DALLAS SEMICONDUCTOR MAXIM

Microcontroller ENGINEERING REVIEW

Volume 7

Unlocking the Secrets of the MAXQ

The MAXQ® processor family are powerful 8-, 16-, and 32-bit, single-cycle microcontrollers that perform multiple operations in one clock cycle. This article explores the internal workings of the MAXQ20 core, and showcases its immense power.

Programmer's Model

The MAXQ20 core is a 16-bit CPU, meaning that all accumulators and most working registers (stack, data pointers, counters) are 16 bits in length. The MAXQ20 can address 64kWords of code space (that is, 64kB instructions) and 64kWords (128kB) of data space (**Figure 1**).

Note that, for a processor based on the MAXQ20 core, much of this memory space will be vacant. Additionally, because the utility ROM and data RAM reside in the upper 32kb of code space, access to user code in this region requires special features in the core that are beyond the scope of this article.

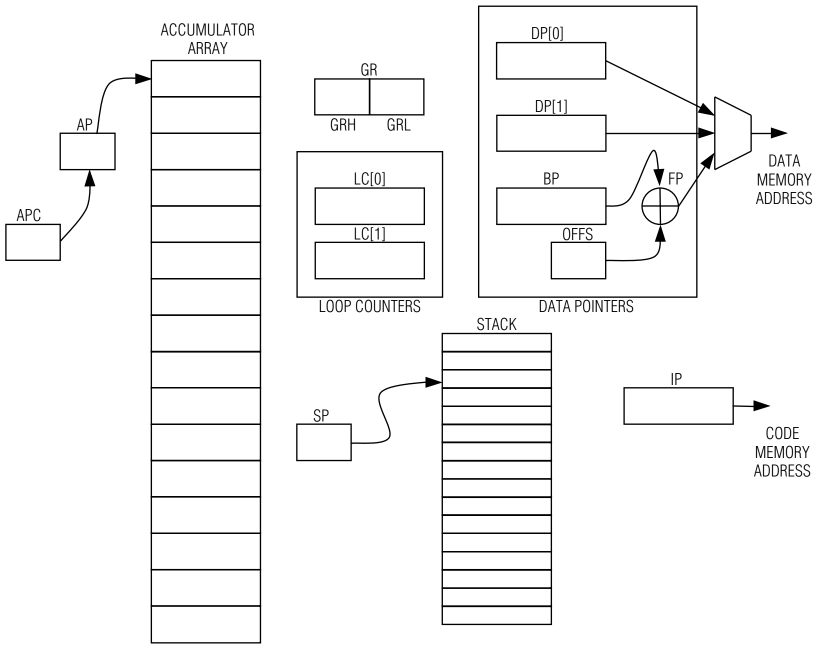


Figure 1. The programmer's model for the MAXQ20 core consists of 16 general-purpose accumulators, two loop counters, and a set of data pointers.

Accumulators

Sixteen registers known as "accumulators" form a general-purpose register array. The register to which the Accumulator Pointer register (AP) points is designated the "active accumulator," which is the target of arithmetic and logical operations. Thus, by changing the value in the AP register, any of the 16 accumulators can be designated the target of an arithmetic logic unit (ALU) operation. The Accumulator Pointer Control register (APC) causes the AP to increment or decrement automatically whenever the active accumulator is accessed, thus making multiprecision arithmetic simple. In **Figure 2**, A[0] is the active accumulator, but any accumulator access can make A[1] or A[15] the active accumulator, depending on the value of the APC register.

GR Register

The General Register (GR) aids in the extraction of individual bytes from a 16-bit word. A programmer can use GR to assemble bytes into a word: load the low byte into GRL (General Register–Low byte), the high byte into GRH (General Register–High byte), and read the assembled word in GR. Alternately, a programmer can use the GR register to decompose a word into its constituent bytes. A word loaded into GR can be read in a byte-swapped format in GRS (General Register–Swapped). Finally, a byte loaded into the GRL register can be sign-extended to a word by reading GRXL (General Register eXtend Low byte). See **Figure 3**.

Table of Contents

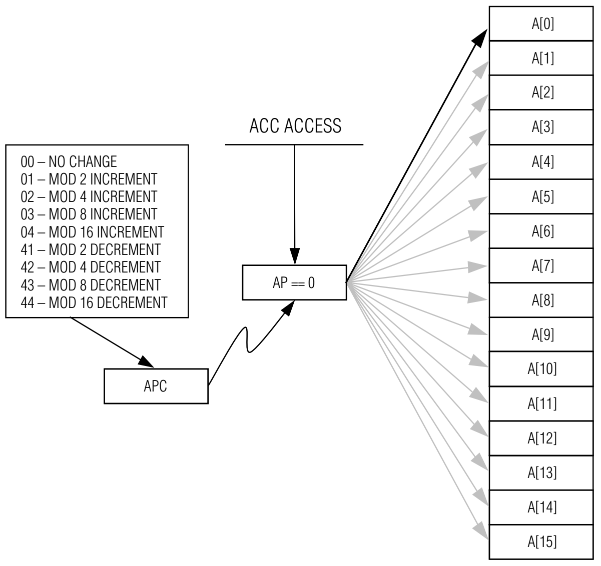


Figure 2. The active accumulator is designated by the AP register, which itself can be modified by accumulator access instructions.

Loop Counters

There are two loop counters: Loop Counter 0 (LC[0]) and Loop Counter 1 (LC[1]). These registers can be used as general-purpose registers, but are intended as loop counters for decrement and jump if the counters are nonzero (DJNZ) instructions.

Stack

The MAXQ20 core has a dedicated, 16-level internal stack. A stack pointer indicates the next stack location to be used or indicates PUSH or CALL operations.

Data Memory Pointers

The MAXQ microcontroller has three pointers to access data memory. Two, DP[0] and DP[1], are simple 16-bit pointers. The third pointer is formed by adding a base address pointer (BP) to an 8-bit unsigned offset (OFFS).

Note that the data memory, as addressed by one of the three data pointers, is distinctive from the code memory, addressed by the instruction pointer. While all MAXQ processors include a memory management unit (MMU) that allows any memory segment to be treated as code or data, *the code and data buses are separate*. This separation of buses for code and data fetch operations is a fundamental element of the MAXQ20 technology, and allows simultaneous code and data access in a single clock cycle.

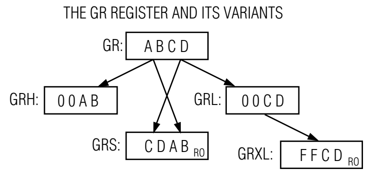


Figure 3. The GR register supports byte extraction, byte-swapping, and 16-bit sign extension.

Transfer-Triggered Architecture

By inspecting the programmer's model, one could conclude that there is a conventional instruction fetch-decode unit that loads an instruction, decodes it, and then activates certain elements of the CPU. That, however, would be a misconception. What sets the MAXQ architecture apart from other, more conventional CPUs is the *transfer-triggered* nature of the MAXQ core.

Transfer-triggering is a technique that allows a simple MOVE instruction to perform every function available in the CPU. While the MAXQ assembler supports more than 30 instruction op codes, one could encode every instruction in the MAXQ instruction set as:

```
move Ma[b], Mc[d]
```
or
```
move Ma[b], #immediate_value
```

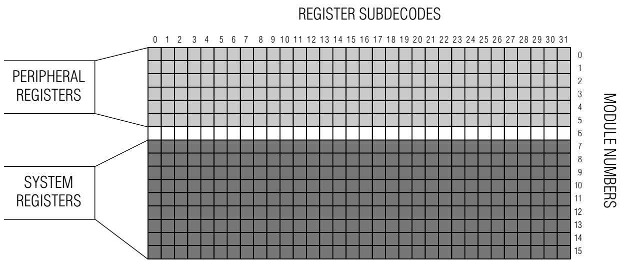


Figure 4. Register assignments in the MAXQ20 core are split into two regions: register banks 0 to 5 are peripheral registers and can change from one MAXQ part to another; banks 7 to 15 are system registers and remain relatively fixed on all MAXQ parts.

where the designation `Ma[b]` describes register module a and register subdecode b. Simply stated: every instruction—ADD, bit manipulation, reference to external memory—is coded as a move between two registers or as a move of an immediate value into a register.

When a MAXQ instruction is executed, the destination register is loaded with the contents of the source register or with an immediate value. In addition, this transfer of data can trigger other events like incrementing or decrementing a pointer, setting some status bit, or some other function. Hence, the architecture is transfer-triggered. To support this architecture, a large register complement is needed. In the MAXQ20 core, there is a total of 512 register addresses divided into two broad sections: peripheral register space and system register space (**Figure 4**).

The first six register modules (modules 0 to 5) are dedicated to peripheral registers; the last nine modules (modules 7 to F) are assigned as system registers. (Module 6 is reserved.) While the peripheral register modules change from one type of MAXQ processor to another, the system registers remain the same across all MAXQ processors (**Figure 5**).

