Application Note 707
Using the DS80C400 to Maximize System Performance

www.maxim-ic.com

OVERVIEW

Applications originally destined for 16-bit or 32-bit microprocessors are making their way into the 8-bit realm. With this move, new I/O capabilities are required, such as communicating using TCP/IP network protocol over Ethernet. Because of the TCP/IP protocol stack complexity and multiple simultaneous connection ability, a multiprocess OS is a necessity for efficient operation. IP packets require extensive processing on transmit and receive to build headers and checksum data, and each connection requires a unique state machine to be maintained. These requirements increase the CPU load on the micro, taking away CPU resources from the critical application. Any reduction in protocol CPU use improves the performance of the primary application.

Overall microprocessor system performance usually depends heavily on a few operations or function calls. The DS80C400 contains several functional blocks that help the system designer speed up the code paths that tend to be the bottleneck of system performance. This application note explores the features of the DS80C400 in detail and provides usage examples.

IMPROVED FEATURES

The DS80C400 is a step above its DS80C390 predecessor. It includes the features of the DS80C390 as well as the features of the high-speed family of Dallas microcontrollers.

FEATURES	HIGH-SPEED µC	DS80C390	DS80C400
4-Clock Cycle Core	✓	✓	✓
Second Data Pointer	✓	✓	✓
Low-Power Modes	✓	✓	✓
Watchdog Timer	✓	✓	✓
Power-Fail Circuitry	✓	✓	✓
Additional Serial Port	✓	✓	✓
Math Accelerator		✓	✓
Automatic Increment Data Pointer Select		✓	✓
1k Extended Stack		✓	✓
24-Bit Memory Addressing		✓	✓
3.3V/5V Tolerant I/O			✓
Max Frequency of 75MHz vs. 40MHz for DS80C390			✓
Built-In Ethernet MAC			✓
Auto Increment/Decrement Data Pointers			✓
Two Additional Data Pointers			✓
Optimized Single Increment/ Decrement Data Pointer			✓
Hardware TCP/IP Checksum Generator			✓
Third Serial Port			✓
1-Wire® Master			✓

1-Wire is a registered trademark of Dallas Semiconductor Corp.

Code Examples

Memory Copy/Stack Save

One of the most basic routines in microcontroller software development is the memory copy function. Speeding up memory copies using the automatic increment data pointer is described In *Application Note 604: Fast Memory Transfers with the Ultra-High-Speed Microcontroller*. This functionality also applies to stack save/restore on a multiprocessing OS to lower the latency of the context switch. Lower latency increases responsiveness of the entire system and reduces the amount of CPU time spent in OS overhead.

In the unoptimized example below, the loop pops 1 byte off the stack, stores it to RAM, and increments the data pointer. By using the auto increment instruction in the optimized example, the `inc dptr` can be removed from the critical path, resulting in a speed improvement of 42%.

Stack save before optimization:

```
; dpl,dph points to location to save
stack
; Assume using internal 1K stack
        mov   R0,SP
stack_loop:
        ; Get stack byte
        pop   ACC
        ; Store stack byte
        movx  @dptr,A
        inc   dptr
        djnz R0,stack_loop
```

Machine cycles per loop, DS80C390: 10

Stack save after optimization:

```
; dpl,dph points to location to save
stack
; Assume using internal 1K stack
        mov   R0,SP
        ; Enable auto increment data
pointer
        orl   DPS,#010H
stack_loop:
        ; Get stack byte
        pop   ACC
        ; Store stack byte
        movx  @dptr,A
        djnz R0,stack_loop
        ; Disable auto increment data
pointer
        anl   DPS,#0EFH
```

Machine cycles per loop, DS80C400: 7

42.8% speed improvement

Even without changing existing code to take advantage of the special data-pointer operations, system speed can noticeably increase because the `inc dptr` instruction takes only one machine cycle on a DS80C400 instead of three on other Dallas high-speed microcontrollers. The simple, nonoptimized copy loop below is over 30% faster running on a DS80C400 compared to a DS80C390, or any other 8051 derivative.

```
copy_loop:
        movx  A,@dptr
        inc   dptr
        inc   dps
        movx  @dptr,A
        inc   dptr
        inc   dps
        djnz  R0,copy_loop
```

