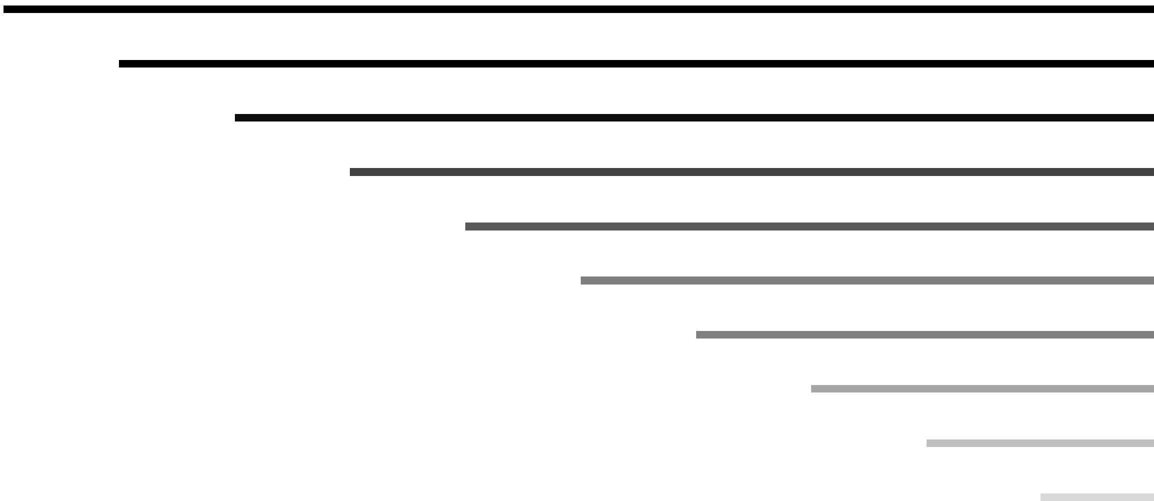
Embedding it better...

µTasker Document

Controller Area Netwok (CAN)



Copyright © 2011 M.J.Butcher Consulting

Table of Contents

1. Introduction....3
2. CAN Bit Timing....4
3. CAN Frame Format....5
4. Configuring for CAN bus use....6
5. Opening the CAN interface....7
6. Sending CAN Data Messages....9
7. Receiving CAN data messages....11
5.1. Remote Frames....12
8. CAN Monitoring and Simulation....14
9. Testing CAN Operation in the µTasker Project....17
10. Conclusion....18
Appendix A – Hardware Dependencies....19
a) FlexCAN....19

1. Introduction

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.

This document discusses the use of the µTasker CAN driver and includes the necessary technical details of the CAN protocol to enable the user to make decisions as to the best configuration and use in a particular environment.

The µTasker CAN interface and the µTasker project CAN demonstration are described as well as a technique for integrating the µTasker simulator into CAN networks for powerful development and debugging capabilities.

In the appendix there are additional hardware details concerning the CAN controllers supported on particular processors.

2. CAN Bit Timing

The bit speed defines the data rate of the Can bus, limited to maximum 1Mb/s.

One bit time is divided into 4 time segments, which are used for bit level synchronisation. These 4 time segments, in the order in which they take place, are called:

- Synchronisation segment [*an edge is expected to lie in this segment*]

- Propagation time segment [*a compensation for physical and driver bus delays – twice the actual delay value*]

- Phase buffer segment 1 [*Together with buffer segment 2 the sample point can be synchronised to compensate for edge phase errors*]

- Phase buffer segment 2

The actual sample point occurs at the end of Phase buffer segment 1 / start of Phase buffer segment 2.

The flexibility with which these phases can be adjusted depends on the Time Quantum used. The Time Quantum divides the bit time into units which can be distributed between the 4 time segments. The smaller the Time Quantum, the higher the resolution of possible adjustments. The number of Time Quanta is normally adjustable from at least 8 to 25.

The synchronisation segment has always a fixed single Time Quantum unit.

The Propagation time segment and Phase buffer segment 1 can be programmed from 1 to 8 Time Quanta units.

CAN controllers have also an IPT (Information Processing Time) which is also expressed in Time Quanta. Typically it has a value of 2.