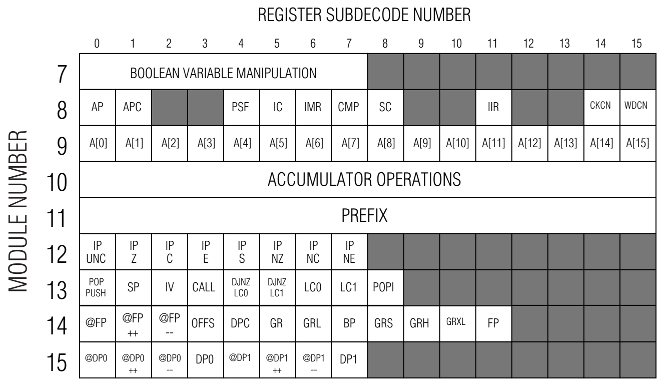
Decoding a MAXQ Instruction

Because every MAXQ instruction is really a MOVE, every instruction can be broken down into three fields: a SOURCE field that designates where the data is moved FROM; a DESTINATION field that designates where the data is moved TO; and a format bit that indicates whether the source is an immediate value (FORMAT == 0) or a register designator (FORMAT == 1) (**Figure 6**).

Take the instruction op code `0x0923`, for example. In this instruction, the FORMAT bit is clear, indicating that the source designator `(23)` should be treated as an 8-bit immediate value. The destination module is module 9, the accumulator array. Register 0 within that array is the accumulator A[0]. So, the effect of the instruction is to load the value `0x0023` into register A[0]. In this case, there are no side effects associated with either the source or destination designators.

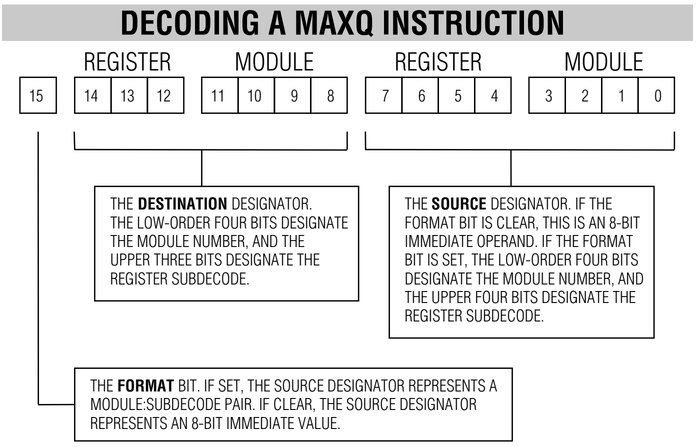
For a second example, consider `0xBF09`. In this instruction, the FORMAT bit is set, meaning that the source designator should be interpreted as a register. Module 9 register 0 was covered above: it is the accumulator A[0]. On the destination side, module F is the data pointer module, and register 3 (bits 14:12 in the instruction) represents data pointer DP[0]. Therefore, this instruction moves the contents of A[0] to DP[0].

Note that in some cases individual locations inside a register module may or may not refer to actual registers. Alternatively, they can refer to an actual register but then cause some side effect to occur when that register subdecode is accessed. For example, let us modify the previous example slightly with `0xAF09`. Only the destination subdecode has changed. Now instead of loading the register DP[0], the instruction decrements DP[0] and then begins a store operation to the new memory location to which DP[0] points. That is, the instruction performs an indirect store on a predecremented pointer. In the MAXQ assembler this would be coded as `move @--DP[0], A[0]`, but it could be as easily coded as `move M15[2], M9[0]`.



MODULE NUMBER	REGISTER SUBDECODE NUMBER															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	BOOLEAN VARIABLE MANIPULATION								▨	▨	▨	▨	▨	▨	▨	▨
8	AP	APC	▨	▨	PSF	IC	IMR	CMP	SC	▨	▨	IIR	▨	▨	CKCN	WDCN
9	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]
10	ACCUMULATOR OPERATIONS															
11	PREFIX															
12	IP UNC	IP Z	IP C	IP E	IP S	IP NZ	IP NC	IP NE	▨	▨	▨	▨	▨	▨	▨	▨
13	POP PUSH	SP	IV	CALL	DJNZ LC0	DJNZ LC1	LC0	LC1	POPI	▨	▨	▨	▨	▨	▨	▨
14	@FP	@FP ++	@FP --	OFFS	DPC	GR	GRL	BP	GRS	GRH	GRXL	FP	▨	▨	▨	▨
15	@DP0	@DP0 ++	@DP0 --	DP0	@DP1	@DP1 ++	@DP1 --	DP1	▨	▨	▨	▨	▨	▨	▨	▨

Figure 5. *The MAXQ system register map consists of the registers present in all MAXQ20-based processors and additional decodes to implement the instruction set.*



THE **DESTINATION** DESIGNATOR. THE LOW-ORDER FOUR BITS DESIGNATE THE MODULE NUMBER, AND THE UPPER THREE BITS DESIGNATE THE REGISTER SUBDECODE.

THE **SOURCE** DESIGNATOR. IF THE FORMAT BIT IS CLEAR, THIS IS AN 8-BIT IMMEDIATE OPERAND. IF THE FORMAT BIT IS SET, THE LOW-ORDER FOUR BITS DESIGNATE THE MODULE NUMBER, AND THE UPPER FOUR BITS DESIGNATE THE REGISTER SUBDECODE.

THE **FORMAT** BIT. IF SET, THE SOURCE DESIGNATOR REPRESENTS A MODULE:SUBDECODE PAIR. IF CLEAR, THE SOURCE DESIGNATOR REPRESENTS AN 8-BIT IMMEDIATE VALUE.

Figure 6. *A MAXQ instruction consists of three parts: a source designator, a destination designator, and a source format bit to determine if the source is an immediate operand or a register operand.*

The Prefix Register

There are 32 registers per module, but only four bits to select a source register and only three bits to designate a destination register. At first glance, this implies that half the register subdecodes could not be read, and fully three-quarters of register subdecodes could not be written. Fortunately, the MAXQ architecture design works around this. Every MAXQ processor provides a prefix register to supply these additional register address bits, and to provide the upper byte of a word-wide move. See the *Module 11—Prefix* section for details.

Creating the MAXQ Instruction Set One Module at a Time

The following sections detail the system register modules and how they interact to create all the documented and undocumented instructions. We first investigate the heart of the MAXQ20 core: the accumulator array.

The MAXQ processor family is a collection of powerful 8-, 16-, and 32-bit, single-cycle microcontrollers that perform multiple operations in one clock cycle.

The MAXQ20 core is a 16-bit CPU, meaning that all accumulators and most working registers (stack, data pointers, counters) are 16 bits in length.

Module 9—Accumulators

The MAXQ architecture supports up to 32 accumulators, although in most variants only 16 are implemented. The accumulators are directly accessed through module 9. Each subdecode within this module represents a single accumulator. Module 9 is conceptually the simplest of the modules, but there are two more modules that affect the accumulator array.

Module 8—System Control

This module contains a number of registers that manage aspects of system operation, such as interrupt control and program status flags. Many of these registers are beyond the scope of this article, so refer to the device specifications for more information.

The AP and APC registers deserve special attention. The AP register determines which of the accumulator registers is the active accumulator; that is, it designates the target for arithmetic, logic, and bitwise operations. It can point to any accumulator in the array.

The APC register contains a set of bits that define how the AP register is modified following any accumulator operation. Thus, the AP register can be incremented or decremented with the count rolling over on a selectable power-of-two modulus, making multiprecision arithmetic simple.

Module 10—Accumulator Functions

Module 10 is where most of the accumulator's actual work is accomplished. It provides access to the traditional ALU functions and bit-level access to the active accumulator. Module 10 is unique; it behaves differently depending on whether it serves as the source, as the destination, or as both source and destination.

The MAXQ20 can address 64kWords of code space (that is, 64kB instructions) and 64kWords (128kB) of data space.

If the source is module 10 and the destination is any module other than module 10, the accumulator's contents are moved to the destination. If the subdecode was zero, the AP register is modified according to the bits in the APC register. If the subdecode was 1, the AP register is not modified.

Note that current versions of the macro assembler do not support subdecode 1. This is because there is no mnemonic or modifier to designate subdecode 1. So, the instruction `move A[1],ACC` will always generate the op code `0x990A` and never `0x991A`.

When module 10 is specified as a destination and the source is either an immediate value or any module other than module 10, the source is routed through the ALU; the destination is taken from the ALU's output, not directly from the source. This is how arithmetic and logical instructions are implemented.

Note that there is no restriction on what may serve as a source register. It can be an immediate value, an indirect memory location, or even a value on the stack or a peripheral register.

When both the source and destination designate is module 10, it is either an accumulator-only instruction or a bit manipulation involving the carry bit. In all cases, both the source and destination subdecodes are used to designate the operation.

Destination subdecode 0 is the home of the accumulator-only instructions, including complement, negate, and all shift, rotate, and exchange instructions. Destination subdecodes 1, 2, 3, 6, and 7 involve bitwise loads and operations that use the carry bit. Finally, destination subdecode 5 has the carry-only operations: load 0 and 1 and complement.

Sixteen registers known as "accumulators" form a general-purpose register array.

