Micriμm

μC/HTTPs

HyperText Transfer Protocol (Server)

User's Manual

www.Micrium.com

Table of Contents

1.00 Introduction

HTTP (Hyper Text Transfer Protocol) is a protocol designed to give access to resources of the internet in a uniform way. The WWW is based on the HTTP protocol. HTTP has been implemented on top of the Internet Transmission Control Protocol (TCP).

µC/HTTPs is an add-on product to µC/TCP-IP that implements the HTTP protocol. The 's' in µC/HTTPs stands for 'server'. The µC/HTTPs module implements a part of:

RFC 1945:	ftp://ftp.rfc-editor.org/in-notes/rfc1945.txt
RFC 2145:	ftp://ftp.rfc-editor.org/in-notes/rfc2145.txt
RFC 2616:	ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt

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

µC/HTTPs executes as a task under a Real-Time Operating System (RTOS). The RTOS can be just about any RTOS that supports multi-tasking but we used µC/OS-II in our example.

µC/HTTPs assumes the presence of a File System which allows you to read files from some mass storage device (RAM disk, SD/MMC, IDE, on-board Flash, etc.). µC/HTTPs 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/HTTPs code to use your own file system. In fact, we recommend that you create a file that simulates µC/FS's API. This way, if we make modifications to µC/HTTPs, you would not have to redo this work.

You may notice also that packages µC/DHCPc and µC/FTPs are needed to build the example application but are not needed by µC/HTTPs. These packages have been used in the example for convenience. µC/DHCPc automatically obtains a valid IP configuration from a DHCP server (our example had a router connected to the network that provided this service). µC/FTPs is used to install the test web page or any user web page onto the target.

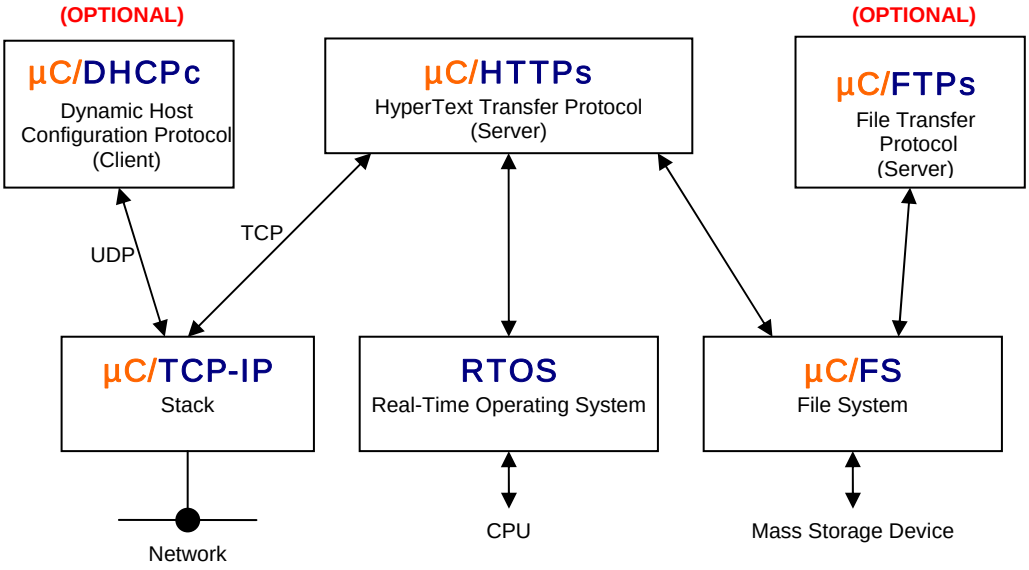

Figure 1-1, Relationship between the different modules.

2.00 Directories and Files

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

`\Micrium\Software\uC-HTTPs`

This is the main directory for µC/HTTPs.

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

This directory contains a ***template*** of µC/HTTPs configuration.

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

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

```
http-s.c
http-s.h
```

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

`\Micrium\Software\uC-HTTPs\`**`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 `http-s_os.c`.

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

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

3.00 Using µC/HTTPs

Figure 3-1 shows the test setup to demonstrate the use of µC/HTTPs. We used a Cogent Computer CSB337 AT91RM9200 based target (ARM9) connected to a Windows-based PC through a 100-Base-T Hub. The setup also included a 'router' which provided a DHCP server.

The PC's IP address was set to `192.168.0.102` and the target address was `192.168.0.3` (both obtained from the DHCP server). Depending on your router configuration, you may obtain different addresses.

The setup also used an RS-232C serial port on the target to provide a console output. We opened a Hyper Terminal session on the Windows PC to monitor the console output. When the target application is started, the IP address obtained for the target is output onto the Console Port.

