Embedding it better...

µTasker

µTasker Document

µTasker – User Files

Table of Contents

1. Introduction

The µTasker project includes a simple but highly efficient file system called µFileSystem which is primarily used to store web pages and similar data. These files can be transferred and viewed via FTP for easy management and updating, which avoids the need to compile data into a project.

In some situations there may however be advantages when one or more files can be included as a part of compiled code or even be constructed in SRAM. This document explains how this option can be used together with the FTP and HTTP server and gives examples of its use in some typical applications.

A further option of embedding the user files in a single uFileSystem file is then described together with the operation of the uTaskerFileCreater utility, which can be used to automate the generation of such a file or for creating an include file for a collection of user files.

2. Activating User File Support

To work with user files the define `FLASH_FILE_SYSTEM` must be activated in `config.h`. This enables the operation together with the µFileSystem, FTP and HTTP servers, whereby the user files have priority over files in the µFileSystem; if two files were to have the same name, the user file would be the one used instead of the one in the µFileSystem space. User files have read-only characteristics and so generally cannot be modified without their content being recompiled. In some cases user files may be constructed in SRAM and so can be freely modified during operation by application level code.

3. User File Lists

Per default there are no user files which can be accessed by the FTP and HTTP servers, so the application must enter these before they can be used. This has however an advantage in that they can be deactivated if desired or a different set entered during operation. For example, when there are no files in the µFileSystem the application can choose to enter a set of default web pages which exist in user code space. These can then enable the upload of new web pages to the µFileSystem, after which the uploaded files are used in preference. If required, file content can also be created in SRAM and accessed as a user file.

The following code shows two simple HTML user files created in program code space:

```
static const CHAR start_page[] =
    "<html><head><title>uTasker Test</title></head><body>This is a start test page
embedded in the code<br><a href=""0.htm"">Go to main start page</a><br><a
href=""link.htm"">Test another embedded page</a></body></html>";
static const CHAR link_page[]  = "This is another embedded page - <a
href=""0.htm"">Go to main start page</a>";

static const USER_FILE user_files[] = {
    {"link.htm", (unsigned char *)link_page, (sizeof(link_page) - 1),
                                        MIME_HTML, FILE_VISIBLE},,
    {"0HTTP", (unsigned char *)start_page, (sizeof(start_page) - 1),
                                        MIME_HTML, (FILE_VISIBLE | FILE_ADD_EXT)},
                                        // define start on web server contact
    {0}                                 // end of list
};
```

The `USER_FILE` list specifies a name for the file, its content type, its location in memory and its content size. Note that the content type doesn't need to match with the file name – this has some special advantages as seen later in this document. The file name `0HTTP` is a special name which will be accessed in preference by any contact with the web server without a defined file name; it corresponds to a user defined start web page. The files are marked as being visible files, which means that they will be displayed as read-only files via FTP. In the case of the user file with name `0HTTP`, which has no inherent extension, this will be displayed as `0HTTP.htm` via FTP.

This user file list is however only valid when the application also enters it by calling:

```
    fnEnterUserFiles((USER_FILE *)user_files);  // enter the user file list
```

If the list is to be revoked the call can be repeated with a null pointer as shown below:

```
    fnEnterUserFiles(0);                        // disable user file list
```

Alternate file lists (*either const lists or dynamic lists constructed in SRAM*) can be set at any time.

4. Interaction with the FTP server

Depending on a file's defined properties it can be visible via FTP or it can be hidden. See the AJAX examples later in the document for cases for using invisible files. Furthermore, files which have no inherent extension (eg. `0HTTP` in the example in the previous section) can be displayed with the defined MIME extension type. This means that the file `0HTTP`, accessed without an extension, can be nevertheless be displayed as `0HTTP.htm` via FTP.

All visible files are displayed as read-only files and can neither be deleted nor updated by FTP.

5. Interaction with the HTTP server

When a remote user establishes a connection with the HTTP server from a web browser, the initial access doesn't specify a file to be served but leaves the web server to decide which start page is to be displayed. When the user file list is not activated this start page will always be the file "`0.htm`" from the µFileSystem. Should this file not exist the 404 error page will be displayed.