Note that one source subdecode of destination subdecode 5 is the designated NOP instruction. While any operation that both has no side effects and addresses a vacant register location will serve as an NOP, a `MOVE M10[5],M10[3]` is specifically guaranteed to perform no operation in current or future MAXQ devices. This is the op code that is generated (`0xDA3A`) in all current assemblers for the NOP mnemonic.

Module 12—Instruction Pointer

Module 12 is unique because it contains a number of conditional load operations. If module 12 is used as a source module, the IP is simply copied to the destination designator. But if module 12 is the destination, no operation is performed unless the specified condition is met.

What sets the MAXQ architecture apart from other, more conventional CPUs is the transfer-triggered nature of the MAXQ core.

Module 12 is also unique because when loaded from an 8-bit immediate source, the source value is interpreted as a signed integer and is added to the previous preincremented contents of the instruction pointer. This addition facilitates relative short jumps, thus offering a significant savings in code size. It also means that any short or long jump instruction can be conditional.

This module supports only simple load and store of the Instruction Pointer register (IP). The CALL instruction is considered a stack instruction that also loads the IP, rather than as an IP instruction that pushes to the stack. Consequently, the transfer for a CALL instruction is in the stack pointer module (module 13). Also, there is no explicit RET instruction; this is cast as a POP IP.

Module 13—Stack Pointer

Module 13 contains not only the stack-pointer-related registers but also the loop counters and interrupt vector. Note that several of the subdecodes are valid only as a destination. Subdecode 8 is valid only as a source and facilitates the RETI instruction by popping a value from the stack and clearing the interrupt in-service bit.

Subdecodes 3, 4, and 5 serve as proxies for the IP register. Subdecode 3 loads the instruction pointer after the incremented instruction pointer is pushed to the stack, thus implementing a traditional CALL instruction. Subdecodes 4 and 5 load a predecremented version of the designated loop counter back to the loop counter, and also load the instruction pointer with the source operand if the predecremented loop counter was nonzero. The source to load into this destination subdecode can be anything; the instruction `DJNZ LC[0], A[1]` is perfectly valid. In this case, the instruction would decrement LC[0] and jump to the address in A[1] if the result of the decrement operation is nonzero.

While all MAXQ processors include a memory management unit (MMU) that allows any memory segment to be treated as code or data, the code and data buses are separate.

Module 14—GR, BP, and DPC

Module 14 contains the DPC register, GR register, and all registers associated with the base pointer and the offset register.

The Data Pointer Control register (DPC) describes how the data pointers behave. In particular, it contains a bit for each data pointer that defines whether that pointer is operating in word mode or byte mode. It also contains a field that defines which pointer is the current source pointer. This is necessary because the source is accessed when the source pointer is loaded and there is only one bus for operand data.

The GR register is convenient when byte access is required for 16-bit data. Once GR is loaded with 16-bit data, the low- and high-order bytes can be retrieved through the GRL and GRH registers, respectively. The GRS register contains the byte-swapped version of GR; the GRXL register is the same as the GRL register, except that the high byte is the sign extension of the low-order byte.

The Base Pointer register (BP) is one of three data-memory pointer registers in the MAXQ architecture, and the only one to support an offset register. BP typically points to the base of a data structure, and the 8-bit unsigned offset register points to a data element within the structure. Note that the increment and decrement versions of this register modify only the offset register and never the base register.

Every MAXQ processor provides a prefix register to supply these additional register address bits, and to provide the upper byte of a word-wide move.

Module 15—Data Pointers

Module 15 contains two of the three data pointers in the MAXQ architecture. Depending on the subdecode, access to this module will perform a direct or indirect load or store, and may increment or decrement the data pointer following an indirect access. These register subdecodes can be used as either source or destination registers.

Module 7—Boolean Variable Manipulation

The Boolean Variable Manipulation (BVM) module (Module 7) allows bit extraction and bit setting/clearing for many registers in a typical MAXQ processor (**Figure 7**). Note that not all modules have a connection to the BVM machine. Typically, only the peripheral modules connect to the BVM; system registers do not. Consequently, moving data between the BVM and a system register likely causes unpredictable consequences.

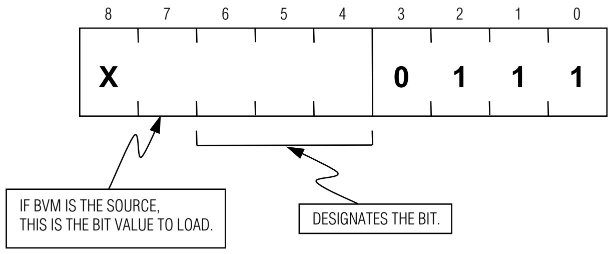


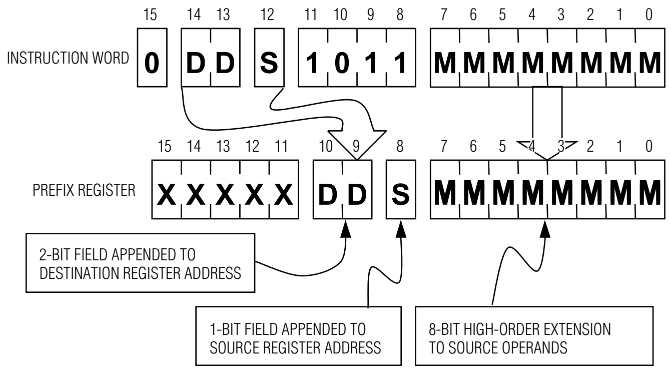
Figure 7. Subdecodes of module 7 designate the bit to extract or replace, if a source designator, the immediate bit value.

As a destination designator, the BVM serves as a proxy for the carry bit. One bit of the source is extracted and copied to the carry bit. If the BVM is a source designator, the value given in bit 3 of the subdecode (bit 7 of the complete source designator) is copied to the specified bit of the destination.

Note that the BVM only works with bits 0 to 7 of the peripheral register. This is acceptable for most peripheral registers because many registers (I/O ports in particular) are only 8 bits in length. But when accessing 16-bit peripheral registers, only the low-order 8 bits are available.

Module 11—Prefix

The prefix module is a unique feature of the MAXQ architecture that addresses a limitation of all 16-bit microcontrollers. With 16-bit registers, immediate load instructions require a 16-bit operand, meaning that an effective immediate load instruction requires more than 16 bits.



INSTRUCTION WORD
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 D D S 1 0 1 1 M M M M M M M M
PREFIX REGISTER
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X D D S M M M M M M M M
2-BIT FIELD APPENDED TO DESTINATION REGISTER ADDRESS
1-BIT FIELD APPENDED TO SOURCE REGISTER ADDRESS
8-BIT HIGH-ORDER EXTENSION TO SOURCE OPERANDS

Figure 8. When the prefix register is a destination, the 8-bit immediate source provides the high-order byte for 16-bit immediate operands; the destination subdecode provides additional bits to permit addressing of all 32 registers in each module for both source and destination operands.

There are several solutions to this limitation, including variable length instructions and registers that allow independent access to the low and high bytes (the MAXQ GR register is an example of this). None of the solutions is ideal because they complicate decode logic or involve new registers (**Figure 8**).

The prefix mechanism improves on this process in two ways. First, by prefixing only those instructions that specifically require additional bits, the mechanism saves code space and execution time. And second, by providing additional bits not only for immediate operands but also for register designators, the mechanism preserves the overall architecture while extending the size of the register space.

Remember that while there are 32 registers per register module, only four bits designate a source register and only three bits designate a destination register. The prefix mechanism provides these additional bits.

The prefix mechanism is unique in several ways. Firstly, certain bits in the destination part of the instruction are used as immediate source bits for accessing register subdecodes above 15 for source addresses and above seven for destination addresses. In this way, a single prefix instruction can provide access from any register or immediate value to any register subdecode.

Secondly, the prefix register is unique because any value loaded into it survives for one clock cycle only. After that, the register is automatically cleared to zero. This means that any move to the prefix register must be the instruction immediately before the instruction to be modified by the prefix register. It also means that the prefix instruction is noninterruptible. If an interrupt occurred following a prefix operation, the prefix information would be lost when the interrupt returned to the main function.

As shown in **Figure 9**, bits from the prefix register go to the source designator, destination designator, and immediate value. So, while most instructions execute in a single cycle, two cycles are required for instructions that: address a destination register subdecode greater than 7; address a source register subdecode greater than 15; or load an immediate value greater than 255.

To illustrate this process, consider the instruction `move A[0], #010h`. As this is moving an immediate value to module 9 register 0, the assembler would create the following op code: `0910`. But if the instruction were `move A[10],#0320h`, the assembler would have to automatically insert a prefix instruction: `2B03 2920`.

Without the prefix instruction, the op code `2920` would translate to `move A[2],#020h`. But the prefix adds a bit to the destination specifier and additional bits to the immediate value, allowing the processor to load any value to any register subdecode and never taking more than two cycles.

Conclusion

Even though the MAXQ core is small and apparently simple, its transfer-triggered architecture gives it a significant edge in speed and flexibility. Because peripherals are addressed directly through the register interface, the speed of data transfer through the embedded peripherals can be impressive. Overall, the MAXQ core in any of its forms is an excellent choice for a wide range of microcontroller applications.

