Application Note

AN2221/D
Rev. 1, 08/2002

MI Bus Software Driver for the MC9S12DP256

by Mark Houston
8/16 Bit Applications Engineering
Freescale, East Kilbride

Introduction

The MI Bus is a serial communications protocol that supports distributed real time control. It is suitable for medium speed networks requiring very low cost multiple wiring, providing high data integrity as a result of continuous push-pull communications from the master to the slaves. A single wire is used to connect the slave devices to the master, with up to 8 slave devices being able to connect at one time.

The MI Bus is suitable to be used to control smart switches, motors, sensors and actuators. In automotive applications the MI Bus can be used to control systems such as air conditioning, mirrors, seats, window lift and head light levellers.

This application note will describe software that will replicate a MI Bus Master and describe the operation of such software. The Slave will not be discussed in this document.

The MI Bus Concept

The MI Bus utilises a push/pull sequence to transfer data between the master and slaves. The master sends a push field to the slave devices connected to the bus. This field contains data, plus the address of one of the slaves. The slave addressed then responds to the received data, transmitting the Pull field on the MI Bus. The Master then collects this transmitted data from the bus. The data returned by the slave is likely to be status bits representing internal or external information.

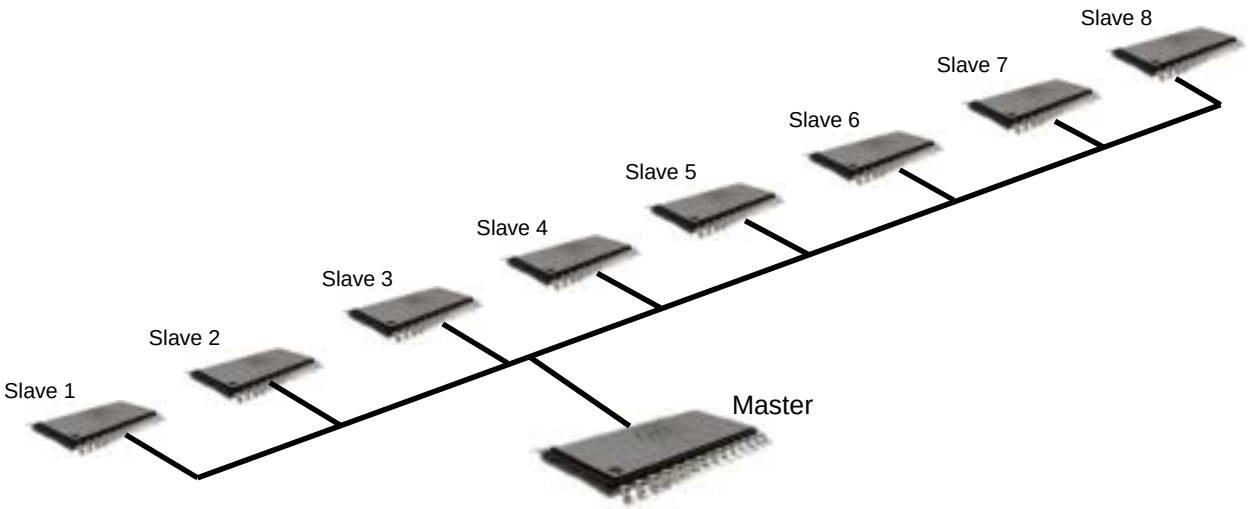


Figure 1. Basic MI Bus Topology

Push/Pull Sequence

Communication between the Master and Slave always uses the same frame organisation. The Master initialises the communication by transmitting serial data on the MI Bus; this is called the Push field. Once the Push field has been completed, the Slave device that has been addressed returns the values it holds. This response is called the Pull field.

Push Field

The Push field contains four sections. A start bit, a push synchronisation bit, a push data field and push address field. The start bit consists of 3 time slots held at logic zero, violating the rules of bi-phase encoding. The synchronisation bit consists of a Bi-phase encoded ‘0’, bi-phase encoding will be discussed later. The data field contains 5 bits of bi-phase encoded data, and the address field consists of 3 bits of bi-phase encoded data. The address data occupies the upper bits of the data register, with the data bits occupying the lower 5 bits.

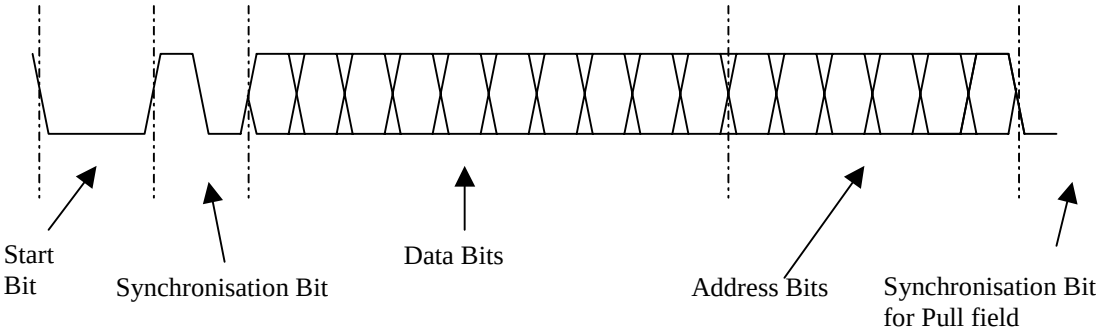


Figure 2. Push Field

Pull Field

The pull field contains three sections. A pull synchronisation bit. This consists of a bi-phase coded ‘1’ which is initiated by the Master in the time slot after the last address bit in the push field. The Pull data field, which comprises of three bits of Non Return to Zero coded transmission, with each bit taking a single time slot. The final section in the pull field is the End of Frame field. This is composed of a square wave signal typically having the frequency of 20KHz +/- 1% tolerance.

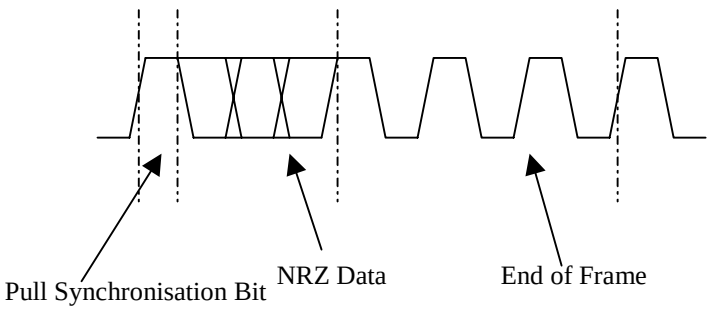


Figure 3. Pull Field

MI Bus Timing

Each of the level changes in the push/pull fields happens in a pre-determined time slot. For example the Start bit in the Push field requires three time slots. Similarly for the three data bits being returned by the slave on the Pull field, each of these must be a single time slot. Below is a diagram taken from the HC912D60 databook. This diagram shows how each of the time slots correlates to a single bit on the MI Bus. It also shows how the push/pull sequence affects the MI Bus Wire.

