Introduction

µTasker is an operating system designed especially for embedded applications where a tight control over resources is desired along with a high level of user comfort to produce efficient and highly deterministic code.

The operating system is integrated with TCP/IP stack and important embedded Internet services along side device drivers and system specific project resources.

µTasker and its environment are essentially not hardware specific and can thus be moved between processor platforms with great ease and efficiency.

However the µTasker project setups are very hardware specific since they offer an optimal pre-defined (or a choice of pre-defined) configurations, taking it out of the league of “board support packages (BSP)” to a complete “project support package (PSP)”, a feature enabling projects to be greatly accelerated.

This document discusses the implementation and use of the µTasker CAN driver.

Overview of CAN

CAN (Controller Area Network) is an asynchronous serial communication protocol. It is implemented as a bus with the following limiting speed/distance combinations:

- Maximal speed of Can bus is 1M, with a range of about 40m.
- At 100kb/s a range of about 500m is possible
- At 10kb/s approximately 1km is possible

Can is often used in automotive electronics to control such things as engine management, lights, sensors, ABS etc. but it is not restricted to this use and has found wide acceptance in many industrial applications.

Nowadays the CAN specification 2.0B is commonly respected by the controllers.

Configuring for CAN bus use

The uTasker is delivered with general CAN support in the file `can_drv.c`. This is the generic driver which is essentially hardware independent. The low level interfacing with the CAN controller itself is performed in the hardware file (for example, `M5223X.c` in the case of this device with internal CAN support).

To activate the necessary support in the project the following define should be set in config.h:

```
#define CAN_INTERFACE
```

This will enable all the necessary code and also example routines in the demo project.

Then it is necessary to specify how many logical CAN interfaces you would like to use. This ensures that the optimum amount of resources are reserved for the interface but does mean that you will have to think about the number that you actually require and enter this correctly… I have used the words ‘logical CAN interfaces’ since the CAN interface is a bus interface and even if there is only one hardware CAN interface the µTasker allows it to be used as if it were several independent ones (if you want). Therefore if the hardware has one interface and the software would like to use it as three logical interfaces then the number of interfaces is 1 x 3 (not that difficult is it…) and so simply set #define NUMBER_CAN 3.

If you decide you would like a further logical interface, don’t forget to increment the value so that there are enough resources reserved for it to be actually opened.

Should there be more that one physical CAN interface then the total number of logical interfaces is simply the sum of the required number of logical interfaces on each of the physical ones.

It will not hurt to define a larger number than actually required, there will simply be a bit of memory reserved for them which will not in fact be used. But, best not to waste anything.

The rest of the configuration takes place when opening the interface for use, which is described in the next section.

Don't forget to use a device with CAN support since the CAN interface is not something which a processor can do using bit banging. It is no use activating the software support if the hardware doesn't do it. The code will also not compile. But this is rather obvious so let's get on to opening the communication channel.

Opening the CAN interface

The CAN interface is opened and configured by using the generic open function fnOpen(). The interface type is specified to be of TYPE_CAN for input and output along with some parameters:

```
QUEUE_HANDLE CAN_interface_ID = fnOpen( TYPE_CAN, FOR_I_O,
&tCANParameters );
```

The first time that the open if performed for the hardware channel, the controller is configured for use. When subsequent opens are called, the interface parameters are not set or modified but only logical interfaces activated. This means that the first caller of the open function is responsible for setting its operating characteristics.

```
CANTABLE tCANParameters; // table for passing information to driver
tCANParameters.cTask_to_wake = OWN_TASK; // wake us on buffer events
tCANParameters.ucChannel = 0;         // first hardware interface
tCANParameters.ulSpeed = 1000000;     // 1 Meg speed
tCANParameters.ulTxID = 0x102;        // default ID of destination
tCANParameters.ulRxID = (CAN_EXTENDED_ID | 0x00000105);  // our ID
tCANParameters.ulRxIDMask = CAN_EXTENDED_MASK;
                                // use all bits for compare
tCANParameters.usMode = 0;      // use normal mode
tCANParameters.ucTxBuffers = 2; // assign two tx buffers for use
tCANParameters.ucRxBuffers = 3; // assign three rx buffers for use
```

The CAN controller will, in this example, be configured for 1M operation with its own receive ID and a default destination ID. The receive ID can use mask bits but the operation of these will be a little hardware specific and more details are given in the corresponding hardware dependent appendix.