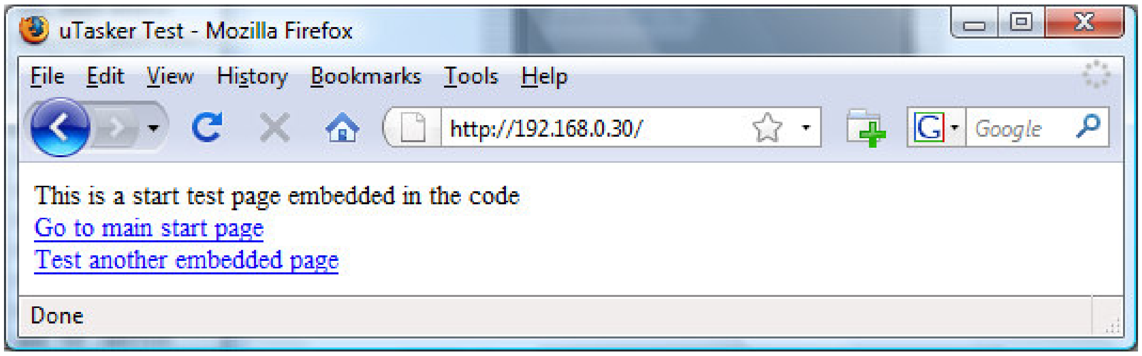
In order to specify that a certain file from the user file space is to be used it should be entered with a name "`0HTTP`" and mime type `MIME_HTML`. This will be used in preference to "`0.htm`" if it exists.

User files are referenced using their full names, including extension. For example "`link.htm`", as in the example in section 3. "`link.htm`" will take priority over a µFileSystem file with the name "`l.htm`". Since the user file list uses full names, the µFileSystem file "`l.htm`" will be accessed if the full name doesn't match completely. For example if "`L.htm`" or "`li.htm`" or "`lin.htm`" were to be referenced. User files are also case-insensitive and so "`link.htm`" is equivalent to "`LINK.htm`" etc.

Since the HTTP server works autonomously with files the user doesn't need to know whether the files being served are located in the µFileSystem memory, which can be in internal or external SPI based FLASH, or are located in user code (or user RAM) space. HTML type user files will still be parsed by the web server to allow adding or generating dynamic content if this is enabled in the project.

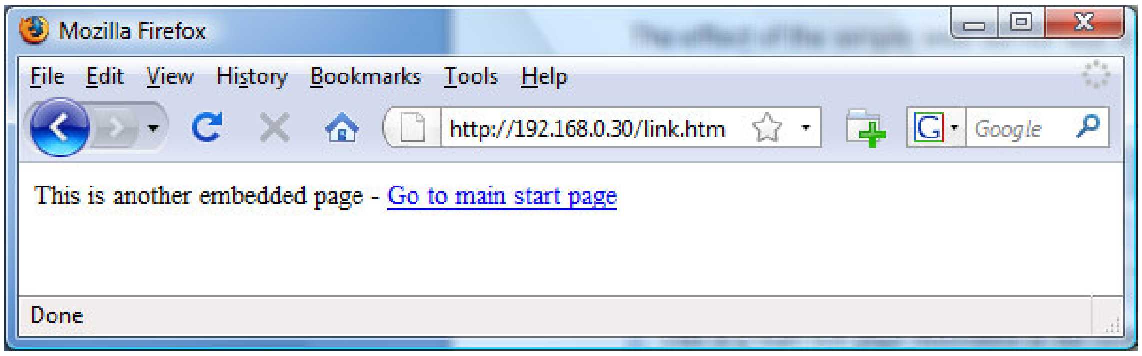
Code based user files operate correctly when working with the µTasker simulator, FTP and web server, enabling comfortable development and testing of code based and also dynamic RAM constructed web files.

The effect of the simple web server test is the following page being displayed when the web server is first contacted:


uTasker Test - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.0.30/
Google
Done

This is a start test page embedded in the code
Go to main start page
Test another embedded page

Clicking on the link "Test another embedded page" results in the second user file being served:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.0.30/link.htm
Google
Done

