Micriµm

µC/TFTPs

Trivial File Transfer Protocol (Server)

User's Manual

www.Micrium.com

Table of Contents

1.00 Introduction

µC/TFTPs is an add-on product to µC/TCP-IP that implements the Trivial File Transfer Protocol (TFTP). TFTP is a simple, lock-step, file transfer protocol which allows a client to get or put files onto an embedded product. The 's' in µC/TFTPs stands for 'server'. The µC/TFTPs module implements a part of RFC 1350 and RFC 2349 (ftp://ftp.rfc-editor.org/in-notes/rfc1350.txt and ftp://ftp.rfc-editor.org/in-notes/rfc2349.txt).

TFTP has been implemented on top of the Internet User Datagram Protocol (UDP). The only thing TFTP can do is read and write files from/to a remote server.

Figure 1-1 shows a block diagram that presents the relationship between components needed to make µC/TFTPs in a target application. You should note that µC/TFTPs assumes the presence of µC/TCP-IP but could actually be modified to work with just about any TCP/IP stack.

µC/TFTPs executes as a task under a Real-Time Operating System (RTOS). The RTOS can be just about any RTOS that supports multi-tasking. µC/TFTPs requires only one task.

Finally, µC/TFTPs assumes the presence of a File System which allows you to read and write files from and to some mass storage device (RAM disk, SD/MMC, IDE, on-board Flash, etc.). µC/TFTPs assumes the API of Micrium's µC/FS. If you use a different file system, you will need to 'simulate' µC/FS's API calls or, modify the µC/TFTPs code to use your own file system. We recommend that you create a file that simulates the µC/FS API. This way, if we make modifications to µC/TFTPs, you would not have to redo this work.

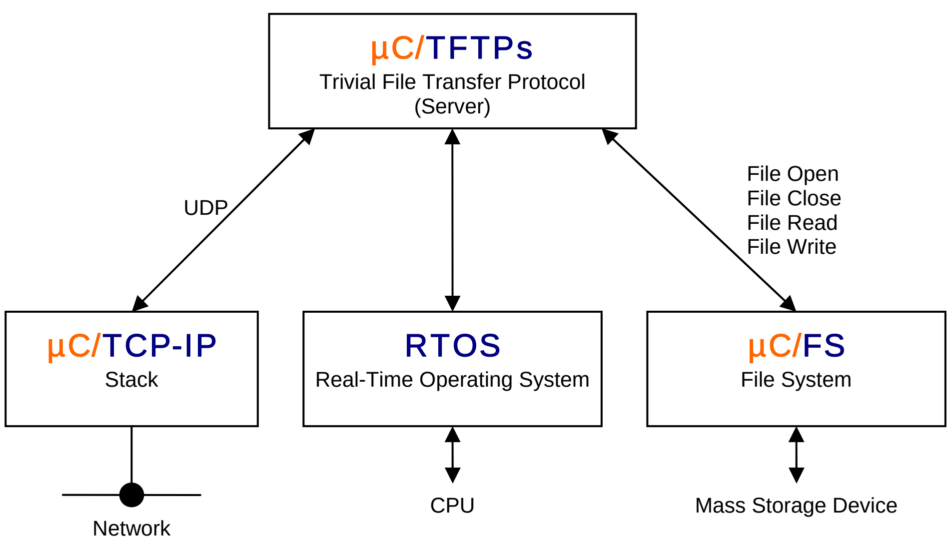

Figure 1-1, Relationship between µC/TFTPs, µC/TCP-IP, an RTOS and a µC/FS

2.00 Directories and Files

The files for µC/TFTPs are placed in multiple directories as described below.

`\Micrium\Software\uC-TFTPs`

This is the main directory for µC/TFTPs.

`\Micrium\Software\uC-TFTPs\`**`CFG\Template`**

This directory contains a template of µC/TFTPs configuration.

`\Micrium\Software\uC-TFTPs\`**`Source`**

This directory contains the source code for the RTOS independent code for µC/TFTPs. TFTP is fairly easy to implement and thus, the code is found in just two files:

```
tftp-s.c
tftp-s.h
```

Note that the 's' at the end of `tftp-s.c` means server and thus it contains 'server' side code. `tftp-s.h` is a header file that contains server declarations for TFTP.

