

# Micrium

© Copyright 2005-2006, Micrium  
All Rights reserved

## μC/FTP<sub>S</sub>

File Transfer Protocol (Server)

**User's Manual**

[www.Micrium.com](http://www.Micrium.com)

# Table of Contents

1.00	Introduction.....	3
2.00	Directories and Files .....	5
3.00	Using $\mu$ C/FTP.....	6
3.01	Commands .....	9
3.02	Integration with $\mu$ C/FS .....	11
3.03	Modules.....	12
3.04	Configuration .....	13
3.05	Creating the test application.....	15
3.06	Creating the test application.....	16
3.06.01	Example Application Source Files.....	17
3.06.02	BSP (Board Support Package) Source Files .....	21
3.06.03	CPU Source Files.....	22
3.06.04	$\mu$ C/CPU Source Files .....	22
3.06.05	$\mu$ C/DHCP Source Files.....	23
3.06.06	$\mu$ C/FS Source Files .....	23
3.06.07	$\mu$ C/LIB Source Files .....	24
3.06.08	$\mu$ C/OS-II Source Files .....	24
3.06.09	$\mu$ C/TCP-IP Source Files .....	24
	References.....	25
	Contacts.....	26

## 1.00 Introduction

FTP (File Transfer Protocol) is a protocol designed to enable reliable transfer of documents over the internet. FTP has been implemented on top of the Internet Transmission Control Protocol (TCP).

$\mu$ C/FTP<sub>s</sub> is an add-on product to  $\mu$ C/TCP-IP that implements the FTP protocol. The 's' in  $\mu$ C/FTP<sub>s</sub> stands for 'server'. The  $\mu$ C/FTP<sub>s</sub> module implements a part of:

RFC 959:	<a href="ftp://ftp.rfc-editor.org/in-notes/rfc959.txt">ftp://ftp.rfc-editor.org/in-notes/rfc959.txt</a>
RFC 2389:	<a href="ftp://ftp.rfc-editor.org/in-notes/rfc2389.txt">ftp://ftp.rfc-editor.org/in-notes/rfc2389.txt</a>
Draft IETF:	draft-ietf-ftptext-mlst-16.txt

Figure 1-1 shows a block diagram that shows the relationship between components needed to run  $\mu$ C/FTP<sub>s</sub> in a typical embedded target application. You should note that  $\mu$ C/FTP<sub>s</sub> assumes the presence of  $\mu$ C/TCP-IP but could actually be modified to work with just about any TCP/IP stack.

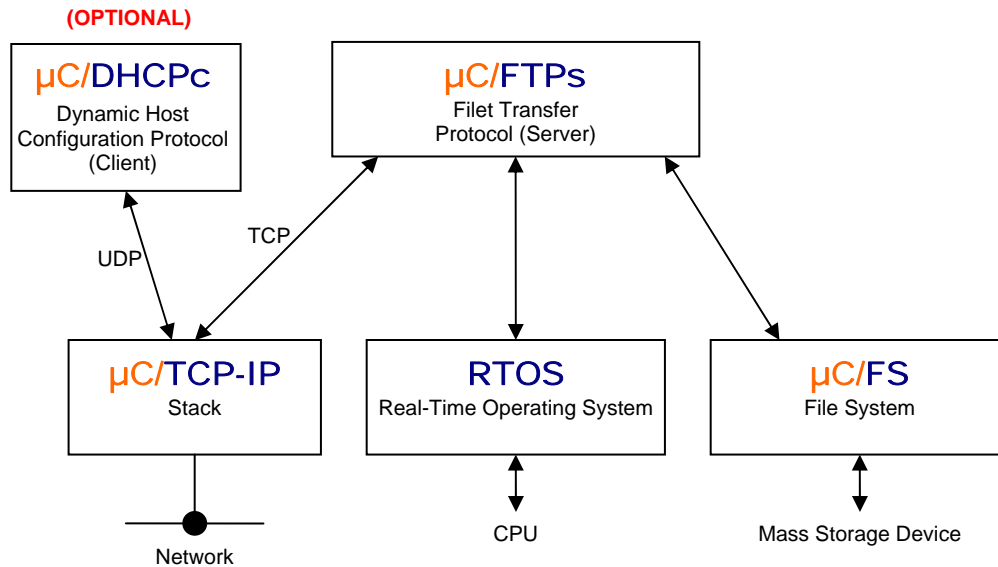
$\mu$ C/FTP<sub>s</sub> executes as two tasks under a Real-Time Operating System (RTOS). The RTOS can be just about any RTOS that supports multi-tasking but we used  $\mu$ C/OS-II in our example.

