## Introduction

The µTasker web server interface allows the user to control various HTML elements such as strings to represent values in the system or to display whether an option is set or not. This makes it simple to use the web server and HTML code to interact with the embedded software.

In some circumstances it is not the HTML content which defines the operation but rather the embedded system that should define HTML content itself or allow the HTTP connection to be used to transfer other data.

This document discusses the ***Dynamic Content Generation*** features of the µTasker TCP/IP stack and gives some examples to show how the user can utilise them in order to produce powerful effects or efficient advanced controls.

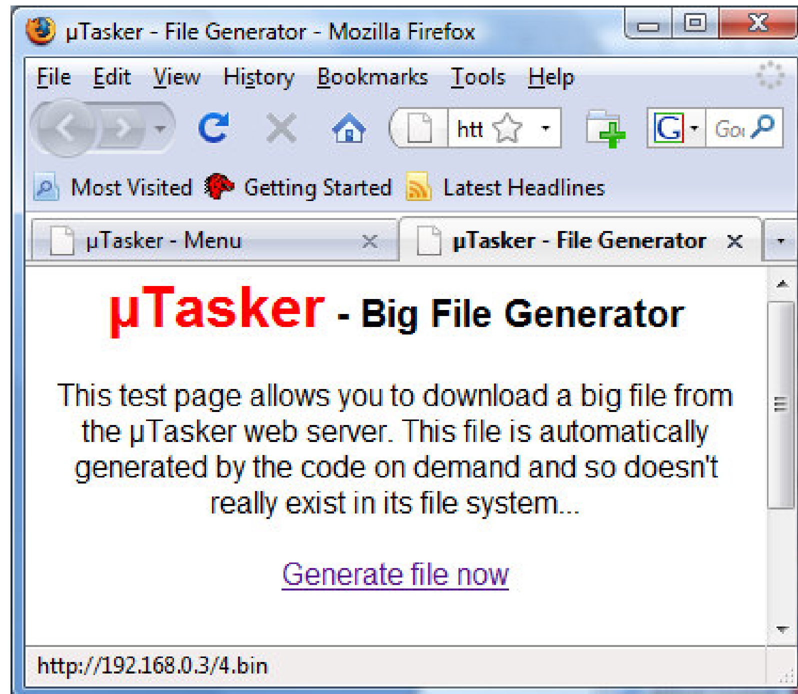
In order to use the dynamic content generation in the µTasker project ensure that the define `HTTP_DYNAMIC_CONTENT` is active as well as the basic web server with `USE_HTTP and WEB_PARAMETER_GENERATION`.

## File content generation

The HTTP interface is not only useful for display and control tasks but is often used to upload and download data. The data can be file content and this example show how the user can generate file content, which can either be the contents of internal or external memory (such as external storage data) or be formatted content which is created on demand.

This example shows how large binary content is "generated" on demand to be sent to the Browser Client as data file.

Starting with a simple web page a link has been set to another internal file which is displayed as http://192.168.0.3/4.bin when the user positions the mouse over it (see the status line display below).




The HTML content to do this is standard - `<a href="4.bin">Generate file now</a>`

The file *4.bin* doesn't however actually exist. It is the file which we want to generate when the user clicks on it which will then be downloaded as a binary file. We have in reality loaded a file called *4.htm* to the file system – this will be opened by the link since the uFileSystem doesn't use the 'bin' extension directly. That means, it doesn't try to open the file 4.bin but instead will finds that '4.htm' exists (assuming really loaded) and this will be served.

This file contains only the following (*yes, the file is only 4 bytes in size!!*):

```
£H10
```

Since the file is however a HTM type it is not server directly but parsed by the HTTP server and results in the following sequence being interpreted:

```
#define WEB_PARSER_START          '£'
#define WEB_INSERT_DYNAMIC        'H'
```

`static CHAR *fnInsertString(unsigned char *ptrBuffer, LENGTH_CHUNK_COUNT TxLength, unsigned short *usLengthToSend, HTTP *http_session)` is called with `TxLength` = 1 and the user's handler can interpret the dynamic content type '1' to generate the file as shown below.

```
if (*ptrBuffer == '1') {                // generate large file content
    static unsigned char ucTable[256]; // table to be sent in each chunk
    int x = 0;
    http_session->ucDynamicFlags |= GENERATING_DYNAMIC_BINARY;
    // ensure the HTTP server knows that we are generating binary content
    if (TxLength > ((8*1024*1024)/256)) { // after generating 8 Meg stop
        return 0;
    }
    while (x < sizeof(ucTable)) {
        ucTable[x] = x;
        x++;
    }
    *usLengthToSend = sizeof(ucTable);
    if (TxLength == ((8*1024*1024)/256)) {// signal last chunk to ensure
                                            that no padding is added
        http_session->ucDynamicFlags |= LAST_DYNAMIC_CONTENT_DATA;
    }
    return (CHAR*)ucTable;
}
```