`\Micrium\Software\uC-TFTPs\`**`OS`**

This directory contains RTOS specific implementation files. We recommend that you place your own RTOS implementation files (if needed) under the `OS` directory and that you name the implementation files `tftp-s_os.c`.

`\Micrium\Software\uC-TFTPs\`**`OS\uCOS-II`**

µC/TFTPs comes with the implementation file to interface to µC/OS-II.

3.00 Using µC/TFTPs

Figure 3-1 shows the test setup to demonstrate the use of µC/TFTPs. We used a CSB337 (ARM9) target connected to a Windows-based PC through a 10-Base-T Hub. The PC's IP address was set to `10.10.10.111` and the target address was hard coded to `10.10.10.64`. On the Windows PC, we opened a 'Command Prompt' window and typed either:

```
tftp –i 10.10.10.64 put test.txt
```

to write the file `test.txt` onto the target or,

```
tftp –i 10.10.10.64 get test.txt
```

to read the file from the target.

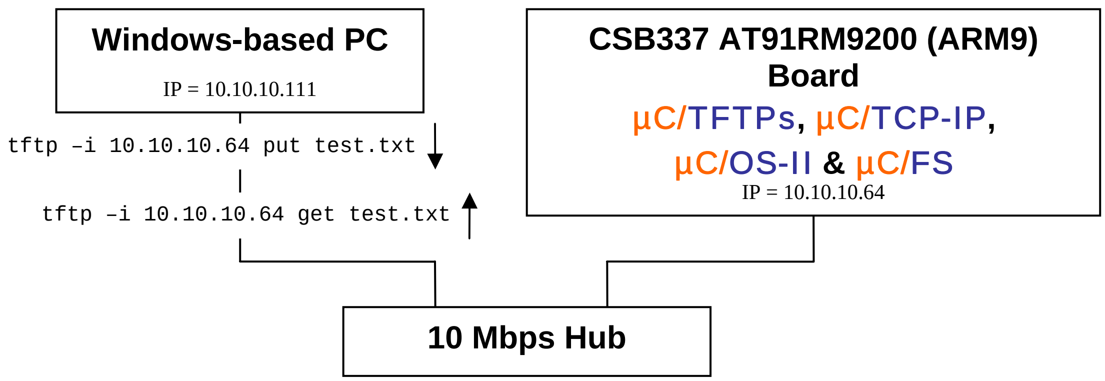

Figure 3-1, Test setup

3.01 Modules

As shown in figure 3-1, the target board runs the following Micrium software component:

µC/DHCPc
µC/FS
µC/OS-II
µC/TCP-IP
µC/TFTPs

Not shown in figure 3-1 are the following support modules:

µC/CPU
µC/LIB

µC/DHCPc is an implementation of the client portion of the DHCP (dynamic host configuration protocol). This protocol enables dynamic IP configuration to be distributed from one central source (the server) to booting clients.

µC/FS is an embedded file system that allows you to save and retrieve information using the FAT12, FAT16 or FAT32 format. µC/FS supports a number of mass storage media such as SD (Secure Digital), MMC (MultiMedia Card), SMC (Smart Media Card), CF (Compact Flash), IDE (Integrated Drive Electronics), RAM disk, Linear Flash and more. For the test application, we created a RAM-disk.

µC/OS-II is a real-time, multitasking kernel that allows you to have up to 250 application tasks. µC/TFTPs runs as one of those tasks. You can use just about any other RTOSs with µC/TFTPs but you would need to provide RTOS adaptation functions.

µC/TCP-IP is an embedded TCP/IP stack that provides IPv4 networking support.

µC/CPU is a module that removes the dependencies of the target CPU. In other words, µC/CPU defines common data types used by all Micrium software components and thus allows these components to be easily ported to different CPUs simply by changing the µC/CPU implementation files.