The first task is the server task. It listens on a socket (usually TCP port 21, but is configurable) for clients. When a client connects, a new TCP port (in the automatic port range) is assigned to that connection, so the TCP port 21 returns to the listen state, waiting for another client.

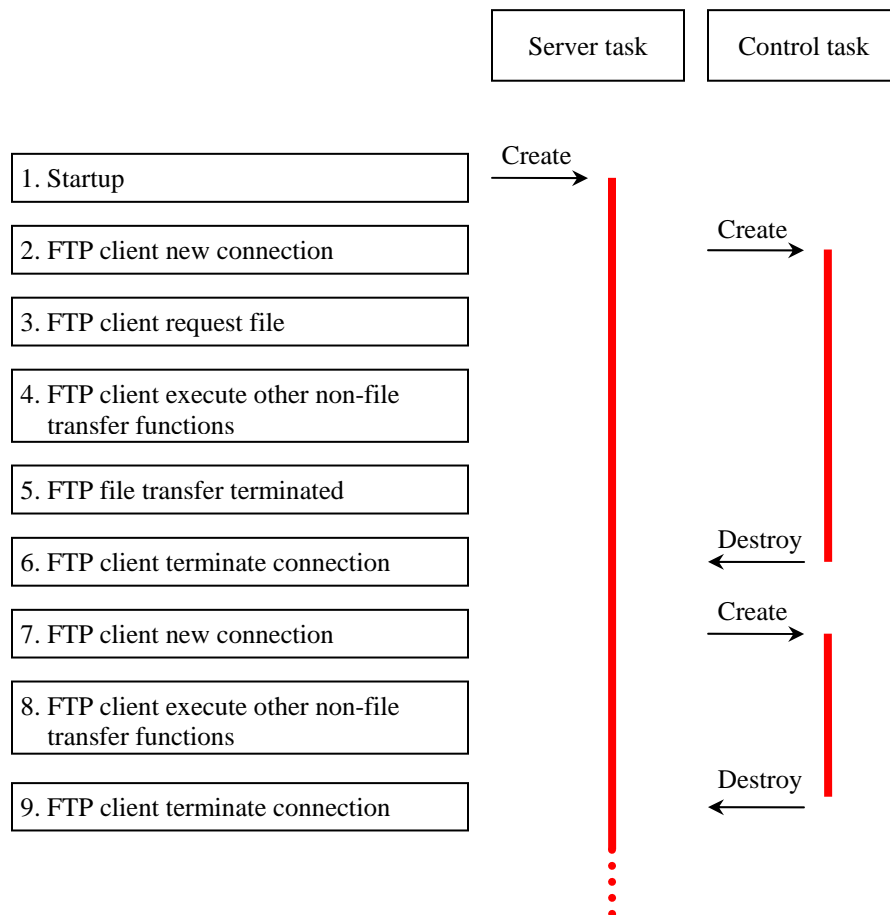
At this time, the control task is spawned (i.e. created) and waits for client requests. This task receives, interprets and executes the majority of FTP commands. Currently,  $\mu$ C/FTP<sub>s</sub> supports one client connection at a time, so there is only one control task at a time. When the FTP client close the connection, the socket is released and the task is terminated (i.e. deleted).

$\mu$ C/FTP<sub>s</sub> 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.).  $\mu$ C/FTP<sub>s</sub> assumes the API of Micrium's  $\mu$ C/FS. If you use a different file system, you will need to 'simulate'  $\mu$ C/FS's API calls or, modify the  $\mu$ C/FTP<sub>s</sub> code to use your own file system. In fact, we recommend that you create a file that simulates  $\mu$ C/FS's API. This way, if we make modifications to  $\mu$ C/FTP<sub>s</sub>, you would not have to redo this work.

You may notice also that package  $\mu$ C/DHCP<sub>c</sub> is needed to build the example application but are not needed by  $\mu$ C/FTP<sub>s</sub>. This package has been used in the example for convenience.  $\mu$ C/DHCP<sub>c</sub> automatically obtains a valid IP configuration from a DHCP server (our example had a router connected to the network that provided this service).



**Figure 1-1, Relationship between the different modules**



**Figure 1-2, Timeline representation of tasks execution.**

## 2.00 Directories and Files

The files for  $\mu$ C/FTP<sub>s</sub> are placed in multiple directories as described below.

`\Micrium\Software\uC-FTPs`

This is the main directory for  $\mu$ C/FTP<sub>s</sub>.

`\Micrium\Software\uC-FTPs\CFG\Template`

This directory contains a template of  $\mu$ C/FTP<sub>s</sub> configuration.

