

Demonstration Lab: Motor Control using SMAC (ZigBee[™] Technology)

By Edgard Palomera Mena
RTAC Americas
06/2005

Overview

This paper presents an example laboratory exercise based on the previous Motor Control Demonstration lab¹ using the same MCU MC9S12C32 but incorporating the ZigBee technology to drive a DC motor remotely. NI ELVIS is used as platform to connect components, provide power, and analyze circuit operation. The CSM12C32 and the ZigBee Module (RF MC13192U board) are plugged into the NI ELVIS connectors; basic circuitry is used to control the motor. MCU program development and debug are done with Metrowerks CodeWarrior[™] software running on the same host PC as the NI ELVIS and LabVIEW[™] software.

Introduction

The main purpose of this demonstration lab exercise is to introduce the ZigBee Technology and to show how easy can be use this technology. But what is ZigBee? Is a Standardized protocol for Ultra Low Power Wireless Personal Area Networks (WPANs). The ZigBee architecture is based on the IEEE 802.15.4 Specification and Incorporates all layers of software including the Application Layer and below (Network, MAC, PHY); the SMAC (Simple Media Access Controller) is an ANSI C based code stack that provides a layer of simple primitives that controls basic transceiver activities.

The ZigBee technology can be used in different markets such as consumer electronics, health care, industrial control, building automation, and residential/light commercial control.

This motor control Lab exercise was developed to explore how the NI ELVIS system² could complement existing microcontroller development tools in the engineering laboratory environment. The Freescale MC9S12C32 MCU³ and MC13192⁴ are used. The MC13192 is a short range, low power, 2.4 GHz ISM band transceiver which contains a complete 802.15.4 physical layer (PHY) modem designed for the IEEE 802.15.4 wireless standard. Metrowerks CodeWarrior⁵ is used as the MCU development tool. The breadboard provides a convenient platform for wiring peripheral circuitry and multiple power supplies for the lab exercise. As an added convenience, NI ELVIS virtual instruments such as the oscilloscope provide a way to look at various signals such as the PWM signal that drives the motor speed.

This paper discusses the implementation of this demonstration lab exercise. The implementation and theory of operation is given for each part of the system. Several possible variations are discussed that could be used to adapt this lab to slightly different audiences. For example, some classes might concentrate on developing a more sophisticated LabVIEW front panel that could monitor and record the duty cycle of the PWM drive signal and the resulting motor speed. Another class might develop a more sophisticated application using the ZigBee technology. Yet another class might extend this lab to implement a closed lab to implement a closed loop motor control speed controller using either PID or fuzzy logic techniques.

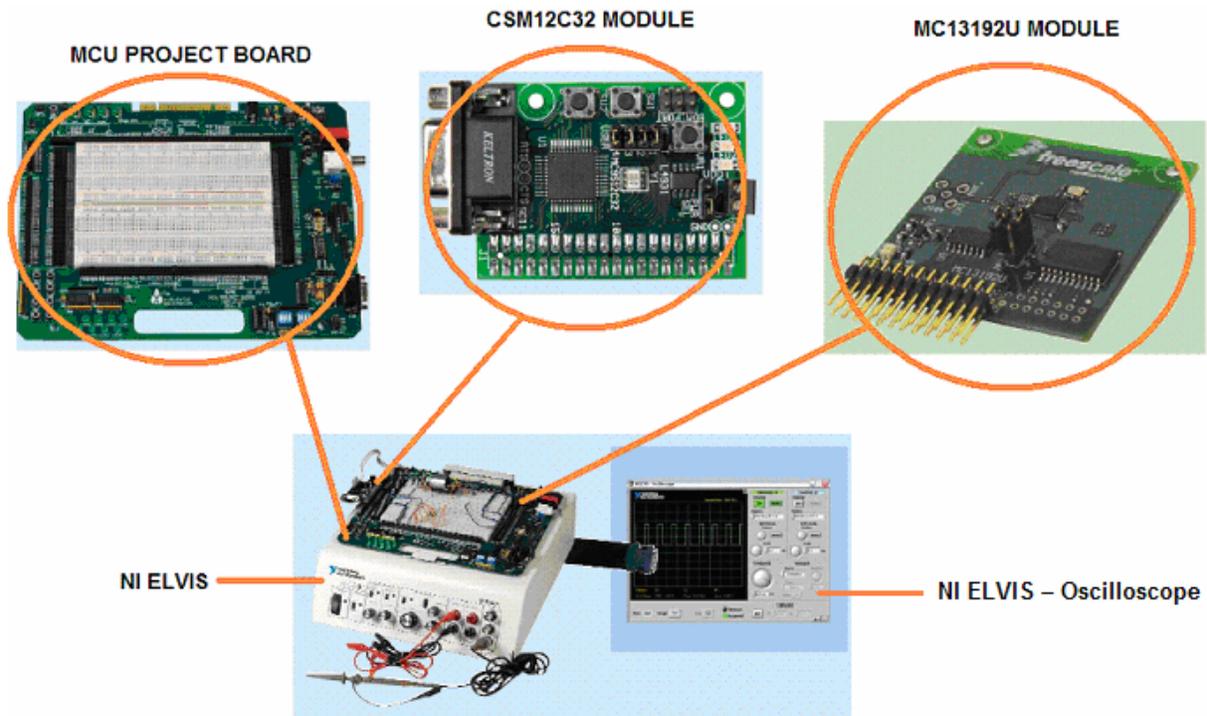


Figure 1. System modules.

