CAN is an asynchronous serial data communications protocol that excels in noisy environments. You can think of Controller Area Networks as high-speed intelligent serial communications networks with everything you wished you could have in a comparable RS-232 connection. A Controller Area Network is just as easy to physically implement as a legacy RS-232 network.

The chipKIT Max32 CAN Module..

Although the PIC32MX795F512L includes a twin-engine CAN Module, the chipKIT Max32 does not natively support Microchip's MCP2551 CAN Transceivers. Each of the chipKIT Max32's CAN engines can access a Controller Area Network Bus via it associated MCP2551 CAN Transceiver. The MCP2551 CAN Transceivers are mounted on the chipKIT Network Shield. MCP2551 CAN Transceivers at the extreme ends of the Controller Area Network Bus must have their Controller Area Network Bus pins terminated with 120Ω resistors.

In addition to a CAN protocol engine, the chipKIT Max32 CAN Module includes message acceptance filters and message assembly buffers. Incoming CAN messages are filtered by the message acceptance filters and masks. If the incoming CAN message meets the filter and mask requirements, the received CAN messages can then be routed to the receive message assembly buffer. Conversely, an outgoing CAN message is assembled in the transmit message assembly buffer before being handed over to the CAN protocol engine for transmission.

The chipKIT Max32 CAN Module contains absolutely zero buffer area. All of the buffered data must reside within a block of preallocated SRAM. The chipKIT Max32 CAN Module can transfer data to and from the SRAM buffer area without any CPU intervention.

Bringing Up the chipKIT Max32 CAN Module..

As far as initialization is concerned, both CAN1 and CAN2 are coded identically. Activating the chipKIT Max32‘s CAN Module is a process that involves manipulating the bits within a number of PIC32MX795F512L 32-bit CAN registers.

The PIC32MX CAN Module registers are organized into four functional groups:

1. CAN Engine/CAN Module Bit Rate Configuration Registers
2. Interrupt and Status Registers
3. Mask and Filter Configuration Registers
4. FIFO Control Registers

The chipKIT Max32 CAN Module can be configured and activated by writing the correct bit patterns to each of the four sets of CAN Module registers. The CAN demonstration code that accompanies this document is based on the Microchip PIC32MX CAN Module Peripheral Library and Section 34 of the PIC32MX Family Reference Manual.

There are six chipKIT Max32 CAN Module operation modes:

- Configuration Mode
- Normal Operation Mode
- Listen Only Mode
- Listen All Messages Mode
- Loopback Mode

- Disable Mode

Each operation mode is represented by a bit pattern placed in the Request Operation Mode (REQOP) bit block of the CAN Control register (CxCON). The CAN Module acknowledges the successful entry into the requested mode by duplicating the REQOP bits in the Operation Mode (OPMOD) bit block, which is also located in the CAN Control register. The polling of the OPMOD bits is one way of verifying the change of operational modes. You can also detect an operation mode change using the Mode Change Interrupt, which is enabled by the MODIE bit in the CAN Interrupt register.

Our CAN Module initialization begins by enabling the CANx Engine and pushing it into Configuration Mode:

```
CANEnableModule(CAN1,TRUE);

CANSetOperatingMode(CAN1, CAN_CONFIGURATION);
while(CANGetOperatingMode(CAN1) != CAN_CONFIGURATION);


CANEnableModule(CAN2,TRUE);

CANSetOperatingMode(CAN2, CAN_CONFIGURATION);
while(CANGetOperatingMode(CAN2) != CAN_CONFIGURATION);
```

The CAN Module Configuration register (CiCFG) cannot be modified outside of Configuration mode. With the CAN Engines enabled and configurable, the next step is to setup the clocking, whose bits lie inside of CiCFG (CAN Baud Rate Configuration Register).

Clocking the chipKIT Max32 CAN Module ..

A CAN bit time consists of four segments. Each segment is made up of a number of Time Quanta (T_Q) periods. For our purposes, the number of T_Q periods per bit (N) will be set to 10, which will be distributed as $3T_Q$ per phase segment. The synchronization segment will consist of a single T_Q.