`\Micrium\Software\uC-FTPs\Source`

This directory contains the source code for the RTOS independent code for  $\mu$ C/FTP<sub>s</sub>. FTP is fairly easy to implement and thus, the code is found in just two files:

`ftp-s.c`  
`ftp-s.h`

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

`\Micrium\Software\uC-FTPs\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 `ftp-s_os.c`.

`\Micrium\Software\uC-FTPs\OS\uCOS-II`

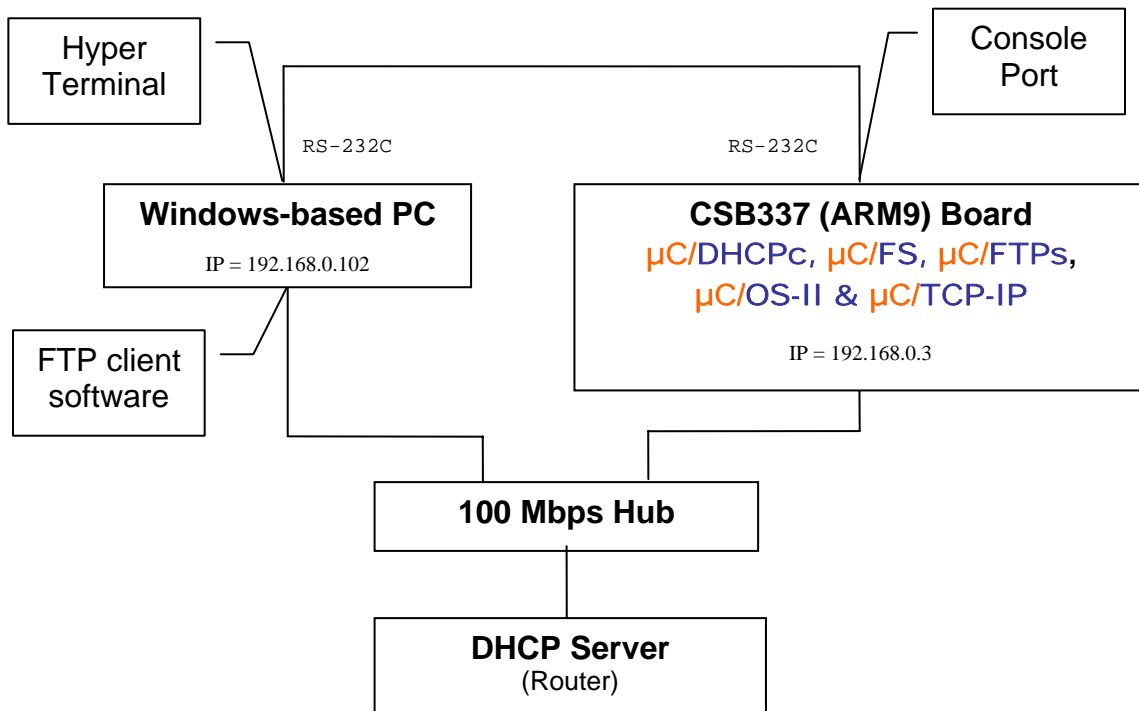
$\mu$ C/FTP<sub>s</sub> comes with the implementation file to interface to  $\mu$ C/OS-II.

## 3.00 Using $\mu$ C/FTP

Figure 3-1 shows the test setup to demonstrate the use of  $\mu$ C/FTP. 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 SOHO '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.



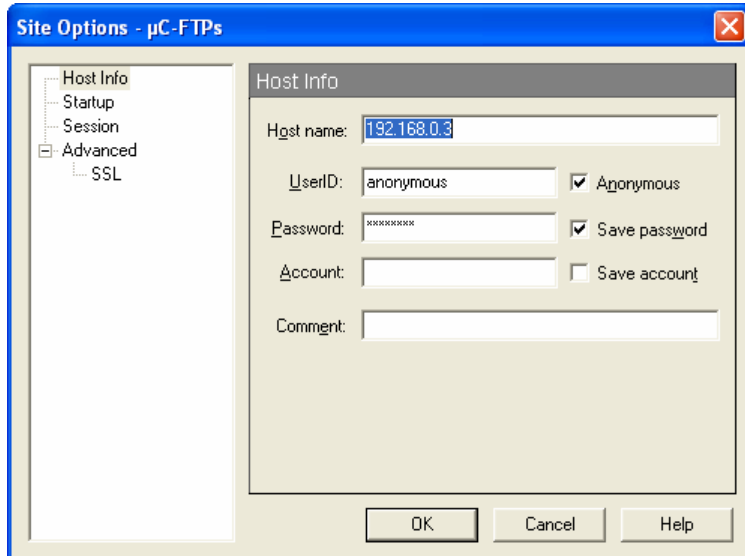
**Figure 3-1, Test setup**

On the target side, we are running the  $\mu$ C/FTP sample application. It 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.

