AN2714

MPLAB® Code Configurator CAN 2.0B Module for PIC18 Microcontrollers

Introduction

Authors: Mary Tamar Tan, William Stuart, Microchip Technology Inc.

Microchip's MPLAB® Code Configurator (MCC) Enhanced CAN (ECAN™) module provides an easy-to-use GUI for users who want to speed up the development of their PIC18 Controller Area Network (CAN) projects. The main advantage of using the ECAN MCC module is that it eliminates the trouble of manually calculating the individual acceptance masks, filter and Configuration register values. The user can add and remove CAN Rx IDs easily without worrying about how to configure the ID-Filter-Mask-Buffer combinations in the PIC18F registers. The ECAN MCC module can help designers save significant time from the low-level CAN protocol implementation and focus on the development of higher level applications.

The ECAN MCC module supports PIC18F 8-bit devices that contain the on-chip ECAN peripheral. One such family of devices is the PIC18FXXK8X. The module supports the three ECAN functional modes: Legacy (mode 0), Enhanced Legacy (mode 1) and Enhanced FIFO (mode 2). The MCC-generated APIs depend on the mode specified by the user in the GUI. The PIC18F functional modules configure the ECAN module's hardware to different settings, allowing flexibility in the module to meet the needs of various users.

This document focuses on the features and implementation of the ECAN MCC module. It is assumed that the reader has basic knowledge about the CAN protocol as well as the Enhanced Controller Area Network (ECAN) peripheral on 8-bit PIC® MCUs. For consistency, this application note refers to the ECAN MCC software library as 'ECAN MCC module' and the hardware ECAN module as 'ECAN peripheral'.

Table of Contents

1. Features

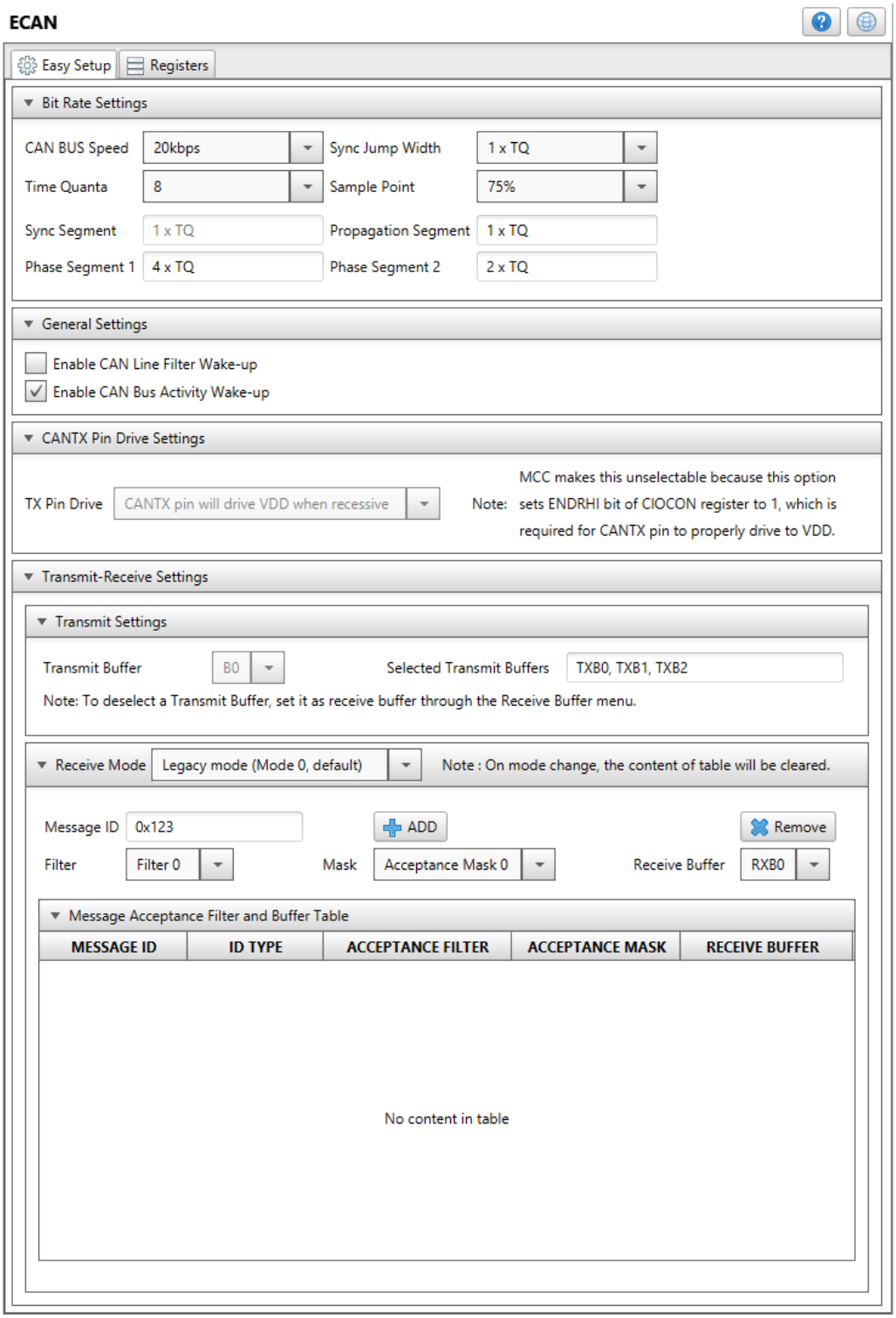
The ECAN MCC module complements the hardware ECAN peripheral by providing the following features:

- Selectable functional modes of operation between mode 0, mode 1 and mode 2
- Automatic calculation of Baud Rate Control Register values based on the user-provided CAN bus speed and device clock
- Automatic bit segment calculation based on selected TQ multiplier and sample point
- Automatic calculation of filter and mask values
- A tabulated summary of entered CAN ID, ID Type, Mask, Filter and Buffer combinations
- Verbose comments in the generated code that details the ECAN module's configuration

2. ECAN Module GUI

The default ECAN module GUI is shown in the figure below.

Figure 2-1. ECAN™ GUI Default View



2.1 Easy Setup

The **Easy Setup** tab allows the setting of different CAN bus parameters, enabling and disabling of CAN wake-up features, and functional mode selection.

2.1.1 CAN BUS Settings

- **CAN BUS Speed** – Provides the user the ability to select CAN Nominal Bit Rates (NBR) from 20 kbps to 1 Mbps.

Note: To enable multiple nodes to communicate on the same CAN BUS, the bit rate for all nodes on the CAN BUS should be identical.

- **Time Quanta (TQ)** – Provides selection of valid TQ multipliers. When a value is selected, the GUI automatically updates with valid sample point options.
- **Sync Segment** – This field cannot be modified by the user. It is shown for completeness in detailing the bit rate TQ breakdown. The Sync Segment is always equal to 1xTQ. The Sync Segment's purpose is to synchronize the various nodes on the bus.
- **Propagation Segment** – The Propagation Segment exists to compensate for physical delays between nodes. This field can take a value between 1xTQ to 8xTQ.
- **Phase Segment 1 (PS1)** and **Phase Segment 2 (PS2)** – The two phase segments are used to compensate for edge phase errors in the bus. PS1 can be lengthened or PS2 can be shortened by resynchronization. The allowable value for PS1 and PS2 is from 1xTQ to 8xTQ.
- **Sync Jump Width** – Provides options for the Synchronization Jump Width (SJW) from 1xTQ to 4xTQ. SJW is an integral multiple of TQ that defines the maximum value of lengthening and shortening of a bit's length during resynchronization. The SJW adjusts the bit clock as necessary to maintain synchronization with the transmitted message. To learn more on SJW refer to the data sheet or application note AN00754, "Understanding Microchip's CAN Module Bit Timing".
- **Sample Point** – Provides the actual sample point position for one bit period. The Sample Point is the point of time at which the bus level is read and interpreted as the value of that respective bit. Bit sampling takes place between PS1 and PS2.

2.1.2 General Settings

- **CAN Activity wake-up** – This check box enables/disables the CAN bus activity wake-up feature. Enabling this option allows the peripheral to generate an interrupt when activity is detected on the CAN bus while in internal Sleep mode.
- **CAN Line Filter wake-up** – This check box enables/disables the use of a low-pass filter function to the RXCAN input line while in internal Sleep mode. This prevents the device from waking up due to noise or short glitches in the bus.

2.1.3 CANTX Pin Drive Settings

By default, the ECAN MCC module sets the CANTX pin to drive V_{DD} when recessive. This configuration is required to allow the CANTX pin to drive to V_{DD} properly.

Note: This section is only available for devices with Enable High Drive (ENDRHI) bit in the CAN I/O Control (CIOCON) register such as the PIC18FXXK80.

2.1.4 Transmit-Receive Settings

2.1.4.1 Transmit Settings

- **Transmit Buffer** – Shows all generic ECAN message buffers that are programmable to be used as transmit or receive buffers. When the user selects a generic message buffer within MCC then it is

automatically set as a Tx buffer and the **Selected Transmit Buffers** text field is updated. This field is disabled in mode 0, because functional mode 0 does not support the generic buffers which could be configured into transmit buffers.

- **Selected Transmit Buffers** – Shows all the buffers that will be used for transmit operation. By default, only the dedicated transmit buffers are displayed.

2.1.4.2 Receive Mode

Mode selection allows the user to select between Legacy mode (mode 0, default), Enhanced Legacy mode (mode 1) and Enhanced FIFO mode (mode 2). See section 3. Supported Modes for more information.

The user should manually enter a valid CAN ID and select the associated Acceptance Filter, Acceptance Mask and Receive Buffer for that ID. The ECAN MCC module has an ID validator feature designed in that will only accept valid CAN receive IDs in hexadecimal format. This feature flags IDs that have invalid characters (i.e., $, %, z), IDs that are out of range, and IDs already present in the Message Acceptance Filter and Buffer Table. After valid ID entry, the user should click the **Add** button which will append a row with the ID, ID Type, Mask-Filter-Buffer linkage to the Message Acceptance Filter and Buffer Table.
Note: A valid Standard ID is any hexadecimal value in the range of '0x00-0x7FF'. A valid Extended ID is any hexadecimal value in the range of '0x-1FFFFFFFx', where the appended 'x' correspond to 'extended' CAN identifiers.

The ECAN MCC module also does not allow changes to individual table cells. To modify, the user needs to remove the entire row and add another row with the correct values. The ECAN module, however, allows remapping of all entries for Acceptance Filter to a new Receive buffer.

In mode 2, the user can select the Rx FIFO size. All programmable buffers that are not part of the FIFO are automatically configured as transmit buffers.

2.2 Registers

The **Registers** tab shows the different ECAN registers with their corresponding values based on the parameters set by the user in the **Easy Setup** tab. The module also assigns default values based on the specific device data sheet to registers that are not affected by the Easy Setup configuration. The register values as well as the setting for each of the register bits are displayed in the GUI. Some of the register bits settings can be modified by the user through the corresponding bit's choice box. The user is advised to consult the specific device data sheet when modifying register values.

It is also under the **Registers** tab that the user can enable and disable interrupt features.

Note: The MCC has a **Notifications** tab that lists all the configuration warnings. Usually, this shows notifications from all peripheral modules and libraries selected by the user, including the ECAN module. The user *must* act on these warnings to avoid generation of incorrect code.

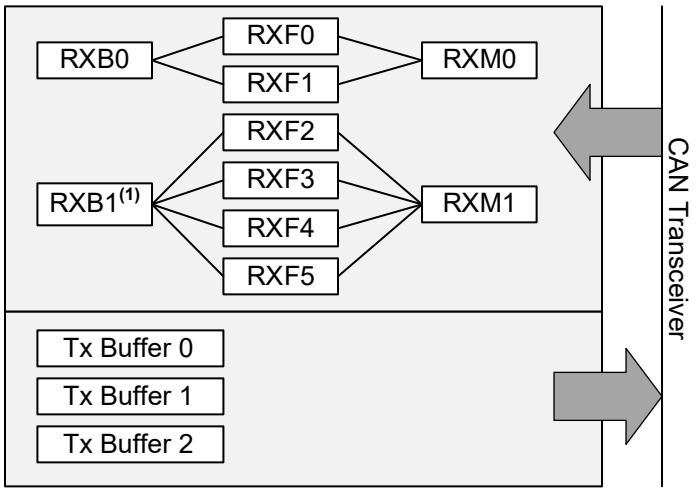
3. Supported Modes

The ECAN MCC module supports all three hardware ECAN functional modes.

3.1 Mode 0 – Legacy Mode

In mode 0, only the CAN ID and filter can be set by the user. There are three transmit buffers, two receive buffers, two acceptance masks, and six acceptance filters. Filters 0 and 1, and Acceptance Mask 0, are associated with RXB0. Filters 2, 3, 4 and 5, and Acceptance Mask 1 are associated with RXB1. Refer to the figure below.

Figure 3-1. Mode 0 Buffer-Filter-Mask Association

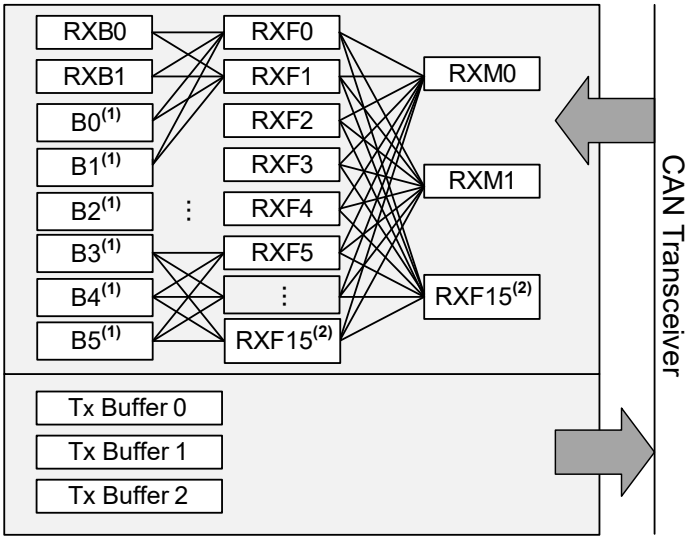


NOTE 1: RXB0 can overflow into RXB1

3.2 Mode 1 – Enhanced Legacy Mode

Compared to mode 0, mode 1 has ten additional acceptance filters, creating a total of 16 available filters, as shown in Figure 3-2. Filter 15 can either be used as an acceptance filter or acceptance mask. There are also six additional buffers (B0-B5) that can be programmed as either Tx or Rx. By default, these buffers are configured as receive buffers. Each of these acceptance filters can be dynamically associated with any of the receive buffers. Each filter can also be dynamically associated with available Acceptance Masks.

Figure 3-2. Mode 1-Buffer-Filter-Mask Association

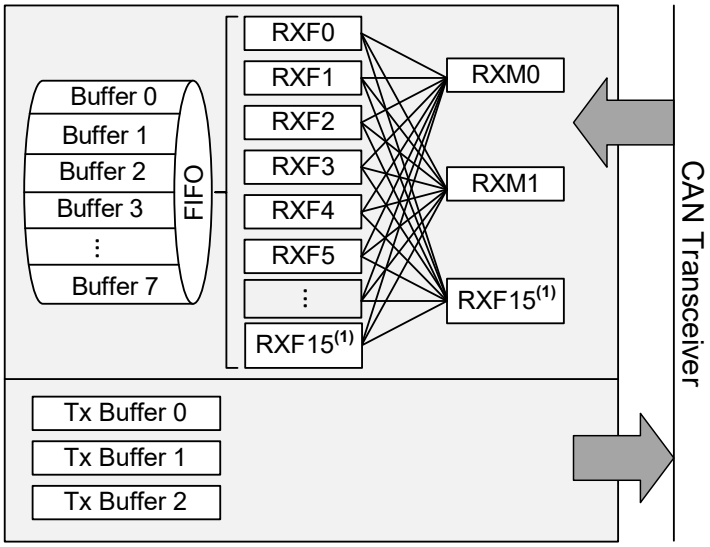


NOTE **1:** This can be configured as a Tx or a Rx buffer.
2: RXF15 can be used as a mask or a filter.