For an expanded version of this article, go to:
www.maxim-ic.com/AN3960.

MAXQ is a registered trademark of Maxim Integrated Products, Inc.

Even though the MAXQ core is small and apparently simple, its transfer-triggered architecture gives it a significant edge in speed and flexibility.

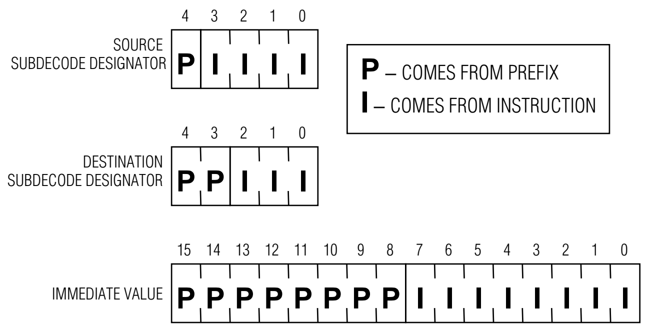

SOURCE SUBDECODE DESIGNATOR
4 3 2 1 0
P I I I I
DESTINATION SUBDECODE DESIGNATOR
4 3 2 1 0
P P I I I
IMMEDIATE VALUE
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
P P P P P P P P I I I I I I I I
P – COMES FROM PREFIX
I – COMES FROM INSTRUCTION

Figure 9. *The prefix register provides the additional bits needed for 16-bit immediate operands and to address all 32 registers in each module as both source and destination.*

SD Media Format Expands the MAXQ2000's Space for Nonvolatile Data Storage

The MAXQ2000 stores nonvolatile data in flash memory and has 32kWords (64kB) flash capacity that is shared with user code space.

The low-power, low-noise MAXQ2000 microcontroller is suitable for a variety of applications. The MAXQ2000 stores nonvolatile data in flash memory and has 32kWords (64kB) flash capacity that is shared with user code space. But what if your application requires more nonvolatile storage? This article demonstrates how to use the Secure Digital (SD) media format to expand the MAXQ2000's nonvolatile data storage.

Design Considerations for External Storage

The first design considerations for your application are supply voltage and current requirements. In a typical MAXQ2000 application, a dual linear regulator is employed to run the processor core voltage (V_{DD}) at the lowest voltage necessary for the selected design clock rate. The MAXQ2000 V_{DD} supply can be as low as 1.8V. The I/O pins on the MAXQ2000 are supplied by V_{DDIO}, which has an allowable lower range of V_{DD} and an upper limit of 3.6V. Acceptable current draw for external storage is dictated by the current rating of the power supply and, in the case of a battery-powered device, the capacity of the battery system.

Secondly, the number of MAXQ2000 I/O lines used to connect the external storage must be kept to a minimum, while still providing sufficient bandwidth for the intended application. The Atmel AT29LV512 flash chip, for example, requires 15 address lines, eight data lines, and three control lines when interfaced with a host microcontroller. Because the MAXQ2000 does not have an external address/data bus, software would need to control the bus transactions in that example. For some applications, that method is not an efficient use of the MAXQ2000's I/O pins.

SPI™ and I^2C-based external flash devices, however, require only three or four interface pins. The MAXQ2000 has a hardware SPI module, while I^2C must be implemented on the MAXQ2000 by the user in software (i.e., "bit banging"). This integrated capability means that the SPI-interface is the prime avenue for access to external nonvolatile storage.

SD Memory Card Format

The SD media format is a nonvolatile external memory that satisfies the above considerations for many applications. The SD format is the successor to the "MultiMedia Card" format, or MMC. SD card memories typically operate from 3.3V supplies with modest current requirements. SD card capacities range from a few megabytes up to a maximum capacity of 4GB. This wide range of available sizes provides ample external storage for many applications.

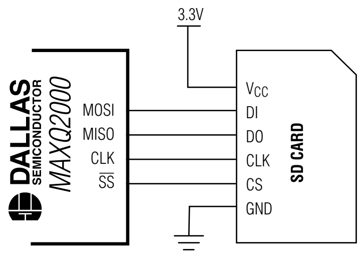


Figure 1. The MAXQ2000 easily interfaces to an SD memory card.

At first glance, SD may not appear to interface easily with the MAXQ2000 due to the former's proprietary shared bus. However, SD inherited MMC's secondary bus format, SPI. Thus, interfacing is simple because the MAXQ2000 contains hardware support for SPI.

The schematic in **Figure 1** shows a typical application circuit. The SD card requires full-duplex, 8-bit SPI operation. Data is clocked into the card's DI pin from the MAXQ2000's MOSI pin, and out of the card's DO line into the MISO pin of the MAXQ2000. Data is clocked simultaneously in and out of the card on the rising edge of the CLK line. Eight extra clocks must be provided at the end of each transaction to permit the SD card to complete any outstanding operations. The input data during these extra clocks must be all ones. Clock rate must be limited to a maximum of 400kHz during the identification phase, but can be increased up to 25MHz once the SD card has been identified.

MAXQ2000 SPI Module

The MAXQ2000 contains a hardware SPI module that is easily configured for the SD card interface. To configure the clock polarity and data length, the SPICF register is programmed to all zeros. This configures the SPI module to latch data on rising clock edges and sets the data length to eight bits. For this application, the MAXQ2000's system clock frequency is 16MHz. In that case, the SPICK register is programmed to 0x28, which results in an SPI clock of approximately 380kHz. The SPI master mode must be enabled by setting the two lower bits of the SPICN register.

The MAXQ2000 contains a hardware SPI module that is easily configured for the SD card interface.

SD SPI Data Format

The SD card's SPI protocol is similar to its SD bus protocol. Instead of receiving valid data from the SD card's DO pin at every clock edge, a card with no data to send will hold the DO pin at an idle state of all ones. When the card has data to send back to the host, specialized tokens with a zero start bit are sent before the data. All data transmitted from the SD card is sent immediately after these tokens and is of fixed length. As the receiver has a prior knowledge of the number of bytes to expect, no length bytes are contained in the response. Additionally, as the idle state cannot occur until after the start token and data have been sent, all data bytes are transmitted unaltered and unprefixed. Tokens, as with all other traffic on the bus, are aligned on the 8-bit boundaries of the SPI transaction. Commands and data from the host to the card follow a similar format, with all ones denoting an idle bus. All transactions except status tokens are protected by a cyclic redundancy check (CRC) code appended to the end of data. Two CRC algorithms are provided: CRC-7 for short blocks of data, and CRC-16 for longer blocks of data. The CRC is an optional part of the SD SPI interface, but should be used to guarantee data integrity unless application constraints prevent its use.

SD Command Format

Commands are issued to the card in a 6-byte format (**Figure 2**). The first byte of a command can be constructed by ORing the 6-bit command code with hex 0x40. The next four bytes provide a single 32-bit argument, if required by the command; the final byte contains the CRC-7 checksum over bytes 1 through 5. **Table 1** lists important SD commands.

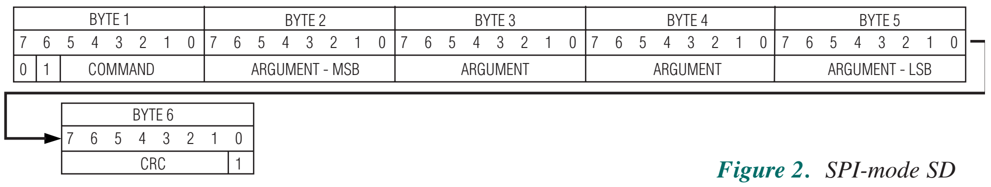


Figure 2. SPI-mode SD commands are issued to the card in a 6-byte format.

Initializing the SD Card in SPI Mode

At power-up, the SD card defaults to the proprietary SD bus protocol. To switch the card to SPI mode, the host issues command 0 (GO_IDLE_STATE). The SD card detects SPI mode selection, because the card select (CS) pin is held low for this and all other SPI commands. The card responds with response format R1 (**Figure 3**). The idle state bit is set high to signify that the card has entered idle state. To maintain compatibility with MMC cards, the SPI clock rate must not exceed 400kHz at this stage.

Table 1. Selected SD Memory Card Commands

Command	Mnemonic	Argument	Reply	Description
0 (0x00)	GO_IDLE_STATE	<none>	R1	Resets the SD card.
9 (0x09)	SEND_CSD	<none>	R1	Sends card-specific data.
10 (0x0a)	SEND_CID	<none>	R1	Sends card identification.
17 (0x11)	READ_SINGLE_BLOCK	address	R1	Reads a block at byte address.
24 (0x18)	WRITE_BLOCK	address	R1	Writes a block at byte address.
55 (0x37)	APP_CMD	<none>	R1	Prefix for application command
59 (0x3b)	CRC_ON_OFF	Only Bit 0	R1	Argument sets CRC on (1) or off (0).
41 (0x29)	SEND_OP_COND	<none>	R1	Starts card initialization.

At first glance, SD may not appear to easily interface with the MAXQ2000 due to the proprietary shared bus. However, SD inherited MMC's secondary bus format, SPI.