On the PC side, we are running an FTP client (for this example, IPswitch's WS\_FTP Pro 8.0 (see [www.ipswitch.com](http://www.ipswitch.com))). This client has been configured to use with  $\mu$ C/FTP. Here are some snapshots of the site options windows of WS\_FTP Pro 8.0.

In the “host info” window, we have connection information. “anonymous” is the default user defined by FTP protocol. If you use authentication in your application, you may accept or not “anonymous” logins. If your application refuses “anonymous” logins, you have to set another UserID and password here.

The “account” field is not used with  $\mu$ C/FTPs.



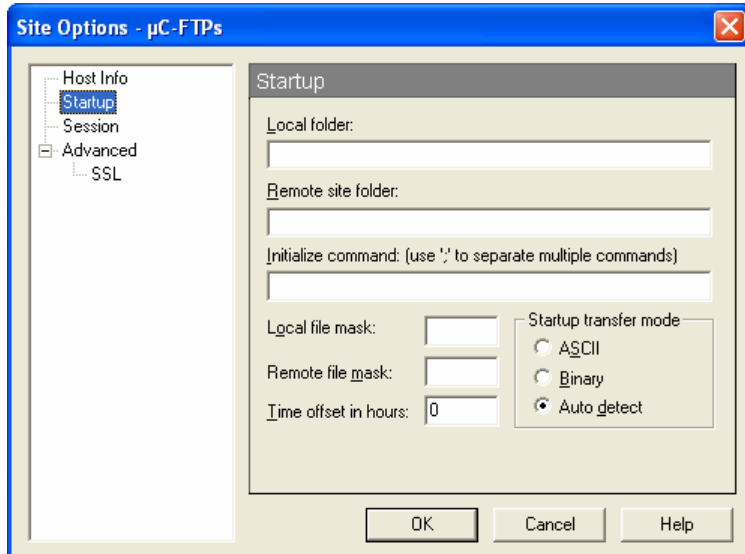
The "Site Options - μC-FTPs" dialog box has a tree view on the left with "Host Info" selected. The "Host Info" tab contains the following fields and options:

- Host name: 192.168.0.3
- UserID: anonymous ☒ Anonymous
- Password:  ☒ Save password
- Account:  ☐ Save account
- Comment:

Buttons at the bottom: OK, Cancel, Help.

In the “startup” window, we have information about initial state of the connection (startup path, startup commands, startup transfer mode and time offset between the client and the server in hours (for correct time handling when transferring files). We have let all the default values here.

Local and remote file mask are UNIX filesystem properties and are not used by  $\mu$ C/FTPs.



The "Site Options - μC-FTPs" dialog box has a tree view on the left with "Startup" selected. The "Startup" tab contains the following fields and options:

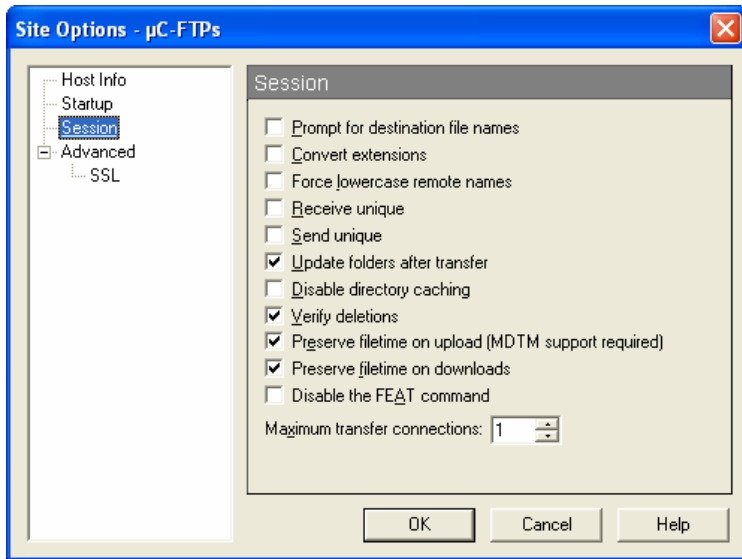
- Local folder:
- Remote site folder:
- Initialize command: (use ';' to separate multiple commands)
- Local file mask:
- Remote file mask:
- Time offset in hours: 0
- Startup transfer mode:
  - ☐ ASCII
  - ☐ Binary
  - ☒ Auto detect

Buttons at the bottom: OK, Cancel, Help.