System Block Diagram

The system consists of 2 NI ELVIS workstations. Each NI-ELVIS workstation has 1 MCU project board, 1 CSM12C32 module and 1 MC13192U module as shown in Figure 1. One of these workstations uses the MC13192U module as receiver and the other one uses the MC13192U module as transmitter. The receiver workstation sends the data to the host PC through the data acquisition card connected to the NI ELVIS system. The NI ELVIS block includes power supplies, digital I/O, as well as a large breadboard area for experiment circuitry. The MCU block includes an MC9S12C32 16-bit MCU along with a crystal, RS-232 level shifters, and a BDM connector for programming and debugging the application software. CodeWarrior development software supports debugging through a BDM pod connected to a printer port or USB port on the PC. The ZigBee Module includes the low power transceiver MC13192FC and the antenna printed out on the PCB. The dc motor interface block includes the motor, a 2-transistor circuit to translate the 0-5V PWM signal from the MCU into a 0-15V signal to drive the motor. Figure 2 shows the overall block diagram⁶ of the system.

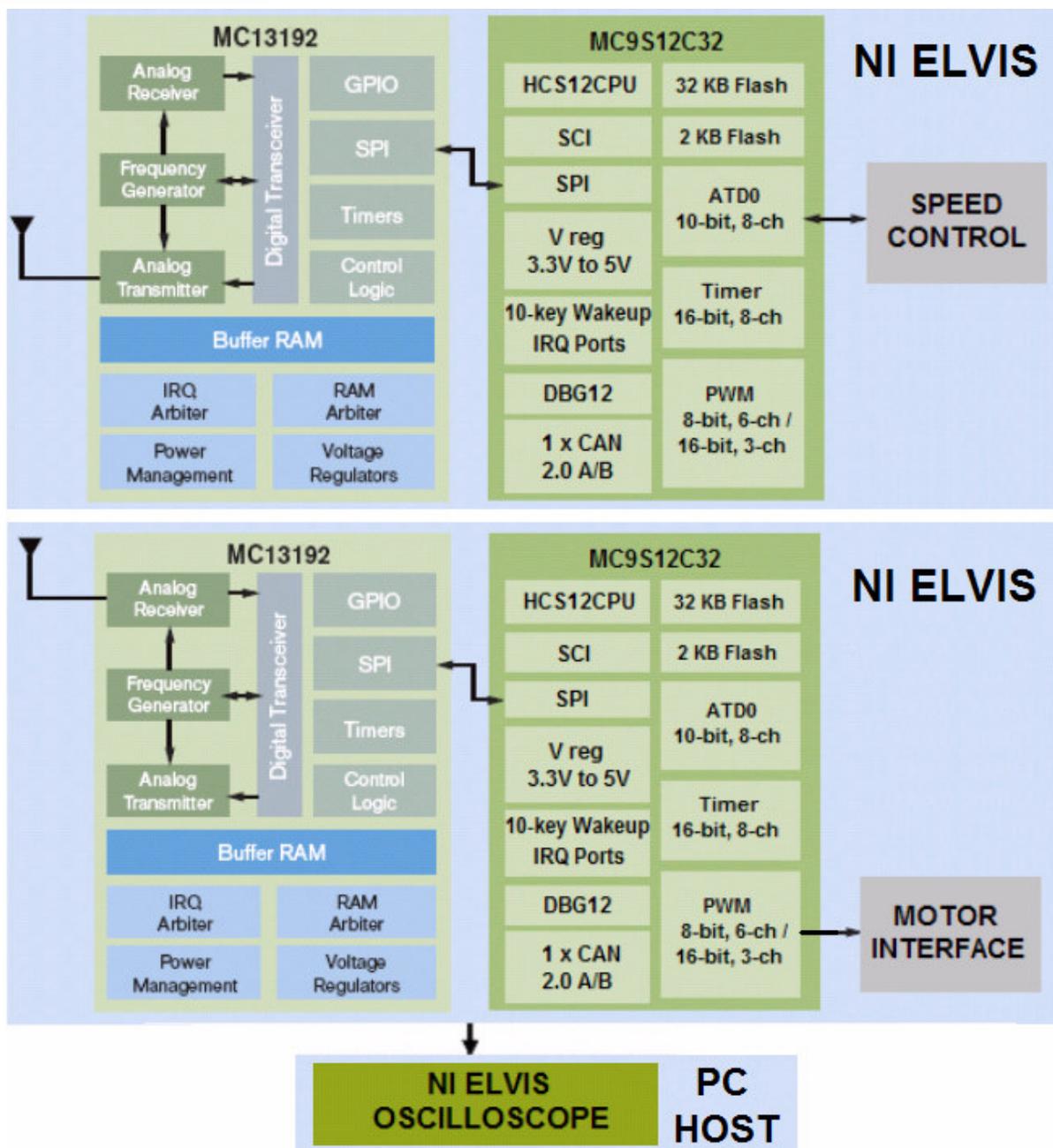


Figure 2. Block Diagram of the Motor Control – ZigBee Demonstration Lab.

NI ELVIS – Oscilloscope

The virtual Oscilloscope is used to display the PWM signal from the MCU. The oscilloscope VI is a virtual instrument that appears as an interactive window on the PC screen with buttons and knobs similar to those found on traditional oscilloscopes. The probes for this oscilloscope can be connected to the BNC connectors on the front of the NI ELVIS workstation, or they can be wires from the CHA+/- and CH_B+/- connectors at the upper left corner of the breadboard. The student can make the same kind of measurements with this virtual oscilloscope as they could with a physical oscilloscope, without the need for a separate piece of test equipment.