3. CAN Frame Format

Messages can use either a standard format with 11 bit ID or extended format with 29 bit ID

Standard frames are built up of:

- SOF (1 Start-Of-Frame bit)
- **ID (11 bit ID)**
- RTR (Remote Transmission Request bit – ‘0’)
- IDE (Identifier extension bit – ‘0’)
- R0 (Reserved bit – set ‘0’)
- DLC (Data Length Control – 4 bits)
- DATA (0..8 bytes of 8 bits)
- CRC (15 bit CRC)
- CRC delimiter (‘1’)
- ACK Slot (Transmitter sends ‘1’ and any receiver can assert a dominant ‘0’)
- ACK delimiter (‘1’)
- EOF (End-Of-Frame 7 recessive bits)

Extended frames are built up of:

- SOF (1 Start-Of-Frame bit)
- **Identifier A (11 bit ID)**
- SRR (Substitute Remote Request – ‘1’)
- IDE (Identifier Extension Bit – ‘1’)
- **Identifier B (18 bit ID)**
- RTR (Remote Transmission Request bit – ‘0’)
- R0, R1 (2 reserved bits – set ‘0’)
- DLC (Data Length Control – 4 bits)
- DATA (0..8 bytes of 8 bits)
- CRC (15 bit CRC)
- CRC delimiter (‘1’)
- ACK Slot (Transmitter sends ‘1’ and any receiver can assert a dominant ‘0’)
- ACK delimiter (‘1’)
- EOF (End-Of-Frame 7 recessive bits)

Frame types:

- Data frame
- Remote frame
- Error frame
- Overload frame

The code stream uses bit stuffing. After 5 consecutive identical levels (0 or 1) a complimentary bit is inserted into the bit stream which is then removed at the receiver.

4. 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 demonstration 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 based on the application in hand. Although a CAN interface is a single hardware connection to the CAN bus it can be envisaged as a number of logical interfaces which share it – there may for example be several independent software modules communicating via the bus and so each of these three software modules can have its own virtual communication channel on the bus.

In the case of a single hardware interface and controller but three software modules sharing the bus the define `#define NUMBER_CAN 3` reserves the necessary resources for each of these to operate on their own virtual interface.

Often the processors include more than one CAN controller and so, if each of the CAN controllers were associated with 3 software modules using each interface the value `#define NUMBER_CAN (2 * 3)` would be appropriate.

5. 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 the hardware's 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 MHz speed
tCANParameters.ulTxID = 0x102;           // default ID of destination
tCANParameters.ulRxID = (CAN_EXTENDED_ID | 0x00000105);   // our extended 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 configuring 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.

