DALLAS SEMICONDUCTOR MAXIM

Microcontroller
ENGINEERING REVIEW

Volume 3

Introduction to the MAXQ™ architecture

Microcontroller system designers today have a myriad of choices when it comes to selecting a microcontroller for a project—8-bit, 16-bit, RISC, CISC, or something in between. As a rule, many criteria are considered during the selection process. These can include price, performance, power, code density, development time, and even future migration-path alternatives. To complicate the selection process, tight demands for one criterion generally influence the options in other areas. Factors critical in one application may have little importance in another. Consequently, there is no one microcontroller that is perfect for all projects. But to be successful, a modern microcontroller must excel in many of the areas under consideration.

The MAXQ RISC architecture combines high performance and low power with a variety of complex analog functions.

When world-renowned analog chipmaker, Maxim Integrated Products, joined forces with the industry-leading high-performance microcontroller supplier, Dallas Semiconductor, an opportunity to integrate superior analog functionality with leading-edge microcontrollers was created. One result of this partnership is the MAXQ RISC architecture, a new microcontroller core that combines high performance and low power with a variety of complex analog functions.

When integrating complex analog circuitry with high-performance digital blocks, the operating environment should be kept as quiet and noise-free as possible. However, the clocking and switching that occur in the digital circuits of a microcontroller core inject noise into the sensitive analog section. Therein lies the difficulty facing the mixed-signal designer: to achieve high microcontroller performance, but minimize clock noise that can affect sensitive analog circuits.

The MAXQ architecture reduces noise through intelligent clock management and utilization. This means that 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. Additionally, the MAXQ architecture performs many functions on each clock to maximize its performance. This article provides an overview of the MAXQ architecture and highlights its competitive advantages.

No wasted clock cycles

The MAXQ architecture was designed to achieve a high performance-to-power ratio. The first requisite in generating a high-efficiency machine is to maximize utilization of the clock cycles for user code execution.

The most fundamental way that the MAXQ achieves high utilization is through single-cycle instruction execution. Single-cycle instruction execution benefits the end user by increasing instruction bandwidth that leads to higher performance, and/or reduced power consumption made possible by the ability to reduce clock frequency. All MAXQ instructions execute in a single clock cycle except long jump/long call and certain extended register accesses. While many RISC microcontrollers claim to support single-cycle execution, this often applies to a small subset of instructions or addressing modes. With the MAXQ, single-cycle execution is the norm.

Secondly, the MAXQ architecture achieves increased clock-cycle utilization because it does not require an instruction pipeline (common to many RISC microcontrollers) to achieve single-cycle

Table of Contents

...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.

operation. The MAXQ instruction decode and execution hardware is so simple (and timing so fast) that these operations are moved into the same clock cycle as the program fetch itself, with minimal impact to the maximum operating frequency. To illustrate the benefit of eliminating the instruction pipeline, consider the generic RISC CPU that executes from a pipeline. When a program branch occurs, the CPU uses one or more clock cycles (depending upon pipeline depth) to divert program fetching to the target branch address and discards the instruction(s) already fetched. Clearly, using clock cycles to discard instructions, versus executing them, is wasteful and undesirable as it reduces performance and increases power consumption. While the operation is undesirable to the user, the clocks stolen by the CPU to reload the pipeline are an artifact of the architecture and are unavoidable. The MAXQ architecture distinguishes itself from other 8-bit and 16-bit RISC microcontrollers by offering single-cycle execution without an instruction pipeline (and the wasted clock cycles that accompany it).

The MAXQ instruction word

The MAXQ instruction word is unique because there is only one instruction in the classical sense, the “MOVE” instruction. The source and destination operands for the “MOVE” instruction are the basis for creating instructions and memory accesses, and triggering hardware operations. Dissecting the 16-bit MAXQ instruction word reveals only two components: a 7-bit destination field and an 8-bit source field accompanied by a source format bit. The source format bit, when coded as 0, allows any immediate or literal byte value (i.e., #00h–#FFh) to be supplied as a source operand. Unrestricted support for any immediate byte source within a single instruction word can be very valuable during register initialization routines and when performing ALU operations. The nonliteral source and destination possibilities are subdivided into smaller groups, or modules. **Figure 1** illustrates the 16-bit MAXQ instruction word.

FORMAT	DESTINATION	SOURCE
f	ddd dddd	ssss ssss
1	INDEX MODULE	INDEX MODULE
0	INDEX MODULE	IMMEDIATE BYTE DATA (i.e., 00h–FFh)

Figure 1. The MAXQ instruction word is simple, yet very powerful.

All machine instructions reduce to source and destination operands for a transfer operation. These operands can be used to select physical MAXQ device registers. This type of transfer is the most basic and quite easy to imagine. In the MAXQ machine, however, the source and destination operands are not rigidly associated with physical registers.

The MAXQ architecture uses this same source-to-destination transfer construction when performing indirect memory access. Certain destination and/or source encodings are identified as indirect access portals to physical memories such as the stack, accumulator array, and data memory. These indirect memory access portals use physical pointer registers to define the respective memory address locations for access. As an example, one way that the data memory can be accessed indirectly is using the “@DP[0]” operand. This operand, when used as a source or destination respectively, triggers an indirect read or write access to the data memory location addressed by the Data Pointer 0 (DP[0]) register.

The MAXQ architecture also uses special destination and/or source encodings to trigger underlying hardware operations. This trigger mechanism serves as the basis for creating MAXQ instructions that are implicitly linked to certain resources. For example, math operations (ADD, SUB, ADDC, and SUBB) are implemented as special destination encodings that implicitly target one of the working accumulators, with only the source operand supplied by the user. Conditional jumps implicitly target the instruction pointer (IP) for modification and are implemented as separate destination encodings for each status condition that can be evaluated.

The indirect memory access and underlying hardware-operation triggers are combined whenever possible to create new source/destination operands, which provide dual benefits. The auto-increment/decrement indirect-access mnemonics for the data pointers demonstrate this combination. When reading from data memory with DP[0], the user can optionally increment or decrement the pointer following the read operation using the “@DP[0]++” or “@DP[0]--”source operand, respectively.

Numerous advantages come as a result of the MAXQ instruction word. The instruction word contains modularly grouped source and destination operands, which allow simple and fast instruction-decoding hardware and limit signal switching for those modules not involved in the transfer, thus reducing dynamic power consumption and noise. The instruction word uses its full 16 bits to specify source and destination operands, producing an abundant address space for physical registers, indirect memory access, and hardware-triggered operations. Ultimately, coupling the abundant source/destination address space with minimal restrictions on source-destination combinations gives rise to a highly orthogonal machine.

...coupling the abundant source/destination address space with minimal restrictions on source-destination combinations gives rise to a highly orthogonal machine.

MAXQ system highlights

The MAXQ system not only provides the basic hardware resources and capabilities expected by today's microcontroller users, but it also enhances these resources and adds new features to expand device functionality and utility. While it is not feasible to document all the MAXQ system resources, some are discussed here.

Working accumulators

The MAXQ architecture thus far has been addressed as a single entity. However, two slightly different versions, the MAXQ10 and MAXQ20, will be implemented in the initial MAXQ product family launches in early 2004. The primary difference between the MAXQ10 and MAXQ20 options is the standard width of the working accumulators and supporting arithmetic-logic unit (ALU). The MAXQ10 supports 8-bit (byte-wide) accumulators and ALU operations, while the MAXQ20 supports 16-bit (word-wide) accumulators and ALU operations. The MAXQ devices come equipped with a minimum quantity of eight accumulators and, depending on the application, can have as many as 16 accumulators. In the source/destination transfer map, these accumulators are located in a system register module and are each directly accessible as A[n], where *n* corresponds to their respective index. So, a MAXQ device equipped with 16 accumulators would contain accumulators A[0], A[1]…A[14], and A[15]. Any one of the accumulators can be designated as the active accumulator and indirectly accessed through the *Acc* mnemonic by setting the accumulator pointer register, AP, to its specific index (i.e., Acc = A[AP]). The AP register implements only the number of bits necessary to provide a binary decode into the accumulator array, so four bits are required in MAXQ devices having 16 accumulators. All ALU operations implicitly specify the active accumulator as the destination for the operation being performed. Take, for example, the "ADDC src" instruction. This instruction always performs the addition operation between the active accumulator, the carry flag, and the source (src) operand specified. A wealth of bit manipulation and shift/rotate instructions surround the active accumulator.

Additional hardware is attached to the accumulator pointer to expedite ordered and predictable accesses to the accumulator file. The accumulator-pointer control (APC) register provides bits for resetting AP and for streamlining increment, decrement, and modulo operations on the accumulator-pointer register.

The processor status flag (PSF) register contains five status flags, which have special meaning in relation to the active accumulator status and ALU operations. These are the (C)arry, (Z)ero, (S)ign, (E)qual, and (OV)erflow status flags. Some of these flags can be evaluated for performing conditional jumps and returns. The PSF register also provides two additional general-purpose flags (GF1 and GF0) for user software needs.

The MAXQ exploits a transfer-triggered architecture to achieve the objectives of high bandwidth, high efficiency, and high orthogonality.

