An example application using the MAXQ2000 Evaluation Kit

The availability of standard ANSI C tools and development environments that integrate these tools significantly eases application development for new or unfamiliar processors. The tools available for the MAXQ line of processors include IAR's ANSI C compiler and the IAR Embedded Workbench integrated-development environment. With these programs and a basic knowledge of the MAXQ special-purpose registers, a developer can quickly and easily begin writing applications for the MAXQ architecture. The easiest way to demonstrate how simple the development process can be on the MAXQ architecture is with an example application.

The availability of standard ANSI C tools and development environments that integrate these tools significantly eases application development for new or unfamiliar processors.

The application described here uses the MAXQ2000 processor and the MAXQ2000 Evaluation Kit. The MAXQ2000 has a wide range of integrated peripherals, including:

- 132-segment LCD controller
- Integrated SPI port with master and slave modes
- 1-Wire Bus Master
- Two serial UARTs
- Hardware multiplier
- Three 16-bit timers/counters
- Watchdog timer
- 32-bit real-time clock with subsecond and time-of-day alarms
- JTAG interface with support for in-circuit debugging

Application overview

This example showcases the uses of the LCD controller, the master mode of the SPI port, one of the UARTs, the hardware multiplier, and one of the timers. The timer is used to generate periodic interrupts. When an interrupt occurs, the MAXQ2000 takes a temperature reading and outputs the results to an LCD and one of its serial ports. The SPI port interfaces with the MAX1407 data-acquisition system (DAS), which contains an ADC. Temperature readings are then taken by connecting a thermistor to the MAX1407's ADC.

Using the LCD controller

To use the LCD, two control registers must be configured. Once these registers have been set, segments on the LCD can be turned on by setting a bit in one of the LCD data registers. The following code shows how the LCD controller is configured for the example application.

```
void initLCD()
{
  LCRA_bit.FRM = 7;    // Set up frame frequency.
  LCRA_bit.LCCS = 1;   // Set clock source to HFClk / 128.
  LCRA_bit.DUTY = 0;   // Set up static duty cycle.
  LCRA_bit.LRA = 0;    // Set R-adj to 0.
  LCRA_bit.LRIGC = 1; // Select external LCD drive power.

  LCFG_bit.PCF = 0x0F;// Set up all segments as outputs.
  LCFG_bit.OPM = 1;    // Set to normal operation mode.
  LCFG_bit.DPE = 1;    // Enable display.
}
```

Communicating over SPI

Writing to the SPIB register initiates the two-way communication between the SPI master and slave.

Three registers control the various SPI modes supported by the MAXQ2000. To communicate with the MAX1407, the following code is used to initialize the SPI component and place it in the correct mode.

```
PD5 |= 0x070;             //  Set CS, SCLK, and DOUT pins as output.
PD5 &= ~0x080;            //  Set DIN pin as input.
SPICK = 0x10;             //  Configure SPI for rising edge, sample input
SPICF = 0x00;             //  on inactive edge, 8 bit, divide by 16.
SPICN_bit.MSTM = 1;       //  Set Q2000 as the master.
SPICN_bit.SPIEN = 1;      //  Enable SPI.
```

Once the SPI configuration registers have been set, the SPIB register is used to send and receive data. Writing to this register initiates the two-way communication between the SPI master and slave. The STBY bit in the SPICN register signals when the transfer is complete. The SPI send-and-receive code is shown below.

```
unsigned int sendSPI(unsigned int spib)
{
  SPIB = spib;            //  Load the data to send
  while(SPICN_bit.STBY);  //  Loop until the data has been sent.
  SPICN_bit.SPIC = 0;     //  Clear the SPI transfer complete flag.
  return SPIB;
}
```

Writing to a serial port

In the example application, one of the MAXQ2000's serial ports is used to output the current temperature reading. Before any data can be written to the port, the application must set the baud rate and the serial port mode. Again, just a few registers need to be initialized to enable serial port communications.

```
void initSerial()
{
  SCON0_bit.SM1 = 1;      //  Set to Mode 1.
  SCON0_bit.REN = 1;      //  Enable receives.
  SMD0_bit.SMOD = 1;      //  Set baud rate to 16 times the baud clock.
  PR0 = 0x3AFB;           //  Set phase for 115200 with a 16MHz crystal.
  SCON0_bit.TI = 0;       //  Clear the transmit flag.
  SBUF0 = 0x0D;           //  Send carriage return to start communication.
}
```

For the MAXQ architecture, interrupts must be enabled on three levels: globally, for each module, and locally.

As with the SPI communication routines, a single register sends and receives serial data. Writing to the SBUF0 register will initiate a transfer. When data becomes available on the serial port, reading the SBUF0 register will retrieve the input. The following function is used in the example program to output data to the serial port.

```
int putchar(int ch)
{
  while(SCON0_bit.TI == 0); // Wait until we can send.
  SCON0_bit.TI = 0;         // Clear the sent flag.
  SBUF0 = ch;               // Send the char.
  return ch;
}
```

Generating periodic interrupts with a timer

The last component used in this example application is one of the 16-bit timers. The timer generates interrupts that trigger temperature readings twice a second. To configure the timer for this example, the programmer must set the reload value, specify the clock source, and start the timer. The following code shows the steps required for initializing timer 0.

```
T2V0 = 0x00000;         //  Set current timer value.
T2R0 = 0x00BDC;         //  Set reload value.
T2CFG0_bit.T2DIV = 7;   //  Set div 128 mode.
T2CNA0_bit.TR2 = 1;     //  Start the timer.
```

Using this timer as an interrupt source like the example requires a few more steps. For the MAXQ architecture, interrupts must be enabled on three levels: globally, for each module, and locally. Using IAR's compiler, enable global interrupts by calling the __enable_interrupt() function. This effectively sets the Interrupt Global Enable (IGE) bit of the Interrupt and Control (IC) register. Since timer 0 is located in module 3, set bit 3 of the Interrupt Mask Register (IMR) to enable interrupts for the module. Enable the local interrupt by setting the Enable Timer Interrupts (ET2) bit in Timer/Counter 2 Control Register A (T2CNA). These steps, as executed in the example application, are shown below.

```
__enable_interrupt()
T2CNA0_bit.ET2 = 1;     //  Enable interrupts.
IMR |= 0x08;            //  Enable the interrupts for module 3.
```

Finally, using an interrupt requires initializing the interrupt vector. IAR's compiler allows a different interrupt handling function for each module. Setting the interrupt handler for a particular module requires using the #pragma vector directive. The interrupt-handling function declaration should also be preceded by the __interrupt keyword. The example application declares an interrupt handler for module three in the following way.

Programmers can easily develop applications for MAXQ processors after learning the details of a few peripheral registers.

```
#pragma vector = 3
__interrupt void timerInterrupt()
{
  // Add interrupt handler here.
}
```

Conclusion

As these code samples illustrate, learning the details of a few peripheral registers enables programmers to easily develop applications for the MAXQ2000 processor and the MAXQ line of processors. The addition of IAR's Embedded Workbench speeds up the development process by allowing code to be written in ANSI-compliant C code.

The complete source code for this sample application can be downloaded at www.maxim-ic.com/MAXQ_code. Read the description and comments found at the beginning of the code for details on the required wiring and setup. For more details on using IAR's Embedded Workbench, refer to the second article in this publication entitled, "Programming in the MAXQ Environment."