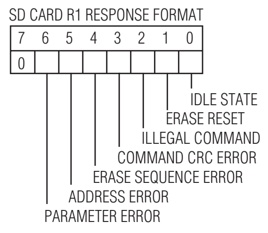
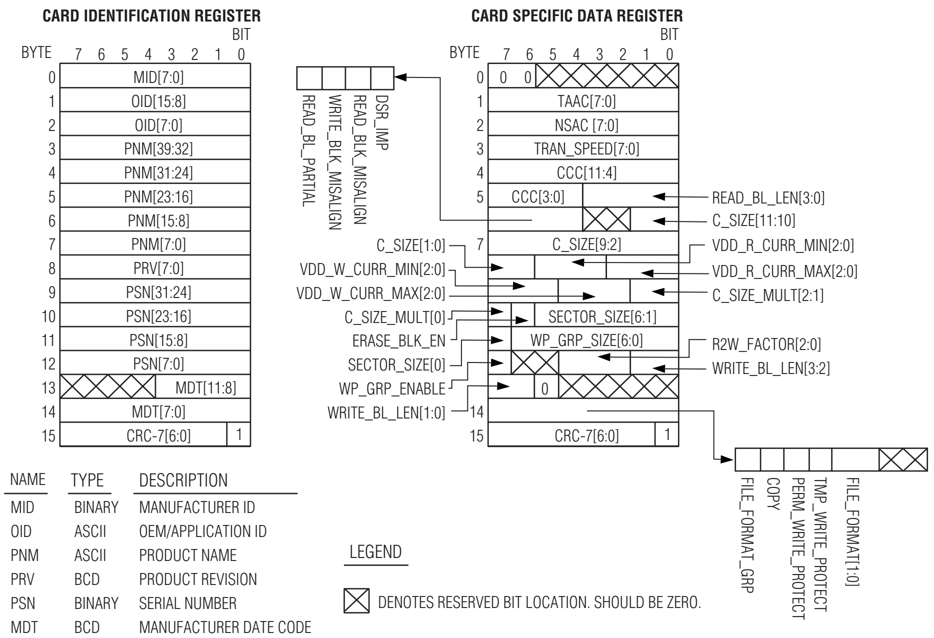


Figure 3. Response format R1 signals the success or failure of the issued command.

Now that the SD card is in SPI mode, the SD specification requires that the host issue an initialization command before any other requests can be processed. To differentiate between MMC and SD cards, SD cards implement an alternative initialization command, to which MMC cards do not respond. Sending command 55 (APP_CMD) followed by application command 41 (SEND_OP_COND) to the card completes this important step. MMC cards do not respond to command 55, which can be used to reject MMC cards as invalid media. This command sequence is repeated until all bits in the R1 response from the card are zero (i.e., the IDLE bit goes low).

```
while(status && (errors < retries)) {
  printf("-> Send CMD55_APP_CMD\r\n");
  xmitcmd(CMD55_APP_CMD, arg);
  if (waitForR1(&rxdata, 0) < 0) {
    /* If this is a MultiMediaCard (not SD), it will not respond here */
    printf("ERROR: Timeout! Perhaps this is a MMC card?\r\n");
    return TR_TIMEOUT;
  }
  check_r1(rxdata, R1_IDLE);

  printf("-> Send ACMD41_SEND_OP_COND\r\n");
  xmitcmd(ACMD41_SEND_OP_COND, arg);
  if (waitForR1(&rxdata, 0) < 0) {
    printf("ERROR: Timeout on ACMD41_SEND_OP_COND\r\n");
    return TR_TIMEOUT;
  }
  status = rxdata & R1_IDLE;
  if (status) {
    /* Pause here for a bit to let the card start up */
    for (i = 0; i < 10000; i++); /* busy loop */
  }
}
```



NAME	TYPE	DESCRIPTION
MID	BINARY	MANUFACTURER ID
OID	ASCII	OEM/APPLICATION ID
PNM	ASCII	PRODUCT NAME
PRV	BCD	PRODUCT REVISION
PSN	BINARY	SERIAL NUMBER
MDT	BCD	MANUFACTURER DATE CODE
CRC-7	BINARY	CRC-7 CHECKSUM

Figure 4. The CSD and CID registers provide information about the SD card.

The SD card contains several important registers that provide information about the SD card. The most important register is the Card Specific Data register (CSD). For our sample application we are interested in the block size and total size of the memory. We must also pay attention to the Card Identification Data register (CID), as it contains details about the cards' manufacturer and serial number. **Figure 4** shows the layout of the CSD and CID registers.

Examining the SD Card Responses

To read card registers or blocks from the card, we must first understand how the card responds to our inquiries. In SPI mode, the SD card replies to the commands SEND_CSD (9), SEND_CID (10), and READ_SINGLE_BLOCK (17) with a R1 format reply. A start token, the requested data, and finally a CRC-16 checksum over the data follow. We must not assume that the R1 reply and the data start token occur immediately one after the other, as the bus can go to the idle state for some time between these two events. **Figure 5** details the data response.

Reading the CSD and CID Register Meta-Data

The SEND_CSD and SEND_CID commands send back register contents used to determine the SD card parameters. These commands return a fixed number of bytes, which corresponds to the size of the CSD or CID register, respectively. The argument contained within the command bytes is ignored by the SD card for these SEND commands.

Reading a Block of Data from the SD Card

Reading a block of data from the SD card is quite simple. The host issues the READ_SINGLE_BLOCK command with a starting byte address as the argument. This address must be aligned with the beginning of a block on the media. The SD card then evaluates this byte address and responds back with an R1 command reply. An out-of-range address is indicated in the command reply.

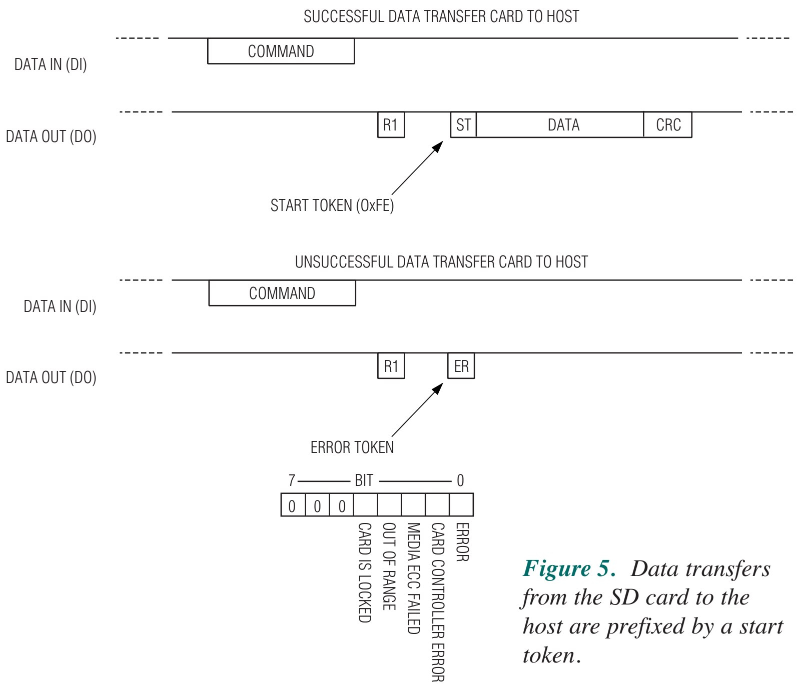


Figure 5. Data transfers from the SD card to the host are prefixed by a start token.

If the read is completed from the SD media without error, a start data token is sent followed by a fixed number of data bytes and two bytes for the CRC-16 checksum. The start data token is not sent if the SD card encounters a hardware failure or media read error. Rather, an error token is sent and the data transfer is aborted.

Writing a Block of Data to the SD Card

Writing a block of data is similar to reading, as the host must supply a byte address that is aligned with the SD card block boundaries. The write block size must equal READ_BL_LEN, which is typically 512 bytes. A write is initiated by issuing the WRITE_BLOCK (24) command, to which the SD card responds with the R1 command response format. If the command response indicates that the write can proceed, the host transmits the data start token followed by a fixed number of data bytes, and ends with a CRC-16 checksum of the sent data. The SD card returns a data response token indicating the acceptance or rejection of the data to be written.

Clock rate must be limited to a maximum of 400kHz during the identification phase, but can be increased up to 25MHz once the SD card has been identified.

If the data is accepted, the SD card holds the DO line low continuously while the card is busy. The host is not obligated to keep the card select low during the busy period, and the SD card releases the DO line if CS is deasserted. This process is useful when more than one device is connected to the SPI bus. The host can wait for the SD card to release the busy indication, or check the card by periodically asserting the chip select. If the card is still busy, it will pull the DO line low to indicate this state. Otherwise, the card returns the DO line to the idle state (see **Figure 6**).

SPI Command and Data Error Detection

The CRC-7 and CRC-16 checksums can be used to detect errors in the communication between the host and SD card. Error detection allows for robust error recovery in the event of physically induced errors such as contact bounce during insertion and removal or nonideal contact-mating situations inherent with detachable media. The use of checksums, by issuing the CRC_ON_OFF (59) command with the lowest bit set in the argument, is highly recommended.

