

# Motorola Semiconductor Application Note

---

## AN1711

## DMA08 Systems Compatibilities

By **Bill Getka**  
CSIC Design Engineering  
Austin, Texas

### Introduction

---

The direct memory access (DMA) module for the HC08 Family architecture (DMA08) provides numerous system functions. Some functions are germane to the fact that it is a DMA, such as the ability to do efficient block transfers, whereas other functions are not as obvious, like the ability to service module interrupts without having to exit the CPU from low-power mode.

To demonstrate the advantages of using the DMA, this application note illustrates many of the system capabilities the DMA offers through a single code example that has the DMA simultaneously servicing three separate module interrupts while the CPU is either doing other work or is in a lower-power mode.



The DMA's main features that are highlighted in this application note are:

- Ability to service a block transfer to a module while the CPU does something else concurrently
- Ability to do either a block memory transfer under software control or to be interrupt driven under module control (doing a block 1 byte at a time as requested by a module).
- Ability of DMA and other modules to operate in wait mode where the address bus and data bus are inactive except when DMA transfers occur
- Other miscellaneous DMA features, including byte mode versus word mode, loop mode versus no loop mode, and selectable interrupt creation at the end of block transfers

Although many of the DMA capabilities are explained in detail in this application note, a complete description of the DMA08 module is not contained here. Refer to the *DMA08 Direct Memory Access Reference Manual* (Motorola order number DMA08RM/AD) for a complete description of the module's functionality. The reference manual also contains numerous application examples describing how to use the DMA to do specific tasks such as software-initiated block transfers and service of serial communication and a timer. In each case, performance improvements achieved by using the DMA versus not using the DMA are noted. In addition to this reference, the general release specification of every microcontroller unit (MCU) containing a DMA has a specific chapter describing its functionality.

## Advantages of the DMA

---

Using a DMA in a system has many advantages. These advantages can be seen by analyzing a typical interrupt service routine. Instead of having to stack the registers, check and clear interrupt flags, manage data and corresponding pointers, and unstack each time a simple interrupt occurs, the DMA in two cycles can service many types of interrupt requests. This is particularly valuable if the MCU is kept primarily in wait mode except when servicing interrupts. This reduction in bus activity from dozens of cycles to two can add up to significant power savings. Even when the MCU is not in wait mode, this improved efficiency frees the CPU to handle other tasks. Efficient interrupt servicing also means that other pending interrupts can be serviced more quickly, thereby reducing the overall system latency.<sup>1</sup> And, since the DMA can be programmed easily to implement complex queuing functions, its usage will many times reduce code size and complexity. Still other advantages are possible, and these will be highlighted later in the application note. This short list is meant to convince the reader that exploring usage of the DMA is worthwhile.

---

1. See the appendices of the DMA08 reference manual for some examples of cycle efficiency gained when using the DMA. Some specific examples related directly to this application code appear later as well.

### DMA Operation

---

In the HC08 architecture, the DMA08 acts as a secondary bus master to the CPU. The DMA has the ability to steal a programmable percentage of bus cycles from the CPU to do memory-to-memory transfers. These transfers can be between any two addressable memory locations, such as RAM, ROM, module data registers, and port data registers. Each DMA channel has a separate 16-bit source and destination address pointer that can be programmed to increment, decrement, or remain static with each byte transferred. In addition, each channel has a block length register to control the total number of bytes transferred and a byte count register to keep track of how many bytes have been transferred. Other control registers dictate whether block transfers are repeated (loop mode), whether a CPU interrupt occurs at the end of each block, whether the DMA can be active in wait mode, and other transfer properties. With this flexibility, the DMA can implement customized queuing functions to service module data requests or transfer data blocks in an efficient manner.

A DMA transfer can be initiated by either a hardware module (such as a serial communications interface (SCI) or analog-to-digital (A/D) converter) or by the software directly. In the case of hardware initiation, a given module will have a control bit that indicates whether the generated interrupt flag should be directed to the CPU or DMA. Typical interrupt flags might indicate a transmit buffer is empty or a receive buffer is full. When configured for DMA servicing, this module's interrupt flag will cause the DMA to do a transfer with the appropriate register. That is, a write of data might be done to service a transmit empty request whereas a read might be done to service a receive buffer full request. This read or write access automatically will cause the module to lower its interrupt flag to the DMA. Once the next byte is needed, the module reasserts its interrupt until the transfer is complete again. This simple handshaking protocol continues until the DMA is disabled, which is usually when the block length is reached.

A hardware-initiated DMA example could be a given program that is intended to transfer 50 bytes of data out of the SCI while simultaneously receiving 30 bytes. Two DMA channels could be used to manage these transfers, one for the transmission and one for the reception. To handle data transmission, the first DMA channel's source register would be set to the beginning of a buffer containing the 50 bytes of data to be sent. Its destination register would be set to the SCI data register, and its block length register would be set to 50. The addressing modes of the pointer registers would be set such that the source address would increment with each byte transferred while the destination address would remain static. Once both the SCI and DMA were enabled properly, the SCI transmitter empty interrupt to the DMA would cause the DMA to do a read from the data buffer followed by a write of that data to the SCI data register. This write action would clear the request to the DMA. The next transmitter empty request would cause the next byte in the buffer to be transferred until all 50 were finished. Once the byte count was reached, the DMA could have been configured to disable this channel automatically and interrupt the CPU to indicate the end of the transfer. The reception of data would occur on the second channel in much the same way. In this case, the source address is set statically to the SCI data register, and the destination is an incremented buffer pointer. Here transfers are caused by the receiver full flag, which is cleared by reads from the data register. Again, the transfer completes when the byte count is reached or no more receiver full flags are set.

In a software-initiated transfer, a bit is set in a DMA control register to initiate the transfer. The transfer involves doing a read from the source location and a write to the destination location for each byte of the block. Again, different addressing modes allow the source and destination pointers to increment, decrement, or remain static. The transfer runs at the programmed bandwidth until complete unless it is interrupted by a higher priority channel. The fundamental difference between a hardware-initiated transfer and a software-initiated transfer is that a hardware transfer typically progresses one byte (or word) at a time at a rate dictated by the module's needs, whereas a software transfer typically does a full block transfer at a programmable rate without interruption.

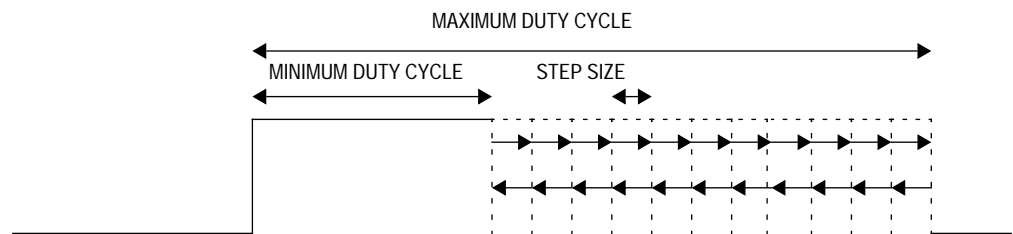
As previously mentioned, the DMA is very flexible. It has various controls that allow the user to customize the DMA transfer properties to best suit the application. The bandwidth control allows the user to select what percentage of bus cycles an active DMA transfer uses. Choices are 100%, 67%, 50%, or 25%. A separate control determines whether or not a CPU interrupt can interrupt an active DMA transfer. This same control assigns the priority levels to the existing channels. Yet another control determines if a channel is disabled at the end of a transfer or allowed to loop on the block. A related control determines if the end of a block can cause a CPU interrupt. A different control determines if each transfer is a byte long or a word long, and another determines if transfer can happen in wait mode.

A final control maps potential interrupt sources to available channels. For instance, the HC08XL36 has eight sources that can cause DMA interrupts, but it has only three DMA channels. Each channel has a 3-bit field in a control register which maps a given source to that channel. This mapping capability enables the system to use all available hardware efficiently. If a given channel is meant for software transfers, then this channel has to be mapped to an unused DMA interrupt source. See the DMA08 reference manual for a precise description of all available control registers, as well as a step-by-step programming sequence to use to configure these registers for the desired results.

## Overview of the Application Code

---

The code written for this application note is meant to tie several features provided by the DMA into a concrete example around which the user can develop his own applications. This sample application produces a simple PWM waveform that can be viewed on an oscilloscope. The PWM duty cycle varies from a user-selectable minimum to maximum at a user-selectable step size. (See [Figure 1.](#)) Increasing the step size makes the waveform appear to move more quickly between the minimum and maximum values. To get a good view of the waveform, configure the oscilloscope to display 2 volts/div and 10 ms/div with positive edge triggering.



**Figure 1. Example of PWM Waveform Produced on Oscilloscope**

The PWM waveform is supplied by the output compare function of the TIM08 module in conjunction with the DMA. The various pulse width values come from a RAM table via the DMA. For the waveform to move slowly enough to be seen, the timer is clocked externally. This external clock actually is provided by wiring the timer's external clock input to the SPI's MOSI output. (See [Figure 6](#) on page 43.) With the help of the DMA, the SPI is sending \$OF at its slowest baud rate continuously, which effectively creates a slow-running clock.

In addition to the waveform, the application also provides a user interface. This user interface communicates through a standard serial communication interface (SCI) module at 9600 baud, in conjunction with an RS-232 chip to a dumb terminal. Keystrokes entered on the dumb terminal direct changes in the output waveform. A typical screen display of the dumb terminal is shown in [Figure 2.](#)

Currently generating a waveform that varies from a duty cycle of 25 to 75 at a step size of 1.

Please choose which you would like to alter.

Would you like to change

- 0) back to the default values
- 1) the minimum duty cycle value
- 2) the maximum duty cycle value
- 3) the step size of the change in duty cycle

? 1 ←————— 1 entered by user.

Please enter the minimum duty cycle [must be an integer between 10 and 75--the current maximum duty cycle]: 50 ← 50 entered by user.

Currently generating a waveform that varies from a duty cycle of 50 to 75 at a step size of 1.

Please choose which you would like to alter.

**Figure 2. Example Display of User Interface**

The messages printed are built up in a RAM buffer. The DMA brings pieces from ROM data structures into RAM under CPU control. Once an entire message is constructed, it is broadcast out of the SCI module under DMA control. With this project overview in mind, the remainder of this application note describes specific code details that highlight advantages of using the DMA.



