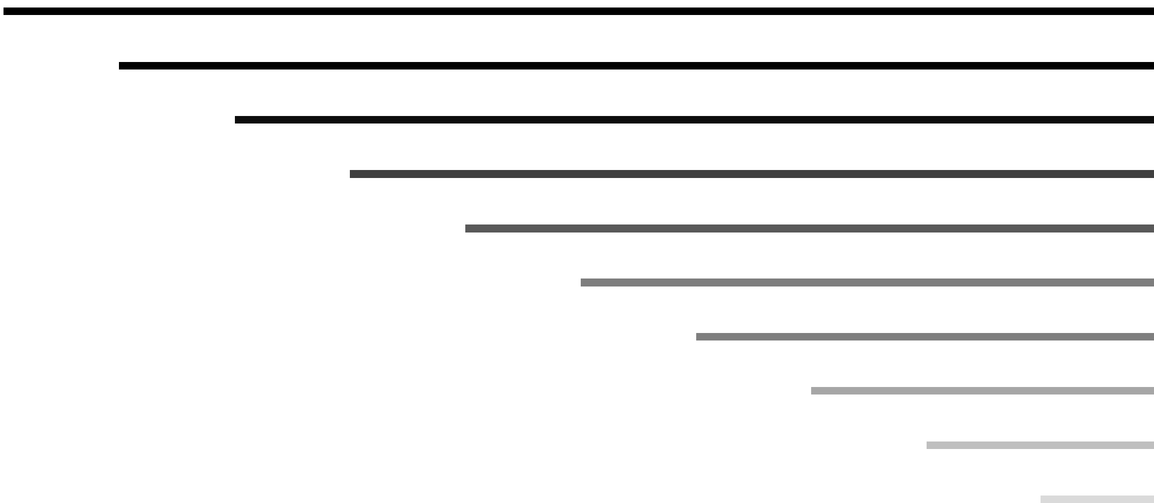
Embedding it better...

µTasker Document

µTasker User Guide – First Steps for New Users



Copyright © 2009 M.J.Butcher Consulting

Table of Contents

1. Introduction 3
2. Creating Your Own New Project 4
3. Customising the Simulator for Your New Project 5
4. Adding New Service Packs...7
5. Adding a New Task to Your Project 8
6. Second Example – The Task being configured by another Task....................................11
7. Third Example – Adding a Queue to Your Task and getting it to do a bit more Work....12
8. Adding some existing code to your project 15
9. Conclusion 17
Appendix A - µTasker Commands use in this Document 18

1. Introduction

The µTasker project is based on a demonstration project. This project is configured to work with the µTasker simulator and various compilers (IDEs) for its various supported processors. New users are advised to start by working through the µTasker tutorial delivered with the package in order to get to know the µTasker simulator, the basics of the demonstration project and generally working with the hardware.

Once the demonstration project has been used for first experimentation with the simulator and on the real target you should have a good feel for what it offers. In fact you should recognise that it is not just showing how to compile and load a simple project which flashes LEDs (although it does do that of course) but that it represents a project base which is suitable for a large amount of real and professional projects – it configures an ***Ethernet interface*** with ***TCP/IP*** services based on user parameters in the ***µParameterSystem*** based on swap blocks to ensure no data loss during power down sequences; it allows ***interactive web pages to*** be tested and modified in the ***µFileSystem*** (which can be either in internal FLASH or in external SPI based FLASH); it has an active ***watchdog*** monitoring correct operation and recovering from system failures; it has optional ***low power*** support using the low power wait/sleep modes of the processors under control of a low-power task; it supports a menu driven interface via ***UART, USB*** (if available in the chip) or ***TELNET*** showing alternative methods of monitoring and configuring; it even supports the ***encrypted upload*** of new firmware via a web browser interface (and USB). Basically it delivers the base for most real projects which can be simply adapted for project specific purposes - the user can add new application tasks and concentrate on this part of the job rather than having to redesign a complete system and lose precious project time.

This guide takes you through the process involved to use this base to make fast progress on integrating your own specific code, including how to add your project to the environment through to how to add your new tasks which will later run alongside any of the existing services which you would like to retain. You will find that the environment and its features can be easily configured and also hardware interfaces flexibly assigned. Your own code can be added and make use of the µTasker operating system support and your modules be constructed in a modular and easily understandable framework.