µC/LIB declares functions to replace the standard C `str???()` functions, `mem???()` functions and more. The reason we did this is to make it easier to get our software validated by third parties (e.g. FAA, FDA, etc.) because we provide all the source code to our products. You might argue that most compiler vendors also provide source code for these libraries. However, it's better (from a certification point of vue) to always certify the same source code instead of different source from multiple compiler vendors.

3.02 Creating the test application

This section describes the directories and files needed to build the sample application. You will notice that a fair amount of software components are needed to create the executable that runs on the target.

You may also notice also that package µC/DHCPc is needed to build the example application. This package is been used for convenience. µC/DHCPc obtains automatically a valid IP configuration from a DHCP server.

In the discussion of directories, we will assume that the directories listed are relative to the install directory:

```
\Micrium\Software\
```

In other words,

```
.\EvalBoards\Cogent\CSB337\IAR\BSP
```

Actually means:

```
\Micrium\Software\EvalBoards\Cogent\CSB337\IAR\BSP
```

3.02.01 Example Application Source Files

The test application was placed in the following directory:

```
.\EvalBoards\Cogent\CSB337\IAR\uC-Apps\Ex1
```

and consist of the following files:

```
app.c
Ex1.*
app_cfg.h
fs_conf.h
includes.h
net_cfg.h
os_cfg.h
```

The entire example application uses seven tasks. Two tasks for µC/OS-II (idle and statistics), two for µC/TCP-IP, one for µC/TFTPs, the startup task and the LED task.

app.c

This file contains the application code for example #1. As with most C programs, code execution start at `main()` which is shown in listing 3-1.

Listing 3-1

```
int  main (void)
{
    INT8U err;


    BSP_Init();            /* (1) Initialize the CSB335 BSP                          */

    APP_DEBUG_TRACE ("Initializing: uC/OS-II\n");

    OSInit();              /* (2) Initialize uC/OS-II                                */

                           /* (3) Create start task                                  */
    OSTaskCreateExt(AppTaskStart,
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[APP_START_TASK_STK_SIZE - 1],
                    APP_START_TASK_PRIO,
                    APP_START_TASK_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    APP_START_TASK_STK_SIZE,
                    (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

                           /* (4) Give a name to tasks                               */
#if (OS_TASK_NAME_SIZE > 16)
    OSTaskNameSet(APP_START_TASK_PRIO, "Start task", &err);
#endif

    APP_DEBUG_TRACE ("Starting    : uC/OS-II\n");
    OSStart();             /* (5) Start uC/OS-II                                     */
}
```

L3-1(1) `main()` starts off by initializing the I/Os we'll be using on this board.

L3-1(2) The example code assumes the presence of an RTOS called µC/OS-II and `OSInit()` is used to initialize µC/OS-II.

L3-1(3) µC/OS-II requires that we create at least ONE application task. This is done by calling `OSTaskCreateExt()` and specifying the task start address, the top-of-stack to use for this task, the priority of the task and a few other arguments.

L3-1(4) µC/OS-II allows you to assign names to tasks that have been created. We thus assign a name to the application task. These names are used mostly during debug. In fact, task names are displayed during debug when using IAR's C-Spy debugger (or other µC/OS-II aware debuggers).

L3-1(5) In order to start multitasking, your application needs to call `OSStart()`. `OSStart()` determines which task, out of all the tasks created, will get to run on the CPU. In this case, µC/OS-II will run `AppTaskStart()` because it's the most important task create (based on its priority).

The first, and only 'application' task that µC/OS-II runs is shown in listing 3-2.

Listing 3-2

```
static  void  AppTaskStart (void *p_arg)
{
    (void)p_arg;                    /* Prevent compiler warning                        */

    BSP_InitIntCtrl();          (1) /* Initialize the interrupt controller             */

    APP_DEBUG_TRACE("Initializing: Timers\n");
    Tmr_Init();                 (2) /* Start timers                                    */

#if (OS_TASK_STAT_EN > 0)
    APP_DEBUG_TRACE("Initializing: uC/OS-II Statistics\n");
    OSStatInit();               (3) /* Start uC/OS-II's statistics task                */
#endif

    AppInit_TCPIP();            (4) /* Initialize uC/TCP-IP                            */

    AppInit_DHCPc();            (5) /* Initialize DHCP client (if present)             */

    AppInit_FS();               (6) /* Initialize the file system                      */

                                (7) /* Initialize the TFTP server                      */
    TFTPs_Init(1000);

                                (8)
    APP_DEBUG_TRACE("Creating Application Tasks.\n");
    AppTaskCreate();                /* Create application tasks                        */

    LED_Off(1);
    LED_Off(2);
    LED_Off(3);

                                (9)
    while (DEV_YES) {               /* Task body, always written as an infinite loop.  */
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
```