This is another embedded page - Go to main start page

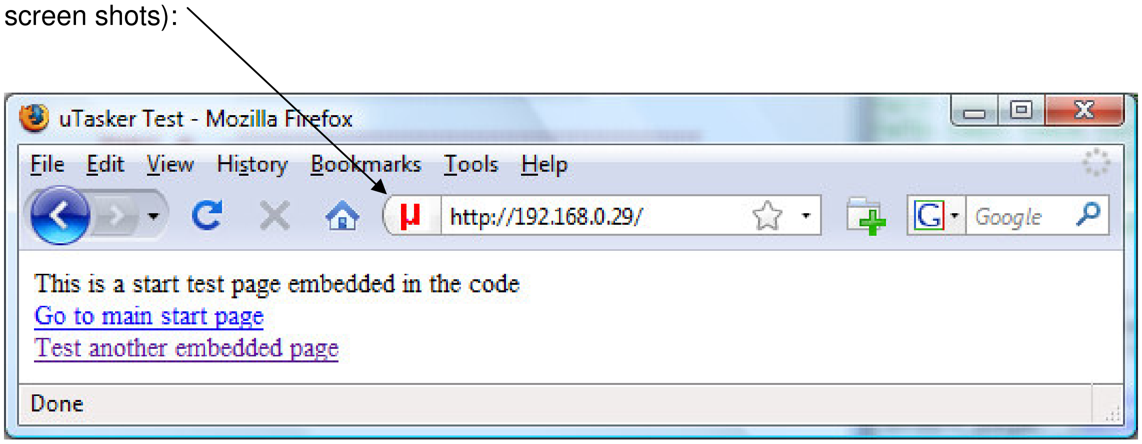
From this page the main start side can be navigated to, which is taken from the uFileSystem rather than the user file space.

6. Further Examples of the Use of User Files

6.1. Adding a Favicon

A `favicon.ico` is a small icon image which can be displayed at the start of the URL in the web browser. This simple image is a useful way of adding a personal touch to the web server.

A favicon can be seen in the following browser screen shot (compare with the previous screen shots):


uTasker Test - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.0.29/
Google
Done

This is a start test page embedded in the code
Go to main start page
Test another embedded page

The favicon is requested (by file name "`favicon.ico`") by most web browsers when connecting to a web server. Often this is only requested the first time that contact is made to a new web server and then cached, meaning that the cache may need to be cleared to force a new image to be updated by the web server.

Adding a favicon is extremely easy and requires one additional entry in the user file table:

```
..
{"favicon.ico", (unsigned char *)uTaskerfavicon, sizeof(uTaskerfavicon),
                                                   MIME_ICON, FILE_VISIBLE},
..
```

For completeness, the following shows the µTasker favicon image content, added as an array:

```
static const unsigned char uTaskerfavicon[] = {
    0x00, 0x00, 0x01, 0x00, 0x01, 0x00, 0x10, 0x10,   0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x28, 0x01,
    0x00, 0x00, 0x16, 0x00, 0x00, 0x00, 0x28, 0x00,   0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x20, 0x00,
    0x00, 0x00, 0x01, 0x00, 0x04, 0x00, 0x00, 0x00,   0x00, 0x00, 0xc0, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00, 0x80,   0x00, 0x00, 0x00, 0x80, 0x80, 0x00, 0x80, 0x00,
    0x00, 0x00, 0x80, 0x00, 0x80, 0x00, 0x80, 0x80,   0x00, 0x00, 0x80, 0x80, 0x80, 0x00, 0xc0, 0xc0,
    0xc0, 0x00, 0x00, 0x00, 0xff, 0x00, 0x00, 0xff,   0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0xff, 0x00,
    0x00, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0xff,   0x00, 0x00, 0xff, 0xff, 0xff, 0x00, 0xf9, 0x99,
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xf9, 0x99,   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xf9, 0x99,
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xf9, 0x99,   0x89, 0x98, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,
    0x99, 0x99, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,   0xdf, 0xf9, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,
    0xff, 0xf8, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,   0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,
    0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,   0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,
    0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,   0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,
    0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xf9, 0x99,   0xff, 0xff, 0x99, 0x9f, 0xff, 0xff, 0xff, 0xff,
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,   0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,   0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

This array was generated from a suitable icon image by using the conversion utility "`uTaskerFileCreate`" in the µTasker "`Tools`" project directory.

6.2. Handling AJAX

AJAX is a client side technology which works together with JAVA script. One of its main advantages is that it allows web server content to be updated in a web browser without the complete page needing to be reloaded, for example a single value being continuously updates without the need to transfer and re-draw the complete page content. There is a simple introduction to using AJAX on the µTasker forum: http://www.utasker.com/forum/index.php?topic=159.0

For the web server this involves responding to `GET` requests of the type `XML?value_x`, which represents a request of, for example, the variable `value_x`. The web server simply serves the variable in the format "`<value_x>£vX4</value_x>`" so that the web browser can update its display locally.

Although not the only method, user files can be used as a simple was of interpreting such `GET` requests to return the correct page content.

