include the two new directory paths where the oscilloscope control target code is located:

```
$\Micrium\Software\uC-Probe\Target\Scope
```

```
$\Micrium\Software\uC-Probe\Target\Scope\Cfg
```

G-3 CONFIGURING THE CODE IN YOUR EMBEDDED TARGET PROJECT

The C header file `probe_scope_cfg.h` allows you to configure the oscilloscope control to satisfy your application's footprint and performance requirements through a series of pre-processor macros as shown in the code listing below:

```
/*
*********************************************************************************************
*                                UC/PROBE SCOPE CONFIGURATION
*********************************************************************************************
*/

#define PROBE_SCOPE_MAX_CH                   8  /* Max number of channels: [1,8].         */ (1)
#define PROBE_SCOPE_MAX_SAMPLES           1000  /* Max number of samples per channel.     */ (2)
#define PROBE_SCOPE_16_BIT_EN                1  /* Max size of sample is 16-bits: [0,1].  */ (3)
#define PROBE_SCOPE_32_BIT_EN                1  /* Max size of sample is 32-bits: [0,1].  */ (4)
```

Listing G-1 **Oscilloscope Configuration**

LG-1(1) The C pre-processor macro `PROBE_SCOPE_MAX_CH` allows you to specify the maximum number of channels.

LG-1(2) The C pre-processor macro `PROBE_SCOPE_MAX_SAMPLES` allows you to specify the maximum number of samples to acquire per channel.

LG-1(3) The C pre-processor macro `PROBE_SCOPE_16_BIT_EN` allows you to enable or disable support for 16-bit sample channels.

LG-1(4) The C pre-processor macro `PROBE_SCOPE_32_BIT_EN` allows you to enable or disable support for 32-bit sample channels.

G-4 INITIALIZING THE OSCILLOSCOPE CONTROL

In order to initialize the oscilloscope control, you need to call the function `ProbeScope_Init()` as shown in the code listing below:

```
int  main (void)
{
    ProbeScope_Init(10000);                /* Initialize the uC/Probe scope ocntrol..  */    (1)

    ...

}
```

Listing G-2 **Initialization**

LG-2(1) Call the function ProbeScope_Init() with the sampling frequency as an argument. That is the frequency in hertz at which you intend to call this function.

G-5 DATA ACQUISITION

Whenever you want to acquire samples, you call the function `ProbeScope_Sampling()`.

The most typical use is to configure a hardware timer to guarantee that the samples are equally spaced in time. That is your choice. In any case, it is assumed that the samples are equally spaced in time.

The following diagrams summarize the previous discussion and the two scenarios:

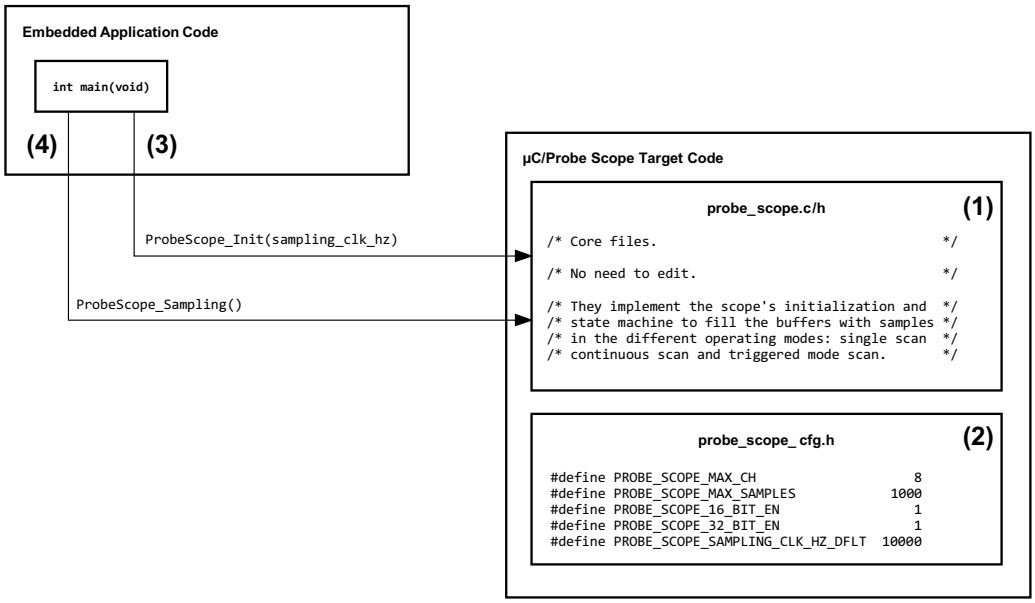


Figure G-3 **µC/Probe Oscilloscope Control: Data Acquisition: No hardware timer**

FG-3(1) Include the `probe_scope.c` and `probe_scope.h` files in your project.