Our chipKIT Max32 Demo Controller Area Network will operate with a baud rate of 250Kbps. Using the aforementioned number of T_Q periods per bit and the network baud rate, we can calculate the Time Quantum Frequency (FT_Q):

$FT_Q = N * F_{BAUD} = 10 * 250Kbps = 2.5MHz$

We need the Time Quantum Frequency to calculate the CAN Module baud rate prescaler value. The chipKIT Max32's PIC32MX795F512L is running at 80MHz. So, F_{SYS} is 80MHz:

```
C2CFG<BRP> = (FSYS/(2 * FTQ)) – 1
C2CFG<BRP> = (80MHz/(2 * 2.5Mhz)) – 1 = 15
```

As a result of our calculations, we conclude:

```
#define SYS_FREQ          (80000000L)
#define CAN_BUS_SPEED     250000
#define CAN2_BRPVAL       0x0F
```

Don't get too caught up in this math as the chipKIT Max32's CAN Module automatically computes the baud rate prescaler value in the *CANSetSpeed* function.

The idea is to size the Time Segments to allow reliable operation of the Controller Area Network. The phase and propagation segments insure that any drift on the Controller Area Network Bus due to oscillator shift or propagation time are addressed. The phase and propagation segments are coded in this manner:

```
canBitConfig.phaseSeg2Tq            = CAN_BIT_3TQ;
canBitConfig.phaseSeg1Tq            = CAN_BIT_3TQ;
canBitConfig.propagationSegTq       = CAN_BIT_3TQ;
canBitConfig.phaseSeg2TimeSelect    = TRUE;
canBitConfig.sample3Time            = TRUE;
canBitConfig.syncJumpWidth          = CAN_BIT_2TQ;

CANSetSpeed(CAN2,&canBitConfig,SYSTEM_FREQ,CAN_BUS_SPEED);
```

The *canBitConfig* structure was spawned from the *CAN_BIT_CONFIG* parent structure that is found in the PIC32MX CAN Peripheral Library's *CAN.h* file. Note that all of the parameters we took into consideration for our calculations are used by the *CANSetSpeed* function.

CAN Module Message Buffer Memory Area..

We can use the PIC32MX CAN Peripheral Library functions to easily setup separate transmit and receive buffer areas in chipKIT Max32 SRAM with a minimum of coding. Let's specify enough Message Buffer area for a transmit channel and a receive channel with each channel containing 8 Message Buffers of 16 bytes each:

```
UINT8 CAN2MessageFifoArea[2 * 8 * 16];
CANAssignMemoryBuffer(CAN2,CAN2MessageFifoArea,2 * 8 * 16);
```

The chipKIT Max32's CAN Module automatically allocates the specified memory space for a transmit FIFO and a receive FIFO according to the arguments of the *CANAssignMemoryBuffer* function. Once the Message Buffer memory is allocated, we can tell the chipKIT Max32's CAN Module to organize it into addressable transmit and receive buffer areas:

```
CANConfigureChannelForTx(CAN2,CAN_CHANNEL0,8,CAN_TX_RTR_DISABLED,CAN_LOW_MEDIUM_PRIORITY);
CANConfigureChannelForRx(CAN2,CAN_CHANNEL1,8,CAN_RX_FULL_RECEIVE);
```

The chipKIT Max32 will transmit CAN messages on Channel 0 and receive CAN messages on Channel 1. Each Channel is supported by 8 16-byte Message Buffers with the receive Message Buffer able to capture the entire CAN message, which includes a time stamp, the message ID and data payload.

ChipKIT Max32 CAN Module Filters..

Every CAN messages is a broadcast messages. That means every CAN node on the Controller Area Network Bus has the ability to receive every message that is transmitted. For the sake of this demo, we want a CAN node to only accept CAN messages that are addressed to it. We do this by setting up a message acceptance filter. Each CAN SID (Standard ID) message has an 11-bit ID field that we can sift through our filter:

```
CANConfigureFilter      (CAN1, CAN_FILTER0, myaddr, CAN_SID);
```

To set up a similar filter for the CAN2 Engine, simply change CAN12 to CAN1. Since we're only interested in receiving SID messages in this demo application, we will trigger our filter on all 11 bits of the ID and reject and EID (Extended ID) messages:

```
CANConfigureFilterMask  (1 CAN_FILTER_MASK0, 0xFFF, CAN_SID,   CAN_FILTER_MASK_IDE_TYPE);
```

Our first CAN1 filter mask value of 0xFFF covers all 11 bits of the incoming SID message's ID field while the other arguments make sure that nothing but SID messages are allowed to flow to the receive buffer.