As well as configure the controller, this first open has defined two transmit buffers and three receive buffers for use by the logical interface. For illustration purposes the receiver ID is a 29-bit extended format ID but the destination ID is a standard 11-bit format. The number of available buffers in the hardware controller limits the maximum number of logical interfaces and once the buffers have all been allocated, no more logical interfaces can be defined. Since the number of usable buffers is hardware dependent you will have to know the basics of your hardware to get the most out of it. Generally simply try to use up the maximum number of buffers possible since this is the best solution, whereas the more buffers available the more efficient it can work and less are the chances of receiver overruns or blocked transmission buffers.

Notice that the calling task has entered its coordinates in `cTask_to_wake` so that it will be notified of CAN events belonging to the logical interface which it has defined. It is also best to open multiple logical interfaces from individual tasks for each so that the specific task can handle its own messages without having to check more than one interface handle.

Here is how a second task (which is started later than the first, which has configured the operating parameters) opens up a second logical interface on the same hardware port:

```
QUEUE_HANDLE CAN_interface_ID_2;
CANTABLE tCANParameters; // table for passing information to driver
tCANParameters.cTask_to_wake = OWN_TASK; // wake us on buffer events
tCANParameters.ucChannel = 0;  // first hardware interface
tCANParameters.ulSpeed = 0;    // speed already defined
tCANParameters.ulTxID = (CAN_EXTENDED_ID  | 0x00001234);
                           // default extended ID of destination
tCANParameters.ulRxID = 0x144;  // our standard ID
tCANParameters.ulRxIDMask = CAN_STANDARD_MASK;
                                // use all standard bits for compare
tCANParameters.usMode = 0;      // use standard mode
tCANParameters.ucTxBuffers = 6; // assign six tx buffers for use
tCANParameters.ucRxBuffers = 5; // assign five rx buffers for use


CAN_interface_ID_2 = fnOpen( TYPE_CAN, FOR_I_O, &tCANParameters );
```

Since the second open has requested 6 transmission buffers and 5 reception buffers, the total number of buffers in the hardware is assume to be at least 16 (which is the amount available for example in the FlexCAN controller of the M5223X). All buffers have been allocated in this case and so no further would be possible. The number for the define NUMBER_CAN is 2 since there are two logical interfaces.

It is also to be expected that the second logical interface is the one which is to be subjected to the most traffic since it has the most buffers allocated to it.

Sending CAN data messages

Once the interface has been opened and each logical interface has received a queue handle the CAN interface can be used. We can send a CAN message by calling the generic `fnWrite()` routine.

When sending data it must be sent to a specific destination ID and can have the data length from 1 to 8 bytes, as according to the CAN specification. A default destination was set for this logical interface when opened and if this is the destination address of the message to be sent then it is very easy:

```
unsigned char ucTestMessage[] = {1,2,3,4,5,6,7};  // Test message
if (fnWrite(CAN_interface_ID, ucTestMessage, sizeof(ucTestMessage))
                                 != sizeof(ucTestMessage)) {
    // Error. Eg. no transmission buffer free
}
```

The driver will find a free transmission buffer (the more defined for use by the logical interface, the more chance that one of them will be free) and organise the transmission. Just because the write is successful doesn't mean that the message can actually be delivered, it means that that the message has been successfully passed to the CAN controller transmission buffer and the CAN controller will try to deliver it. The delivery process can take a short time if the bus is heavily loaded and collisions take place. This is the job of the CAN controller and it will either successfully deliver the message at some time in the near future or it will fail, for example if the destination address doesn't exist. We will learn about what

actually happens with the message later when the CAN driver informs us of either success or failure as described in the next section.