```
..
    {"XML?value_x", (unsigned char *)ajax1, (sizeof(ajax1) - 1),
                                                         MIME_HTML, FILE_INVISIBLE},
..
```

This causes the specific GET request to directly serve a predefined file content, such as that specified by

```
static const CHAR ajax1[] = "<value_x>£vX4</value_x>";
```

Note that the file is declared to be invisible (because it is not actually a file in the usual context) and so will not be listed via FTP. The file name has no extension since it is used to reference a particular XML request, but it has been given HTM type characteristics. This is very useful since the web server, when serving this simple file content, will parse it as any other HTM type file. The standard web content interface can then be used to insert the content referenced by `£vX4` as is usual for the µTasker content inserter. This enables highly flexible content to be inserted also in XML responses using a simple XML interpreter technique. *The following µTasker forum topic discusses inserting string content* http://www.utasker.com/forum/index.php?topic=94.0

6.3. RAM Based File Content

Both the user file table and its content can be situated in code FLASH, data FLASH or RAM. When RAM is used, the table and/or the content can be dynamically modified, which allows complete web page content to be changed by the application code to suit instantaneous requirements.

If the user file table is maintained in RAM, the file content type can also be modified dynamically, allowing it to be parsed (*with HTM type and type with lower value*) for additional content insertion, or excluded from parsing (*with any other type with a higher value*).

The user file table can also be exchanged at any time for an alternative set (eg. when the system is preparing new dynamic content which shouldn't be disturbed), by recalling the `fnEnterUserFiles()` routine.

One useful advantage of using a RAM based file is that its contents can be frequently updated without any issues of FLASH wear-out.

7. Embedded User Files and the Utility uTaskerFileCreate

The utility uTaskerFileCreate is delivered with the µTasker project and is contained in the directory `\Tools`.

The simplest use of the utility is to generate a C-code array from any other file, which may be an image file or HTM file, etc, as was shown in section 6 to generate code from a `favicon.ico`. The syntax to perform this is

```
      uTaskerFileCreate favicon.ico favicon.c uTaskerfavicon
```

where the first argument is the input file, the second the output file (containing the C-code) and the third an optional array name (if nothing is specified it defaults to array[]).

To check the version of the utility it can be called with the version argument:

```
      uTaskerFileCreate –v
```

A further, and more powerful use of the utility, is to generate a collection of files to form a user *file system collection*. This allows automation of the generation of the content of each file as well as the user table. The syntax for performing this is:

```
      uTaskerFileCreate –f user_files.txt 1UserFiles.bin
```

where the `–f` informs the program that it is receiving a file containing instructions to perform the job. This file is, in this example, called `user_files.txt`. The final argument is the file name of a binary image to be produced. In this case it is defined to be `1UserFiles.bin` (a bin file) but this will also cause a second file to be generated called `1UserFiles.c`. This second one contains complete C-code for the user file system and its individual file contents. The binary output will be discussed after first looking at the C-file created by the specific command file.

7.1. Complete User File System in a single C-File

The following shows the contents of a command file to generate an output file containing a number of individual files with differing properties:

```
// This file is used as input to which files are added to the packed user file
table (avoid tab use)

#define FILE_HEADER_LEN        5         // this must match with uFileSystem
setting
#define MAX_FILE_LENGTH        4         // size in bytes
#define BIG_ENDIAN             1         // target is big-endian - set 0 for little
#define ALIGN                  1         // align the table to allow direct
accesses to long pointers - set 0 if not important

#define FILE_VISIBLE           0         // defines for use when generating binary
content - use only decimal input
#define FILE_INVISIBLE         1
#define FILE_ADD_EXT           2

#define MIME_HTML              0
#define MIME_JPG               1
#define MIME_GIF               2
#define MIME_CSS               3
#define MIME_JAVA_SCRIPT       4
#define MIME_BINARY            5
#define MIME_TXT               6
#define MIME_ICON              7
#define MIME_BMP               8

%19800                        // location (hex) in uFileSystem (eg. corresponding
to '1')