CAN_FILTER0 is the first filter we defined and CAN_FILTER_MASK0 is the first filter mask we defined. We can specify up to 32 filters (CAN_FILTER0-CAN_FILTER31) and up to 4 filter masks (CAN_FILTER_MASK0-CAN_FILTER_MASK3). The zero in the names of the filter and masks we coded do not associate them with Channel 0. After all, Channel 0 is our transmit channel. We need to associate the filter and mask we just created to the receive Channel, which happens to be Channel 1:

```
CANLinkFilterToChannel  (CAN1, CAN_FILTER0, CAN_FILTER_MASK0, CAN_CHANNEL1);
```

Now that CAN_FILTER0 and CAN_FILTER_MASK0 are attached to the receive Channel, we can activate CAN_FILTER0:

```
CANEnableFilter         (CAN1, CAN_FILTER0, TRUE);
```

CAN Module Interrupts..

We will monitor the CAN 1 and CAN2 receive activity using interrupt handler routines.

```
CANEnableChannelEvent(CAN1, CAN_CHANNEL1, CAN_RX_CHANNEL_NOT_EMPTY, TRUE);
CANEnableModuleEvent(CAN1, CAN_RX_EVENT, TRUE);
```

The PIC32MX Interrupt Peripheral Library contains the necessary functionality to complete the interrupt definitions:

```
INTSetVectorPriority(INT_CAN_1_VECTOR, INT_PRIORITY_LEVEL_4);
INTSetVectorSubPriority(INT_CAN_1_VECTOR, INT_SUB_PRIORITY_LEVEL_0);
INTEnable(INT_CAN1, INT_ENABLED);
```

Now we can exit Configuration Mode and transition into Normal Operation Mode:

```
CANSetOperatingMode(CAN1, CAN_NORMAL_OPERATION);
while(CANGetOperatingMode(CAN1) != CAN_NORMAL_OPERATION);
```

Transmitting a CAN Message..

The chipKIT Max32 CAN Module will transmit messages that are stacked into a transmit FIFO. However, we can't just throw data into the transmit FIFOs in an ad hoc fashion. To that end, the PIC32MX CAN Peripheral Library has done much of the transmission grunt work for us by setting up transmit message structures, bit fields and logic.

The SID bit field is 11 bits long and lies in the least significant 11 bits of the CMSGID register. The rest of the 32 bits in the CMSGSID area are not used. The CMSGSID bits are be defined within the PIC32MX CAN Peripheral Library in a structure:

```
typedef        struct
{
        unsigned SID:11;     //standard ID field – 0x0-0x7FF
        unsigned :21;        //unused
}CAN_TX_MSG_SID;
```

we won't be sending EID messages in the demo. Howevder, we still need to twiddle some bits in the CMSGEID register. The DLC (Data Length Control) bits specify the size of the data payload section of the CAN packet. The bit to enable or disable RTR is also part of the CMSGEID bit field. Another bit that is important to us is the IDE bit. This bit needs to be clear to indicate SID message transmission. This is the EID parent structure:

```
typedef struct
{
        unsigned DLC:4;      //valid range 0x00-0x08

        unsigned RB0:1;      //reserved - clear to 0
        unsigned :3;

        unsigned RB1:1;      //reserved - clear to 0

        unsigned RTR:1;      //0 = RTR disabled

        unsigned EID:18;     //extended ID field – 0x0 – 0x3FFFF

        unsigned IDE:1;      //clear for SID

        unsigned SRR:1;      //ignored for SID
        unsigned :2;         //unused bits

}CAN_MSG_EID;
```

Now that you know how the PIC32MX CAN Peripheral Library transmit message structures are coded, I think you'll have no problem in interpreting the union that represents a CAN transmit Message Buffer:

```
typedef union {

        struct
        {
      // This is SID portion of the CAN TX message.
                CAN_TX_MSG_SID msgSID;        //32 bits = 1 word

      // This is EID portion of the CAN TX message.
                CAN_MSG_EID msgEID;           //32 bits = 1 word

      // This is the data portion of the CAN TX message.
                BYTE data[8];                 //64 bits = 2 words
        };
```

```
    // This is CAN TX message organized as a set of 32 bit
    // words.
        UINT32 messageWord[4];                  //4 words

}CANTxMessageBuffer;
```

I counted words within the union to show you that the messageWord array can cover all of the words in the structure if you wish it to. The messageWord array can be used to quickly clear the Message Buffer.