L3-2(1) Initialises µC/OS-II tick interrupt and the AT91RM9200's interrupt controller.

L3-2(2) Initialize and start the kernel timer tick. This timer determines how many times per seconds the kernel will be informed that the time advances.

L3-2(3) If you set `OS_TASK_STAT_EN` to `1` in `os_cfg.h` then the µC/OS-II statistic task is initialized by calling `OSStatInit()`. `OSStatInit()` basically figures out how fast the CPU is running in order to determine how much CPU usage your application will be consuming. Details as to how this is done can be found in the µC/OS-II book (see References).

L3-2(4) We then initialize the TCP/IP stack by calling `NetInit()`. `NetInit()` initializes all of µC/TCP-IP's data structures and creates two tasks. One task waits for packets to be received and the other task manages timers. Because in this example we are using the µC/DHCPc service that will request an IP configuration from an external server and the µC/DHCPc module MUST be initialized after µC/TCP-IP, we MUST configure the µC/TCP-IP stack with generic values. This is needed for proper execution of the DHCP client. In the example code, you will notice that the MAC address (e.g. hardware address) is obtained from Micromonitor. You MUST replace this line by a section of code to obtain the MAC address from FLASH, EEPROM, etc.

L3-2(5) Configure and start the DHCP protocol. The µC/DHCPc package will try to contact a DHCP server to obtain IP configuration. The IP configuration obtained from the DHCP server is applied to the IP stack. See µC/DHCPc documentation for more details.

If your have not purchased the µC/DHCPc package, you can set your IP address, mask and gateway manually. See µC/TCP-IP documentation for more details.

L3-2(6) `AppInit_FS()` initializes the file system and `FS_IoCtl()` creates a RAM driver of 1440 Kbytes (similar to a 1.44MB diskette size). The RAM filesystem is formatted. Files are written and read from this RAM drive.

L3-2(7) Initialize µC/TFTPs module. The module is ready to accept requests from clients.

L3-2(8) Initialize LED task. This task blinks one of the LEDs. You should note that this task doesn't do anything other than blink the LED. Blinking the LED on the CSB337 board gives us an indication that the application is running.

L3-2(9) At this point the TCP/IP stack is initialized with the desired settings and the task enters an infinite loop. Now, you can use your preferred TFTP client, connect to your target and test it!

app_cfg.h

This file is used to establish the task priorities of each of the tasks in your application as well as the stack size for those tasks. The reason this is done here is to make it easier to configure task priorities for your entire application. In other words, you can set the task priorities of all your tasks in one place.

Ex1.*

These files are IAR embedded workbench project files.

fs_conf.h

This file is used to configure µC/FS and define runtime environment parameters such as RTOS support, POSIX support, type of hardware used to hold the file system and hardware-specific configuration.

includes.h

includes.h is a 'master' header file that contains #include directives to include other header files. This is done to make the code cleaner to read and easier to maintain.

net_cfg.h

This file is used to configure µC/TCP-IP and defines the number of timers used in µC/TCP-IP, the number of buffers for packets reception and transmission, the number of ARP (Address Resolution Protocol) cache entries, the number of Sockets that your application can open and more. In all, there are about `50` or so `#define` to set in this file.

os_cfg.h

This file is used to configure µC/OS-II and defines the number maximum number of tasks that your application can have, which services will be enabled (semaphores, mailboxes, queues, etc.), the size of the idle and statistic task and more. In all, there are about `60` or so `#define` that you can set in this file. Each entry is commented and additional information about the purpose of each `#define` can be found in the µC/OS-II book.

