DALLAS SEMICONDUCTOR MAXIM

Microcontroller

ENGINEERING REVIEW

Volume 4

The MAXQ architecture realized: introducing the MAXQ2000

The MAXQ™ family of microcontrollers from MAXIM®/Dallas Semiconductor are high-performance, 16-bit RISC devices ideal for battery-powered mixed-signal applications. Designed for reduced noise operation, the MAXQ integrates high-precision analog functions alongside digital components, resulting in reduced chip-count solutions.

The MAXQ uses a Harvard memory map to keep data, code, and register space on separate buses.

The MAXQ uses a Harvard memory map to keep data, code, and register space on separate buses. The primary advantage of this type of memory map is flexible word lengths, allowing system and peripheral registers to be 8 or 16 bits wide. Typical MAXQ devices have 16-bit data, code, and register memory. Because the MAXQ instruction word is 16 bits, the microcontrollers always have a 16-bit instruction bus. Another advantage of the Harvard architecture is that the memory is always accessed using registers. This enables direct memory access for peripherals such as analog-to-digital converters (ADCs) and hardware coprocessors.

The MAXQ architecture is based on a very simple concept: all operations are ultimately achieved using a simple transfer operation. Each of the 33 total instructions reduces to either writing an immediate value to a destination register/ memory location, or moving data between registers and/or memory locations (**Figure 1**). The architectural simplicity makes it ideal for tool vendors to optimize code development and use the smallest possible memory requirements for most applications. In addition, each instruction in the MAXQ is processed in a single cycle, ensuring the fastest possible code execution (1 MIPS/MHz).

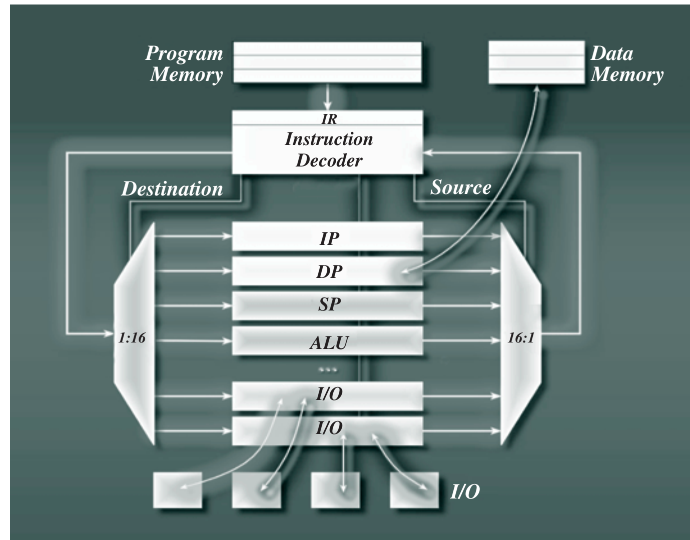

Figure 1. Each of the MAXQ2000's 33 instructions reduces to either writing an immediate value to a destination register/memory location, or moving data between registers and/or memory locations.

Introducing the MAXQ2000

The MAXQ2000 is the first of many products in the MAXQ family. This microcontroller integrates a 16-bit CPU with 64kB of flash memory, 2kB of SRAM, and a 4 x 36-segment LCD controller. The on-board LCD controller generates signals for an LCD based on display memory content. The application code sets up user-configurable options and writes to the display memory. The LCD controller then generates the necessary segment and common signals at the selected display frequency, so the microcontroller does not have to constantly manage the display and directly drive the LCD. In addition, the controller supports four types of display modes.

Table of Contents

The MAXQ architecture realized: introducing the MAXQ2000 1

Programming in the MAXQ environment 4

Accessing the functions provided in the MAXQ utility ROM 10

An example application using the MAXQ2000 Evaluation Kit 13

Signal processing with the MAXQ multiply-accumulate unit (MAC) 16

...the MAXQ core enables clocks only to those circuits that require clocking at any instant, thus reducing power consumption and providing a quiet environment optimal for analog integration.

1) Static

2) 1/2 duty multiplexed with 1/2 bias voltages

3) 1/3 duty multiplexed with 1/3 bias voltages

4) 1/4 duty multiplexed with 1/3 bias voltages

Seventeen bytes of display memory are available for use by the LCD controller, or for general-purpose application storage. Another enhanced LCD feature is found in its integrated voltage-divider resistors, which eliminate the need for external components, and can be used for contrast adjustment. **Figure 2** shows the LCD-drive voltage configuration for a static display. Each of the 36 segment pins is configurable for general-purpose I/O when not connected to an LCD.

The MAXQ2000 also offers five choices for the system clock.

MAXQ2000 Features

- 16-Bit RISC Core
- Single-Cycle Execution
- 64kB Flash Memory
- 2kB Data RAM
- 16-Level Hardware Stack
- Four Common, 36-Segment LCD Controller
- Real-Time Clock
- 16 x 16 Multiply with 40-Bit Accumulator

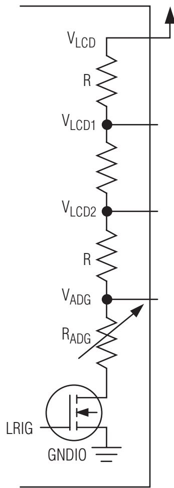

Figure 2. Shown here is the LCD-drive voltage configuration for static display.

1) Internal ring oscillator

2) Internal high-frequency oscillator using external crystal or resonator circuit

3) External high-frequency clock signal

4) Internal 32kHz oscillator using external crystal or resonator circuit

5) External 32kHz clock signal

Multiple power-management modes (PMMs) ensure minimum current consumption. Divide-by-256 mode allows all operations to continue as normal, but at a reduced clock rate from the high-frequency input. For further power reduction, PMM2 mode enables the microcontroller to run from the 32kHz clock. All operations continue as normal, but at an extremely reduced clock frequency. Because some inputs to the microcontroller must be executed at full speed, switchback mode is available. Switchback mode provides automatic exit from power management when higher speed operation is required, such as UART, SPI™, or external interrupts.