The above code results is a binary file being downloaded by the web browser user which is filled with the binary sequence 0x00, 0x01, 0x02...0xff repeatedly and having a total length of 8 Mega Bytes! By changing the length (`8*1024*1024`) smaller or even much longer files can be generated.

A binary file is quite practical because the browser will automatically ask where it should be saved to. It also doesn't need any specific formatting. To save other types of file which do require formatting, the formatting can be added to the routine as discussed later – for example a WAV file has a simple header containing the coding details and the data length and this can be added the first time that the routine is called. By linking to a WAV file rather than a Bin file (`<a href="4.wav">Generate file now</a>`), the Browser will even open the file and play it back. Should the embedded project have previously recorded speech or music data and saved it to memory it can insert the recording rather than the generated data and it can thus simply be played via a Web browser using this technique.

**Detailed description of the code used to generate binary file content**

The user code to define the generation of the file is quite simple. All details about putting it into the HTTP TCP data stream and handling repetitions in case of lost frames, or limiting data when the receiver cannot accept the data rate is handled by the µTasker HTTP server. It doesn't require a transmit buffer to do this and can achieve fast throughput with minimum memory requirements.

The user must generate "*chunks*" of data. These are packets of data of a suitable size – the smaller the size the more often the routine will have to add them to the outgoing data stream but smaller sizes may often be more practical to generate. In the example 256 has been chosen (requiring a small **static** buffer called `ucTable[]` to hold the content stable between frames). Assuming 1400 byte TCP data content per transmitted frame (see the define `HTTP_BUFFER_LENGTH` in `app_hw_xxxx.h`) the user's routine would be called 5 times to fill the TCP buffer. In fact the routine will also be called a 6$^{th}$ time but the HTTP server will not be able to fit the 6$^{th}$ data block into the TCP frame. This means that a better choice for the chunk size in this case would be 280 bytes for efficiency (also the TCP frames will be filled out with 1400 bytes each and not 1280 bytes) but it would have complicated the task of generating the binary data pattern.

The user code doesn't actually know any details about the TCP frame. It just shouldn't try to create a single chunk larger that the TCP frame data size (1400 as default) since it would never fit!

Each time the user's code is called, the parameter TxLength is incremented. On the very first call it has the value 1 (never 0) and in this example it is used to know when to stop generating more data. In other cases it may be used to calculate the offset in memory to get the next data block, or to define other details about the content; for example when it is 1 it could add a fixed header to the data and afterwards add the data content. It may also stop the generation when the end of the memory content that it is collecting is reached rather than simply transferring a fixed number of chunks.

```
    http_session->ucDynamicFlags |= GENERATING_DYNAMIC_BINARY;
```

This is used only in the case of generating binary data content so that the HTTP server knows that it should not add any HTML content padding, which may otherwise corrupt the binary content. In this example the flag is set on every call but could also be set just once on the first call. The flag is always automatically cleared on the next HTTP session.

```
    if (TxLength > ((8*1024*1024)/256)) { // after generating 8 Meg stop
        return 0;
    }
```

When the user code returns 0 it signifies that there is no more data to be generated. The parser will then continue parsing the input file (4.htm in this case) and terminate the HTTP transfer if there is nothing more to do.

```
    *usLengthToSend = sizeof(ucTable);
```

When the user call provides a chunk of data to be added to the stream it must give the length of the chunk. In this example each chunk is of fixed length but it may vary. When transferring a file it is often necessary to send a smaller final chunk depending on the real file size.

```
    if (TxLength == ((8*1024*1024)/256)) {// signal last chunk to ensure
                                             that no padding is added
        http_session->ucDynamicFlags |= LAST_DYNAMIC_CONTENT_DATA;
    }
```

On the very last chunk of binary content it is useful to set the `LAST_DYNAMIC_CONTENT_DATA` flag. This informs the HTTP server that it should not pad the buffer, which it generally does for HTML content. This is not needed when generating HTML content since padding (some ASCII white spaces) is ignored by a Browser.

```
    return (CHAR*)ucTable;
```

The chunk must exist in a stable buffer (static) and the user's call returns a pointer to this buffer as return value when there is a chunk to be sent. It is not allowed to return a pointer to const data since the HTTP server will need to be able to modify the data content!