3.02.02 BSP (Board Support Package) Source Files

The concept of a BSP (Board Support Package) is to hide the hardware details from the application code. It is important that function names in a BSP reflect the function and does not make references to any CPU specifics. For example, the code to turn on an LED is called `LED_On()` and not `CSB337_led()`. If you use `LED_On()` in your code, you can easily port your code to another processor (or board) simply by rewriting `LED_On()` to control the LEDs on a different board. The same is true for other services. You will also notice that BSP functions are prefixed with the function's group. LED services start with `LED_`, Timer services start with `Tmr_`, etc. In other words, BSP functions don't need to be prefixed by `BSP_`.

The CSB337 BSP is found in the following directory:

`.\EvalBoards\Cogent\CSB337\IAR\`**`BSP`**

The BSP directory contains the following files:

```
bsp.c
bsp.h
net_bsp.c
net_bsp.h
net_isr.c
CSB33x_lnk_ram.xcl
```

`bsp.c` and `bsp.h`

`bsp.c` contains I/O, timer initialization code, LED control code, and more. The I/Os that we use on the board are initialized when `BSP_Init()` is called. `bsp.c` also contains functions to turn ON and OFF LEDs, toggle LEDs, configure CPU interrupts and more.

`net_bsp.*`

This file contains code specific to the NIC (Network Interface Controller) used and other functions that are dependent of the hardware. Specifically, this file contains code to read data from and write data to the NIC, provide delay functions, control power to the NIC, get a time stamp and more.

`net_isr.c`

This file contains code to initialize interruptions from the NIC and clear them after they are handled.

`CSB33x_lnk_ram.xcl`

This file contains the linker command file for the IAR toolchain. This file specifies where code and data is placed in memory. In this case, all the code is placed in RAM to make it easier to debug. When you are ready to deploy your product, you will most likely need to create a `CSB33x_lnk_flash.xcl` to locate your code in flash instead of RAM.

3.02.03 CPU Source Files

CPU manufacturers typically provide you with C header files that define their CPUs and the I/Os found on some of these chips. The CSB337 board contains a Atmel AT91RM9200 and the I/O definitions are found in the following directory:

```
.\CPU\Atmel\AT91RM9200\*.*
```

You should note that our directory structure architecture allows us to support multiple CPUs and thus, we would create CPU directories using the following scheme:

```
.\CPU\<manufacturer>\<CPU>\*.*
```

Where:

<manufacturer>	is the name of the CPU manufacturer.
<CPU>	is the name of the CPU.

3.02.04 µC/CPU Source Files

Some support functions are needed to adapt the software to different CPUs and compilers. This is different from the files provided by the CPU manufacturer as described in the previous section. Specifically, we need to specify what C data type is needed for a 16-bit unsigned integer, a 16-bit signed integer, a 32-bit signed integer, etc. Some compilers might assume that an `int` is 16 bits while others might assume that an `int` is 32 bits. To avoid the confusion, we defined data types that remove this confusion. In other words, we declare the following data types:

```
CPU_VOID
CPU_BOOLEAN
CPU_CHAR
CPU_INT08U
CPU_INT08S
CPU_INT16U
CPU_INT16S
CPU_INT32U
CPU_INT32S
CPU_FNCT_PTR
```

We also declared two functions that are used to disable and enable interrupts, `CPU_SR_Save()` and `CPU_SR_Restore()`, respectively.

The CPU/compiler specific files are placed in the following directory:

```
.\uC-CPU\<CPU>\<compiler>\*.*
```

Where:

<CPU>	is the name of the CPU or, a generic name that represents a family of CPUs.
<compiler>	is the name of the compiler manufacturer.

3.02.05 µC/DHCPc Source Files

The µC/DHCPc package enables the test application to run anywhere without IP configuration, assuming a DHCP server is present on the local network. At runtime, the test application is provided an IP address, a network mask, and the IP address of the network gateway by the DHCP server. The µC/DHCPc package assumes infinite IP address leases only.