Note that as long as there are free transmission buffers available then more than one message can be sent (queued) for transmission. The more buffers allocated for a logical interface, the more messages can be queued…

If it is necessary to send a message to a specific destination address then this can be performed by sending it as follows:

```
unsigned char ucTestMessage[] = {(CAN_EXTENDED_ID | 0x00), 0x01,
0x02, 0x03, 1,2,3,4,5,6,7};
                    // Test message starting with destination ID
if (fnWrite(CAN_interface_ID, ucTestMessage,
                     (sizeof(ucTestMessage) | SPECIFIED_ID))
                                 != sizeof(ucTestMessage)) {
    // Error. Eg. no transmission buffer free
}
```

Here the first 4 bytes of the message passed to the write function are the destination ID in network byte order (or big-endian) – in this case 0x00010203 and it is an extended format ID. The length to be transmitted includes the ID length (which is not actually transmitted of course) and the flag `SPECIFIED_ID` is set in the length field so that the driver knows that it must enter this address and remove it from the data before sending. The default destination ID is not overwritten and so can be used later as normal so that sending to the default destination ID remains simple. To send a standard format ID, the first four bytes should be specified without the `CAN_EXTENDED_ID` flag in the high order byte as in the following example: 0x00, 0x00, 0x00, 0x05 [standard format ID 5].

So what happens now? Well we still have to know whether the message or messages which we just sent off actually arrived or whether there was a transmission error or, more likely, if the destination was not reachable. It may be that we are using a higher level protocol and so we expect the destination to return a message so the basic way of operation is that the driver is silent when messages could be delivered without error. It only disturbs us when there is bad news.

If however there are benefits in receiving local confirmation of a delivered frame, this can be activated by setting the `CAN_TX_ACK_ON` flag in the length field of the write function. This is probably of most use in systems where only one transmission is outstanding at one time or for general test purposes. When a frame could be delivered, the owner task is optionally woken with an interrupt event of type `CAN_TX_ACK`.

When a transmission fails, the owner task is woken with an interrupt event of type `CAN_TX_ERROR`. The task can then read the erroneous buffer using the read call and thus retrieve the undelivered message. In fact the CAN controller will try repeatedly to deliver a message and report an error periodically and quite frequently. The error message is sent only after a number of unsuccessful tries since this indicates that the destination is almost certainly not reachable. The buffer is then set to an inactive state until the application retrieves the undeliverable message. Here is an example of how a task can interpret the interrupt event and retrieve the message contents:

```
......   case INTERRUPT_EVENT:      // on an interrupt event
             switch (ucInputMessage[MSG_INTERRUPT_EVENT]) {
             case CAN_TX_ERROR:  // no ACK to a message we sent
                 Length = fnRead(CAN_interface_ID, ucInputMessage,
                          GET_CAN_TX_ERROR); // read error
                 // ucInputMessage[0]..[3] contain the dest. ID
                 // ucInputMessage[4]..[Length + 3] contain the message
                 break; ......
```

Just what the application does with the buffer contents is up to the application. The buffer is however freed for further use after reading the error so the read is important. Such an error normally means that the destination is not working so repeating is not very useful although may be at a later time once the problem has been fixed…

Receiving CAN data messages

When a CAN data message is received it is placed in a free receive buffer and the owner task is notified of the fact via an interrupt event. It can then pick up the data by calling the generic fnRead(). It is not necessary to specify the read length since data should be read as a block from a single buffer, however there are a number of options available.

```
......   case INTERRUPT_EVENT:     // on an interrupt event
            switch (ucInputMessage[MSG_INTERRUPT_EVENT]) {
            case CAN_RX_MSG::      // A CAN rx. message is waiting
                Length = fnRead(CAN_interface_ID, ucInputMessage,
                         (0 | GET_CAN_RX_TIME_STAMP | GET_CAN_RX_ID));
                          // collect the message along with some details
                break; ......
```

In this example the message is returned with a 16 bit time stamp and also the receive ID of the local CAN buffer (in case we should have forgotten it…). The information is put into the receive buffer `ucInputMessage` beginning with a byte signifying the receive status, the 2 byte time stamp, followed by the 4 byte ID and then the data, whose length is returned by the `fnRead`() call including any requested details. The time stamp and ID are optional and if they are not specified only the data will be returned, with a leading byte specifying the receive status.

Note that the `fnRead`() will return the first message found in a buffer belonging to the calling task. If there is more than one reception waiting, there will also be more than one CAN_RX_MSG to wake the task. It is also possible to enclose the read of the CAN handle in a loop until a length of zero is returned so that multiple receptions take place as fast as possible.

Remote Frames

Remote Frames are a nice little feature of the CAN bus which allow one node to request data from another from a pick-up mail box. The node supplying the data doesn't have to wait for the other node to request it but can put the data in a local mail box and this will be automatically sent in response to the Remote Frame.