The user's function must respect the value of TxLength since it is the only reference to the part of the complete data that is to be returned as single chunk. It is normal that the HTTP server can call the routine with the same value of TxLength (for example when the last chunk couldn't be put into the last data frame) or with repeats of earlier values (when TCP data has to be regenerated due to frame loss in the network).

The data type used for TxLength is defined in types.h in th project directory.

```
typedef unsigned short    LENGTH_CHUNK_COUNT; // http string insertion and
                                    chunk counter for dynamic generation {}
```

This allows up to 64'000 chunks to be generated. If the amount needs to be even larger the typedef can be changed to unsigned long.

### HTML content generation

The user call can be used to generate its own HTML code. This is especially useful for generating things like variable size tables or drop down lists that adapt themselves to suit parameters and quantities.

The µTasker project contains an example of generating a multiplication table which will be described here in detail since it shows several features which are of particular interest. For example it shows how the table supports multiple users who not only request the table to be displayed but do this at the same time with different parameters! (*See later in the section for views of the actual web page*).

The web page for controlling the multiplication table can be found in the web page directory to the µTasker project. It is either a standard page or an alternative page (eg. In the M5223X project it is alternative page and replaces another page when it is tested – in `\Applications\uTaskerV1.3\WebPages\WebPagesM5223X\AlternativePages\AMulTable.htm`)

The relevant code in the html file looks like this:

```
<form action=AMulTable.htm name=r>
Columns (1..12) <input maxLength=2 size=2 name=H1
value="£vH1"><br>
Rows (1..1000) <input maxLength=4 size=4 name=H2
value="£vH2"><br><br>
<input type=submit value="Generate" name=r></form>
£vH0
<table border="3" style="background-color:silver">£H00</table>
```

This allows two fields to be modified to set the number of columns and rows and the table to be generated by pressing the “Generate” button.

Serving this page consists of two steps:

- first the table size parameters are passed by the GET function started when the button is pressed so that the dimensions are known for the particular session (don’t forget that more than one person may be doing this at the same time – up to the maximum number of HTTP sessions supported). The following is a typical GET command visible in the URL when executed

```
http://192.168.0.3/AMulTable.htm?H1=10&H2=12&r=Generate
```

This is requesting a 10 x 12 multiplication table (H1 and H2) and are handled in `fnHandleWeb()` in `webInterface.c` as follows.

```
case 'H':
{
    MULTIPLICATION_TABLE *ptrMulTable;
    if (http_session->ptrUserData == 0) {// if no user data space belonging
                                            to this session create it
        http_session->ptrUserData = uMalloc(sizeof(MULTIPLICATION_TABLE));
    }
    ptrMulTable = (MULTIPLICATION_TABLE *)http_session->ptrUserData;
    if (*ptrData == '1') {
        ptrMulTable->ucColumns = (unsigned char)fnDecStrHex(ptrData + 2);
        if (ptrMulTable->ucColumns < 1) {
            ptrMulTable->ucColumns = 1;
        }
        else if (ptrMulTable->ucColumns > 12) {
            ptrMulTable->ucColumns = 12;
        }
    }
    else if (*ptrData == '2') {
        ptrMulTable->usRows = (unsigned short)fnDecStrHex(ptrData + 2);
        if (ptrMulTable->usRows < 1) {
            ptrMulTable->usRows = 1;
        }
        else if (ptrMulTable->usRows > 1000) {
            ptrMulTable->usRows = 1000;
        }
    }
}
break;
```

The first thing to note is that each session uses an application specific pointer pointing to a struct which contains its table parameters. If the pointer is zero (which will be the case the first time that the session is used) it will obtain memory using uMalloc().

```
typedef struct stMULTIPLICATION_TABLE    // structure used by demo
{
    unsigned short usRows;
    unsigned char  ucColumns;
} MULTIPLICATION_TABLE;
```

Then the parameters are checked for validity (too large values are restricted) and copied to the struct for use later on with this session. The session is a single HTTP TCP connection which exists only until the web page has been completely served. If two users were to perform the action at the same time, each user session would have its individual table parameters stored individually.