MC9S12C32 MCU Block

The MCU block is a pre-assembled module made by Axiom Manufacturing that includes the MC9S12C32 MCU and a small amount of common support circuitry. Figure 4 shows the pin assignments for the Transmitter MCU block. The transmitter MCU reads the potentiometer's voltage using an Analog to Digital Converter (ADC channel 0) to control the motor speed remotely.

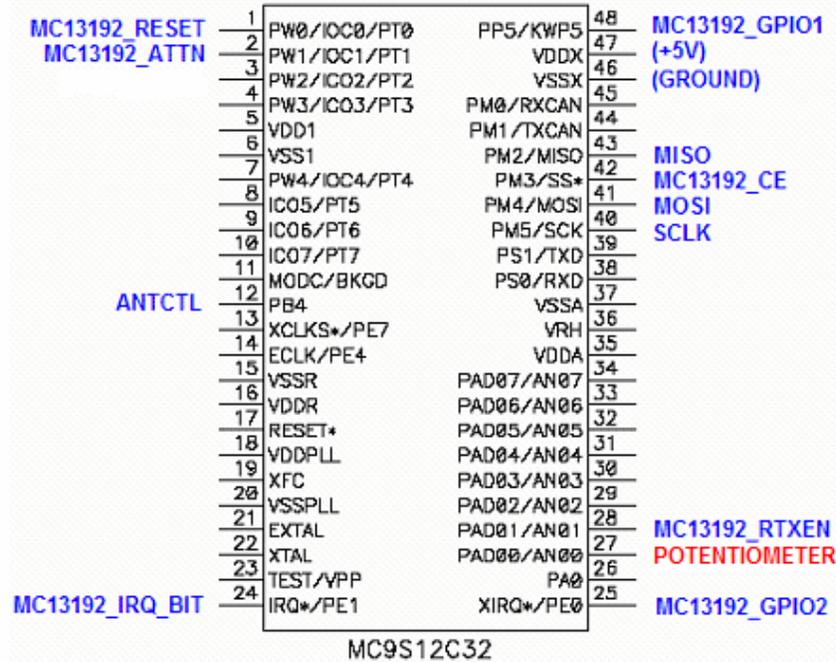


Figure 4. Pin Assignments for the Transmitter MCU Block.

The receiver MCU provides the PWM signal to drive the motor according to the potentiometer reading on the Transmitter MCU board. Figure 5 shows the pin assignments for the Receiver MCU block.

MC13192_RESET	1	PW0/IOC0/PT0	PP5/KWP5	48	MC13192_GPIO1
MC13192_ATTEN	2	PW1/IOC1/PT1	VDDX	47	(+5V)
PWM SIGNAL	3	PW2/IOC2/PT2	VSSX	46	(GROUND)
	4	PW3/IOC3/PT3	PM0/RXCAN	45	
	5	VDD1	PM1/TXCAN	44	
	6	VSS1	PM2/MISO	43	MISO
	7	PW4/IOC4/PT4	PM3/SS*	42	MC13192_CE
	8	IOC5/PT5	PM4/MOSI	41	MOSI
	9	IOC6/PT6	PM5/SCK	40	SCLK
	10	IOC7/PT7	PS1/TXD	39	
	11	MODC/BKGD	PS0/RXD	38	
ANTCTL	12	PB4	VSSA	37	
	13	XCLKS*/PE7	VRH	36	
	14	ECLK/PE4	VDDA	35	
	15	VSSR	PAD07/AN07	34	
	16	VDDR	PAD06/AN06	33	
	17	RESET*	PAD05/AN05	32	
	18	VDDPLL	PAD04/AN04	31	
	19	XFC	PAD03/AN03	30	
	20	VSSPLL	PAD02/AN02	29	
	21	XTAL	PAD01/AN01	28	MC13192_RTXEN
	22	TEST/VPP	PAD00/AN00	27	
MC13192_IRQ_BIT	23	IRQ*/PE1	PA0	26	
	24		XIRQ*/PE0	25	MC13192_GPIO2

MC9S12C32

Figure 5. Pin Assignments for the Receiver MCU Block.

The next circuit provides the required current to drive the motor, in other words the main purpose of the circuit is to be a source of current for the dc motor. The next circuitry is going to convert the 0-5V PWM signal coming out from the MCU into a 0-15V PWM signal, which is the signal required to drive the motor. The function of R3 is to be a pull up resistor to ensure that Q2 is going to be off when the switch Q1 is open. And the function of R2 is just to limit the current that comes out from the Q2 base when the switch Q1 is on. As we are switching on and off an inductor (motor) we need to add some diodes called freewheeling diodes in order to avoid the spikes generated by the motor. Figure 6 shows the circuitry used to drive the motor.