In the “session” window, we have many configurable options. Since **µC/FTP**s supports long file names with uppercase and lower case characters, the first five options have been leaved unchecked to preserve these names exactly.

The next three options are for WS\_FTP behavior only, we let their default values. The next two options maintain file date/time on download and uploads.

“Disable FEAT command” is not checked. The FEAT command enable FTP clients to know about **µC/FTP**s capabilities beyond standard RFC959 commands.

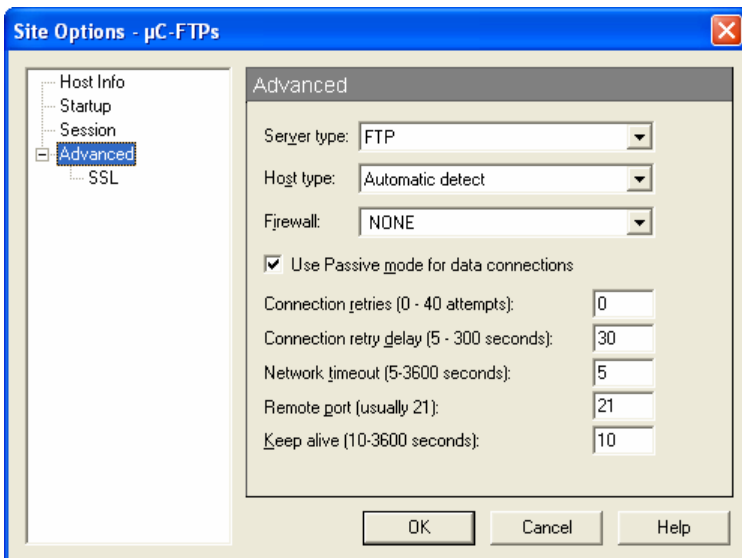


And the last but not least, the “Maximum transfer connections” **MUST** be set to “1”, because **µC/FTP**s supports one file transfer at a time. Failure to set correctly this option will result of wrong behavior and transfer failures. This is one of the reasons why we recommend to use a standalone FTP client instead of Windows Explorer or Internet Explorer, which start many connections at the same time and may cause **µC/FTP**s to hang temporarily.

In the “advanced” window, we assume many default values.

The option “use passive mode for data connection” can be checked or not. However, passive mode works better when there are firewalls between FTP client and server, so we leave it checked. Note that in this window we have a “firewall” setting be, in our understanding of what it configures, it should be renamed “Proxy settings” instead of “Firewall settings”.

Network timeout have been modified form its default value to “5”, the file transfer socket timeout configured for **µC/FTP**s. Failure to do that will result in file transfer aborts by the server.



Also, the control socket timeout is configured for 30 seconds in **µC/FTP**s' configuration file, so we set the “keep alive” signal to a value under 30 seconds. Failure to do that will result in server disconnection after 30 seconds.



## 3.01 Commands

FTP is a command-based protocol. Every FTP operation is the result of an exchange of human-readable commands and replies in clear text. For convenience, we list here all the supported commands of  $\mu$ C/FTPs, their arguments and replies.

However, there are many commercial FTP clients, like WS-FTP that abstract these commands. Some of these tools are very user friendly and look & feel like Windows Explorer and even integrate with.

So this list is provided for your convenience and for debugging purposes, but you don't need to remember them to use  $\mu$ C/FTPs!

- NOOP :** This is the NULL command. Used to keep alive connection between FTP client and server and beat connection time-outs.
- QUIT :** Terminate the FTP session.
- REIN :** Reinitialize FTP parameters (MODE, TYPE, STRU and Passive/Active connection to their default values).
- SYST :** Return a string describing the system type.
- FEAT:** Return list of extended features of the server beyond RFC959 definitions.
- HELP:** Return list of server supported commands.
- USER :** Authenticate user to the FTP server. The next command **MUST** be **PASS**.
- PASS :** Set password for user authentication. This command **MUST** be preceded by the **USER** command.
- MODE :** Set the transfer mode. RFC 959 defines three modes: **STREAM**, **BLOCK** and **COMPRESSED**. Only the **STREAM** mode is supported.
- TYPE :** Set data representation type. RFC 959 defines four types: **ASCII**, **IMAGE** (binary), **LOCAL** and **EBCDIC**. Only **ASCII** and **IMAGE** types are supported.
- STRU :** Set data structure. RFC 959 defines three structures: **FILE**, **RECORD** and **PAGE**. Only **FILE** structure is supported.
- PASV :** Enter passive mode. In passive mode, the server returns to the client information about an IP address and a port listening for a data connection. The client **MUST** connect to that address and port within a certain amount of time. The passive mode is more convenient with the use of firewalls because every connection is initiated from client to server. You **MUST** ensure that the information about the IP address and port sent by the server reflect its public IP address and port and not its internal information.  $\mu$ C/FTPs have provision to configure public IP address and port.