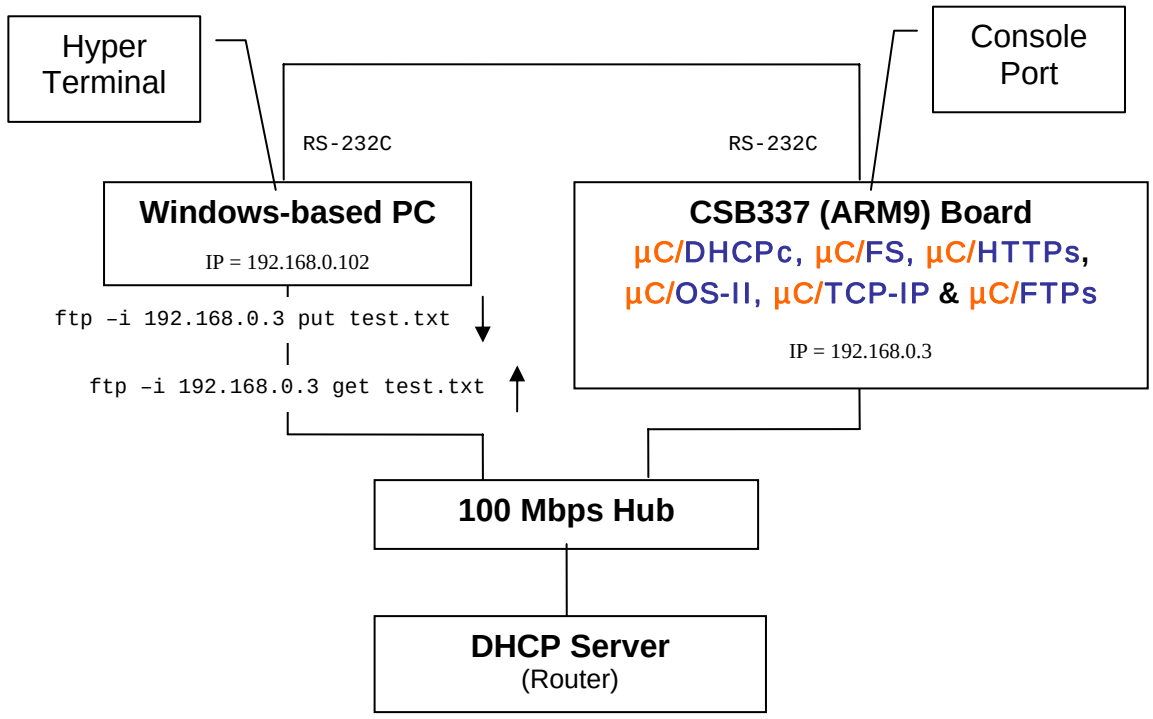

Figure 3-1, Test setup

The sample application was created using the IAR Embedded Workbench V4.31a and debugged with IAR's C-Spy which connects to the target using a J-Tag emulator.

3.01 Modules

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

µC/DHCPc
µC/FS
µC/FTPs
µC/HTTPs
µC/OS-II
µC/TCP-IP

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 medias 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 but we could just as well have used the on-board CF interface of the CSB337 board.

µC/HTTPs is an implementation of the server portion of the HTTP protocol. This protocol enables transfer of any type of document from a central source (server) using uniform addressing between all servers. In other words, µC/HTTPs can serve up web pages as well as other files to an HTTP client (e.g. a Web Browser).

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

µC/TCP-IP is an embedded TCP/IP stack that provides IP v4 networking support.

µC/FTPs is an implementation of the FTP (file transfer protocol). This protocol enables transfer of files between a host (Windows PC in our example) and a target (CSB337 in our example).

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 µC/HTTPs Configuration

The µC/HTTPs module requires some configuration. In the µC/HTTPs package, there is a file named `http-s-cfg.h` which is a template for configuration variables. You should copy this configuration to your `app-cfg.h` file and modify the values according to your specific needs. Here is the list of values and description of each variable:

```
#define  HTTPS_OS_CFG_TASK_NAME          "HTTP (Server)"
```

This is a value for the µC/HTTPs task, used mostly for debugging purposes. With µC/OS-II aware debuggers, you can recognize easily this task among the others in the system.

```
#define  HTTPs_OS_CFG_TASK_PRIO            13
```

This is a value for the µC/HTTPs task priority. The right value depends of the software architecture of your system and the relative importance of the response time of this task against the others.

```
#define  HTTPs_OS_CFG_TASK_STK_SIZE      2048
```