3.3 Mode 2 – Enhanced FIFO Mode

In mode 2, there is no one-to-one relationship between the receive buffer and acceptance filter registers. Any selected filter that is linked to the FIFO receive buffer can generate acceptance and cause FIFO to be updated. Two of the dedicated receive buffers in combination with one or more programmable transmit/receive buffers are used to create a maximum of eight buffers deep FIFO.

Figure 3-3. Mode 2 Buffer-Filter-Mask Association



NOTE **1:** RXF15 can be used as a mask or a filter

4. APIs

The generated APIs will vary based on the user selected ECAN functional mode. The APIs are composed of global and local functions. Table 4-1 provides the global functions available in all modes. Local functions are mode-specific and are not advised to be modified by the user.

Table 4-1. MCC-Generated Global Functions For ECAN™

Function Name	Parameters	Returns	Description
`ECAN_Initialize()`	—	—	This routine sets all the ECAN™ module register (filter, mask, and timing) values based on the set parameters in the GUI. Puts the ECAN™ in Configuration mode then switch to Normal mode after initializing the ECAN™ registers.
`CAN_sleep()`	—	—	This routine enables the wake-up from bus activity feature before putting the ECAN™ to Sleep mode.
`CAN_transmit()`	`*tempCanMsg`	`True` if message was loaded to transmit buffer. Otherwise, returns `False`.	Looks for an empty transmit buffer based on prioritization. Sets the Transmit Request Status (TXREQ) bit after writing the converted raw ID, DLC, and eight data bytes values to the corresponding registers.
`CAN_receive()`	`*tempCanMsg`	`True` if a new message is received. Otherwise, returns `False`.	Checks the CAN buffers for received message. If a valid message is received, the ECAN™ register values are copied to temporary software registers. The ID register values are automatically converted to raw ID.
`CAN_messagesInBuffer()`	—	Total number of messages in the buffers.	Checks for the number of messages in the buffer.

Function Name	Parameters	Returns	Description
CAN_isBusOff()	—	True if module is in Bus-off. Otherwise returns False.	Checks if module is in Bus-off mode.
CAN_isRXErrorPassive()	—	True if module is in Rx Error Passive. Otherwise returns False.	Checks if module is Rx Error Passive.

Note: MCC generates the APIs but the user is required to add the application code.

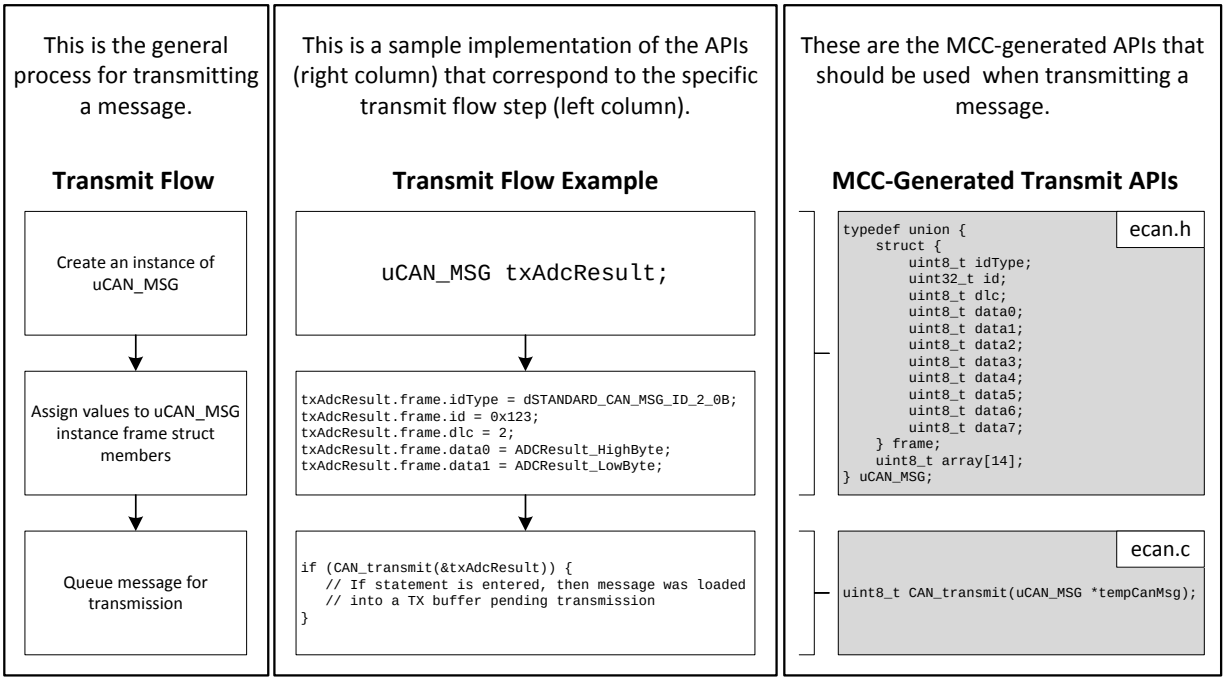
4.1 Sample Implementation of CAN Transmit and Receive Functions

The ECAN ID, ID Type and data bytes are arranged within a union of 'uCAN_MSG' type. Every union declared as 'uCAN_MSG' is an instance of this type. Each member of an instance is accessed using the *<instance_name>.<struct_name>.<member_name>* format. To better understand the implementation, two examples are provided below for transmitting and receiving a CAN message.

4.1.1 Transmitting a Message

The following figure shows how to transmit a single message with standard ID '0x123', and two bytes of data: the high byte and low byte result of an ADC.

Figure 4-1. CAN Message Transmission



The 'CAN_transmit()' function accesses the value of its argument via a 'uCAN_MSG' type pointer. It uses a dereference 'uCAN_MSG' type pointer (denoted by the dereference operator: '*') as parameter, hence it takes a referenced 'uCAN_MSG' type union as an argument (denoted by the address of operator: '&').

In this example, the address of txAdcResult ('`&txAdcResult`') is used as argument to allow the CAN transmit function to access each member of '`txAdcResult`' utilizing the *txAdcResult.frame.<member_name>* format. The function copies the value of each member to its corresponding ECAN peripheral register.

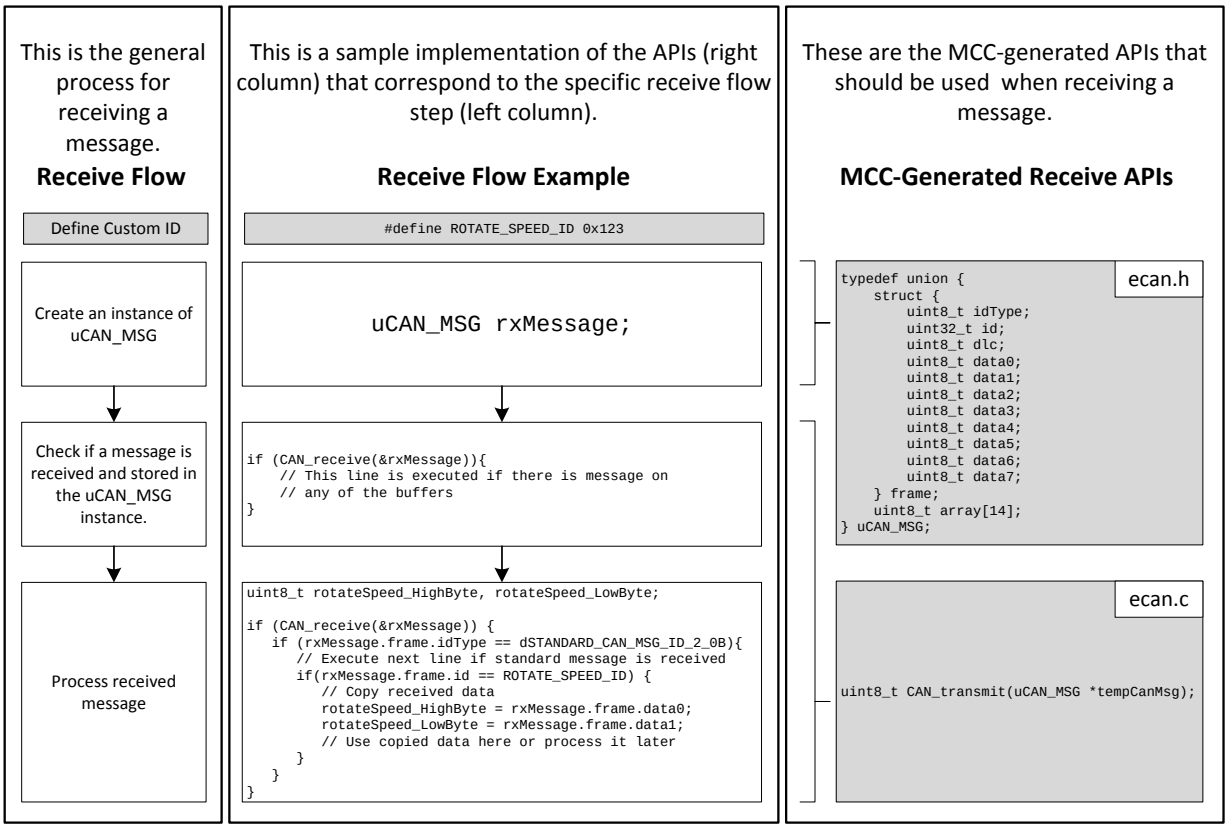
The user can verify if a message was successfully queued for transmission by checking if the function returns a value of '`1`', otherwise, the message is lost and might need retransmission.

Calling the '`CAN_transmit()`' function does not necessarily mean that the message is sent immediately to the bus. The message is simply queued for transmission through the next available transmit buffer based on priority. The ECAN module will only send out the message when the higher priority transmit buffers are empty.

4.1.2 Receiving a Message

The following example shows how to receive the message transmitted in the previous example. Here, standard ID '0x123' is defined beforehand using a macro.

Figure 4-2. CAN Message Reception



Similar with the transmit function, the '`CAN_receive()`' function accesses the value of its argument via a '`uCAN_MSG`' type pointer. In this example, the argument – which is the address of the '`rxMessage`' – is written in the format '`&rxMessage`'. The use of pointer allows the value of each member of '`rxMessage`' to be modified inside the function.

In the example above, '`rxMessage`' is declared as a '`uCAN_MSG`' type union. The actual received ID, ID type, data length and data bytes of the message are transformed by the CAN receive function to the appropriate data types and stored as members of '`rxMessage`'.

The user can then process the data by accessing the data members of '`rxMessage`' using the *rxMessage.frame.<member_name>* format. In this example, the two 8-bit data members are copied to two locally defined 8-bit variables: '`rotateSpeed_HighByte`' and '`rotateSpeed_LowByte`'.

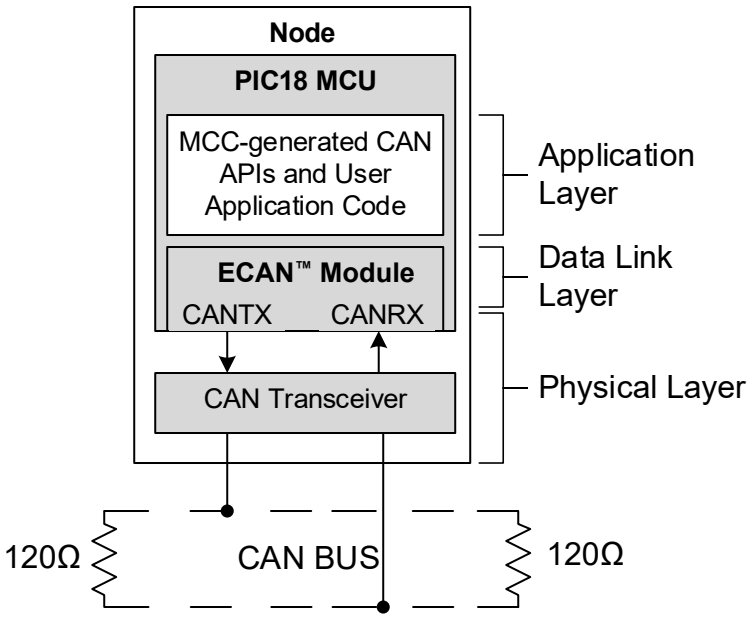
The user can declare as many instances of '`uCAN_MSG`' as the available data memory would allow. Careful allocation might be necessary for large programs since each instance occupies 14 bytes of memory.

5. CAN Node Using PIC18 ECAN

Figure 5-1 shows a typical node connection to the bus using a PIC18 MCU with embedded CAN controller. MCC-generated APIs and their implementation in the user application code falls in the OSI model's application layer.

The embedded CAN protocol controller (ECAN module) consists of a protocol engine and message buffering and control. The ECAN module uses dedicated CANTX and CANRX pins to interface with a CAN transceiver. The CAN transceiver then translates the digital signals from the controller to differential output signals suitable for transmission over the bus.

Figure 5-1. CAN NODE Using PIC18 with ECAN™ Module



For a complete list of CAN transceivers from Microchip visit http://www.microchip.com/design-centers/can. A detailed discussion about the CAN physical layer is also provided in AN228, "A CAN Physical Layer Discussion" (DS00228).

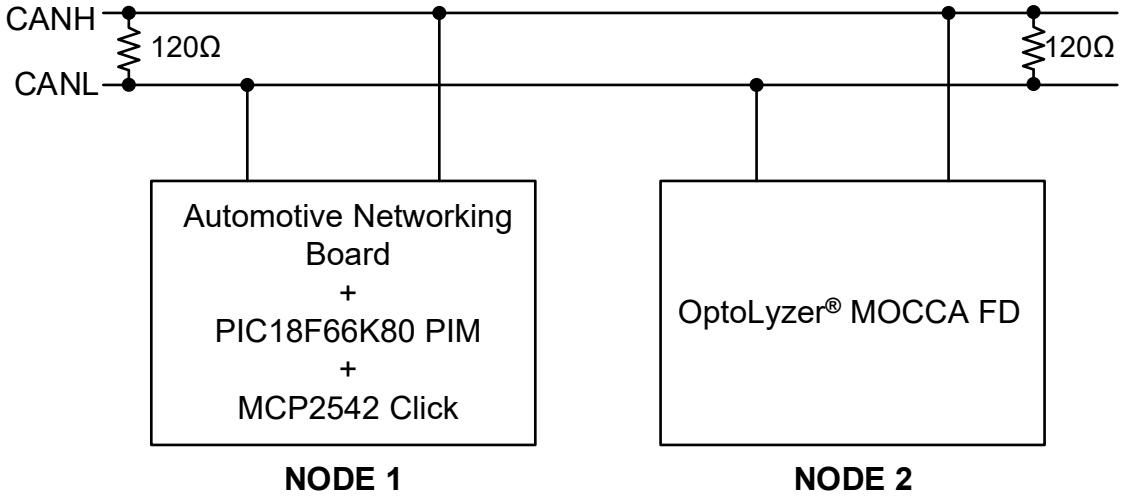
6. Demo

This demo is composed of two parts. The first example shows a network with two nodes. In the second example, a third node will be added to the network.

