DMX512A

Author: *Michael Pearce*
Microchip Technology Inc.

INTRODUCTION

This application note presents an overview of DMX512A. It shows and explains the recommended electrical and electronic requirements, then presents how the protocol was implemented in the Microchip Technology DMX512 Library. Included is a discussion on using the back-channel capabilities.

DMX512 is the most common lighting communications protocol used in theatrical lighting, and is often found in architectural and other lighting systems. It was created in 1986 by the United States Institute for Theatrical Technology (USITT) to be a more reliable replacement for the 0 to 10V standard, which was commonly used at the time. In 1998, it was taken over by the Entertainment Services and Technology Association (ESTA). It became DMX512A in 1998, and was revised in 2008. It has been an ANSI standard since 2004 (ANSI E1.11-2008).

Uses of DMX512A include dimming theatrical lights, running color mixing lighting fixtures, robot scanner lighting fixtures, strobe lights, fog machines and more. Some vendors are using DMX512 to control commercial or domestic lighting fixtures and even giant high definition television displays have been built using multiple DMX512 universes. Due to there being no error checking, there are some restrictions of use. Basically, any use that could potentially harm or kill a human or animal is not allowed. This includes, but is not limited to, moving stages, moving trusses and pyrotechnic control.

DMX512A is a simple, single master serial protocol that uses the RS-485 Electrical layer at a data rate of 250 Kbaud, no error checking and up to 512 bytes of data. A DMX512 Universe is made up of 512 channels of information sent from a single controller. Multiple universes can be created by using additional controllers. The electrical requirements are very well defined in the specification, including what circuit designs to use and the exact type of cable and connectors that are allowed. Despite this, there are many manufacturers that do not follow the specification and use invalid circuit layout and incorrect connectors, which can cause issues with compatibility and reliability.

The data packet is retransmitted continuously, which helps correct any data glitches. If no packet has been transmitted within one second, then most receivers will switch to a default setting or to an Off mode. The maximum refresh rate for 512 channels is 44 updates per second, which allows for nice dimming with little or no visible flicker on incandescent bulbs. LEDs may require smart control to smooth out fading.

DMX512 TERMINOLOGY

- **Controller**: This is the device transmitting the DMX512 data
- **Receiver**: This is the device that receives and processes the DMX512 data
- **Universe**: Group of up to 512 channels run from a single controller
- **Terminator**: 120Ω resistor at the end of the data bus between D+ and D-
- **Packet**: A BREAK, Mark After Break (MAB), START, and the following 512 data slots
- **Back Channel**: Special mode where a receiver sends data back to the controller.
- **Data Link**: The physical wires the data is transmitted across.

ELECTRICAL REQUIREMENTS

The electrical requirements are based on the RS-485 standard, with slight variation. It uses RS-485 differential transceivers, maximum transmission line length of 1200m (3900 feet), and up to 32 receivers per RS-485 transmitter. The driver output range is +/- 1.5V to +/- 6V and receiver sensitivity of +/-200 mV.

For DMX512A, it is required that the receivers are isolated, which helps avoid earth loop and potentially lethal voltage differences. Many low-cost receivers, often used by DJs and for party lighting, are still non-isolated and can cause communication issues, or be a safety hazard, especially if used in a larger network of lights.

Cable requirements are specified as a cable with nominal characteristic impedance of 120Ω with a shield and two twisted pairs. Only one of the pairs is generally used, as the second pair is set aside for a secondary channel that is not commonly used. Because of this, many DMX512 cables use a single pair. Due to these specific requirements and hard wearing environment of stage lighting, there are a number of cable manufacturers that make DMX512 specific data cable that is flexible, very heavy duty and have DMX, DMX512 or DMX512A printed on the cable itself. The DMX512A specification introduced the use of CAT5E cable and RJ45 connectors strictly for use in permanent installation, where the cable is not being moved, and the connectors are only used on occasion. This is more cost-effective, but would not last in a non-permanent setting.

The standard only specifies the use of 5-pin XLR type connectors for DMX512 communication, and DMX512A added the use of RJ45 connectors for permanent installation. For the XLR connectors, the plug is used for the DMX IN, and the socket is used for DMX OUT. The pinouts for the 5-pin XLR and RJ45 are shown in Table 1.

FIGURE 1: DMX512A CONNECTORS – RJ45 AND 5-PIN XLR

TABLE 1: CONNECTOR PINOUTS

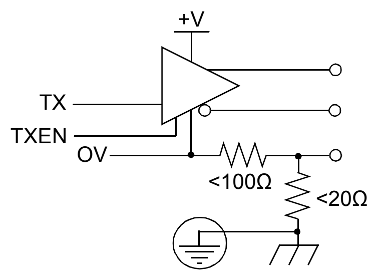
Pin #	5-Pin XLR	RJ45
1	Signal Common	Data 1+
2	Data 1- (Primary Data Link)	Data 1-
3	Data 1+ (Primary Data Link)	Data 2+
4	Data 2- (Optional secondary link)	Not Assigned
5	Data 2+ (Optional secondary Link)	Not Assigned
6		Data 2-
7		Signal Common for Data 1
8		Signal Common for Data 2

Despite the standard specifying the connectors, many low-cost manufacturers used 3-pin XLR connectors commonly used for professional microphones. This introduced a number of issues from different pinouts, and the common mistake of using microphone cable instead of DMX data cable, and accidental plugging of DMX and audio equipment together. It is highly recommended that the user comply with the DMX512A specified cables and connectors to avoid these issues, and to ensure DMX512A compliance.