This is a value for the µC/HTTPs task stack size. With this value, there is enough stack size for most environments, but you should verify this on your system for reliability or tweaking purposes.

```
#define  HTTPs_CFG_IPPORT                  80
```

This value defines the IP port which µC/HTTPs will listens for requests. The default value is `80`. You can use a different value. Example: if you use the value `8080`, you will have to enter `http://192.168.0.3:8080/<html_page.htm>` to access the `html_page.htm` file.

```
#define  HTTPs_CFG_MAX_ACCEPT_TIMEOUT_S                    -1
#define  HTTPs_CFG_MAX_RX_TIMEOUT_S                         5
#define  HTTPs_CFG_MAX_TX_TIMEOUT_S                         5
```

These values define the timeout values (in seconds) for sockets. If the server doesn't reply back in this amount of time after a request, the request will be aborted.

```
#define  HTTPs_CFG_MAX_ACCEPT_RETRY                        -1
#define  HTTPs_CFG_MAX_RX_RETRY                             3
#define  HTTPs_CFG_MAX_TX_RETRY                             3
```

These values define the maximum number of retries when a request fails before returning an error to the application.

```
#define  HTTPs_CFG_FS_ROOT                "/HTTPRoot"
```

This is a value for the path in your filesystem that µC/HTTPs will see as its own root path. With this example value, if the target has the IP address `192.168.0.3` and you put a file named "`test.hml`" in the path `/HTTPRoot` in your filsystem, µC/HTTPs will "see" it at the address http://192.168.0.3/test.html. You will notice that with this configuration, there is no way for µC/HTTPs to access files which are not under the `/HTTPRoot` path. This is a security feature and you should use it. Place your internal files in other path!

```
#define  HTTPs_CFG_DFLT_FILE              "index.htm"
```

This value defines the default file returned to browser when no file specified. With this value, if the address http://192.168.0.3/ only is entered, µC/HTTPs will return the `index.htm` file.

```
#define  HTTPs_CFG_MAX_VAR_LEN             255
#define  HTTPs_CFG_MAX_VAL_LEN             255
```

These values define the maximum size of dynamic variables in you HTML files, e.g. the `${variables}` and the maximum size of their replacement values by your application.

```
#define  HTTPs_CFG_ERR_MSG_HTML_NOT_FOUND

   "<HTML>\r\n" \
   <…>
   "</HTML>\r\n"
```

This value defines the message returned by µC/HTTPs to your browser when a requested file is not found.

3.03 Transferring Web Pages to the Target

The example code assumes the presence of µC/FTPs which can be used to transfer web pages from a host (Windows PC) to your target. However, µC/FTPs is not required, especially if you use a CompactFlash card (or other mass-storage medium) and you simply copy the web pages from a Windows PC or other.

After the target has booted and µC/TCP-IP has initialized, µC/DHCPc (used in our example) will obtain an IP configuration.

To use FTP to upload HTML pages onto the target, you simply open a 'Command Prompt' window and typed either:

```
ftp 192.168.0.3
```

Remember that <192.168.0.3> may be replaced with the actual IP address of your target obtained by DHCP. You will enter into the FTP shell with all necessary commands to send and retrieve files to/from your target. You can also use a commercial, graphical & more intuitive FTP client.

Ensure that you will place your web page into your target in the same directory that you have configured the HTTPs_CFG_FS_ROOT in app_cfg.h. µC/HTTPs will use this directory as the web page root directory. Note that it is strongly recommended in a production application to place the WEB page alone in its own directory and set HTTPs_CFG_FS_ROOT to that directory. If you set HTTPs_CFG_FS_ROOT to the filesystem root path, users will be able to download or upload ANY file in your file system, possibly creating a security issue.

When your WEB page is uploaded, you are ready to browse it. Using your favorite browser, enter: http://192.168.0.3/webpage.htm. You will see the content of webpage.htm. If you have uploaded a file named HTTPs_CFG_DEFAULT_FILE (see app_cfg.h), generally "index.htm", you can enter http://192.168.0.3. The index.htm file will be displayed automatically.

3.04 Web Pages Format

When a WEB document is requested (GET command, see `rfc1945.txt`), if the file suffix is `.htm` or `.html`, then the server will parse the file prior to sending it to the client. If the syntax `${TEXT_STRING}` is found within the file, then the string `TEXT_STRING` is passed to an application-specific function and the all characters from the opening '$' to the closing '}' are omitted and replaced with whatever the application-specific code wants to replace it with (see section 3.04.02, Handling `${TEXT_STRING}`).