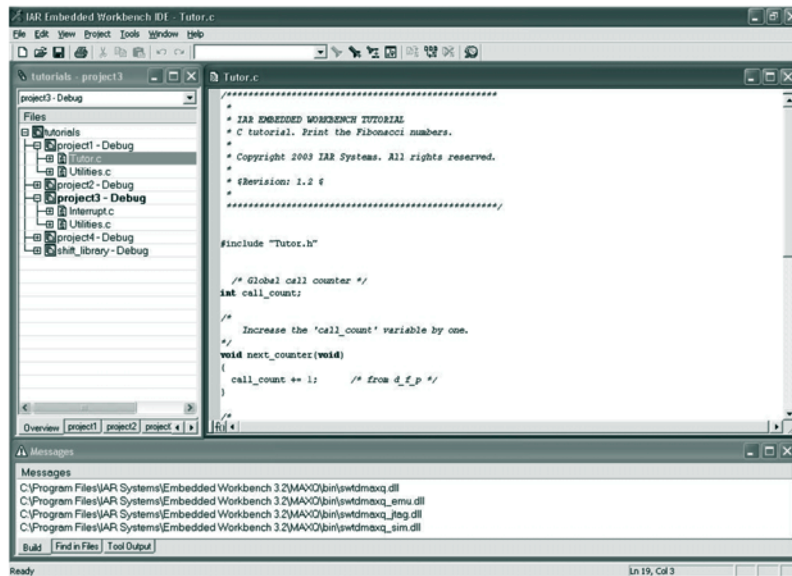

Figure 3. IAR Embedded Workbench provides an online tutorial for the MAXQ product line.

World-class tools

Embedded-application designs demand faster development time. To meet this demand, DALLAS® is providing world-class tools for integrated development environments, emulation tools, and complete in-circuit emulators. For assembly language development, a free MAXIDE can be downloaded from the MAXIM website at www.maxim-ic.com/microcontrollers. The MAXIDE includes an assembler, an IDE with codeword highlighting, support for the MAXQ JTAG interface, and a simulator for each available MAXQ microcontroller. C development is supported through our tools partners. IAR, with IAR Embedded Workbench®, is the first tools provider with a complete IDE for the MAXQ (**Figure 3**). Other partners are working on compilers as well as IDEs.

The MAXQ microcontrollers include support for JTAG emulation. Each microcontroller integrates an emulation engine, allowing firmware development to begin on flash-based versions of the microcontrollers in real system designs. Firmware engineers can completely design, write, and debug their application using the real chip and application circuit. Support includes breakpoints, which are configurable to be triggered on registers, code, or data. Background debug executes the microcontroller at full speed. When a debug event is triggered, such as a breakpoint, the microcontroller switches to foreground debug mode. In this mode, registers, memory, and single-step trace mode are possible.

The MAXQ2000 integrates a 16-bit CPU with 64kB of flash memory, 2kB of SRAM, and a 4 x 36-segment LCD controller.

Conclusion

The MAXQs' innovative single-clock-cycle execution, power-management modes, and wide range of mixed-signal peripherals make them ideal for today's power conscious, high-performance applications. Now with the introduction of the MAXQ2000, you can start designing and evaluating integrated analog-digital designs today.

MAXIM is a registered trademark of Maxim Integrated Products, Inc. All rights reserved.
DALLAS is a registered trademark of Dallas Semiconductor Corp.
MAXQ is a trademark of Maxim Integrated Products, Inc.
SPI is a trademark of Motorola, Inc.
IAR Embedded Workbench is a registered trademark of IAR Systems.

Programming in the MAXQ environment

The in-circuit debugging and program-loading features of the MAXQ2000 micro-controller combine with IAR's Embedded Work-bench development environment to provide C or assembly-level application develop-ment and testing.

The MAXQ architecture was developed for application programmers. Each MAXQ microcontroller includes a hardware debug engine that is tightly integrated with the microcontroller core. The first chip in this architecture is the MAXQ2000, and this article provides examples and tips on the use of the IAR Embedded Workbench with the MAXQ2000 Evaluation Kit.

The in-circuit debugging and program-loading features of the MAXQ2000 microcontroller combine with IAR's Embedded Workbench development environment to provide C or assembly-level application development and testing. The hardware-based debug engine and bootloader of the MAXQ2000 run over a dedicated JTAG port to allow full debugging access with minimal impact on system resources.

In-circuit debug features

A hardware debug engine, which is tightly integrated with the microcontroller core, controls the MAXQ2000's debugging features. This debug engine can invoke service routines in the on-board utility ROM to support a wide array of debugging features.

- Read access to the integrated flash program memory.
- Read/write access to the on-board data SRAM.
- Read access to the 16 x 16 stack memory.
- Read/write access to all MAXQ2000 system and peripheral registers.
- Step-by-step (trace) program execution.
- Up to four address-based breakpoints to stop program execution at a particular location in code memory.
- Two data memory-matching breakpoints to stop program execution when a particular location in data memory is accessed.
- Two register-based breakpoints to stop program execution when write access to a particular system or peripheral register occurs (cannot be used simultaneously with the data memory matching breakpoint) and the data being written to the register matches a specified value.
- Password matching function (to unlock the remaining debug functions).

All communication with the debug engine takes place over the MAXQ2000's dedicated JTAG TAP interface, which is compatible with the JTAG IEEE Standard 1149.

All communication with the debug engine takes place over the MAXQ2000's dedicated JTAG Test Access Port (TAP) interface, which is compatible with the JTAG IEEE Standard 1149. This interface consists of four signals, multiplexed with MAXQ2000 port pins as follows: TMS (Test Mode Select)—multiplexed with P4.2; TCK (Test Clock)—multiplexed with P4.0; TDI (Test Data In)—multiplexed with P4.1; and TDO (Test Data Out)—multiplexed with P4.3.

While the JTAG TAP port is dedicated to in-system debug and in-system programming uses, the four port pins that carry the JTAG TAP port signals may be released for other purposes once application development is complete. The JTAG port is active by default following reset, but once running, the application software can deactivate the JTAG port, leaving the four associated port pins free for other uses.

The JTAG interface and the debug engine operate asynchronously with respect to the MAXQ2000 core. Communication over the JTAG port need not take place at the same clock rate that the MAXQ2000 is running, although the frequency of TCK is limited to a maximum of 1/8 the system clock rate for the MAXQ2000.

Breakpoint settings can be read and written through the debug engine while the MAXQ2000 is executing code. This mode is known as background mode, where the debug engine operates independently of the CPU core.

A hardware debug engine, which is tightly integrated with the microcontroller core, controls the MAXQ2000's debugging features.

To perform other operations such as memory and register read and write, the debug engine takes control of the MAXQ2000 core, and switches execution to one of the debug service routines located in the utility ROM. This mode is known as debug mode, in which the debug engine interrupts normal program execution. The user application is suspended temporarily in these cases and resumes execution once the debug function has been completed, in the same way that interrupt routines are handled.

Because the JTAG TAP port is not used for application software purposes, the port pins comprising the JTAG port can be reclaimed by the application software. All additional code required for debugging functions is located in the utility ROM, so the only system resources consumed by the debugging functions are a small amount of data SRAM and one level of the program stack (used to store the return address when a debugging routine is called). The highest 19 bytes of data SRAM (addresses 0x07ED to 0x07FF) are reserved for use by the debugging service routines. If in-circuit debugging will not be used for a particular application, these data SRAM locations are available for application use.