All of the subjects discussed in this document can be easily tested in the µTasker simulator and once the simple introduction, including clear examples, has been worked through there should be nothing standing in way between you and your first powerful project operating!

2. Creating Your Own New Project

The µTasker demo project is indeed ready to run and it can also be modified or extended with your own code. However there is a better way to start so that the original project remains intact in case of the need for a reference or to simplify service pack upgrades later which will add improvements and more features to the project environment.

First we will take a quick look at what the original demo project looks like when viewed in the file manager:

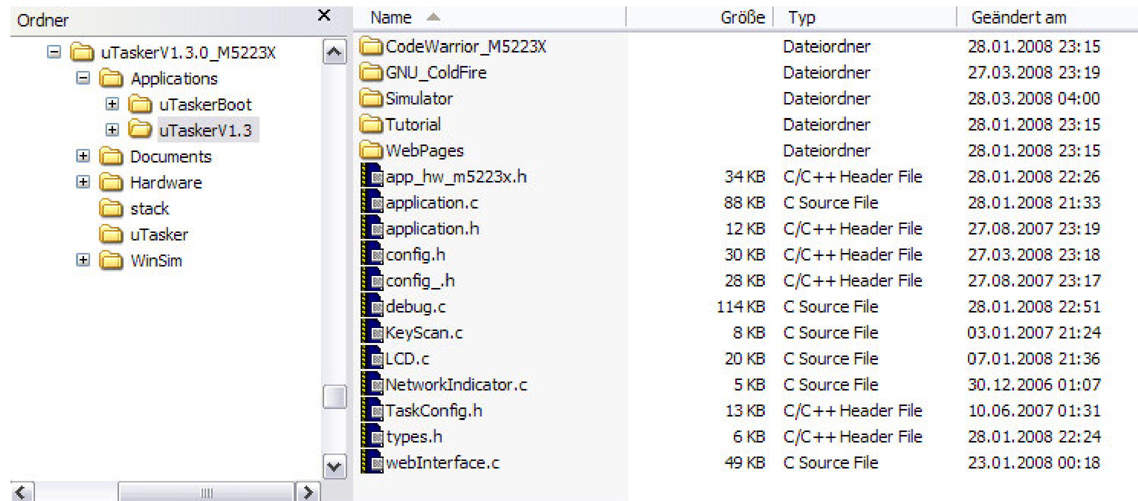


Figure 1. View of the µTasker project directory in the file manager

This is an example of a fresh project (for the Coldfire) showing that the `applications` directory in fact contains 2 projects – one is the uTaskerV1.3 demo application and the other is the uTasker boot loader project. Some packages will not have a boot loader project and some may even have more projects. The right hand side shows the contents of the uTaskerV1.3 application containing some application files and directories for the µTasker simulator and some compiler targets.

To add a new project, or several new projects as time goes by, start off by making a copy of the uTaskerV1.3 application. Rename it to suit your new project so that the directories now look like this:

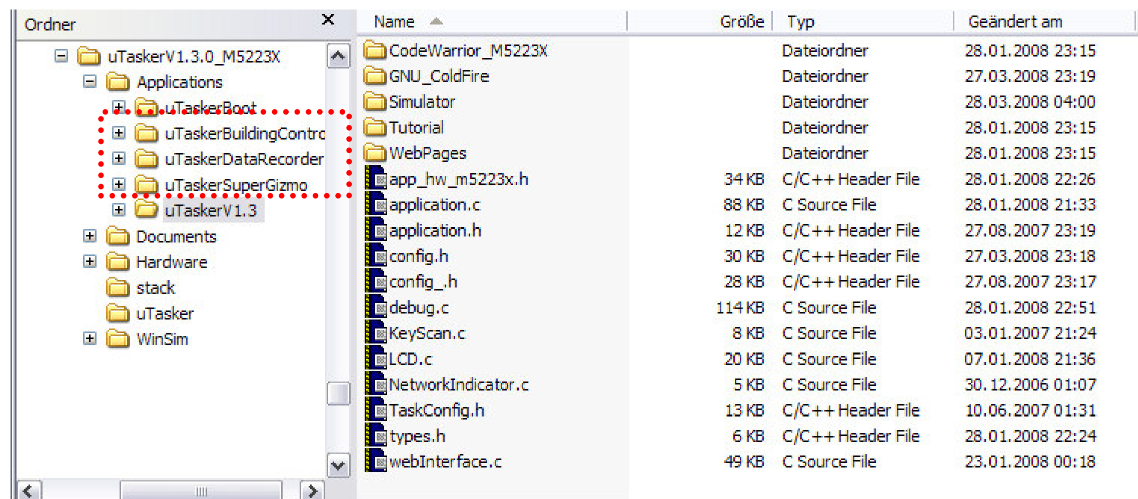


Figure 2. View after adding three new application project sub-directories

Here a further three applications have been added, each containing a copy of the content of the original uTaskerV1.3 application, which you can in each case use as base for your new project.

In fact it is possible to build the new project in exactly the same way as the uTaskerV1.3 project since all projects use purely referenced paths to locate their source files. But by editing the actual target projects with new project names you can then customise the outputs to generate targets corresponding to the actual project if you want.

One important thing to remember is that everything in the new application directory is project specific code. It can be changed as desired without affecting general project code (which can be shared between projects). You can set up `config.h` and `app_hw_XXXX.h` to suit the specific project and hardware. `types.h` can also be optimised for a project. The other application stuff can be deleted if not required, or renamed, or new ones can be added as needed.

3. Customising the Simulator for Your New Project

If you look at the contents of the simulator directory you will find something similar to this:

Name	Größe	Typ	Geändert am
cursor1.cur	1 KB	Cursor	09.03.2005 23:38
cursor2.cur	1 KB	Cursor	31.08.2005 12:37
cursor3.cur	1 KB	Cursor	16.05.2007 18:47
cursor4.cur	1 KB	Cursor	26.12.2007 15:41
keypad.bmp	237 KB	Bitmap	19.12.2006 02:42
NIC.ini	1 KB	Konfigurationseinst...	31.03.2008 21:34
resource.h	3 KB	C/C++ Header File	26.12.2007 15:41
SMALL.ICO	1 KB	Symbol	13.09.2005 18:24
StdAfx.cpp	1 KB	C++ Source File	25.12.2004 19:19
uTasker.aps	15 KB	aps File	24.01.2008 11:41
uTasker.ICO	2 KB	Symbol	13.09.2005 18:30
uTasker.rc	9 KB	Resource Script	24.01.2008 11:41
uTaskerV1-3.dsp	26 KB	VC++ 6 Project	31.03.2008 00:48
uTaskerV1-3.dsw	1 KB	VC++ 6 Workspace	29.12.2006 23:39
uTaskerV1-3.ncb	1'113 KB	Visual C++ IntelliSe...	01.04.2008 18:44
uTaskerV1-3.opt	60 KB	OPT-Datei	01.04.2008 18:44
uTaskerV1-3.plg	1 KB	HTML Document	31.03.2008 22:38

Figure 3. Simulator directory contents

This is how it looks with the original VS6.0 project but it may have some slightly different files if you have converted it to a newer format. Since your project is no longer **uTaskerV1-3** but maybe **uTaskerSuperGizmo** the project can be quite easily customised by renaming the files with the name uTaskerV1.3 to the new project name. Then it is necessary to open up the renamed files called `uTaskerSuperGizmo.dsw` and `uTaskerSuperGizmo.dsp` using a text editor (not VisualStudio!!) and do a search and replace of all occurrences of `uTaskerV1-3` with `uTaskerSuperGizmo`. You may see a warning in the files, something like "`# WARNING: DO NOT EDIT OR DELETE THIS WORKSPACE FILE!`" but this can be ignored – probably Microsoft will not be so pleased about this technique being used but it works very simply.

You can do something similar with your cross compiler hardware target but the details will probably depend a bit on exactly which one you are using.

Don't forget also that any changes to the source files (removed ones, renamed or added ones) will also have to be made in each of the target environments (VS simulator project, GNU make file, CodeWarrior IDE etc.). But this is nothing new and you are probable very familiar with how this is done anyway.