Machine cycles per loop, Dallas high-speed microcontroller: 17
Machine cycles per loop, DS80C400: 13
30.7% speed improvement

TCP/IP Checksum and Built-In MAC

The DS80C400 includes a built-in Ethernet MAC that simplifies network interfacing and removes the requirement for an external memory-mapped Ethernet controller. The MAC is accessed through an 8kB shared-memory interface that allows the use of the fast memory copies previously mentioned. Because the memory is internal, no stretch cycles are necessary, and memory accesses execute at full speed, giving a 1.8MBps transfer rate to and from the Ethernet controller. If Ethernet is not used, this memory is available for general-purpose use.

An additional network helper is the hardware TCP/IP checksum generator. All TCP/IP network headers and data must be checksummed, a time-consuming process in software. By offloading the checksum to hardware, the network throughput of the microprocessor increases drastically. The built-in checksum is accessed through a single special function register (SFR).

In the unoptimized example, each 16-bit word is added to the running checksum, and any carry is folded back into the result. The management of intermediate data and checking for carry consumes the majority of the cycles in the loop. In the optimized version, the hardware handles all the details, and all that is required is two writes to an SFR. **Note:** Auto-increment data-pointer functionality was left out of the optimized example to highlight the checksum optimization. If auto increment is added, the loop is two machine cycles faster.

TCP/IP Checksum code before optimization:

```
;***********************************
;* Function Name: ip_checksum
;*
;* Description: Generate IP One's
Complement Checksum.
;*
;* Input(s):
;* R4,R5   - Number of 16 bit words to
checksum
;* R0,R1   - running checksum
;* dpl,dph - data to checksum
;*
;* Outputs(s):
;* R0,R1 – running checksum
;***********************************
next_word:
          movx  A,@dptr
; Read high byte of word.
          mov   b, a
          inc   dptr
          movx  A,@dptr
; Read low byte of word.
          inc   dptr
          add   a, r0
; Add to low byte of sum.
          mov   r0, a
          mov   a, b
; Get high byte of word.
          addc  a, r1
; Add it to high byte of sum.
          mov   r1, a
          jnc   ip_ics_no_carry

          mov   a, r2
          addc  a, #0
; Deal with carry.
          mov   r2, a
          jnc   ip_ics_no_carry

          mov   a, r3
          addc  a, #0
; Another possible carry.
          mov   r3, a

ip_ics_no_carry:
          djnz  r4, next_word
          djnz  r5, next_word
```

Machine cycles per loop, DS80C390: 24–35

84.6% to 169.2% speed improvement

TCP/IP Checksum code after optimization:

```
;***********************************
;* Function Name: ip_checksum
;*
;* Description: Generate IP One's
Complement Checksum.
;*
;* Input(s):
;* R4,R5   - Number of 16 bit words to
checksum
;* R0,R1   - running checksum
;* dpl,dph - data to checksum
;*
;* Outputs(s):
;* R0,R1 – running checksum
;***********************************
next_word:
          movx  A,@dptr
; Read high byte of word.
          inc   dptr
          mov   OCAD,A
; Load One's Complement Adder
          movx  A,@dptr
; Read low byte of word.
          inc   dptr
          mov   OCAD,A
; Load One's Complement Adder
          djnz  r4, next_word
          djnz  r5, next_word
```

Machine cycles per loop, DS80C400: 13

Additional Data Pointers

The DS80C400 provides two additional data pointers and an additional data-pointer select SFR, DPS1, for ease of pointer management in large applications. This pair of data pointers has the same features as the original pair and is selected using the SEL1 bit in DPS.