Integrated flash-memory programs over JTAG

The JTAG TAP port is also used for an additional bootloader function, which is available even if the debugging functions will not be used. By setting three configuration bits over the JTAG TAP interface and then releasing the MAXQ2000 from reset, control can be transferred to the built-in bootloader routines located in the utility ROM. The configuration bits that control access to the bootloader are as follows.

- SPE: System Program Enable Bit (ICDF.1). When this bit is set to 1, the MAXQ2000 executes the bootloader routine in the utility ROM following system reset.
- PSS[1:0]: Programming Source Select (ICDF.3-2). The settings of these bits determine whether the JTAG port (PSS[1:0] == 00b) or the serial 0 UART (PSS[1:0] == 01b) is used for bootloader communication.

Once these bits are set and the MAXQ2000 is released from reset, the utility ROM bootloader begins communicating with the host system over the selected port (JTAG or serial 0 UART). In either case, the protocol used is the same and provides the following functions.

- Reads the ID banner of the MAXQ2000 (identifies utility ROM version).
- Returns the size of internal program and data memory.
- Reads, writes, verifies, and CRCs the integrated flash program memory.
- Reads, writes, verifies, and CRCs the internal data SRAM.
- Password matches (to unlock memory read and write commands).

By setting three configuration bits over the JTAG TAP interface and then releasing the MAXQ2000 from reset, control can be transferred to the built-in bootloader routines located in the utility ROM.

While the bootloader can communicate over the serial 0 UART instead of the JTAG port, the JTAG interface must be used to place the bootloader into serial communications mode. However, the application software can also invoke the bootloader in serial communications mode by setting the SPE and PSS bits appropriately, then resetting the MAXQ2000 (by letting the watchdog timer expire or by external hardware means). The method for causing the bootloader to be invoked (such as a signal on a port pin) must be determined by the application software.

Password protection for debug and bootloader functions

A basic password-protection scheme restricts access to the debugging and bootloader functions on the MAXQ2000. This password must be provided by the host system before access is allowed to any functions that read or modify the contents of memory or system and peripheral registers.

A basic password-protection scheme restricts access to the debugging and bootloader functions on the MAXQ2000.

The password is 16 words or 32 bytes long. The value for the password is located in the internal flash memory at word locations 0x0010 to 0x001F. These values can be included in an application as a static array, or they can simply be the values of the instruction codes stored in those locations. Either way, the password is automatically written when the application is loaded. If no application has been loaded, the password will be a default value with all words equal to 0xFFFF.

Even if the password is not known, the MAXQ2000's internal flash memory can always be erased through the bootloader. This effectively clears the password value (to all 0xFFFF words) and allows other programming and debugging operations to proceed. The password protection simply ensures that existing code may not be read from the MAXQ2000 without first matching the 32-byte password value.

Using the serial-to-JTAG adapter module

Integrated development environments for the MAXQ2000 microcontroller (such as MAXIDE and IAR Embedded Workbench) include software libraries to support communication with the MAXQ2000 JTAG interface. However, as the PCs running this software do not typically have JTAG ports included, a hardware layer is needed to interface the two systems.

The serial-to-JTAG adapter module, included with the MAXQ2000 Evaluation Kit, provides a turnkey solution to this interface problem (**Figure 1**). Software running on the PC (such as IAR Embedded Workbench) communicates with the serial-to-JTAG adapter module over a standard COM serial port. The serial-to-JTAG adapter module then interfaces to the JTAG port of the MAXQ2000, passing commands to the bootloader or the debugging engine. The adapter module also handles level translation and supports MAXQ microcontrollers running over a range of different supply voltages, as well as removes the need for the PC to provide precise timing for the JTAG waveforms.

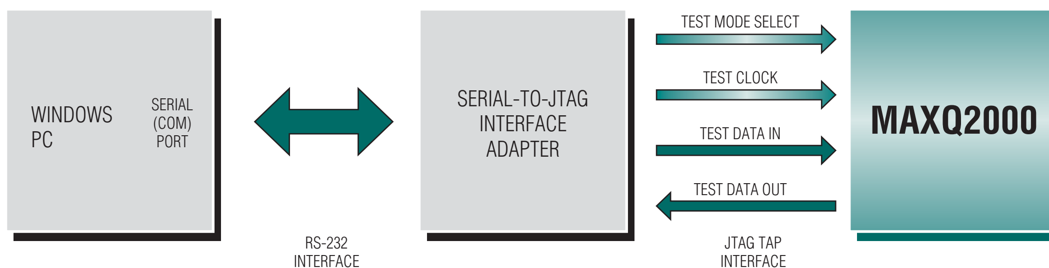

Figure 1. *The serial-to-JTAG adapter module allows software running on the PC to access the JTAG TAP interface of the MAXQ2000 microcontroller.*

Using the MAXQ2000 Evaluation Kit hardware

The MAXQ2000 Evaluation Kit provides a complete hardware development environment for the MAXQ2000 microcontroller, including the following features.

- On-board power supplies for the MAXQ2000 core and VDDIO supply rails.
- Adjustable power supply (1.8V to 3.6V), which can be used for the VDDIO or VLCD supply rails.
- Header pins for all MAXQ2000 signals and supply voltages.
- Separate LCD daughterboard connector.
- LCD daughterboard with 3V, 3.5-digit static LCD display.

- Full RS-232 level drivers for serial 0 UART including flow control lines.
- Pushbuttons for external interrupts and microcontroller system reset.
- MAX1407 multipurpose ADC/DAC IC, connected to the MAXQ2000 SPI bus interface.
- 1-Wire® interface, including iButton® clip and 1-Wire EEPROM IC.
- Bar graph LED display for levels at port pins P0.7 to P0.0.
- JTAG interface for application load and in-system debugging.

With the combination of the MAXQ2000 Evaluation Kit and the serial-to-JTAG adapter module, IAR Embedded Workbench has full access to the JTAG-based bootloader and in-circuit debugging features of the MAXQ2000.

Setting the MAXQ2000 Evaluation Kit board and the serial-to-TJAG interface modules for application development is straightforward. Simply connect the boards by the following steps.

1) Plug a 5V DC-regulated power supply (center post positive, ±5%) into the serial-to-JTAG board power jack J2.
2) Plug a 5V to 9V DC power supply into the MAXQ2000 Evaluation Kit board power jack J1.
3) Connect a straight-through DB9 serial cable from the serial-to-JTAG board J1 connector to one of the COM ports on the PC.
4) Connect the JTAG adapter cable from the 1 x 9 connector P2 on the serial-to-JTAG board to the 2 x 6 connector J4 on the MAXQ2000 Evaluation Kit board.
5) Turn both DC power supplies ON.
6) For standard operation, all DIP switches on the MAXQ2000 Evaluation Kit board should be in the OFF position.