Dedicated hardware stack

The MAXQ architecture contains a dedicated hardware stack. The stack depth for any MAXQ device is product dependent. A dedicated hardware stack has two distinct advantages. Firstly, it allows data memory to be preserved for other application uses instead of being consumed by stack, and secondly, it supports fast PUSH/POP operations because a dedicated read/write port exists, which need not be shared with data memory. If the hardware stack depth is insufficient for the context storage needed, then the stack-like operation of the data pointers (pre-increment/decrement for writes, post-increment/decrement for reads) is ideal for creating software stacks in data memory.

The MAXQ architecture distinguishes itself from other 8-bit and 16-bit RISC microcontrollers by offering single-cycle execution without an instruction pipeline (and the wasted clock cycles that accompany it).

Flexible interrupt architecture

The MAXQ10 and MAXQ20 support a single, user-configurable, interrupt-vector address register. This scheme allows placement of the interrupt identification and servicing routines according to user preference. There is no natural priority forced upon any interrupt source. In addition to the normal individual and global interrupt enables and flags, masking and identification flags are provided at the module level. The individual source enabling, module-to-global level masking, and prioritization of interrupt sources is under the control of user code. The described interrupt support structure can be advantageous. First, no code space is left unused. This generally cannot be said for microcontrollers having dedicated interrupt-vector addresses per source, as code space associated with unused interrupt vectors is often left unused. Second, the user has increased control over which interrupts are enabled and over interrupt prioritization.

Hardware-loop counters reduce overhead

The MAXQ architecture implements a DJNZ instruction that can operate with either of two 16-bit loop counter (LC[0] or LC[1]) registers. In a single-clock cycle, the “DJNZ LC[n], src” instruction decrements the loop counter register and, if the counter has not reached 0, it conditionally branches program execution to the specified address. For competing RISC microcontrollers, updating a counter register and testing for a loop-terminating condition are generally two separate operations. Merging the two actions in the MAXQ means that software loops, commonplace in microcontroller application code, require less code and cycle overhead to manage the loop counter. The single-cycle, DJNZ-triggered loop-counter decrement and conditional branch operation exactly follow our objective of maximizing utilization of clock cycles.

Enhanced data pointers

The MAXQ comes equipped with three 16-bit data pointers (DP[0], DP[1], and BP[Offs]). All three data pointers are individually configurable for either word- or byte-access mode via the Word/Byte Select (WBSn) register bits in the Data-Pointer-Control (DPC) register. All three data-memory pointers support single-cycle indirect memory access with pre-increment/decrement for write operations and post-increment/decrement for read operations. One of the data pointers, the Frame Pointer (FP=BP[Offs]), is generated by the unsigned additive combination of a 16-bit base-pointer (BP) register and an 8-bit offset (Offs) register. This type of pointer is especially important to C compiler development tools, and more specifically, in the handling of stack frames.

Harvard memory architecture with Von Neumann benefits

The MAXQ architecture uses a Harvard memory organization, one in which the program and data memory buses are separate, so that simultaneous access to an instruction word and a data word can occur in the same clock cycle. This style of memory organization is necessary to achieve maximum performance and support single-cycle execution of instructions that access data memory. Microcontrollers that use a Von Neumann memory interface experience performance bottlenecks associated with sharing bus bandwidth among accesses to program memory, data memory, I/O, and peripherals.

Advocates of the Von Neumann memory architecture cite the inability to access program space as data memory and vice versa as a weakness. Having accessibility can simplify constant storage, look-up tables, and in-system or in-application programming alternatives. The MAXQ architectural solution to this weakness is insertion of a memory-management unit (MMU) and fixed-utility ROM that provide logical memory mappings and fixed utility-code routines to support in-system programming and the desired access modes.

Centralized access to resources

Another important feature of the MAXQ architecture is the presence of a single transfer map that contains access points to all resources. The reason for calling this a transfer map and not simply a register map is the transfer-trigger concept upon which the MAXQ architecture is based.

The transfer map is partitioned into 16 modules. Within each module are 32 indexes or individual access points. It should be emphasized, once again, that these access points can be used for direct read/write access to registers, but they may also be used for indirect access to memory or to trigger hardware operations. Of the 16 modules, the first six modules (M0–M5) are allocated for device-specific peripheral functions. This provides a generous amount of space (6 x 32 = 192 locations) in the transfer map for peripheral registers and access. These modules, based upon the specific MAXQ device option, are populated with registers to implement functions such as digital I/O, timers, serial ports, hardware multiplier, LCD driver, ADC, and in-circuit debugger. The last 10 modules (M6–M15) are reserved for MAXQ system functionality. The system modules contain registers that are vital to MAXQ system operation, such as those used for watchdog, system clock, and interrupt control. The system modules additionally contain the working accumulator file, data pointers, and source/destination encodings that trigger indirect memory access and/or special machine operations. The basic system register space is intentionally kept as common as possible among MAXQ device options. **Figure 2** presents an example MAXQ source and destination transfer map.

The MAXQ contains a single transfer map that is partitioned into 16 modules providing access points to all resources.

The prefix register module is a feature of the MAXQ architecture that deserves special mention. There exists a single prefix register in which the data (default = 00h) is used for those transfer operations requiring it. This prefix register, when loaded, holds data for one clock cycle before being returned to a 00h state. An index (n) must accompany the prefix-register (PFX[n]) selection. Since there are 16 modules and 32 indexes per module in the transfer map, certain locations cannot be directly accessed using the source/destination encoding bits available in a single-instruction word. This is true of the latter 16 source indexes and the latter 24 destination indexes in a module. The prefix register solves this problem by opening an access window to these locations, which lasts for one cycle. When the PFX[n] register is loaded, its index "n" supplies the high-order source and destination bits to the instruction immediately following, where $n = dds$. In this respect, the prefix-register module is a means through which additional decoding bits can be supplied to access extended (and/or protected) registers. Operations and accesses that require loading the prefix register are automatically generated by the assembler and need not be manually coded by the user. The prefix-register module can also be used to concatenate source bytes when writing to 16-bit destinations. Although transparent to the user, the prefix register is used exactly in this fashion for jumps and calls to 16-bit absolute addresses. For those interested in future enhancements to the MAXQ architecture, the prefix-register module provides a seamless mechanism for MAXQ instruction-set expansion or extension into currently unused system-module space.

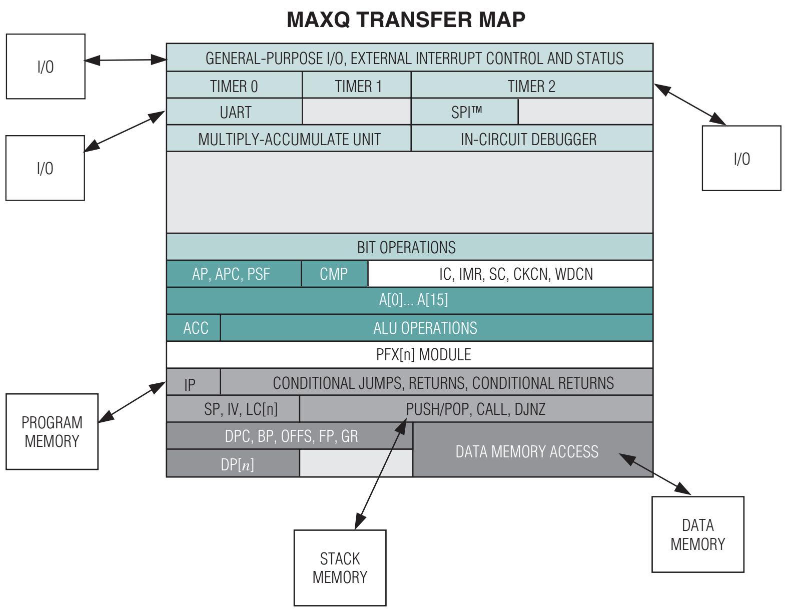


Figure 2. All MAXQ resources are accessible through a central transfer map.

To summarize, the complete transfer map on any MAXQ device contains all system and peripheral registers defined for the device. The same map provides indirect access points to the data memory, stack memory, and accumulator array. The same map contains access points that trigger MAXQ machine instructions and underlying operations, and a mechanism for simple extension of the instruction set in future MAXQ families. With access points to all resources aggregated into a central transfer map, the number of source-to-destination transfer opportunities is very large. The centralized access also simplifies clock distribution to only those resources needing a clock. This promotes a very quiet environment (hence the "Q" in MAXQ) that is advantageous when integrating analog peripherals. The MAXQ architecture allows maximum modularity and portability of peripheral functions. This strategy, intentionally adopted to align

...the modular organization of MAXQ system and peripheral resources leads to compiler optimizations and allows portability of modules for rapid creation of new MAXQ derivatives.

with rapid product development cycles and ever-changing peripheral requirements of the end user, promotes flexibility and reuse. The modularity of peripheral functions minimizes the design time required to duplicate, add, or remove standard MAXQ peripheral modules when creating new MAXQ devices for certain markets or applications.

Conclusion

The MAXQ architecture is truly an innovation in today's microcontroller industry. The MAXQ exploits a transfer-triggered architecture to achieve the objectives of high bandwidth, high efficiency, and high orthogonality. Furthermore, the modular organization of the MAXQ system and peripheral resources leads to compiler optimizations and allows portability of modules for rapid creation of new MAXQ derivatives. Looking to the future, the MAXQ architecture incorporates a built-in mechanism for instruction-set expansion suitable for next-generation products. These compelling benefits make the MAXQ architecture an ideal solution for existing and future projects, as it will inevitably rank high no matter the selection criteria for the project.