Do be sure that you edit and include paths in your cross compiler target. For example, you will usually find a configuration setting called something like `Preprocessor Options -> User Include Directories`. This may have a setting like `..\\..\\..\\Applications\\uTaskerV1.3`, which is informing the compiler environment to use this location as 'second' choice when searching for include files. Make sure that this is edited to specify that it should use the new target directory, eg. `..\\..\\..\\Applications\\uTaskerSuperGizmo` otherwise it may still use some header files from the original demo project, which can lead to some confusion!

4. Adding New Service Packs

Before starting a new project it is advisable to ensure that you have the latest service pack installed – installing new service packs during a project development is also possible but it may involve some checking in the application part to ensure that it is configured to be compatible and make the most of the new support.

Installing service packs is very easy. The following is a step-by-step guide starting with an initial installation and following with an upgrade to the latest service pack.

Note that only the latest service pack needs to be installed (*unless other instructions are given on the service pack web site*). If there are several service packs available, the newer ones always include all contents of the intermediate ones as well. All service packs are left available so that it is always possible to create an intermediate version if ever necessary.

1. The µTasker project is delivered as a zip file. Eg. `uTaskerV1.3_LPC.zip`. This is first placed in the directory where all work with the µTasker is going to be performed.

2. Extract the contents by unzipping the file. This requires the password which was delivered with the file. This will create a file called eg. `uTaskerV1.3_LPC` containing the complete µTasker project

3. When you have a new service pack it is also delivered as a zip file. Eg. `uTaskerV1.3_LPC_SP3.zip`. To install this, place it at the location of the µTasker project (eg. `uTaskerV1.3_LPC`) and ensure that the original zip file is not longer there (or has been renamed). Then rename the service pack zip file to match the original zip file name (eg. rename `uTaskerV1.3_LPC_SP3.zip` to `uTaskerV1.3_LPC.zip`

4. Now extract the contents of this zip file using the same password as for the original project.
   You will be asked whether certain files should be overwritten. Reply always with "Yes, overwrite all".

Once the extraction has completed, the complete project contents will have been updated with the service pack. New files will have overwritten older files which required updates but others will not have been modified.

Note also that the release notes should be checked for special instructions as to how configurations may have to be changed to use the new features and ensure full code compatibility. For each Service Pack release there is a thread in the µTasker forum with some specific details and a list of any known problems and patches – visit www.uTasker.com/Forum/ to check out these details for your particular target and service pack.

5. Adding a New Task to Your Project

The first thing that you will probably want to do is to add a new task of your own to the project. Therefore we will first add a new C-source file to the project with the following content.

```
extern void fnMyFirstTask(TTASKTABLE *ptrTaskTable)
{
    static unsigned char ucCounter = 0;

    fnDebugMsg("Hello, World! Test number ");
    fnDebugDec(ucCounter++, 0, 0);
    fnDebugMsg("\r\n");
}
```

This is your new task which will write the famous "**Hello, World**!" with a counter to the debug output. This requires an active UART to be available (or else a TELNET connection) but we will assume that a UART output is available – ensure that the define `SERIAL_INTERFACE` is active in `config.h` and the required UART is configured in `app_hw_xxxx.h`

```
(eg. #define DEMO_UART    1  // use UART 1 (the UART connector on the board) )
```

If you are testing this in the µTasker simulator (which is recommended since it is much easier and more efficient), ensure that this UART is also mapped to one of the PC's available COM ports in `app_hw_xxxx.h` – eg:

```
#define SERIAL_PORT_0   '3' // if we open UART channel 0 we simulate using com3 on the PC
```

The project will now compile but, if you run it, you will discover that this task never actually gets scheduled. This is because the scheduler has not been informed that this task exists and also doesn't know whether, or when, it should be scheduled.

So you can enter this information and try to get your task to run for the first time after a delay of 5s and then run periodically every 2s. To do this, open up the project specific file called `TaskConfig.h`. Now add the following items:

```
extern void fnMyFirstTask(TTASKTABLE *ptrTaskTable); // prototype

#define TASK_MY_FIRST_TASK      'x'  // my first task's reference
```

This is the reference which is used to identify the task. It can be any letter, number etc. but should be unique in the system – check the other task references and avoid any collisions. Since there is no other task in the reference project with the reference 'x' this can be used safely.