## Highlights of DMA Usage in Application Code

---

The first advantage that can be highlighted in the code is cycle efficiency. An example can be seen in the `initramsci` routine on page 30. Here the DMA is used to initialize the RAM's timer output compare buffer with a series of constants. The data structure used is configured such that every other byte in the table is the absolute maximum duty cycle. Therefore, the DMA was used to copy the decimal word 0099 into each entry in the buffer. This required 164 transfers at two cycles per transfer with 38 cycles of programming overhead, for a total of 366 cycles. **Figure 3** shows a code segment that accomplishes the same end result without DMA assistance. In this case, each word required 10 cycles plus an additional 13 bytes of overhead, for a total of 833 cycles. Hence, there was a significant cycle savings in the DMA approach even when the source data was a constant. A generic move routine would save an even higher percentage. (See the **Software-Initiated Block Transfer** section of the DMA08 reference manual.)

initramsci:	lda #maxbuf	
	lsra	; Do two bytes in each loop
	psha	
	ldhx #bufbegin	; Set dest addr to be buffer pointer
	lda #absmaxduty	; Value to fill in every other byte
nxtbyte:	incx	; Don't care what's in other byte
	sta ,X	; Put constant in this byte
	incx	; Point at next byte to store
	dbnz 1, sp, nxtbyte	; Check for end of block fill
	pula	; Remove byte count from stack

**Figure 3. Code to Fill RAM Buffer without the DMA**

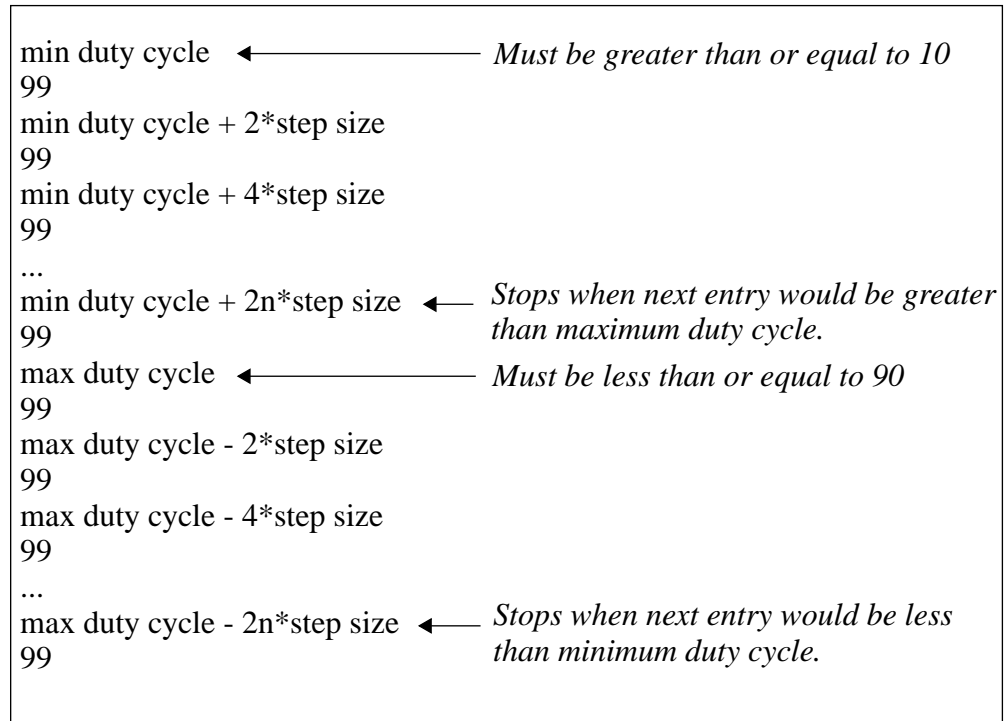
Another feature highlighted by this same code segment is the ability of the DMA channel to be initiated under software control. By writing a binary 4 to the lower three bits of the D2C register, the program maps interrupt source 4 (of 8) to channel 2. After channel 2 is enabled properly, setting bit 4 in the DC2 register begins the transfer just as if the hardware module connected to that source had asserted its interrupt.

Another point to be highlighted about this code segment is how transfers can be done on a word basis instead of on a byte basis. The advantage here is that the desired effect is to transfer a number of 2-byte constants to a buffer. Therefore, the source address had to vary over two locations while the destination address was to increment over 164 locations. By using word mode, the source addressing mode was declared as static while the destination mode was incremented, and the byte count was set to the number of bytes to transfer. This would not have been possible if only byte mode was available because only one of the two bytes of the word would have been chosen. Word mode is equally important when dealing with 16-bit registers (such as the timer's channel registers) for this same reason. Note that word mode transfers will use 100% of the bandwidth regardless of the bandwidth register's setting. The bandwidth setting only applies to byte mode. (See Appendix B of the DMA08 reference manual.)

Another section of code to notice is the start waveform (`srtwvfrm`) routine on page 31. The first part of this routine programs DMA channel 0 to service SPI transmitter empty interrupts. Recall that the SPI is being used (wastefully) to provide a continuous clock output. To achieve this, the source address points at a constant byte (`$OF`) while the destination is the SPI data register. The key feature is that the DMA is put in loop mode so that the transfer continues to occur until the DMA is disabled manually by the CPU. Loop mode not only works on a single byte, but it also can work on a block as well. When the DMA traverses through a block, neither the value of the source nor the destination registers change, even in increment and decrement addressing modes. Instead, the DMA uses a dedicated ALU to add the current byte count to the address pointers to allow them to remain constant. Once the block length is reached, the byte count is simply reset to 0 to start the next loop. An example of this type of multi-byte loop is demonstrated in the `srtwvfrm` routine using TIM channel 0 and DMA channel 1.

Although the DMA usually simplifies the overall application, there are times when slightly different methods need to be employed to make DMA usage possible. The PWM creation highlights this point.

To create the varying PWM output, DMA channel 1 feeds TIM channel 0 with a different value each time a period is complete. The structure of this RAM table is shown in [Figure 4](#).



**Figure 4. RAM Buffer Data Structure for the PWM**

The timer's overflow register is set to 100. Each time the timer hits this value, it toggles its output high. Timer channel 0 is set up such that each time the value in its channel register is reached, it lowers its output and requests a new value from the DMA. The RAM buffer is set up such that the next value fetched is less than the overflow value but greater than any valid table entry so that exactly two table entries are fetched in one period. This is necessary to prevent PWM values from being missed. For instance, if a table entry of 50 were followed by a table value of 60, the output would be cleared when 50 was reached, the value of 60 would be fetched, and the pin would be cleared again when 60 was reached with the next value fetched. This would effectively cause the value of 60 to be skipped. By placing a value of 99 as the next table entry after every valid entry, it is guaranteed that the timer counter is higher than the next value fetched, thereby preventing the false edge. Since reaching the value of

99 in the timer register only causes the already low pin to be cleared, this extra fetch goes unnoticed by the user. Instead, it serves to fetch the next valid entry for the next period.

A final functional aspect of the application worth noting involves how messages are transferred to the user. Many of the messages to the user involve giving status on current waveform properties and asking for new valid entries. These strings, therefore, are made up of constant string messages with occasional integer variable values thrown in (for instance, current max duty cycle, current step size, etc.). To form an entire message, a number of strings and variable values are put together. A full message is built in the RAM buffer by the CPU and, when ready, the DMA transmits it to the SCI. The CPU uses the DMA to transfer message segments from ROM to RAM. This process is shown in the `strxfr` routine found on page 37. If an integer variable value needs to be appended to the end of the string as the message forms, the CPU goes off and begins calculating the ASCII value of that integer at the same time the DMA is transferring the preceding string (see the `h2axfr` routine found on page 36). Since the CPU knows the length of the string, it just places the ASCII result at the end of the RAM segment being filled by the DMA. Once the CPU is finished, it checks to see if the DMA is done. If it is not done, the CPU goes into wait mode until the DMA is done so it can proceed with the next segment (see the `waitdma2` routine found on page 38). This process repeats until the entire string is built, at which point the DMA can start sending bytes of data to the SCI for transmission (see the `xmitstr` routine found on page 38). While this transfer occurs, the CPU either calculates buffer values or goes back into wait. Meanwhile, the DMA simultaneously is servicing the SPI, SCI, and TIM modules.

In wait mode, the address and data buses of the entire MCU are static and many clocks are disabled to reduce power consumption. Enabled modules can continue running. Also the DMA can run in wait mode, transferring data using the otherwise static address and data buses without waking the CPU. Block complete interrupts from the DMA are used to wake the CPU at the end of a transfer. Such an interrupt is used in this case to let the CPU know that the prompt sent to the user via the SCI transfer is complete so the CPU can begin looking for a reply.

As the `waitdma2` routine shows, wait entry must be done carefully. In this case, there is only one system interrupt that can pull the MCU out of wait, the DMA block complete interrupt. Since both block lengths and the amount of work done by the CPU before entering wait mode varies, it is possible for a DMA transfer to end at about the same time the CPU is ready to enter wait mode. In a worst case scenario, the DMA would interrupt the CPU right before the WAIT instruction was executed. The interrupt would be serviced, then the WAIT instruction would be executed. Since the DMA block complete interrupt already occurred, no interrupt would ever pull the MCU out of wait, thereby hanging the system. The code in the `waitdma2` routine makes it impossible for this scenario to occur. It begins by masking interrupts to ensure this critical interrupt cannot occur until after wait mode is entered. Then it verifies that the transfer is still in process. If it is, the necessary interrupt will still become pending at some point in the future. It will not be serviced until after the WAIT instruction is executed because all interrupts remain masked. The WAIT instruction itself clears the interrupt mask, thereby allowing this interrupt later to pull the MCU out of wait. So even if the DMA transfer ends between the check of the flag and the WAIT instruction, the interrupt will not fail to pull the MCU out of wait.

When transferring data to the user, the DMA is simultaneously servicing all three of the SCI, SPI, and TIM requests while the CPU is either doing other work or is in wait mode. Since none of these modules create interrupts very often, this leaves the bus idle for other activities or nothing at all. If the CPU were used to manage the various queues instead of the DMA, a far greater percentage of cycles would be required to maintain the waveform while increasing the latency of other CPU functions.

The remainder of the code is used to manage the user interface and fill the RAM buffer with PWM values calculated from user-selected parameters. This latter function is a good example where the DMA is not appropriate. Even though a large buffer needs to be filled, each byte needs to be calculated based on a simple algorithm. This can be done only by a CPU loop and cannot be done by the DMA because the DMA can only move data, not manipulate it.

This simple MCU system helps to illustrate the many reasons that a DMA can be useful. Although not designed to be used without modifications, many of the code segments can be adapted to real applications. DMA programming sequences and the wait entry protocol are particularly useful. A similar approach to PWM generation may be helpful also. Regardless of the amount of code reuse, the application note should serve to highlight how the DMA can improve the performance of many system solutions.

## Software Flowcharts

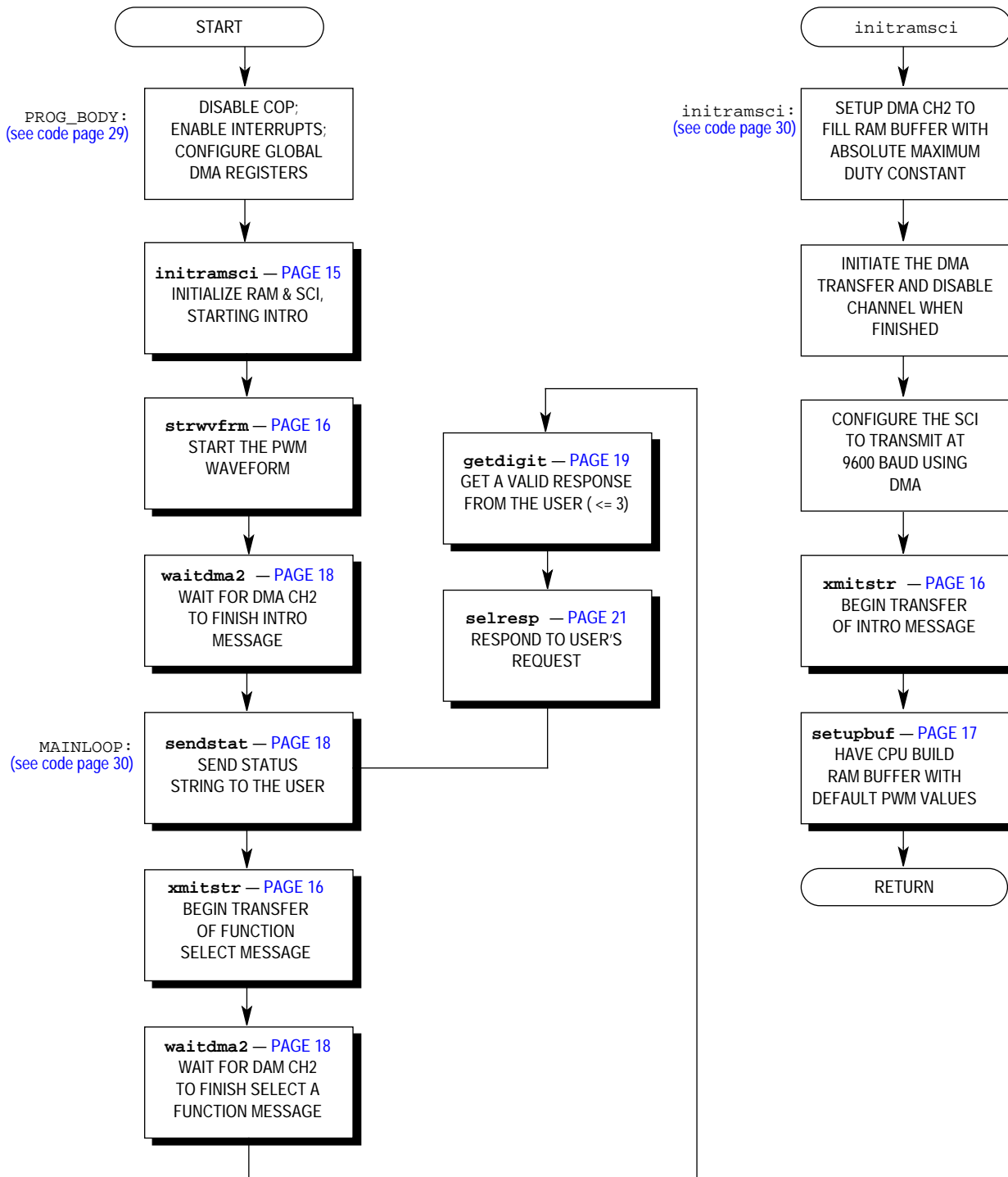


Figure 5. Software Flowchart (Sheet 1 of 13)