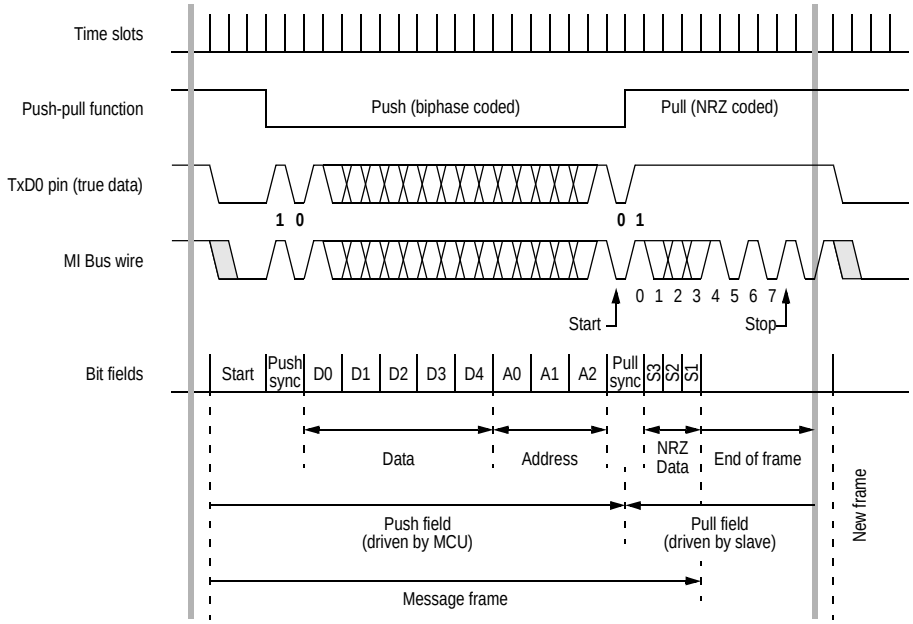


Figure 4. MI Bus Timing Diagram

To synchronise the Push/Pull data fields the synchronisation bits in the respective fields are used, along with the start bit of the Push field. The MI Bus Master transmits the frame data in the format described above, and awaits a response from the slave devices. At the end of the push field the Master takes the MI Bus Wire to the dominant state, initialising the first bit of the Pull synchronisation. The slave watches for the MI Bus wire to go low once its data register is full and knows to begin transmitting the pull field. The Master monitors the MI Bus wire watching for a rising edge one time slot in length after it has completed the push field, indicating that the Pull field is about to be transmitted at the next time slot. When the Master registers the MI Bus Wire going high for a single time slot it pulls the data from the MI Bus Wire.

Bi-phase Coding

The data and address bits in the push field are transmitted using biphase coding. By implementing biphase encoding it allows the detection of a single error at the time slot level. If data is received and both bits of the biphase-coded data contain only values one or zero, then an error has occurred.

The biphase encoding utilizes two time slots to transmit a single bit of data. The logic level “1” or “0” is determined by the order of the received bits; these are always complimentary logic levels of 0 and +5V.

The data is encoded as follows:

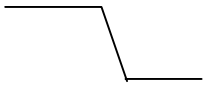
Logic Value to be Encoded	Voltage Levels	Description
0		Time slot 1 = +5V Time slot 2 = 0
1		Time slot 1 = 0 Time slot 2 = +5V

Figure 5. Biphase Encoding Values

MI Bus Clock Rate

The MI Bus Clock operates in a similar way to that of an SCI Clock. Both Master and Slave must be configured to receive and transmit data at the same clock speed. If they are not, synchronisation would be lost between Master and Slave and the data received would be incorrect.

Error Detection

There are several different types of errors that can occur within the MI Bus and be detected. These are not mutually exclusive.

Field Error

The communication between the Master and Slave is valid when the Master reads a pull data field having correct codes (excluding the codes of ‘111’ and ‘000’) followed by a square wave signal, having the frequency of 20 KHz, contained in the end of frame information.

An error occurs when the pull data field contains the code ‘111’ followed by the end of frame tied to logic 1. When this occurs communication between the Master and Slave was not valid. A Field Error can also occur when the Slave detected that the fixed form of the Push field has been violated.

Noise Detector

The Slave device that receives the push field samples the biphase-encoded data twice in each time slot. An error occurs when the sampled bits value do not match.

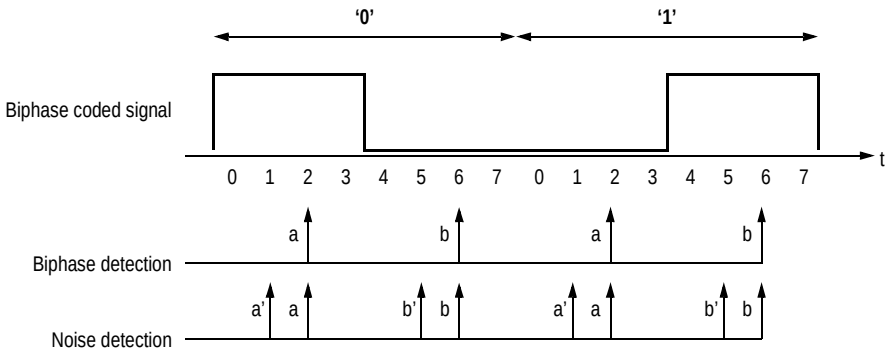


Figure 6. Bi – Phase coding and error detection by Slave

Biphase Detector

If, when tested, the data received by the slave does not follow an Exclusive-OR function the biphase data is incorrect, and an error has occurred. Please see Figure 6 - Bi – Phase coding and error detection by Slave.

Bit Error

The Master can monitor the MI Bus at the same time as transmitting data. An error is detected if the value transmitted on the MI Bus is different from the value that should be sent.

MI Bus Hardware Interface

The MI Bus consists of a single wire to transfer the data packets between the Master and the Slave. Only a few components are required to construct the interface to allow full MI Bus operation. The MI Bus interface is constructed using a single NPN transistor. The transistor serves both to drive the MI Bus during the push field and to protect the MCU TX from voltage transients generated in the wiring. Without this transistor Electro Magnetic Interference (EMI) could damage the TX pin on the Master device.