CIRCUITRY

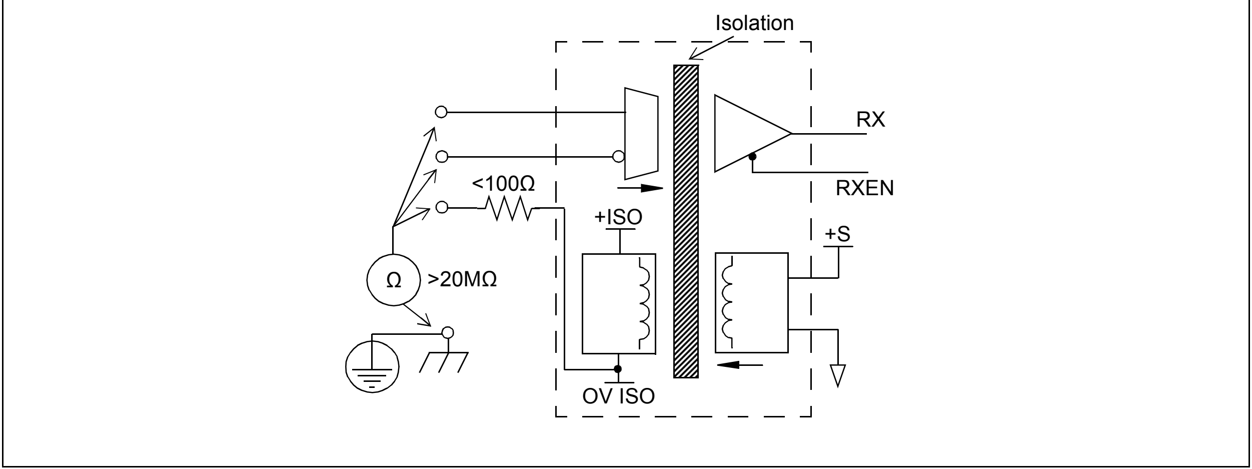
For the controller, the transceiver is usually a ground-referenced transmitter, although it is possible to have it isolated. The recommended circuit from the DMX512A specification (Figure 2) shows how to connect the signal and chassis (earth) grounds using resistors. Even though the resistors are optional, they can possibly reduce earth loop issues if they are included in the design. More details are found in the specification (ANSI E1.11-2008).

FIGURE 2: CONTROLLER TRANSMITTER CIRCUIT



The receiver is isolated, as shown in Figure 3. The resistance from any pin on the DMX connector to chassis ground must be greater than 22 MΩ at 42V. An optional capacitor may be fitted between the data link common and the chassis ground for radio frequency bypass.

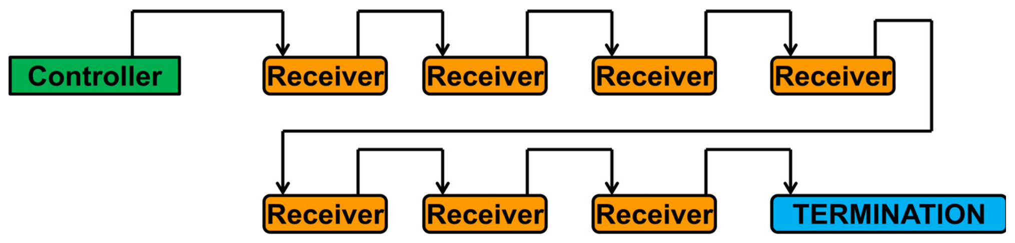
FIGURE 3: RECEIVER CIRCUIT



CONNECTION TOPOLOGY

DMX512 uses a daisy chain connection topology. Up to 32 receivers can be daisy-chained from a single driver with a 120Ω termination resistor at the far end of the chain. To achieve this, there are two connectors on each receiver, often labeled “DMX IN” and “DMX OUT”. These two connectors have the data and ground lines connected between them, so it electrically passes the data along the chain. Figure 4 demonstrates the daisy chain principle.

FIGURE 4: DMX512 DAISY CHAIN

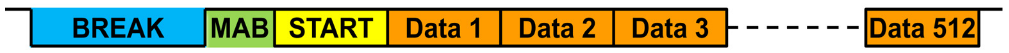


To connect more than 32 receivers to a single controller, then DMX splitters are required. These can be as simple as an isolated RS-485 receiver that feeds to multiple RS-485 transmitters or a more elaborate system that uses microcontrollers to capture and retransmit data with the possibility of limited back channel capability. Because of this, receivers with back channel capability should be connected directly to the controller’s bus to avoid potential problems.

THE DATA PACKET

The data packet consists of a BREAK, MAB (Mark After Break), a START code, then 512 bytes of data at a rate of 250 Kbits/second. A data byte is a Start bit, eight data bits and two Stop bits with LSB sent first, which happens to match standard UART transmission capabilities and makes it easier for us. The BREAK occurs when the data is held low for greater than 92 us. The MAB is when the data is held high for more than 12 us, but less than one second. The START code is a byte of data that determines the type of data that will be sent, the most common is dimmer data, which is a value of `0x00`.

FIGURE 5: THE DMX512 DATA PACKET



Each receiver has a manually-programmed base address, often set by DIP or rotational switches, and more recently, by a display and buttons. This address is used to select where the receiver starts reading data from. After a BREAK, MAB and a valid START code, a receiver counts the bytes until the count matches the preset base address. The receiver reads the number of bytes it requires, then discards anything else until the next break. This makes addressing very simple and allows multiple units to use the same base address, which is very useful in theatrical and architectural lighting.

BIDIRECTIONAL COMMUNICATION

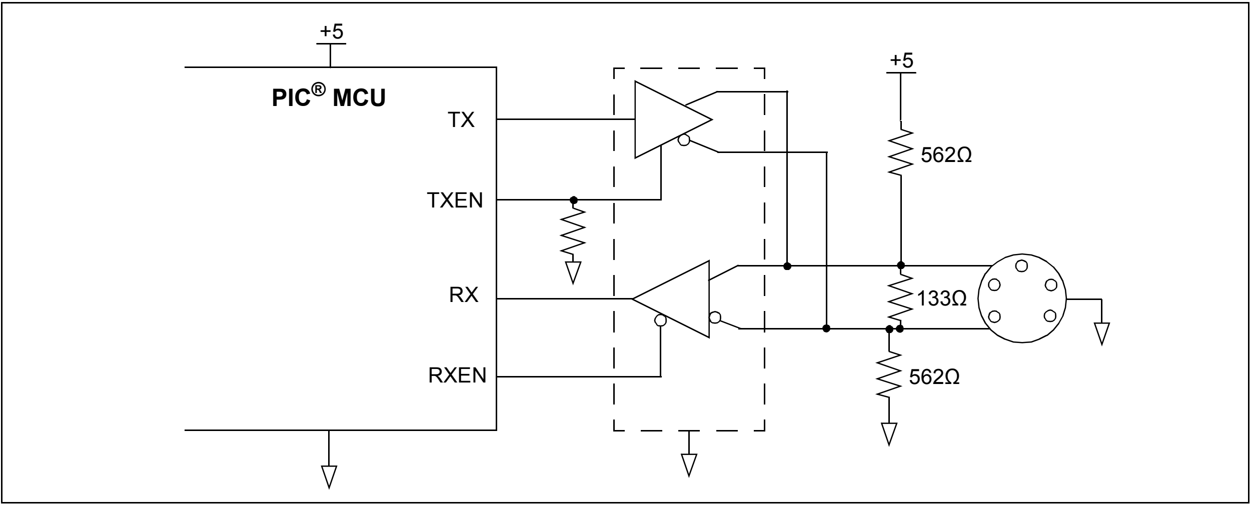
The DMX512 standard does allow for bidirectional communication, and is covered in *ANSI E1.11-2008 Annex B – Enhanced DMX512*. Communication can operate in Full or Half-Duplex modes using just the primary data link, or using both primary and secondary data links. The data protocols and any timing requirements are not specified in the standard.