In the Xor_Strings example below, three pointers are necessary, two input and one output. Managing these pointers requires saving an input pointer, restoring the output pointer, saving the output pointer, and finally restoring the input pointer. The nonoptimized example requires that one of the two built in pointers be saved on the stack, output pointer loaded in place of input pointer, output pointer incremented, output pointer saved, and the original pointer restored from the stack. This time consuming operation must be performed on each iteration of the loop. The optimized example uses dpl2 and dph2 to store the output pointer. No state must be saved, and only one bit of state is modified in DPS to allow access to the new data pointer.

```
;**********************************
;* Function Name: Xor_Strings
;*
;* Description: XOR String A with
String B and place result in String C
; C = A XOR B
;*
;* Input(s):
; R0        - Number of bytes in
string
; dpl,dph   - String A
; dpl1,dph1 - String B
; R4,R5     - String C
;*
;* Outputs(s):
;* None.
;**********************************
Xor_Strings:
        ; Get A
        movx  A,@dptr
        mov   B,A
        inc   dptr
        inc   dps
        ; Get B
        movx  A,@dptr
        inc   dptr
        inc   dps
        xrl   A,B
        push  dpl
        push  dph
        mov   dpl,R4
        mov   dph,R5
        ; Write C
        movx  @dptr,A
        inc   dptr
        mov   R4,dpl
        mov   R5,dph
        pop   dph
        pop   dpl
        djnz  R0,Xor_Strings
        ret
```

Machine cycles per loop: 36

```
;**********************************
;* Function Name: Xor_Strings
;*
;* Description: XOR String A with
String B and place result in String C
; C = A XOR B
;*
;* Input(s):
; R0        - Number of bytes in
string
; dpl,dph   - String A
; dpl1,dph1 - String B
; dpl2,dph2 - String C
;*
;* Outputs(s):
;* None.
;**********************************
Xor_Strings:
        ; Get A
        movx  A,@dptr
        mov   B,A
        inc   dptr
        inc   dps
        ; Get B
        movx  A,@dptr
        inc   dptr
        inc   dps
        xrl   A,B
        orl   DPS,#008H
        ; Write C
        movx  @dptr,A
        inc   dptr
        anl   DPS,#0F7H
        djnz  R0,Xor_Strings
        ret
```

Machine cycles per loop: 24

33% speed improvement

1-Wire Master

Another unique addition is the 1-Wire Master. This hardware block takes the timing generation and state machine management complexity out of 1-Wire interface software. The block is set to run off a divide of the CPU clock, and is able to generate accurate 1-Wire time slots throughout the clock frequency range of the DS80C400 (1MHz–75MHz). Using the 1-Wire master reduces CPU load, speeds 1-Wire communication, simplifies software development, and reduces code size. In a pure software implementation of a 1-Wire bit, care must be taken to turn off interrupts during the critical timing portions of the 1-Wire timeslot. In the case of a read or write 0, this interrupt off time can be as long as 60µs. This translates to CPU dead time, where no useful code is executed, and no interrupts are allowed to run. On the other hand, the hardware generation of 1-Wire bits is not timing critical, interrupts may run, and CPU dead time is reclaimed for more important tasks.

The hardware generation of a 1-Wire bit involves enabling single bit mode, writing the data to transmit, polling the transmit complete flag, and unloading the receive data. The code below demonstrates the generation of a single 1-Wire bit. For comparison, see Appendix A for a non-hardware-based 1-Wire master.

```
;***************************************************************************
;* Function Name: OWM_Bit
;*
;* Description: Generate a 1-wire bit and return the result.
;*
;* Input(s):
;* acc.0 -> transmit bit
;*
;* Outputs(s):
;* acc.0 -> receive bit
;***************************************************************************
OWM_Bit:
        mov   OWMAD,#OWM_CONTROL                 ; Change to single bit mode
        orl   OWMDR,#OWM_BIT_CTL_MASK            ; "

        mov   OWMAD,#OWM_TRANSMIT_BUFFER         ; Send a single 1-Wire bit
        mov   OWMDR,A                            ; "

        mov   OWMAD,#OWM_INTERRUPT_FLAGS         ; Look at the flags
OWM_Bit_wait:
        mov   A,OWMDR                            ; Wait for end of bit command
        jnb   OWM_RBF_BIT,OWM_Bit_wait           ; "

        mov   OWMAD,#OWM_RECEIVE_BUFFER          ; Get the result of the single
bit
        mov   A,OWMDR                            ; "

        mov   OWMAD,#OWM_CONTROL                 ; Change to byte mode
        anl   OWMDR,#NOT(OWM_BIT_CTL_MASK)       ; "

        ret
```