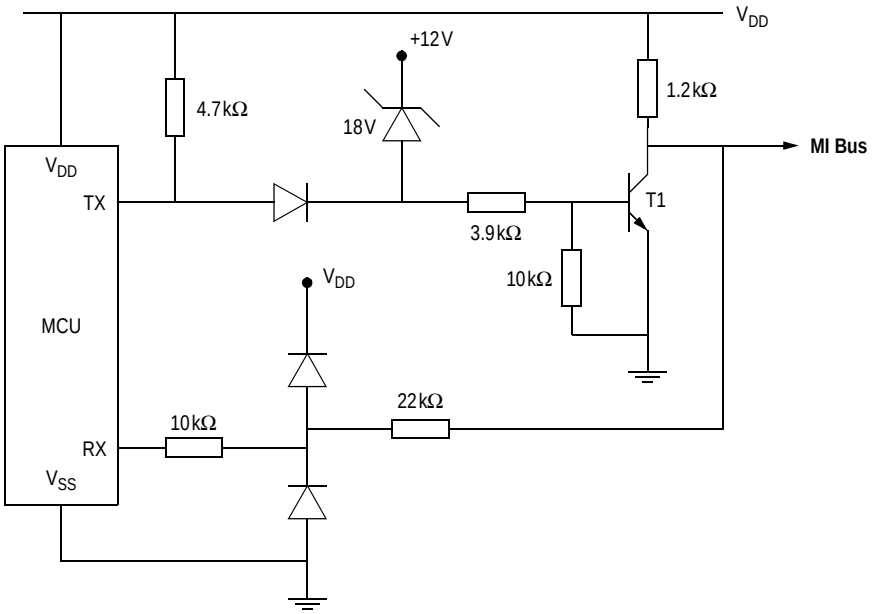


Figure 7. MI Bus Interface

The pin used to receive the pull field of the MI Bus is protected with two diodes and two resistors, which limit the value of the transient current generated by the EMI. The circuit clamps the voltage to the limit of the voltage supply (VDD and VSS) to the MCU. When a load dump occurs, the Zener diode is switched on and therefore turns on the transistor generating the logic ‘0’ on the MI Bus.

MI Bus Levels

The MI Bus line can have two states, recessive or dominant. The dominant state (‘0’) is represented by a maximum voltage of 0.3V. The recessive state (‘1’) is represented by +5V, through a pull-up resistor of 10Kohms.

Termination Network

The busload depends on the number of devices connected on the bus. Each device has a pull up resistor of 10Kohms. An external termination resistor is used to stabilise the load resistance of the bus at 600 Ohms.

Implementation of MI Bus on MC9S12DP256

The MI Bus concept has been taken and applied to the MC9S12DP256. The MC9S12DP256 has been designed to operate as the Master device in the MI Bus Network, with the software been written so that most of the features available on the hardware modules of the MI Bus have been replicated in software.

Development Environment

Metrowerks Codewarrior MOT Version 1.0 was used as the software development environment.

Feature Replication

There are some areas of the MI Bus hardware module that have not been replicated, due to either not being required or the function being redundant, but most of the functionality that is available on the MC68HC912D60 hardware module has been replicated.

Many of the registers and flags used in the hardware module of the MI Bus have been implemented in the design of the software driver. They are not all necessary in the current revision of the software, but allow the software to be easily integrated into another larger scale project that may require the flexibility they bring.

Omitted Features

The features that have not been included in the software design are detailed below.

Omitted Functionality

- The MI Bus can be run at many different bus speeds, but within the design of the software the bus was set to run at a nominal 20KHz. Adjusting the software timing could easily vary the bus speed but this functionality was not implemented.
- Serial communications interface STOP in WAIT mode. Not implemented.
- Send Break; no requirement
- MI Bus TxD0 polarity; no requirement.
- No test was implemented for the end of frame. The frequency of the MI Bus is running at 20KHz, therefore as long as the field data that have been sampled by the data receive function are correct, then the end of frame has been received.

Omitted Flags and Registers

- In Status Register 2, the flag for enabling the MI Bus Module is not required. Enabling the MI Bus function will be carried out by calling the MI Bus function.
- MI Bus Clock Rate Control Register, used to modify the MI Bus Clock speed.

MC9S12DP256 Hardware Features utilised.

Some of the hardware features found on the MC9S12DP256 were implemented in the design of the MI Bus Driver. These are described below.

- The PLL module was initialised to run the data bus at 25MHz.
- Port P was used for sending and receiving the data. Port P was selected, as this is one of three ports that allow interrupts to be generated by a rising or falling edge on the port. This is used to capture the data transmitted by the slave. Port P also allows the implementation of the Bit Error Test. A bit error is detected when the value monitored on the MI Bus does not match the value transmitted. By using the Port P wired – or mode this feature can be implemented.

The remaining features of the MI bus were implemented in software.

Software Design

The flow charts for the main software routines are shown below.

Main Routine

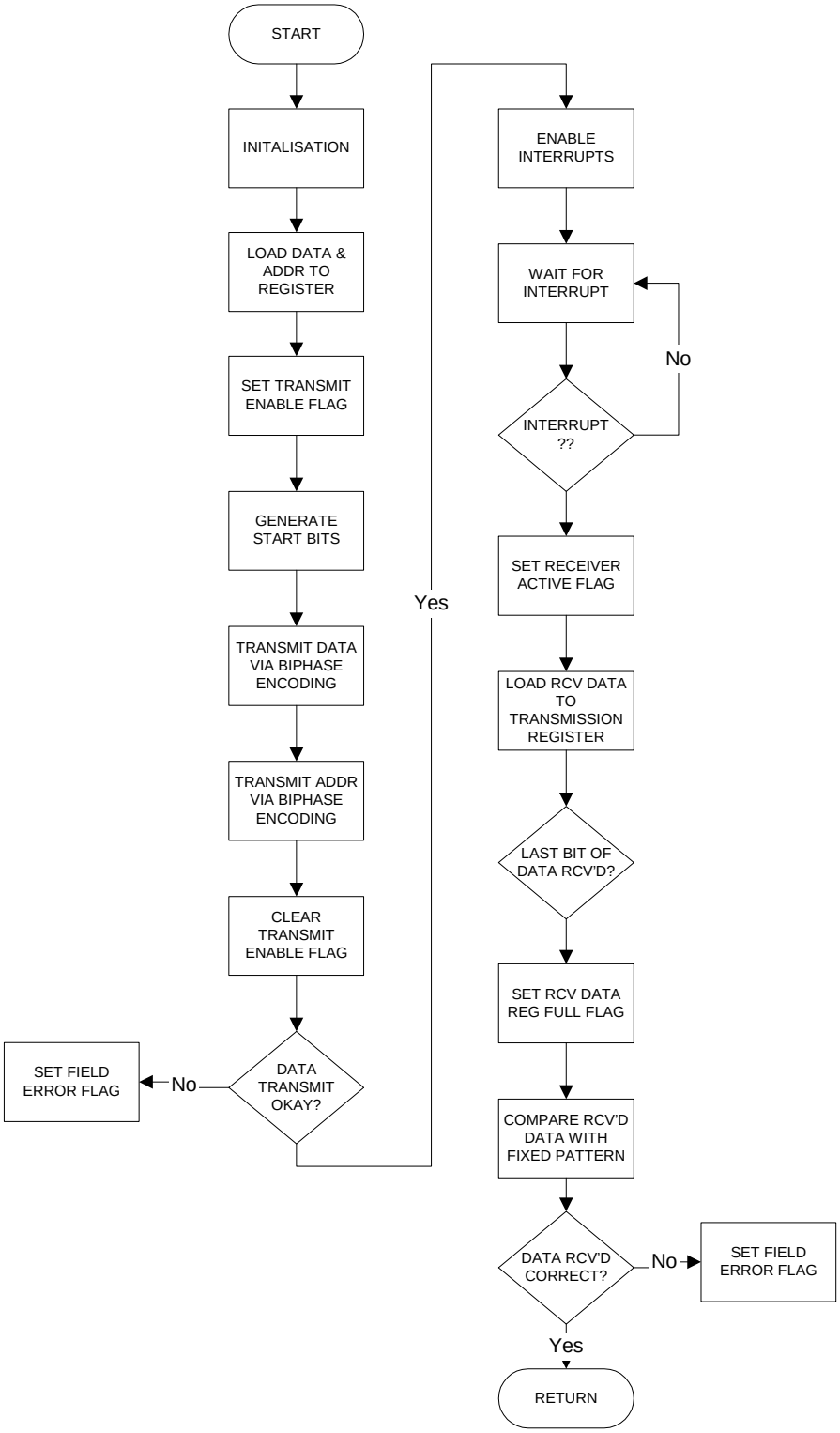


Figure 8. Main Flow chart

The flow chart above shows the flow of the main loop. The section on the left hand side depicts the transmit data, and the section on the left shows the receive data. The transmit data function would be called from within the main loop of the software, and the receive data would be generated by an interrupt initiated by the port pin receiving a rising edge.