The most common use of the bidirectional communication is known as “Remote Device Management (RDM)”, and is defined in ANSI E1.20 – 2010. RDM uses the primary data link in Half-Duplex mode and has special START codes that are used. There are other uses of bidirectional communication that are not covered by the specification.

To achieve bidirectional communication, all the transceivers must be bidirectional and controlled using an additional I/O pin. In normal operation, the controller should be configured for Transmit mode and the receiver for Receive mode. The controller will initiate a data request using the same BREAK, MAB, then a special START code followed by appropriate addressing and control data. The controller switches to Receive mode, the receiver switches to Transmit mode, and then data can be transmitted back to the controller. The data returned is normal bytes of data with one Start byte, eight bits of data and two Stop bits. The communications shall follow the normal timing for a DMX512 packet, so that other receivers are kept alive.

Additional termination may be required to achieve reliable communications. This may be as simple as using resistors at both ends of the data link, or there may be requirements to do specific biasing of the data lines, or other methods to ensure data integrity. These additional termination methods are beyond the scope of the DMX512A standard. Figure 6 is an example of termination with biasing implemented at the DMX Controller. (See *ANSI E1.20-2010, Section 2.5, Command Port Reference Circuit* for details).

FIGURE 6: TYPICAL BIDIRECTIONAL CONTROLLER WITH BIASED TERMINATION



The DMX512A Remote Device Management is defined in ANSI E1.20 – 2010 and uses the bidirectional communications, as mentioned in *ANSI E1.11-2008, Annex B and Annex E*. There are specific termination requirements and use bidirectional half-duplex operation over the primary data link. Timing requirements for BREAK and MAB are more stringent, and include additional inter-slot and other timing requirements.

The RDM protocol is much more complicated than general DMX512 communications, as it has a specific data format and command set to be able to auto-find devices using a binary search tree, configure devices, report faults and more. Implementation of an RDM responder (DMX receiver that has RDM capability) can be done in a microcontroller, as the minimum command set that needs to be implemented is small. An RDM controller is more complicated and usually requires notably more processing power and data storage.

Details on how to implement RDM are beyond the scope of this application note, but the user needs to be aware that it exists and the importance of ignoring non-valid START codes. This allows your receiver product to operate correctly when used on a DMX universe that has RDM communications.

IMPLEMENTING THE DMX512 RECEIVER

Using the EUSART found on many PIC® devices, the BREAK can easily be detected using the framing error interrupt. The data rate of 250 Kbaud is a clean divide down from common oscillator frequencies, such as 4, 8 or 32 MHz, which results in a theoretical 0% data rate error. These features make implementing DMX512 a relatively simple task.

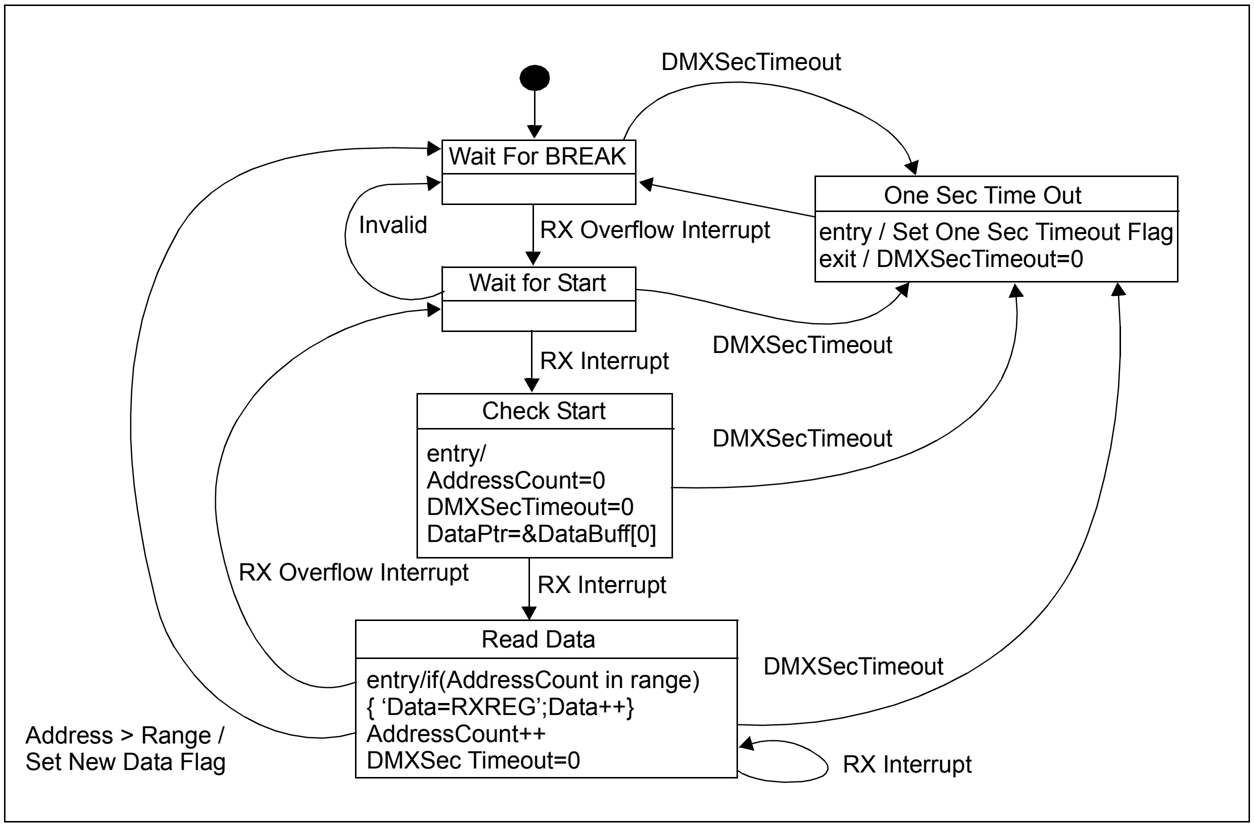
According to the DMX512A specification, the receiver should be isolated. There are a number of isolated RS-485 transceivers that can be used for this task, each with their own pros and cons and often not very cost-effective. The most cost-effective parts are usually an isolated transceiver with an external-isolated power supply. The least cost-effective parts combine the isolated transceiver and the isolated power supply into one package with little or no additional components.