Benchmarking the MAXQ instruction-set architecture vs. RISC competitors

This article compares the MAXQ instruction set with competing microcontrollers, including the PIC16CXXX (mid-range devices), AVR, and MSP430. A table details the strengths and weaknesses of each instruction set and architecture. We will use selected code algorithms and operations for judging code density and code performance. A final section introduces and highlights the MIPS (millions of instructions per second)/mA ratio for each code example.

Overview of MAXQ instruction set

The MAXQ instruction set is founded upon the transfer-trigger concept. The instruction word is composed simply of source and destination operands. While these source and destination operands may represent physical registers, the encodings may also represent indirect access points to data memory, stack memory, and the working accumulators, and/or may implicitly trigger hardware operations. Additional information on the MAXQ transfer-triggered architecture can be found in the previous article of this journal. Source and destination encodings for specific MAXQ devices are defined in the MAXQ User Guide(s) associated with the device. While some source and destination encodings may be device specific, such as those designated for peripheral hardware functions, certain fixed encodings are identified for building the MAXQ base instruction set. **Figure 1** gives the MAXQ instruction word and instruction set mnemonics.

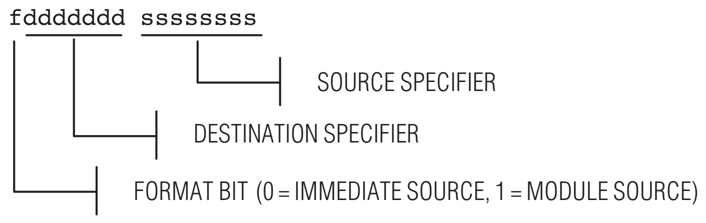


Figure 1. The source-to-destination transfer illustrated in the MAXQ instruction word produces a small, yet very potent instruction set.

MNEMONIC	DESCRIPTION	MNEMONIC	DESCRIPTION
BIT MANIPULATION		LOGICAL	
MOVE C, #0/#1	Clear/Set Carry	AND	Logical AND
CPL C	Complement Carry	OR	Logical OR
AND Acc.<b>	Logical AND Carry with Accumulator Bit	XOR	Logical XOR
OR Acc.<b>	Logical OR Carry with Accumulator Bit	CPL,NEG	One's, Two's Complement
XOR Acc.<b>	Logical XOR Carry with Accumulator Bit	SLA,SLA2, SLA4	Shift Left Arithmetically 1,2,4
MOVE C, Acc.<b>	Move Accumulator Bit to Carry	SRA,SRA2,SRA4	Shift Right Arithmetically 1,2,4
MOVE Acc.<b>,C	Move Carry to Accumulator Bit	SR	Logical Shift Right
MOVE C, src.<b>	Move Register Bit to Carry	RR,RRC	Rotate Right Carry (Ex/In)clusive
MOVE dst.<b>, #0/#1	Clear/Set Register Bit	RL,RLC	Rotate Left Carry (Ex/In)clusive
MATH		DATA TRANSFER	
ADD, ADDC	Add Carry (Ex/In)clusive	XCHN	Exchange Accumulator data nibbles
SUB, SUBB	Subtract Carry (Ex/In)clusive	XCH *(MAXQ20)*	Exchange Accumulator data bytes
FLOW CONTROL AND BRANCHING		MOVE dst, src	Move source to destination
JUMP {C/NC/Z/NZ/E/NE/S}	Jumps - unconditional or conditional, relative or absolute	PUSH/POP	Push/Pop stack
DJNZ LC[n], src	Decrement Counter, Jump Not Zero	POPI	Pop stack and enable interrupts (INS≤0)
CALL	Call – relative or absolute	Other	
RET {C/NC/Z/NZ/S}	Return – unconditional or conditional	NOP	No Operation
RETI {C/NC/Z/NZ/S}	Return from Interrupt – unconditional or conditional	CMP	Compare with Accumulator

...the instruction set and the device architecture (instruction cycle, memory model, register set, addressing modes, etc.) are inseparable and must be considered together.

Table 1. Instruction Set Comparisons

ISA	STRENGTH	WEAKNESS
AVR	• 32 general-purpose working registers (accumulators) • Data pointers are part of the directly addressable working registers; allow easy masking and bit-manipulation of high/low pointer bytes. • Read from pointer + displacement (0 to 63-byte displacement) • Stack limited only by internal RAM (except 90S1200 with no RAM, then stack depth = 3) • Single-cycle operation • Relative jumps ±2k (two-cycle) • All AVR have data EEPROM • Explicit instructions to set/clear each status register flag; large group of bit-manipulating instructions • Separate interrupt vectors	• Pipelined instruction fetch • Beyond the 32 regs, load (LD)/store (ST) overhead becomes a factor LD/ST @X,Y,Z = two cycles, • LPM = 3 cycles • Reduced support/scope on literal operations (no ADDC, EORI; only CPI, ORI, ANDI, SUBI, SBCI, LDI work on R16–R31) • No rotate instructions exclusive of carry • Conditional jump range only +63/-64 (two-cycle) • CALL/RET/RETI = four cycles
PIC16CXXX	• Source, destination bit encoded into ALU operations • Direct data access (symbolic addressing mode) can produce dense code and is conducive to data overlays	• four-clock core yields poor execution speed • Pipelined instruction fetch • Access to upper data-memory banks requires paging (RP1:0 bank select) • Indirect data access requires INDF,FSR registers • Cannot directly load W (accumulator) • No ADDC, SUBB • Stack depth = 8 • No relative jumps/branches—only absolute (CALL, GOTO) or conditional skips (BTFSx) • RETLW for code memory reads = wasted code space and does not allow CRC of code space • CALL/GOTO/RET/RETFIE/RETW all require eight clock cycles (two instruction cycles) • Single interrupt vector

MAXQ vs. other instruction-set architectures

One could attempt to compare the MAXQ instruction mnemonics against those of other architectures, but this analysis would be difficult and unjustified because each instruction set is architected around specific device resources and addressing modes. For this reason, the instruction set and the device architecture (instruction cycle, memory model, register set, addressing modes, etc.) are inseparable and must be considered together. **Table 1** summarizes the strengths and weaknesses of the instruction-set architectures being compared.

Code examples

The best way to compare instruction-set architectures is to define some set of tasks and write the code to perform those tasks. The sections that follow describe certain tasks to be performed and summarize the code density and performance results for each instruction-set architecture. Example code for the first routine is included in the document, while the routines that follow will only be summarized with graphs and text. The code routines corresponding to each set of statistics are available from Dallas Semiconductor upon request.

Table 1. Instruction Set Comparisons (continued)

ISA	STRENGTH	WEAKNESS
MSP430	• Extensive source, destination addressing modes are encoded within the op code—can yield dense code • 16-bit internal data path • Internal memory accessible as word or byte • Constant generator (CG) for -1, 0, 1, 2, 4, 8 • Single-cycle operation • Stack limited only by internal RAM • Conditional/relative jump destination range = ±512 (two-cycle) • Separate interrupt vectors, single-source flags automatically cleared	• Von Neumann memory map + elaborate addressing modes = many cycles. The ONLY single-cycle instructions are those dealing exclusively with Rn. Peripheral register access = three to six cycles • Literals not supported by CG require extra word • Destination operand cannot be register indirect or register indirect auto-increment • No auto-decrement support for register indirect • Symbolic addressing limits the ability to reuse code routines
MAXQ	• System and peripheral registers are accessible as source or destination in the same logical memory space, yielding the fastest data transfers • Single-cycle operation and no pipelining • Single-cycle conditional jump (+127/-128) or two-cycle absolute jump (0–65,535) • Single-cycle CALL/RET/RETI • Auto-decrementing loop-counter registers eliminate overhead normally wasted when maintaining a counter • Three data pointers with auto-increment/decrement support. One data pointer, FP, supports base pointer + offset addressing (i.e., BP[Offs]). • Auto-increment/decrement/modulo controls for accumulator (working register) file • Selectable word or byte-access mode for each data pointer • Prefixable op code allows a simple means for instruction set extensions or enhancements	• Active accumulator is always the implicit destination for ALU operations • Single-port, synchronous, SRAM data memory requires that a data pointer be activated (selected) before being used • Default stack depth = 16, however, data pointer hardware is ideal for implementing a soft stack in data memory

Memory copy (*MemCpy64*)

System and peripheral registers are accessible as source or destination in the same logical memory space, yielding the fastest data transfers.