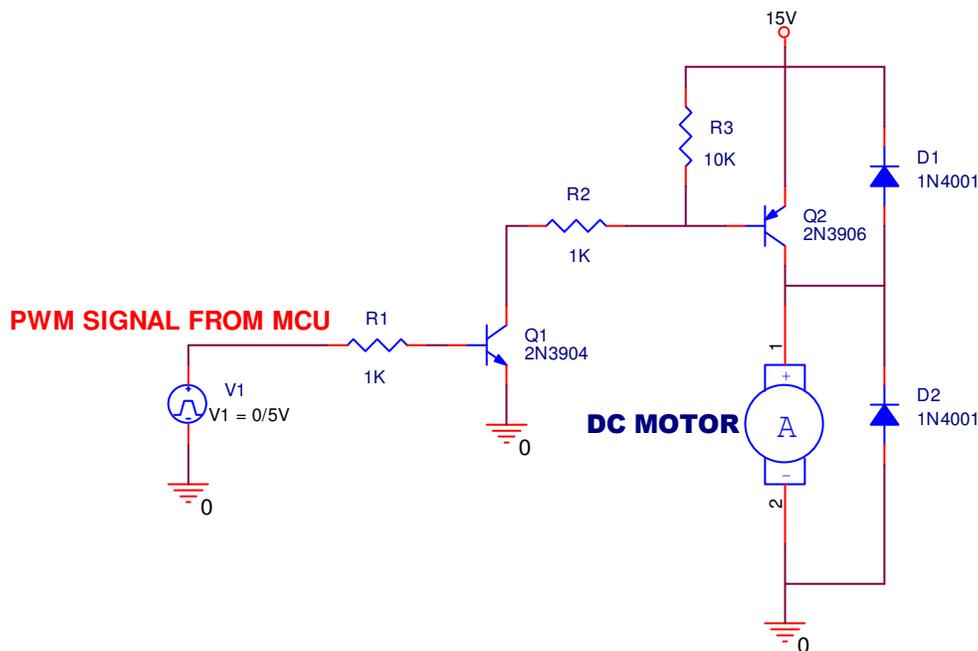


Figure 6. DC Motor Interface.

I/O Characteristics

For this lab exercise, we use an analog (ATD) input to measure the Potentiometer voltage which is used to set the motor speed. The ATD input to the MCU is between 0-5 volts. The potentiometer is located at the lower left corner of the MCU Project Board.

The PWM signal is a digital output pin with the same characteristics as a general purpose digital output. The PWM frequency for this motor control demonstration lab is 3.9 KHz so that the oscilloscope VI could be used to examine the waveform even with a relatively slow DAQ card. In a commercial application, you would probably want to use a PWM frequency of 20 KHz or higher so that you could not hear this frequency at a low motor speeds.

The serial port of the PC uses standard RS232 voltage levels (from -X volts to +X volts where X can be from 6 to 15 volts depending upon the particular PC). A level shifter device is included on the MCU block to convert the RS232 levels to the 0-5volt levels that are required at the RxD and TxD pins of the MCU. We have wired the level shifted RxD and TxD pins to the DSUB connector. This would be used if you were using the serial monitor rather than a BDM pod to interface CodeWarrior to the MCU. The serial monitor eliminates the need for a BDM pod, but is not as elegant (unobtrusive) as the BDM pod.

Background Debug Connections

The ideal way to allow CodeWarrior to access the MCU is through the background debug interface. The MCU module includes a 2x3 square-post header for this purpose. This interface uses a single dedicated pin on the MCU plus ground and optional connections to reset and Vdd. CodeWarrior communicates with a BDM pod through a parallel or USB port on the PC. The BDM pod converts commands from CodeWarrior into a custom serial BDM protocol. Using this interface, CodeWarrior can execute primitive commands to read or write memory locations (even while application programs are running), read or write CPU registers, set breakpoints, or trace single instructions. These primitive commands allow Code Warrior to program the Flash memory and debug user programs.

Serial Monitor Option

The serial monitor⁶ is a small 2 Kbytes program that is programmed into the Flash memory of the MC9S12C32. This program emulates primitive debugging commands similar to those available through the background debug interface. CodeWarrior has the ability to choose either a BDM pod or this serial monitor as the interface to the target MCU.

With the serial monitor, a simple serial cable is used to connect the serial I/O port of the PC to the (level shifted) SCI pins of the target MCU. This eliminates the cost of the BDM pod.

Motor Circuits

The circuit provided in Figure 6 is used to translate the 0-5 volt PWM signal from the MCU to the 0-15 volt levels required for the motor.

This motor is a basic dc brush motor. This motor could be driven by an analog voltage between 0 and 15 volts, or it can be driven by a PWM signal from the MCU. By adjusting the duty cycle of the PWM signal, you can control motor's speed. The 1N4001 across the motor reduces the amount of noise generated by the motor when it is driven by a PWM signal. If you remove this diode while the motor is running, you should notice the speed drop slightly and the sound will change to be slightly raspier.

MICROCONTROLLER SOFTWARE

The software for this paper can be downloaded from Freescale Semiconductor web site. For this lab, the software is built using CodeWarrior software stationery. The software stationery provides all the software necessary to access internal microcontroller register space and program memory through software labels. The stationery associates all register and control bit names from the MCU data sheet with the appropriate address and bit position. The stationery provides the foundation for students to write their own embedded programming routines.