**PORT :** Enter active (default) mode. The client MUST provide its public IP and port data. The server MUST connect to the client within a certain amount of time. The active mode may cause problems with firewalls because the connections are in different directions and firewalls fail to correlate them.

**PWD :** Return the working directory. The returned directory is the FTP virtual directory. The FTP virtual directory is mapped to your filesystem using an FTP base directory.

**CWD :** Change working directory. Client MUST give a new FTP virtual directory. It can be relative to current directory or absolute.

**CDUP :** Go up one directory level. Similar to “CWD ..”

**MKD :** Create a new directory. Client MUST give a new FTP virtual directory. It can be relative to current directory or absolute.

**RMD :** Remove an existing directory. Client MUST give an FTP virtual directory. It can be relative to current directory or absolute.

**NLST :** Return file name list into a directory. Client MUST give an FTP virtual directory. It can be relative to current directory or absolute.

**LIST :** Return full file list into a directory. Client MUST give an FTP virtual directory. It can be relative to current directory or absolute.

**RETR :** Retrieve (download) a file. File name can be relative to current directory or absolute.

**STOR :** Store (upload) a file. File name can be relative to current directory or absolute.

**APPE:** Append (upload) a file. File name can be relative to current directory or absolute. If the file doesn't exist it will be created.

**REST:** Restart an interrupted transfer. The parameter is the offset (in bytes) of the file to restart. MUST be followed by a RETR or STOR command.

**DELE :** Delete a file. File name can be relative to current directory or absolute.

**RNFR :** Rename from a file or directory. File name of directory name can be relative to current directory or absolute. MUST be followed by an RNTD command.

**RNTD :** Rename to a file or directory. File name of directory name can be relative to current directory or absolute. MUST be preceded by an RNFR command.

**SIZE :** Get the size of a file. File name can be relative to current directory or absolute.

**MDTM :** Get or set the modification time of a file or directory.

## 3.02 Integration with $\mu$ C/FS

The integration of  $\mu$ C/FS and  $\mu$ C/FTP is more tight than with other  $\mu$ C/TCP-IP add-ons ( $\mu$ C/HTTP and  $\mu$ C/TFTP). This is needed because FTPs really emulate a file system with command that enables storage, retrieving, renaming, destruction and attribute modification of files and directory tree in the file system. Thus, more  $\mu$ C/FS commands, and some data structures are used to achieve this.

Here are the things that you must consider if you want to use another file system API other than  $\mu$ C/FS and emulate its API:

1. You MUST configure  $\mu$ C/FTP with the path separator used by your file system (typically “/” or “\”). Also, you MUST configure the way to reference the root path, the current path and the parent path. These configuration values are stored in `ftp-s.h` file.
2. You MUST have these file management functions (similar to C standard library):
  - **FS\_FOpen** (Open/Create file)
  - **FS\_FRead** (Read file)
  - **FS\_FWrite** (Write to file)
  - **FS\_FError** (Get last error)
  - **FS\_FClose** (Close file)
  - **FS\_FSeek** (Seek file to an offset)
3. You MUST have these directory management functions:
  - **FS\_MkDir** (Create directory)
  - **FS\_RmDir** (Remove directory)
  - **FS\_Move** (Move file/directory)
  - **FS\_Remove** (Remove file)
4. You MUST have directory browsing functions and data types:
  - **FS\_OpenDir** (Open directory for browsing)
  - **FS\_ReadDir** (Get next directory entry)
  - **FS\_CloseDir** (Close directory)
  - **FS\_SetFileTime** (Set file date and time)
  - A data type representing entire directory containing a list of directory entries
  - A data type representing directory entries containing the following data:
    - File/Directory name
    - File/Directory size
    - File/Directory date/time of modification
    - File/Directory attributes

## 3.03 Modules

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

`μC/DHCPc`  
`μC/FS`  
`μC/FTPc`  
`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 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/FTPc` is an implementation of the server portion of the FTP protocol. This protocol enables transfer of documents over the internet.

`μC/OS-II` is a real-time, multitasking kernel that allows you to have up to 250 application tasks. `μC/FTPc` runs as two of those tasks. You can use just about any other RTOSs with `μC/FTPc` 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 view) to always certify the same source code instead of different source from multiple compiler vendors.