The memory copy example demonstrates the microcontroller's ability to indirectly manipulate blocks of data memory. The task is to copy 64 bytes from a data-memory source location to a nonoverlapping data-memory destination. The code routines for each microcontroller are provided on the following pages, along with graphs that summarize the cycle count and byte count for the copy operation. These routines assume that the pointer and byte count have already been defined before the copy operation, and that the bytes to be copied are word-aligned in memory so the word access modes of the MSP430 and MAXQ20 can be used.

```
;=======================================AVR======================================
; ramsize=r16         ;size of block to be copied
; Z-pointer=r30:r31   ;src pointer
; Y-pointer=r28:r29   ;dst pointer
; USES:
; ramtemp=r1          ;temporary storage register
loop:                          ; cycles
       ld      ramtemp,Z+      ; 2 @src => temp
       st      Y+,ramtemp      ; 2 temp => @dst
       dec     ramsize         ; 1
       brne    loop            ; 2/1
       ret                     ;  4/5
                               ;---------
                               ;(7*bytecount) + return –1(last brne isn’t taken).
```
; WORD COUNT = 5 ; CYCLE COUNT = 451

```
;======================================MAXQ10=====================================
; DP[0] ; src pointer (default WBS0=0)
; DP[1] ; (dst-1) pointer (default WBS1=0)
; LC[0] ; byte count (Loop Counter)
loop:                                 ;words & cycles
       move    DP[0], DP[0]           ; 1  implicit DP[0] pointer selection
       move    @++DP[1],@DP[0]++      ; 1
       djnz    LC[0], loop            ; 1
       ret                            ; 1
                                      ;----------
                                      ; 4 / (3*bytecount) +1
```
; WORD COUNT = 4 ; CYCLE COUNT = 193

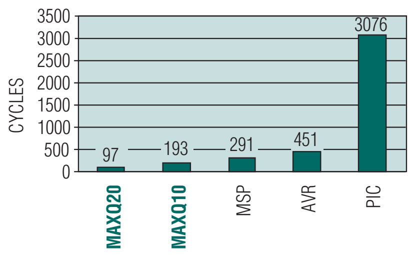
```
;=====================================MAXQ20======================================
; Assuming bytes are word aligned (like MSP430 code) for comparison
; DP[0] ; src pointer (default WBS0=1)
; DP[1] ; (dst-1) pointer (default WBS1=1)
; LC[0] ; byte count  (Loop Counter)
loop:                                 ;words/cycles
       move    DP[0], DP[0]           ; 1  implicit DP[0] pointer selection
       move    @++DP[1],@DP[0]++      ; 1
       djnz    LC[0], loop            ; 1
       ret                            ; 1
                                      ;----------
                                      ; 4 / (3*bytecount/2) +1
```
; WORD COUNT = 4 ; CYCLE COUNT = 97

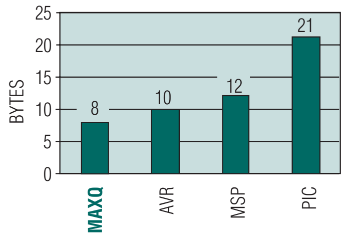
```
;=====================================MSP430======================================
; MSP430 has a 16-bit data bus
; assuming bytes are word aligned, only requires (blocksize/2 transfers).
; R4   ;src pointer
; R5   ;dst pointer
; R6   ;size of block to copy
loop:                          ;words/cycles
       mov     @R4+, 0(R5)     ;2 / 5  @src++ => dst
       add     #2, R5          ;1 / 1  const generator makes this 1/1
       decd.b  R6              ;1 / 1  really sub #2, R6
       jz      loop            ;1 / 2
       ret                     ;1 / 3
                               ;----------
                               ;6 / (9*(bytecount/2)) + return
```
; WORD COUNT = 6 ; CYCLE COUNT = 291

```
;====================================PIC16CXXX====================================
; a     ; src pointer base
; b    ; dst pointer base
; i    ; byte count held in reg file
; USES:
; temp  ; temp data storage
loop:                                 ; cycles
        decf     i, W                 ; 1 i-- => W
        addlw    a                    ; 1 (a+i--) => W starting at end
        movwf    FSR                  ; 1 W => FSR
        movfw    INDF                 ; 1 W <= @FSR get data
        movwf    temp                 ; 1 W => temp
        movlw    (b-a)                     ; 1 diff in dest-src
        addwf    FSR, F                    ; 1 (b+i--) => W
        movfw    temp                      ; 1 temp => W
        movwf    INDF                      ; 1 W => @FSR store data
        decfsz   i, F                      ; 2/1 i--
        goto     loop                      ; 2
        return                             ;   2
                                           ;----------
                                           ;11 / (12*bytecount) +1 (ret instead of
goto, +1 on decfsz)
; WORD COUNT = 12 ; CYCLE COUNT = 769 (*4clks/inst cycle = 3076)
```

The MAXQ advantages illustrated in the memory copy example translate into similar gains for applications requiring frequent input/output buffering in data memory.

MemCpy64 **CYCLE/BYTE COUNT COMPARISON**





The MAXQ devices provide the best code density and are the clear winners in execution speed. The MAXQ10 performs the copy operation slower than the MAXQ20 because it uses the default byte-access mode for the data pointers. For a MAXQ10 application, if execution speed is deemed more important than code density and the data memory to be copied is word-aligned (an assumption already being made for the MSP430 and MAXQ20 example), it could use word-access mode for the source and destination data pointers. Enabling word mode would allow the MAXQ10 copy loop to be cut in half, but would require additional instructions to enable/disable word-access mode. The overwhelming performance advantage demonstrated by the MAXQ devices over the competition can be attributed to the following architectural strengths:

1) No pipelining—branches do not incur the overhead of flushing the instruction prefetch as other devices do.

2) Auto-decrement loop counter—alleviates the need to do this manually.

3) Harvard memory map—program and data do not share the same physical space, allowing simultaneous program fetch and data access.

4) Post-increment/decrement indirect destination pointer—simplifies and speeds advancement of the destination pointer. This is a weakness of the MSP430, which uses 0(R5) to denote @R5, and then must advance that destination pointer in the following instruction.

The MAXQ advantages illustrated in the memory copy example translate into similar gains for applications requiring frequent input/output buffering in data memory. In terms of performance, the nearest competitor is the MSP430. As an example where data memory buffering may be desired, suppose we have an MSP430 device equipped with an ADC peripheral with a 16-bit output register. Transferring data from the peripheral output register into data memory and incrementing the pointer in preparation for the next ADC output sample might be handled with code such as this:

```
                                          ; words/cycles
       mov.w    &ADAT,0(R14)              ; 3 / 6       Store output word
                 incd.w   R14             ; 1 / 1       Increment pointer
                                          ; 4 / 7
```

The same transfer operation would look like this on the MAXQ20:

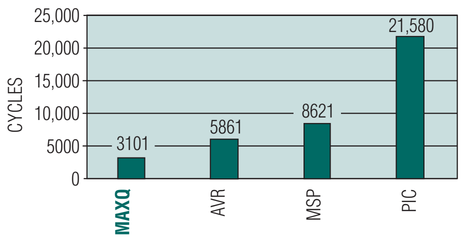
```
       move    @DP[0]++, ADCOUT           ; 1 / 1
```

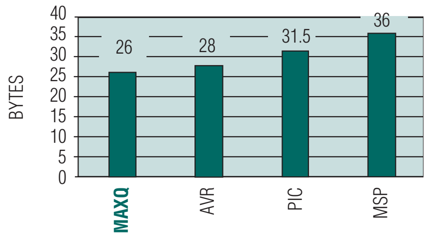
The bubble sort routine not only demonstrates the ability to access data memory efficiently, but also performs arithmetic and/or comparison operations between data bytes and conditionally reorders the bytes.

Bubble sort (*BubbleSort*)

The bubble sort routine not only demonstrates the ability to access data memory efficiently, but also performs arithmetic and/or comparison operations between data bytes and conditionally reorders the bytes. The code routine sorts 32 data-memory bytes so they are left in an ascending or descending order. The cycle counts assume that byte reordering occurs approximately half of the time as a result of adjacent byte comparisons. The graphs below summarize the cycle count and byte count for the sort operation on each microcontroller.

***BubbleSort* CYCLE/BYTE COUNT COMPARISON**



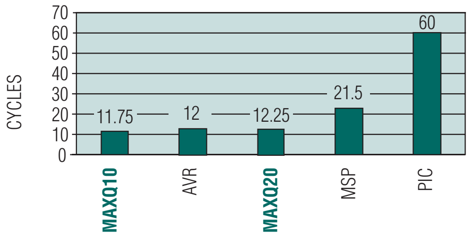


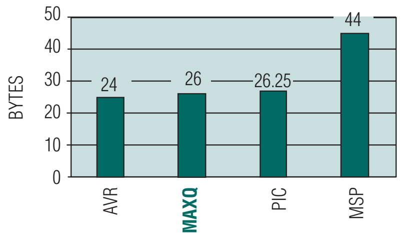
The MAXQ devices, once again, yield the best code density and are the clear winners in execution speed. The MAXQ advantages can be attributed to the same architectural strengths discussed in the memory copy example.

Hex-to-ASCII conversion (*Hex2Asc*)

This conversion routine tests the scope of the microcontrollers' arithmetic and logical operations. It also tests their support of literal byte data when translating and expanding data contained within a single byte. The cycle count represents an average value, given that each nibble can be one of 16 hex values—0 to 9, A to F. The graphs below summarize the cycle count and byte count for the conversion operation on each microcontroller.

***Hex2Asc* CYCLE/BYTE COUNT COMPARISON**





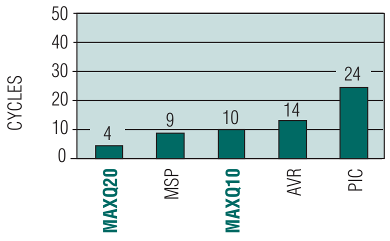
For this test routine, the AVR requires one fewer word since its working registers are directly accessible, whereas the most efficient method for the MAXQ requires a manual update of the accumulator pointer. The MSP code density suffers because it lacks operations for manipulating nibbles, and because literals (#nnnnh) not supported by the constant generator must be encoded in a separate word. The MAXQ devices and the Atmel AVR achieve similar results in the performance area, while other devices lag behind. The MSP430 performance suffers from the extra code words to perform the operation.