To open the demo software, first extract the file labeled NI_ELVIS_ZIGBEE_DEMO.zip to your chosen directory. The extraction will form a directory labeled NI_ELVIS_ZIGBEE_DEMO (ex. C:\NI_ELVIS_MCU_DEMO\), which includes a directory labeled NI_ELVIS_ZIGBEE_DEMO_C32. The NI_ELVIS_ZIGBEE_DEMO_C32 project directory is built for the MC9S12C32 MCU Block. To proceed, make sure to have Metrowerks CodeWarrior for HC(S)12 installed on your computer. Within the NI_ELVIS_ZIGBEE_DEMO_C32 you will find a file labeled NI_ELVIS_ZIGBEE_DEMO_C32.mcp, which is the Metrowerks CodeWarrior project file for this lab's demo software. Double-click on the file and the demo software will be loaded in the CodeWarrior development environment. Figure 6 shows the lab's demo software open in the CodeWarrior development environment.

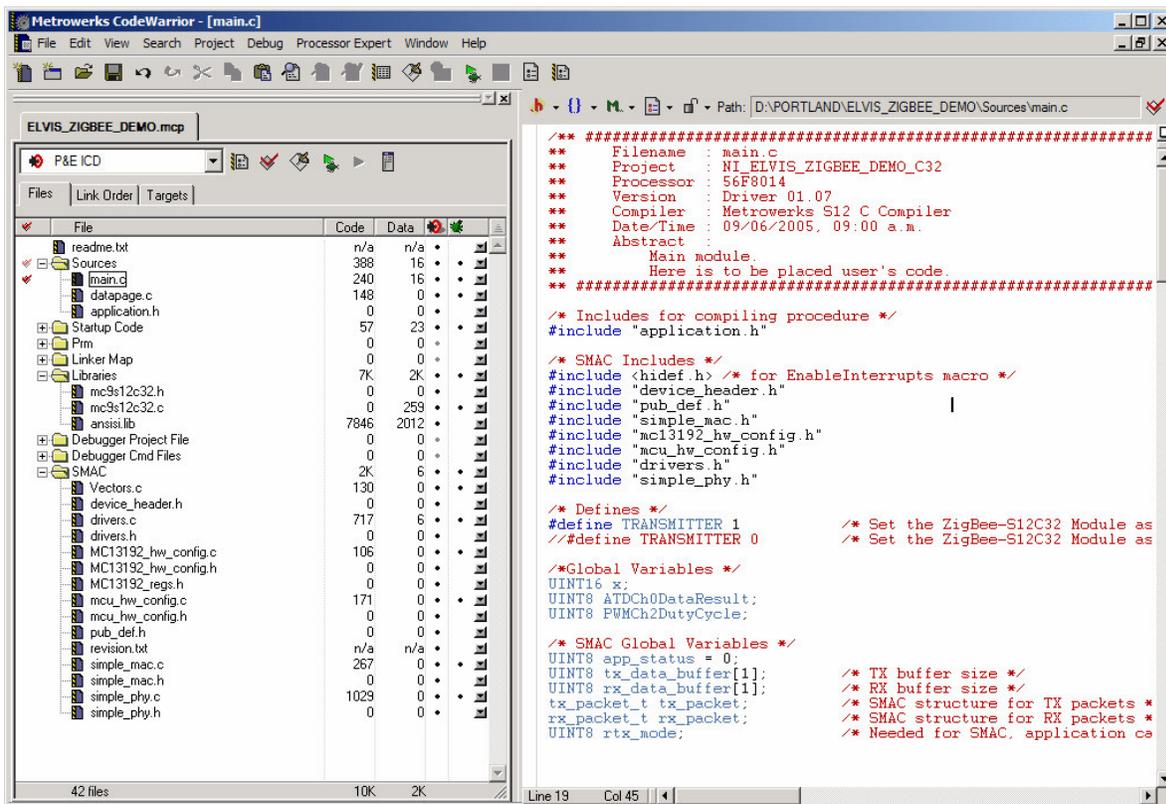


Figure 7. Motor Control-ZigBee Project in CodeWarrior.

The software is divided mainly in to parts. The SMAC (Simple Media Access Controller) section and the motor control application part. The SMAC⁷ is an ANSI C based code stack available as sample source code, which can be used to develop proprietary RF transceiver applications using the MC1319x integrated circuit. For a better understanding of the SMAC code you can download the SMACRM User's guide from the freescale website.

Basically the SMAC section contains the Initialization and Hardware configuration needed to interface the MCS12C32 with the MC13192 transceiver. The communication between the MCU and the transceiver is implemented via the Serial Peripheral Interface (SPI). The hardware interface uses five signals (Reset, RTXEN, ATTN, and IRQ) besides the four required for the SPI module.

The user needs to choose if the software is for the transmitter board or for the receiver board as shown in figure 8.