## 3.04 Configuration

The  $\mu$ C/FTPs module requires some configuration. In the  $\mu$ C/FTPs package, there is a file named `ftp-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  FTPs_SRV_OS_CFG_TASK_NAME          "FTP (Server)"
#define  FTPs_CTRL_OS_CFG_TASK_NAME        "FTP (Control)"
```

$\mu$ C/FTPs uses two tasks. The first (Server) is the task listening for clients. The second (Control) is the task answering client requests.

These are values for the two  $\mu$ C/FTPs tasks, used mostly for debugging purposes. With  $\mu$ C/OS-II aware debuggers, you can recognize easily these tasks among the others in the system.

```
#define  FTPs_SRV_OS_CFG_TASK_PRIO          13
#define  FTPs_CTRL_OS_CFG_TASK_PRIO        14
```

These are values for the two  $\mu$ C/FTPs tasks priority. The right value depends of the software architecture of your system and the relative importance of the response time of these tasks against the others. It is recommended, however, to give consecutive task priorities to the two  $\mu$ C/FTPs tasks, and the server task should have the highest priority of the group.

```
#define  FTPs_SRV_OS_CFG_TASK_STK_SIZE      1024
#define  FTPs_CTRL_OS_CFG_TASK_STK_SIZE    2048
```

These are values (in processor words) for the two  $\mu$ C/FTPs tasks stack size. With these values, there is enough stack size for most environments, but you should verify this on your system for reliability or tweaking purposes. On an ARM7 system, we have tested stack requirements using  $\mu$ C/OS-View and found the following values (in 32-bit words):

```
Server task:          527
Control task:         1583
```

```
#define  FTPs_CFG_CTRL_IPPORT              21
#define  FTPs_CFG_DTP_IPPORT              20
```

These values define the IP ports which  $\mu$ C/FTPs will listens for requests. The default value is 21 for the control socket and 20 for the data transfer socket. You can use different values according to FTP client configuration.

```

#define  FTPs_CFG_CTRL_MAX_ACCEPT_TIMEOUT_S      (infinite)
#define  FTPs_CFG_CTRL_MAX_RX_TIMEOUT_S          30
#define  FTPs_CFG_CTRL_MAX_TX_TIMEOUT_S          5

#define  FTPs_CFG_DTP_MAX_ACCEPT_TIMEOUT_S        5
#define  FTPs_CFG_DTP_MAX_CONN_TIMEOUT_S          5
#define  FTPs_CFG_DTP_MAX_RX_TIMEOUT_S            5
#define  FTPs_CFG_DTP_MAX_TX_TIMEOUT_S            5

```

These values define the timeout values (in seconds) for control and data transfer sockets. Since the control socket may not receive commands for a while, the timeout can be longer. However, this FTP server is limited to one client at a time. If a client gets stuck, it may take a while before the server will release the socket. So we recommend you to keep the control socket connection timeout low and activate the “keep-alive” functions of your FTP client, which send “NOOP” command periodically.

```

#define  FTPs_CFG_CTRL_MAX_ACCEPT_RETRY          (infinite)
#define  FTPs_CFG_CTRL_MAX_RX_RETRY              3
#define  FTPs_CFG_CTRL_MAX_TX_RETRY              3

#define  FTPs_CFG_DTP_MAX_ACCEPT_RETRY            3
#define  FTPs_CFG_DTP_MAX_CONN_RETRY              3
#define  FTPs_CFG_DTP_MAX_RX_RETRY                3
#define  FTPs_CFG_DTP_MAX_TX_RETRY                3

```

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

```

#define  FTPs_CFG_MAX_USER_LEN                    32
#define  FTPs_CFG_MAX_PASS_LEN                    32

```

These values define the maximum length of user and password got from FTP client that will be transferred to application for validation.

## 3.05 Implementing FTP user authentication

To control the access of your data via the FTP protocol, there is some authentication provision that you MUST implement.

When an FTP client tries to access your FTP server (here `µC/FTPs` on your target), it sends a user name and a password to the FTP server. Authentication of these user name avec password is out of `µC/FTPs`' scope. Thus, these user name and password MUST be validated and some information about the user's home directory and browsing permission MUST be returned by your application.

This authentication is implemented in your application using the callback function from `µC/FTPs`:

```
CPU_BOOLEAN  FTPs_AuthUser(FTPs_SESSION_STRUCT  *ftp_session);
```

The user name is contained in `ftp_session->User`.

The password is contained in `ftp_session->Pass`.

If your application authenticates the user, the function `FTPs_AuthUser` MUST return `DEF_OK`, else `DEF_FAIL` value.

Also, your application MUST return two directories information.

First, the base path, located at `ftp_session->BasePath`, represents the highest directory in the file system that the user is allowed to see and access.

Second, the relative path, located at `ftp_session->RelPath`, represents the current relative directory for base path that the user sees and access.

Examples:

1. If you want the user "user1" to access his or her own account "/FTPROOT/home/user1" in the file system and be locked in his or her own account and lower directories (so he or she can't access the rest of the file system such as other accounts), set the base path to "/FTPROOT/home/user1" and relative path to "/".
2. If you want the user "user2" to access his or her own account "/FTPROOT/home/user2" in the file system and the rest of the FTP accessible files (assuming that `FTPROOT` defines the root of FTP accessible files), set base path to "/FTPROOT" and the relative path to "/home/user2". With that scheme, this user session will start in its account, but he or she is authorized to go up and browse all FTP accessible files, like other accounts.
3. If you want the user "root" to access his or her own account "/FTPROOT/home/root" and ALL the other file system files, set the base path to "/" and the relative path to "FTPROOT/home/root". With that scheme, this user session will start on its account, but he or she is authorized to go up and browse all the file system, like other accounts, system configuration files, etc. This scheme can lead to a system security hazard and is thus not recommended.

## 3.06 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
```



## 3.06.01 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 eight tasks. Two tasks for  $\mu$ C/OS-II (idle and statistics), two for  $\mu$ C/TCP-IP, two for  $\mu$ C/FTP (server and control), the startup task and the LED task. It uses a maximum of four sockets, one for the server task, one for the control task and one for the data transfer task.

### app.c

This file contains the application code for example #1. As with most C programs, code execution starts 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 *)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 FTP server */
    FTPs_Init(ip, NET_UTIL_HOST_TO_NET_16(20));
                                (8)
    APP_DEBUG_TRACE ("Creating Application Tasks.\n");
    AppTaskCreate();            /* Create application tasks */

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

                                (9)
    while (DEF_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 the 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 you 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/FTPs](#) 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 FTP 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  $\mu$ C/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  $\mu$ C/TCP-IP and defines the number of timers used in  $\mu$ C/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  $\mu$ C/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  $\mu$ C/OS-II book.

## 3.06.02 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.

### 3.06.03 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.

### 3.06.04 $\mu$ C/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.

### 3.06.05 $\mu$ C/DHCPc Source Files

The  $\mu$ C/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  $\mu$ C/DHCPc package assumes infinite IP address leases only.

To use the  $\mu$ C/DHCPc package, the following files need to be included in your application:

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

### 3.06.06 $\mu$ C/FS Source Files

$\mu$ C/FTPc requires the presence of a file system to work properly. In fact,  $\mu$ C/FTPc assumes  $\mu$ C/FS as the file system and are tightly integrated with it. If you want to use  $\mu$ C/FTPc with another file system, you should read section 3.02 of this file to know what to change.

In order to use  $\mu$ C/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  $\mu$ C/FS. If we used a different media, we would include a different driver with the application.

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

### 3.06.07 $\mu$ C/LIB Source Files

$\mu$ C/DHCPc,  $\mu$ C/FTPc and  $\mu$ C/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 view) to always certify the same source code instead of different source from multiple compiler vendors. The libraries used by  $\mu$ C/DHCPc,  $\mu$ C/FTPc and  $\mu$ C/TCP-IP are found in the following directory :

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

### 3.06.08 $\mu$ C/OS-II Source Files

$\mu$ C/FTPc and  $\mu$ C/TCP-IP require the presence of a Real Time Operating System (RTOS). The sample application uses  $\mu$ C/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  $\mu$ C/OS-II port for this CPU is found in the directory shown above. In fact, the  $\mu$ C/OS-II ARM port is generic and can thus be used with other ARM CPUs.

### 3.06.09 $\mu$ C/TCP-IP Source Files

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

```
.\uC-TCP-IP\IF\*.*  
.\uC-TCP-IP\IF\Ether\*.*  
.\uC-TCP-IP\NIC\Ether\AT91RM9200\*.*  
.\uC-TCP-IP\OS\uCOS-II\*.*  
.\uC-TCP-IP\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, 2<sup>nd</sup> 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](mailto: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](http://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](mailto:Info@IAR.com)

WEB : [www.IAR.com](http://www.IAR.com)

### **Micrium**

949 Crestview Circle

Weston, FL 33327

USA

+1 954 217 2036

+1 954 217 2037 (FAX)

e-mail: [Jean.Labrosse@Micrium.com](mailto:Jean.Labrosse@Micrium.com)

WEB: [www.Micrium.com](http://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](mailto:Sales@ValidatedSoftware.com)

WEB: [www.ValidatedSoftware.com](http://www.ValidatedSoftware.com)