The flow of the receiver is as follows:

1. Wait for a BREAK
2. Receive a valid START code
3. Read and discard data until the appropriate addressing is matched.

Anytime a BREAK occurs the process starts over. If the START code is not valid, then all data is discarded until the next BREAK. This is shown in the state diagram below (Figure 7).

FIGURE 7: STATE DIAGRAM FOR THE RECEIVER INTERRUPT



To implement this in code, we configure the EUSART for 250 Kbps baud rate and eight bits of data with interrupt enabled for reception.

`BREAK` condition causes a framing error in the USART. This error is tested in the interrupt routine before the switch statement.

EXAMPLE 1: CHECKING FOR A `BREAK` CONDITION

```
if(RC1STAbits.FERR)                  // Check for Framing Error - i.e. BREAK
{
  RxDat=RC1REG;                      // Clear the Framing Error by reading the RCREG
  DMX_Flags.RxBreak=1;               // Indicate a break occurred
  DMX_RxState=RX_WAIT_FOR_START;     // Move/Force to Wait For Command state
  DMX_RxTimer=0;                     // Clear the DMX timeout
}
```

The next byte read is the `start` byte. If the `start` is not valid, then the code moves back to waiting for a BREAK. When a valid `start` is read, the address counter, data pointers, etc., are reset and the state machine moves to waiting for data.

On each data read the address is incremented. When a match occurs, the data is stored in the local buffer. If it does not match, then the data is discarded. Once all the valid data has been read, then the state machine goes back to waiting for a BREAK.

The following example shows the interrupt function used on a PIC16F1947 (see Example 2).

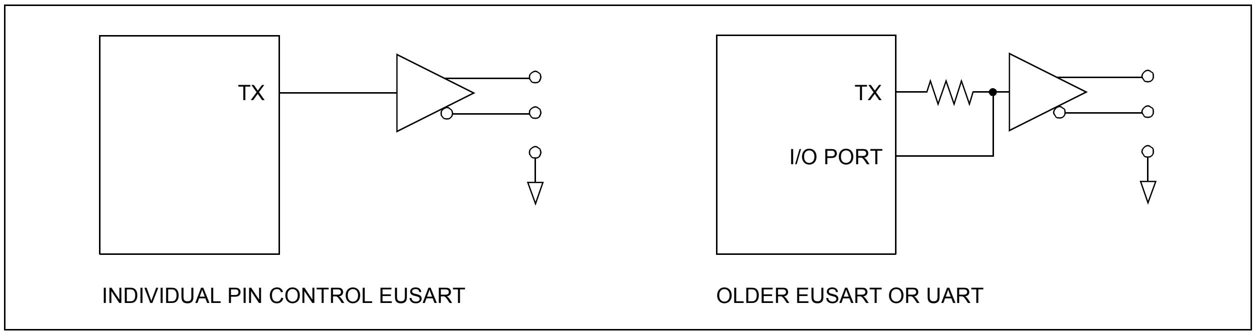
EXAMPLE 2: INTERRUPT FUNCTION

```
void dmx_rx_interrupt(void)
{
    uint8_t RxDat;

    if(RC1IE & RC1IF)
    {
        if(RC1STAbits.FERR)      // Check for Break - i.e. Framing Error
        {
           RxDat=RC1REG;                    // Clear the Framing Error
           DMX_Flags.RxBreak=1;             // Indicate a break
           DMX_RxState=RX_WAIT_FOR_START;// Go to START State
           DMX_RxTimer=0;                   // Clear the DMX timer (i.e. still working)
        }

        switch(DMX_RxState)
        {
            case RX_WAIT_FOR_BREAK:
                RxDat=RC1REG;    // Keep clearing the buffer until overflow.
                break;

            case RX_WAIT_FOR_START:
                if(DMX_RC1IF)    // make sure there is data available (i.e. not a break)
                {
                    RxDat=RC1REG;              // Read the data also clears RC1IF
                    if(RxDat==DMX_RxSTART)     // Check START Byte
                    {
                        // Valid START Received
                        DMX_RxState = RX_READ_DATA;     // Move to next state
                        DMX_RxDataPtr = &DMX_RxData[0]; // Point to Buffer
                        DMX_RxAddrCount = 0;            // Reset current addr
                        DMX_Flags.RxStart = 1;          // Indicate a command
                    }
                    else
                    {
                        DMX_RxState=RX_WAIT_FOR_BREAK; // INVALID! Wait for next BREAK
                    }
                }
                break;

            case RX_READ_DATA:
                RxDat=RC1REG;// Read Data
                if(DMX_RxAddrCount >= DMX_RxChannel) // Test if in range
                {
                    *DMX_RxDataPtr=RxDat;   // If in Range then Store is
                    DMX_RxDataPtr++;        // Point to next data in buffer
                }
                DMX_RxAddrCount++;

   // Check for end of valid data range
                if(DMX_RxAddrCount >= (DMX_RxChannel + DMX_RX_BUFFER_SIZE))
                {
                    DMX_Flags.RxNew=1;
                    DMX_RxState=RX_WAIT_FOR_BREAK;
                }
                break;
        }
    }
}
```

IMPLEMENTING THE DMX512 CONTROLLER

Sending data out is simple to do. The tricky part of the process is configuring the hardware to generate the BREAK. More recent EUSARTS allow individual control of the TX and RX I/O pins, and with these parts it is possible to disable only the TX pin. This allows a BREAK to be manually be generated using the same I/O pin. On the parts without this feature, it is easier to use a second I/O pin and a resistor, as shown in Figure 8. Setting the secondary pin low, then clearing the TRIS bit, overrides the normal TX pin operation. Setting the TRIS bit for that pin, then ends the break and passes control back to the USART.