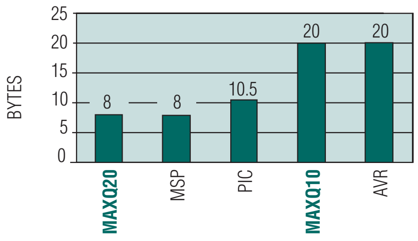
Arithmetic shift right 2 positions (*ShRight*)

This routine demonstrates the microcontrollers' ability to support 16-bit word data-memory access and ALU operations. The desired operation is to arithmetically shift (i.e., preserving the most significant bit) a 16-bit word that resides in data memory. It is assumed that the word resides in the first 256 bytes of data memory and is aligned in memory to be word addressable by those microcontrollers with the capability. The following graphs summarize the cycle count and byte count for the shift operation on each microcontroller.

Both microcontrollers that support 16-bit ALU operations, the MAXQ20 and MSP430, provide significantly better code density. With exception of the PIC, all of the 8-bit machines require at least twice the number of code words to accomplish the same arithmetic shift. The MAXQ20 offers the best performance, and the MAXQ10, while supporting only 8-bit ALU operations, approaches the performance of the 16-bit MSP430.

***ShRight* CYCLE/BYTE COUNT COMPARISON**



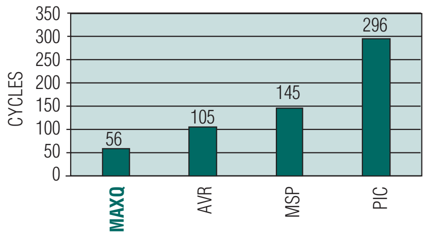


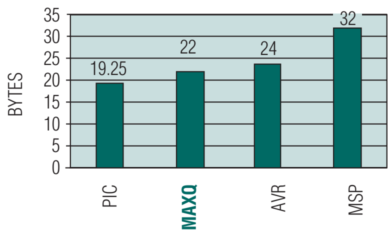
The MAXQ20 and MSP430 demonstrate higher code density because of their ability to handle 16-bit data more efficiently than the 8-bit machines. Each does so, however, in a slightly different fashion. The MAXQ20 transfers the 16-bit word to be shifted into a working register (accumulator) where it can use a multibit arithmetic shift. The MSP430 performs single-bit arithmetic shift operations using the register indirect-addressing mode (RRA @R5), and does not explicitly transfer the word from its memory location. While offering higher performance, the MAXQ20 can provide the same or better code density as the MSP430, when the arithmetic shifting of a 16-bit word can use one of the multibit arithmetic shift op codes (SRA2, SRA4, SLA2, SLA4).

Bit-bang port pins (*BitBang*)

This example tests the ability of an instruction-set architecture to decompose bytes, either by direct bit manipulation or through shift/rotate, and send the individual bits to a port pin ("bit-banging"). The port-pin outputs separately represent clock and data, with the requirement that data must be valid on the rising edge of clock. Since the code is directly manipulating the port pins, this test also demonstrates the ease with which I/O port registers can be accessed. The graphs below summarize the cycle count and byte count for the port bit-bang operation on each microcontroller.

***BitBang* CYCLE/BYTE COUNT COMPARISON**





The MAXQ devices again are clearly the best performers. The PIC performance is limited here (as in other examples) because of the underlying 4-cycle core architecture. The MSP430 performance is worse and can be attributed to both its Von Neumann memory architecture and required use of absolute addressing to access the port output register.

The BitBang example tests the ability of an instruction-set architecture to decompose bytes, either by direct bit manipulation or through shift/rotate, and send the individual bits to a port pin ("bit-banging").

With respect to code density, the MAXQ and PIC have the same word count. Yet the PIC edges out the MAXQ among the RISC machines because of its 14-bit program word versus the 16-bit program word of the MAXQ. The MSP430 code density suffers because it must use at least two words to access its peripheral registers with the absolute-addressing mode (i.e., &register) or when using literals that cannot be reduced by the constant generator (e.g., #3h).

The MSP430 method of accessing its peripheral registers deserves further comment. The microcontroller's primary duty is to interface in some way with the outside world. Thus it must control, monitor, and process activity that occurs at I/O pins. If the microcontroller embeds very few peripheral-hardware modules, the burden of this activity is left to the software. For the

The MIPS/mA metric provides a simple means for assessing the code efficiency of a microcontroller while factoring in active current consumption.

software to do anything meaningful, it must read and write the port pins. On the MSP430, these port-pin registers reside in the peripheral register space that requires use of the absolute-access mode. Now consider a microcontroller that is rich with "smart" peripherals. There will undoubtedly be more peripheral registers that must be configured, controlled, and accessed during the course of using the on-chip, dedicated hardware to perform the necessary function. On the MSP430, these registers reside in the peripheral register space that requires use of the absolute-access mode. Consequently, there is no escape around the code density and performance penalty associated with the MSP430 absolute addressing mode.

The "MIPS/mA" metric

Power consumption is often a significant factor in the selection of a processor or core architecture. The overall power consumption of a given system depends upon many factors such as supply voltage and operating frequency, and its ability to use low-power modes whenever possible. Reduced supply voltage(s) and/or operating frequency, along with frequent use of low-power modes, can greatly reduce the total system power consumption. While the minimum supply voltage for a given microcontroller depends greatly upon the device fabrication process technology, the ability to reduce operating frequency and use low-power mode(s) is largely dependent upon application requirements that can be determined by the system designer. The MIPS/mA metric provides a simple means for assessing the code efficiency of a microcontroller while factoring in active current consumption. A common supply voltage should be chosen to create meaningful MIPS/mA comparisons between different devices. For the forthcoming comparison, a 3V-supply voltage is assumed. To factor in differences and efficiencies in the instruction-set architectures being compared (i.e., AVR, MSP430, PIC16, MAXQ), it is also necessary to generate separate MIPS/mA ratios for each code example generated.

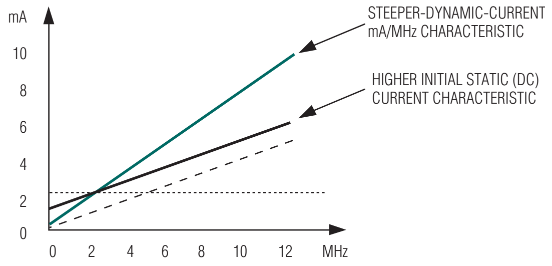


Figure 2. This example for IccActive vs. MHz illustrates the effects of increased static and dynamic current.

To determine the "mA" portion of the MIPS/mA ratio, we examine data sheets of the devices. Most microcontroller vendors specify typical and maximum active current associated with the maximum operating frequency of the device. Assuming very small static (DC) current, these data points allow one to derive typical and maximum mA/MHz approximations used for extrapolating active current at any clock frequency. The mA/MHz ratio can be better quantified and defined relative to specific system environmental conditions if the vendor provides active current vs. temperature/frequency characterization data. Otherwise, we must simply rely on the discrete data points and our assumption of very small static current. Increased static (DC) current changes the starting point for the mA vs. MHz characteristic curve, thereby limiting the overall gain seen by the system designer when reducing clock frequency (reducing dynamic current). **Figure 2** gives an example IccActive vs. MHz graph. **Table 2** compares mA/MHz numbers for the various cores and cites the source for the information. The highlighted mA/MHz number for each architecture is used when this term is required in later calculations.

The "MIPS" portion of the MIPS/mA metric is used to quantify the difference in performance. We will start by giving a simple equation for MIPS in **Figure 3**.

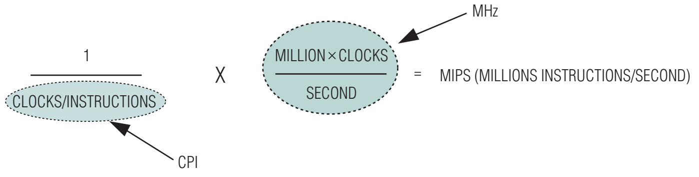


Figure 3. The MAXQ architecture achieves a high-MIPS performance ratio by executing nearly all instructions at one clock per instruction.

The number of clocks per instruction (CPI) is highly important when assessing MIPS for a given architecture. Architectures such as the Microchip PIC, for example, require multiple clocks per instruction cycle. Additionally, architectures often require multiple instruction cycles to execute certain instructions or need cycles to flush the instruction pipeline when performing jumps/branches. When comparing architectures, the average performance in MIPS is often much less than the peak performance (MIPS) and varies depending upon instruction mix.