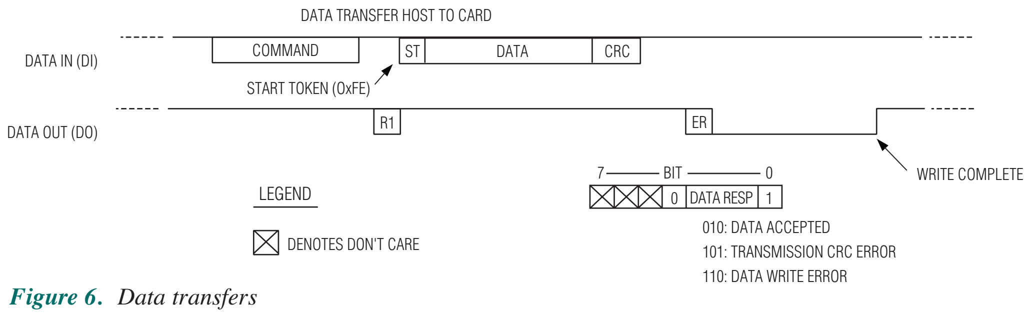


Figure 6. *Data transfers from the host to the SD card involve a more complex handshake.*

```
/* Enable CRC to protect against error */

printf("-> Send CMD59_CRC_ON_OFF\r\n");
arg[3] = 0x01; /* LSB set to 1 enables CRC verification */
xmitcmd(CMD59_CRC_ON_OFF, arg);
CLEAR_ARGS(arg);
if (waitForR1(&rxdata, 0) < 0) {
  printf("ERROR: Timeout on CMD59_CRC_ON_OFF\r\n");
  return -1;
}
if (rxdata != 0x00) {
  printf("WARNING: R1 status 0x%02x, expecting 0x00\r\n", rxdata);
}
```

By using the hardware SPI support provided by the MAXQ2000 microcontroller, SD media cards can be accessed with very little overhead.

Conclusion

The SD media card format represents a compact, low-power nonvolatile memory solution for embedded systems. By using the hardware SPI support provided by the MAXQ2000 microcontroller, SD media cards can be accessed with very little overhead. The reference software provided by Maxim at www.maxim-ic.com/MAXQ2000_SD demonstrates a minimal implementation, which includes the essential operations required to read blocks from and write blocks to an SD card.

References

Further information on the SD media format can be obtained from the Secure Digital Association at www.sdcard.org. The SD media breakout board used in this project can be ordered from www.sparkfun.com/commerce/product_info.php?products_id=204.

For an expanded version of this article, go to: www.maxim-ic.com/AN3969.

SPI is a trademark of Motorola, Inc.

Using Rowley CrossWorks and the MAXQ3100* Evaluation Kit to Create a Temperature Logging Application

In this article, we use Rowley Associates' CrossWorks and the MAXQ3100 evaluation kit (EV kit) to create a simple temperature logger. Because the MAXQ3100 integrates a temperature sensor, no additional components are required.

CrossWorks for MAXQ provides a full-featured development environment for C applications on the MAXQ platform.

CrossWorks for MAXQ provides a full-featured development environment for C applications on the MAXQ platform. CrossWorks includes: an ANSI C compiler; an assembler and linker optimized for the MAXQ architecture; and a debugger designed to interface with both the hardware-based debug engine and the JTAG interface found on most MAXQ microcontrollers. All these elements are integrated into a project-based development environment that has a comprehensive help system, built-in support, and examples for MAXQ microcontrollers such as the MAXQ2000 and the MAXQ3100. Note that the Rowley CrossWorks for MAXQ is currently only available for Windows® platforms. Download a 30-day evaluation license at: www.rowley.co.uk/maxq/index.htm.

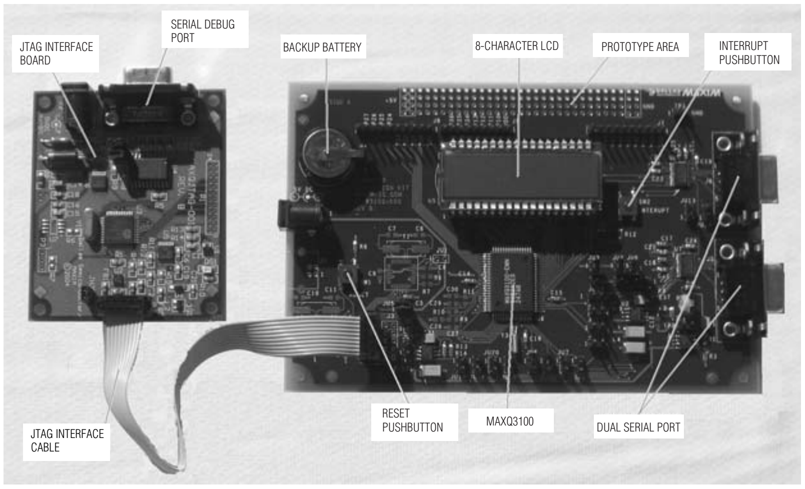
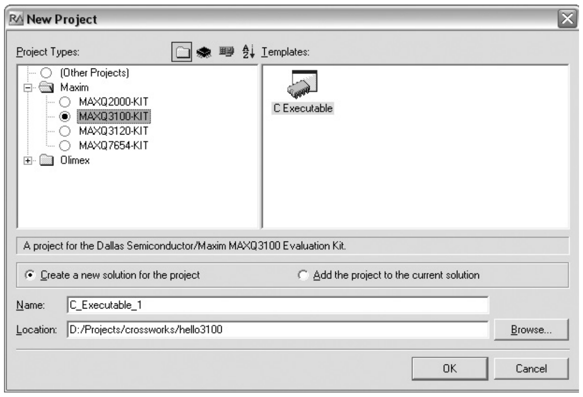


Figure 1. The MAXQ3100-KIT and serial-to-JTAG boards combine to form a complete system for application development.

The MAXQ3100 EV kit includes components that demonstrate the MAXQ3100's key features (available at: www.maxim-ic.com/MAXQ3100) and allows software development to begin immediately while custom hardware design is underway. The kit includes the following features:

- MAXQ3100 microcontroller in an 80-pin MQFP package with a 32kHz crystal
- Serial-to-JTAG interface board for programming and debugging from a PC host
- 3.6V linear regulator to run the MAXQ3100 directly from the JTAG interface
- Manually adjustable linear regulator that outputs between 1.8V and 3.6V
- 8-character (14 segments per character) LCD 1/4-duty display driven directly by the MAXQ3100
- Two level-shifted serial ports with DB9 connectors
- Reset and external interrupt pushbuttons
- Prototyping area with access to MAXQ3100 port pins, comparator inputs, and power supplies

Combined with the serial-to-JTAG interface board (**Figure 1**), this system allows full access to the MAXQ3100's in-system bootloader and debugging features.



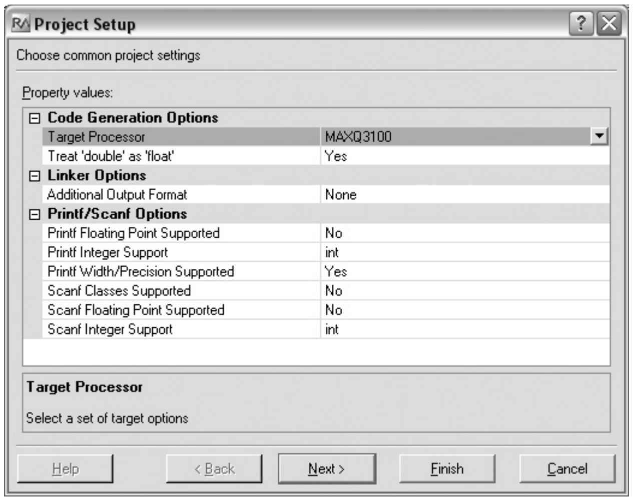
A project for the Dallas Semiconductor/Maxim MAXQ3100 Evaluation Kit.

[✓ Create a new solution for the project] Add the project to the current solution

Name: C_Executable_1

Location: D:/Projects/crossworks/hello3100

Figure 2. The first step in creating a new project is to select the target device and executable type.


Project Setup

Choose common project settings

Property values:

Code Generation Options	
Target Processor	MAXQ3100
Treat 'double' as 'float'	Yes
Linker Options	
Additional Output Format	None
Printf/Scanf Options	
Printf Floating Point Supported	No
Printf Integer Support	int
Printf Width/Precision Supported	Yes
Scanf Classes Supported	No
Scanf Floating Point Supported	No
Scanf Integer Support	int

Target Processor

Select a set of target options

Figure 3. The MAXQ3100 is the target processor for this project.

Creating a Project in Rowley Crossworks

Once you complete product activation (see “Support: Evaluating CrossWorks” on the Rowley website), start CrossWorks. To create a new project, select **File ➔ New ➔ New Project** from the menu. In the New Project dialog (**Figure 2**) select “MAXQ3100 Kit” and “C Executable,” and enter a name and location for the new project. In the Project Setup dialog that follows (**Figure 3**), verify that the MAXQ3100 is selected for the “Target Processor” option. The remaining settings can remain at their default values. Click **Finish** to generate the new project.