FIGURE 8: CONTROLLER SCHEMATIC – WITH AND WITHOUT RESISTOR



The BREAK and MAB timing requirements are wide. This gives the option of using a soft-timed BREAK and MAB. Alternately, a timer can be used to accurately generate the BREAK and MAB, as well as providing a 1 ms tick and millisecond, minute and hour time information for general purpose use. Using the timer also makes it possible to run the entire transmit code in an interrupt.

IMPLEMENTING DMX512 HALF-DUPLEX COMMUNICATION

Half-duplex communication requires hardware that can be dynamically configured as receive or transmit. It also has active termination at the controller to correctly bias the data lines when the output is tri-stated. Typical controller and receiver circuits are shown below in Figure 9 and Figure 10.

FIGURE 9: HALF-DUPLEX CONTROLLER SCHEMATIC

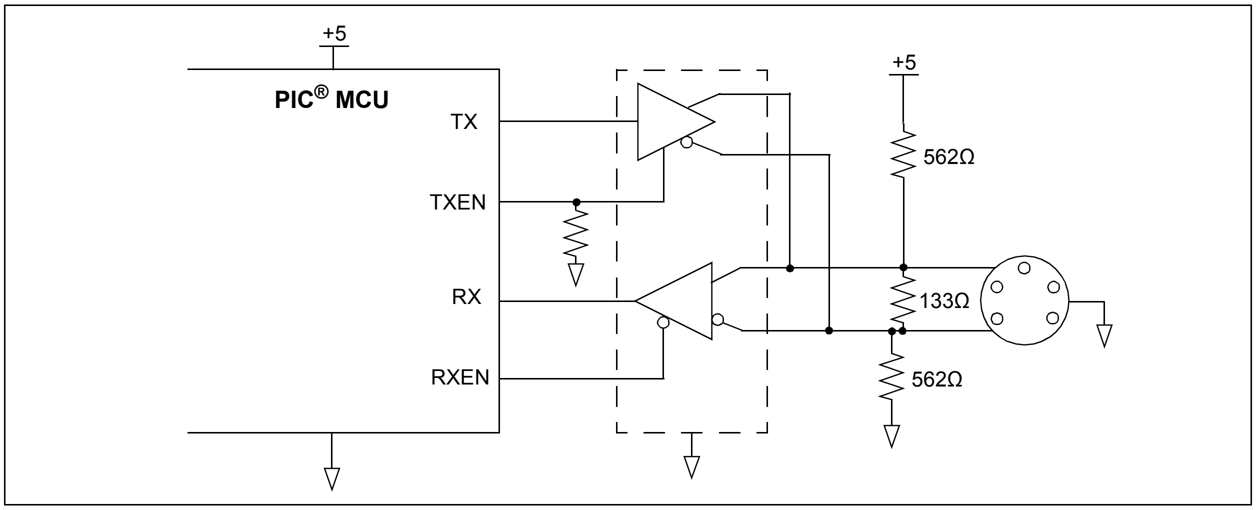
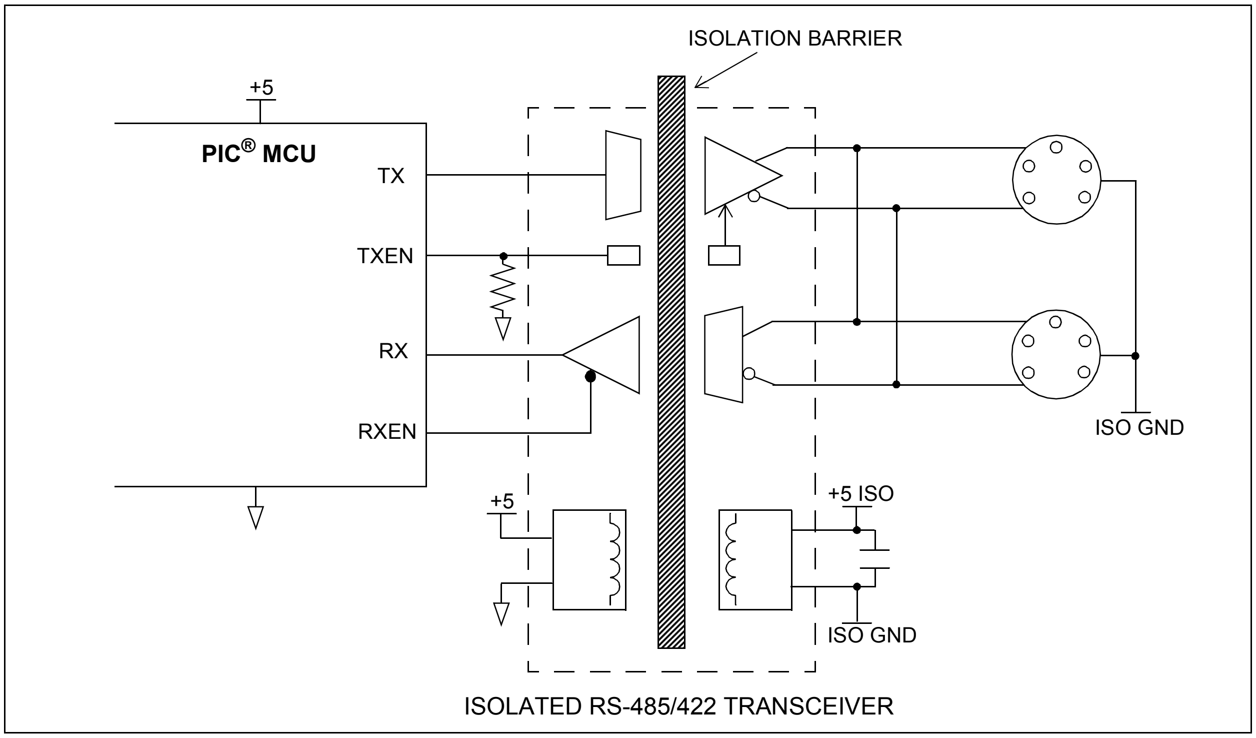