0Menu.htm -w -T=MIME_HTML     // remove additional white space to minimise size for
all html files
1Lan.htm  -w -T=MIME_HTML
3I_O.htm  -w -T=MIME_HTML
BLogo.gif    -T=MIME_GIF      // no white space removal defined for images
9Generate.bin -c=FILE_INVISIBLE -T=MIME_HTML // this file will be invisible since
it is used for dynamic content control only
5admin.htm -w                 // when no type is specified the default is HTML
7help.htm -w
8serial.htm -w
AStats.htm -w
Dback.jpg    -T=MIME_JPG      // no white space removal defined for images
favicon.ico  -T=MIME_ICO      // favicon from the chip manufacturer (or project
specific)
// end
```

The relevant file names for the generation of the C-file output are shown highlighted above and results in each of the files being converted to its own array with the name `_file` (eg. `0Menu.htm` will result in the array `_0Menu[]`). For html files the `-w` argument can be used to cause unnecessary white space in the original file to also be deleted in order to save unnecessary array space. All of these individual arrays are collected together in the output C-file.

In addition to the arrays, the user file table `user_files` is also automatically generated. The argument `-T` is used to specify the mime-type of each file and the argument `-c` is used to specify special properties (characteristics). These entries were discussed in detail in earlier sections of this document.

The C-content looks like this (data content and several arrays removed):

```
static const unsigned char _0Menu[] = {
0x3c,0x68,0x74,0x6d,0x6c,0x3e,0x3c,0x68,0x65,0x61,0x64,0x3e,0x3c,0x6d,0x65,0x74,0x61,0x20,0x68,0x74,0x74,0x
70,0x2d,0x65,0x71,0x75,0x69,0x76,0x3d,0x22,0xa3,0x76,
.......
};
static const unsigned char _1Lan[] = {
0x3c,0x68,0x74,0x6d,0x6c,0x3e,0x3c,0x68,0x65,0x61,0x64,0x3e,0x3c,0x6d,0x65,0x74,0x61,0x20,0x68,0x74,0x74,0x
70,0x2d,0x65,0x71,0x75,0x69,0x76,0x3d,0x22,0x63,0x6f,
.......
};
static const unsigned char _3I_O[] = {
0x3c,0x68,0x74,0x6d,0x6c,0x3e,0x3c,0x68,0x65,0x61,0x64,0x3e,0x3c,0x6d,0x65,0x74,0x61,0x20,0x68,0x74,0x74,0x
70,0x2d,0x65,0x71,0x75,0x69,0x76,0x3d,0x22,0x63,0x6f,
.......
};

.....

static const USER_FILE user_files[] = {
{"0Menu.htm", (unsigned char *)_0Menu, sizeof(_0Menu), MIME_HTML, FILE_VISIBLE},
{"1Lan.htm", (unsigned char *)_1Lan, sizeof(_1Lan), MIME_HTML, FILE_VISIBLE},
{"3I_O.htm", (unsigned char *)_3I_O, sizeof(_3I_O), MIME_HTML, FILE_VISIBLE},
{"BLogo.gif", (unsigned char *)_BLogo, sizeof(_BLogo), MIME_GIF, FILE_VISIBLE},
{"9Generate.bin", (unsigned char *)_9Generate, sizeof(_9Generate), MIME_HTML, FILE_INVISIBLE},
{"5admin.htm", (unsigned char *)_5admin, sizeof(_5admin), MIME_HTML, FILE_VISIBLE},
{"7help.htm", (unsigned char *)_7help, sizeof(_7help), MIME_HTML, FILE_VISIBLE},
{"8serial.htm", (unsigned char *)_8serial, sizeof(_8serial), MIME_HTML, FILE_VISIBLE},
{"AStats.htm", (unsigned char *)_AStats, sizeof(_AStats), MIME_HTML, FILE_VISIBLE},
{"Dback.jpg", (unsigned char *)_Dback, sizeof(_Dback), MIME_JPG, FILE_VISIBLE},
{"favicon.ico", (unsigned char *)_favicon, sizeof(_favicon), MIME_ICO, FILE_VISIBLE},
{0} // end of list
};
```

The file can be linked into a project and used as user file content by simply entering its `user_file` with