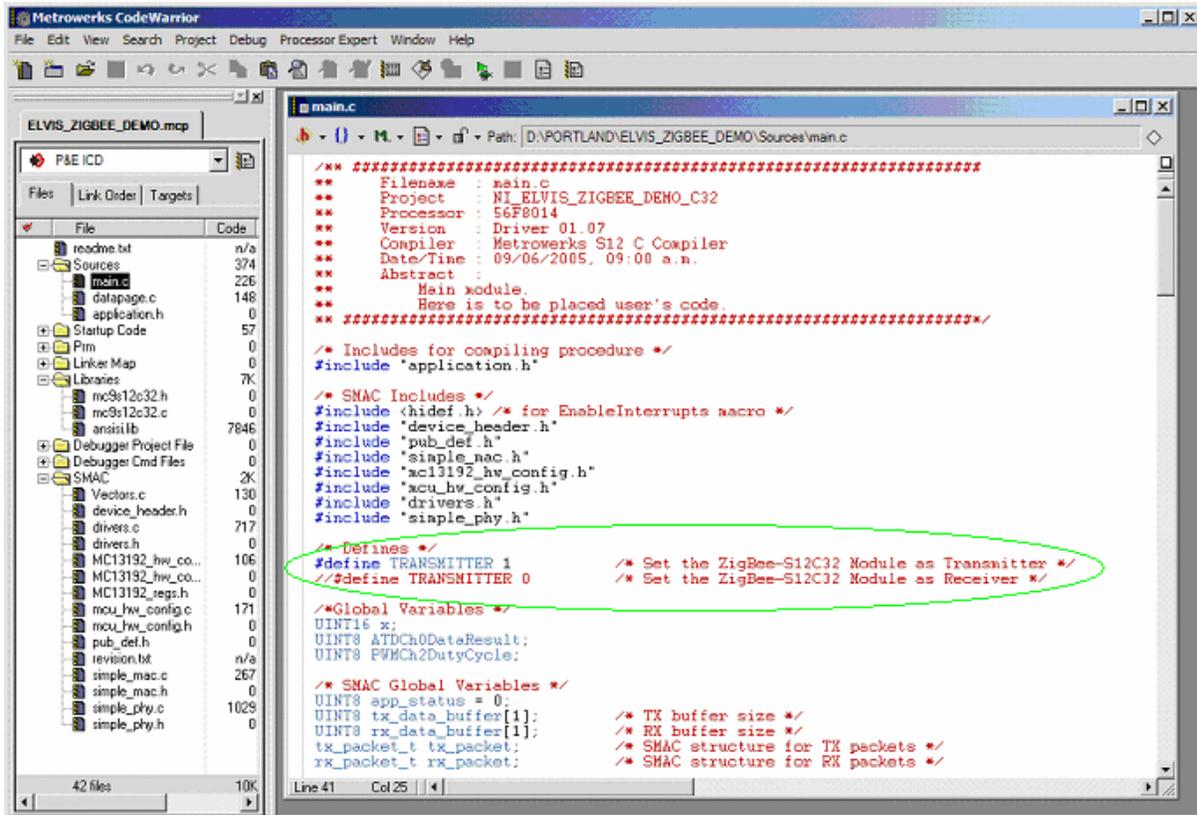


Figure 8. Transmitter and receiver mode selection.

Figure 9 shows the initialization part. In this section 3 functions are called in order to make the necessary initialization. The function `AppInit()`; initializes the packet structure as much as for the transmitter packet as for the receiver packet. The function `mcu_init()`; makes the proper initialization for the Microcontroller, initializes the SPI, GPIO's etc. The function `MC13192_init()` Initializes the MC13192 register map.

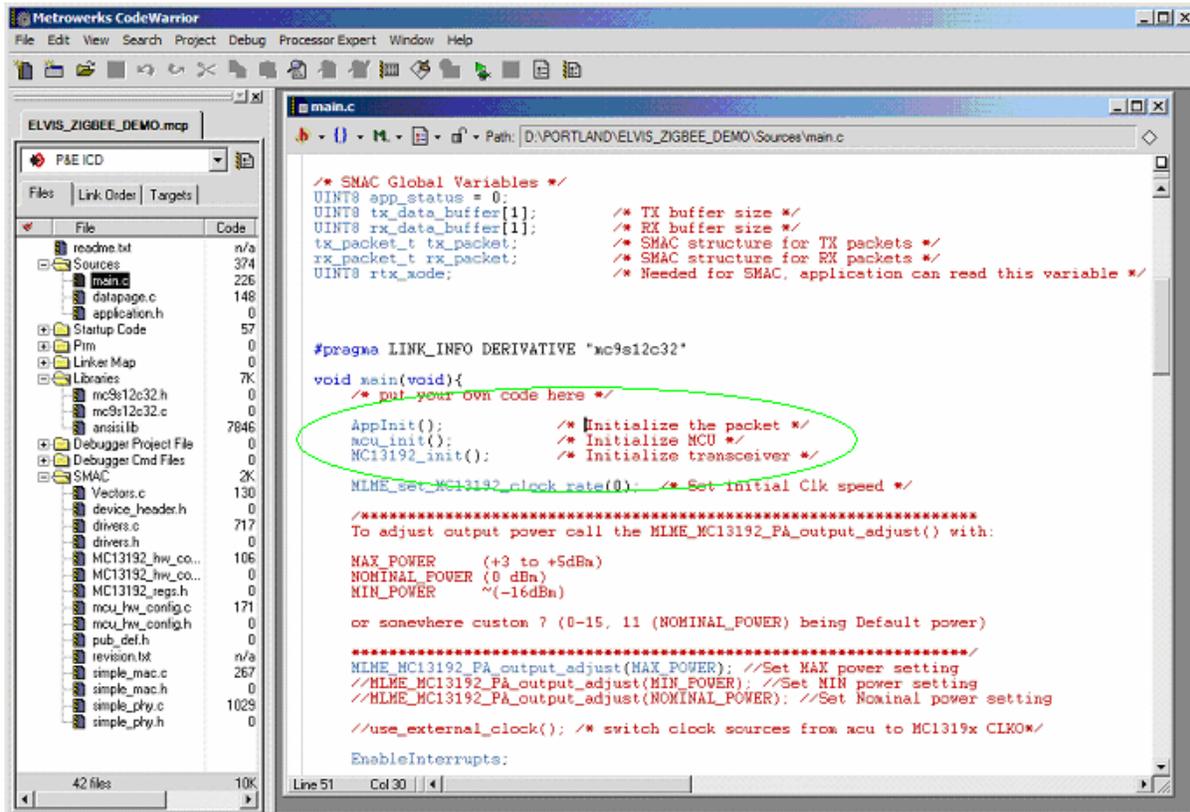


Figure 9. Initialization Section.