- The second step is the serving the web page's contents, which then parses the html content where it finds 4 tags:

```
£vH1 handled by fnInsertString() to add the columns string value
£vH2 handled by fnInsertString() to add the rows string value
£vH0 handled by fnInsertString() to add instructions
£H00 dynamic content generation command (discussed later)
```

The string insertion is pretty much standard stuff. The only interesting thing is the way that the instruction text is chosen since this depends on whether the user has just established contact with the web page or has just pressed the generate button. We can easily distinguish between the two cases based on whether the session struct has values (rather than no

pointer or zero values), so we either instruct with "**Change values and click generate!!**" and enter default table dimensions of 10 x 10 or else we write "**Here is the requested multiplication table**:"

```
case 'H':
{
    MULTIPLICATION_TABLE *ptrMulTable;
    unsigned short usRows = 10;
    unsigned char  ucCols = 10;     // default start values
    int iSessionValid = 0;

    if (http_session->ptrUserData != 0) {
        ptrMulTable = (MULTIPLICATION_TABLE *)http_session->ptrUserData;
        if (ptrMulTable->ucColumns != 0) {  // if values have just been set
valid
            ucCols = ptrMulTable->ucColumns;
            usRows = ptrMulTable->usRows;
            iSessionValid = 1;
        }
    }
    switch (*ptrBuffer) {
    case '0':
        if (iSessionValid != 0) {
            cPtr = (uStrcpy(cValue, "<br><br>Here is the requested
multiplication table:<br><br>") - 1);
        }
        else {
            cPtr = (uStrcpy(cValue, "<br><br>Change values and click
generate!!<br><br>") - 1);
        }
        break;

    case '1':                                           // columns
        cPtr = fnDebugDec(ucCols, 0, cValue);
        break;

    case '2':                                           // rows
        cPtr = fnDebugDec(usRows, 0, cValue);
        break;

    default:
        return 0;
    }
    *usLengthToSend = (cPtr - cValue);
}
break;
```

The more interesting part is how the dynamic content generation support is used to generate a table and values, so here it is:

```
if (ptrBuffer == 0) {  // special case when session is terminating. This is
used to clear any session specific data and only occurs when there is a
valid user data pointer
    uMemset(http_session->ptrUserData, 0, sizeof(MULTIPLICATION_TABLE));
                                                        // clear content
    return 0;
}
```

Comment: When the TCP connection is closed, which happens after the complete page has been served, the session data has to be deleted otherwise it would not be possible to distinguish whether the next session on a particular TCP socket is a new connection or not. By deleting the session's struct this ensures that this is possible. *This call with zero pointer from the HTTP server only takes place when the session uses the* `http_session->ptrUserData` *pointer*.

```
if (*ptrBuffer == '0') {                // The multiplication function
    MULTIPLICATION_TABLE *ptrMulTable;
    LENGTH_CHUNK_COUNT Chunk = (TxLength - 1);
    LENGTH_CHUNK_COUNT X, Y;

    if (http_session->ptrUserData == 0) {
        http_session->ucDynamicFlags = NO_DYNAMIC_CONTENT_TO_ADD; // inform
                          that we don't want to generate anything this time
        *usLengthToSend = 0;
        return cValue;    // user session unknown so don't generate
                      anything. This occurs when the window is first opened
    }
```

Comment: This catches the first contact with a new session and stops the table being generated until the user has actually pressed the generation button.

```
ptrMulTable = (MULTIPLICATION_TABLE *)http_session->ptrUserData;

if (TxLength > (LENGTH_CHUNK_COUNT)(ptrMulTable->ucColumns * ptrMulTable-
>usRows)) {
    if (ptrMulTable->ucColumns == 0) {      // no valid input data
        http_session->ucDynamicFlags = NO_DYNAMIC_CONTENT_TO_ADD; // inform
                          that we don't want to generate anything this time
        *usLengthToSend = 0;
        return cValue;
    }
    return 0;                    // complete table content generated
}
```

Comment: The complete table has been generated so this terminates the present section. The web server will then continue and add the remains of the HTML file which caused the generation to start.