To use the µC/DHCPc package, the following files need to be included in your application:

```
.\uC-DHCPc\Source\*.*
```

3.02.06 µC/FS Source Files

µC/TFTPs requires the presence of a file system to work properly. In fact, µC/TFTPs assumes µC/FS as the file system but, you should be able to easily replace µC/FS with a different file system by simply 'emulating' µC/FS's API (Application Programming Interface). In fact, µC/TFTPs only uses 4 functions from µC/FS:

```
FS_FOpen()
FS_FClose()
FS_FRead()
FS_FWrite()
```

As you would expect, these functions replaces the standard C functions `fopen()`, `fclose()`, `fread()` and `fwrite()`, respectively. However, the µC/FS arguments are not identical to the standard C functions and thus, you should consult the µC/FS manual for details about the API.

In order to use µC/FS, we need to include all the source files found in the following directories when building the sample application:

```
.\uC-FS\FS\API\*.*
.\uC-FS\FS\CLIB\*.*
.\uC-FS\FS\DEVICE\RAM\*.*
.\uC-FS\FS\FSL\FAT\*.*
.\uC-FS\FS\LBL\*.*
.\uC-FS\FS\System\FS_X\fs_os.h
.\uC-FS\FS\System\FS_X\fs_x_ucos_ii.c
```

Because we decided to create a RAM disk, the file system needs to be compiled with the RAM 'DEVICE' specific code for µC/FS. If we used a different media, we would include a different driver with the application.

3.02.07 µC/LIB Source Files

µC/DHCPc, µC/TFTPs and µC/TCP-IP doesn't make use of any of the standard C library functions `strcpy()`, `strcat()`, `memcpy()`, `memset()` etc. Instead, these and other functions have been re-written from scratch to provide similar functionality. The reason we did this is to make it easier to get our software validated by third parties (e.g. FAA, FDA, etc.) because we provide all the source code to our products. You might argue that most compiler vendors also provide source code for these libraries. However, it's better (from a certification point of vue) to always certify the same source code instead of different source from multiple compiler vendors. The libraries used by µC/DHCPc, µC/TFTPs and µC/TCP-IP are found in the following directory :

```
.\uC-LIB\*.*
```

3.02.08 µC/OS-II Source Files

µC/TFTPs and µC/TCP-IP require the presence of a Real Time Operating System (RTOS). The sample application uses µC/OS-II. The following files need to be included in your application:

```
.\uCOS-II\Source\*.*
.\uCOS-II\Ports\ARM\Generic\IAR\*.*
```

The CSB337 board contains a Atmel AT91RM9200 (ARM9) CPU. The µC/OS-II port for this CPU is found in the directory shown above. In fact, the µC/OS-II ARM port is generic and can thus be used with other ARM CPUs.

3.02.09 µC/TCP-IP Source Files

µC/DHCPc and µC/TFTPs assume the presence of µC/TCP-IP, a TCP/IP stack designed specifically for embedded systems. The following files need to be included in your application:

```
.\uC-TCPIP\IF\*.*
.\uC-TCPIP\IF\Ether\*.*
.\uC-TCPIP\NIC\Ether\AT91RM9200\*.*
.\uC-TCPIP\OS\uCOS-II\*.*
.\uC-TCPIP\Source\*.*
```

You should note that the CSB337 use a Atmel AT91RM9200 microcontroller which contains its own ethernet controller and thus, we need to include the driver for that chip in our build (as shown in the directory above).

References

µC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
CMP Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks
Jean J. Labrosse
CMP Books, 2000
ISBN 0-87930-604-1

Contacts

CMP Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
e-mail: rushorders@cmpbooks.com
WEB: http://www.cmpbooks.com

Cogent Computer Systems, Inc.
1130 Ten Rod Road, Suite A-201
North Kingstown, RI 02852 USA
USA
+1 401 295 6505
+1 401 295 6507 (Fax)
WEB: www.CogComp.com

IAR Systems
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
e-mail: Info@IAR.com
WEB : www.IAR.com

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com

Validated Software
Lafayette Business Park
2590 Trailridge Drive East, Suite 102
Lafayette, CO 80026
USA
+1 303 531 5290
+1 720 890 4700 (FAX)
e-mail: Sales@ValidatedSoftware.com
WEB: www.ValidatedSoftware.com