6.1 Two-Node Network Demo

This section shows implementation of the ECAN MCC module to generate the driver for Node 1 of the CAN network shown in the figure below.

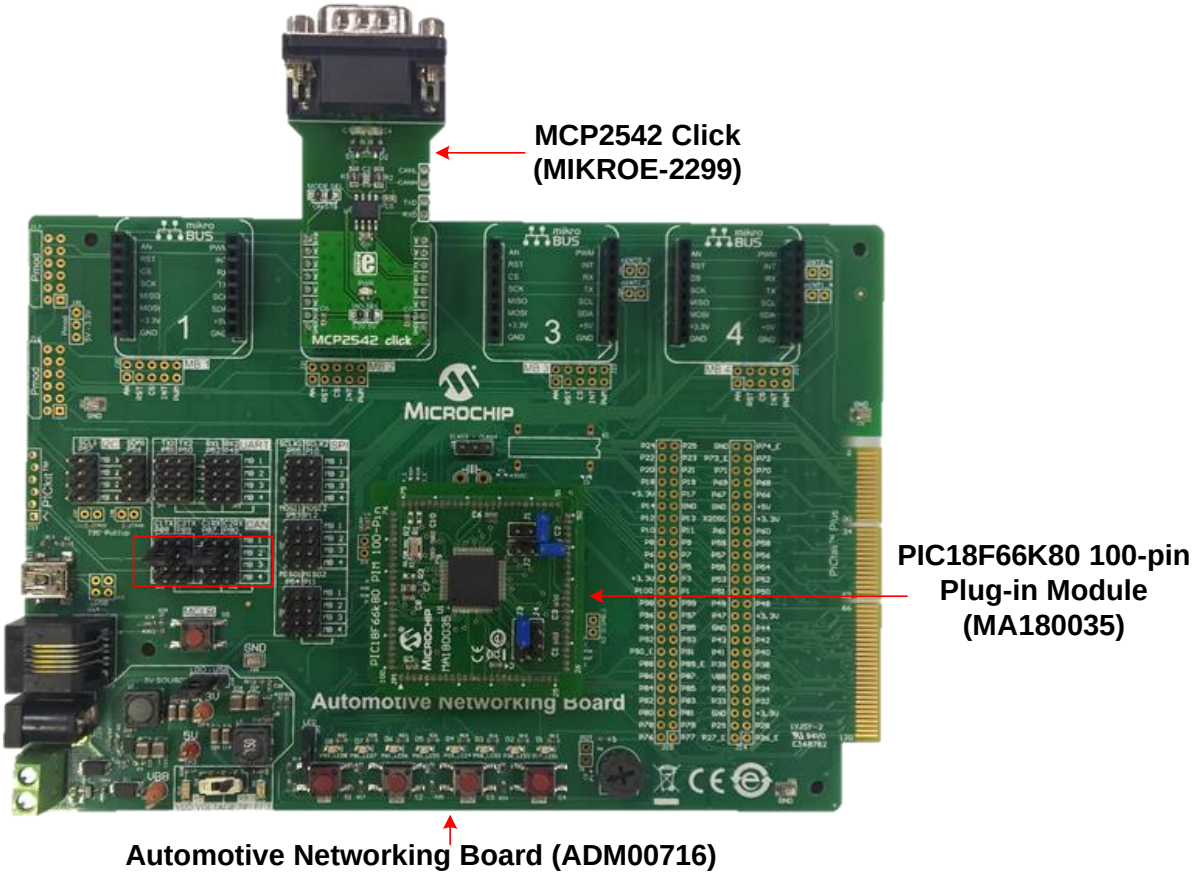
Figure 6-1. Two-Node CAN Network



Node 1

The first node is comprised of an Automotive Networking Board with a PIC18F66K80 100-pin PIM and an MCP2542 CAN Transceiver Click Board™. The Automotive Networking Board has provision for up to four click boards. For this example, MCP2542 click is plugged in mikroBUS™ socket 2 with jumpers positioned as shown in the following figure.

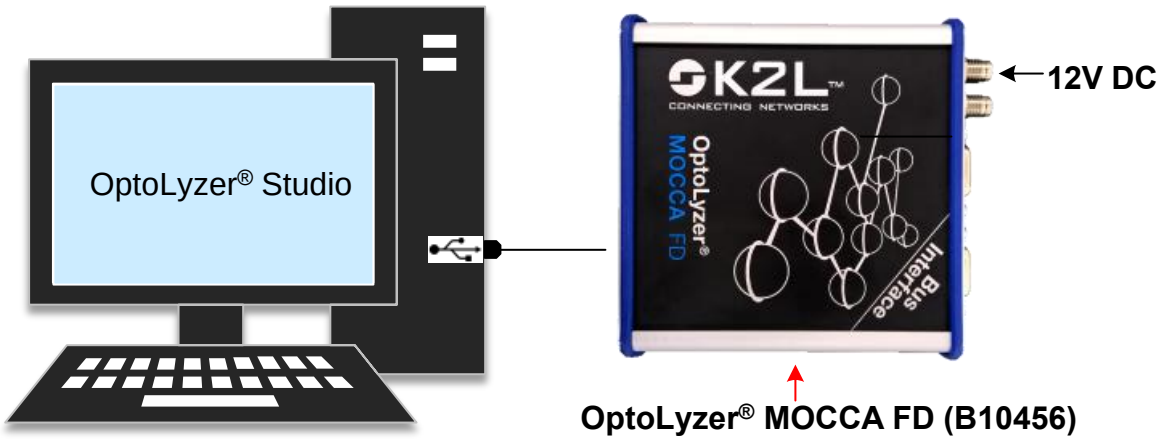
Figure 6-2. Node 1 - PIC18F66K80 and MCP2542



Node 2

The MOCCA FD from K2L is used as the second CAN node (see figure below). The MOCCA FD is interfaced to a PC. The OptoLyzer® Studio software application is used to display real-time messages transferred between all nodes connected on the bus.

Figure 6-3. Node 2 - MOCCA FD Tool Box



6.1.1 Two-Node Network Details

The following figure shows the network details for the individual messages expected on the CAN bus for the two-node network demo.

Figure 6-4. Two-Node CAN Network Details

Msg ID	Dir.	Type of Msg	DLC	Data								Message TX Trigger	Details
				DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7		
Node 1 (PIC18F66k80 + Automotive Networking Development Board)													
0x100	TX	Periodic	8	ALL TX Msg Cnt				ALL TX Periodic Msg Cnt				500 msec	
0x110	TX	Periodic	4	ALL RX Msg Cnt								500 msec	
0x120	TX	Periodic	4	Potentiometer								300 msec	
0x12345x	TX	Periodic		SW1 State	SW2 State	SW3 State	SW4 State					100 msec	Switch state Press == 1 or Released == 0 is transmitted
0x150	TX	Event	4	SW0 Count	SW1 Count	SW2 Count	SW3 Count					Press/release of any switch	Counts show how many times the buttons were pressed and released
0x160	TX	Event	4	0x160 Msg Cnt								Press/release of SW4	
0x200	TX	Event	2	Major Ver	Minor Ver							Response to message 0x700	Provides BUS the Application Software Version
0xD34FFx	RX		1	Set LED									Sets LEDs for this node. Maps the lower 5 bits to LED5 to LED1.
0x201FFx	RX		1	Clear Counters									Clears ALL message Counters
0x700	RX		1	Request Ver									Get software version
Node 2 (CAN Tool)													
0x700	TX	Event	1	Request Ver									
0x201FFx	TX	Event	1	Clear Counters									
0xD34FFx	TX	Event	1	Set LED									
All IDs	RX												The Tool will receive all messages on the network

The table shows the CAN message ID, direction (TX or RX), type (periodic or event-triggered), DLC, data bytes, message trigger (if TX), and the more specific details for each message that should be taken care of in the application software. This demo teaches how to configure the PIC18F66K80 ECAN module and the MOCCA FD tool to communicate with each other using the messages provided in the table.

6.1.2 ECAN MCC System Requirements

- MPLAB X IDE v4.15 or newer
- MPLAB Code Configurator (MCC) Core v4.45 or newer
- MPLAB Code Configurator (MCC) PIC10/PIC12/PIC16/PIC18 Library v1.65.2 or newer
- Java JRE v1.8 or newer (follow MCC release note to setup MPLAB X for latest Java)

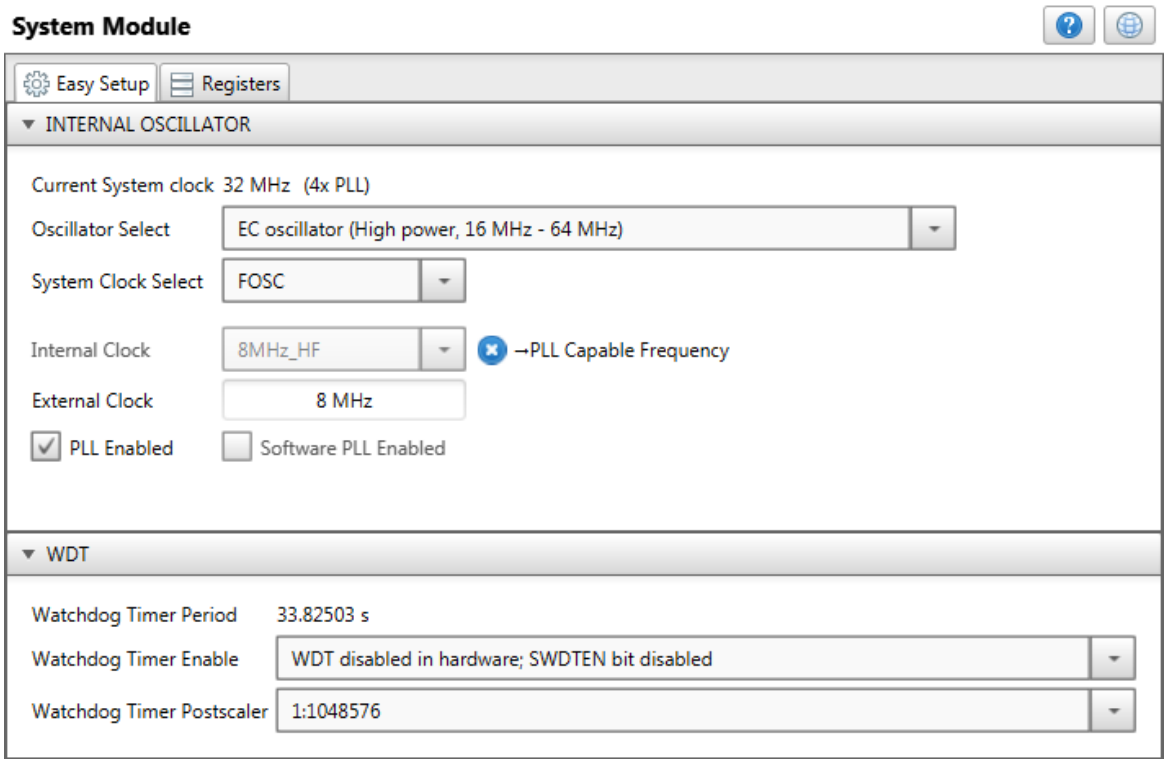
6.1.3 MCC Configuration Settings

The following steps will walk the user through the process of implementing the ECAN module from configuration to code generation for Node 1.

Note: GUI appearance for the different modules may vary per each MCC Core, Plugin or Library release. The screen shots used in this document were taken using MCC v3.45 and ECAN MCC v2.0.

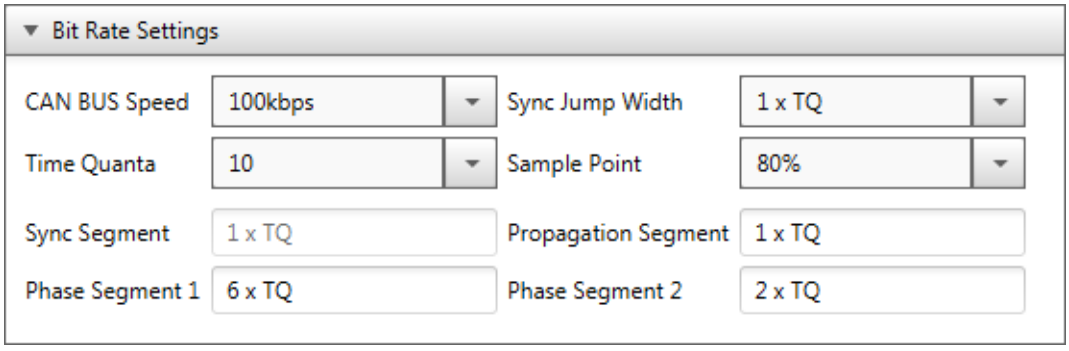
1. In MPLAB X IDE, click on the MCC icon or navigate to*Tools>Embedded>MPLAB Code Configurator v3: Open/Close* to launch MCC version 3.
   Note: Newer MCC versions (v3.55 and up) will ask the user to input the Configuration name in `.mc3` file format soon after launching MCC. If so, input “Node1” as the file name then click **Save**.
2. Under Project Resources, click **System** and select **System Module** (see Figure 6-5).
3. In the **Easy Setup** tab of the System Module, set **Oscillator Select** to 'EC oscillator (High power, 16 MHz–64 MHz)'. Set **System Clock** to 'FOSC' and **External Clock** to '8 MHz'. Enable PLL and disable the Watchdog Timer.

Figure 6-5. PIC18F66K80 System Module



4. Under Device Resources, select *Peripherals>CAN>ECAN*. The ECAN module should automatically move to the Project Resources window.
5. Go to the ECAN module GUI.
6. In the CAN BUS settings, set **CAN BUS Speed** to '100 kbps' and **Time Quanta** to '10'. Notice that once the Time Quanta is set, the different segments are also populated with the recommended values. See figure below.