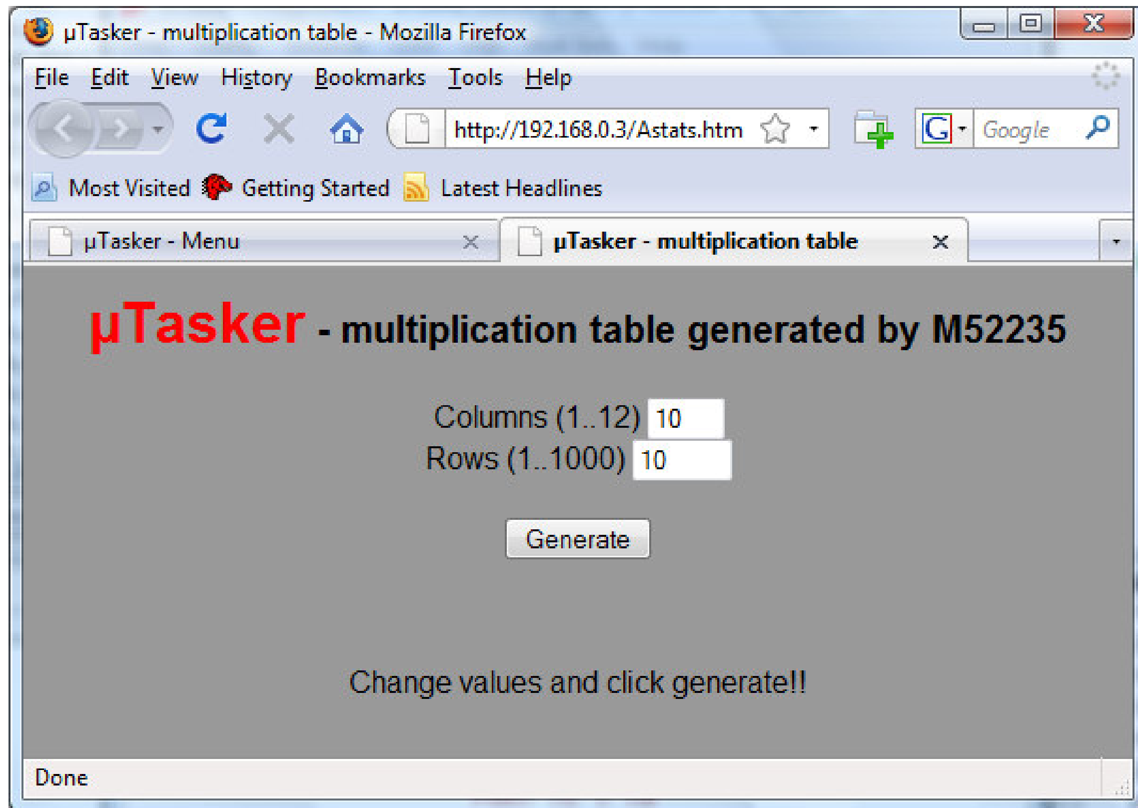
```
X = (Chunk%ptrMulTable->ucColumns);
Y = ((Chunk/ptrMulTable->ucColumns) + 1);
if (X == 0) {
    if (Y == 1) {
        cPtr = (uStrcpy(cValue, "<tr style=""background-color:white""><th
width=""60"">") - 1); // start a new table row
```

```
    }
    else {
        cPtr = (uStrcpy(cValue, "<tr><th style=""background-color:white""
width=""60"">") - 1); // start a new table row
    }
}
else {
    cPtr = (uStrcpy(cValue, "<th width=""60"">") - 1); // start a new
                                                           column
    Y *= (X + 1);                           // the multiplication result
}
cPtr = fnDebugDec(Y, (WITH_TERMINATOR), cPtr); // insert the result
if (X == ptrMulTable->ucColumns) {          // end of this row
    cPtr = uStrcpy(cPtr, "</tr></th>");     // close table row
}
else {
    cPtr = uStrcpy(cPtr, "</th>");          // close a single column
}
*usLengthToSend = (cPtr - cValue - 1);      // length of code to insert
}
```