6. 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 search for 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 (different from the default address) then this can be performed by sending it as follows:

```
unsigned char ucTestMessage[] =
             {((CAN_EXTENDED_ID >> 24) | 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 (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 on to the bus) 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 >> 24)` 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] contains the destination ID
                // ucInputMessage[4]..[Length + 3] contains 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 except for maybe at a later time once the problem has been fixed.

7. 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. 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 typical 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.

5.1. Remote Frames

Remote Frames are a nice feature of the CAN bus which allow one node to request data from another using 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 rather when specifically requested using 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 a multiple number of times 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 in response to a Remote Frame.

This following example shows a Remote Frame being sent to the default destination ID, meaning that the pick-up mail box at the destination will be collected:

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

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).

8. CAN Monitoring and Simulation

In order to monitor the activity on a CAN bus a CAN analyser is usually used which can interpret and display the individual messages being exchanged. For developing and debugging, the additional capability of generating CAN data or acknowledging CAN data on the bus (acting as an active receiver) can be useful.

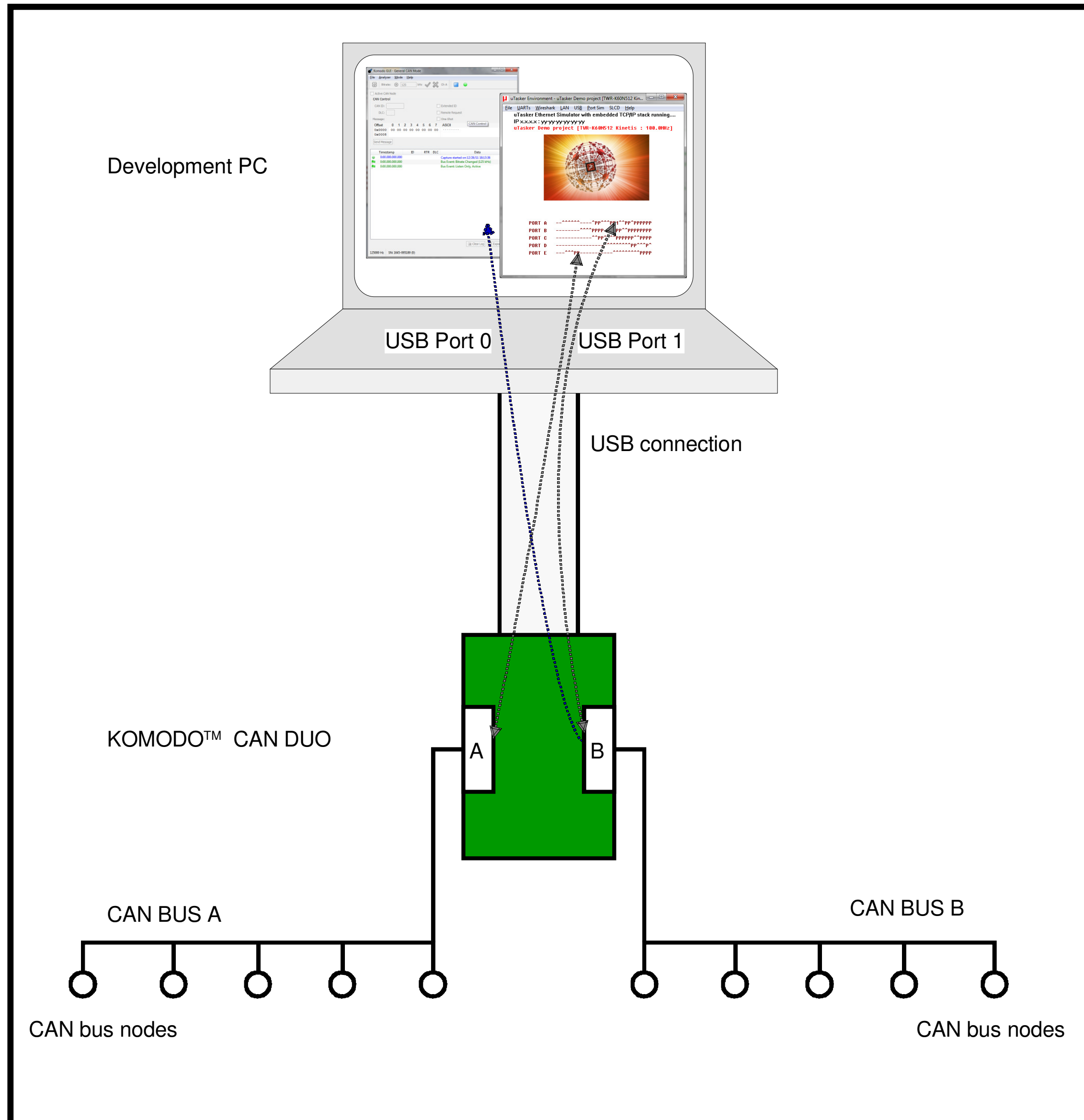
The µTasker simulator integrates support for such a CAN analyser which also gives the ability to connect the µTasker simulator to the CAN bus via the analyser in order to transmit data to the physical network and to receive data from it. This allows the µTasker simulator to act as a real node in a CAN bus system, simplifying development and debugging work during projects with CAN.

The CAN analyser chosen for this job is the KOMODO™ CAN DUO from TOTAL PHASE, Inc. http://www.totalphase.com/products/komodo_canduo/

This analyser supports 2 independent CAN interfaces allowing the traffic on two CAN buses to be monitored in parallel. It also allows the µTasker simulator to interact with up to two CAN controller interfaces at the same time.

Before explaining the µTasker simulator it is useful to first look at how the KOMODO™ CAN DUO operates:

It is a USB powered device which is connected to a PC via a USB cable. The USB connection has two ports (port 0 and port 1) which allows two applications on the PC to work with the device at the same time, sharing the single cable and USB connection. The analyser is equipped with two CAN channels (CAN A and CAN B), allowing up to two applications on the PC to work with up to two CAN nodes (or CAN buses) at the same time. This is illustrated in the figure below whereby one PC application is displayed as a CAN monitor (TOTAL PHASE supplies the Komodo GUI for this purpose) on one CAN bus and the other as the µTasker simulator. Two CAN interfaces are shown for a microcontroller with 2 CAN controllers; when only one CAN controller is available or only one is used, only one of the CAN interfaces will be active.



This particular usage configuration allows the µTasker simulator to interface to two CAN buses (the embedded application interacts with two CAN controllers) so that the embedded application can be developed, tested and debugged. The KOMODO GUI is used as a CAN analyser monitoring the traffic one of the buses to verify the system operation.

When the KOMODO™ CAN DUO is to be used for CAN interface simulation, the project define `SIM_KOMODO` needs to be activated and the USB port used for communication specified:

```
#define KOMODO_USB_PORT 1      // use this USB port (0 or 1) - any additional
                                  monitor program sharing the Komodo can use
                                  the other port
```

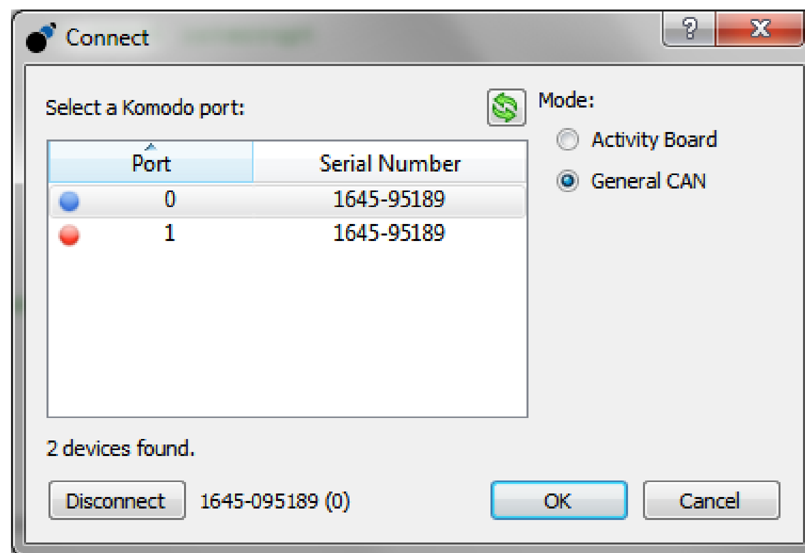
When the embedded project code opens the CAN interface (see the `fnOpen()` command description earlier in this document) it will also open a connection to the KOMODO™ CAN DUO on the specified USB port, whereby CAN controller 0 will use the CAN A interface and CAN controller 1 will use the CAN B interface. The activity light will be enabled on the

corresponding CAN interface to show that the connection has been established. When the simulator is terminated normally (using the exit command) the USB port will be terminated and the interfaces deactivated (the lights will no longer be on). Stopping the µTasker simulator by terminating debugging will leave the activity lights enabled but the USB port will still be terminated after a short delay by Windows during clean-up work.

When the embedded application running in the µTasker environment sends to the CAN interfaces the messages are sent to the corresponding physical CAN bus. Receptions from the CAN bus that match the CAN controller configuration will be received by the embedded application.

If the KOMODO™ CAN DUO is not connected when the µTasker simulator is operated the USB port will not be able to be opened and so no physical CAN operation will take place but the general operation of the simulator is otherwise not disrupted.

Since the µTasker simulator will be using the specified USB port during operation in CAN based projects the KOMODO GUI will not be able to use this port; the fact that the port is in use is also displayed by the connect dialogue:



In this instance Port 1 is displayed a red (in use) and so the user needs to connect the monitor to Port 0 instead, which is free as displayed by the blue signal.

The KOMODO GUI user's guide explains all details concerning its use: http://www.totalphase.com/download/pdf/komodo-gui-v1.20.pdf

9. Testing CAN Operation in the µTasker Project

When CAN operation is enabled in the µTasker project a test interface is activated in the file `can_tests.h` and a CAN menu is available on the command line interface.

Can_tests.h is a part of the application task, specifically handling the reception of CAN interrupt messages. It handles the CAN events types and displays corresponding details:

- `CAN_TX_REMOTE_ERROR`
- `CAN_TX_ERROR`
- `CAN_OTHER_ERROR`
- `CAN_TX_ACK`
- `CAN_TX_REMOTE_ACK`
- `CAN_RX_REMOTE_MSG`
- `CAN_RX_MSG`

The CAN commands can all be executed from the command line menu (on the serial interface, TELNET or USB CDC, depending on which is available and presently active).