FIGURE 10: HALF-DUPLEX RECEIVER SCHEMATIC

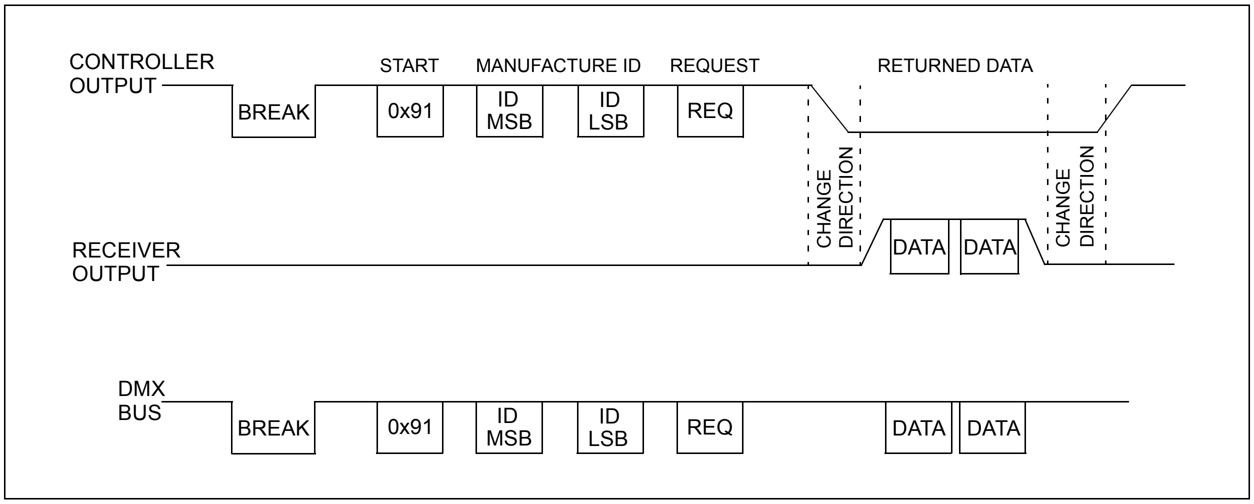


Initiating a read from a receiver requires an alternate START code and possibly a Manufacturer ID. These are issued by the standards committee. START codes `0xF0` to `0xF7` are allowed for experimental use while waiting for a registered START code to be assigned. Products cannot be shipped using these codes. Also, a START code of `0x91` is followed by a 2-byte Manufacturer ID that is issued by the standards committee and allows for more custom options.

Once the START code and/or Manufacturer ID have been sent, the protocol is completely open for the remainder of the frame. At some point, you may want to read from a receiver. To achieve this, you send a custom sequence or receiver address, change the direction of the transceiver on the controller, small delay, change direction on the receiver hardware, then send bytes from the receiver to the controller. Once the data has been sent, the controller and receiver switch the hardware back to normal operation. These frames must be interleaved with standard DMX512 frames to keep everything else on the bus running.

Implementing the half-duplex communication requires the addition of a simple USART receiver on the controller hardware and a USART transmitter on the receiver hardware. Detection of the alternate START code and appropriate Manufacture ID needs to be added so that the custom communications can begin.

FIGURE 11: EXAMPLE OF HALF-DUPLEX COMMUNICATION



Because the implementation is custom to each individual application, we will not go into any more detail here. The required functions to enable half-duplex communication may be found in a future release of the DMX512 library.

THE DMX512 LIBRARY

The library is completely interrupt-driven and requires the use of a EUSART and an one timer. The timer generates the BREAK and the MAB timing for the controller and the data time out for the receiver. The timer also generates millisecond, minute and hour ticks and counts that can be accessed for general use.

The library currently implements a DMX controller, receiver, or both, depending on how it is configured. A future version may add half-duplex communication capability and possibly a basic implementation of RDM.

Currently, the DMX512 Library is targeted at the Microchip Technology 8-bit microcontroller family with a EUSART, but can be easily ported to the 16-bit and 32-bit families and some non-EUSART parts.

Files:

- *dmxconfig.h*
- *dmx.h*
- *dmx.c*

dmxconfig.h is used to configure the various options that are available including the mode of operation, buffer sizes and some default values. Please copy the template to your own project and modify accordingly.

dmx.h is the common header file for the library API that is included in your C project files

dmx.c is the combined controller and receiver library you need to include in your project.

EXAMPLE CONTROLLER CODE

This is an example of a basic lighting console for manually controlling RGBW LEDs. The following code simply samples four sliding potentiometers and outputs it to the DMX512 bus.

EXAMPLE 3: SIMPLE DMX CONTROLLER SOURCE CODE

```
#include <xc.h>
#include “dmx.h”
#include “board.h”

void main(void)
{
     uint8_t R,G,B,W;

     OSCCON= 0b11110000;     // 4xPLL, 8MHz(32MHz), Config bits source

     board_init();          // Initialise the boards hardware, ADC etc
     dmx_init();            // Initialise DMX512

     GIE=1;                 // Enable the Interrupts

     while(1)
     {
         if(dmx_timer_ms())  // Use the DMX ms timer to update the reading
         {

            R=read_slider(CH_RED);
            G=read_slider(CH_GREEN);
            B=read_slider(CH_BLUE);
            W=read_slider(CH_WHITE);

            dmx_write_byte(0,R);
            dmx_write_byte(1,G);
            dmx_write_byte(2,B);
            dmx_write_byte(3,W);
        }
     }
}

void interrupt isr(void)
{
    dmx_interrupt();                        // Process the DMX512 interrupts
}
```

EXAMPLE RECEIVER CODE

The following code demonstrates how a RGBW fixture can be set up using the DMX512 Library. The address is set by a 9-bit switch (zero to 511) spread between PORTA and PORTB, and is only updated on Reset. The data received is in the order: Red, Green, Blue, White and output to the four PWMs.

EXAMPLE 4: SIMPLE DMX RECEIVER SOURCE CODE

```
#include <xc.h>
#include "dmx.h"
#include “board.h”

uint8_t RGBW[4];

void main(void)
{
     uint16_t Address;
     OSCCON= 0b11110000;

     board_init();
     dmx_init();


     Address = ((PORTB & 0x01)<<8) | PORTA;  // Read the 9bit switch
     dmx_set_address(Address);

     GIE=1;

     while(1)
     {
         if(dmx_new_data())
         {
             LED_DataToggle();   // Toggle LED to show data RX
             dmx_read(0,RGBW,4); // Read the data
             led_set_rgbw(RGBW[0],RGBW[1],RGBW[2],RGBW[3]); // Set the PWMs
         }
     }
}

void interrupt isr(void)
{
    dmx_interrupt();
}
```

As shown above, the DMX512 library is easy to use.

For full code samples based on the Lighting Communications Kit, please download the library from http://www.microchip.com/.