Table 2. Comparison of mA/MHz Numbers for Various Cores

DEVICE	TYPICAL mA/MHz	MAX mA/MHz	SOURCE
PIC16C55X	0.7	1.25	PIC16C55X data sheet: DC Table 10.1, D010 (V_{CC} = 3V, 2MHz); XT or RC
PIC16C62X	0.7	1.25	PIC16C62X data sheet: DC Table 12.1, D010 (V_{CC} = 3V, 2MHz); XT or RC
PIC16LC71	0.35	0.625	PIC16C71X data sheet: DC Table 15.2, D010 (V_{CC} = 3V, 4MHz); XT or RC
PIC16F62X	0.15	0.175	PIC16F62X data sheet: DC Table 17.1, D010 (V_{CC} = 3V, 4MHz)
PIC16LF870/1	0.15	0.5	PIC16F870/1 data sheet: DC Table 14.1, D010 (V_{CC} = 3V, 4MHz); XT or RC
AT90S1200	0.33	0.75	AT90S1200 data sheet: EC Table (3V, 4MHz), Figure 38, 4mA/12MHz (typ)
AT90S2313	0.50	0.75	AT90S2313 data sheet: EC Table (3V, 4MHz), Figure 57, 7.5mA/15MHz (typ)
MSP430F1101	0.30	0.35	MSP430x11x1 data sheet: DC specs IccActive (V_{CC} = 3V, FMCLK = 1MHz)
MSP430C11X1	0.24	0.30	MSP430x11x1 data sheet: DC specs IccActive (V_{CC} = 3V, FMCLK = 1MHz)
MSP430Fx12x	0.30	0.35	MSP430x12x data sheet: DC specs (V_{CC} = 3V, FMCLK = 1MHz, FACLK = 32kHz)
MAXQ10	0.30		**Simulations**
MAXQ20	0.30		**Simulations**

To produce a more useful indicator and generate a value that helps us reach our MIPS/mA target metric, we divide MIPS by MHz. The MIPS/MHz ratio can be interpreted as the average number of instructions that execute in a single clock (for the given code example). Using the MIPS/MHz number and the mA/MHz number calculated earlier, the MIPS/mA ratio can be generated. The tables below show the MIPS/MHz and MIPS/mA numbers, respectively, for each of the earlier code-routine comparisons.

Table 3. Comparison of MIPS/MHz and MIPS/mA for Selected Code Algorithms

CORE	MIPS/MHz					
	MemCpy64	BubbleSort	Hex2Asc	ShRight	BitBang	*Peak*
MAXQ10	**1.00**	**0.99**	**1.00**	**1.00**	**1.00**	*1*
MAXQ20	**1.00**	**0.99**	**1.00**	**1.00**	**1.00**	*1*
PIC	0.23	0.20	0.23	0.25	0.21	*0.25*
MSP	0.44	0.39	0.64	0.33	0.38	*1*
AVR	0.57	0.62	0.90	0.71	0.61	*1*

CORE	MIPS/mA				
	MemCpy64	BubbleSort	Hex2Asc	ShRight	BitBang
MAXQ10	**3.33**	**3.30**	**3.33**	**3.33**	**3.33**
MAXQ20	**3.33**	**3.30**	**3.33**	**3.33**	**3.33**
PIC	1.53	1.35	1.53	1.67	1.40
MSP	1.85	1.62	2.66	1.39	1.55
AVR	1.71	1.86	2.69	2.14	1.83

To take the analysis one step further, we must factor in differences between core architecture and instruction-set efficiency by dividing the MIPS/mA ratio by the number of instructions that are actually executed for the given code sample. The rationale for this extra calculation is that the execution of three single-cycle instructions (with the highest MIPS/MHz ratio = 1) is really no better than one 3-cycle instruction (MIPS/MHz ratio = 0.33). Nonetheless, the resultant MIPS/mA ratio differs drastically. In fact, most would prefer a single instruction to three if the same task were accomplished. By dividing the MIPS/mA ratio by the number of instructions

The MIPS/MHz ratio can be interpreted as the average number of instructions that execute in a single clock (for the given code example).

executed, we are adjusting the MIPS/mA ratio to the instruction mix used by a given microcontroller to perform a specific task. The resultant values have been normalized to the highest performer and are presented in the table below.

Table 4. Comparison of Normalized MIPS/mA Values

CORE	NORMALIZED (MIPS/mA)				
	MemCpy64	**BubbleSort**	**Hex2Asc**	**ShRight**	**BitBang**
MAXQ10	0.50	**1.00**	**1.00**	0.40	**1.00**
MAXQ20	**1.00**	**1.00**	0.96	**1.00**	**1.00**
PIC	0.06	0.29	0.39	0.33	0.38
MSP	0.42	0.45	0.68	0.56	0.48
AVR	0.19	0.48	0.88	0.26	0.48

Conclusion

The normalized "MIPS/mA" metric gives us a relative performance-to-current ratio for comparing microcontrollers with different architectures, instruction sets, and current-consumption characteristics. A higher normalized "MIPS/mA" ratio generally can yield one or both of the following benefits: (1) system clock frequency can be reduced, and (2) the duration of time spent in a low-power or sleep mode can be increased. Both of these possibilities serve to reduce the system's overall power consumption. Alternately, higher overall system performance can be realized while remaining within a given current/power budget. No matter the benefit, the high MIPS/mA ratio produced by the MAXQ architecture is a trustworthy indication of efficiency.

Programming in C for the DS80C400

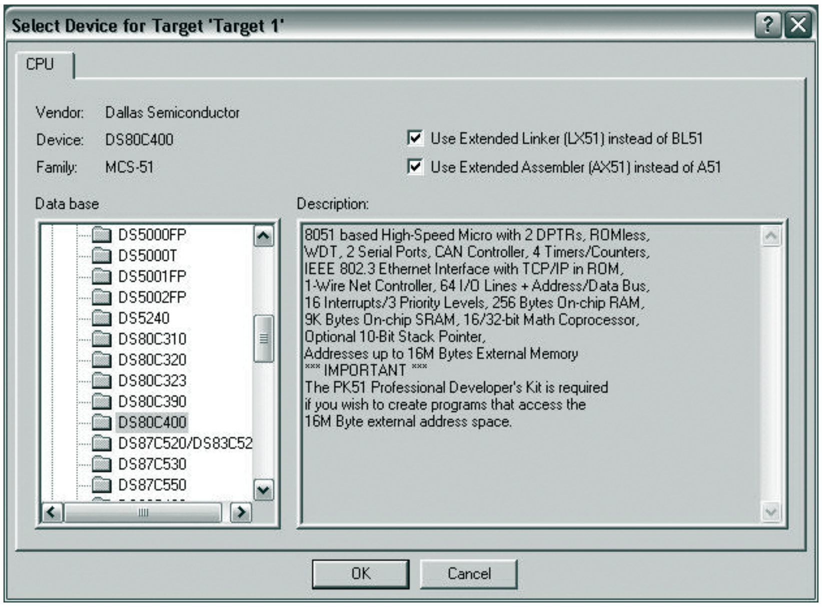
Since the introduction of the TINI® Runtime Environment for the DS80C390, developers have clamored for a way to use the power of TINI without using the Java™ language. Unfortunately, the network stack and other features of TINI were too intertwined with the Java virtual machine and runtime environment to be used from a C or assembly program. Later, when the ROM for the DS80C400 networked microcontroller was designed, a suite of functionality was exposed that could be accessed from programs written in 8051 assembly, C, or Java. Size constraints limited the functionality in the ROM to a subset of the functionality in the TINI Runtime Environment. The ROM would therefore be a useful starting block for building C and assembly programs because it offers a proven network stack, process scheduler, and memory manager. Simple programs like a networked speaker could easily be implemented in assembly language, while C could be used for more complex programs like an HTTP server that interacts with a file system.

This article starts with a C implementation of Hello World and moves on to a simple HTTP server. It describes how to set up the tools to write a simple tutorial program, and then demonstrates how to make use of the DS80C400's ROM functionality. All development was done using the TINIm400 verification module and Keil µVision2™ version 2.37, which includes the C compiler "C51" version 7.05.

Getting started with Keil's µVision2

You can build a simple Hello World-style program written in C using the Keil µVision2 development suite. Follow these instructions to complete your first C application for the DS80C400.

- Select **Project->Create New Project**. Enter the name of the project.
- The *Select Device for Target* dialog will pop up. Under *Data base*, select **Dallas Semiconductor** and the **DS80C400**. Select *Use Extended Linker*, and then select *Use Extended Assembler*. Hit **OK** to continue. **Figure 1** shows the proper configuration for this dialog.
- The dialog will ask, *Copy Dallas 80C390 Startup Code to Project Folder* and *Add File to Project?* Select **No**. We will supply our own startup code.
- When the project window opens on the left, open *Target 1*. Right click on *Source Group 1*, and select *Add files to group 'Source Group 1.'* In the file dialog that pops up, change *files of type* to *Asm Source file*. Add the file **startup400.a51**. This file can be found in the zip file at www.maxim-ic.com/HelloWorld.



Vendor: Dallas Semiconductor
Device: DS80C400
Family: MCS-51

☑ Use Extended Linker (LX51) instead of BL51
☑ Use Extended Assembler (AX51) instead of A51

Description:
8051 based High-Speed Micro with 2 DPTRs, ROMless,
WDT, 2 Serial Ports, CAN Controller, 4 Timers/Counters,
IEEE 802.3 Ethernet Interface with TCP/IP in ROM,
1-Wire Net Controller, 64 I/O Lines + Address/Data Bus,
16 Interrupts/3 Priority Levels, 256 Bytes On-chip RAM,
9K Bytes On-chip SRAM, 16/32-bit Math Coprocessor,
Optional 10-Bit Stack Pointer,
Addresses up to 16M Bytes External Memory
*** IMPORTANT ***
The PK51 Professional Developer's Kit is required
if you wish to create programs that access the
16M Byte external address space.