Application development using IAR Embedded Workbench

The IAR Embedded Workbench development environment provides C-based or assembly-based application development for the MAXQ2000. Using the previous hardware configuration that includes the MAXQ2000 Evaluation Kit board and the serial-to-JTAG adapter module, IAR Embedded Workbench has full access to the JTAG-based bootloader and in-circuit debugging features of the MAXQ2000.

IAR Embedded Workbench provides the following features when developing applications for the MAXQ2000.

- Load compiled applications to the MAXQ2000 integrated program-flash memory.
- Step-by-step (trace) program execution at the C or assembly level.
- Display of code, data, hardware stack, and utility ROM memory.
- Call stack tracing.
- Breakpoint setting at the C or assembly level.
- View and edit of all MAXQ2000 system and peripheral registers.

Creating and compiling a project for the MAXQ2000

The IAR Embedded Workbench development environment provides C-based or assembly-based application development for the MAXQ2000.

Because IAR Embedded Workbench includes integrated support for the MAXQ microcontroller family, creating a new project for the MAXQ2000 microcontroller requires only a few specific settings.

After starting IAR, select **File**, then **New** from the menu. Select *Workspace* from the **New** dialog box and click **Ok**. Enter a new name for the project workspace (stored as a ".eww" file) and click **Save**.

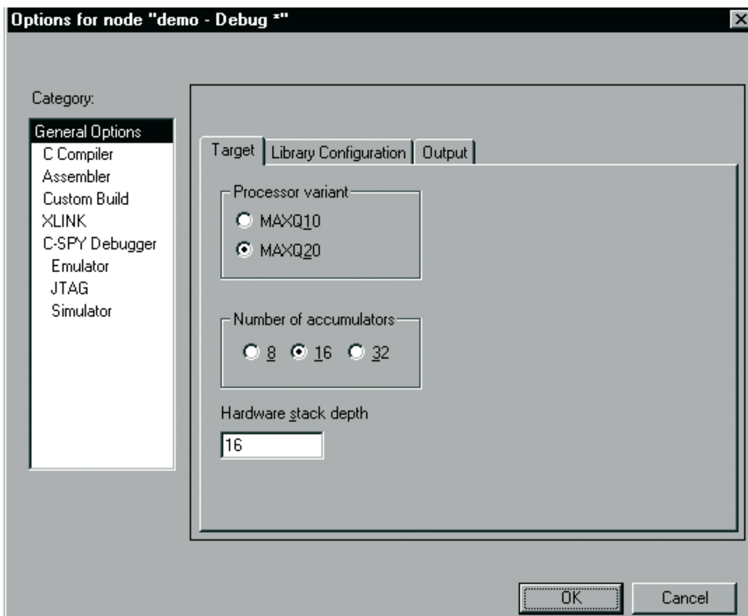

Figure 2. The General Options section of the Options dialog allows the user to specify the processor core type (MAXQ10/20), the number of accumulators available, and the hardware stack depth. The settings shown are for the MAXQ2000.

Once the workspace window opens, select **Project,** then **Create New Project** from the menu. The *MAXQ* tool chain is the default for the new project. Enter a file name for the new project (stored as a *.ewp file) and click **Create**.

Next, select **Project**, then **Settings** from the menu. A dialog box will appear with the settings for the newly created project, as shown in **Figure 2**.

In the *General Options* tab of the **Options** dialog box, the following settings should be selected for the MAXQ2000 microcontroller.

- **Processor Variant** should be set to **MAXQ20**, as the MAXQ2000 has a MAXQ20-type core.
- **Number of accumulators** should be set to **16** for the MAXQ2000.
- **Hardware stack depth** should be set to **16** for the MAXQ2000.

In the *C-SPY Debugger* tab of the **Options** dialog box, the following settings should be selected for the MAXQ2000 (**Figure 3**):

- Set the **Driver** setting to **JTAG** to connect to the serial-to-JTAG interface board over a PC COM port. The other two possible settings are **Simulator** (used to run with the MAXQ2000 software simulator) and **Emulator** (used to run with the MAXQ2000 In-Circuit Emulator system).

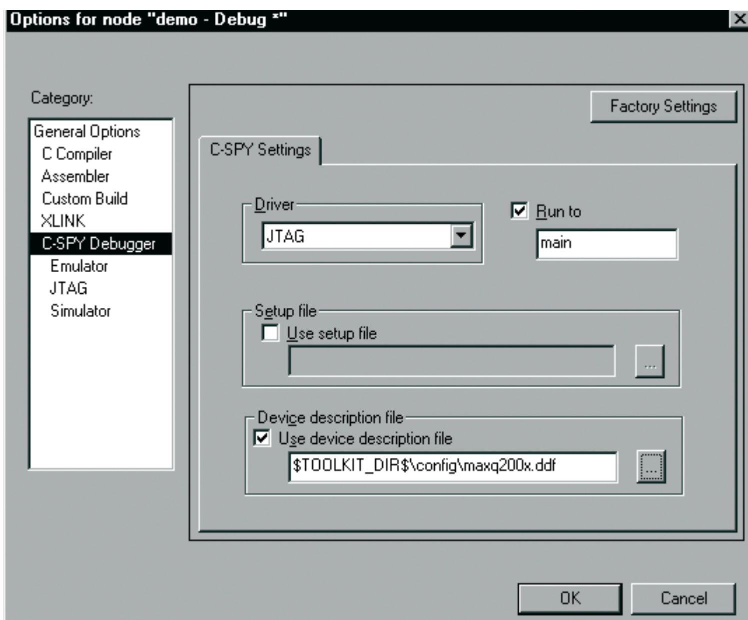

Figure 3. The C-SPY Debugger section of the Options dialog allows the user to specify settings for debugging sessions. The settings shown are for debugging the MAXQ2000 using the serial-to-JTAG adapter module.

- The **Use Device Description File** box should be checked. The device description file (*.ddf) should be the file provided for the MAXQ2000 microcontroller (maxq200x.ddf). This file defines the memory spaces and peripheral register set for a particular MAXQ micro-controller for use by the IAR environment.

Under the *JTAG* section of the **Options** dialog box, the **Command line options** field contains the COM port of the PC used to connect to the serial-to-JTAG board. **Figure 4** shows the option setting for connecting to COM port 1.