The next code section determines if the ADC or the PWM modules is going to be initialized. If the transmitter mode has been selected by the user, the Analog to Digital Converter module Channel 0 is going to be initialized, and after that is going to call the TXMotor(); which is the transmitter main loop. And if the receiver mode has been selected by the user the Pulse Width Modulation module Channel 2 is going to be initialized, and after that is going to call the RXMotor(); which is the receiver main loop. See Figure 10.

```

//use_external_clock(); /* switch clock sources from acu to MC1319x CLK0*/
EnableInterrupts;
app_status = INITIAL_STATE; /* Initial Mode */
if (MIME_set_channel_request(5) == SUCCESS) { //Select channel here (0-15)
    app_status = RECEIVER ALWAYS ON;
}

if (TRANSMITTER == 1){ /* If the Transmitter mode is selected */
    ATDCh0Init(); /* Initialize ADC Channel 0 */
    TXMotor(); /* Transmit Mode Loop */
}
else{ /* If the Receiver mode is selected */
    PWMCh2Init(); /* Initialize PWM Channel 2 */
    RXMotor(); /* Receive Mode Loop */
}
}

void AppInit(void){
    tx_packet.dataLength = 0; /* Initialize the tx_packet global */
    tx_packet.data = &tx_data_buffer[0]; /* Set the pointer to point to the tx_buffer */
    rx_packet.dataLength = 0; /* Initialize the rx_packet global */
    rx_packet.data = &rx_data_buffer[0]; /* Initialize the rx_packet global */
    rx_packet.maxDataLength = 2; /* Arbitrary size */
    rx_packet.status = 0; /* Initialize status to 0 */
}

/*
 * RxAccLoop : This is the main thread of execution when the module is in
  
```

Figure 10. ADC or PWM modules initialization.

The following code section is going to be executed only if the user selects previously the receiver mode. This section switches the application status to a receiver always on state. The PWM duty cycle register is updated each cycle. See Figure 11.

```

}
/*
 * RxAccLoop : This is the main thread of execution when the module is in
 * Receiver Mode.
 * Parameters : None
 * Return : None
 */
void RXMotor(void)
{
    for (;;)
    {
        PWMDTY2 = PWCh2DutyCycle; /* Update PWM Duty Cycle 0 - 100% (0x00 - 0xFF) */

        switch (app_status) {
            case IDLE_STATE:
                /* Switch to RECEIVER_ALWAYS_ON */
                app_status = RECEIVER_ALWAYS_ON;
                break;

            case RECEIVER_ALWAYS_ON:
                if (rtx_mode == IDLE_MODE) {
                    MLME_RX_enable_request(&rx_packet, 0);
                }
                LOW_POWER_WHILE; /* Wait mode */
                break;

            default:
                /* Should not get here */
                app_status = RECEIVER_ALWAYS_ON;
        }
    }
}
  
```

Figure 11. Receiver main loop.

The following section of the code is going to be executed only if the user selects previously the transmitter mode. In this section of the code, the data (ADC reading) loaded into the buffer is sent to the receiver board. See Figure 12.

```

)
}
}

/*
 * TxAccLoop : This is the main thread of execution when the module is in
 *             Transmitter Mode
 * Parameters : None
 * Return : None
 */
void TXMotor(void){
    /* TXMotor main loop */
    for (;;)
    {
        tx_data_buffer[0] = ATDCh0DataResult; /* Load buffer to send the ADC conversion r
        tx_packet.dataLength = 1;           /* Specifies the data lenght */
        MCPS_data_request(&tx_packet);     /* Transmit data */
        PORTA_BIT0 = ~PORTA_BIT0;         /* Toggle the LED1 */
    } /* End TXMotor main loop */
}

/*
 * MCPSPDataIndication : Receive data packet indication
 * MAC Layer data indication handler. RX packet is in the global structure
 * Parameters : sRxPacket - Data packet pointer
 */

```

Figure 12. Transmitter main loop.

The next function is called every time a data has been received, in this section of the code the data received is read and loaded into the PWMCh2DutyCycle variable; and after that the duty cycle register is updated with this value. See Figure 13.

```

Metrowerks CodeWarrior
File Edit View Search Project Debug Processor Expert Window Help

ELVIS_ZIGBEE_DEMO.mcp
P&E ICD
Files Link Order Targets
File Code
Sources 374
main.c 226
datapage.c 148
application.h 0
Startup Code 57
Pim 0
Linker Map 0
Libraries 7K
mc9s12c32.h 0
mc9s12c32.c 0
ansilib 7846
Debugger Project File 0
Debugger Cmd Files 0
SMAC 2K
Vectors.c 130
device_header.h 0
drivers.c 717
drivers.h 0
MC13192_hw_co... 106
MC13192_hw_co... 0
MC13192_regs.h 0
mcu_hw_config.c 171
mcu_hw_config.h 0
pub_def.h 0
revision.txt n/a
simple_mac.c 267
simple_mac.h 0
simple_phy.c 1029
simple_phy.h 0
42 files 10K

main.c
Path: D:\PORTLAND\ELVIS_ZIGBEE_DEMO\Sources\main.c

/*
 * MCPSDataIndication : Receive data packet indication
 * MAC Layer data indication handler. RX packet is in the global structure
 * Parameters : sRxPacket - Data packet pointer
 * Return : None
 * Place your code here to handle a mac layer data indication.
 * RX packet is in the global structure
 * rx_packet.dataLength and rx_packet.data
 */
void MCPS_data_indication(rx_packet_t *rx_packet) {
    if (rx_packet->status == SUCCESS){
        PWMCh2DutyCycle = rx_data_buffer[0]; /* Read the data received in the buffer
        PORTB_BIT4 = ~PORTB_BIT4; /* Toggle the LED1 */
    }
}

/*
 * MLMEMC13192ResetIndication : Notifies you that the MC13192 has been reset.
 * Parameters :
 * Return : None
 */
void MLMEMC13192_reset_indication(){
    /* Application must handle this here. */
}

Line 99 Col 25
  