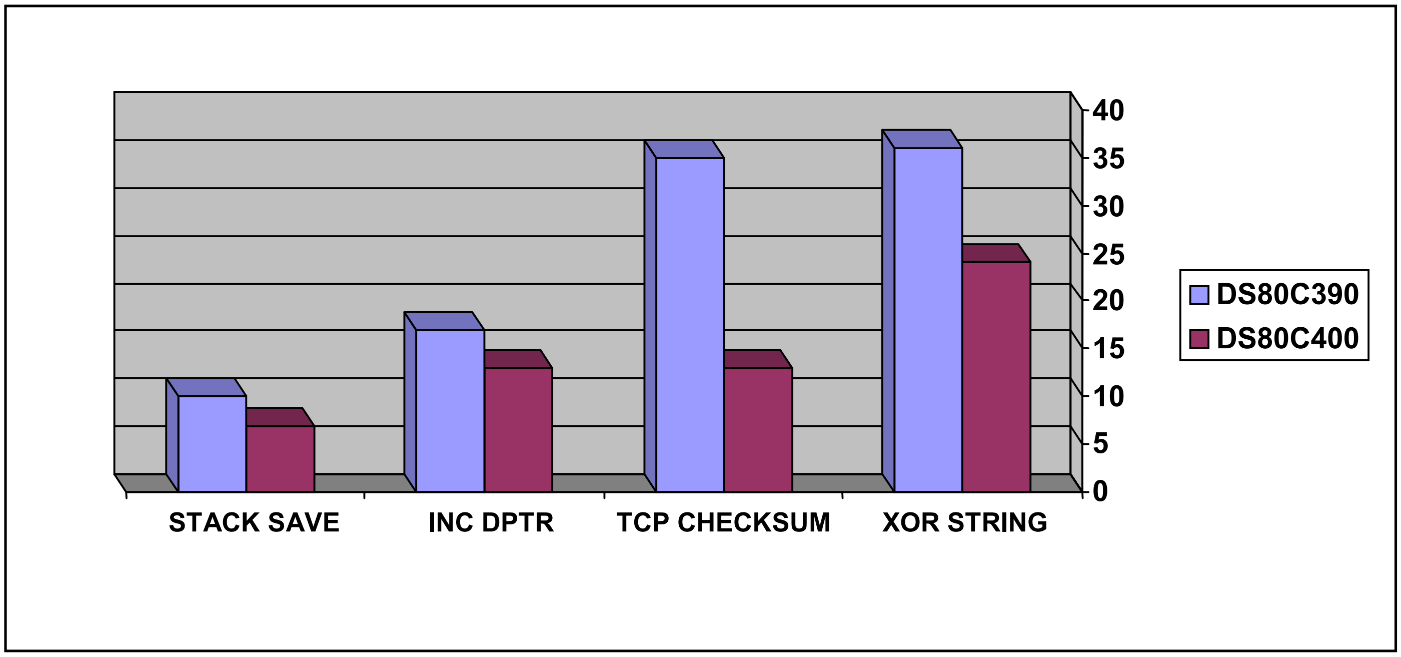
Example Performance Results: TINI

An example of a software system that takes advantage of the DS80C400 features is the TINI® Runtime Environment (TRE). The TRE is a platform developed to provide system designers with a flexible and cost effective means to bridge the gap between hardware devices and networks. This system provides a TCP/IPv4/6 network stack, I/O drivers, task scheduler, and a Java™ Virtual Machine to glue everything together. The TRE runs on both the DS80C390 and DS80C400, with minor code modifications to take advantage of the improved features of the DS80C400. The results of these optimizations improve network speed and latency, context switch latency, memory mapped moves, 1-Wire CPU utilization, and Java code execution speed. Table 1 shows typical performance numbers for the TRE on each microcontroller.