Entry Conditions:Data and Address loaded to Data Register
Exit Conditions: None
Subroutine Calls:
Transmit_Start_Data
Transmit Sync Data
Load_Transmit_Val
Transmit_Biphase_Data

Transmit_Biphase_ Data Function

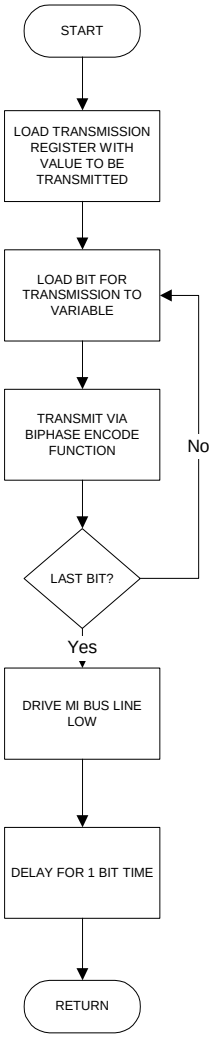


Figure 9. Data Transmission Flow Chart

The flow chart above shows the steps taken to transmit the data via the port pin. The software delay is used to set-up the time before the pull data is received.

Entry Conditions: None
Exit Conditions: None
Subroutine Calls: Biphase_Encode
Delay

Biphase_Encode Function

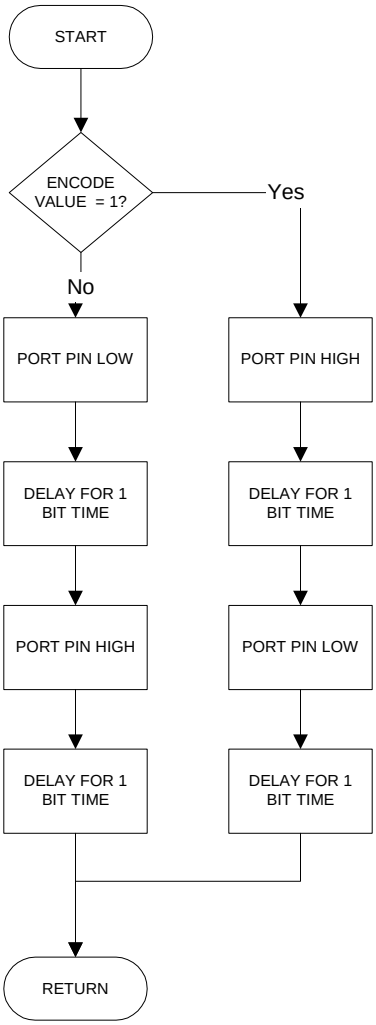


Figure 10. Biphase Encode function

The biphase encode function takes a bit value of either 1 or 0 and encodes this into the 2-bit biphase value. It then transmits these values via the port pin.

Entry Conditions: Encode_Value contains bit to be encoded
Exit Conditions: None
Subroutine Calls: Delay