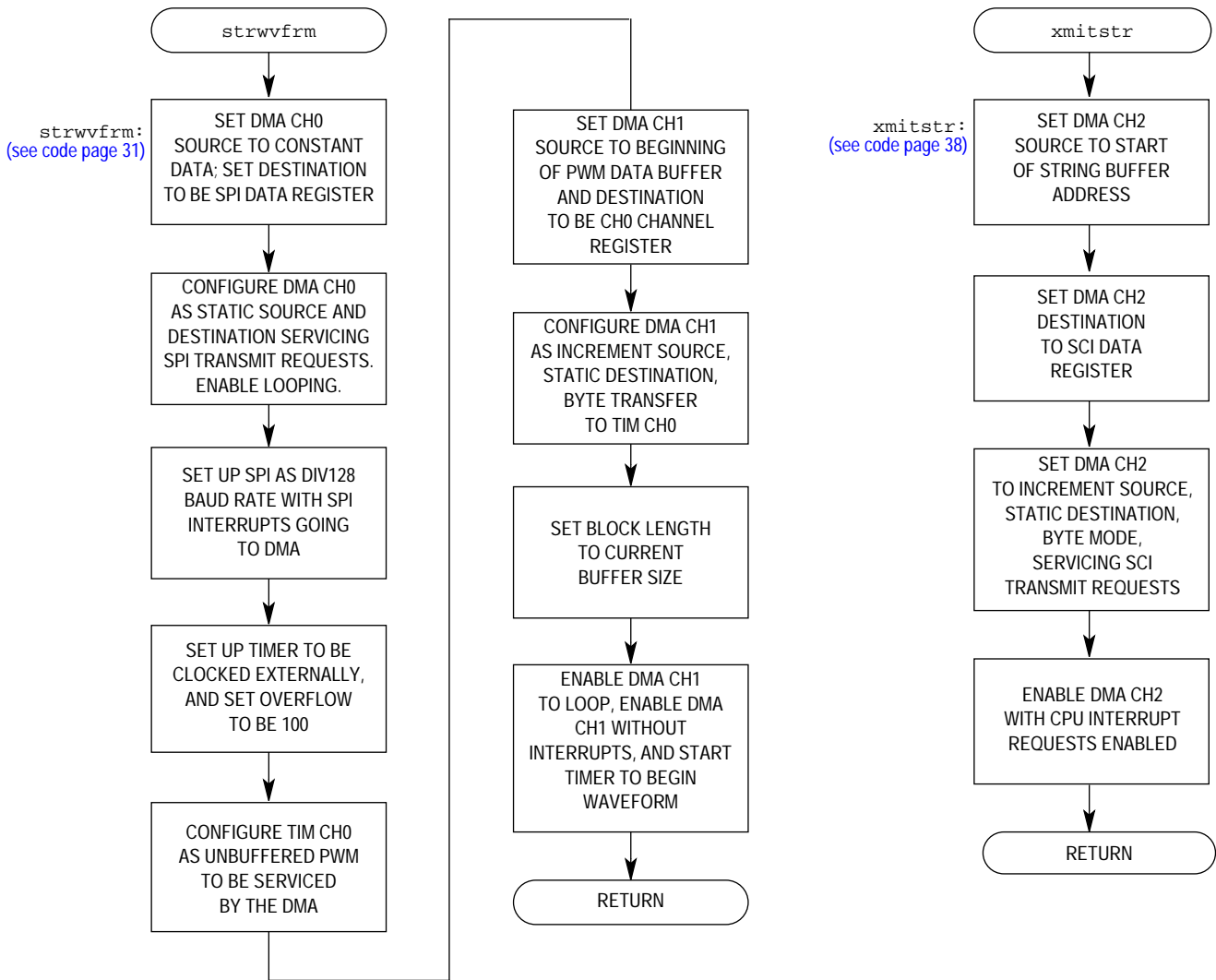


Figure 5. Software Flowchart (Sheet 2 of 13)



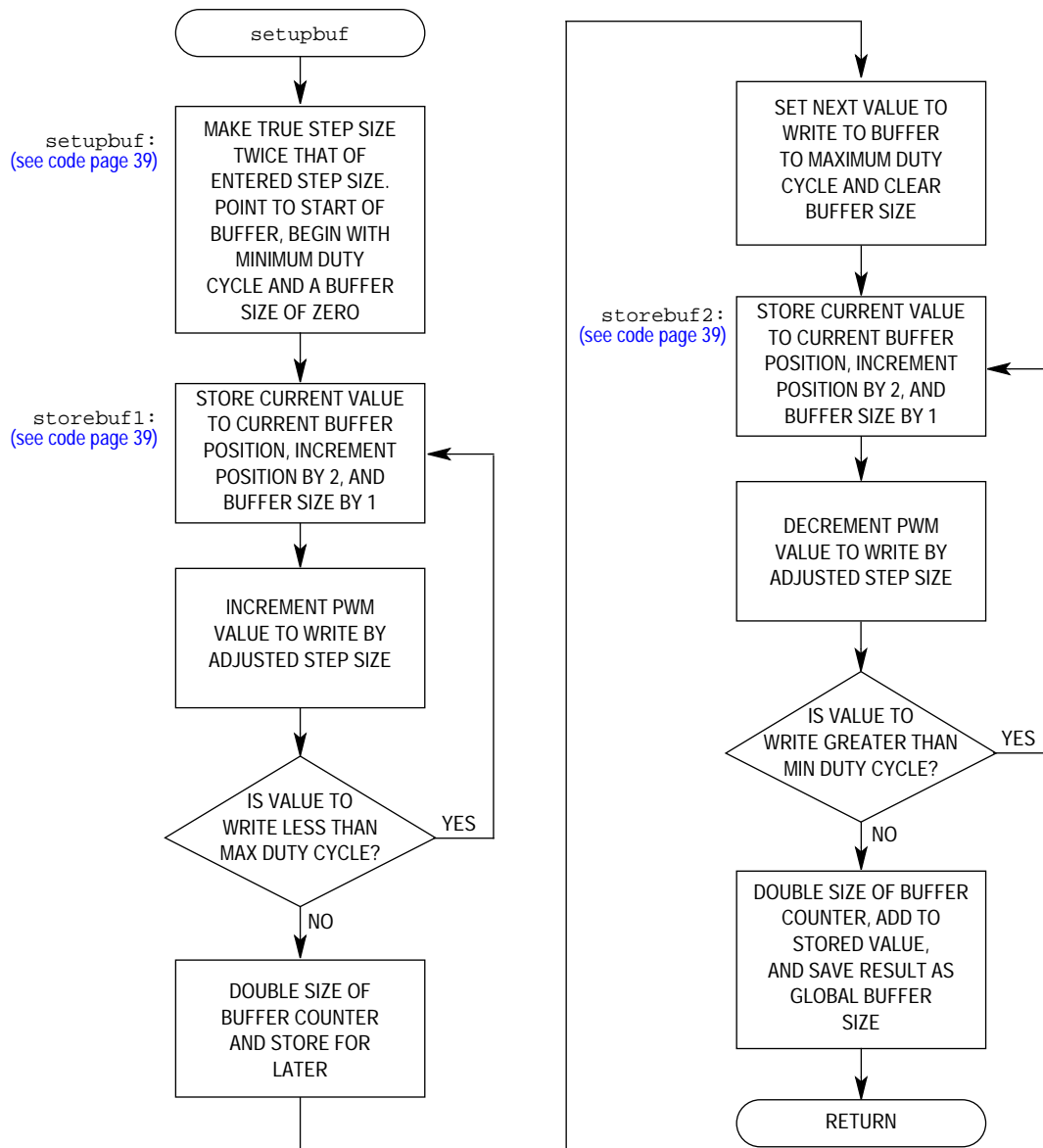


Figure 5. Software Flowchart (Sheet 3 of 13)

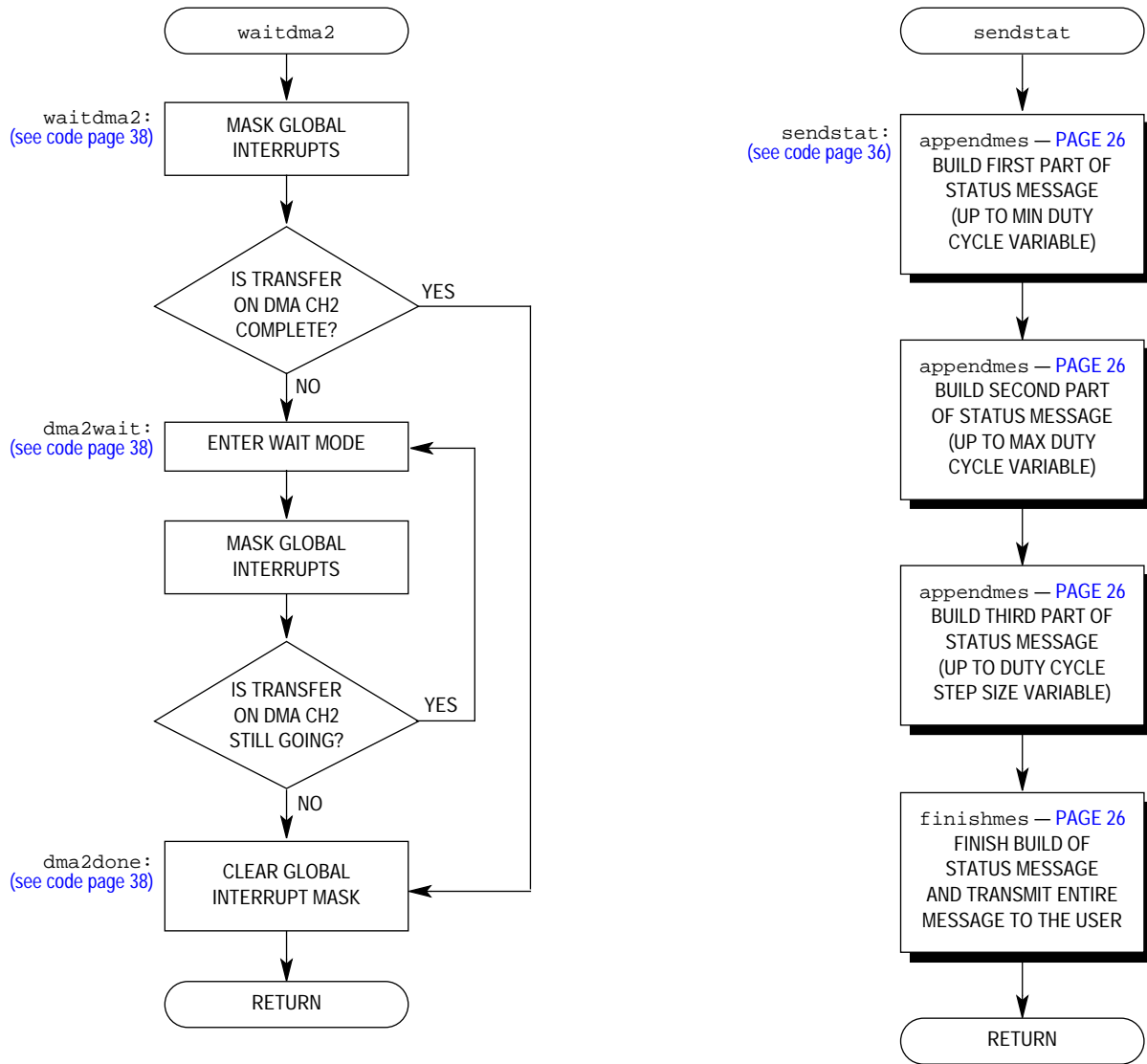


Figure 5. Software Flowchart (Sheet 4 of 13)