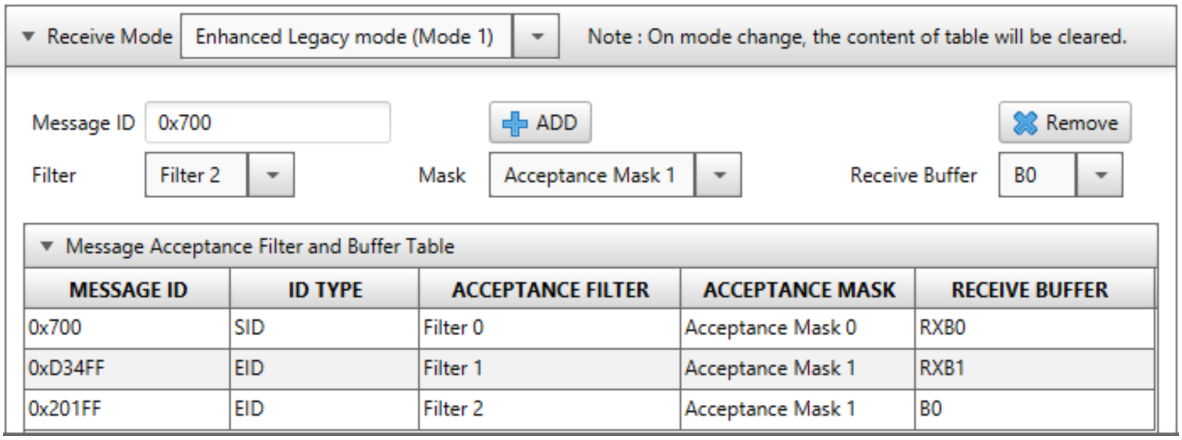
Figure 6-6. CAN Bus Settings



7. Set the **Mode** to 'Enhanced Legacy mode (Mode 1)'.
8. Edit the CAN ID Masks and Filters as shown in Figure 6-7. To remove a row, click anywhere within the row that needs to be to removed and click **Remove** button. To add another row, input the ID on

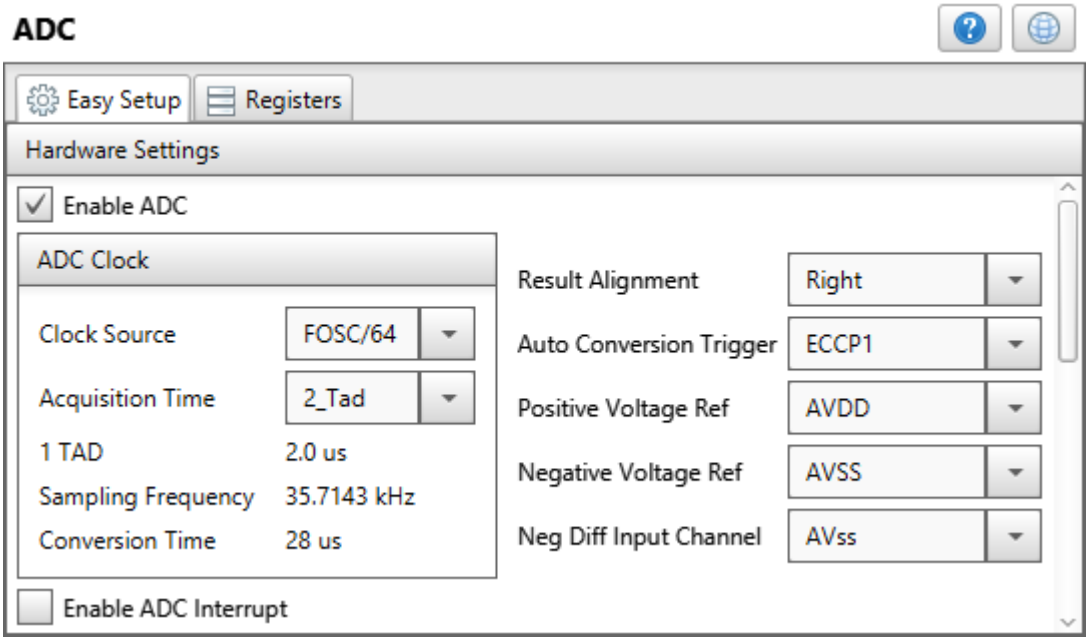
the ID text field, select the corresponding **Acceptance Mask**, **Filter** and **Buffer**, then click **Add**. The Extended IDs should be appended with an 'x'. For example, Extended 0xD34FF should be inputted as 0xD34FFx.

Figure 6-7. Receive Mode Selection and CAN Table



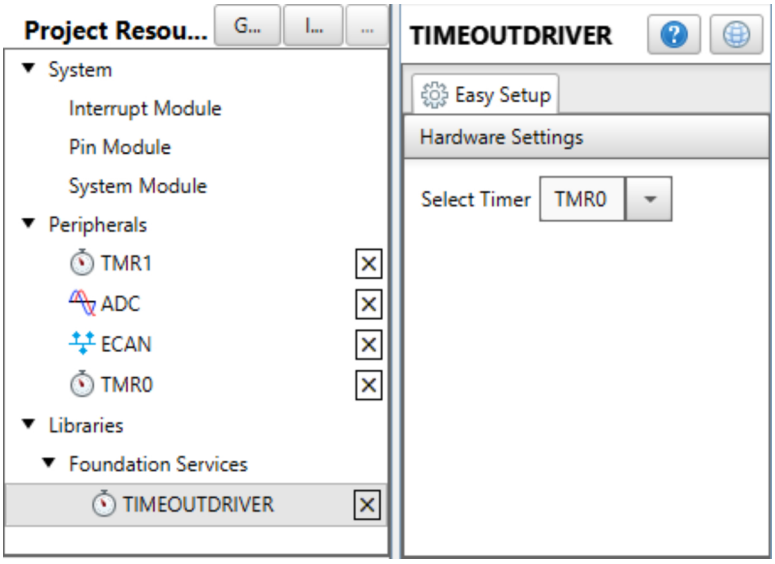
9. Go to **Device Resources** and select **ADC**. Edit the **ADC** GUI as shown in the following figure.

Figure 6-8. ADC Module



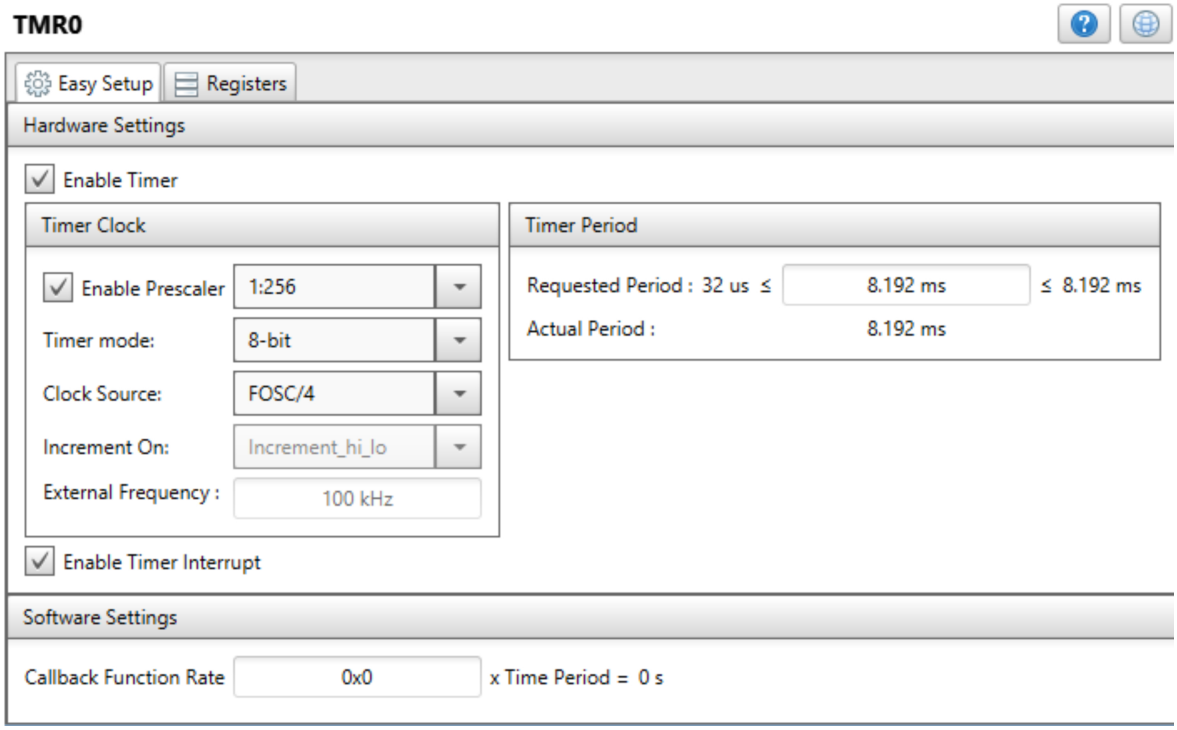
10. Under **Device Resources**, go to *Libraries>Foundation Services>TIMEOUTDRIVER*. The TIMEOUTDRIVER and TMR0 will be automatically added to **Project Resources**. The TIMEOUTDRIVER will be used to create the timer callbacks for periodic ECAN TX messages, based on the TMR0 tick.

Figure 6-9. Time-Out Driver Module



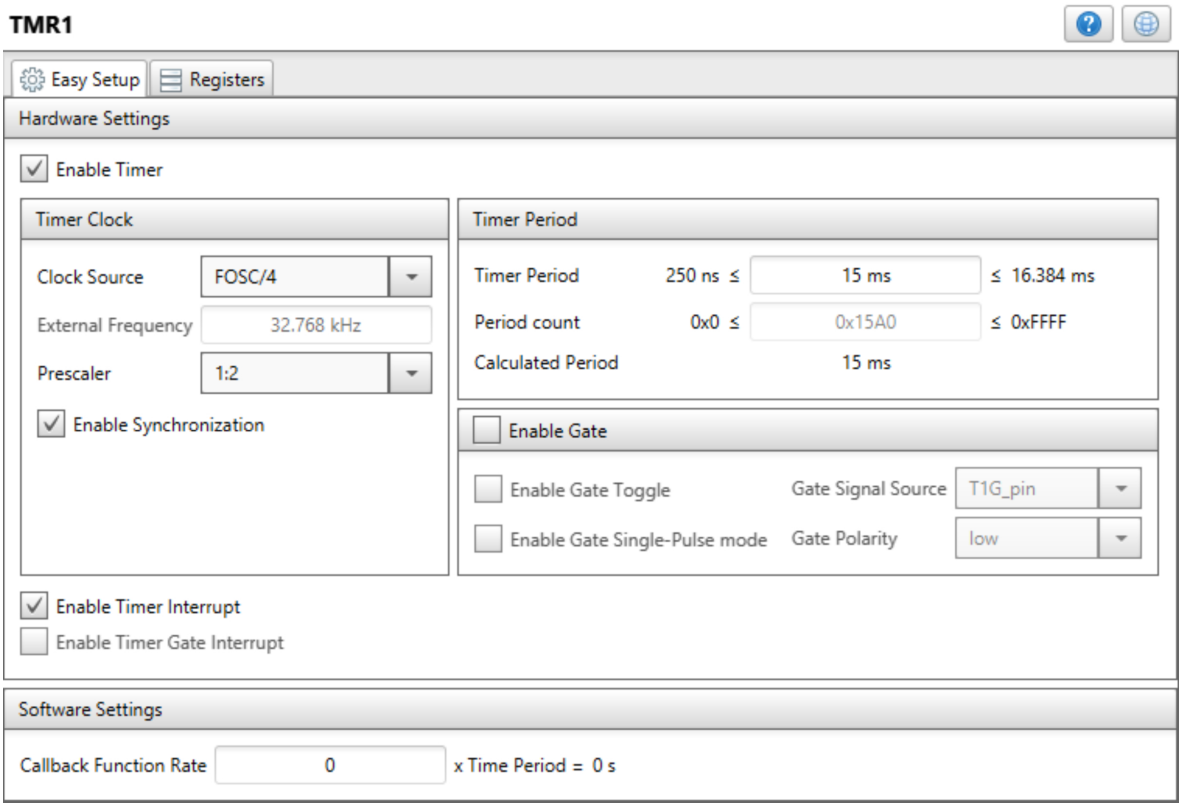
11. Configure TMR0 as shown below.

Figure 6-10. TMR0 Module



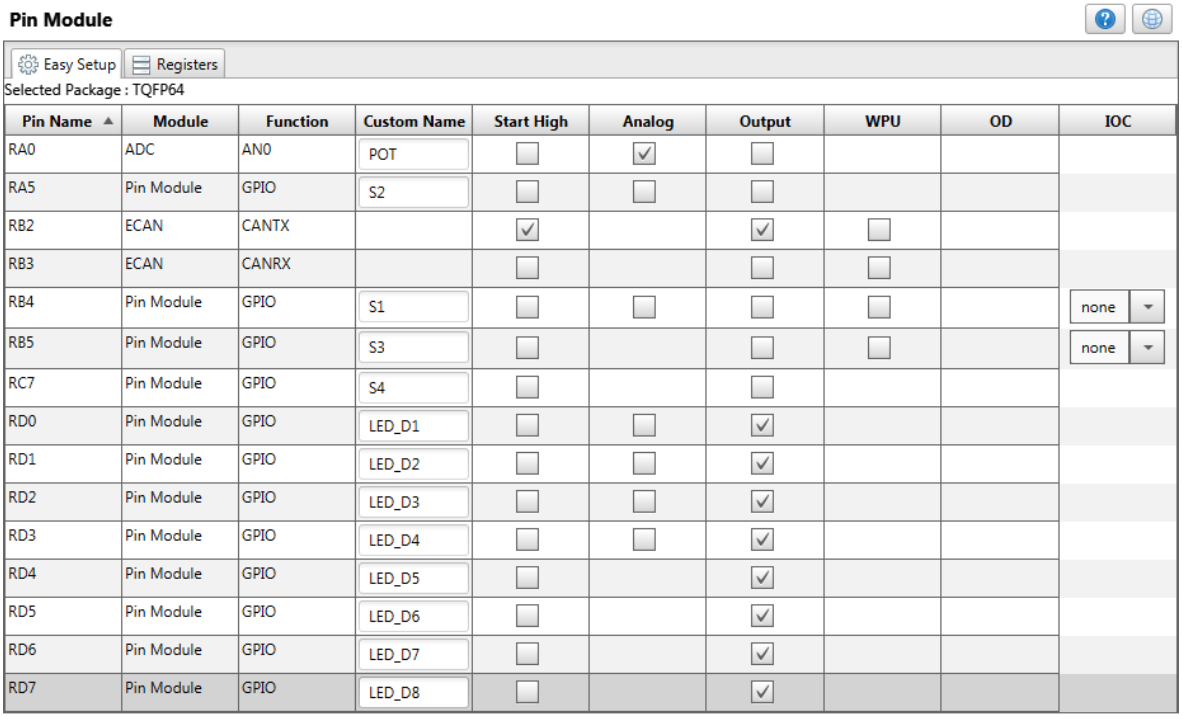
12. In this example, Timer1 is used to debounce the switches by delayed sampling. Select TMR1 and configure as shown below.