In the array `ctNodes[]` add your new task's reference

```
const UTASK_TASK ctNodes[] = {
  DEFAULT_NODE_NUMBER,     // configuration the single node
  TASK_WATCHDOG,
  ...
  TASK_MY_FIRST_TASK,      // add the task to the configuration
  0,                       // end of single configuration

  // insert more node configurations here if required
  0                        // end of configuration list
};
```

Be sure to put the new task definition *before* the end of the configuration. The µTasker supports multiple node configurations and also multi-start with different node configurations, but these capabilities are not discussed here – we'll keep it simple since we are interested in seeing the first new task in operation…

We are almost finished, but first need to add the task characteristics – we have already specified that there is a task to be basically added but now we have to be more specific. To do this, add the following to the `ctTaskTable[]` array

```
const UTASKTABLEINIT ctTaskTable[] = {
{ "Wdog",fnTaskWatchdog,NO_QUE,0,(DELAY_LIMIT)(0.2 * SEC),  UTASKER_GO}, //
watchdog task (runs immediately and then periodically)
   ...
{ "x marks my first task!!", fnMyFirstTask, NO_QUE,(DELAY_LIMIT)(5 * SEC),
(DELAY_LIMIT)(2 * SEC),  UTASKER_STOP},      // my first task (runs after a
delay of 5s and then periodically every 2s)
...
};
```

Note that the task string must start with the task's reference (x)!!!

After compiling and running this, you should see a first message sent out after 5s and then every 2s period a further. This is what it looks like in a terminal emulator:

```
Hello World!! Test number 0
Hello World!! Test number 1
Hello World!! Test number 2
Hello World!! Test number 3
Hello World!! Test number 4
Hello World!! Test number 5
Hello World!! Test number 6
Hello World!! Test number 7
Hello World!! Test number 8
Hello World!! Test number 9
Hello World!! Test number 10
Hello World!! Test number 11
Hello World!! Test number 12
Hello World!! Test number 13
Hello World!! Test number 14
```

And so on…

This output also operates via TELNET if you have TELNET activated in your µTasker project. When there is a TELNET connection the debug output will be diverted to this connection rather than go via the UART output.

Note also that there is also an introduction to the basic definition of tasks in the document http://www.utasker.com/docs/uTasker/uTaskerV1.3.PDF

This has shows a very simple task which is scheduled in a periodic fashion. It has not used an input queue for communication and it has not used any operating system functions apart from the debug output calls which are supplied as a part of the project. Note also that the debug output was actually configured by the application task (with user definable parameters such as speed and parity). The application task (in `application.c`) was also responsible for starting the TELNET support if you used that. You will also see - assuming that you haven't stripped everything else from the project – that all other demo stuff like the web server is still there and working. Your new task is also operating alongside this. You can however work in your new file almost as if it is the only thing there; it is not disturbed by the other stuff and it doesn't disturb the operation of the other stuff – this modularity helps greatly simplify programming and results in cleaner project structures!

6. Second Example – The Task being configured by another Task

In the first example your new task was configured to start after an initial delay and then periodically. This resulted in a very simple configuration but in many cases more flexibility is required.

This time we will not configure the task to be scheduled periodically but instead we will get another task to start it and then let it schedule itself.

To do this, make the following changes to the task characteristics:

```
{  "x marks my first task!!",
    fnMyFirstTask,
    NO_QUE,(DELAY_LIMIT)(NO_DELAY_RESERVE_MONO),
   (DELAY_LIMIT)(0),  UTASKER_STOP},
   // my first task (a timer has been reserved for its use but not need activated)
```

As noted in the comment, the task has reserved timer resources for its use later – if the delay and periodic values are left as 0 it will not receive any timer resources and will thus not be able to use its own timer later – *this saves resource space if no timer is required*.

The task will however never be scheduled so its scheduling will have to be started from another position in the code – we chose the application task (`application.c`), so place the following code there, just after the code to open the serial interface:

```
if (!fnSetNewSerialMode(FOR_I_O)) {                  // open serial port for I/O
    return;                     // if the serial port could not be opened we quit
}
DebugHandle = SerialPortID;         // assign our serial interface as debug port
uTaskerStateChange(TASK_MY_FIRST_TASK, UTASKER_ACTIVATE);
                                    // activate the new task
uTaskerMonoTimer( TASK_MY_FIRST_TASK, (DELAY_LIMIT)(3 * SEC), 0 );
                                    // set periodic interval
```

Now you will see that your new task is scheduled to run when the application task starts (this is provoked by the call `uTaskerStateChange()` setting the task state to `UTASKER_ACTIVE`). The application task starts after a delay of 100ms so this happens quite quickly. Since the application task has also configured the UART for debug use it is also ready for use by your new task as soon as it is scheduled.