Collect_Data Function

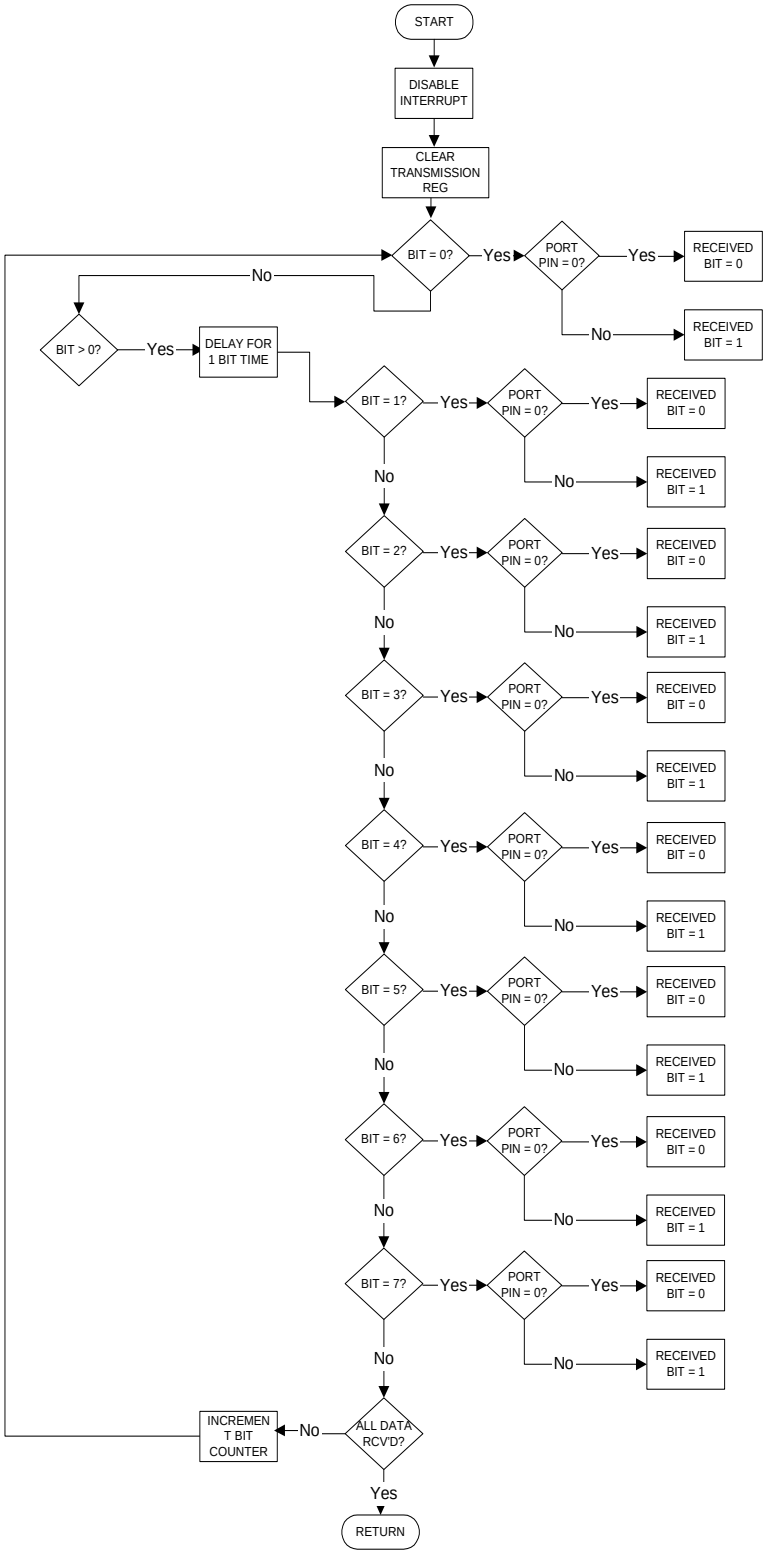


Figure 11. Collect_Data Flow chart

The collect data function reads the data on the port pin as it is received. This function is called by the interrupt service routine. On the first pass of the loop the function reads the value of the port pin without any delay. The latency in calling the interrupt service routine places the read time of the received data almost in the middle of the first data bit, therefore reducing the likelihood of the data being read on the port pin going out of synchronisation with the data being received. On the next pass of the loop the data read is delayed by a bit time to keep it in synchronisation with the data. The time taken to complete the loop is negligible, therefore the delay is set to approximately 25µS for a 20KHz signal.

Entry Conditions: None
Exit Conditions None
Subroutine Calls: Delay

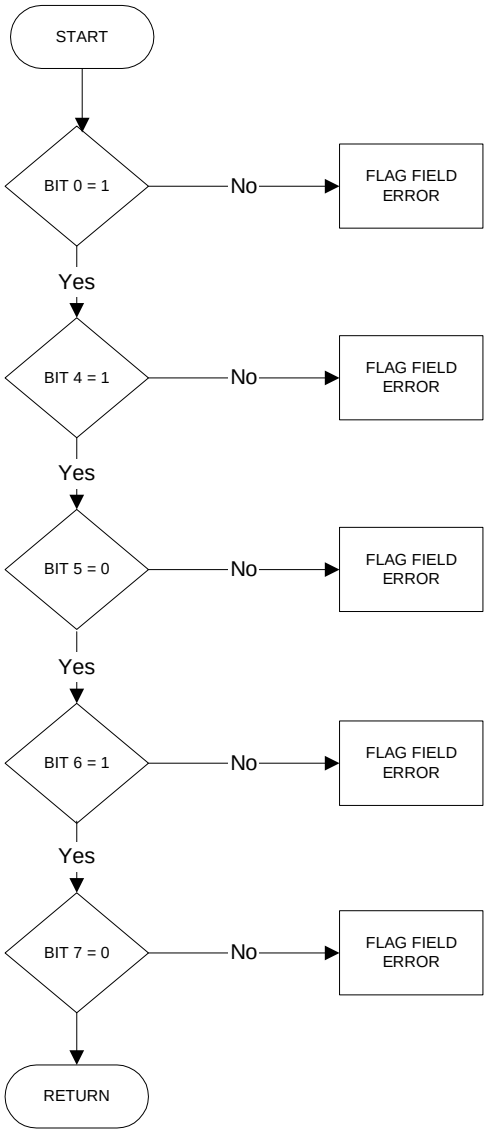


Figure 12. Field Test

The final flow chart shows the field test. Each bit of the fixed form field is tested to ensure that it is the correct value, if not an error is then flagged.

Entry Conditions: None
Exit Conditions: Pass or Fail
Subroutine Calls: None

Summary

The Freescale Interconnect Bus is a basic communication protocol that allows communication between Master and Slaves to occur easily and reliably. As shown by the implementation of the MI Bus Software Driver the device no longer has to have the hardware modules designed into the device to allow communication to be carried out via the MI Bus protocol, therefore allowing many different devices to implement the advantages associated with the MI Bus.

The suggested functionality shown in this application note is simple to implement, however, there are additional features that can enhance the system further. For example by using the timer channels on board the MC9S12DP256 the accuracy of the delays used to create the waveforms could be increased. This would further reduce the likelihood of incorrect data being received.

By decreasing the time for the delays implemented within the source code, the baud rate could be increased to allow faster communication between Master and Slave. This has the obvious advantages associated with higher communication rates.

In conclusion the MI Bus Software Driver is a useful building block for implementing a reliable communication protocol between a master device and several slave devices. There are a few alterations that could be made to increase the functionality of the driver further but a basic reliable structure has been implemented that can be built upon.

Code Implementation

The source code to implement the MI Bus Software Driver follows, including header file.

Main Program Routine

```
/*****************************************************************************
Function Name               :           main
Engineer                    :           M.Houston
Date                        :           10/11/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           main routine calls initialisation routines

*****************************************************************************/
int main(void)
{

                    static tU16          count = 0;
                                         /* clear user defined flags */
                    SystemFlags.byte = 0x00;
                                         /* macro for 'asm sei' */
                    DisableInterrupts;
                                         /* Disable COP */
                    Crg.copctl.byte = 0;
                                         /* Disable IRQ */
                    Regs.intcr.bit.irqen = 0;
                                         /* map registers at default location */
                    Regs.initrg.byte = 0;

                    InitPorts();


                    /* configure pll with default multiplier & divider values */
#ifdef USING_PLL
                    if( SetPll() == FAIL)
                             {
                                         /* display pll error flags on led array */
                             Regs.portb.byte = ~(SystemFlags.byte & 0x03);
                             while(FOREVER);
                             }
#endif /* USING_PLL */


                                         /* Main program */

                             ControlReg2.bit.te =1;     /* Set Transmiter Enable Flag in Control Reg 2 */

                             Transmit_Start_Data();              /* Start Bits */
                             Transmit_Sync_Data ();              /* Synchronisation Bit */

                             Data = 0x15;               /* set value of data and address variables */
                             Addr = 0x02;


                             Load_Transmit_Val(Addr,Data);       /* Load MI Bus Data Reg */
                             Transmit_Biphase_Data();            /* Data transmission */

                             ControlReg2.bit.te =0;     /* Clear Transmitter Enable Flag in Control Reg 2 */
```

```
                        EnableInterrupts;                   /* macro for 'asm cli' */

                        ControlReg2.bit.rei =1;             /* Set MI Bus interrupt enable Flag */
                        ControlReg2.bit.re = 1;             /* Set MI Bus receiver enable Flag */

                        KnightRider();
}
```

Load Transmit Value Routine

```
/*************************************************************************
Function Name               :           Load_Transmit_Val
Engineer                    :           M.Houston
Date                        :           23/10/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Loads data into the Transmit_Val to allow transmission of
                                        data patterns and address data. BITS 4:0 are data pattern,
                                        BITS 7:5 are address data.

*************************************************************************/
void Load_Transmit_Val(unsigned char Address_Pattern, unsigned char Data_Pattern)
{
                  DataRegLow.bit.adr = Address_Pattern;
                  DataRegLow.bit.data = Data_Pattern;


}
```