```
   CAN commands
===================
up             go to main menu
can            Send to default [ch] <data hex>
can_s          Send to id [ch] <id> <data hex>
can_e          Send to ext. id [ch] <id> <data hex>
can_r          Request remote default [ch]
can_rs         Request remote id [ch] <id>
can_re         Request remote ext. id [ch] <id>
can_f          Free remote rx when no response
can_d          Deposit remote message [ch] <data hex>
can_c          Clear remote message [ch]
help           Display menu specific help
quit           Leave command mode
```

If no CAN channel is specified the default channel 0 is used. The following are examples of use:

`can 1 1122334455` sends the message `0x11, 0x22, 0x33, 0x44, 0x55` to the default destination on CAN channel 1

`can_s 0 123 442277` sends the message `0x44, 0x22, 0x77` to the standard ID `0x123` on CAN channel 0

`can_e 123abc 7722` sends the message `0x77, 0x22` to the extended ID `0x123abc` on CAN channel 0

`can_r 1` requests a remote frame from the default destination on CAN channel 1

`can_rs 0 9a` requests a remote frame from the standard ID `0x9a` on CAN channel 0

`can_re 16ff9a` requests a remote frame from the extended ID `0x16ff9a` on CAN channel 0

`can_f 1` frees the message buffer on CAN channel 1 that is waiting for a response to a transmitted request frame that didn't arrive

`can_d` deposits a remote response in the output mailbox of CAN controller 0 which will be sent automatically when a remote request is received. This will be sent a multiple number of times until cancelled by the `can_c` command. A repeat of the command can be used to change the message content.

Note also that there will be a `CAN_TX_REMOTE_ACK` event received when the remote mailbox is retrieved by any CAN node on the first retrieval only.

`can_c 0` clears the output mailbox of CAN controller 0 so that remote receptions no longer cause a transmission.

When port/external interrupt are enabled test CAN messages are sent when input edges are detected (or when buttons on particular boards are pressed/released). This allows for simple transmission tests.

For the inputs used see the file `Port_Interrupts.h`, where `#define IRQ_TEST` needs to be enabled. The call `fnSendCAN()` is called on certain events.

10. Conclusion

This document has given a brief introduction to CAN and described the µTasker driver interface for embedded projects using CAN enabled processors.

The µTasker simulator support for the KOMODO™ CAN DUO, allowing interfacing the simulation environment to up to two physical CAN buses, has been explained as well as how to verify the CAN operation in the µTasker project.

Additional hardware dependent details concerning CAN enabled processors can be found in the appendix.

Modifications:
V0.01 18.07.2006: Initial version
V0.02 27.12.2011: Revised version with new simulation interface and hardware appendix
V1.00 31.12.2011: Added `can_f` command and released

Appendix A – Hardware Dependencies

a) FlexCAN

The Freescale Coldfire and Kinetis processors use the FlexCAN controller. Each controller manages 16 individual CAN buffers which can be used for either transmission or reception. These message buffers are contained in the memory of the CAN controller and each has the following construction:

```
unsigned long          ulCode_Len_TimeStamp;
unsigned long          ulID;
unsigned char          ucData[8];
```

The first long word field contains a code informing of the present buffers use (eg. as reception or transmission buffer) and state (eg. whether empty, busy or full), and a data length and is filled with a time stamp when operation takes place.

The ID field is used to specify the destination ID when used as a transmission buffer.

The 8 byte data array is used to hold transmission data for the CAN controller to save reception data to.

Each CAN controller has a 16 bit timer which runs at the bit rate speed of the CAN bus. This is used for setting time stamps on reception and transmission.

Transmission involves using a free buffer and writing the data content and ID fields, followed by setting the code field to signal that the content should be sent (activation of the message buffer). The CAN controller then performs transmission, including retried in case of collisions. The time-stamp and code field is updated accordingly and an interrupt generated if enabled so that the driver can verify that there were no errors and inform the application if required.

The FlexCAN is clocked from either an internal bus clock or directly from an oscillator input signal. When possible the oscillator is preferred since it tends to have lower jitter than internal bus clocks generated using PLL.

Kinetis users should note that the CAN controller only works in early silicon versions when the system oscillator is enabled. The following define can be enabled in `app_hw_kinetis.h` when it is sure that the devices used do not suffer from this problem, otherwise the use of CAN in a project will automatically cause the system oscillator to be enabled.

```
#define ERRATA_E2583_SOLVED      // in early silicon the CAN controllers only work
                                    when the OSC is enabled (enable if the chip
                                    revision used doesn't suffer from the problem)
```