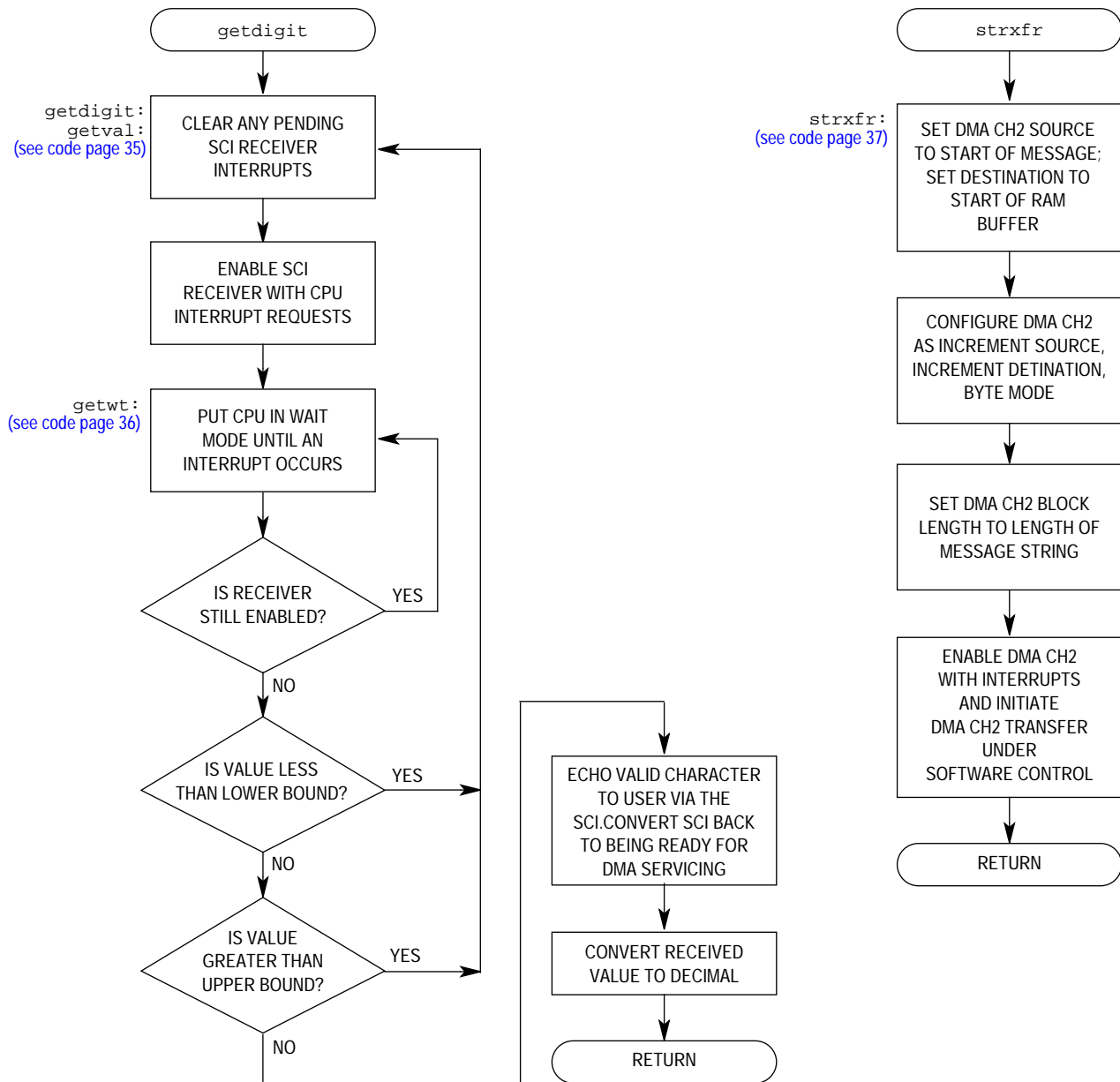
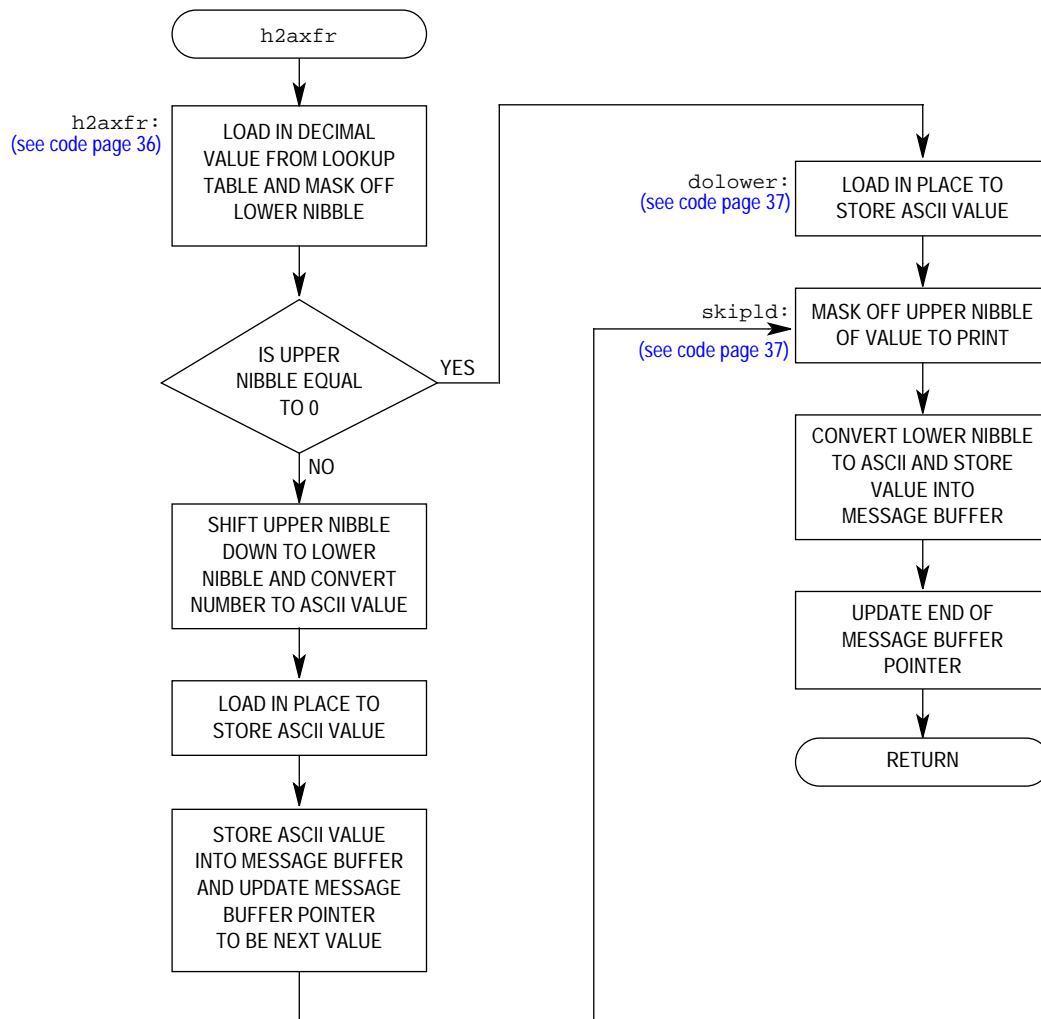


Figure 5. Software Flowchart (Sheet 5 of 13)



**Figure 5. Software Flowchart (Sheet 6 of 13)**

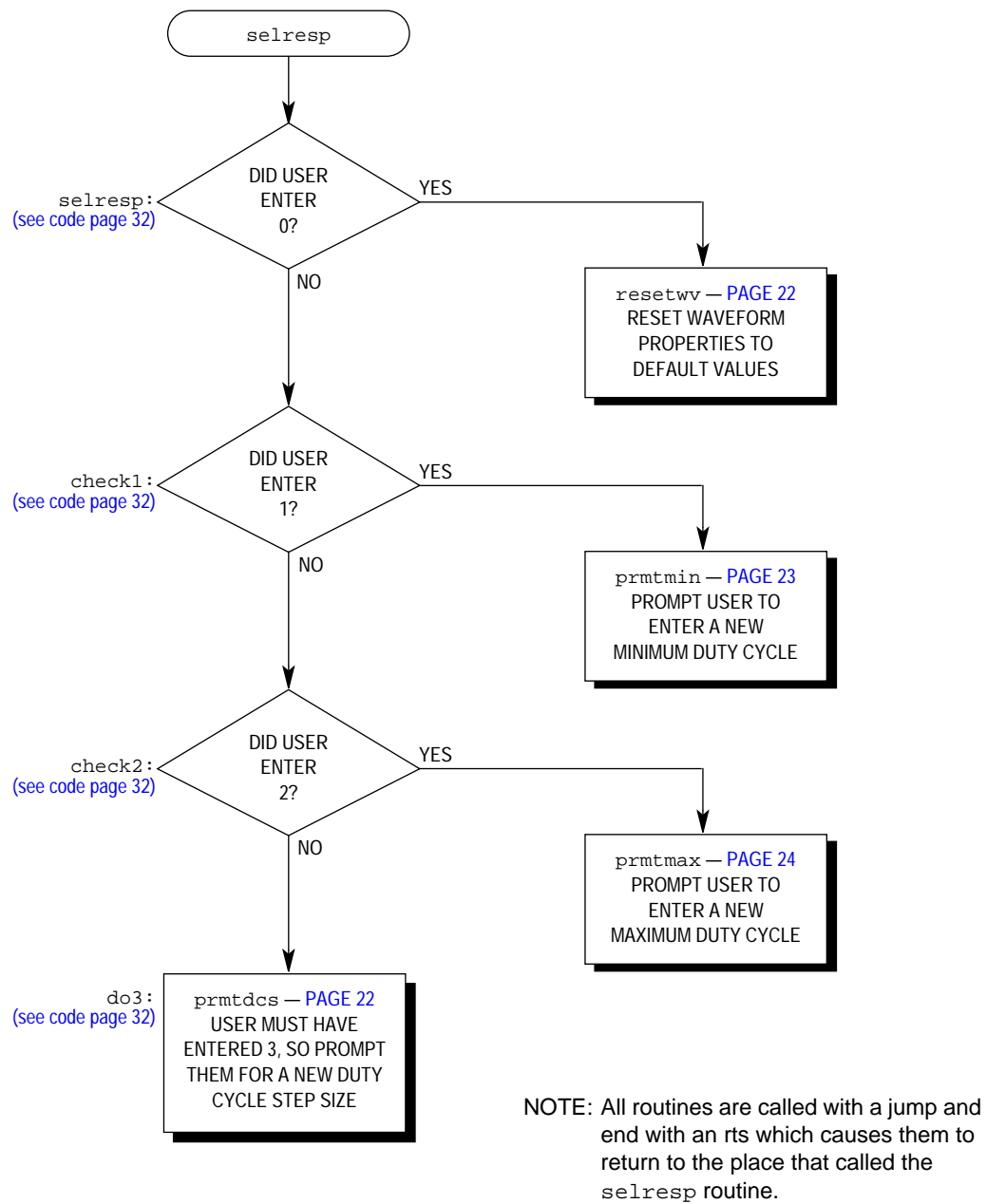


Figure 5. Software Flowchart (Sheet 7 of 13)