Transmit Biphase Data Routine

```
/*************************************************************************
Function Name               :           Transmit_Biphase_Data
Engineer                    :           M.Houston
Date                        :           18/10/01

Parameters                  :           Transmit_Val
Returns                     :           NONE
Notes                       :           Transmits data contained in TransmissionByte register.
                                        The TransmissionByte0 data consists of 5 bits of data (4:0)
                                        and 3 bits of address data (7:5). This data is transmitted
                                        via manchester encoding, 10 times slots for data and 6
                                        time slots for address data.

*************************************************************************/
void Transmit_Biphase_Data(void)
{
                    unsigned char Transmit_Val;

                    TransmissionByte.byte = DataRegLow.byte;

                    /* Transmit bits via manchester encoding */
                    Transmit_Val = TransmissionByte.bit.bit0;           /* Load value of bit to be sent */
                    Biphase_Encode(Transmit_Val);                   /* Encode data and transmit on portp*/

                    Transmit_Val = TransmissionByte.bit.bit1;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit2;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit3;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit4;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit5;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit6;
                    Biphase_Encode(Transmit_Val);

                    Transmit_Val = TransmissionByte.bit.bit7;
                    Biphase_Encode(Transmit_Val);

                    Pim.ptp.bit.ptp0 = 0; /* data line low */

                    /* Set data direction register to input */
                    Pim.ddrp.byte = 0x00;

                    Delay(delayRate2, delayDecVal2); /* delay for 1 bit time for 20 KHz waveform*/

}
```

Transmit Start Data Routine

```
/*************************************************************************
Function Name               :           Transmit_Start_Data
Engineer                    :           M.Houston
Date                        :           18/10/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Start bit of MI Bus protocol. Consists of 3 time slots
                                        held at 0. This violates the rules of biphase encoding.

*************************************************************************/
void Transmit_Start_Data (void)
{
                    Pim.ptp.bit.ptp0 = 0;
                    Delay(3, delayDecVal2); /* value passed to delay routine for 150uS delay */
}
```

Transmit Synchronisation Data Routine

```
/*************************************************************************
Function Name               :           Transmit_Sync_Data
Engineer                    :           M.Houston
Date                        :           18/10/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Synchronisation bit for MI Bus protocol. Consists of a
                                        biphase encoded 0.

*************************************************************************/
void Transmit_Sync_Data (void)
{
                    Pim.ptp.bit.ptp0 = 1;
                    Delay(delayRate2, delayDecVal2);
                    Pim.ptp.bit.ptp0 = 0;
                    Delay(delayRate2, delayDecVal2);
}
```

Biphase Encode Routine

```
/*************************************************************************
Function Name               :           Biphase_Encode
Engineer                    :           M.Houston
Date                        :           19/10/01

Parameters                  :           Encode_Value
Returns                     :           NONE
Notes                       :           Routine to encode data in manchester format.
                                        Value 0 = transition from 1 to 0
                                        Value 1 = transition from 0 to 1

*************************************************************************/
void Biphase_Encode(int Encode_Value)
{
                    if(Encode_Value == 0)
                    {
                            Pim.ptp.bit.ptp0 = 1;
                            Delay(delayRate2, delayDecVal2);

                            /* read of PTIP register to ensure value is correct, no short circuit*/
                            if( Pim.ptip.bit.ptip0 == 0)
                                    {
                                            /* display error flags on led array */
                                            Regs.portb.byte = (SystemFlags.byte & 0x03);
                                            while(FOREVER);
                                    }

                            Pim.ptp.bit.ptp0 = 0;
                            Delay(delayRate2, delayDecVal2);

                            /* read of PTIP register to ensure value is correct, no short circuit*/
                            if( Pim.ptip.bit.ptip0 == 1)
                                    {
                                            /* display error flags on led array */
                                            Regs.portb.byte = (SystemFlags.byte & 0x03);
                                            while(FOREVER);
                                    }

                    }
                    else
                    {
                            Pim.ptp.bit.ptp0 = 0;
                            Delay(delayRate2, delayDecVal2);

                            /* read of PTIP register to ensure value is correct, no short circuit*/
                            if( Pim.ptip.bit.ptip0 == 1)
                                    {
                                            /* display error flags on led array */
                                            Regs.portb.byte = (SystemFlags.byte & 0x03);
                                            while(FOREVER);
                                    }

                            Pim.ptp.bit.ptp0 = 1;
                            Delay(delayRate2, delayDecVal2);

                            /* read of PTIP register to ensure value is correct, no short circuit*/
                            if( Pim.ptip.bit.ptip0 == 0)
                                    {
                                            /* display error flags on led array */
                                            Regs.portb.byte = (SystemFlags.byte & 0x03);
                                            while(FOREVER);
                                    }
                    }
}
```

Collect Data Routine

```
/*************************************************************************
Function Name               :           Collect_Data
Engineer                    :           M.Houston
Date                        :           7/11/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Function to collect data value from portp0, and store in
                                        the approriate bit of the 8 bit TransmissionByte Register.

*************************************************************************/
void Collect_Data(void)
{
                    char y;
                    y = 0;

                    /* Disable Interrupts */
                    DisableInterrupts

                    /* Disable interrupts on port p */
                    Pim.piep.byte = 0x00;

                    /* Clear MI Bus Interrupt enable Flag */
                    ControlReg2.bit.rei =0;

                    /* Set MI Bus Receiver Active Flag */
                    StatusReg2.bit.raf =1;

                    /* Clear Receive data register full flag */
                    StatusReg1.bit.rdrf =0;

                    TransmissionByte.byte = 0x00;

                    do
                    {

                    if(y == 0) /* test for DataRcv Reg Bit */
                    {

                              Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
                              if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
                              {
                                        TransmissionByte.bit.bit0 = 0;
                              }
                              else
                              {
                                        TransmissionByte.bit.bit0 = 1;
                              }

                              Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
                    }

                    else if(y>0)
                    {
                              Delay(delayRate4, delayDecVal4); /* delay until the middle of next bit */

                              if(y == 1) /* test for DataRcv Reg Bit */
                              {
                                        Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
                                        if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
                                        {
                                                  TransmissionByte.bit.bit1 = 0;
                                        }
                                        else
                                        {
```

```
                        TransmissionByte.bit.bit1 = 1;
            }
            Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
}


else if(y == 2) /* test for DataRcv Reg Bit */
{
            Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
            if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
            {
                        TransmissionByte.bit.bit2 = 0;
            }

            else
            {
                        TransmissionByte.bit.bit2 = 1;
            }
            Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
}

else if(y == 3) /* test for DataRcv Reg Bit */
{
            Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
            if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
            {
                        TransmissionByte.bit.bit3 = 0;
            }

            else
            {
                        TransmissionByte.bit.bit3 = 1;
            }
            Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
}

else if(y == 4) /* test for DataRcv Reg Bit */
{
            Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
            if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
            {
                        TransmissionByte.bit.bit4 = 0;
            }

            else
            {
                        TransmissionByte.bit.bit4 = 1;
            }
            Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
}

else if(y == 5) /* test for DataRcv Reg Bit */
{
            Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
            if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
            {
                        TransmissionByte.bit.bit5 = 0;
            }

            else
            {
                        TransmissionByte.bit.bit5 = 1;
            }
            Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
}

else if(y == 6) /* test for DataRcv Reg Bit */
{
```

```
                        Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
                        if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
                        {
                                TransmissionByte.bit.bit6 = 0;
                        }

                        else
                        {
                                TransmissionByte.bit.bit6 = 1;
                        }
                        Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
                }

                else if(y == 7) /* test for DataRcv Reg Bit */
                {
                        Regs.porte.bit.pte7 = 1; /* toggle port to allow debug */
                        if(Pim.ptp.bit.ptp1 == 0) /* test for value of port pin */
                        {
                                TransmissionByte.bit.bit7 = 0;
                        }

                        else
                        {
                                TransmissionByte.bit.bit7 = 1;
                        }
                        Regs.porte.bit.pte7 = 0; /* toggle port to allow debug */
                }
        }


        y++; /* increment y to next datarcv reg bit */
        }while (y<8); /* DataRcv Reg is 16 bit */

        /* Set Receive data register full flag */
        StatusReg1.bit.rdrf =1;

        /* Clear MI Bus Receiver Active Flag */
        StatusReg2.bit.raf =0;

        /* Clear receiver enable flag */
        ControlReg2.bit.re =0;

        /* test received data to ensure matches fixed field */

        if( Field_Test() == FAIL)
                {
                        /* display error flags on led array */
                Regs.portb.byte = (SystemFlags.byte & 0x03);
                while(FOREVER);
                }

}
```