Figure 6-11. TMR1 Module



13. Go to Project Resources and select **Pin Module**. Make sure that the pins are configured similar to Figure 6-12.

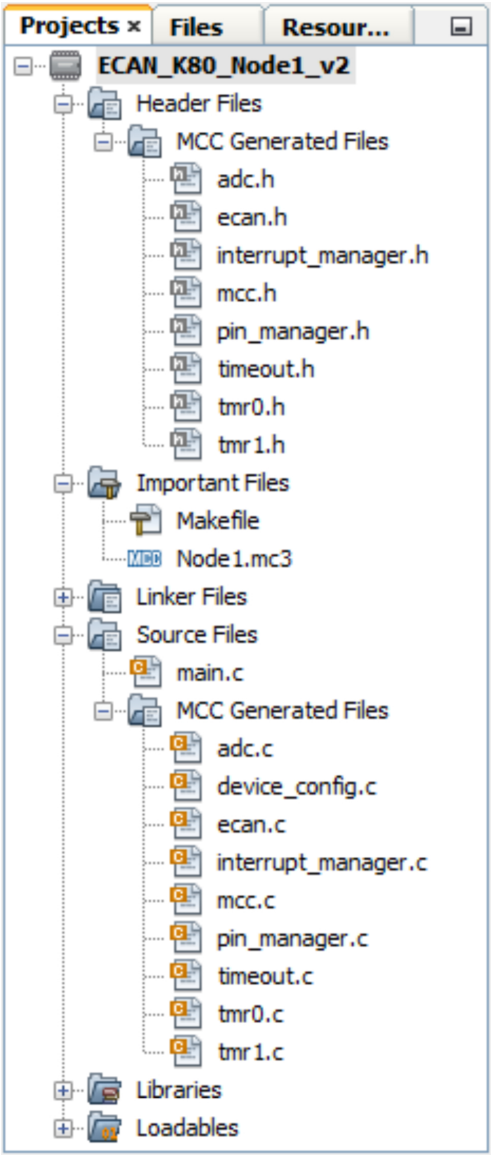
Figure 6-12. Pin Module



Pin Name	Module	Function	Custom Name	Start High	Analog	Output	WPU	OD	IOC
RA0	ADC	AN0	POT	☐	☑	☐			
RA5	Pin Module	GPIO	S2	☐	☐	☐			
RB2	ECAN	CANTX		☑		☑	☐		
RB3	ECAN	CANRX		☐		☐	☐		
RB4	Pin Module	GPIO	S1	☐	☐	☐	☐		none
RB5	Pin Module	GPIO	S3	☐		☐	☐		none
RC7	Pin Module	GPIO	S4	☐		☐			
RD0	Pin Module	GPIO	LED_D1	☐	☐	☑			
RD1	Pin Module	GPIO	LED_D2	☐	☐	☑			
RD2	Pin Module	GPIO	LED_D3	☐	☐	☑			
RD3	Pin Module	GPIO	LED_D4	☐	☐	☑			
RD4	Pin Module	GPIO	LED_D5	☐		☑			
RD5	Pin Module	GPIO	LED_D6	☐		☑			
RD6	Pin Module	GPIO	LED_D7	☐		☑			
RD7	Pin Module	GPIO	LED_D8	☐		☑			

14. Check the **Notifications[MCC]** tab for any warnings. 'Severe' type notifications should always be resolved by the user on their respective modules.
15. Next to Project Resources, click the **Generate** button.
    Note: If you haven't set the MCC configuration file name after launching MCC, a pop-up window will appear asking for the configuration settings to be saved in a `.mc3` file format. Input "Node1" as the file name, then click **Save**.
16. The MCC configuration is now complete. Figure 6-13 shows all the files that will be added to the user's project upon code generation.

Figure 6-13. MCC Generated Files



6.1.4 Application Code

Edit the main file as shown in example below.

Example 6-1. PIC18F66K80 Main File

```
#include "mcc_generated_files/mcc.h"
#define MilliSeconds    (uint32_t)31    // 32us=1 tick, this means 31 ticks=1ms
#define PRESSED         1
#define NOT_PRESSED     0
#define MASK_32BITS     0x000000FF
#define MASK_8BITS      0x01
#define MAJOR_VERSION   1
#define MINOR_VERSION   0
#define REQ_VERSION     1
#define CLEAR_COUNTERS  0

typedef struct switchStat{
    bool status;
    uint8_t port;
```

```
    uint8_t pressCount;
    uint8_t state;
    uint8_t sampleLevel;
}switchStat_t;

// Function prototypes for periodic messages and heartbeat
void add_PeriodicMessages(void);
uint32_t send_TxRxMsgCount(void *payload);
uint32_t send_PotStatus(void *payload);
uint32_t send_SwitchStatus(void *payload);
void send_TxMsgCount(void);
void send_RxMsgCount(void);
void heartbeat(void);

// Function prototypes for event-triggered messages
bool checkButton(switchStat_t *switchNumber, uint8_t port);
void send_SwitchPressCount (void);
void send_Switch4PressCount (void);
void send_softwareVersion (void);

// Function prototypes for receive message
void processRxMessage(void);
void LED_Display (uint8_t ledStat);
void clear_Counters (void);

// Local variables
switchStat_t s1, s2, s3, s4;
uint32_t rxMessageCount = 0;
uint32_t txMessageCount = 0;
uint32_t txPeriodicMessageCount = 0;
uint32_t s4MsgCount = 0;
uint8_t LEDs;

uCAN_MSG txMessage;
uCAN_MSG rxMessage;

timerStruct_t periodicTxMessages[3] = {
    {send_TxRxMsgCount},
    {send_PotStatus},
    {send_SwitchStatus}
};

/*
                         Main application
 */
void main(void)
{
    SYSTEM_Initialize();

    INTERRUPT_GlobalInterruptEnable();
    INTERRUPT_PeripheralInterruptEnable();
    TMR1_SetInterruptHandler(send_SwitchPressCount);
    add_PeriodicMessages();

    while (1)
    {
        if (CAN_receive(&rxMessage)) {
            processRxMessage();
            rxMessageCount++;
        }
        timeout_callNextCallback();
    }
}

void add_PeriodicMessages (void){
    timeout_create(&periodicTxMessages[0], 500 * MilliSeconds);
    timeout_create(&periodicTxMessages[1], 300 * MilliSeconds);
    timeout_create(&periodicTxMessages[2], 100 * MilliSeconds);
}

uint32_t send_TxRxMsgCount(void* payload){
    send_TxMsgCount();
    send_RxMsgCount();
    heartbeat();
    return 500 * MilliSeconds;
}
```

```
// Periodic TX Messages
void send_TxMsgCount(void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x100;
    txMessage.frame.dlc = 8;
    txMessage.frame.data0 = (txMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data1 = (txMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data2 = (txMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data3 = (txMessageCount) & MASK_32BITS;
    txMessage.frame.data4 = (txPeriodicMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data5 = (txPeriodicMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data6 = (txPeriodicMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data7 = (txPeriodicMessageCount) & MASK_32BITS;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
}

void send_RxMsgCount(void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x110;
    txMessage.frame.dlc = 4;
    txMessage.frame.data0 = (rxMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data1 = (rxMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data2 = (rxMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data3 = (rxMessageCount) & MASK_32BITS;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
}

uint32_t send_PotStatus(void* payload){
    ADC_GetConversion(channel_AN0);

    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x120;
    txMessage.frame.dlc = 2;
    txMessage.frame.data0 = ADRESH;
    txMessage.frame.data1 = ADRESL;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
    return 300 * MilliSeconds;
}

void heartbeat(void){
    LED_D8_Toggle();
}

// Event triggered messages
uint32_t send_SwitchStatus(void* payload){
    txMessage.frame.idType = dEXTENDED_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x12345;
    txMessage.frame.dlc = 4;
    txMessage.frame.data0 = (bit) ~S1_PORT;
    txMessage.frame.data1 = (bit) ~S2_PORT;
    txMessage.frame.data2 = (bit) ~S3_PORT;
    txMessage.frame.data3 = (bit) ~S4_PORT;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
    return 100 * MilliSeconds;
}

void send_SwitchPressCount (void){

    s1.status = checkButton(&s1, S1_PORT);
    s2.status = checkButton(&s2, S2_PORT);
    s3.status = checkButton(&s3, S3_PORT);
    s4.status = checkButton(&s4, S4_PORT);
```

```
    if(s4.status || s3.status || s2.status || s1.status){
        txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
        txMessage.frame.id = 0x150;
        txMessage.frame.dlc = 4;
        txMessage.frame.data0 = s1.pressCount;
        txMessage.frame.data1 = s2.pressCount;
        txMessage.frame.data2 = s3.pressCount;
        txMessage.frame.data3 = s4.pressCount;
        CAN_transmit(&txMessage);
        txMessageCount++;
    }
    if(s4.status){
        s4MsgCount++;
        send_Switch4PressCount();
    }
}

void send_Switch4PressCount (void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x160;
    txMessage.frame.dlc = 4;
    txMessage.frame.data0 = (s4MsgCount >> 24) & MASK_32BITS;
    txMessage.frame.data1 = (s4MsgCount >> 16) & MASK_32BITS;
    txMessage.frame.data2 = (s4MsgCount >> 8) & MASK_32BITS;
    txMessage.frame.data3 = (s4MsgCount) & MASK_32BITS;
    CAN_transmit(&txMessage);
    txMessageCount++;
}

void send_softwareVersion (void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x200;
    txMessage.frame.dlc = 2;
    txMessage.frame.data0 = MAJOR_VERSION;
    txMessage.frame.data1 = MINOR_VERSION;
    CAN_transmit(&txMessage);
    txMessageCount++;
}

bool checkButton(switchStat_t *switchNumber, uint8_t port){
    uint8_t previousSample;

    previousSample = switchNumber->sampleLevel;
    switchNumber->sampleLevel = port;

    if ((switchNumber->sampleLevel == previousSample) &&
           (switchNumber->sampleLevel != switchNumber->state)) {
     if (switchNumber->sampleLevel == LOW) {
            (switchNumber->pressCount)++;
            switchNumber->state = switchNumber->sampleLevel;
            return true;
        }
        switchNumber->state = switchNumber->sampleLevel;
        return false;
    }else{
        return false;
    }
}

// Receive Functions

void processRxMessage(void) {
    if (rxMessage.frame.idType == dSTANDARD_CAN_MSG_ID_2_0B) {
        switch (rxMessage.frame.id) {
            case 0x700:
                if(rxMessage.frame.data0 == REQ_VERSION){
                    send_softwareVersion();
                }
                break;
        }
        LED_D6_Toggle();
    } else if (rxMessage.frame.idType == dEXTENDED_CAN_MSG_ID_2_0B) {
        switch (rxMessage.frame.id) {
            case 0xD34FF:
                LEDs = rxMessage.frame.data0;
                LED_Display(LEDs);
```

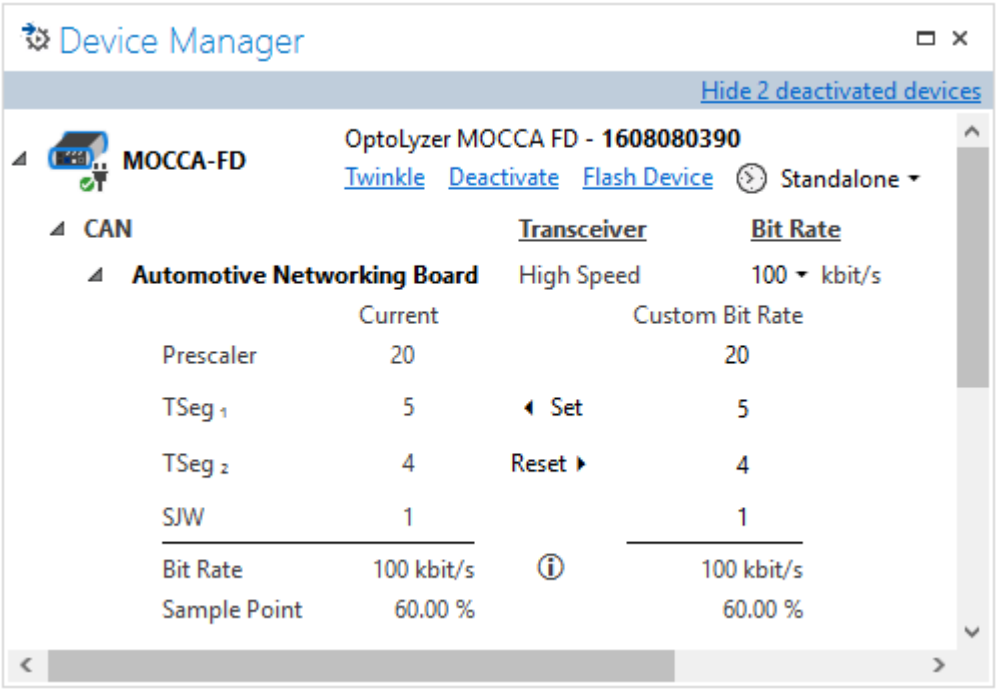
```
                break;
            case 0x201FF:
                if(rxMessage.frame.data0 == CLEAR_COUNTERS){
                    clear_Counters();
                }
                break;
        }
        LED_D7_Toggle();
    }
}

void LED_Display (uint8_t ledStat){
    LED_D5_LAT = (ledStat >> 4) & MASK_8BITS;
    LED_D4_LAT = (ledStat >> 3) & MASK_8BITS;
    LED_D3_LAT = (ledStat >> 2) & MASK_8BITS;
    LED_D2_LAT = (ledStat >> 1) & MASK_8BITS;
    LED_D1_LAT = ledStat & MASK_8BITS;
}

void clear_Counters (void){
    rxMessageCount = 0;
    txMessageCount = 0;
    txPeriodicMessageCount = 0;
    s1.pressCount = 0;
    s2.pressCount = 0;
    s3.pressCount = 0;
    s4.pressCount = 0;
    s4MsgCount = 0;
}
```

6.1.5 OptoLyzer MOCCA FD Settings

To proceed with the next discussion, the user should be familiar with the OptoLyzer Studio and OptoLyzer MOCCA FD tool. To review the different features of the tool: in OptoLyzer Studio, go to *File>Help>Help*.

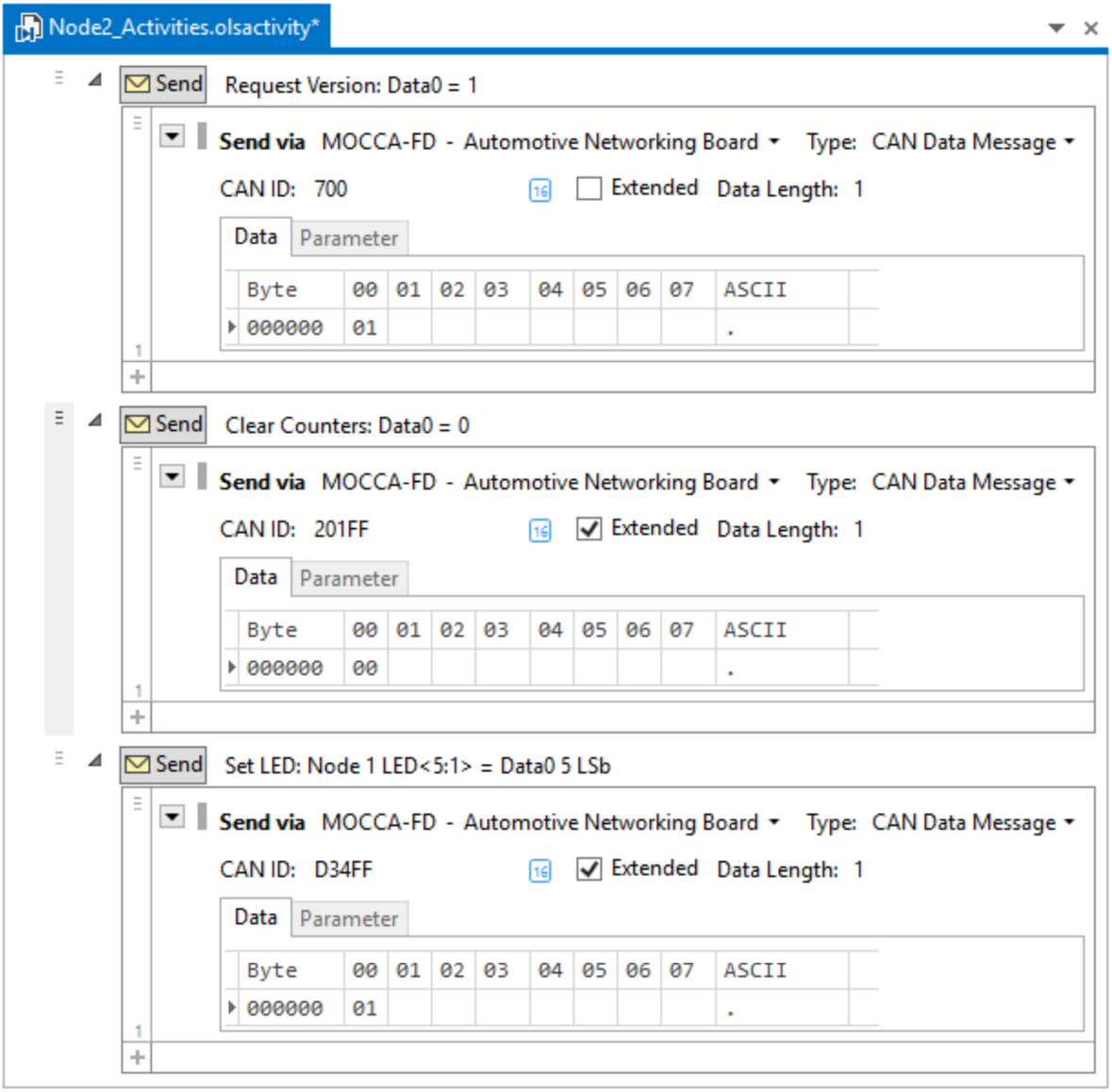
1. In OptoLyzer Studio, go to *View>Device Manager*. Set up the OptoLyzer MOCCA FD as shown in the following figure.