Figure 1. Select the DS80C400 for a new Keil µVision2 project.

The DS80C400's ROM is a useful starting block for building C and assembly programs because it offers a proven network stack, process scheduler, and memory manager.

- It is essential that the application is built for address 400000h, which corresponds to the beginning of the flash on the TINIm400. Open the file **startup400.a51** by double clicking on it. Find the segment declaration for `?C_CPURESET?0`. Make sure that this code segment is declared at `400000h`:

```
?C_CPURESET?0
SEGMENT CODE AT 400000h
```

- Additionally, there should be a "`DB'TINI'`" line followed by another single DB, with the comment "Target bank." This declaration is part of a tag that tells the DS80C400 ROM to execute the code starting at address 400000h. This ensures that the application is built for address 400000h, which should correspond to the beginning of the flash on the TINIm400. Make sure that line reads:

```
          DB    40h         ; Target bank
```

- Create a new file. Save it as "main.c." Write the following in that file:

```
#include <stdio.h>

void main()
{
     printf("Test 400 Program\r\n");
     while (1) { }
}
```

- Save the contents of this file. Right click on *Source Group 1* and add the source file **main.c**. The source file should now be added to the project.

- Right click on *Target 1* on the left. Select *Options for target 'Target 1'* to view an option dialog. The first tab selected should be *Target*. Change *Memory Model* to **Large: variables in XDATA**. Change *Code Rom Size* to **Contiguous Mode: 16MB program**. Select the check boxes for *Use multiple DPTR registers* and *far memory type support*. Under *Off-chip Code memory*, add the first entry with a *Start* of **0x400000** and *Size* of **0x80000**. For *Off-chip XData memory*, add an entry with a *Start* of **0x10000** and a *Size* of **0x4000**. **Figure 2** shows this dialog after it has been configured. Note that the last '0' in **0x400000** is not displayed in the window.

  These settings are based on the memory configuration of the TINIm400 reference module, which includes 512k of RAM at address 0 and 1M of flash at address 400000h. The starting addresses and sizes in the Keil configuration should be changed for custom DS80C400 designs.

- Select the *Output* tab. Click on *Create HEX File* and select *HEX-386* in the drop-down box.

- Press F7 to build the application. If every task was done correctly, the application should build with no errors or warnings. A hex file should have been generated. You can now load the application onto your board.

Loading the sample application onto the TINIm400 module

This section describes how to load the hex file produced by the Keil compiler onto the TINIm400 verification module by using the tool **JavaKit**.

To use **JavaKit**, you must have the Java Runtime Environment (at least version 1.2) and the Java Communications API installed. The Java Runtime Environment can be downloaded at http://java.sun.com/j2se/downloads.html, and the Java Communications API can be found at http://java.sun.com/products/javacomm/index.html. The **JavaKit** tool is included with the TINI Software Development Kit, available at www.maxim-ic.com/TINIdevkit. Instructions for running **JavaKit** can be found in the file **Running_JavaKit.txt** in the *docs* directory of the TINI Software Development Kit. If you encounter technical issues when running **JavaKit**, it is possible someone already had a similar problem, which is chronicled in the archives of the TINI Interest List. You can search the archives for this list at www.maxim-ic.com/TINI/lists.

Use this command line to have the **JavaKit** talk to the TINIm400 module.

```
java JavaKit -400 -flash 40
```

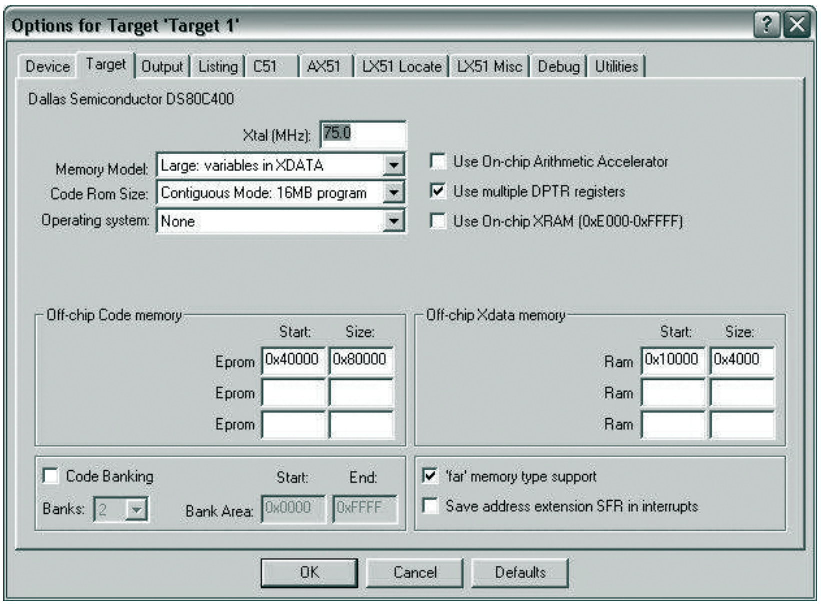
Once **JavaKit** is running, select the serial port you will use to communicate with the TINIm400. Open the serial port using the **Open Port** button. Then press the **Reset** button. The loader prompt for the DS80C400 should print and look like this:

```
DS80C400 Silicon Software -
Copyright (C) 2002 Maxim
Integrated Products

Detailed product information
available at http://www.maxim-
ic.com

Welcome to the TINI DS80C400
Auto Boot Loader 1.0.1

>
```



Dallas Semiconductor DS80C400

Xtal (MHz): 75.0
Memory Model: Large: variables in XDATA
Code Rom Size: Contiguous Mode: 16MB program
Operating system: None

☐ Use On-chip Arithmetic Accelerator
☑ Use multiple DPTR registers
☐ Use On-chip XRAM (0xE000-0xFFFF)

Off-chip Code memory

	Start:	Size:
Eprom	0x40000	0x80000
Eprom		
Eprom		

Off-chip Xdata memory

	Start:	Size:
Ram	0x10000	0x4000
Ram		
Ram		

☐ Code Banking — Banks: 2 — Bank Area: Start: 0x0000 End: 0xFFFF

☑ 'far' memory type support
☐ Save address extension SFR in interrupts

Figure 2. *The Target Options dialog is used to enter configuration information for the target platform. The configuration shown is suitable for use with the TINIm400 module.*

From the *File* menu at the top of **JavaKit**, select *Load HEX File as TBIN*. Find the *helloworld.hex* file that we just created, and select it. The *Load HEX File as TBIN* option converts the input hex file to a TBIN file, and then loads it. This operation is faster than loading it as a hex file because an ASCII hex file is more than twice as large as a binary file for the same data set.

There are two ways to execute your program once it is loaded. Since the program was loaded into bank 40, you can type:

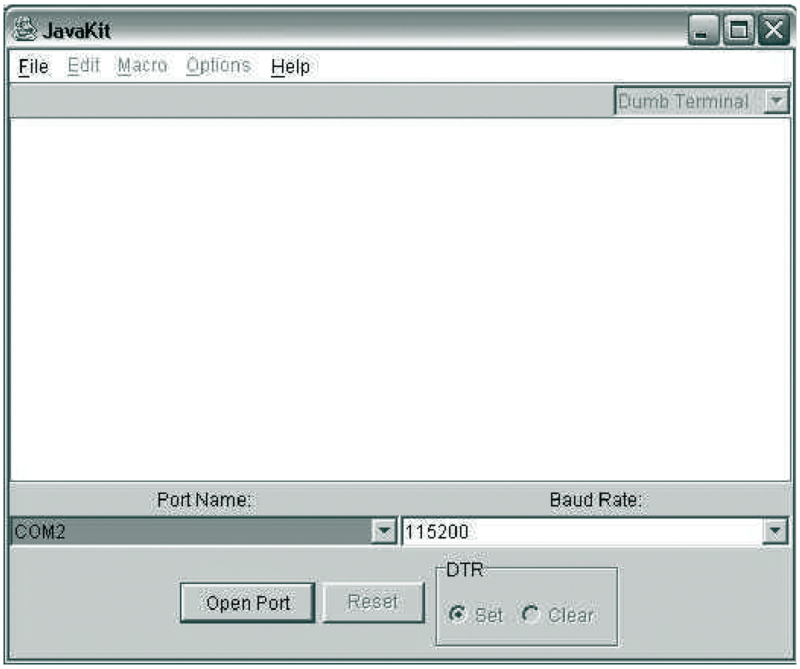
```
> B40
> X
```

To select bank 40 and execute the code there, you can also type:

```
> E
```

This will make the ROM search for executable code, a special tag signifying that the current bank has executable code. This tag consists of the text "TINI" followed by the current bank number. It is located at address 0002 of the current bank. Our *Hello World* program declares this tag in the *startup400.a51* file with the following lines:

```
?C_STARTUP:    SJMP   STARTUP1
               DB     'TINI'    ; Tag for TINI Environment 1.02c
                                ; or later (ignored in 1.02b)
               DB     40h       ; Target bank
```



Port Name: COM2
Baud Rate: 115200
DTR: [✓ Set] Clear

Figure 3. *The JavaKit program is used to load applications and communicate with the serial port of the DS80C400.*

In addition to the ROM libraries, other libraries (more are still being written) provide useful functionality not included in the ROM. Libraries have been developed for file system operations, DNS lookups, I²C communication, and 1-Wire communication.