After setting the options for the project, select **Project**, then **Add Files** to add a C code file to the project. Once the project file(s) have been added, select **Project**, then **Make** to compile the project, followed by **Project**, then **Debug** to start a debugging session. This downloads the compiled project over the JTAG interface and places IAR into debug mode, as **Figure 5** shows.

Debugging operations in IAR

Once the debugging session has started, **Step Over** (F10), **Step Into** (F11), and **Step Out** (Shift+F11) can be used to trace through the C code of the project. To run code, select **Debug**, then **Go** from the menu, or hit F5.

Address breakpoints can be set or cleared by placing the cursor on a line of source code and clicking the **Toggle Breakpoint** button in the toolbar. Up to four address breakpoints can be set at once.

The **Memory** window can be used to display the **Code** (internal flash memory), **Data** (internal SRAM), **Hw stack** (internal 16-level stack), and utility ROM memories of the MAXQ2000. The memory display can be set to byte, word, or doubleword format, and displays in both hex (for all widths) and ASCII (for byte width) formats.

The **Register** window displays the system and peripheral registers for the MAXQ2000. These are displayed in logical groups.

- **CPU Registers**: Accumulator and accumulator control registers, data pointers and data pointer control registers, instruction pointer, loop counter, and program status flags.
- **Interrupt Control**: Interrupt vector, module mask, and identification registers.
- **Cycles**: Displays the number of instruction cycles that have executed.
- **Parallel Ports**: Input, output, and port direction registers for P0 to P4.
- **External Interrupt**: Enable, edge select, and flag registers for external interrupts.
- **Timers**: Registers for timer/counters 0 to 2.
- **Serial Port**: Control and buffer registers for the SPI and serial ports.
- **Multiplier**: Registers related to the hardware multiplier module.

Writeable registers can be edited by clicking on the register value and entering a new value. Display of the individual bits or bit fields within registers can be expanded or collapsed by clicking the plus/minus sign next to the register name.

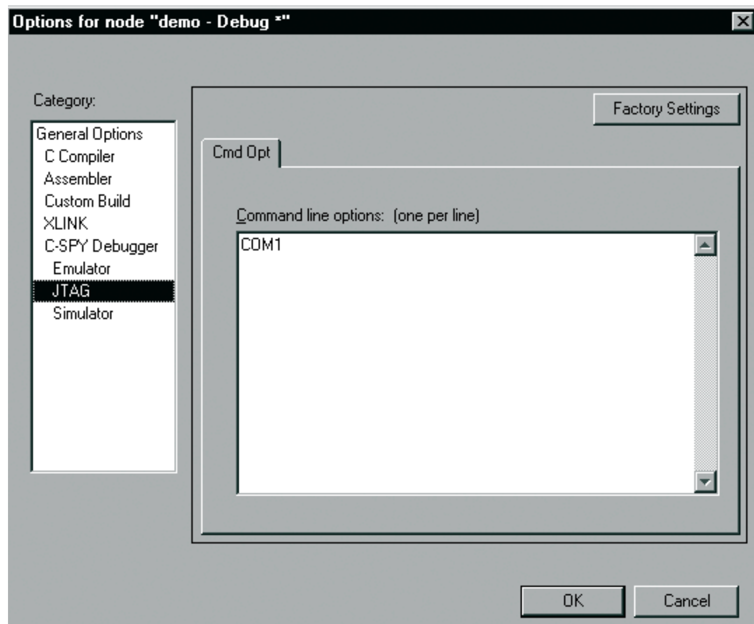

Figure 4. The C-SPY Debugger (JTAG) section of the Options dialog allows the user to change settings specific to the serial-to-JTAG adapter module. The settings shown are for a serial-to-JTAG adapter connected to the PC port COM1.

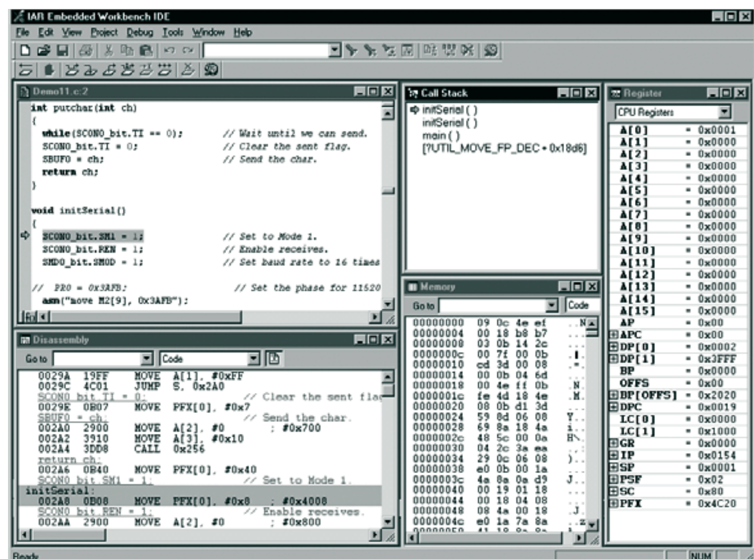

Figure 5. Using the serial-to-JTAG adapter module, IAR Embedded Workbench can perform step-by-step execution on the MAXQ2000, as well as read and modify on-chip memory and register values.

Conclusion

The high-level, C-project-based environment of IAR Embedded Workbench integrates with the MAXQ2000's low-level debugging interface to allow fine-tuned debugging at either the C or assembly code levels. The MAXQ2000's built-in debugging and in-circuit programming features, and their low-level impact on system resources, allow the same hardware design to be used for both the application development process and for the final release of the finished project.

1-Wire and iButton are registered trademarks of Dallas Semiconductor.

Accessing the functions provided in the MAXQ utility ROM

All MAXQ utility ROMs include routines for accessing data and tables stored in the program space.

Using lookup tables within the application code is a common programming practice when working with microcontrollers. Because of the single-cycle nature of the MAXQ core, application software cannot read directly from code space and, therefore, cannot directly access any tables defined within the application code. To alleviate this issue, all MAXQ utility ROMs include routines for accessing data and tables stored in the program space. In addition to these core functions, the ROM for each MAXQ variation may have routines specific to that part. Because these functions might be located anywhere within the ROM, and could move with each revision of a ROM, a standard technique was developed for accessing the routines. This allows code written for one version of the ROM to be reused with all subsequent revisions without needing to rewrite or recompile the code.

For all variants of the MAXQ processor, the utility ROM has a table with addresses for each of its supported functions. The location of this table can vary from part to part, so a pointer to this table is always stored at address 800Dh. The addresses for the supported functions can then be found by indexing into the table. This table always maintains the same order for the functions throughout all revisions of a particular ROM. **Table 1** lists the MAXQ2000 functions and their entry point within the table.