Figure 6-14. OptoLyzer® MOCCA FD Settings



2. Create a new 'CAN Data Frame' activity as shown below. The IDs shown are hexadecimal values.

Figure 6-15. CAN Demo Activity Window

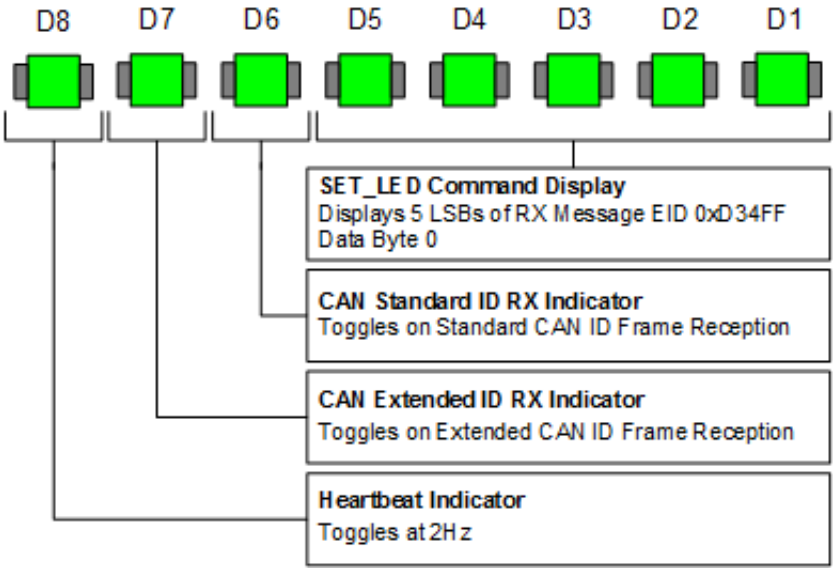


3. In Optolyzer® Studio press the **Start** button.

6.1.6 Automotive Networking Demo Board LEDs Display for Heartbeat and Received Messages

The figure below shows the function of each of the eight LEDs in Node 1.

Figure 6-16. Automotive Networking Board LEDs Display



On-board LED D8 is toggled every 500 ms and acts as a heartbeat to indicate that the PIC18F66K80 ECAN is actively transmitting and monitoring messages on the bus. LEDs D7 and D6 act as Extended and Standard message RX indicator, respectively. LEDs D5 to D1 display the five LSBs of the RX message with Extended ID 0xD34FF. By default, LEDs D7 to D1 are turned off.

6.1.7 Periodic Message Transmission

1. Refer to the OptoLyzer Log window. Upon pressing the **START** button, only the periodic data frames should be displayed on the log window.

   Figure 6-17. Periodic TX Messages

Type	Count	Time	Device	Channel	Summary	ID	Data Length	Data
CAN Data Frame	12	20:34:23.964.673	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 00 00
CAN Data Frame	4	20:34:23.560.130	MOCCA-FD	Automotive Networking Board	0x120 (Standard)	0x120	0x2	0F FF
CAN Data Frame	2	20:34:23.159.708	MOCCA-FD	Automotive Networking Board	0x100 (Standard)	0x100	0x8	00 00 00 1F 00 00 00 1F
CAN Data Frame	2	20:34:23.160.577	MOCCA-FD	Automotive Networking Board	0x110 (Standard)	0x110	0x4	00 00 00 00

   Note: All values in this example are expressed in hexadecimal, unless otherwise stated.

   ID 0x100 shows the count of transmitted messages from Node 1. The number of TX messages are mapped on DB0-DB3 while the number of periodic TX messages are mapped on DB4-DB7. The TX message count and periodic TX message should remain equal until a non-periodic (event-triggered) message is transmitted.

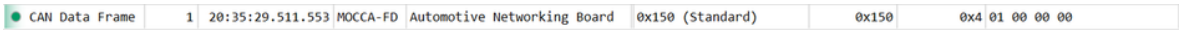
2. Rotate the potentiometer and observe ID 0x120 data. The data should show the result of the right-justified 12-bit ADC in which the high byte and low byte results are mapped to DB0 and DB1, respectively. Rotating the potentiometer clockwise increases the display value. Data should range between `00 00` and 0F FF.

6.1.8 Event-Triggered Message Transmission

1. Press switch S1. A new row should appear with CAN message ID 0x150. This frame maps the press/release count of the four on-board switches, S1 to S4 of the Automotive Networking Board, to

DB0 to DB3, respectively. Each switch count ranges from 00 to FF. The count restarts to 00 when FF is exceeded.
Upon pressing S1, DB0 should display 01. Notice also that ID 0x100 DB3 is now greater than DB7 because a non-periodic message has already been transmitted.

Figure 6-18. S1-Triggered TX Message



CAN Data Frame	1	20:35:29.511.553	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	01 00 00 00

2. Press S2, DB1 should set to 01. Press S3, DB2 should set to 01. Press S4, DB3 should set to 01 and a new row with CAN ID 0x160 should appear.

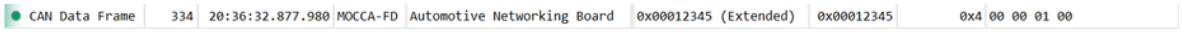
Figure 6-19. Switch-Triggered TX Message

CAN Data Frame	4	20:36:05.081.172	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	01 01 01 01
CAN Data Frame	1	20:36:05.082.032	MOCCA-FD	Automotive Networking Board	0x160 (Standard)	0x160	0x4	00 00 00 01

Notice that CAN ID 0x160 DB3 is equal to CAN ID 0x150 DB3. This is because the CAN ID 0x160 frame simply counts the number of S4 press/release. The only difference is that four bytes are allocated for S4 press/release count, which means that the count will range from 00 00 00 00 to FF FF FF FF.

3. Press any of the switches and observe that the count for each corresponding ID 0x150 data byte increments by one on every press/release, while IDx160 data only increments on every S4 press/release.
4. Notice that when you press any of the switches, the corresponding CAN ID 0x12345x data bytes display the status of the pressed switch. The example below indicates that S3 is currently pressed.

Figure 6-20. Switch-State TX Message



CAN Data Frame	334	20:36:32.877.980	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 01 00

6.1.9 Transmit Messages from OptoLyzer Studio

1. Open the CAN Demo Activity (Figure 6-15). Send the Request Version message with CAN ID 700 and DB0 = 01. CAN ID 0x110 DB3 that maps the RX message count's LSB should display 01 which means that Node 1 received the first message from another node. A couple of rows should be added to the log window. One corresponds to the ID 0x700 and the other is the response transmitted by Node 1 with ID 0x200. The example below shows that Node 1 application software is major version 01 and minor version 00. Notice also that LED D6 is now turned on (toggled from OFF state) because a Standard message is received.

Figure 6-21. Request Version Message and Response

CAN Data Frame	1	20:36:58.437.057	MOCCA-FD	Automotive Networking Board	0x700 (Standard)	0x700	0x1	01
CAN Data Frame	1	20:36:58.438.017	MOCCA-FD	Automotive Networking Board	0x200 (Standard)	0x200	0x2	01 00

2. Send the "Clear Counters" message with CAN ID 201FF and DB0 = 00. This will clear all the message counters in Node 1. LED D7 should be toggled from OFF state because an Extended message is received. Press S4 and notice that the counters have reset to 00.

Figure 6-22. First Message After Node 1 Clears Its Message Counters

CAN Data Frame	10	20:37:44.423.080	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	00 00 00 01
CAN Data Frame	3	20:37:44.423.938	MOCCA-FD	Automotive Networking Board	0x160 (Standard)	0x160	0x4	00 00 00 01

3. Send the "Set LED" message with CAN ID D34FF and DB0 = 01. Notice that LED D7 switches off and D1 turns on. Replace DB0 value with FF. Send the new message. LED D7 and LEDs D5 to D1 should turn on. Try changing DB0 with other values and notice that the 5 LSBs are displayed on LEDs D5 to D1, while D7 toggles on every message transmission.

6.1.10 Two-Node Network Output

The figure below shows a log window summary for the two-node demo. The output should display all the messages defined in the table in Figure 6-4.

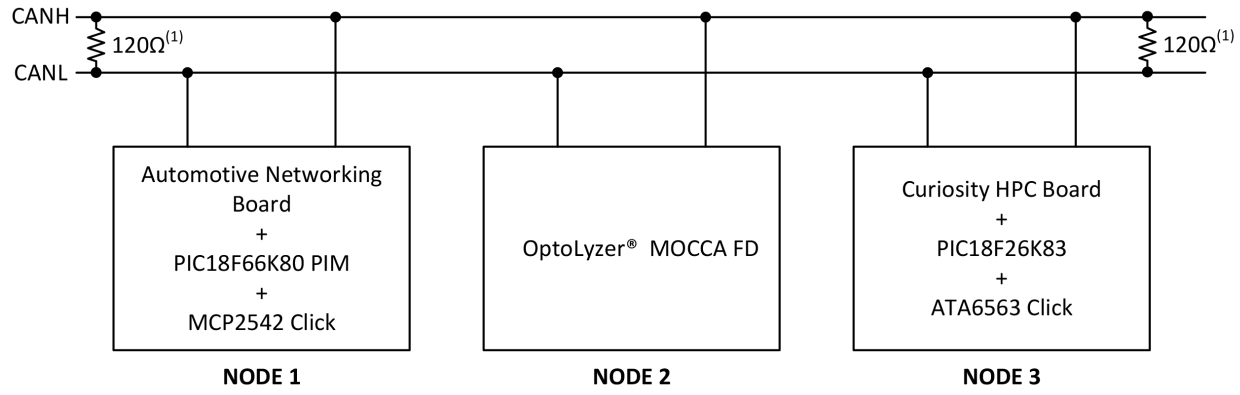
Figure 6-23. Two-Node Network CAN Messages

Type	Count	Time	Device	Channel	Summary	ID	Data Length	Data
CAN Data Frame	912	20:40:24.285.039	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 00 00
CAN Data Frame	304	20:40:23.722.843	MOCCA-FD	Automotive Networking Board	0x120 (Standard)	0x120	0x2	0F FF
CAN Data Frame	182	20:40:23.314.211	MOCCA-FD	Automotive Networking Board	0x100 (Standard)	0x100	0x8	00 00 02 CA 00 00 02 C8
CAN Data Frame	182	20:40:23.315.080	MOCCA-FD	Automotive Networking Board	0x110 (Standard)	0x110	0x4	00 00 00 03
CAN Data Frame	10	20:37:44.423.080	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	00 00 00 01
CAN Data Frame	3	20:37:44.423.938	MOCCA-FD	Automotive Networking Board	0x160 (Standard)	0x160	0x4	00 00 00 01
CAN Data Frame	1	20:36:58.437.057	MOCCA-FD	Automotive Networking Board	0x700 (Standard)	0x700	0x1	01
CAN Data Frame	1	20:36:58.438.017	MOCCA-FD	Automotive Networking Board	0x200 (Standard)	0x200	0x2	01 00
CAN Data Frame	1	20:37:38.799.798	MOCCA-FD	Automotive Networking Board	0x000201FF (Extended)	0x000201FF	0x1	00
CAN Data Frame	2	20:39:14.399.036	MOCCA-FD	Automotive Networking Board	0x000D34FF (Extended)	0x000D34FF	0x1	FF

6.2 Three-Node Network Demo

A three-node CAN network is created by adding a third node to the existing network, as shown in the figure below. The previous configuration of Nodes 1 and 2 are retained in this example. There is no need to change Node 1 software when adding Node 3 to allow both nodes to interact with each other.

Figure 6-24. Three-Node Network



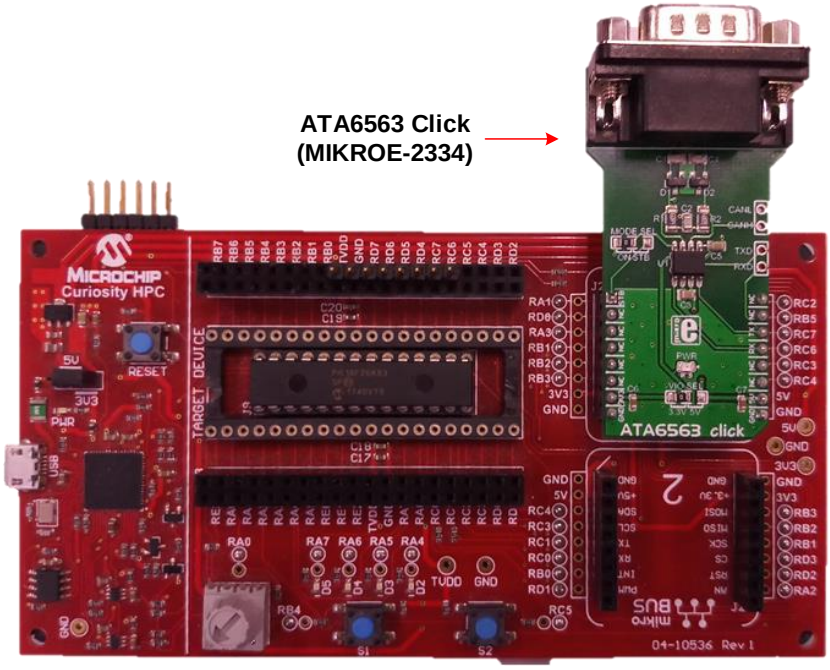
Note:

1. These 120Ω termination resistors are already included in the CAN transceiver clicks used.

Node 3

Node 3 consists of a Curiosity HPC Development Board and an ATA6563 CAN Transceiver Click Board as shown in the following figure.

Figure 6-25. Node 3 – PIC18F26K83 and ATA6563



6.2.1 Three-Node Network Details

The following figure shows the network details for the individual messages expected on the CAN bus for the three-node network demo.