DMX512 DEVELOPMENT TOOLS

- *Lighting Communications Motherboard (DM160214)*
- *Lighting Communications Prototyping Board (AC160214)*
- *DMX512 Adapter (AC160214-2)*
- *USB to DMX512 Adapter (DM160214)*

APPENDIX A: DMX512A LIBRARY FUNCTIONS

EXAMPLE A-1: COMMON FUNCTIONS

```
void dmx_init(void)

Initializes the DMX512A Library
Please add near the beginning of you main() function before your main loop

void dmx_interrupt(void)

This processes all the DMX512 related Interrupts.
Please add this to your interrupt function.


void dmx_set_start(uint8_t start)

 This selects the DMX512A start code.
 For DMX_CONTROLLER this is the start code that is transmitted.
 For DMX_RECEIVER this is the start code that will be responded to.

 The dmx_init() function defaults this value to '0' which is the most
 common start code that is used.
Parameters:
 cmd - uint8_t - Sets the start code to use.
```

EXAMPLE A-2: DMX512A CONTROLLER FUNCTIONS

```
void dmx_tx_interrupt(void)

DMX Transmit interrupt.
This is called as part of dmx_interrupt() but it can call seperately for optimization.
Do not use this unless you really have to.


void dmx_tx_set_start(uint8_t cmd)

This selects the DMX512A start code for the controller
The dmx_init() function defaults this value to '0' which is the most common start code that is
used. This is needed if you are using custom start codes.

Parameter:
 uint8_t cmd - Sets the start code to use.


uint8_t dmx_tx_get_start(void)

Returns the DMX512A start code in use


void dmx_write_byte(uint16_t addr, uint8_t data)

This writes a single byte to the specific address in the TX buffer.

Parameters:
 uint16_t addr - range 1 to DMX_TX_BUFFER_SIZE. 0 is invalid.
 uint8_t data  - data to insert in the buffer at location addr


uint8_t dmx_tx_read_byte(uint16_t addr)

This Reads a byte from the tx buffer and is useful for manipulation of data.

Paramter:
  uint16_t addr -  Selects the address in the buffer to read from.
                       Range 1 to DMX_TX_BUFFER_SIZE. 0 in invalid.

Returns byte from the buffer at the address selected.


uint8_t dmx_tx_done(void)

Indicates when a packet has been sent and starting a new one. Self clears.
Returns 1 if new data sent or 0 if not.



void dmx_tx_enable(uint8_t enable)

Enables or disable the transmission sequence. i.e. Turns output on/off

Parameter:
 uint8_t enable - 1 = turn TX on, 0 = turn TX Off
```

EXAMPLE A-3: DMX512 CONTROLLER FUNCTIONS

```
uint8_t dmx_tx_get_enable(void)
 Returns 1 if tx is enabled, 0 if disabled


void dmx_write( uint16_t addr, uint8_t *data, uint8_t num)


Copies an array of data to the output buffer.

This buffer write is performed autonomously - i.e. interrupts disabled

Parameters:
  uint16_t addr - address in th ebuffer to start writing to
  uint8_t *data - pointer to the data to copy to the buffer
  uint8_t  num  - number of bytes to copy to the buffer. Limited range.
```

EXAMPLE A-4: DMX512 RECEIVER FUNCTIONS

```
void dmx_rx_interrupt(void)

DMX Receive interrupt.
Called as part of dmx_interrupt() but can call separately for optimization.
Do not call this function unless you really need it.


void dmx_rx_set_start(uint8_t cmd)

This selects the DMX512A start code for the receiver.
The dmx_init() function defaults this value to '0' which is the most common start code that is
used.

Parameter:
 uint8_t cmd - Sets the start code to use.



uint8_t dmx_rx_get_start(void)

 Returns the DMX512A start code in use


void dmx_set_address(uint16_t base)

Sets the base address to start reading data from.
Valid address range for DMX512 is 1 to 512  (But will work beyond the DMX range for custom
protocol)
An address of "0" will not respond to DMX data causing the DMX 1.2 second timeout to occur
effectively disabling DMX.

Parameter
  uint16_t base - The base address to use 1 to 512. 0 to disable RX.



uint16_t dmx_get_address(void)

Returns the base address that is in use.
```

EXAMPLE A-5: DMX512 RECEIVER FUNCTIONS

```
uint8_t dmx_new_data(void)

Indicates if new data has been received. Self clears.

Returns 0 for no new data, or the number of bytes of new data received.


uint8_t dmx_read_byte(uint8_t offset)

Reads a byte of the received data from the buffer.

Parameter:
 uint8_t offset - sets the offset into the RX buffer to read from
                      Range 0 to DMX_RX_BUFFER_SIZE-1

Returns the byte read from the RX Buffer at the offset address


uint8_t dmx_read( uint8_t offset, uint8_t *data, uint8_t num)

Reads a block of data from the receive buffer.

Parameters
 uint8_t offset - sets the offset into the RX buffer to read from
 uint8_t *data - Pointer to where to copy the data to
 uint8_t num - Number of bytes to copy. Limited to buffer range.

Returns Number of Bytes copied.
uint8_t dmx_rx_timeout(void)

Indicates if a data timeout has occurred.
Default trigger is 1200ms of no DMX data received which indicates an error, or a DMX512 system
shut down.
Changing the DMX_RX_TIMEOUT_MS changes the timeout limit (ms counter)

Returns 1 if timeout in effect, 0 if everything is still running
```

EXAMPLE A-6: TIMER FUNCTIONS

```
void dmx_timer_interrupt(void)

DMX Timer interrupt.
Called as part of dmx_interrupt() but can call separately for optimization.
Do not use this call unless you really need to.


uint8_t dmx_timer_ms(void)

DMX Timer millisecond tick.
Returns ‘0’ if tick has not occurred.
Returns ‘1’ is tick occurred since last time this function was called.


uint16_t dmx_ms_count(void)

Returns current ms count 0 to 999


void dmx_ms_clear(void)

Clears the current ms count


uint8_t  dmx_timer_sec(void)

One second tick.
Returns 0 if a one second tick has not occurred.
Returns 1 if a one second tick has occurred.

uint8_t dmx_sec_count(void)

Returns the current second count 0 to 59

void dmx_sec_clear(void)

Clears the second count.
```