This feature has been integrated into the µTasker CAN driver so that it is very simple to use.

There are two cases:

The first is when some data is to be put into the mail box for collection. It is like sending data but the data is not sent immediately but only when specifically requested by a Remote Frame from another node. When there is more than one Remote Frame received, the same data is sent repeatedly on each request until the Remote Frame buffer is subsequently deactivated.

The second is when a remote frame is to be sent to pick up some data.

Both use the generic `fnWrite`() with the following parameters in the length field:

```
unsigned char ucTestMessage[] = {1,2,3,4,5,6,7};  // Test message
if (fnWrite(CAN_interface_ID, ucTestMessage,
                       (sizeof(ucTestMessage) | TX_REMOTE_FRAME))
                                   != sizeof(ucTestMessage)) {
    // Error. Eg. no transmission buffer free
}
```

This example shows data being queued to be sent, but only on request by a Remote Frame.

Note that the data will be transmitted as long as the buffer remains valid and so can be sent again and again and again without writing the data again. If you would like a notification once the data has been sent for the first time, use the flag CAN_TX_ACK_ON in the length field. To stop the Remote data message call `fnWrite`() with just TX_REMOTE_STOP in the length

field and a null pointer in the data field, which will convert the transmission buffer back to a normal transmission buffer.

When a transmission buffer is being used as a Remote Frame transmission buffer, it is not available for normal data transmission. Only one of the specified transmission buffers will be used on request as a Remote Frame buffer at any one time. If further data is written for Remote Transmission it will overwrite an existing Remote Frame transmission buffer, thus modifying the data sent on a Remote Frame.

```
if (fnWrite(CAN_interface_ID, 0, 0) == 0)) {
    // Error. Eg. no transmission buffer free
}
```

This example shows a Remote Frame being sent to the default destination ID.

If a specific destination ID is required for this Remote frame then this is possible as follows:

```
unsigned char ucID[] = {0x00,0x01,0x02,0x03};  // Remote ID
if (fnWrite(CAN_interface_ID, ucID, SPECIFIED_ID) == 0)) {
    // Error. Eg. no transmission buffer free
}
```

The transmission of a Remote Frame is rather special since the transmission buffer used to send the Remote frame is subsequently used as a temporary receive buffer to accept the requested data. Assuming that the Remote Frame transmission was successful and the corresponding data is received, the fact is signalled by an interrupt event of the type CAN_RX_REMOTE_MSG. In this case the data can be collected by using the normal fnRead() call. If it should prove necessary to be sure that the data read is the response from the Remote Frame and not another data reception occurring at the same moment in time it can be selectively collected by specifying GET_CAN_RX_REMOTE in the length field.

The fact that the temporary Remote Frame reception buffer has been read by the application frees it and converts it back to a free transmission buffer as it was before the Remote Frame was sent.

Normally the answer to the Remote Frame transmission will be received immediately however it is not guarantied that there will be an answer to the Remote Frame transmission. If the other node has not prepared data for the transfer it will never take place. In this case the CAN buffer will remain in the reception state indefinitely. The controlling software should thus convert the buffer back to its original free transmission state once it has detected that there will be no reception by reading it with the FREE_CAN_RX_REMOTE flag set which will then convert it for further transmission use. (It's probably best to use a short timer to signal that there has not been a response in an acceptable time and then clear as described).