For POST command, (generated by the HTML "form" submission), the input is assumed to be formatted as "POST /action" where 'action' defines what the server is to do with the next incoming TCP stream. This next stream is the data entered through the user interface presented by the HTML file that contains the form mechanism (the cgi name/value pairs). It can simply be an HTML file that is processed in the same way as was discussed above for the GET command (see section 3.04.04, Handling POST /action).

Similarly, if the action of a form is a `.htm` or a .html file, the server will first parse the incoming name/value list and pass each name value pair into an app-specific function called `HTTPs_ValRx()` to allow the application to properly parse the data entered by the user of the client. Then the HTML file specified as the action will be processed as discussed above and sent to the client.

3.04.01 Web Pages Format, Example ${TEXT_STRING}

Assume we have a file called `myip.htm` on the target that looks like this:

```
<html><body><center>
This system's IP address is ${My_IP_Address}
</center></body></html>
```

When a client attaches to this server and requests (via GET) `myip.htm`, the server (i.e. µC/HTTPs) will parse the file, find the `${My_IP_Address}` syntax and pass the string `"My_IP_Address"` into an application specific function called `HTTPs_ValRx()`. That function will then build a replacement text string that the server will give to the client in place of the `${My_IP_Address}` text. The file seen by the client would look something like this:

```
<html><body><center>
This system's IP address is 135.17.115.215
</center></body></html>
```

Note that this server does not define any syntax within the `${}`. This is 100% application-specific and can be used for simple variable name conversion or something more elaborate if necessary.

3.04.02 Handling ${TEXT_STRING}

When a client request a web page containing `${variable}` tokens, µC/HTTPs recognize them and call back the application (`HTTPs_ValReq()`) function once of each variable. You need to implement this call back in your application. Here is the code for the callback implemented in the example application.

`AppIPAddr`, `AppIPMsk` and `AppIPGw` are global values for IP address, mask and gateway, respectively. The code returns values for known variables.

```
CPU_BOOLEAN HTTPs_ValReq (CPU_CHAR     *Variable,
                          CPU_CHAR    **Val,
                          CPU_INT32U    MaxSize)
{
    static  CPU_CHAR  buf[20];
            NET_ERR   err;


    Str_Copy(&buf[0], "%%%%%%%%");
   *Val = &buf[0];

[…]
    if (Str_Cmp(Variable, "IP"  ) == 0) {
        NetASCII_IP_to_Str(NET_UTIL_NET_TO_HOST_32(AppIPAddr), &buf[0], DEF_NO,
             &err);

    } else if (Str_Cmp(Variable, "MASK") == 0) {
        NetASCII_IP_to_Str(NET_UTIL_NET_TO_HOST_32(AppIPMsk),  &buf[0], DEF_NO,
             &err);

    } else if (Str_Cmp(Variable, "GW"  ) == 0) {
        NetASCII_IP_to_Str(NET_UTIL_NET_TO_HOST_32(AppIPGw),   &buf[0], DEF_NO,
             &err);
    }
[…]

    APP_DEBUG_TRACE ("HTTP-s: VALUE REQUESTED: %s = %s.\n", Variable, *Val);
    return (DEF_OK);
}
```

3.04.03 Web Pages Format, Example POST /action

A HTTP POST action occurs generally when you fill a form online and you "post" it. While the HTTP GET action is designed to transfer information from the server to the browser, POST is designed to send information to the server.

The two most important parts of a POST action is the URL and the data. The document pointed by the URL is usually a script processing the data but can also be an HTML page.

The data part is transferred to the server after the connection to the URL was accepted. Here is an example of a form used on internet:

When the "`I Accept`" button is pressed, a script on the server is requested with a POST command and the data in the form is transferred to the script. The data is encoded in a special manner, like this:

```
&required-name=Joe+Dalton&required-company=every+bank&required-
phone=1234567890&required-address1=free&address2=&required-
city=far+west&required-state=YYY&required-zip=12345&required-
country=USA&required-e-mail=joe@dalton.com&submit=I+Accept
```

Spaces are replaced by '`+`', special characters are encoded by there ASCII value, preceded by the '`%`' token and other transformations of the data occurs. This is called "URL encoding".

You will notice that form fields have labels (hidden from the user). These labels and their values are transferred in the form `&<label>=<value>` to the script.

You can use form with µC/HTTPs. To use them, create a standard HTML page with a form. When the form is posted (the way to write a form is beyond the scope of this document; see HTML standards), µC/HTTPs will handle the POST action and will call back your application for every `&<label>=<value>` transmitted.

You can use any label as long as you handle them in your application. See section 3.04.04 for example of code handling POST action.