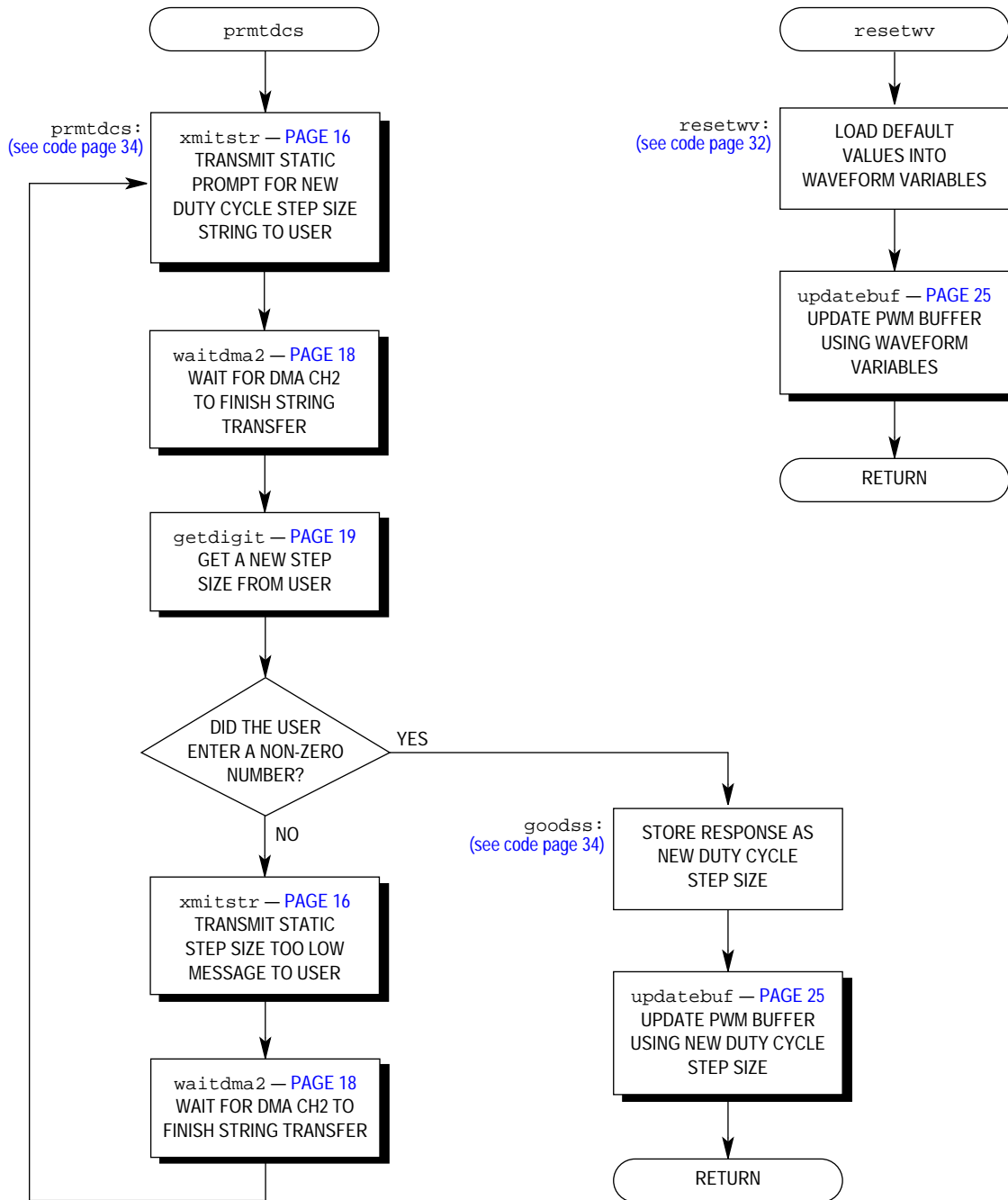


Figure 5. Software Flowchart (Sheet 8 of 13)

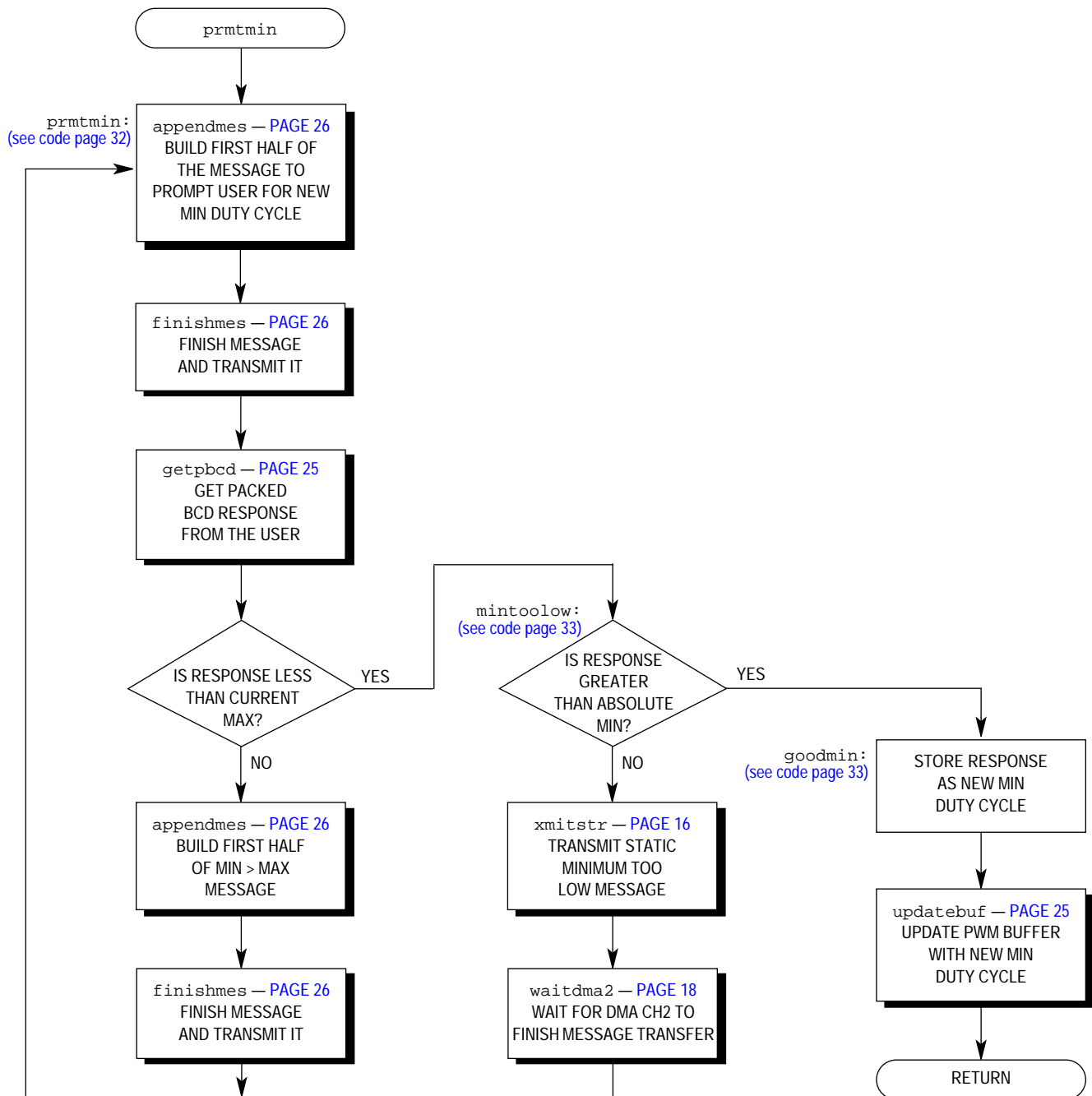


Figure 5. Software Flowchart (Sheet 9 of 13)

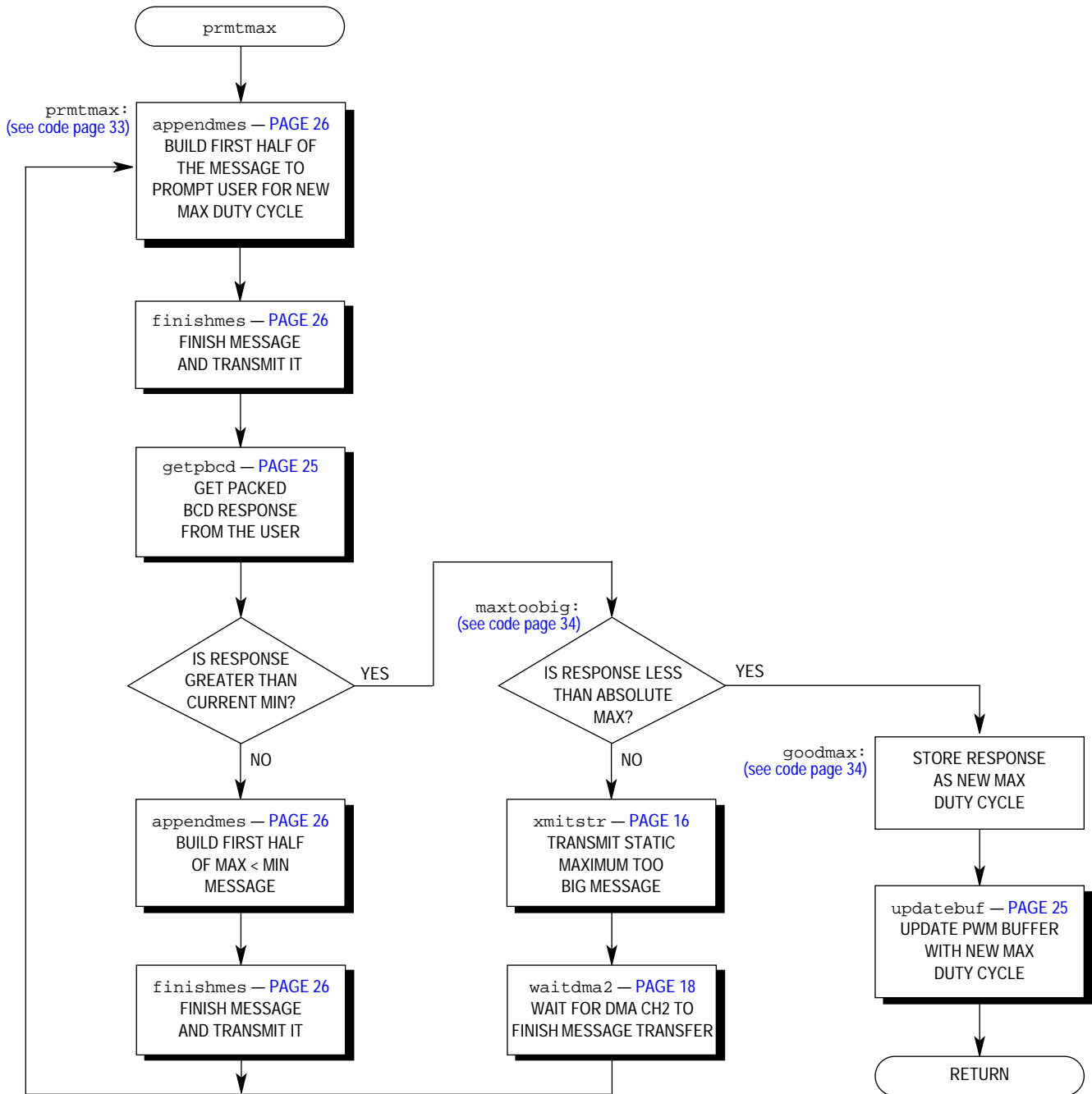


Figure 5. Software Flowchart (Sheet 10 of 13)