```

Figure 13. Receive data indication.

The next section shows the ADC interrupt. Each time an ADC conversion is completed the following code is executed. The variable ATDCh0DataResult is loaded with the value in the register ATDDR0L, the value contained in the register ATDDR0L is the conversion's result of the ADC reading. See Figure 14.

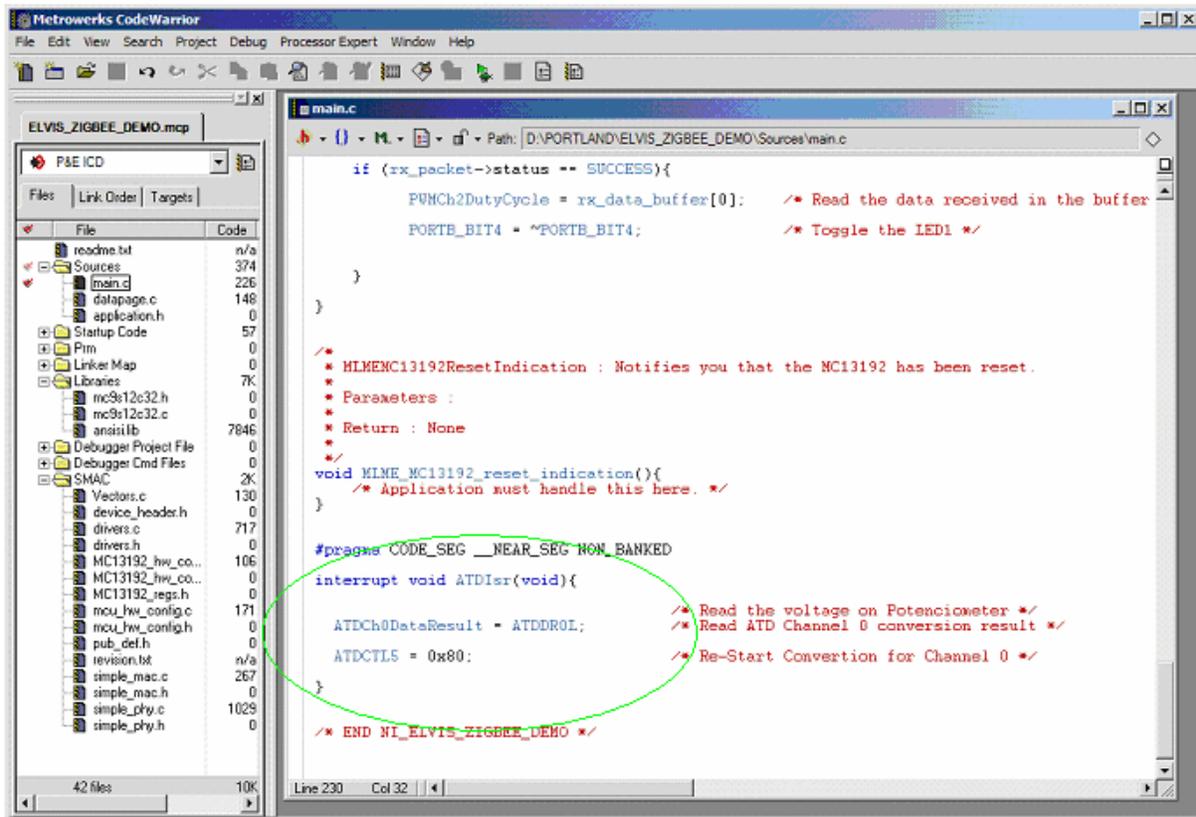


Figure 14. ADC interrupt

REFERENCES

1. Eduardo Montanez and Jim Sibigroth, Motor Control Demonstration Lab, Freescale Semiconductor Inc., Austin, TX(2003).
2. National Instruments, NI Educational Laboratory Virtual Instrumentation Suite (NI ELVIS) User Manual, National Instruments, Austin, TX (2003).
3. Freescale, MC9S12C32 Users Guides, Freescale Semiconductor Inc., Austin, TX (2003).
4. Freescale, MC13192DS, Freescale Semiconductor Inc., (2004).
5. Metrowerks, CodeWarrior Development Studio for HC(S)12 Microcontrollers (Special Edition), Metrowerks, Austin, TX (2003).
6. Freescale, MC13192FS, Freescale Semiconductor Inc., (2004).
7. Freescale, SMAC User's Guide (SMACRM), Freescale Semiconductor Inc., (2004).