Table 1. Typical Performance Numbers for the TINI Runtime Environment

TINI RUNTIME ENVIRONMENT	**DS80C390 (36.864MHz)**	**DS80C400 (36.864MHz)**
TCP Transmit (bytes/s)	133,000	266,240
TCP Receive (bytes/s)	117,000	252,900
UDP Transmit (bytes/s)	160,000	268,200
UDP Receive (bytes/s)	140,000	268,200
TCP Latency (ms)	7.2	6.2
Memory Copy Bandwidth (bytes/s)	655,000	1,875,000

Figure 1. Selected Performance Results



Conclusion

The additional hardware performance enhancements and functionality of the DS80C400 can boost overall system speed by more than 30%. Applications on existing 8-bit microprocessors can be moved to the DS80C400 to free up CPU cycles, add new features, and avoid being forced to use a larger, more expensive processor. Minor software modifications, as described in the examples above, allow existing code to take advantage of the accelerated data pointer hardware support, giving any application added speed.

With its low-power consumption, low-voltage core, standard 8051 memory interface, compatibility with existing software tools, and on-board peripherals, the DS80C400 is an attractive target for existing and new system development.

Appendix A: 1-Wire Bit Bang Example

Notes:

- BURN_MS is a macro function call that burns 1µs of time at the clock rate of the microprocessor.
- P3.5 is used as the 1-Wire I/O line

```
;*********************************************************************
;* Function Name: OW2_Bit_Regular
;*
;* Description: Generate a 1-wire bit at regular speed
;*
;* Input(s):
;* acc.0 - transmit bit
;*
;* Outputs(s):
;* acc.0 - receive bit
;*********************************************************************
OW2_Bit_Regular:
        mov     R1,A                    ; Save transmit bit.

        push    IE
        clr     EA
        clr     P3.5                    ; Start time slot.
        BURN_MS                         ; Wait 5µs.
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        mov     A,R1                     ; Restore transmit bit.
        mov     C,ACC.0
        mov     P3.5,C
        BURN_MS                         ; Wait 10 more microseconds
        BURN_MS                         ;    before sampling the bus.
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        BURN_MS
        mov     C,P3.5                ; Sample the bus as close to 15µs as possible.
        mov     ACC.0,C
        mov     R2,A                  ; Save receive bit.

        mov     A,R1                            ; Check to see if we're doing a write zero.
        jb      ACC.0,OW2_Bit_finish_quick      ; If we are not, we can shorten the
                                                 ; time slot.

        mov     R0, #10H                        ; Wait 48µs more for end of time slot.
OW2_Bit_finish:
        BURN_MS                                 ; Wait out the time slot.
        BURN_MS
        BURN_MS
        djnz    R0,OW2_Bit_finish

        setb    P3.5                            ; Restore 1-Wire to idle state.
        pop     IE                              ; Restore interrupt state
        sjmp    OW2_Bit_exit

OW2_Bit_finish_quick:
        pop     IE                    ; We can enable interrupts here, as the
                                      ; one wire will be coming up on its own.

        mov     A,R2
        jb      ACC.0,OW2_Bit_finish_one

        mov     R0,#080H                        ; Wait 48µs more for end of time slot,
                                                 ; or onewire coming back high.
OW2_Bit_finish_quick_loop:
        BURN_MS                                 ; Wait out the time slot.
        mov     C,P3.5
        jc      OW2_Bit_exit                    ; If the line comes high, exit.
        djnz    R0,OW2_Bit_finish_quick_loop
        sjmp    OW2_Bit_exit

OW2_Bit_finish_one:
        mov     R0,#010H
OW2_Bit_finish_one_loop:
        BURN_MS                                 ; Wait out the time slot.
        djnz    R0, OW2_Bit_finish_one_loop

OW2_Bit_exit:
        BURN_MS                                 ; Do 1us bus recovery.

        mov     A,R2                            ; Restore receive bit.

ret
```