The second call `uTaskerMonoTimer()` configures periodic scheduling of the task with a period of 3s – note that this occurs because the final parameter is 0, otherwise it will cause the task to be scheduled after the defined delay once with the event number as specified by the past parameter.

Furthermore, if the initial `uTaskerStateChange()` is omitted, your new task will be periodically scheduled with its first start 3s after the application task changed its characteristics!!

7. Third Example – Adding a Queue to Your Task and getting it to do a bit more Work

Up to now your task has used timer resources but has not used an input queue. The input queue is also a characteristic of the task which is configured in the task table. When a length of 0 is set the task has no input queue and so uses no resources for this.

Now we will add a small queue which we will then use to communicate with the task (initially by sending it a simple event) and then for its own timer use.

Please do the following:

Give your task a small queue by modifying its task entry – there are a few queue lengths defined in `uTasker.h` but you can in fact use any value here.

```
{   "x marks my first task!!",
    fnMyFirstTask,
    SMALL_QUEUE,
    (DELAY_LIMIT)(NO_DELAY_RESERVE_MONO),
    (DELAY_LIMIT)(0),
    UTASKER_STOP},
```

Now instead of starting your new task from the application task, get the application task to send it a start-up event (we will use a simple interrupt event – which can be defined in `application.h`).

Instead of the `uTaskerStateChange(),` use the following in `application.c`

```
fnInterruptMessage(TASK_MY_FIRST_TASK, WAKE_UP_LITTLE_BABY);
```

In `application.h` you can add a new event as following (at the bottom of the file). The event number is in fact not critical, but avoid the value 0 since this is a null-event and will not wake it up.

```
#define WAKE_UP_LITTLE_BABY   132
```

In your new task's file, change its content to look like this:

```
#define OWN_TASK            TASK_MY_FIRST_TASK
#define E_TIMER_PERIODIC    1


extern void fnMyFirstTask(TTASKTABLE *ptrTaskTable)
{
    static unsigned char ucCounter = 0;
    QUEUE_HANDLE  PortIDInternal = ptrTaskTable->TaskID;         // queue ID for task input
    unsigned char ucInputMessage[HEADER_LENGTH];       // reserve space for simple messages

    while ( fnRead( PortIDInternal, ucInputMessage, HEADER_LENGTH )) { // check input queue
        switch ( ucInputMessage[MSG_SOURCE_TASK] ) {                   // switch on source
        case TIMER_EVENT:                                              // timer event
            if (E_TIMER_PERIODIC == ucInputMessage[MSG_TIMER_EVENT]) {
                fnDebugMsg("Test number ");
                fnDebugDec(ucCounter++, 0, 0);
                fnDebugMsg("\r\n");
                if (ucCounter == 10) {
                    fnDebugMsg("Work done!!\r\n");
                    ucCounter = 0;
                }
                else {
                    uTaskerMonoTimer( OWN_TASK, (DELAY_LIMIT)(1*SEC), E_TIMER_PERIODIC );
                }
            }
            break;

        case INTERRUPT_EVENT:
            if (WAKE_UP_LITTLE_BABY == ucInputMessage[MSG_INTERRUPT_EVENT]) {
                fnDebugMsg("Hello World!!\r\n");
                uTaskerMonoTimer( OWN_TASK, (DELAY_LIMIT)(6*SEC), E_TIMER_PERIODIC );
            }
        }
    }
}
```

Now you have something a bit more powerful and controllable.

When the application task starts it will send an interrupt event to your new task so that it can start controlling itself. It reads its input queue (*which is used for receiving the interrupt event, timer events and other messages*) and reacts to the event `WAKE_UP_LITTLE_BABY` by writing the text "`Hello, World!!`" and scheduling itself to be woken by a timer event called `E_TIMER_PERIODIC` after an initial delay of 6 seconds. When this timer fires it will then start itself again with a delay of just 1 second and repeat this for a certain amount of time before terminating.