Figure 6-26. Three-Node Network Details

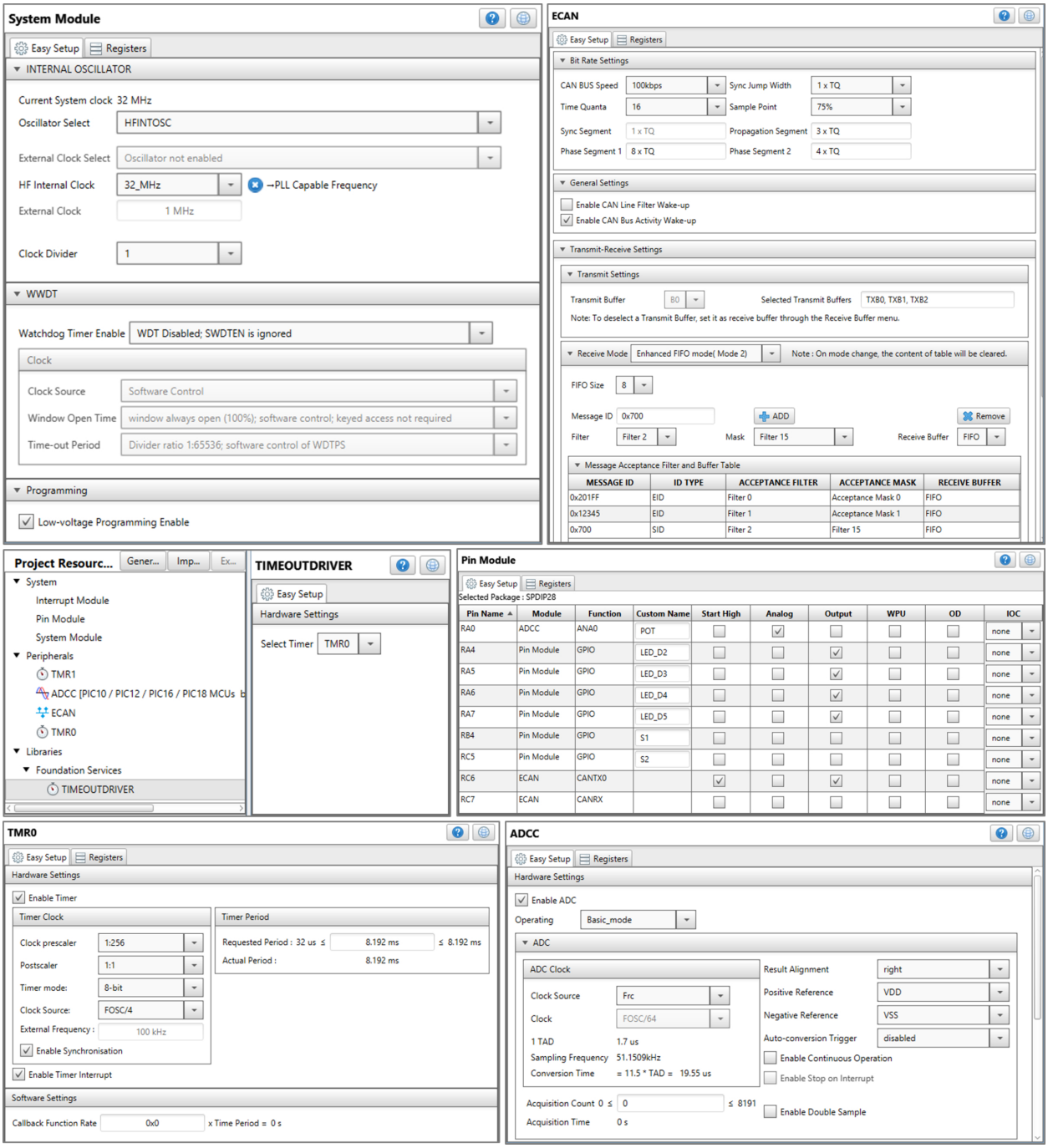
Msg ID	Dir.	Type of Msg	DLC	Data DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7	Message TX Trigger	Details
Node 1 (PIC18F66k80 + Automotive Networking Development Board)													
0x100	TX	Periodic	8	ALL TX Msg Cnt				ALL TX Periodic Msg Cnt				500 msec	
0x110	TX	Periodic	4	ALL RX Msg Cnt								500 msec	
0x120	TX	Periodic	4	Potentiometer								300 msec	
0x12345x	TX	Periodic		SW1 State	SW2 State	SW3 State	SW4 State					100 msec	Switch state Press == 1 or Released == 0 is transmitted
0x150	TX	Event	4	SW0 Count	SW1 Count	SW2 Count	SW3 Count					Press/release of any switch	Counts show how many times the buttons were pressed and released
0x160	TX	Event	4	0x160 Msg Cnt								Press/release of SW4	
0x200	TX	Event	2	Major Ver	Minor Ver							Response to message 0x700	Provides BUS the Application Software Version
0xD34FFx	RX		1	Set LED									Sets LEDs for this node. Maps the lower 5 bits to LED5 to LED1.
0x201FFx	RX		1	Clear Counters									Clears ALL message Counters
0x700	RX		1	Request Ver									Get software version
Node 2 (CAN Tool)													
0x700	TX	Event	1	Request Ver									
0x201FFx	TX	Event	1	Clear Counters									
0xD34FFx	TX	Event	1	Set LED									
All IDs	RX												The Tool will receive all messages on the network
Node 3 (PIC18F26k83 + HPC Curosity Board)													
0x105	TX	Periodic	8	ALL TX Msg Cnt				ALL TX Periodic Msg Cnt				500 msec	
0x115	TX	Periodic	4	ALL RX Msg Cnt								500 msec	
0x125	TX	Periodic	4	Potentiometer								300 msec	
0x155	TX	Event	2	SW1 Count	SW2 Count							Press/release of any switch	
0xD34FFx	TX	Event	1	Set LED								Press/release of any switch	SW1: Set LED = 0x00 or SW2: Set LED = 0xFF
0x205	TX	Event	2	Major Ver	Minor Ver							Response to message 0x700	Response to message 0x700
0x201FFx	RX		1	Clear Counters									Clears ALL message Counters
0x12345x	RX		4	SW0 State	SW1 State	SW2 State	SW3 State						Sets LEDs based on Node 1's switch actions. Maps Node 1's SW3 and SW4 to Node 3's LEDs D2 and D3.
0x700	RX		1	Request Ver									Get software version

The table shows the same messages for Node 1 and Node 2 from Figure 6-4 in the two-node demo. The only difference is that new rows are added for Node 3 messages. This demo shows how the three nodes will communicate with each other when a third node is added to the CAN bus without any changes on the application software and configuration of the nodes from the previous demo.

6.2.2 PIC18F26K83 MCC Configuration Settings

A summary of the MCC peripheral configuration for the PIC16F26K83 is shown in the following figure.

Figure 6-27. PIC18F26K83 MCC Peripheral Settings



6.2.3 Application Code

Edit the main file as shown in the example below.

Example 6-2. PIC18F26K83 Main File

```
#include "mcc_generated_files/mcc.h"

#define MilliSeconds    (uint32_t)31    // 32us=1 tick, this means 31 ticks=1ms
#define PRESSED         1
#define NOT_PRESSED     0
#define MASK_32BITS     0x000000FF
#define MASK_8BITS      0x01
#define MAJOR_VERSION   1
#define MINOR_VERSION   0
#define REQ_VERSION     1
#define CLEAR_COUNTERS  0

typedef struct switchStat{
    bool status;
    uint8_t pressCount;
    uint8_t state;
    uint8_t sampleLevel;
}switchStat_t;

// Function prototypes for periodic messages and heartbeat
void add_PeriodicMessages(void);
uint32_t send_TxRxMsgCount(void *payload);
uint32_t send_PotStatus(void *payload);
void send_TxMsgCount(void);
void send_RxMsgCount(void);
void heartbeat(void);

// Function prototypes for event-triggered messages
bool checkButton(switchStat_t *switchNumber, uint8_t port);
void send_SwitchPressCount (void);
void send_softwareVersion (void);
void send_setNode1Leds (void);
void send_clearNode1Leds (void);

// Function prototypes for receive message
void processRxMessage(void);
void clear_Counters (void);

// Local variables
switchStat_t s1, s2;
uint32_t rxMessageCount = 0;
uint32_t txMessageCount = 0;
uint32_t txPeriodicMessageCount = 0;

uCAN_MSG txMessage;
uCAN_MSG rxMessage;

timerStruct_t periodicTxMessages[] = {
    {send_TxRxMsgCount},
    {send_PotStatus},
};

/*
                        Main application
 */
void main(void)
{
    SYSTEM_Initialize();

    INTERRUPT_GlobalInterruptEnable();
    TMR1_SetInterruptHandler(send_SwitchPressCount);
    add_PeriodicMessages();

    while (1)
    {
        if (CAN_receive(&rxMessage)) {
            processRxMessage();
            rxMessageCount++;
        }
        timeout_callNextCallback();
    }
}
```

```
void add_PeriodicMessages (void){
    timeout_create(&periodicTxMessages[0], 500 * MilliSeconds);
    timeout_create(&periodicTxMessages[1], 300 * MilliSeconds);
}

// Periodic TX Messages

uint32_t send_TxRxMsgCount(void* payload){
    send_TxMsgCount();
    send_RxMsgCount();
    heartbeat();
    return 500 * MilliSeconds;
}

void send_TxMsgCount(void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x105;
    txMessage.frame.dlc = 8;
    txMessage.frame.data0 = (txMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data1 = (txMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data2 = (txMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data3 = (txMessageCount) & MASK_32BITS;
    txMessage.frame.data4 = (txPeriodicMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data5 = (txPeriodicMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data6 = (txPeriodicMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data7 = (txPeriodicMessageCount) & MASK_32BITS;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
}

void send_RxMsgCount(void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x115;
    txMessage.frame.dlc = 4;
    txMessage.frame.data0 = (rxMessageCount >> 24) & MASK_32BITS;
    txMessage.frame.data1 = (rxMessageCount >> 16) & MASK_32BITS;
    txMessage.frame.data2 = (rxMessageCount >> 8) & MASK_32BITS;
    txMessage.frame.data3 = (rxMessageCount) & MASK_32BITS;

    CAN_transmit(&txMessage);
    txMessageCount++;
    txPeriodicMessageCount++;
}

uint32_t send_PotStatus(void* payload){
    ADCC_GetSingleConversion(POT);

    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x125;
    txMessage.frame.dlc = 2;
    txMessage.frame.data0 = ADRESH;
    txMessage.frame.data1 = ADRESL;
    CAN_transmit(&txMessage);

    txMessageCount++;
    txPeriodicMessageCount++;
    return 300 * MilliSeconds;
}

void heartbeat(void){
    LED_D5_Toggle();
}

// Event triggered messages
void send_SwitchPressCount (void){

    s1.status = checkButton(&s1, S1_PORT);
    s2.status = checkButton(&s2, S2_PORT);

    if(s2.status || s1.status){
        txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
        txMessage.frame.id = 0x155;
        txMessage.frame.dlc = 2;
        txMessage.frame.data0 = s1.pressCount;
```

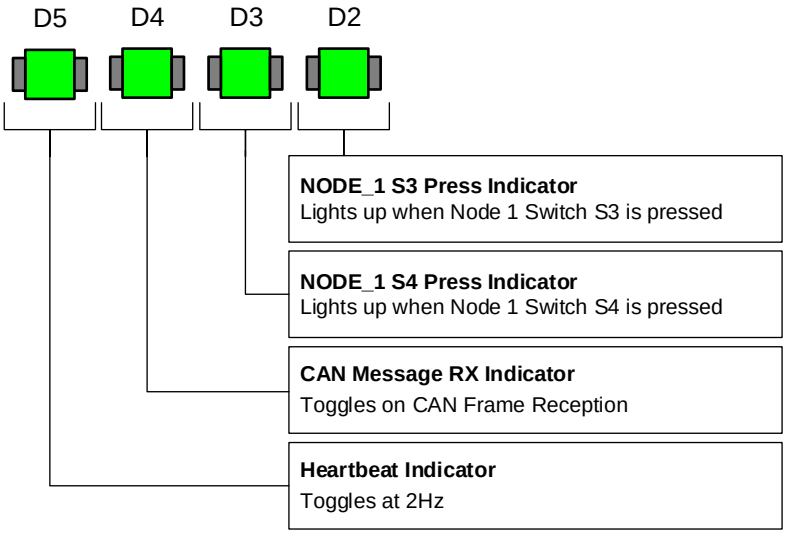
```
        txMessage.frame.data1 = s2.pressCount;
        CAN_transmit(&txMessage);
        txMessageCount++;
    }

    if(s1.status){
        send_clearNode1Leds();
    }else if(s2.status){
        send_setNode1Leds();
    }
}

void send_setNode1Leds (void){
    txMessage.frame.idType = dEXTENDED_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0xD34FF;
    txMessage.frame.dlc = 1;
    txMessage.frame.data0 = 0xFF;
    CAN_transmit(&txMessage);
    txMessageCount++;
}

void send_clearNode1Leds (void){
    txMessage.frame.idType = dEXTENDED_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0xD34FF;
    txMessage.frame.dlc = 1;
    txMessage.frame.data0 = 0x00;
    CAN_transmit(&txMessage);
    txMessageCount++;
}

void send_softwareVersion (void){
    txMessage.frame.idType = dSTANDARD_CAN_MSG_ID_2_0B;
    txMessage.frame.id = 0x205;
    txMessage.frame.dlc = 2;
    txMessage.frame.data0 = MAJOR_VERSION;
    txMessage.frame.data1 = MINOR_VERSION;
    CAN_transmit(&txMessage);
    txMessageCount++;
}

bool checkButton(switchStat_t *switchNumber, uint8_t port){
    uint8_t previousSample;

    previousSample = switchNumber->sampleLevel;
    switchNumber->sampleLevel = port;

    if ((switchNumber->sampleLevel == previousSample) &&
           (switchNumber->sampleLevel != switchNumber->state)) {
     if (switchNumber->sampleLevel == LOW) {
            (switchNumber->pressCount)++;
            switchNumber->state = switchNumber->sampleLevel;
            return true;
        }
        switchNumber->state = switchNumber->sampleLevel;
        return false;
    }else{
        return false;
    }
}

// Receive Functions
void processRxMessage(void) {
    if (rxMessage.frame.idType == dSTANDARD_CAN_MSG_ID_2_0B) {
        switch (rxMessage.frame.id) {
            case 0x700:
                if(rxMessage.frame.data0 == REQ_VERSION){
                    send_softwareVersion();
                }
                break;
        }
    } else if (rxMessage.frame.idType == dEXTENDED_CAN_MSG_ID_2_0B) {
        switch (rxMessage.frame.id) {
            case 0x12345:
                LED_D2_LAT = rxMessage.frame.data2;
                LED_D3_LAT = rxMessage.frame.data3;
                break;
```

```
            case 0x201FF:
                if(rxMessage.frame.data0 == CLEAR_COUNTERS){
                    clear_Counters();
                }
                break;
        }
    }
    LED_D4_Toggle();
}

void clear_Counters (void){
    rxMessageCount = 0;
    txMessageCount = 0;
    txPeriodicMessageCount = 0;
    s1.pressCount = 0;
    s2.pressCount = 0;
}
```

6.2.4 Curiosity HPC Demo Board LEDs Display for Heartbeat and Received Messages

The figure below shows the function of each of the four LEDs in Node 3.

Figure 6-28. Curiosity HPC Demo Board LEDs Display



On-board LED D5 is toggled every 500 ms and acts as a heartbeat to indicate that the PIC18F26K83 ECAN is actively transmitting and monitoring messages on the bus. LED D4 acts as a CAN RX indicator and toggles on every message reception (both Extended and Standard). LEDs D2 to D3 indicate the status of Node 1 switches S3 and S4.

6.2.5 Node 3 Message Transmission

1. Refer to the OptoLyzer Log window. Upon connecting Node 3, three rows will be added to the log window. One each for Node 3 periodic messages with IDs 0x125, 0x105 and 0x115. Notice that IDs 0x105 and 0x115 constantly increment. These IDs simply mirror the behavior of IDs 0x100 (TX message count) and 0x110 (RX message count) of Node 1.