```
fnEnterUserFiles((USER_FILE *)user_files);  // enter the user file list
```

7.2. User File Collection Embedded in a µFileSystem File

For further flexibility, the same user files as linked in to the project using the generated C-file can by transferring as binary output file to the µFileSystem (using FTP or HTTP post, etc.). For this to be possible the define `EMBEDDED_USER_FILES` should be active in `config.h` as well as the `INTERNAL_USER_FILES` define.

The extra information in the control file is used to generate the content of the binary file so that it also contains the complete data, including the *user file table* in a form compatible with the project and used compiler.

- First, the defines should match with project defines so that the generated table values are correct.
- Secondly, the value for the physical address location of the µFileSystem file is specified by the hexadecimal value `%19800`. This is the address of the file '1' (*also corresponding to the generated binary file name according to µFileSystem rules*) in a particular project. Thus, by copying the file `1UserFiles.bin` by FTP it will be saved starting at the address `0x19800` as an embedded user file collection, with the possibility of deleting and updating without the need to recompile and reload the complete project code.

7.3. Activating an Embedded User File Collection

Before the *embedded user file collection* is activated it is important that its validity is checked. The checking and activation is supported by the function

`extern USER_FILE *fnActivateEmbeddedUserFiles(CHAR *cFile, int iType);`

If the function can verify that the file contains a valid *user file table* it will return a pointer to the table, or else it will return 0. The following shows this in operation, whereby a code based user file will be entered as default if the embedded collection is not available at the specified location (the location being µFileSystem file '1')

```
    if (fnActivateEmbeddedUserFiles("1", USER_FILE_IN_INTERNAL_FLASH) == 0) {
                              // if valid embedded user file space is
                                 found activate it, else use code embedded version
        fnEnterUserFiles((USER_FILE *)user_files); // code based user_files
    }
```

Note that this needs to be repeated after loading a new *embedded user file collection* or else it is valid at the following restart. The same is true if the *embedded user file collection* should be deleted during operation.

It is very important to understand the effect of the parameter `USER_FILE_IN_INTERNAL_FLASH` since this may only be used when the µFileSystem file containing the embedded user file collection is addressable in memory. This allows the user file table to be access directly as if it were in code space.

When working with the µFileSystem in external memory, which cannot be addressed directly (like SPI FLASH) it is important to use the parameter `USER_FILE_IN_EXTERNAL_SPACE` instead. This then allows the operation with external memory accesses. To achieve this it does however need to first make a copy of the user table in local RAM (taken from the µMalloc heap). In addition, the file names (strings) are also copied from the embedded file content to local SRAM, again on µMalloc heap. This backup to SRAM is performed in the routine `fnActivateEmbeddedUserFiles()`.

7.4. Embedding Files without Extensions and using XML? Directives

When generating an embedded user file collection containing files which have no extension (eg. `0HTTP` rather than `0HTTP.htm`) it is necessary to instruct the parser that it doesn't need to search for a file extension. This is performed by using the symbol `|` as shown below:

```
0HTTP| -w -T=MIME_HTML
```

In order to include `XML?` directives (see the chapter about AJAX for more details about their use) the following technique can be used:

```
XML?Dummy| -c=FILE_INVISIBLE -w -T=MIME_HTML
```

The `XML?Dummy` input file cannot be saved on a PC with this name since it is an illegal name, therefore it should be stored as a file named `XML-Dummy` (without extension). The conversion utility will automatically look for a corresponding input file with this name when it encounters `XML?` and this results in the normal entry `XML?Dummy` being made in the user file table.

Conclusion

The user file system has been introduced with examples of its practical use in typical embedded projects. Since the user file table and its content can be located in either FLASH or RAM its use is extremely flexible in handling various requirements, including an AJAX XML interpreter interface.

User files can individually be given characteristics to allow them to be parsed by the web server, made visible or invisible to the FTP server.

The utility `uTaskerFileCreate` has been introduced to show how it can be used to generate C-code arrays of either single files (images, HTM etc.) or for automatically generating complete C-code for the entire user file contents, for simple linking into a project.

Finally, the additional flexibility offered by embedding a user file collection into a single µFileSystem file has been demonstrated, allowing deleting and updates of the complete user files. This embedded user file collection can reside either in internal or external SPI FLASH.

Modifications:

V0.00 8.4.2009: Initial draft – work in progress.

V0.01 27.5.2009: FTP and additional use examples. Not officially released.

V0.02 26.6.2009: Add embedded user files and uTaskerFileCreate.exe.

V0.03 29.9.2010: Add new uTaskerFileCreate.exe options to allow files without extensions and embedded XML? commands