The new debug output now looks like this:

```
Hello World!!  (immediately after the application task starts)
Test number 0  (after 6s delay)
Test number 1  (after 1s and then every further 1s)
Test number 2
Test number 3
Test number 4
Test number 5
Test number 6
Test number 7
Test number 8
Test number 9
Work done!!
```

Now your new task has full control over itself and has a structure very suited for state-event operation: it is woken on events (*in this case an interrupt event and timer events*) and otherwise sleeps until the next event occurs.

If the application were to resent the original interrupt event the process would start all over again!

8. Adding some existing code to your project

Now it may be that you have some code which you would like to add to the project which looks something like this:

```
int main(void)
{
    fnInitialiseSomeThings();

    while (1) {
        if (event1 != 0) {
            fnHandleEvent1();
        }
        if (event2 != 0) {
            fnHandleEvent2();
        }
        if (event3 != 0) {
            fnHandleEvent3();
        }
    }
}
```

Such code don't assume any operating system and is build around polling for events to take place, which may be flagged from interrupt routines or by other events. *Open-source code modules are also very commonly supplied in this form*.

The question is how to allow this code, or several similar modules which are constructed like this, to operate in a way that all receive processing resources and all can live alongside the µTasker base project (or similar) with its TCP/IP resources?

The answer is in this case, thankfully, very simple. Since it is based on polling anyway we simply add each module to a polling task:

```
extern void fnMyFirstTask(TTASKTABLE *ptrTaskTable)
{
    static iTaskState = 0;

    if (iTaskState == 0) {
        fnInitialiseSomeThings();
        iTaskState = 1;
    }

    if (event1 != 0) {
        fnHandleEvent1();
    }
    if (event2 != 0) {
        fnHandleEvent2();
    }
    if (event3 != 0) {
        fnHandleEvent3();
    }
}
```

To operate your new task in polling mode it is simply configured with the state UTASKER_GO.

```
{       "x marks my first task!!",
        fnMyFirstTask,
        NO_QUE,
        0,
        0,
        UTASKER_GO},            // my first task runs in polling mode
```

Now we have these modules and the complete demo project working alongside each other. By making use of the µTasker operating system resources it is usually quite easy to improve the efficiency of such a task by removing unnecessary polling in favour or event based operation without greatly changing its basic operation in any way!

9. Conclusion

This simple introduction has shown how new users can quickly take first steps in starting a new project, adding a new task and testing some simple but useful basic operation. All of the new code can still operates alongside the existing demo project resources to enable new users to also immediately benefit from these integrated resources as part of the new project.

The source to the new code to the examples can be found at the following thread to avoid the need to type it in when following the document:

http://www.utasker.com/forum/index.php?topic=541.0

Modifications:
- V0.0 31.3.2008 First provisional draft.
- V0.1 6.4.2008 Added instructions for installing service packs.
- V0.2 6.3.2009 Added header, conclusion and notes about target project include paths.

Appendix A - µTasker Commands use in this Document

```
extern QUEUE_TRANSFER fnDebugMsg (CHAR *ucToSend);
                                                   // send string to debug interface
```

```
extern CHAR *fnDebugDec(signed long slNumberToConvert,
                       unsigned char iStyle,
                       CHAR *ptrBuf);            // take a value and send it as
                                                    decimal string over the debug
                                                    interface or put in buffer
```

```
extern void uTaskerStateChange(UTASK_TASK pcTaskName,
                              unsigned char ucSetState);
                                                  // change the state of a task
```

```
extern void uTaskerMonoTimer(UTASK_TASK pcTaskName,
                            DELAY_LIMIT delay,
                            unsigned char time_out_nr ); // schedule task after
                                                            delay, with timeout event
```

```
extern QUEUE_TRANSFER fnInterruptMessage(UTASK_TASK Task,
                                        unsigned char ucIntEvent);
                                               // send an interrupt event to a task
```

```
extern QUEUE_TRANSFER fnRead(QUEUE_HANDLE driver_id,
                             unsigned char *input_buffer,
                             QUEUE_TRANSFER nr_of_bytes );
                                        // read contents of input queue to a buffer
```