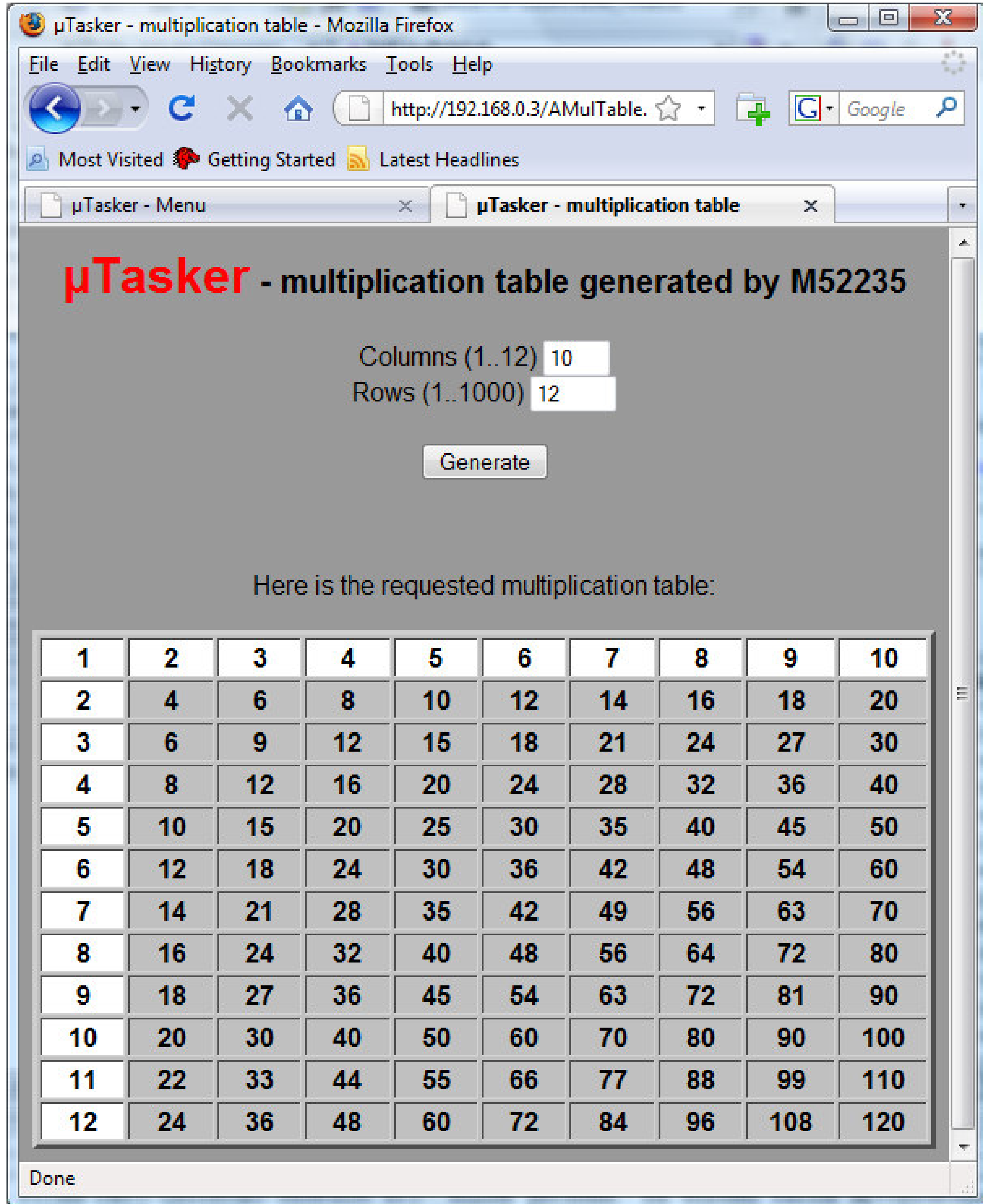
The further routine content is adding HTML code as strings to build up a HTML table of the defined number of rows and columns. It is colouring some of the table elements using a background colour to make it look nicer and adding the row and column multiplication results. This is quite standard software which may need a bit of debugging to get absolutely right but the result is simply chunks of a HTML table – strings in fact – which are simply generated one after another when the routine is called with incrementing TxLength value. In this case a chunk was chosen to be a single row since this is easy to manage. Again it is to be noted that the value of TxLength may be repeated or it may also go back to a lower value during the process (due to chunks that don't fit into the present TCP frame or repeats due to lost transmissions). The user code doesn't need to know no such details but is just required to generate each chunk that is demanded.

It should be clear that the user code is generally small and easy to write because the only goal is to generate the correct chunk corresponding to the Txlength value, which starts at 1 and increments to whatever limit the code sets. The result is however very powerful – not only can interesting and useful effects be generated but such functions are quite easily possible supporting multiple users with different parameters are the same time!!

So what does the result look like when the code runs?




Initial contact with the web page, displaying default values and instructions as to what to do




After changing the Rows value and clicking on the Generate button.

**Conclusion**

This document has introduced the Dynamic Content Generation support in the µTasker web server. It has shown two typical examples of its use to generate a data file for downloading via Web Browser and for generating program controlled HTML content. Both examples show how the user code can concentrate on its task of generating data 'chunks' and leaving the web server to perform the tasks of ensuring correct delivery of the resulting HTTP data session.

User code is generally small and easy to maintain but benefits from inbuilt features allowing multiple users to generate individual web pages with different parameters even at the same time.

Modifications:
- V0.0 13.11.2008 First draft.