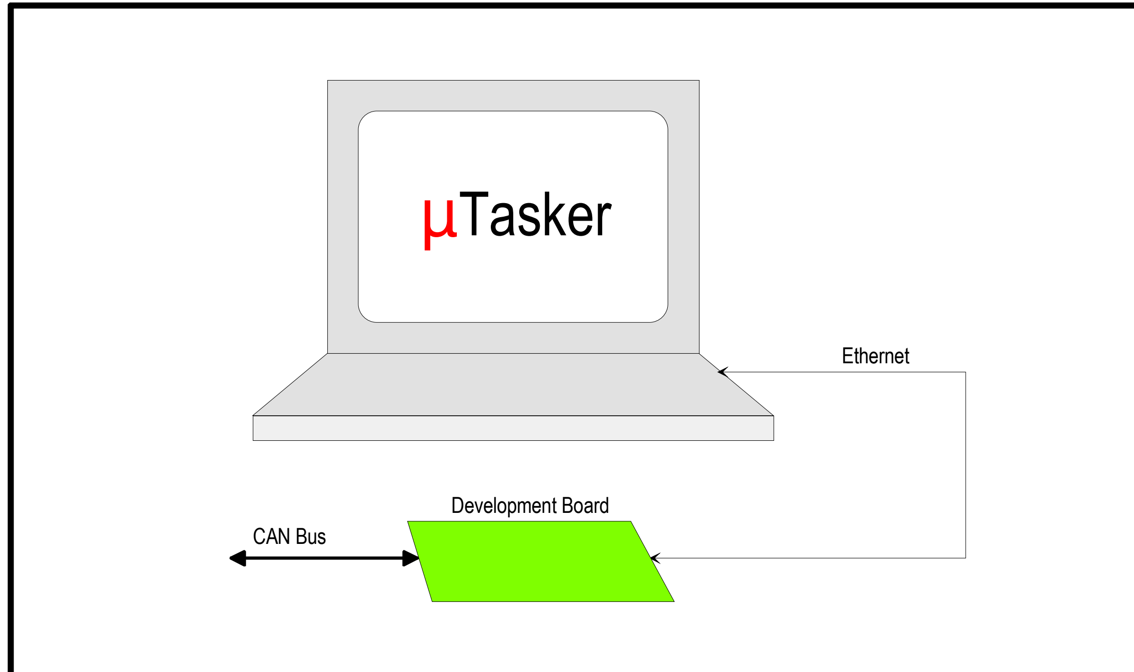
µTasker CAN simulator

Probably the most powerful feature of the µTasker environment is its real-time simulator which allows large amounts of projects for embedded systems to be developed and debugged comfortably on the PC before final testing on the hardware. Some projects can even be completely developed and maintained in the simulator environment making for great advances in efficiency in comparison to conventional methods based on the traditional `code → compile → download → test → target debug` cycle.

Since most PCs have serial and Ethernet interfaces the µTasker maps the processor's internal peripherals to these so that simulated projects can run on the PC, which is connected to the real network. The problem with normal PCs is that they do not have an integrated CAN interface and so the question is how do we develop our CAN based project within the µTasker environment so that we can continue profiting from its unique simulation capabilities?

One method would be to define a PC CAN interface which would then have to be purchased and connected to the PC so that it can be used in a similar fashion to the already supported, and normally integrated, COM port and Ethernet interface. But I decided against this because most of us wanting to develop an interface for a certain piece of hardware will have some form of development board with this interface on it. Since the µTasker concentrates on single-chip embedded processors with integrated Ethernet then it seems more logical to use this hardware for the job – I mean the final code will be tested on it anyway so why not use it also for an extension to the simulator in the meantime.

This gives the following solution which is basically hardware independent, requiring an Ethernet interface and the hardware peripherals to be 'simulated'.



Although the µTasker environment controls the peripheral registers as normal, the simulator communicates such changes to the development board so that it can communicate with a real CAN bus. This can of course be extended to all types of hardware but the most important principle if that the development board is not completely specified – it must simply have an Ethernet interface and the hardware interface of interest and some software running on it to do communicate with the µTasker simulator.

The software is a simple µTasker project which can be adapted to suit any hardware available fulfilling the basic needs. The processor on the development board must not be the same one as the µTasker is simulating – it must simply be capable of performing the CAN / Ethernet conversion as required.

So here's an example of the CAN project development environment used when I integrated the M5223X CAN driver:
- µTasker project requiring the CAN interface.
- M52235EVB board programmed with the µTaskerHWSim project code and set with known IP and TCP Port address.
- Ethernet connection between the development PC running the µTasker project requiring CAN interface.
- Power supply to the M52235EVB.
- Connection between the CAN interface of the M5223EVB and the real CAN bus which is to be connected to.

Note that it will normally be necessary to have at least one other node on the CAN bus to be able to really do anything of use but if you are doing a CAN project this should be obvious. In fact I used a M5213EVB as a second test node, also running a µTasker project when doing my first tests but you may have an operating CAN bus which you would like to hook up to or even a CAN analyser which may make life also somewhat easier.