Table 1. MAXQ2000 Utility ROM User-Function Table

FUNCTION NUMBER	FUNCTION NAME	ENTRY POINT (USERTABLE = ROM[800Dh])
0	Reserved	ROM[userTable + 0]
1	Reserved	ROM[userTable + 1]
2	Reserved	ROM[userTable + 2]
3	moveDP0	ROM[userTable + 3]
4	moveDP0inc	ROM[userTable + 4]
5	moveDP0dec	ROM[userTable + 5]
6	moveDP1	ROM[userTable + 6]
7	moveDP1inc	ROM[userTable + 7]
8	moveDP1dec	ROM[userTable + 8]
9	moveFP	ROM[userTable + 9]
10	moveFPinc	ROM[userTable + 10]
11	moveFPdec	ROM[userTable + 11]
12	copyBuffer	ROM[userTable + 12]

Executing a utility ROM function requires four steps. Firstly, retrieve the location of the function table from address 800Dh. Secondly, add the offset for the desired function. Thirdly, retrieve the address of the utility function by reading from the computed location. Finally, execute the function by performing a call to the location found in the table. The following MAXQ assembly function demonstrates these four steps, using the `moveDP1inc` function of the MAXQ2000 as an example.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function:     ReadDataAtDP1
;; Description:  This function uses the utility ROM function "moveDP1inc"
;;               to read from program memory the data stored at the
;;               address in DP[1].  If DP[1] is in word mode two
;;               bytes will be read.  If DP[1] is in byte mode only
;;               one byte is read.  DP[1] is then post incremented.
;; Returns:      The result is returned in GR.
;; Destroys:     ACC and DP[0]
;; Notes:        This function assumes that DP[0] is set to word
;;               mode and the device has 16-bit accumulators.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ReadDataAtDP1:
  move  DP[0], #0800Dh ; This is where the address of the table is stored.
  move  ACC, @DP[0]    ; Get the location of the function table.
  add   #7             ; Add the index to the moveDP1inc function.
  move  DP[0], ACC     ; Point to where the address of moveDP1 is stored.
  move  ACC, @DP[0]    ; Retrieve the address of the function.
  call  ACC            ; Execute the function.
  ret
```

Because future ROM versions of a particular MAXQ variant might place the utility functions in a different location, using a routine similar to the `ReadDataAtDP1` function guarantees forward compatibility. The "cost" of this compatibility is larger code size and longer execution times. In some cases, these tradeoffs might be unacceptable, making it worthwhile to call the utility ROM functions directly. To call a utility function directly, simply determine the location of the desired function and use this location as the destination of a `call`.

Reading a string defined in code space illustrates a common situation requiring the use of utility functions. A programmer might store error strings, informational strings, or even debug strings that get displayed during execution of an application. The code segment below shows one way of achieving this using the `ReadDataAtDP1` function as previously described.

```
Text:
  DB  "Hello World!",0                ; Define a string in code space.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Function:     PrintText
;; Description:  Prints the string stored at the "Text" label.
;; Returns:      N/A
;; Destroys:     ACC, DP[1], DP[0], and GR.
;; Notes:        This function assumes that DP[0] is set to word mode,
;;               DP[1] is in byte mode, and the device has 16-bit
;;               accumulators.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
PrintText:
  move  DP[1], #Text      ; Point to the string to display.
  move  ACC, DP[1]        ; "Text" is a word address and we need a
  sla                     ; byte address, so shift left 1 bit.
  or    #08000h           ; Code space is mapped to 8000h when running
  move  DP[1], ACC        ; from the ROM, so the address must be masked.
PrintText_Loop:
  call  ReadDataAtDP1     ; Fetch the byte from code space.
  move  ACC, GR
  jump  Z, PrintText_Done ; Reached the null terminator.
  call  PrintChar         ; Call a routine to output the char in ACC
  jump  PrintText_Loop    ; Process the next byte.
PrintText_Done:
  ret
```

Having a common way of accessing utility ROM routines also allows developers to write code that will work with all variants of a particular MAXQ processor.

Conclusion

Utility functions give developers an easy way to read data stored in program memory. Having a common way of accessing utility ROM routines also allows developers to write code that will work with all variants of a particular MAXQ processor. Libraries can be constructed once and then reused, eliminating concern that future ROM versions will not be compatible.

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."

Signal processing with the MAXQ multiply-accumulate unit (MAC)

In the modular MAXQ architecture, a single-cycle multiply-accumulate (MAC) unit is incorporated to facilitate operation required for a typical signal-processing technique.

Traditional microcontrollers and digital signal processors (DSPs) are sometimes viewed as standing at opposite ends of the microcomputer spectrum. While microcontrollers are best suited for control applications that require low-latency response to unsynchronized events, DSPs shine in applications where intense mathematical calculations are required. A microcontroller *can* be used in heavy arithmetic applications, but the one-operation-at-a-time nature of most microcontroller ALUs makes such use less than optimal. Similarly, a DSP can be forced into a control application, but the internal architecture of most DSPs render this operation inefficient in both code and time.

Choosing a DSP or a traditional microcontroller becomes more difficult when a mostly control-oriented application requires a small amount of signal processing. In such applications, it is tempting to squeeze the DSP code into the microcontroller. However, the designer often finds that the application spends most time performing DSP functions, thus making the control application suffer.

This dichotomy can be resolved in modern processor architectures, such as the MAXQ architecture. In the modular MAXQ architecture, a multiply-accumulate unit (MAC) can be added to the design and integrated into the architecture with ease. With the hardware MAC, 16 x 16 multiply-accumulate operations occur in one cycle without compromising the application running on the control processor. This article provides some examples of how the MAC module in a typical MAXQ microcontroller can be used to solve such real-world problems.

Using the MAC module with a MAXQ

A common application for DSPs is filtering some analog signal. In this application, a properly conditioned analog signal is presented to an ADC, and the resulting stream of samples is filtered in the digital domain. A general filter implementation can be realized by the following equation:

$$y[n] = \sum b_i x[n-i] + \sum a_i y[n-i]$$

where b_i and a_i characterize the feedforward and feedback response of the system, respectively.

Depending on the values of a_i and b_i, digital filters can be classified into two broad categories: finite impulse response (FIR) and infinite impulse response (IIR). When a system does not contain any feedback elements (all $a_i = 0$), the filter is said to be of the FIR type:

$$y[n] = \sum b_i x[n-i]$$

However, when elements of both a_i and b_i are non-zero, the system is an IIR filter.

As can be seen from the above equation for an FIR filter, the main mathematical operation is to multiply each input sample by a constant, and then accumulate each of the products over the n values. The following C fragment illustrates this:

```
y[n]=0;
for(i=0; i<n; i++)
        y[n] += x[i] * b[i];
```

For a microprocessor with a multiplier unit, this can be achieved according to the following pseudo-assembler code:

```
move ptr0, #x      ;Primary data pointer -> samples
move ptr1, #b      ;Secondary DP -> coefficients
move ctr, #n       ;Loop counter gets number of samples
move result, #0    ;Clear result register
```

```
ACC_LOOP:
        move acc, @ptr0    ;Get a sample
        mul   @ptr1        ;Multiply by coefficient
        add   result       ;Add to previous result
        move  result, acc  ;...and save the result back
        inc   ptr0         ;Point to next sample
        inc   ptr1         ;Point to next coefficient
        dec   ctr          ;Decrement loop counter
        jump  nz, ACC_LOOP ;Jump if there are more samples
        end
```

Thus, even with a multiplier, the multiply and accumulate loop requires 12 instructions and (assuming a one-cycle execution unit and multiplier) 4 + 8n cycles.

The MAXQ multiplier is a true multiply-accumulate unit. Performing the same operation in the MAXQ architecture shrinks code space from 12 words to 9 words, and execution time is reduced to 4 + 5n cycles.

```
        move DP[0], #x        ; DP[0] -> x[0]
        move DP[1], #b        ; DP[1] -> b[0]
        move LC[0], #loop_cnt ; LC[0] -> number of samples
        move  MCNT, #INIT_MAC ; Initialize MAC unit
MAC_LOOP:
        move DP[0], DP[0]     ; Activate DP[0]
        move MA, @DP[0]++     ; Get sample into MAC
        move DP[1], DP[1]     ; Activate DP[1]
        move MB, @DP[1]++     ; Get coeff into MAC and multiply
        djnz  LC[0], MAC_LOOP
```

Note that in the MAXQ multiply-accumulate unit, the requested operation occurs automatically when the second operand is loaded into the unit. The result is stored in the MC register. Note also that the MC register is 40 bits long, and thus can accumulate a large number of 32-bit multiply results before overflow. This improves on the traditional approach where overflow must be tested after every atomic operation. To illustrate how the MAC can be used efficiently in the signal-processing flow, we present a simple application for a dual-tone multi-frequency (DTMF) transceiver.

...in the MAXQ multiply-accumulate unit, the requested operation occurs automatically when the second operand is loaded into the unit.

DTMF overview

DTMF is a signaling technique used in the telephone network to convey address information from a network terminal (a telephone or other device) to a switch. The mechanism uses two sets of four discrete tones that are not harmonically related, i.e., the “low group” (less than 1kHz) and the “high group” (greater than 1kHz). Each digit on the telephone keypad is represented by exactly one tone from the low group and one tone from the high group. See **Figure 1** to learn how the tones are allocated.

The dual-tone multi-frequency (DTMF) signaling technique used in the telephone network conveys address information from a network terminal (telephone or other device) to a switch.

DTMF tone encoder

The encoder portion of the DTMF transceiver is relatively straightforward. Two digital sine-wave oscillators are required, each of which can be tuned to one of the four low-group or high-group frequencies.

There are several ways to resolve the issue of digitally synthesizing a sine wave. One method of sine-wave generation avoids the issue of digital synthesis altogether. Instead, it just strongly filters a square wave produced on a port pin. While this method works in many applications, Bellcore requirements dictate that the spectral purity of the sine waves be higher than can be achieved using this technique.

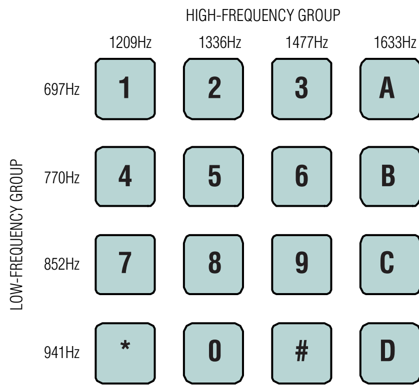

Figure 1. Combining one frequency from the high-frequency group and one from the low-frequency group generates a DTMF signal.

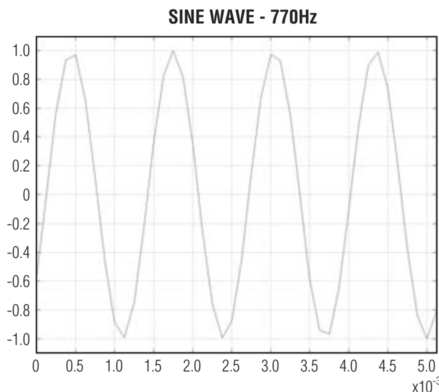

Figure 2. A recursive resonator generates the sine wave.

A second method of generating sinusoidal waveforms is the table-lookup method. In this method, one-quarter of a sine wave is stored in a ROM table, and the table is sampled at a precomputed interval to create the desired waveform. Creating a quarter-sine table of sufficiently high resolution to meet spectral requirements would, however, require a significant amount of storage. Fortunately, there is a better way.

A recursive digital resonator[1] can be used to generate the sinusoids (**Figure 2**). The resonator is implemented as a two-pole filter described by the following difference equation:

$$X_n = k * X_{n-1} - X_{n-2}$$

where *k* is a constant defined as

$$k = 2 \cos(2\pi * toneFrequency / samplingRate)$$

Because only a small number of tones are needed in a DTMF dialer, the eight values of k can be precomputed and stored in ROM. For example, the constant required to produce a Column 1 tone (770Hz) at a sample rate of 8kHz is:

$$k = 2 \cos(2\pi * 770 / 8000) = 2 \cos(0.60) = 1.65$$

One more value must be calculated: the initial impulse required to make the oscillator begin running. Clearly, if X_{n-1} and X_{n-2} are both zero, every succeeding X_n will be zero. To start the oscillator, set X_{n-1} to zero and set X_{n-2} to

$$X_{n-2} = -A * \sin(2\pi * toneFrequency / samplingRate)$$

In our example, assuming a unit sine wave is desired, this reduces to:

$$X_{n-2} = -1 * \sin(2\pi * 770 / 8000) = -\sin(0.60) = -0.57$$

Reducing this to code is simple: first, two intermediate variables (X1, X2) are initialized. X1 is initialized to zero, while X2 is loaded with the initial excitation value (calculated above) to start the oscillation. To generate one sample of the sinusoid, perform the following operation:

$$X0 = k * X1 - X2$$
$$X2 = X1$$
$$X1 = X0$$

The MAXQ microcontroller, together with its MAC unit, is bridging the gap between the traditional microcontroller and the digital signal processor.

Each new sine value is calculated using one multiplication and one subtraction. With a single-cycle hardware MAC on the MAXQ microcontroller, the sine wave can be generated as follows:

```
move DP[0], #X1           ; DP[0] -> X1
move MCNT, #INIT_MAC      ; Initialize MAC unit
move MA, #k               ; MA = k
move MB, @DP[0]++         ; MB = X1, MC=k*X1, point to X2
move MA, #-1              ; MA = -1
move MB, @DP[0]--         ; MB = X2, MC=k*X1-X2, point to X1
nop                       ; wait for result
move @--DP[0], MC         ; Store result at X0
```

DTMF tone detection

Because only a small number of frequencies are to be detected, the modified Goertzel algorithm[2] is used. This algorithm is more efficient than the general DFT mechanisms and provides reliable detection of inband signals. It can be implemented as a simple second-order filter following the format in **Figure 3**.

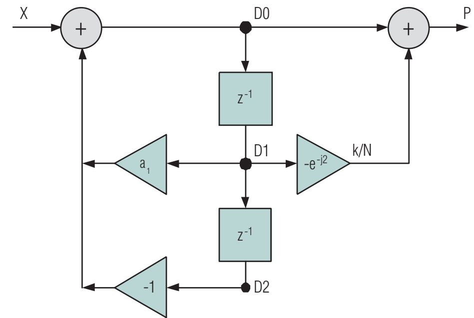

Figure 3. The Goertzel algorithm is implemented as a second-order filter.

To use the Goertzel algorithm to detect a tone of a particular frequency, a constant must first be precomputed. For a DTMF detector, this can be done at compile time. All the tone frequencies are well specified. The constant is computed from the following formula:

$$k = toneFrequency / samplingRate$$
$$a_1 = 2\cos(2\pi k)$$

First, three intermediate variables (D0, D1, and D2) are initialized to zero. Now, for each sample X received, perform the following:

$$D0 = X + a_1 * D1 - D2$$
$$D2 = D1$$
$$D1 = D0$$

After a sufficient number of samples has been received (usually 205 if the sample rate is 8kHz), compute the following using the latest computed values of D1 and D2:

$$P = D1^2 + D2^2 - a_1 * D1 * D2$$

P now contains a measure of the squared power of the test frequency in the input signal. To decode full four-column DTMF, each sample will be processed by eight filters. Each filter will have its own k value, and its own set of intermediate variables. Since each variable is 16 bits, the entire algorithm will require 48 bytes of intermediate storage.

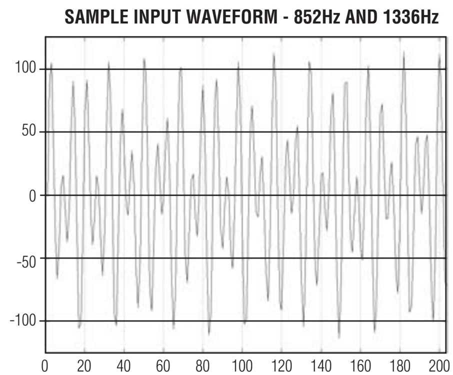

Figure 4. This is the sample input waveform for the DTMF decoder.

Once the P values for various tone frequencies are calculated, one tone in the high and low groups will have values significantly higher than all the other tones, which means more than twice as high, often more than an order of magnitude. **Figure 4** shows a sample input signal to the decoder, and **Figure 5** illustrates the result of the Goertzel algorithm. If the signal spectrum does not meet this criterion, it either means that no DTMF energy is present in the signal, or that there is sufficient noise to block the signal.

A spreadsheet that demonstrates this algorithm is available on our website, as well as sample code for the MAC-equipped MAXQ processor. Go to www.maxim-ic.com/MAXQ_DTMF.

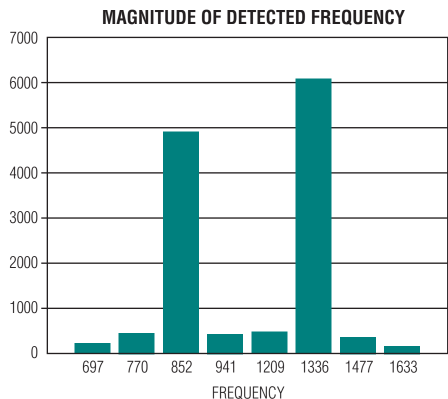

Figure 5. The DTMF decoder detects the magnitude of various frequencies.

Conclusion

The MAXQ microcontroller, together with its MAC, is bridging the gap between the traditional microcontroller and the digital signal processor. With the addition of a hardware MAC, the MAXQ microcontroller offers a new level of signal-processing capability to the 16-bit microcontroller market not previously available. Real-time signal processing is made possible with a single-cycle MAC that provides the functions most often required in real-world applications.

[1] Todd Hodes, John Hauser, Adrian Freed, John Wawrzynek, and David Wessel. Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-99, March 15–19, 1999), pp. 993–996.
[2] Alan Oppenheim and Ronald Schafer, Discrete-Time Signal Processing. Prentice Hall.

FREE SAMPLES AND TECHNICAL INFORMATION

FAX: 408-222-1770

www.maxim-ic.com/samples

Contact us at 1-800-998-8800 (Toll Free)
www.maxim-ic.com/samples

Please send me a sample of:

_______________ _______________
_______________ _______________
_______________ _______________
_______________ _______________

(Limit is 8 part numbers, 2 samples each.)

Request free information about Maxim and our products.

❒ Please send me **Future Issues of the Maxim *Engineering Journal*.**

Please complete the information below.

Name _______________ Title _______________
Company _______________ Department _______________
Address _______________
City _______________ State/Province _______________
Zip Code _______________ Country _______________
Telephone _______________
E-mail Address _______________
My application is _______________ My end product is _______________

MER4 9/04

Request Design Guides from Our Library

	1-Wire® Products
	A/D Converters
	Audio
	Battery Management
	Cellular/PCS Phones
	Communications
	D/A Converters
	Digital Potentiometers
	Displays
	Fiber Cable
	Flat-Panel Displays
	High-Speed ADCs & DACs
	High-Speed Interconnect
	High-Speed Microcontrollers
	Interface
	Low-Power Notebook
	µP Supervisors
	Multiplexers and Switches
	Op Amps and Comparators
	Power Supplies
	Real-Time Clocks
	Signal Conditioners
	System Timing and Control
	Temperature Sensors
	Video
	Voltage References
	Wireless

www.maxim-ic.com
Maxim Integrated Products, Inc.
120 San Gabriel Drive
Sunnyvale, CA 94086

Presorted Standard
U.S. Postage
PAID
St. Joseph, MI
Permit No. 126