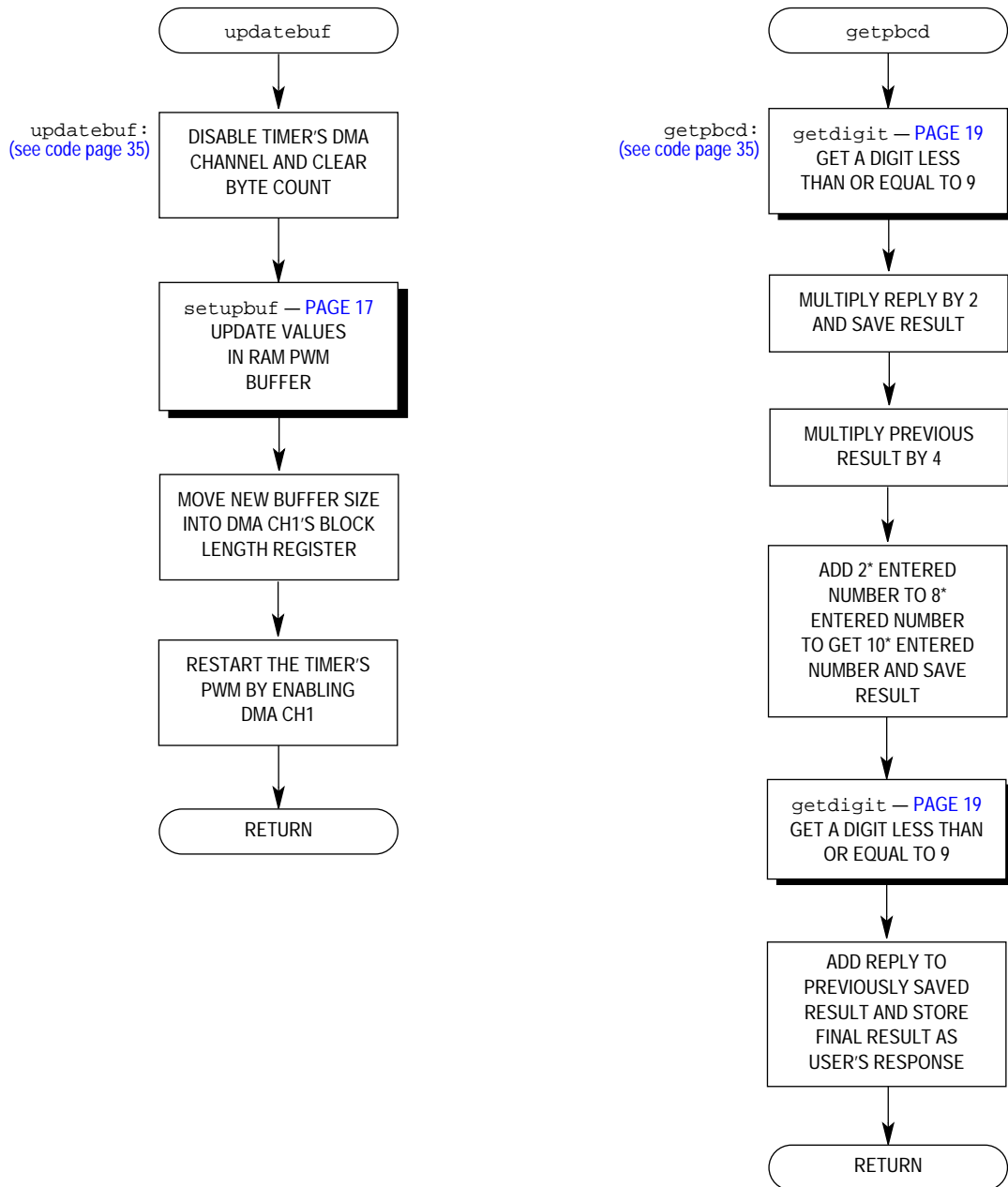
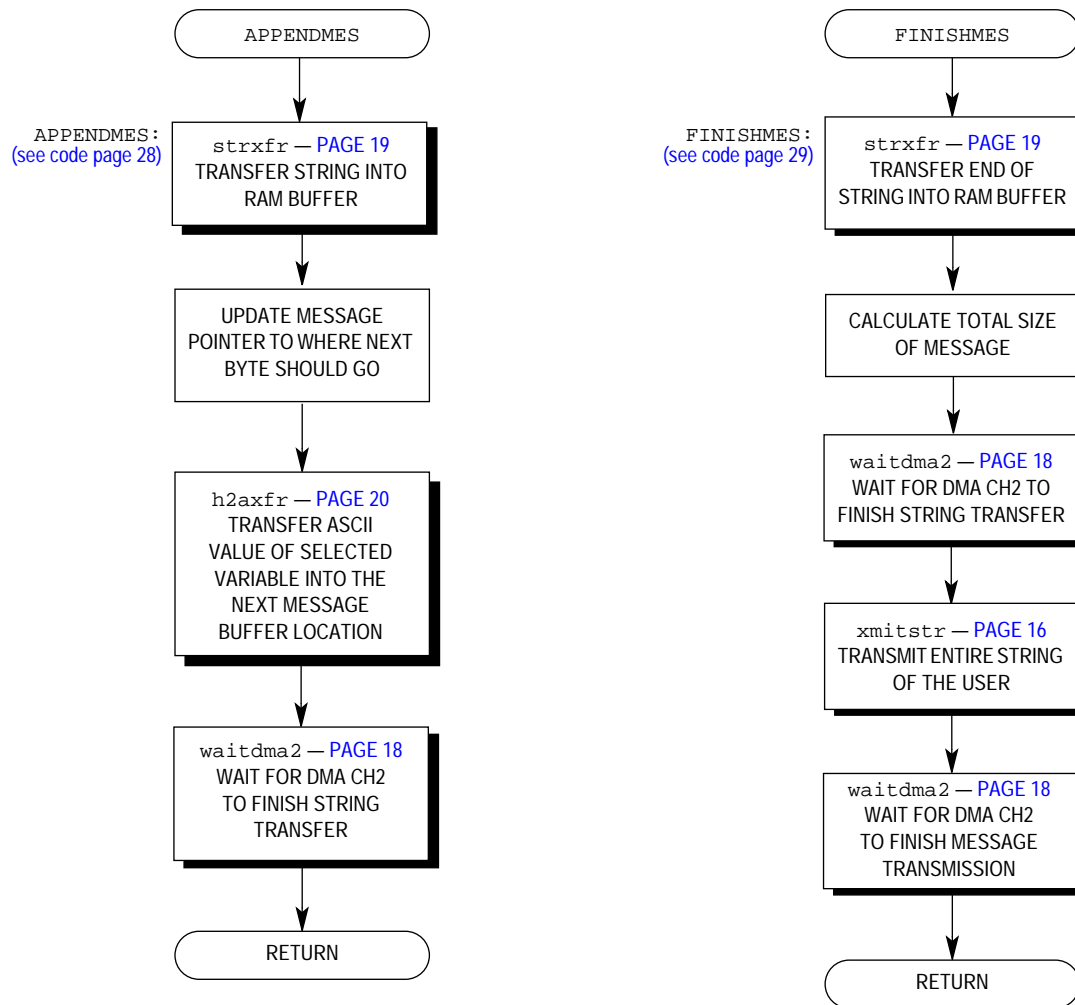
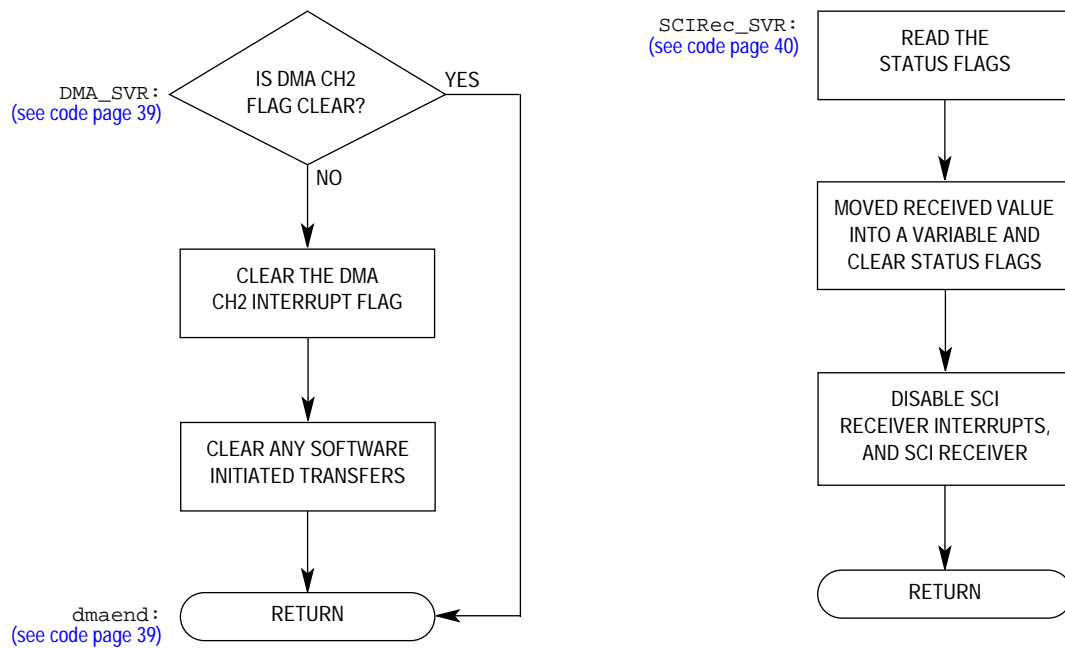


Figure 5. Software Flowchart (Sheet 11 of 13)



NOTE: These code segments are text macros. They will be inserted into the code stream of each calling subroutine instead of being the target of a jump or branch.

**Figure 5. Software Flowchart (Sheet 12 of 13)**



NOTE: These code segments are interrupt service routines.

**Figure 5. Software Flowchart (Sheet 13 of 13)**

## Software Listing

---

```
*****
*      DMA App Note code
*****

$include 'H708XL36.FRK'

* Program Equates
initmin      EQU      !25      ;Initial min duty cycle of 25%
initmax      EQU      !75      ;Initial max duty cycle of 75%
initstep     EQU      !1       ;Initial duty cycle step size = 1
maxbuf       EQU      !164     ;Code requires this to be < 256

* Program variables
                ORG      RAM_Start
rcvbyte      rmb 1
minduty      rmb 1
maxduty      rmb 1
dutystep     rmb 1
bufsize      rmb 1
mesptr       rmb 2

* Data Buffers
bufbegin     rmb maxbuf
mesbuf       rmb !256

*****
*
* APPENDMES: Macro that appends another string to a message under formation
*           Inputs: %1 -- Three character string name indicator
*                   %2 -- Static variable name of a byte to append to
*                   the end of the string as a 1 or 2 byte ASCII char
*           Outputs: The static variable mesptr is updated to indicate
*                   the next available byte in the message buffer.
*
*           This macro should be used to form a segment of a message.  Each message
*           segment will have a string of characters to be printed followed
*           by an integer variable to be printed at the end of that string.
*           For instance, the message 'A max duty cycle of 15 is too small.' would
*           be formed by first calling this macro to pass the first segment of
*           the message.  The string indicated by %1 would consist of
*           'A max duty cycle of ', and the variable would be the one that
*           contained the value 15.  Then the FINISHMES macro would be called
*           to transfer the end of the message.
*
*****
```

```

$MACRO APPENDMES
    ldhx    #str%1        ;Transfer next string into message buffer
    lda     #len%1        ;Load number of bytes in str into Acc
    jsr     strxfr        ;Transfer the string into RAM buffer
    ldhx    mespтр        ;Update message pointer to point to
    aix     #len%1        ; where next string (in this case the
    sthx    mespтр        ; ASCII value) should begin.
    lda     %2            ;Transfer a single byte value
    jsr     h2axfr        ; into the buffer as an ASCII value
    jsr     waitdma2     ;Wait for string transfer to finish