Figure 6-29. CAN Messages After Adding Node 3

Type	Count	Time	Device	Channel	Summary	ID	Data Length	Data
CAN Data Frame	306	20:45:41.873.290	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 00 00
CAN Data Frame	102	20:45:41.819.228	MOCCA-FD	Automotive Networking Board	0x120 (Standard)	0x120	0x2	0F FF
CAN Data Frame	61	20:45:41.416.780	MOCCA-FD	Automotive Networking Board	0x100 (Standard)	0x100	0x8	00 00 01 A0 00 00 01 A0
CAN Data Frame	61	20:45:41.417.648	MOCCA-FD	Automotive Networking Board	0x110 (Standard)	0x110	0x4	00 00 00 03
CAN Data Frame	9	20:43:53.448.843	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	02 02 03 02
CAN Data Frame	2	20:43:49.345.747	MOCCA-FD	Automotive Networking Board	0x160 (Standard)	0x160	0x4	00 00 00 02
CAN Data Frame	1	20:44:00.009.517	MOCCA-FD	Automotive Networking Board	0x700 (Standard)	0x700	0x1	01
CAN Data Frame	1	20:44:00.010.468	MOCCA-FD	Automotive Networking Board	0x200 (Standard)	0x200	0x2	01 00
CAN Data Frame	1	20:44:04.993.023	MOCCA-FD	Automotive Networking Board	0x000201FF (Extended)	0x000201FF	0x1	00
CAN Data Frame	2	20:44:15.562.689	MOCCA-FD	Automotive Networking Board	0x000D34FF (Extended)	0x000D34FF	0x1	FF
CAN Data Frame	245	20:45:42.095.908	MOCCA-FD	Automotive Networking Board	0x125 (Standard)	0x125	0x2	0A 95
CAN Data Frame	147	20:45:42.086.881	MOCCA-FD	Automotive Networking Board	0x105 (Standard)	0x105	0x8	00 00 02 18 00 00 02 18
CAN Data Frame	147	20:45:42.087.716	MOCCA-FD	Automotive Networking Board	0x115 (Standard)	0x115	0x4	00 00 00 B7

LED D4 should blink at a rate of approximately 100 ms - the same rate that Node1 transmits the periodic message for the switch state with ID 0x12345x.

2. Rotate Node 3 potentiometer and observe ID 0x125 data. Rotating the potentiometer counterclockwise will increase the output value display. Data should range between `00 00` and 0F FF.
3. Press switch S1. A new row should appear for ID 0x155. DB0 corresponds to the S1 press/release count and should display `01`.

Figure 6-30. Node 3 S1-Triggered TX Message

CAN Data Frame	1	20:46:21.269.293	MOCCA-FD	Automotive Networking Board	0x155 (Standard)	0x155	0x2	01 00

Also observe that pressing S1 sets the CAN ID 0xD34FF DB0 to `00`. This clears LEDs D5 to D1 of Node 1.

Figure 6-31. Node 1 Set LEDs RX Message

CAN Data Frame	3	20:46:21.270.099	MOCCA-FD	Automotive Networking Board	0x000D34FF (Extended)	0x000D34FF	0x1	00

4. Press S2 and notice that CAN ID 0xD34FF DB0 is now set to FF. Node 1 LEDs D5 to D1 should turn on.

Figure 6-32. Node 3 S2-Triggered TX Message

CAN Data Frame	4	20:47:31.359.651	MOCCA-FD	Automotive Networking Board	0x000D34FF (Extended)	0x000D34FF	0x1	FF

Since both S1 and S2 were already pressed once, ID 0x155 data should also update accordingly.

Figure 6-33. Node 3 Switch Press Count TX Message

CAN Data Frame	2	20:47:31.358.827	MOCCA-FD	Automotive Networking Board	0x155 (Standard)	0x155	0x2	01 01

6.2.6 Node 1 Event-Triggered Message Transmission

1. Press and hold Node 1 switch S3. Message ID 0x12345x DB2 is set to 01 and Node 3 LED D2 lights up.

Figure 6-34. Node 1 Switch State TX Message

CAN Data Frame	772	20:48:48.440.707	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 01 00

2. Do the same for Node 1 switch S4 and notice that Node 3 LED D3 lights up while the switch is held.

6.2.7 Node 2 Event-Triggered Message Transmission

1. Open the same CAN Demo Activity (Figure 6-15) used in the two-node demo. Send the “Request Version” message with CAN ID 700 and DB0 = `01`. A new row should be added to the log window with ID 0x205. The example below shows that Node 2 application software is major version `01` and minor version `00`.

Figure 6-35. Request Version Response

CAN Data Frame	1	20:49:21.853.122	MOCCA-FD	Automotive Networking Board	0x205 (Standard)	0x205	0x2	01 00

2. Send the “Clear Counters” message with CAN ID 201FF and DB0 = 00. This will clear all the message counters in Node 1 and Node 2. Press S2 and notice that the counters have reset to 00.

Figure 6-36. First Message After Node 3 Clears Its Message Counters

CAN Data Frame	3	20:50:08.751.401	MOCCA-FD	Automotive Networking Board	0x155 (Standard)	0x155	0x2	00 01

6.3 Three-Node Network Output

The following figure shows a sample log after adding Node 3 and executing the steps described above. The output should display all the messages defined in the table in Figure 6-26.

Figure 6-37. Three-Node Network CAN Messages

Type	Count	Time	Device	Channel	Summary	ID	Data Length	Data
CAN Data Frame	1144	20:51:17.372.994	MOCCA-FD	Automotive Networking Board	0x00012345 (Extended)	0x00012345	0x4	00 00 00 00
CAN Data Frame	381	20:51:16.771.760	MOCCA-FD	Automotive Networking Board	0x120 (Standard)	0x120	0x2	0F FF
CAN Data Frame	228	20:51:15.560.605	MOCCA-FD	Automotive Networking Board	0x100 (Standard)	0x100	0x8	00 00 01 2F 00 00 01 2F
CAN Data Frame	228	20:51:15.561.463	MOCCA-FD	Automotive Networking Board	0x110 (Standard)	0x110	0x4	00 00 00 02
CAN Data Frame	10	20:48:44.695.746	MOCCA-FD	Automotive Networking Board	0x150 (Standard)	0x150	0x4	00 00 01 00
CAN Data Frame	2	20:43:49.345.747	MOCCA-FD	Automotive Networking Board	0x160 (Standard)	0x160	0x4	00 00 00 02
CAN Data Frame	2	20:49:21.852.395	MOCCA-FD	Automotive Networking Board	0x700 (Standard)	0x700	0x1	01
CAN Data Frame	2	20:49:21.853.812	MOCCA-FD	Automotive Networking Board	0x200 (Standard)	0x200	0x2	01 00
CAN Data Frame	2	20:50:05.443.822	MOCCA-FD	Automotive Networking Board	0x000201FF (Extended)	0x000201FF	0x1	00
CAN Data Frame	5	20:50:08.752.217	MOCCA-FD	Automotive Networking Board	0x000D34FF (Extended)	0x000D34FF	0x1	FF
CAN Data Frame	1368	20:51:17.443.962	MOCCA-FD	Automotive Networking Board	0x125 (Standard)	0x125	0x2	0F FF
CAN Data Frame	820	20:51:16.993.618	MOCCA-FD	Automotive Networking Board	0x105 (Standard)	0x105	0x8	00 00 02 10 00 00 02 0E
CAN Data Frame	820	20:51:16.994.453	MOCCA-FD	Automotive Networking Board	0x115 (Standard)	0x115	0x4	00 00 00 B4
CAN Data Frame	3	20:50:08.751.401	MOCCA-FD	Automotive Networking Board	0x155 (Standard)	0x155	0x2	00 01
CAN Data Frame	1	20:49:21.853.122	MOCCA-FD	Automotive Networking Board	0x205 (Standard)	0x205	0x2	01 00

7. Conclusion

This application note covers all the basic details about the ECAN MCC module. Two sample implementations were included to demonstrate the use of the ECAN MCC module on two different 8-bit microcontrollers from Microchip Technology Inc. These implementations were demonstrated with the CAN tool hardware and GUI from K2L GmbH & Co. KG. Proper ECAN MCC module use can help save significant application development time, especially for users who do not want to dig into all the intricacies of setting up the ECAN peripheral.

8. APPENDIX A: Calculations

The ECAN MCC module frees the user from tedious calculation of the different ECAN register values. The following sections provide supplementary information on how the ECAN module works in the background to define the values of the different registers and bits based on the user's settings in the GUI.

8.1 APPENDIX A-1: Baud Rate Prescaler

The following calculations show how the module derives the Baud Rate Prescaler (BRP) based on the available parameters in the GUI.

First, Nominal Bit Rate (NBR) needs to be defined. NBR is the number of bits per second transmitted in the absence of resynchronization as expressed in the following equation.

Equation 8-1. Nominal Bit Rate

$$NBR = f_{bit} = \frac{1}{t_{bit}}$$

where t_{bit} is the Nominal Bit Time (NBT), which is the summation of nonoverlapping segments as shown in Equation 8-2.

Equation 8-2. Nominal Bit Time

$$NBT = t_{bit} = t_{SyncSeg} + t_{\mathrm{PropSeg}} + t_{PS1} + t_{PS2}$$

Each time segment is an exact integer multiple of one Time Quantum (TQ). As shown in Equation 8-3, TQ is a function of BRP and the system frequency, F_{OSC}.

Equation 8-3. Time Quanta

$$TQ = \frac{2 \times (BRP + 1)}{F_{OSC}}$$

Since NBT is composed of time segments which are multiples of TQ, therefore NBT is also an integer multiple of TQ. Using a TQ multiplier *n*, NBR can also be expressed as shown in Equation 8-4.

Equation 8-4. Nominal Bit Rate as a Function of TQ

$$NBR = \frac{1}{NBT} = \frac{1}{n \times TQ}$$

Manipulating Equation 8-4 will give another equation for TQ (see equation below).

Equation 8-5. Relationship Between TQ and NBR

$$TQ = \frac{1}{n \times NBR}$$

Equating Equation 8-3 and Equation 8-5 will lead to a single equation for the BRP as a function of the given parameters in the GUI (see the following equation).

Equation 8-6. Nominal Bit Rate as a Function of TQ

$$BRP = \frac{F_{OSC}}{2 \times NBR \times n} - 1$$

The ECAN module will generate a warning notification if it cannot calculate a valid BRP based on the entered parameters in the GUI.

8.2 APPENDIX A-2: Sample Point

The Sample Point is always located at the end of PS1. The calculation of sample point as percentage of one bit time is shown in the equation below.

Equation 8-7. Sample Point

$$\%\,SamplePoint = \frac{t_{SyncSeg} + t_{PropSeg} + t_{PS1}}{t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2}} \times 100\,\%$$

$$\%\,SamplePoint = \frac{t_{SyncSeg} + t_{PropSeg} + t_{PS1}}{t_{bit}} \times 100\,\%$$

The Microchip Web Site

Microchip provides online support via our web site at http://www.microchip.com/. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

Customer Change Notification Service

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at http://www.microchip.com/. Under "Support", click on "Customer Change Notification" and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or Field Application Engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://www.microchip.com/support

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.

- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Legal Notice

Trademarks

ISBN: 978-1-5224-3331-6

Quality Management System Certified by DNV

ISO/TS 16949

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: http://www.microchip.com/support Web Address: www.microchip.com **Atlanta** Duluth, GA Tel: 678-957-9614 Fax: 678-957-1455 **Austin, TX** Tel: 512-257-3370 **Boston** Westborough, MA Tel: 774-760-0087 Fax: 774-760-0088 **Chicago** Itasca, IL Tel: 630-285-0071 Fax: 630-285-0075 **Dallas** Addison, TX Tel: 972-818-7423 Fax: 972-818-2924 **Detroit** Novi, MI Tel: 248-848-4000 **Houston, TX** Tel: 281-894-5983 **Indianapolis** Noblesville, IN Tel: 317-773-8323 Fax: 317-773-5453 Tel: 317-536-2380 **Los Angeles** Mission Viejo, CA Tel: 949-462-9523 Fax: 949-462-9608 Tel: 951-273-7800 **Raleigh, NC** Tel: 919-844-7510 **New York, NY** Tel: 631-435-6000 **San Jose, CA** Tel: 408-735-9110 Tel: 408-436-4270 **Canada - Toronto** Tel: 905-695-1980 Fax: 905-695-2078	**Australia - Sydney** Tel: 61-2-9868-6733 **China - Beijing** Tel: 86-10-8569-7000 **China - Chengdu** Tel: 86-28-8665-5511 **China - Chongqing** Tel: 86-23-8980-9588 **China - Dongguan** Tel: 86-769-8702-9880 **China - Guangzhou** Tel: 86-20-8755-8029 **China - Hangzhou** Tel: 86-571-8792-8115 **China - Hong Kong SAR** Tel: 852-2943-5100 **China - Nanjing** Tel: 86-25-8473-2460 **China - Qingdao** Tel: 86-532-8502-7355 **China - Shanghai** Tel: 86-21-3326-8000 **China - Shenyang** Tel: 86-24-2334-2829 **China - Shenzhen** Tel: 86-755-8864-2200 **China - Suzhou** Tel: 86-186-6233-1526 **China - Wuhan** Tel: 86-27-5980-5300 **China - Xian** Tel: 86-29-8833-7252 **China - Xiamen** Tel: 86-592-2388138 **China - Zhuhai** Tel: 86-756-3210040	**India - Bangalore** Tel: 91-80-3090-4444 **India - New Delhi** Tel: 91-11-4160-8631 **India - Pune** Tel: 91-20-4121-0141 **Japan - Osaka** Tel: 81-6-6152-7160 **Japan - Tokyo** Tel: 81-3-6880- 3770 **Korea - Daegu** Tel: 82-53-744-4301 **Korea - Seoul** Tel: 82-2-554-7200 **Malaysia - Kuala Lumpur** Tel: 60-3-7651-7906 **Malaysia - Penang** Tel: 60-4-227-8870 **Philippines - Manila** Tel: 63-2-634-9065 **Singapore** Tel: 65-6334-8870 **Taiwan - Hsin Chu** Tel: 886-3-577-8366 **Taiwan - Kaohsiung** Tel: 886-7-213-7830 **Taiwan - Taipei** Tel: 886-2-2508-8600 **Thailand - Bangkok** Tel: 66-2-694-1351 **Vietnam - Ho Chi Minh** Tel: 84-28-5448-2100	**Austria - Wels** Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 **Denmark - Copenhagen** Tel: 45-4450-2828 Fax: 45-4485-2829 **Finland - Espoo** Tel: 358-9-4520-820 **France - Paris** Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 **Germany - Garching** Tel: 49-8931-9700 **Germany - Haan** Tel: 49-2129-3766400 **Germany - Heilbronn** Tel: 49-7131-67-3636 **Germany - Karlsruhe** Tel: 49-721-625370 **Germany - Munich** Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 **Germany - Rosenheim** Tel: 49-8031-354-560 **Israel - Ra'anana** Tel: 972-9-744-7705 **Italy - Milan** Tel: 39-0331-742611 Fax: 39-0331-466781 **Italy - Padova** Tel: 39-049-7625286 **Netherlands - Drunen** Tel: 31-416-690399 Fax: 31-416-690340 **Norway - Trondheim** Tel: 47-7289-7561 **Poland - Warsaw** Tel: 48-22-3325737 **Romania - Bucharest** Tel: 40-21-407-87-50 **Spain - Madrid** Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 **Sweden - Gothenberg** Tel: 46-31-704-60-40 **Sweden - Stockholm** Tel: 46-8-5090-4654 **UK - Wokingham** Tel: 44-118-921-5800 Fax: 44-118-921-5820