FG-3(2) Configure the scope by declaring these five macros in a file called `probe_scope_cfg.h`.

FG-3(3) Call `ProbeScope_Init(sampling_clk_hz)` from your application to initialize the scope.

FG-3(4) Call `ProbeScope_Sampling()` to acquire the samples periodically.

If you configure your own hardware timer to acquire the samples then the diagram changes slightly as shown below:

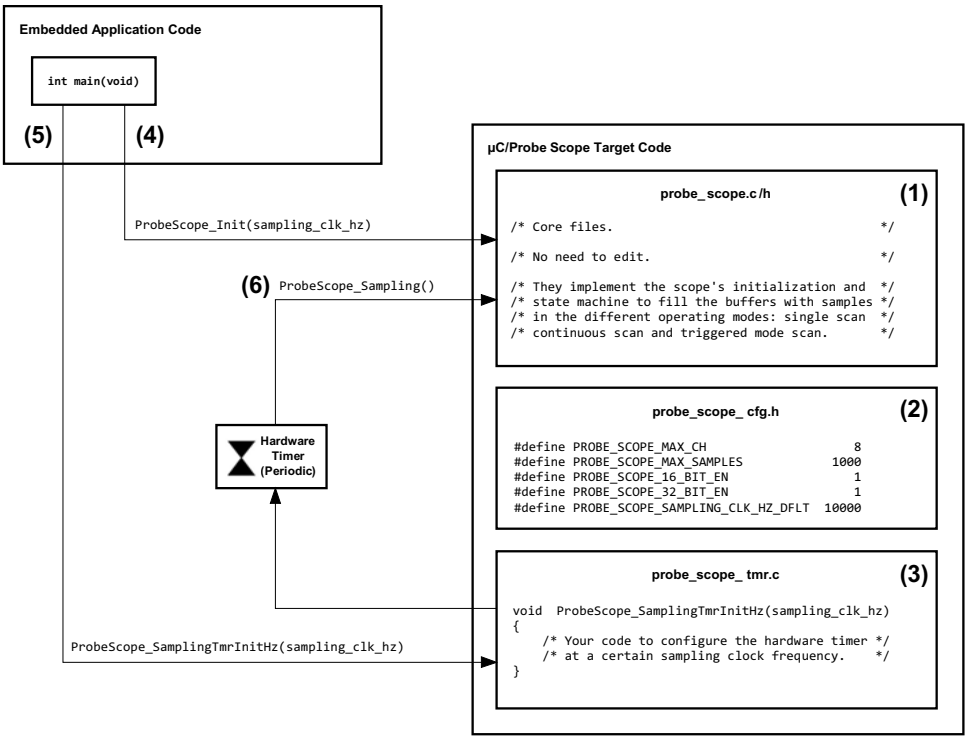


Figure G-4 **µC/Probe Oscilloscope Control: Data Acquisition**

FG-4(1) Include the `probe_scope.c` and `probe_scope.h` files in your project.

FG-4(2) Configure the scope by declaring these five macros in a file called `probe_scope_cfg.h`.

FG-4(3) Declare a function that configures your hardware timer at the specified sampling frequency.

FG-4(4) Call `ProbeScope_Init(sampling_clk_hz)` from your application to initialize the scope.

FG-4(5) Call `ProbeScope_SamplingTmrInitHz(sampling_clk_hz)` from your application to initialize the hardware timer.

FG-4(6) Configure the hardware timer's ISR to call `ProbeScope_Sampling()` to acquire the samples periodically.

Appendix H

Bibliography

- Labrosse Jean. *µC/OS-II The Real-Time Kernel*. R&D Technical Books, ISBN 1-57820-103-9, 2002.

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

- Légaré Christian. *µC/TCP-IP The Embedded Protocol Stack*. Micriµm Press, 2011.

Index

R

S

T

U

Z