```

\$MACROEND

```

*****
*
* FINISHMES: Macro ends the formation of a message and sends it to the user.
*           Inputs: %1 -- Three character string name indicator
*           Outputs: None
*
*****

```

```

$MACRO FINISHMES
    ldhx    #str%1        ;Transfer rest of string into RAM buffer
    lda     #len%1        ;
    jsr     strxfr        ;
    ldhx    mespтр        ;User buffer pointer to calculate total
    aix     #len%1        ; message length
    sthx    mespтр        ;Update pointer properly
    ldhx    #mesbuf       ;Load in pointer to message to send
    txa     ;Put lower byte of address into Acc
    psha   ;Use value to calc # of bytes in message
    lda     mespтр+1      ;Load least significant byte of pointer
    sub     1,sp          ;Subtract least significant byte of start
    ais     #1            ;Remove value from the stack
    jsr     waitdma2     ;Wait for any previous transfer to finish
    jsr     xmitstr       ;Transfer message to the user and
    jsr     waitdma2     ; wait for string to finish transmission

```

\$MACROEND

```

*****
*
* Beginning of program execution
*
*****

```

```

prog_body    ORG        EPROM_Start
              mov        #COPD,MOR    ;Disable the COP--for EPROM cfg
              ldhx       #RAM_End+1   ;Load a pointer to top of RAM
              txs        ;Set stack pointer to top of RAM
              cli        ;Enable interrupts

```

## Application Note

```
* Initialize global DMA configuration registers
      mov    #$88,DSC      ;Set DMAP, Disable Looping, Set DMAWE
      mov    #$80,DC1     ;Set bandwidth of DMA to 67%

      jsr    initramsci   ;Initialize RAM and SCI using DMA CH2
      jsr    srtwvfrm     ;Start waveform w/ SPI/TIM/DMA
      jsr    waitdma2    ;Wait for Intro message to finish

mainloop
      jsr    sendstat    ;Send status message to user
      ldhx  #strsel      ;Prompt user for which function
      lda   #lensel     ; they would like to select
      jsr    xmitstr
      jsr    waitdma2    ;Wait for transfer to finish
      lda   #3          ;User can respond with 0 - 3
      jsr    getdigit    ;Get a valid value--dec result in Acc
      jsr    selresp     ;Respond to user's selection
      bra   mainloop     ;Keep going in the main loop
```

```
*****
*
* initramsci--Use dma channel 2 to initialize RAM buffer and send
*   introduction message via the SCI to the user.
*   Inputs: None
*   Outputs: None
*   Note: The introduction message started in this routine will be using
*   the SCI and DMA CH2. Any routine following this one that uses
*   DMA CH2 should be sure to wait until this transfer is complete
*   by executing a 'jsr waitdma2'.
*
*****
```

```
initramsci    ldhx  #absmaxduty ;Set src addr to be abs max duty const
              sthx  D2SH
              ldhx  #bufbegin  ;Set dest addr to be buffer pointer
              sthx  D2DH
              mov   #$2c,D2C    ;Static src, inc dest, word, and
                               ; set to SPI even though it is software
              mov   #maxbuf,D2BL ;Fill in entire table with constant
              bset  TEC2,DC1    ;Enable DMA CH2 w/o interrupts
              mov   #$10,DC2   ;Initiate DMA transfer
              nop                    ;All DMA word transfers are 100%
              nop                    ; bandwidth. NOPS ensure DMA transfer
                               ; had time to start before clear below
              clr   DC2          ;DMA transfer should be finished now
              bclr  IFC2,DSC    ;Clear DMA CH2 interrupt flag
```

```
* Configure the SCI to ready it to transmit data sent to it from the DMA
      mov    #$03,SCBR      ;Initialize SCI Baud rate to 9600
      bset   ENSCI,SCC1    ;Enable the SCI to ready it to transfer
      mov    #$10,SCC3     ;Enable the DMA SCI transmitter interrupt
      mov    #$88,SCC2     ;Enable the SCI transmitter
```

```

* Transfer intro message via the SCI
    ldhx    #strint      ;H:X must have pointer to start of message
    lda     #lenint     ;Acc must have number of bytes to send
    jsr     xmitstr     ;Transmit introduction to user screen

* Set up RAM buffer with initial waveform to send
    mov     #initmin,min-duty
    mov     #initmax,max-duty
    mov     #initstep,duty-step
    jsr     setupbuf

    rts

*****
* srtwvfrm: Set up SPI and DMA CH0 to create external clock for timer
*           and DMA CH1 and TIM CH0 to start the PWM waveform.
*           Inputs: None
*           Outputs: None
*****

srtwvfrm    ldhx    #spidata      ;Set up pointer to SPI data to send
            sthx    D0SH
            clr     D0DH          ;Destination is SPI data register
            mov     #SPDR,D0DL
            mov     #$05,D0C      ;Static src & dest, byte, SPI Trans
            mov     #$FF,D0BL     ;Since looping on same byte (static src
            ; and dest), byte count is arbitrary
            bset   L0,DSC        ;Make it loop on this transfer
            bset   TEC0,DC1      ;Enable DMA CH0 w/o interrupts
            mov     #$03,SPSCR    ;Set up SPI with div 128 baud rate
            mov     #$63,SPCR    ;Enable SPI as a mstr with dma xmit int

* Set up timer CH0 to create the PWM in conjunction with DMA CH1
    mov     #$37,TSC           ;Stop & reset timer; clock externally
    clr     TMODH              ;Set PWM period by programming
    mov     #!100,TMODL       ; overflow register
    clr     TCH0H              ;Initialize w/ a min duty cycle by
    mov     minduty,TCH0L     ; writing a byte to channel reg 0
    mov     #$5A,TSC0         ;Configure chan 0 as unbuffered PWM
    mov     #$01,TDMA         ; and enable it to be service by DMA

* Set up DMA CH1 to receive timer CH0 interrupt
    ldhx    #bufbegin         ;Load in beginning of buffer
    sthx    D1SH              ;Store in source address of DMA CH1
    ldhx    #TCH0L            ;Load in address of TIM CH0
    sthx    D1DH              ;Store in dest address of DMA CH1
    mov     #$80,D1C          ;Inc src, static dest, byte xfer, TIM CH0
    mov     bufsize,D1BL     ;Load bytes in table for initial case
    bset   L1,DSC            ;Enable looping on this channel
    bset   TEC1,DC1          ;Enable DMA CH1 w/o interrupts

    bclr   TSTOP,TSC         ;Start timer waveform
    rts

```

## Application Note

\*\*\*\*\*

\*  
\* selresp: Select correct action based on user's response to main menu  
\*       Inputs: Acc has user input value (decimal value from 0 to 3)  
\*       Outputs: None, but registers altered  
\*       Note, this routine does not directly execute an RTS. Instead it jumps  
\*       to a routine that takes the appropriate action, and these routines  
\*       are all ended by an RTS.  
\*

\*\*\*\*\*

```
selresp          tsta          ;Did user enter 0
                 bne    check1   ;If not, see if it was 1
                 jmp    resetwv  ;If 0, reset waveform to default values

check1           cmp    #$01     ;Did user ask to do selection 1?
                 bne    check2   ;If not, look to see if it was 2
                 jmp    prmtmin  ;If 1, prompt user for minimum value

check2           cmp    #$02     ;Did user ask to do selection 2?
                 bne    do3      ;If not, must have asked for selection 3
                 jmp    prmtmax  ;If 2, prompt user for maximum value

do3              jmp    prmtdcs   ;Since 3, prompt for duty cycle step size
```

\*\*\*\*\*

\*  
\* resetwv: Routine used to reset waveform back to it's default values  
\*       Inputs: None  
\*       Outputs: None  
\*

\*\*\*\*\*

```
resetwv          mov    #initmin,minduty ;Reset buffer parameters back
                 mov    #initmax,maxduty  ; to the default values as
                 mov    #initstep,dutystep ; requested by the user
                 jsr    updatebuf        ;Update timer PWM buffer
                 rts                    ;All we need to do for selection 0
```

\*\*\*\*\*

\*  
\* prmtmin: Prompt user to enter the minimum duty cycle value  
\*       Inputs: None  
\*       Outputs: None, but register are altered  
\*

\*\*\*\*\*

```
prmtmin          ldhx   #mesbuf        ;Load address to beginning of message buffer
                 sthx   mesptr        ;Reset message pointer to start of buffer
                 APPENDMES gmn maxduty;Prompt user to enter minimum duty
```



```

FINISHMES gmf          ; cycle (strgmn + maxduty + gmf)
jsr   getpbcd         ;Get the packed bcd response from user
cmp   maxduty         ;Is this less than the max duty cycle
blt   mintoollow     ;If so, make sure it isn't too low

ldhx  #mesbuf         ;Load address to beginning of message buffer
sthx  mesptr          ;Reset message pointer to start of buffer
APPENDMES emh maxduty ;Start the min-duty-too-high message
FINISHMES fin         ;Finish the min-duty-too-high message
jmp   prmtmin         ;Prompt them again for the value

mintoollow           cmp   #9             ;Is the value greater than 9
                        bgt   goodmin        ;If so, value is ok
                        ldhx  #streml       ;Tell user they entered too small a value
                        lda   #leneml
                        jsr   xmitstr
                        jsr   waitdma2
                        jmp   prmtmin        ;Prompt them again for the value

goodmin              sta   minduty         ;Checks out ok, so save
                        jsr   updatebuf     ;Update the timer buffer with this value
                        rts                ;All we need to do for 1 selection

*****
*
* prmtmax: Prompt user to enter the maximum duty cycle value
*       Inputs: None
*       Outputs: None, but register are altered
*
*****

prmtmax              ldhx  #mesbuf         ;Load address to beginning of message buffer
                        sthx  mesptr          ;Reset message pointer to start of buffer
APPENDMES gmh minduty ;Prompt user to enter maximum duty
FINISHMES gxf         ; cycle (strgmx + minduty + strgxf)
jsr   getpbcd         ;Get the packed bcd response from user
cmp   minduty         ;Is this greater than the min duty cycle
bgt   maxtoobig       ;If so, make sure it isn't too large

ldhx  #mesbuf         ;Load address to beginning of message buffer
sthx  mesptr          ;Reset message pointer to start of buffer
APPENDMES exl minduty ;Start the max-duty-too-low message
FINISHMES fin         ;Finish the max-duty-too-low message
jmp   prmtmax         ;Prompt them again for the value

```

## Application Note

```
maxtoobig      cmp     #!91          ;Is the value less than 91
                blt     goodmax        ;If so, value is ok
                ldhx   #strexh        ;Tell user they entered too large a value
                lda    #lenexh
                jsr    xmitstr
                jsr    waitdma2
                jmp    prmtmax        ;Prompt them again for the value

goodmax        sta    maxduty          ;Checks out ok, so save
                jsr    updatebuf       ;Update the timer buffer with this value
                rts                    ;All we need to do for 2 selection

*****
*
* prmtdcs: Prompt user to enter the duty cycle step size
*       Inputs: None
*       Outputs: None, but register are altered
*
*****

prmtdcs        ldhx   #strdcs          ;Prompt user to enter duty cycle step size
                lda    #lendcs        ; by sending the get duty cycle step string
                jsr    xmitstr        ; (strdcs) via the SCI/DMA
                jsr    waitdma2       ;Wait for string transfer to finish
                lda    #$9           ;Max value for the step size is 9
                jsr    getdigit       ;Get the digit from the user
                tsta                    ;Did he enter a non-zero number?
                bne    goodss         ;If so, then this is a good step size

                ldhx   #strlss        ;Tell user that step size is too low
                lda    #lenlss
                jsr    xmitstr
                jsr    waitdma2
                jmp    prmtdcs        ;Prompt them again for the value

goodss        sta    dutystep         ;Store value into global variable
                jsr    updatebuf       ;Update the timer buffer with this value
                rts                    ;All we need to do for 3 selection

*****
*
* getbcd: Get a packed bcd number from the user
*       Inputs:None
*       Outputs:Packed BCD value will be in accumulator
*       Note: Only valid decimal digits will be echoed to the user,
*       and the routine requires two digits be entered, without the
*       need for a carriage return.
*
*****
```

```

getpbcd          lda    #$9           ;Set max potential digit to be 9
                 jsr    getdigit      ;Get 1st digit of user's response
                 asla                   ;Multiply digit by 2
                 psha                   ;Save result
                 asla                   ;User's digit times 4
                 asla                   ;User's digit times 8
                 add    1,sp           ;Acc = 8*digit+2*digit = 10*digit
                 sta    1,sp           ;Save result over now useless data
                 lda    #$9           ;Set max potential digit to be 9
                 jsr    getdigit      ;Get second digit of user's response
                 add    1,sp           ;Accumlator now has packed BCD value
                 ais    #1             ;Remove value from the stack
                 rts                    ;Return with bcd number in acc