3.04.04 Handling POST /action

When a client does a POST action, the `HTTPs_ValRx()` function is called for every label in the form submitted. You can handle them and use them in your application.

```
CPU_BOOLEAN  HTTPs_ValRx (CPU_CHAR  *var,
                          CPU_CHAR  *val)
{
    if (Str_Cmp(var, "required-name") == 0) {
        printf ("Name = %s.\n", val);
        if (Str_Cmp(val, "Joe Dalton") == 0) {
            printf ("You are WANTED!\n");
        }
    }

    if (Str_Cmp(var, "required-company") == 0) {
        printf ("Company = %s.\n", val);
    }

    <...>

    APP_DEBUG_TRACE ("HTTP-s: VALUE RECEIVED: %s = %s.\n", Variable,
                     Val);
    return (DEF_OK);
}
```

You will notice that µC/HTTPs removes the 'URL encoding' of the data before to pass it to your application.

3.05 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 packages µC/DHCPc and µC/FTPs are needed to build the example application. These packages have been used for convenience. µC/DHCPc obtains automatically a valid IP configuration from a DHCP server. µC/FTPs is needed to install the test web page or any user web page into the target.

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
```

3.05.01 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
app_cfg.h
Ex1.*
fs_conf.h
includes.h
net_cfg.h
os_cfg.h
```

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 CSB337 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 *),
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

                          /* (4) Give a name to tasks                               */
#if (OS_TASK_NAME_SIZE > 16)
    OSTaskNameSet(OS_TASK_IDLE_PRIO,   "Idle task",  &err);
    OSTaskNameSet(OS_TASK_STAT_PRIO,   "Stat task",  &err);
    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()`.

The 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)
    FTPs_Init;                          /* Initialize the FTP server                         */

                                (8)
    HTTPs_Init();                       /* Initialize the HTTP server                        */

                                (9)
    APP_DEBUG_TRACE("Creating Application Tasks.\n");
    AppTaskCreate();                    /* Create application tasks                          */

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

                               (10)
    while (DEF_YES) {                   /* Task body, always written as an infinite loop.    */
        OSTimeDlyHMSM(0, 0, 0, 100);
    }
}
```

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

L3-2(2) Initialize and start 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.

The MAC address is obtained from the Micromonitor in this example.

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.

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/FTPs module. The module is ready to accept requests from clients.

L3-2(8) Initialize µC/HTTPs module. The module is ready to accept requests from clients.

L3-2(9) Initialize LED task. You can start your own tasks here!

L3-2(14) At this point the TCP/IP stack is initialized with the desired settings and the task enters an infinite loop and blinks one of the LEDs. You should note that this task doesn't do anything other than blink the LED. We could have just as well terminated this task but, blinking the LED on the CSB337 board gives us an indication that the application is running.

Now, you can upload some HTML files to the file system using the optional µC/FTPs or µC/FTPs module, and a FTP client (such as via a Windows Command Prompt). When it's done, you can browse the pages.

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 filesystem 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.

3.05.02 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 `CSB337_lnk_flash.xcl` to locate your code in flash instead of RAM.

3.05.03 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.

3.05.04 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.

3.05.05 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\*.*
```

3.05.06 µC/FS Source Files

µC/HTTPs and µC/FTPs require the presence of a file system to work properly. In fact, µC/HTTPs and µC/FTPs assume µC/FS as the file system but, you should be able to replace µC/FS with a different file system by 'emulating' µC/FS's API (Application Programming Interface. 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.

There is an adaptation layer between the package µC/FS and the Real Time Operating System (RTOS) used in the test application, here µC/OS-II. If you use another RTOS, you have to change the adaptation layer used accordingly.

3.05.07 µC/FTPs Source Files

To test the µC/HTTPs package, we need a test web page present on the file system of the target. The µC/FTPs package enables transfer of the test web page from your PC to the file system of the target prior to test phase.

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

```
.\uC-FTP\Source\*.*
.\uC-FTP\OS\uCOS-II\*.*
```

There is an adaptation layer between the package µC/FTPs and the Real Time Operating System (RTOS) used in the test application, here µC/OS-II. If you use another RTOS, you have to change the adaptation layer used accordingly.

3.05.08 µC/LIB Source Files

µC/DHCPc, µC/HTTPs, µC/TCP-IP and µC/FTPs 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/HTTPs, µC/TCP-IP and µC/FTPs are found in the following directory:

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

3.05.09 µC/OS-II Source Files

µC/HTTPs and µC/FTPs 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. 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.

3.05.10 µC/TCP-IP Source Files

µC/HTTPs and µC/FTPs 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