Field Test Routine

```
/*************************************************************************
Function Name               :           Field_Test
Engineer                    :           M.Houston
Date                        :           20/11/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Function to verify data received from master is correct.
                                        The function will test to ensure that the data received
                                        in the transmission register conforms to the following
                                        pattern:

                                        Bit0 = 1
                                        Bit1 = X
                                        Bit2 = X
                                        Bit3 = X
                                        Bit4 = 1
                                        Bit5 = 0
                                        Bit6 = 1
                                        Bit7 = 0

                                        Bit 1 represents the synchronisation bit from the slave.
                                        Bits 1 to 3 represent the data bits from slave.
                                        Bits 4 to 7 represent the end-of-frame

*************************************************************************/
int Field_Test (void)
{
                    if(TransmissionByte.bit.bit0 == 0)
                    {
                            return FAIL;
                    }
                    else if(TransmissionByte.bit.bit4 == 0)
                    {
                            return FAIL;
                    }
                    else if(TransmissionByte.bit.bit5 == 1)
                    {
                            return FAIL;
                    }
                    else if(TransmissionByte.bit.bit6 == 0)
                    {
                            return FAIL;
                    }
                    else if(TransmissionByte.bit.bit7 == 1)
                    {
                            return FAIL;
                    }
                    else
                    {
                            return PASS;
                    }

}
```

Delay Routine

```
/*****************************************************************************
Function Name      :        Delay
Engineer           :           M.Gallop
Date               :           02/06/00

Parameters         :           int delayTime
Returns            :           NONE
Notes              :           simple software delay dependent on CPU clock frequency and
                               compile strategy
*****************************************************************************/
void Delay(int delayTime, int delayTime2)
{
                   int x;              /*outer loop counter */
                   char y;             /*inner loop counter */

                   for (x=0; x<delayTime; x++)
                   {
                            for (y=0; y<delayTime2; y++);
                   }
}
```

Initialisation Routine

```
/*****************************************************************************
Function Name               :           InitPorts
Engineer                    :           M.Houston
Date                        :           10.10.01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Code to set up i/o ports
*****************************************************************************/
void InitPorts(void)
                    {
                                        /* PORT A */
                    Regs.porta.byte = 0x00;
                                        /* All pins outputs on initialisation */
                    Regs.ddra.byte = 0xFF;

                                        /* PORT B */
                                        /* driving cathode of LEDs so '1' = off and 0xFF = all off*/
                                        /* Connected to LED Array on EVB, used for status flag */
                    Regs.portb.byte = 0xFF;
                                        /* all pins o/p for LEDs */
                    Regs.ddrb.byte = 0xFF;

                                        /* PORT E */
                                        /* select pin PE4 for E clock o/p */
                    Regs.pear.bit.neclk = 0;
                                        /* set port e to output for debug */
                    Regs.ddre.byte = 0xFF;


                                        /* PORT P */
                    /* set port p to receive data */
                                        /* Clear port input register */
                    Pim.ptp.byte = 0x00;
                                        /* All pins output on portp except ptp1*/
                    Pim.ddrp.byte = 0xFD;
                                        /* Set polarity register to rising edge on ptp1 */
                    Pim.ppsp.byte = 0x02;
                                        /* Set pull up/down on pin */
                    Pim.perp.byte = 0xFF;
                                        /* Enable interrupts on port ptp1 */
                    Pim.piep.byte = 0x02;

}
```

PLL Initialisation Routine

```
/*****************************************************************************
Function Name               :           SetPll
Engineer                    :           M.Gallop
Date                        :           18/07/01

Updated                     :           11.10.01 by Mark Houston
Notes                       :           Removed control of PLL by DIP switch on portH. Value of
                                                multiplier and divider declared in header file.

Parameters                  :           N/A

Returns                     :           int (PASS or FAIL)
Notes                       :           PLL frequency will be OSCCLK freq * multiplier / divider.
                                        The routine as implemented uses the sofware delay routine
                                        to generate a pll timeout in case lock fails.
                                        NOTE: It is the responsibility of the user to ensure that
                                        the multiplier and divider values do not cause the MCU bus
                                        specification to be exceeded.

*****************************************************************************/
int SetPll(void)
                  {
                  char x;
                                        /* clear error flags */
                  SystemFlags.bit.pllRangeError = 0;
                  SystemFlags.bit.pllLockFailed = 0;

                                        /* select XTAL/2 as cpu clock */
                  Crg.clksel.bit.pllsel = 0;
                                        /* Turn off PLL */
                  Crg.pllctl.bit.pllon = 0;
                              /* Set new OSCCLK frequency multiplier; for 25MHz Bus SYNR= 0x18 */
    Crg.synr.byte = 0x18;
                              /* Set new divider value; for 25MHz Bus REFDV = 0x03 */
    Crg.refdv.byte = 0x03;
                                        /* Turn on PLL */
                  Crg.pllctl.bit.pllon = 1;
                                        /* and wait for lock */
                  for( x=0; x<100; x++)
                           {
                           if( Crg.crgflg.bit.lock )
                                        {
                                       /* pll locked ok! */
                                        /* select PLL as cpu clock  */
                           Crg.clksel.bit.pllsel = 1;
                                        return PASS;     /* escape! */
                                        }
                                        Delay(10, 100);
                  }
                                        /* pll failed to lock! */
                                        /* Turn off pll */
                  Crg.pllctl.bit.pllon = 0;
            /* flag that pll didn't lock */
                  SystemFlags.bit.pllLockFailed = 1;
                  return FAIL;
}
```