To use all of that pretty union and structure code, we've got to point to it. That's easily done and we'll call the pointer to the CANTxMessageBuffer structure *message*:

```
        CANTxMessageBuffer * message;
```

We're not pointing to anything yet. We've only assigned a pointer to the CANTxMessageBuffer structure. So, let's make sure we're pointing at a valid transmit Message Buffer:

```
        message = CANGetTxMessageBuffer(CAN2,CAN_CHANNEL0);
```

Now we're pointing at a Message Buffer in Channel 0, which happens to be our transmit Channel. A NULL returned to *message* means that we don't have a valid Message Buffer in our grasp. If we are truly pointing to a transmit Message Buffer in Channel 0, we can proceed with our transmission process.

This is how the Message Buffer is cleared in the demo code:

```
    if(message != NULL)
    {
        //clear the Message Buffer
        message->messageWord[0] = 0;
        message->messageWord[1] = 0;
        message->messageWord[2] = 0;
        message->messageWord[3] = 0;
```

This code will send a 1 byte message contained the byte 0x41 to address 0x101:

```
        message->msgSID.SID       = node1can1;
        message->msgEID.IDE       = 0;
        message->msgEID.DLC       = 1;
        message->data[0]          = 0x41;
```

We've posted our CAN message in a valid Message Buffer. Before we do anything else, we need to update the Message Buffer's internal pointers and send the message:

```
        CANUpdateChannel(CAN1,CAN_CHANNEL0);
        CANFlushTxChannel(CAN1,CAN_CHANNEL0);
    }
```

Receiving a CAN Message..

The CAN Module interrupt handlers we mentioned earlier are the first to know that a valid CAN message has been received. After the CAN receive interrupt fires, the CAN receive interrupt handler determines what caused the interrupt and branches accordingly. In our case, Channel 1 will be found to be the cause of the interrupt receive event. To prevent the receive interrupt from triggering again before we have time to service the original receive event, we must disable the receive interrupt trigger. We can then inform the application that a CAN message has been received via a flag, clear the receive interrupt flag and return to the application:

```
void __attribute__((vector(47), interrupt(ipl4), nomips16)) CAN2InterruptHandler(void)
{

 if((CANGetModuleEvent(CAN2) & CAN_RX_EVENT) != 0)
 {

        if(CANGetPendingEventCode(CAN2) == CAN_CHANNEL1_EVENT)
        {

           CANEnableChannelEvent(CAN2, CAN_CHANNEL1, CAN_RX_CHANNEL_NOT_EMPTY, FALSE);
                isCAN2MsgReceived = TRUE;
                }
        }

        INTClearFlag(INT_CAN2);


}
```

The CAN receive message algorithm is similar to the CAN transmit message except we are taking instead of giving from a Message Buffer point of view. We still have to assign a pointer to the receive Message Buffer:

```
CANRxMessageBuffer * message;
```

The CAN receive interrupt handler we just examined determined that a valid message had been posted and set the flag isCAN2MsgReceived to TRUE. So, we can clear the isCAN2MsgReceived flag and obtain the address of the newly received CAN message:

```
if(isCAN2MsgReceived == FALSE)
{
        return;
}

isCAN2MsgReceived = FALSE;

message = CANGetRxMessage(CAN2,CAN_CHANNEL1);
```

Now that we have access to the receive Message Buffer that contains the incoming data payload, we can assess the data payload using a pointer to the members of the pointed-to structure, which is in this case CANRxMessageBuffer:

```
if (UARTTransmitterIsReady(UART1))
        {
                UARTSendDataByte(UART1, message->data[0]);
        }
```

After redirecting the received byte of CAN data to the chipKIT Max32's FT232RQ-based USB portal, we update the receive Message Buffer's internal pointers and enable the receive interrupt trigger with this code:

```
CANUpdateChannel(CAN1, CAN_CHANNEL1);
CANEnableChannelEvent(CAN1, CAN_CHANNEL1, CAN_RX_CHANNEL_NOT_EMPTY, TRUE);
```

We're ready to receive and process the next CAN message.