EXAMPLE A-7: DMX CONFIGURATION TITLE

```
Please copy and rename dmxconfig_template.h from the dmx library directory to dmxconfig.h in
your project directory then set the parameters as required.

Following are the definitions and parameters available in the dmxconfig.h file with a brief
description of what they do. Please note these may change in future versions of the library so
check the information in the header file for any updates.

#define DMX   // Shows the DMX library is being used, must be uncommented.
Uncomment the modes of operation that you require.
It is possible to have both controller and receiver running at the same time.

//#define DMX_CONTROLLER       // Enables the controller functions
//#define DMX_RECEIVER           // Enables the receiver function
//#define DMX_BACKCHANNEL  // Not used in this version of the library
DMX_TIMER
This enables the timer interrupt features. If you disable this function you MUST write your own
function to provide the appropriate timing flags required by the DMX library.

 The best option is to leave it as is unless you really need to use the timer for something else!
In that case, please ensure you include what is required to keep the DMX running!
 i.e. copy then modify the existing timer interrupt section.

#define DMX_TIMER
```

EXAMPLE A-8: CONTROLLER SPECIFIC OPTIONS

```
 DMX_TX_BUFFER_SIZE
  Select the size of the transmit buffer.
  This is the Maximum number of channels that will be sent.
  DMX512A requires a minimum of 20 and a maximum of 512.
  This depends on how much RAM your PIC has available.

 #define DMX_TX_BUFFER_SIZE 128
```

EXAMPLE A-9: RECEIVER SPECIFIC OPTIONS

```
DMX_RX_BUFFER_SIZE

 Select the size of the receive buffer.
 This is the number of DMX512 Channels you will receive.
 For example a RGBW fixture may use 4 channels where a scanner may need 10.

#define DMX_RX_BUFFER_SIZE 8


DMX_RX_TIMEOUT_MS
  Selects the ms between breaks that sets the timeout flag.
  This is usually just over 1 second and is common for DMX Fixtures
  to go into a blackout or default mode if this occurs (i.e. signal lost)

#define DMX_RX_TIMEOUT_MS 1200
```

EXAMPLE A-10: HARDWARE SETTINGS

```
Direction pin is used on the demo board so that both Transmit and receive can be demonstrated
using a single SN75176 transceiver.

#define DMX_USE_DIR_PIN         // Define if using a direction pin
#define DMX_DIR_PIN     LATA1   // LAT for the direction pin
#define DMX_DIR_TRIS    TRISA1  // TRIS for the direction pin
#define DMX_DIR_RX      0       // Pin setting for RX mode (Based on SN75176)
#define DMX_DIR_TX      1       // Pin Setting for TX mode (Based on SN75176)

Define the TX and RX pins that we are using.

#define DMX_RX_PIN      LATB7   // RX pin used
#define DMX_RX_TRIS     TRISB7

#define DMX_TX_PIN      LATB6   // TX  pin used
#define DMX_TX_TRIS     TRISB6


Select the EUSART that will be used by redefining the USART registers and individual bits to DMX
USART registers. Some parts with different style USARTS may need changes here and in the dmx.c
code.

#define DMX_BAUDCON     BAUD1CON
#define DMX_RCSTA       RC1STA
#define DMX_FERR        RC1STAbits.FERR
#define DMX_OERR        RC1STAbits.OERR
#define DMX_TXSTA       TX1STA
#define DMX_TXEN        TX1STAbits.TXEN     // TX Enable Bit
#define DMX_TRMT        TX1STAbits.TRMT     // TX Shift reg status bit
#define DMX_RCREG       RC1REG  // RX Data register
#define DMX_SPBRGL      SP1BRGL
#define DMX_SPBRGH      SP1BRGH
#define DMX_TXREG       TX1REG  // TX data register

#define DMX_RCIE        RCIE
#define DMX_RCIF        RCIF
#define DMX_TXIE        TXIE
#define DMX_TXIF        TXIF
Calculate the Preload values for the EUSART Configuration
 Using BRG16=1 & BRGH=0
 BRGH:BRGL = (FOSC / BAUD / 16) - 1
          = (FOSC / 250000 / 16) -1
 For 32MHZ = (32M / 250K / 16) -1 = 7 = 0x0007

This needs to be automated but at the moment is manually calculated so if you use a different
frequency please recalculate these values.
#define DMX_SPBRGH_LOAD     0
#define DMX_SPBRGL_LOAD     7
#define DMX_BRG16           1       // BRG16=1 & BRGH=0 = Fosc/16 - 16bit H:L
#define DMX_BRGH            0       // BRG16=1 & BRGH=1 = Fosc/4  - 16bit H:L
```

EXAMPLE A-11: HARDWARE SETTINGS

```
#define DMX_BAUDCON_LOAD    0b00000000 | (DMX_BRG16<<3)
// Use 9 bit mode to give the 2 stop bits required - set bit 9 to '1'
#define DMX_STA_LOAD        0b01000001 | (DMX_USE_TX <<5) | (DMX_BRGH << 2 )
#define DMX_RCSTA_LOAD      0b10000000 | (DMX_USE_RX <<4)


Timer hardware options and load values. If you use a different timer then you will have to change
this and possibly the dmx.c source code.

#define DMX_TMRIE       TMR1IE
#define DMX_TMRIF       TMR1IF
#define DMX_TMR         TMR1
#define DMX_TCON        T1CON
#define DMX_TCON_LOAD   0b00110001          // Fosc/4, 1:8 pre
Reload values for the timer to generate 1ms, Break and MAB timing.
If you use a different frequency or BREAK/MAB timing then you will have to recalculate these.
32MHz / 4 / 8 = 1MHz 1/1MHz = 1us
1ms = 0xFFFF - 1000 = 0xFc17

#define DMX_TMR_LOAD_1MS     0xFc17  // Load value for 1ms

20us = 0xFFFF - 20 = 0xFFEB
#define DMX_TMR_LOAD_MAB     0xFFEB  // Load value for MAB      ( 20us)
a180us = 0xFFFF - 180 = 0xFF4B
#define DMX_TMR_LOAD_BREAK   0xFF4B  // Load Value for BREAK    (180us)

800us = 0xFFFF - 800 = 0xFCDF  - Adjust to fine tune the 1ms total
#define DMX_TMR_LOAD_FILL   0xFCDF   // Load value to total 1ms (800us)
```

NOTES:

ISBN: 978-1-63276-245-0

MICROCHIP

Worldwide Sales and Service

AMERICAS

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

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

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

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110

Canada - Toronto
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Dusseldorf
Tel: 49-2129-3766400

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Pforzheim
Tel: 49-7231-424750

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Venice
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Poland - Warsaw
Tel: 48-22-3325737

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

03/25/14