Interrupt Service Routines

```
/*****************************************************************************
Function Name               :           _dummyISR
Engineer                    :           M.Gallop
Date                        :           25/07/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Interrupt service routine for unused interrupt vectors.
*****************************************************************************/
#pragma TRAP_PROC /* Mark Function as Interrupt Function */
void _dummyISR( void )
{
                                                    /* endless cycle */
    while( 1 );
}

/*****************************************************************************
Function Name               :           _portpISR
Engineer                    :           M.Houston
Date                        :           13/11/01

Parameters                  :           NONE
Returns                     :           NONE
Notes                       :           Interrupt service routine for portp.
*****************************************************************************/
#pragma TRAP_PROC /* Mark Function as Interrupt Function */
void _portpISR( void )
{
                                                        /* collect data from pin */
    Collect_Data();

}
```

Header File for MC9S12DP256 MI Bus Software Driver

```
/*****************************************************************************
*                                          COPYRIGHT (c) M
* FILE NAME: basic.h
*
* PURPOSE: header file for basic.c
*
*  ************************************************************************
*  *  THIS CODE IS ONLY INTENDED AS AN EXAMPLE OF CODE FOR THE           *
*  *  METROWERKS COMPILER AND THE STAR12 EVB AND HAS ONLY BEEN GIVEN A   *
*  *  MIMIMUM LEVEL OF TEST. IT IS PROVIDED 'AS SEEN' WITH NO GUARANTEES *
*  *  AND NO PROMISE OF SUPPORT.                                         *
*  ************************************************************************
*
*
* DESCRIPTION:  definitions for 'basic' application parameters
*
*
* AUTHOR: Martyn Gallop    LOCATION: EKB Apps     LAST EDIT DATE: 24.07.01
*
* UPDATE HISTORY
* REV             AUTHOR               DATE           DESCRIPTION OF CHANGE
* ---             ------               ---------      --------------------
* 1.0             M.Gallop             15/07/01       Original coding
* 1.1             M.Houston            11.10.01       MI Bus Implementation
*
*****************************************************************************/


/*===========================================================================*/
/* Freescale reserves the right to make changes without further notice to any */
/* product herein to improve reliability, function, or design. Freescale does */
/* not assume any  liability arising  out  of the  application or use of any */
/* product,  circuit, or software described herein;  neither  does it convey */
/* any license under its patent rights  nor the  rights of others.  Freescale */
/* products are not designed, intended,  or authorized for use as components */
/* in  systems  intended  for  surgical  implant  into  the  body, or  other */
/* applications intended to support life, or  for any  other application  in */
/* which the failure of the Freescale product  could create a situation where */
/* personal injury or death may occur. Should Buyer purchase or use Freescale */
/* products for any such intended  or unauthorized  application, Buyer shall */
/* indemnify and  hold  Freescale  and its officers, employees, subsidiaries, */
/* affiliates,  and distributors harmless against all claims costs, damages, */
/* and expenses, and reasonable  attorney  fees arising  out of, directly or */
/* indirectly,  any claim of personal injury  or death  associated with such */
/* unintended or unauthorized use, even if such claim alleges that  Freescale */
/* was negligent regarding the  design  or manufacture of the part. Freescale*/
/* and the Freescale logo* are registered trademarks of Freescale Semiconductor, Inc..         */
/*****************************************************************************/


/*Include files */

#include <hidef.h>
#include "peripherals.h"


/* additional common definitions  - others can be found in HIDEF.H*/

#define OFF     0
#define ON      1

#define FOREVER 1

#define FAIL                0
#define PASS    1
```

```
/* User Defines */

/* initial values for pll */
/* modify depending on pll filter selection and target clock frequency */
/* PLLCLK = OSCCLK */
#define CLOCK_MULTIPLIER 4
#define CLOCK_DIVIDER 4


/* Function Prototypes */

int                main(void);
void InitPorts(void);
int  SetPll(void);
void Delay(int, int);                                   /* simple software loop delay */
void Transmit_Start_Data (void);
void Transmit_Sync_Data (void);
void Biphase_Encode(int);
void Transmit_Biphase_Data(void);
void KnightRider(void);
void Load_Transmit_Val(unsigned char,unsigned char);
int  Field_Test (void);

/* Interrupt service routine Prototypes */

#pragma CODE_SEG NON_BANKED

void _dummyISR(void);
void _portpISR(void);

#pragma CODE_SEG DEFAULT_ROM


/* User Typedefs */

typedef union               /* global system flags byte - individual bits are */
                  {                                     /* assigned as tokens for tasks */
                  tU08     byte;
                  struct
                           {
                           tU08 pllLockFailed                 :1;                    //pll error flag
                           tU08 pllRangeError                 :1;                    //pll error flag
                  tU08                                        :6;                          //not used
                  }bit;
                  }tFLAGS;

typedef union               /* global variable bit access */
                  {
                  tU08     byte;
                  struct
                           {
                           tU08 bit0                                          :1; /* bit0 of variable */
                           tU08 bit1                                          :1; /* bit1 of variable */
                           tU08 bit2                                          :1; /* bit2 of variable */
                           tU08 bit3                                          :1; /* bit3 of variable */
                           tU08 bit4                                          :1; /* bit4 of variable */
                           tU08 bit5                                          :1; /* bit5 of variable */
                           tU08 bit6                                          :1; /* bit6 of variable */
                           tU08 bit7                                          :1; /* bit7 of variable */
                           }bit;
                  }tVARIABLEBITS;


typedef union               /* global MI Bus Data Register access */
                  {
                  tU08     byte;
                  struct
```

```
                        {
                        tU08 data                                       :5; /* data pattern */
                        tU08 adr                                    :3; /* address data pattern */
                        }bit;
                }tMIBUS_DATA_REG_LOW;

#define data0           0x01;                           /*bit masks *//* Bit0 */
#define data1           0x02;                                                   /* Bit1 */
#define data2           0x04;                                                   /* Bit2 */
#define data3           0x08;                                                   /* Bit3 */
#define data4           0x10;                                                   /* Bit4 */
#define adr0            0x20;                                                   /* Bit5 */
#define adr1            0x40;                                                   /* Bit6 */
#define adr2            0x80;                                                   /* Bit7 */

typedef union           /* global MI Bus Control Register 2 access */
                {
                tU08    byte;
                struct
                        {
                        tU08 rei                        :1; /* receiver interrupt enable */
                        tU08 te                                 :1; /* transmitter enable */
                        tU08 re                                    :1; /* receiver enable */
                        tU08                                          :5; /* not used */
                        }bit;
                }tMIBUS_CONTROL_REG_2;


typedef union           /* global MI Bus Status Register 1 access */
                {
                tU08    byte;
                struct
                        {
                        tU08 rdrf                           :1; /* Receive data register full */
                        tU08 be                                    :1; /* Bit Error Flag */
                        tU08 nf                                 :1; /* Noise Error Flag */
                        tU08 complete                           :1; /* Noise Error Flag */
                        tU08                                          :4; /* Not Used */
                        }bit;
                }tMIBUS_STATUS_REG_1;

typedef union           /* global MI Bus Status Register 2 access */
                {
                tU08    byte;
                struct
                        {
                        tU08 raf                                    :1; /* Receiver Active Flag */
                        tU08                                          :7; /* Not Used */
                        }bit;
                }tMIBUS_STATUS_REG_2;
```

This Page Has Been Intentionally Left Blank

This Page Has Been Intentionally Left Blank

[redacted]

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com