Application Overview

This MAXQ3100 demonstration example showcases four key features of the processor: the ambient temperature sensor, the LCD controller, the real-time clock (RTC), and the UART interface. With these peripherals, we can create an application that measures temperature levels, displays the time and temperature on an LCD, and transmits data back to a PC over the UART interface. Complete code for this example is available online at: www.maxim-ic.com/MAXQ3100_ temp_logger.

Because the program memory for the MAXQ3100 is implemented using word-rewriteable EEPROM, the application can use the remaining unused program memory to store a time and temperature log. The utility ROM on the MAXQ3100 provides in-application programming routines that allow portions of the EEPROM to be rewritten and erased under software control.

Measuring the Temperature

The MAXQ3100’s temperature sensor measures ambient temperature to a resolution down to 0.0625°C. The width of the sample value returned is selectable from 10 bits (0.5°C resolution) to 13 bits (0.0625°C resolution).

```
void
convertTemp(void)
{
   IC = 0;         // Disable interrupts

   TPCFG = 0x06;    // Set resolution to 13 bits
   TPCFG = 0x07;    // Start conversion
   while ((TPCFG & 0x01) == 0x01) {}    // Wait for conversion to complete
   g_lastTemp = TEMPR;                   // Store temperature value

   IC = 1;         // Enable interrupts
}
```

Once the temperature is measured, it can be converted to a floating point value in either degrees Celsius or degrees Fahrenheit.

Displaying the Temperature Value

The MAXQ3100 EV kit includes an 8-character LCD, configured as shown in **Figure 4**, which provides more than enough display space to show the current temperature reading to two decimal places.

Because the LCD characters are 14 segment, special characters such as plus, minus, and the degree sign are easy to display along with standard 0 to 9 and alphabetic characters. The first step is to initialize the LCD controller.

Once the LCD is initialized, we can add code to set any of the eight digits to any of the supported character values. The LCD segments are mapped into the display registers LCD0 to LCD15 (**Table 1**), so showing characters on the display is simply a matter of writing values into the proper register locations.

Two registers are used to store the segment values for each character, which includes the decimal point and indicator annunciators. These registers can be modified while the LCD controller is running.

A similar routine (`displayDP`) is provided to set and clear decimal points on the LCD. With these routines in place, the measured temperature value can be converted into the proper display digits. The application includes a fixed delay between temperature conversions, which means that the display updates several times per second.

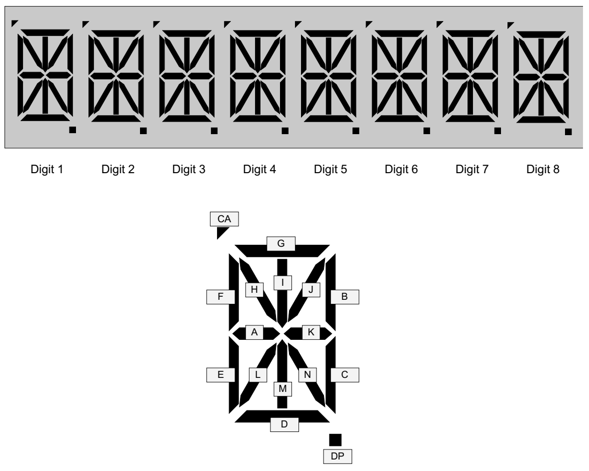


Figure 4. The MAXQ3100 EV kit provides an 8-character, 14-segment LCD for application use.

Starting and Setting the Clock

The recorded temperature samples become more useful if the application provides a way to connect (i.e., to pair) the samples with the time when they were measured. Because the MAXQ3100's RTC is based on a 32kHz crystal oscillator clock, the RTC is the natural choice for a clock value to timestamp each temperature sample. The application displays the clock on the LCD display when requested, and records a time-and-temperature sample pair once a minute in the EEPROM.

Once the RTC is started, it can be used to generate a periodic alarm that triggers an interrupt at a programmable interval. Our application uses this interval alarm to generate an interrupt precisely once a second. This interrupt is then used to increment a global seconds counter in the application.

In CrossWorks, any function can be designated as an interrupt handler by adding the `__interrupt` keyword to the function definition. Then, the built-in `setIV()` function is used to load the programmable interrupt vector register (IV) at the entry point of the interrupt handler function. Because all interrupts on the MAXQ3100 vector to the address contained in the IV register, we only need to define one interrupt handler function. This application uses only one interrupt, but if it needed to use others, additional interrupt flags could be checked inside the interrupt handler function to determine which interrupt triggered the function call.

Table 1. LCD Display Memory Map (1/4 Duty)

Register	Bit 7 COM3	Bit 6 COM2	Bit 5 COM1	Bit 4 COM0	Bit 3 COM3	Bit 2 COM2	Bit 1 COM1	Bit 0 COM0
LCD0	1D	1E	1F	1CA	1DP	1C	1B	1G
LCD1	1N	1K	1J	1I	1M	1L	1A	1H
LCD2	2D	2E	2F	2CA	2DP	2C	2B	2G
LCD3	2N	2K	2J	2I	2M	2L	2A	2H
LCD4	3D	3E	3F	3CA	3DP	3C	3B	3G
LCD5	3N	3K	3J	3I	3M	3L	3A	3H
LCD6	4D	4E	4F	4CA	4DP	4C	4B	4G
LCD7	4N	4K	4J	4I	4M	4L	4A	4H
LCD8	5D	5E	5F	5CA	5DP	5C	5B	5G
LCD9	5N	5K	5J	5I	5M	5L	5A	5H
LCD10	6D	6E	6F	6CA	6DP	6C	6B	6G
LCD11	6N	6K	6J	6I	6M	6L	6A	6H
LCD12	7D	7E	7F	7CA	7DP	7C	7B	7G
LCD13	7N	7K	7J	7I	7M	7L	7A	7H
LCD14	8D	8E	8F	8CA	8DP	8C	8B	8G
LCD15	8N	8K	8J	8I	8M	8L	8A	8H

In CrossWorks, any function can be designated as an interrupt handler by adding the `__interrupt` keyword to the function definition.

Once the RTC is started, it can be used to generate a periodic alarm that triggers an interrupt at a programmable interval.

The interrupt handler function increments the seconds (0 to 59), minutes (0 to 59), and hours (1 to 12) counters appropriately. Every time the seconds counter rolls over (once a minute), the function uses `asm_writeLog` to record the current time and last measured temperature value to the nonvolatile EEPROM log. Also, the leftmost caret on the LCD display (left of digit 7) is toggled each time the seconds interrupt is triggered, thus providing a heartbeat indicator. This operation occurs even when the time is not currently displayed.

The starting point of the log in EEPROM is fixed using a constant (LOG_START), which must be set to an area beyond the end of the compiled application. For example, the current version of the application occupies 8972 code bytes, which means that the loaded application runs from word address 0000h to 1185h in program memory. Setting LOG_START to 1800h puts the log well beyond the end of the application and reserves (2000h – 1800h) = 2048 words of EEPROM for logging use. As two words are used for each entry, the application can log 1048 time/temperature value pairs. At one log entry per minute, this results in a rolling log of about 17 hours in duration. As each new log entry is written, the oldest previous entry is also deleted.

Normally, CrossWorks handles all the potential issues introduced by an interrupt handler routine. When the interrupt triggers, the working registers and context of the currently executing routine are saved. This information is restored after the interrupt handler exits. Users should, however, pay attention to the following circumstances:

- If the interrupt handler routine accesses global variables used elsewhere in the application, other sections of code that access these same variables should do two things: disable interrupts before reading or writing the variables, and enable interrupts once the operation has completed.
- Take care when implementing code that performs operations directly on MAXQ3100 low-level registers. CrossWorks does not automatically save or restore the states of these registers when an interrupt is triggered. For example, if an interrupt occurs during a `printf()` to a serial port, the interrupt handler should not contain calls to `printf()` or code that writes directly to the serial-port registers.
- Any interrupt flags that triggered the interrupt call must be cleared before the interrupt handler routine exits. Otherwise, the interrupt immediately triggers again. (In the `incr_seconds()` routine, for example, the alarm subsecond flag RCNT.7 must be cleared.) Interrupt flags are not cleared by hardware automatically.

Interfacing with Assembly Routines

CrossWorks allows native MAXQ assembly routines to be linked into the application along with C code. This allows specialized assembly routines to be included in an application to perform tasks that are difficult to do with compiled C code. In our application, assembly functions are used to interface with MAXQ3100 utility ROM routines that read from locations in program space, and write to or erase locations in the EEPROM.

```
int asm_readLog(int addr);
int asm_writeLog(int addr, int wval);
int asm_erasePage(int addr);
```

When writing assembly routines that will be called from C code, one must first determine how parameters and return values will be passed back and forth. In CrossWorks, up to four integer parameters can be passed into a function by using the accumulator registers A[7] through A[4]. For example, if a function is defined as:

```
int asm_func1(int foo, int bar, int frob, int nitz)
```

then upon entry into the function, A[7] is set to the value of foo, A[6] is set to the value of bar, A[5] is set to frob, and A[4] is set to nitz. (Larger values, such as longwords, are passed in register pairs; if a function has too many input parameters to pass in A[7] to A[4], the parameters are passed on the soft stack. Refer to the CrossWorks help system for more details.) Integer return values from a function are stored in A[7] for passage back to the calling C code.