Note that the `SJMP STARTUP1` statement is located at address 0000 of bank 40. It is followed by the executable tag { 'T', 'I', 'N', 'I', 40h }, located at address 0002, since the `sjmp` statement is two bytes.

When you type "E," the ROM searches downward through the memory banks for executable code. If you type "E" and some other code executes, it means that the ROM has found an executable tag at an address higher than 400000h, where your code was loaded. You may need to find that tag and delete the contents of that bank. You can erase a flash bank by using the **Z** loader command:

```
> Z41
You sure? Y
```

To erase all banks of flash, you need to zap from bank 40h to bank 4Fh.

Interfacing to the ROM and the ROM libraries

Calling the ROM functions from C is complicated. (The procedure for calling ROM functions is described in the *High-Speed Microcontroller User's Guide: DS80C400 Supplement*.[1]) Parameters must be converted from the Keil C Compiler's conventions to the conventions used by the ROM. The Keil compiler passes parameters in a combination of XDATA locations and registers. The ROM functions accept parameters in different ways. For example, the socket functions accept parameters stored in a single parameter buffer, and many utility functions accept parameters passed in special function registers or direct memory locations. Dallas Semiconductor wrote libraries for accessing the ROM functions to translate from Keil calling conventions to the ROM's parameter conventions.

Using ROM functions in your C programs requires only importing the library and including a header file. To import a library in your project, right click on *Source Group 1* in your Keil project window and select *Add Files to Group 'Source Group 1.'* Change the file filter to '*.lib' and select the library you need to include. Then include the header file at the top of your source. You can use any of the library functions. There are ROM libraries to support ROM initialization, DHCP client operations, process management, socket functions, TFTP client operations, and utility functions such as CRC and pseudo-random number generation.

Using the extension libraries

In addition to the ROM libraries, other libraries (more are still being written) provide useful functionality not included in the ROM. Libraries have been developed for file system operations, DNS lookups, I²C™ communication, and 1-Wire® communication.

The C Library project (including documentation, sample applications, and release notes) for the DS80C400 can be found at www.maxim-ic.com/ds80C400/libraries.

A simple HTTP server and SNTP client application

Dallas Semiconductor wrote a small application to demonstrate the functionality of these libraries, specifically the file system, sockets, process scheduler, and TFTP libraries. The sample application consists of an SNTP client and an HTTP server that responds only to 'GET' requests. It uses the core Dallas Semiconductor-provided libraries to call socket and scheduler functions. It also uses the file system to store a few web pages. The application consists of two processes: (1) the HTTP server is spawned as a new process that handles connections on port 80, and (2) the main process sits in a loop, attempting a time synchronization approximately every 60 seconds. The source code and project files for this application are available at www.maxim-ic.com/timeserver.

[1] Available online at www.maxim-ic.com/DS80C400UG.

Initializing the file system

Compared to the TINI Runtime Environment, applications written in C allow more space for user code and data.

Before the HTTP server can be started, the file system must be initialized. The demonstration program ensures that two static files, a home page (*index.html*) and the source to the program (*source.html*), are in the file system before the server starts.

The program initializes its file system by downloading the files it needs from a TFTP server. In our example, a TFTP server is running at a known IP address. The files *index.html* and *source.html* are requested from the TFTP server, then written to the file system.

SolarWinds provides a free TFTP server for Windows® platforms that was used in the development of this demonstration. From SolarWinds' website (www.solarwinds.net), follow the **Downloads—Free Software** menu to find the TFTP server download. After installing, use the **Configure** option under the **File** menu to configure the available files. Make sure to change the program to use your TFTP server's IP address (`TFTP_IP_MSB, TFTP_IP_2, TFTP_IP_3`, and `TFTP_IP_LSB`).

The simple HTTP server

The HTTP server in this application is implemented as a simple version of an HTTP server described by RFC 2068. In this version, only the 'GET' method is supported. Input headers are ignored, and few output headers are given.

The server socket is created by calling Berkley-style socket functions, which make the server socket easy to set up. The following code shows how our simple HTTP server creates, binds, and accepts new connections.

```
struct sockaddr local;
unsigned int socket_handle, new_socket_handle, temp;

socket_handle = socket(0, SOCKET_TYPE_STREAM, 0);
local.sin_port = 80;
bind(socket_handle, &local, sizeof(local));
listen(socket_handle, 5);

printf("Ready to accept HTTP connections...\r\n");

// here is the main loop of the HTTP server
while (1)
{
    new_socket_handle = accept(socket_handle,
&address, sizeof(address));
    handleRequest(new_socket_handle);
    closesocket(new_socket_handle);
}
```

Note that when a new socket is accepted, this simple application does not start a new thread or process to handle the request. Rather it handles the request in the same process. Any HTTP server of more-than-demonstration quality would handle the incoming request in a new thread, allowing multiple connections to occur and be handled simultaneously. After the request is handled, close the socket and wait for another incoming connection.

The `handleRequest` method consists of parsing the incoming request for a file name and verifying that the method is 'GET.' No other method (not even 'POST,' 'HEAD,' or 'OPTIONS') is allowed. Two file names are handled as a special case. When the file **time.html** is requested, the server dynamically generates a response consisting of the latest results from the timeserver, and the number of seconds that passed since the last instance the timeserver was queried. When the file **stats.html** is requested, statistics for server uptime and the number of requests are displayed.

If the file is not found or an invalid request method is given, an HTTP error code is reported.

A datagram socket is first created and given a timeout of about 2 seconds (0x800 = 2048ms). This ensures that if the communication fails with our chosen server, we will not wait indefinitely for a response.

The SNTP client

The second major portion of the timeserver application is a Simple Network Time Protocol (SNTP) client, as described in RFC 1361. This is a version of the Network Time Protocol (RFC 1305). SNTP requires UDP communication to request a time stamp from a server listening on port 123. Our timeserver uses the following code to periodically synchronize with the server time.nist.gov. Note that when this article was written, DNS lookup was not supported, so the IP address for the server is set manually. DNS has since been added to the C library website, and the following code can be updated to perform a lookup for the IP address.

```
     socket_handle = socket(0, SOCKET_TYPE_DATAGRAM, 0);

     // set a timeout of about 2 seconds.
     //'timeout' is unsigned long
     timeout = 2000;
     setsockopt(socket_handle, 0, SO_TIMEOUT, &timeout, 4);

     // assume 'buffer' has already been cleared out
     buffer[0] = 0x23;  // No warning/NTP Ver 4/Client

     address.sin_addr[12] = TIME_NIST_GOV_IP_MSB;
     address.sin_addr[13] = TIME_NIST_GOV_IP_2;
     address.sin_addr[14] = TIME_NIST_GOV_IP_3;
     address.sin_addr[15] = TIME_NIST_GOV_IP_LSB;
     address.sin_port = NTP_PORT;
     sendto(socket_handle, buffer, 48, 0, &address,
sizeof(struct sockaddr));
     recvfrom(socket_handle, buffer, 256, 0, &address,
sizeof(struct sockaddr));
     timeStamp = *(unsigned long*)(&buffer[40]);
       timeStamp = timeStamp - NTP_UNIX_TIME_OFFSET;
       // now we have time since Jan 1 1970
       formatTimeString(timeStamp, "London",
last_time_reading_1);
       last_reading_seconds = getTimeSeconds();
       closesocket(socket_handle);
```

A datagram socket is first created and given a timeout of about 2 seconds (0x800 = 2048ms). This ensures that if the communication fails with our chosen server, we will not wait indefinitely for a response.

The next line sets the options for the request. These bits are described in Section 3 of RFC 1361. The value 0x23 requests no warning in case of a leap second, requests that NTP version 4 be used, and states that the mode is 'Client.' After we send the request and receive the reply using the common datagram functions `sendto` and `recvfrom`, the seconds portion of the time stamp value is assigned to the variable `timeStamp`, and then adjusted to the reference epoch January 1, 1970. The function `formatTimeString` is used to convert the time stamp into a readable string, such as **"In London it is 15:37:37 on March 31, 2003."**

The function `getTimeSeconds` is used to determine the last time update, based on the DS80C400's internal clock. Since the program only updates about once every 60 seconds, the HTML page *time.html* uses this value to report the interval since the last time update. Finally, the socket is closed and the SNTP client goes to sleep for another 60 seconds.

Conclusion

The Keil C Compiler and libraries provided by Dallas Semiconductor allow applications written in C to access the power and functionality formerly only accessible through TINI's Java environment. Programs written in C can now access the network stack, memory manager, process scheduler, file system, and many other features of the DS80C400 networked microcontroller. Additionally, applications written in C allow more space for user code and data, compared to the TINI Runtime Environment. Developers using the C language for the DS80C400 can write lean applications with plenty of speed, power, and code space to tackle any problem.

Developers using the C language for the DS80C400 can write lean applications with plenty of speed, power, and code space to tackle any problem.

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 ____________________

Micro ER 2/04

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

DALLAS SEMICONDUCTOR MAXIM

www.maxim-ic.com

Maxim Integrated Products, Inc.
120 San Gabriel Drive
Sunnyvale, CA 94086

Presorted Standard
U.S. Postage
PAID
Greenfield, OH
Permit No. 270