The `asm_writeLog` routine takes two parameters: a location to which to write in program memory, and a word of data to write there. The assembly code passes these parameters along to an assembly function in the utility ROM called `UROM_eepromWriteWord`. Following this function call, the assembly code must ensure that any registers that may be in use by the C code are restored before exit; in CrossWorks, this set of registers includes accumulator registers A[0] to A[3] and A[8] to A[15]. If any of these registers are modified in the assembly routine, they must be restored before the routine exits. DPC must also be restored to the value 18h as shown, which sets DP[0] to byte mode and DP[1] and BP[Offs] to word mode.

CrossWorks allows native MAXQ assembly routines to be linked into the application along with C code.

```
urom_eepromWriteWord  EQU (0x874E << 1)

   public _asm_writeWord

   code                       ; Code segment
   even                       ; Align to word boundary
_asm_writeWord:
   move     DPC,  #0x1C      ; Set all data pointers to word mode
   move     DP[0], A[7]      ; DP[0] is the address that will be written
   move     A[4], A[0]       ; Save off A[0] (used by C code)
   move     A[0], A[6]       ; A[0] is the word value that will be written
   move     A[5], A[1]       ; Save off A[1] (used by C code)
   move     A[6], A[2]       ; Save off A[2] (used by C code)

   call     urom_eepromWriteWord   ; Writes word, destroys A[0],A[1],A[2]

   move     DPC,  #0x18      ; Reset data pointer modes for C code
   move     A[2], A[6]       ; Restore A[2]
   move     A[1], A[5]       ; Restore A[1]
   move     A[0], A[4]       ; Restore A[0]
   ret
```

Note that the address of the UROM_eepromWriteWord routine is defined for the CrossWorks assembler in terms of a byte address (874Eh shifted left by 1). This differs from the MAXQ assembler, which assumes that all program space addresses are word addresses.

In CrossWorks, up to four integer parameters can be passed into a function using the accumulator registers A[7] through A[4]. Integer return values from a function are stored in A[7] for passage back to the calling C code.

Similar functions are implemented to read recorded values back from the EEPROM (`asm_readWord`) and to erase 32 word pages from the EEPROM (`asm_erasePage`). These functions are implemented in a similar manner, and use the utility ROM functions `UROM_moveDP0` and `UROM_eepromErasePage`.

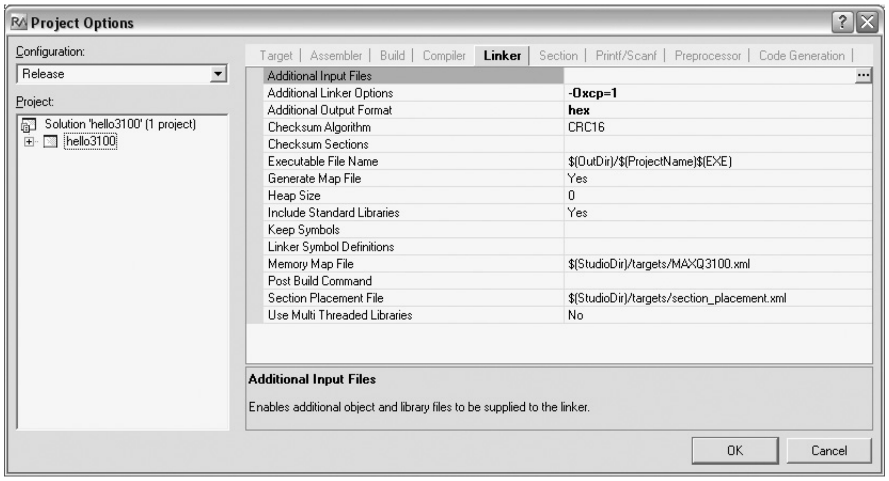
The hardware stack is another issue that arises when assembly routines are in use. Normally, CrossWorks tracks the depth of the hardware stack by using the soft C stack implemented in data memory to store parameters and local variables for function calls. As long as our application is only using C code, the task of ensuring that we do not overflow the hardware stack remains entirely with the compiler.

When an assembly routine is called, however, the compiler has no way of knowing how many stack levels the assembly routine might need. Each CALL, PUSH/POP, or interrupt entry in the assembly code requires one stack word, and utility ROM routines can use stack words as well. In this case, since there are no hardware indicators or interrupts that indicate that a stack overflow occurred, preventing a stack overflow is the user's responsibility.

Once you determine the number of stack levels that the assembly routine will use, you can determine the number the compiler will use. In general, each subroutine call down from `main()` requires one stack level, and if the debugger is active, an additional stack level must be reserved for debug engine calls. The compiler will also, if allowed, use additional stack levels to create subroutines containing repeated code for optimization purposes. To restrict this usage to a specified number of stack levels, select **Project ➔ Properties** from the menu. Then select the **Linker** tab, and enter "-Oxcp=n" in the Additional Linker Options field (**Figure 5**). This instructs the compiler to limit the number of code-factoring optimization passes to n, which means that the maximum number of stack levels used will be:

(Max depth of subroutine calls from `main()`) + (1 if debugger is used) +
(stack levels used in assembly) + n

This dialog also contains a setting to generate a hex output file from CrossWorks (set Additional Output Format = hex), which will generate a version of the compiled application that can be loaded by using MTK or MAX-IDE.


Project Options
Configuration:
Release
Project:
Solution 'hello3100' (1 project)
hello3100
Target | Assembler | Build | Compiler | Linker | Section | Printf/Scanf | Preprocessor | Code Generation
OK
Cancel

Additional Input Files	
Additional Linker Options	**-0xcp=1**
Additional Output Format	**hex**
Checksum Algorithm	CRC16
Checksum Sections	
Executable File Name	$(OutDir)/$(ProjectName)$(EXE)
Generate Map File	Yes
Heap Size	0
Include Standard Libraries	Yes
Keep Symbols	
Linker Symbol Definitions	
Memory Map File	$(StudioDir)/targets/MAXQ3100.xml
Post Build Command	
Section Placement File	$(StudioDir)/targets/section_placement.xml
Use Multi Threaded Libraries	No

Additional Input Files

Enables additional object and library files to be supplied to the linker.

Figure 5. *The maximum number of stack levels used for code optimization can be specified in the Project Options dialog.*

Communicating with a Host System

To download recorded values from the log and to configure the temperature logging system, the application employs one of the two level-shifted serial ports found on the MAXQ3100 EV kit. The DB9 connectors from these ports (serial port 0 and serial port 1 on the MAXQ3100) can be connected directly to the COM port of a PC, thus allowing the user to interface with the temperature-logging demo with the Dumb Terminal mode of MTK or a similar terminal emulator application such as TeraTerm. The serial-port output from the application is in standard 10-bit asynchronous format (1 start bit, 8 data bits, 1 stop bit) and is transmitted at 9600 baud.

CrossWorks provides the standard C `printf()` function as part of its standard library set, so most of the code needed to translate the time and temperature values into ASCII is already done. The only remaining step is to initialize and control the serial port that will be sending the data. The interface is handled by defining the `__putchar` function to output a character over serial port 0. Once this has been done, `printf()` will automatically call the `putchar()` function as needed.

Note that for the log output code to function properly, support for floating point numbers in `printf()`must be explicitly enabled. This support is disabled by default, since enabling it adds a large amount of code (approximately 3500 bytes for this application build). To turn on floating point `printf()` support, go to the **Project Options** dialog, select the **Printf** tab, and set the **Printf Floating Point Supported** field to "Yes."

CrossWorks provides the standard C `printf()` function as part of its standard library set, so most of the code needed to translate the time and temperature values into ASCII has already been done.

The application also allows the user to perform simple configuration and control operations by typing single characters into the terminal emulator on the PC host side. When these characters are received over serial port 0, the application performs the following operations:

- d—Rotates the display mode between Clock, Current Temperature (C), and Temperature (F).
- m—Increments the clock's minutes count by 1.
- h—Increments the clock's hours count by 1.
- F—Outputs all nonzero values currently recorded in the time/temperature log in degrees F.
- C—Outputs all nonzero values currently recorded in the time/temperature log in degrees C.
- Z—Erases all values currently recorded in the log.

Table 2 shows a sample log output in degrees F.

Table 2. Sample Log Output

Time	Temperature (°F)
10:31	77.34
10:32	84.20
10:33	79.25
10:34	75.20
10:35	78.12
10:36	98.37

Conclusion

The MAXQ3100 integrates a temperature sensor, dual UART serial ports, and an LCD controller with enough processing power to run complex C applications. Adding the comprehensive toolset and development environment of Rowley Associates' CrossWorks allows anyone to develop and debug data capture and processing applications quickly with standard ANSI C.

For an expanded version of this article, go to: www.maxim-ic.com/AN3975.

*Future product—contact factory for availability
Windows is a registered trademark of Microsoft Corporation.

Adding the comprehensive toolset and development environment of Rowley Associates' CrossWorks allows anyone to develop and debug data capture and processing applications quickly using standard ANSI C.