```

```

*****
*
* updatebuf: Update timer PWM buffer
*           Inputs: None
*           Outpus: None
*           Dependancy: The three static variables minduty, maxduty, and
*                       dutystep need to be properly setup before this
*                       routine is called. It uses the static variable
*                       bufsize which is altered by the setupbuf routine.
*
*****

```

```

updatebuf       bclr   TEC1,DC1       ;Disable timer's DMA channel
                 clr    D1BC          ; and ready it for a new transfer
                 jsr    setupbuf      ;Update the buffer
                 mov    bufsize,D1BL ;Update the buffer size
                 bset   TEC1,DC1      ;Restart the timer PWM
                 rts                    ;New PWM has begun

```

```

*****
*
* getdigit: Accept input from the terminal, echoing only digits, and
*           returning the decimal values to the calling routine
*           Inputs:Maximum digit value acceptable in Acc
*           Outputs:Decimal digit accepted from user in Acc
*           Description: Clears all pending receiver interrupts, enables
*                       the receiver and its interrupts, and then waits for the
*                       interrupt. The receiver ISR will disable the RE bit,
*                       which tells this routine that a byte has been received.
*                       The ISR places the received byte in the static variable,
*                       rcvbyte. If the received byte is valid, it is echoed to
*                       the user, otherwise no response is made. Once a valid
*                       value is received, the decimal equivalent is returned.
*
*****

```

```

getdigit        psha                   ;Save max value
getval          lda    SCS1            ;Clear any pending SCI receive flags
                 lda    SCD
                 bset   RE,SCC2        ;Enable SCI receiver
                 bset   RIE,SCC2      ; with interrupts

```

## Application Note

```
getwt          wait          ;Wait for digit to be accepted
               brset RE,SCC2,getwt ;If receiver still active, wait more
               lda   #$30      ;Load acceptable lower bound
               cmp   rcvbyte    ;Is value less than lower bound?
               bgt   getval     ;If so, keep looking for valid value
               add   1,sp      ;Acc now has upper limit
               cmp   rcvbyte    ;Is value greater than upper bound?
               blt   getval     ;If so, keep looking for valid value
               pula          ;Clear value off of stack
               lda   rcvbyte    ;Load in the valid received value
               bclr  TIE,SCC2   ;Disable SCI transmitter interrupts
               bclr  DMATE,SCC3 ;Use the CPU to send 1 byte via SCI
               brclr SCTE,SCS1,* ;Wait for TE to become set
               sta   SCD        ;Echo back to screen
               bset  DMATE,SCC3 ;Reconfigure SCI as a DMA interrupt
               bset  TIE,SCC2   ;Reenable SCI transmitter interrupts
               sub   #$30      ;Convert to a decimal value
               rts

*****
* sendstat: Send the status string to the user
*           Inputs: None
*           Outputs: None, but registers are altered
* To do so, we need to build up a status message in the RAM string buffer.
* This message consists of 3 text segments each terminated with a value.
*
*****

sendstat       ldhx   #mesbuf   ;Load address to beginning of message buffer
               sthx   mesptr    ;Reset message pointer to start of buffer

               APPENDMES sbg minduty;Part one of message and minduty const
               APPENDMES smd maxduty;Part two of message and maxduty const
               APPENDMES sed dutystep ;Part three of message and dutystep const
               FINISHMES sfn      ;Finish and send the status message

               rts

*****
* h2axfr: Convert a hex value into an ascii character, and then transfer
*         it to the message buffer
*         Inputs: Hex value to convert is in Acc.
*         Buffer location to place ascii in mesptr variable.
*         Outputs: Mesptra variable updated to point to next free location
*
*****

h2axfr         psha          ;Save registers on the stack to preserve
               pshx
               pshh
```

```

        clrh                ;Clear H so that H:X has proper byte offset
        tax                 ;Transfer value into X to serve as offset
        lda     h2pbcd,x    ;Load in converted value
        psha                ;Save value on stack
        and     #$f0        ;Mask off lower nibble
        beq     dolower     ;Value is < 10, so only print 1 digit
        lsra                ;Shift upper nibble into lower nibble
        lsra
        lsra
        lsra
        add     #$30        ;Convert number to ascii
        ldhx   mesptra     ;Load in place to store value
        sta     ,x          ;Store into message table
        aix     #1          ;Increment to next empty position in table
        bra     skipld     ;skip load of message point--in H:X
dolower  ldhx   mesptra     ;Load in place to store converted value
skipld   pula                ;Restore converted value to print
        and     #$0f        ;Mask off upper nibble
        add     #$30        ;Convert number to ascii
        sta     ,x          ;Store into message table
        aix     #1          ;Increment to next empty position in table
        sthx   mesptra     ;Update static message pointer variable
        pulh                ;Restore registers
        pulx
        pula
        rts

```

```

*****
*
* strxfr: Use DMA CH2 to transfer an ascii string to RAM message buffer
*         Inputs: Pointer to beginning of string to transfer in H:X
*         Number of bytes in string in Acc
*         Place to put string in a RAM message pointer--mesptr
*         Outputs: None
*
*****

```

```

strxfr   pshx                ;H:X will be altered so save on stack
        pshh
        sthx   D2SH          ;Source is beginning of string
        ldhx   mesptra     ;Set dest addr to be value in current
        sthx   D2DH          ; RAM message buffer pointer
        mov    #$A4,D2C     ;Inc src, inc dest, byte, and
                           ; set to SPI receive (unused)
        sta   D2BL          ;Acc has number of bytes in string
        bset   IEC2,DC1     ;Enable DMA CH2 w/ interrupts
        bset   TEC2,DC1     ; so the software bit can be cleared
        bset   4,DC2        ;Initiate DMA transfer
        pulh                ;Restore H:X off of stack
        pulx
        rts

```

## Application Note

```
*****
*
* waitdma2: Wait for DMA CH2 to finish its current transfer before returning
*           Inputs: None
*           Outputs: None
*
*****

waitdma2      sei                      ;Don't allow interrupt that is
              ; needed to pull MCU out of wait
              ; to occur btwn brclr & wait
dma2wait      brclr  TEC2,DC1,dma2done ;Transfer complete already?
              wait                      ;Allow DMA CH2 to complete
              ; Also clears I bit to allow int
              sei                      ;Don't allow interrupt b/f wait
dma2done      brset  TEC2,DC1,dma2wait ;DMA CH2 finished if TEC2 is clear
              cli                      ;Interrupt has been taken,
              ; so allow others to occur
              rts

*****
*
* xmitstr: Subroutine used to initiate a transfer to the SCI via DMA CH2
*           Inputs: 1) 16 bit address pointer to beginning of string in H:X
*                   2) Number of bytes in string in Acc (max of 256).
*           Outputs: None, but DMA CH2 is enabled to transmit to SCI
*           Assumptions: Channel 2 looping is disabled, DMA DMAP and bandwidth
*                       are configured as desired.
*
*****

xmitstr       sthx   D2SH                ;Pointer to start of string -> src reg
              clr    D2DH                ;Move SCI data register (in page zero)
              mov    #SCD,D2DL           ; into destination register
              mov    #$87,D2C            ;Select Inc. Source & Static Dest.,
              ; Byte transfers, and SCI Transmit Int
              sta    D2BL                ;Number of bytes to send -> block len reg
              bset   IEC2,DC1            ;Enable DMA CH2 with interrupts
              bset   TEC2,DC1
              rts

*****
*
* setupbuf: Routine used to fill buffer with values to send to timer
*           to create variable PWM on channel 0--registers unaltered
*           Inputs: Correct values already set in minduty, maxduty, and
*                   dutystep variables.
*           Outputs: Bufsize will contain the number of bytes in buffer
*
*****
```

```

setupbuf      pshh                ;Save value of registers on stack
              pshx
              psha
              asl    dutystep      ;Double step size to keep buffer < 200 bytes
              ldhx  #bufbegin     ;Point to beginning of buffer
              lda   minduty       ;Load in first buffer value
              clr   bufsize       ;Initialize byte count to 0
storebuf1     sta   ,x            ;Store value into buffer
              aix   #2            ;Skip over preset buffer value
              inc   bufsize       ;Increment number of entries
              add   dutystep      ;Increment PWM by step size
              cmp   maxduty       ;Compare to max value
              bls   storebuf1     ;If not exceeded, store and do next
              lda   bufsize       ;Double buffer size to account
              add   bufsize       ; for fixed values stored in buffer
              psha                ;Remember number of bytes stored so far

              lda   maxduty       ;Load in next buffer value
              clr   bufsize       ;Ready byte count for second half
storebuf2     sta   ,x            ;Store value into buffer
              aix   #2            ;Skip over preset buffer value
              inc   bufsize       ;Increment number of entries
              sub   dutystep      ;Decrement PWM by step size
              cmp   minduty       ;Compare to min value
              bhs   storebuf2     ;If still higher, store and do next
              lda   bufsize       ;Double buffer size to account
              add   bufsize       ; for fixed values stored in buffer
              add   1,sp          ;Add in value from first half
              ais   #1            ;Clear value off of the stack
              sta   bufsize       ;Store total off for later
              asr   dutystep      ;Restore step size back to entered value
              pula                ;Restore registers from stack
              pulx
              pulh
              rts

```

```

*****
*
* DMA_SVR: Interrupt service routine for the DMA
*   Inputs: None
*   Outputs: For channel 2, the IFC2 bit is cleared.
*   Note: Only DMA CH2 can create interrupts.
*
*****

```

```

DMA_SVR      brclr  IFC2,DSC,dmaend ;CH2 interrupt service routine
              bclr  IFC2,DSC      ;Clear CH2 flag
              clr   DC2          ;Clear any software initiated transfer
              ; Not needed for SCI servicing
dmaend       rti

```

## Application Note

```
*****
*
* SCIRec_SVR: Interrupt service routine for the SCI receiver
*           Inputs: None
*           Outputs: Received data byte put into static variable rcvbyte
*           Note: SCI receiver is disabled after each received byte
*
*****

SCIRec_SVR      lda      SCS1          ;Load status reg--ignore error flags
                mov      SCD,rcvbyte ;Store received byte for other routines
                bclr    RIE,SCC2      ;Disable the SCI receiver interrupts
                bclr    RE,SCC2       ; and receiver itself between chars
                rti

*** Program constants
absmaxduty      fdb      !99          ;Change to next pulse width at 99% of period
                ; when increasing (two bytes for DMA)
spidata         db       $0f         ;Create a clock with output of SPI MOSI
                ; By changing data, one can change freq
                ; of the clock used to generate PWM
*****

*** Strings to be printed to the user
*** Naming convention: str<name> indicates the beginning of the <name> string
***                      end<name> indicates the end of the <name> string
*** All end<name> labels should be followed by 1 byte to be consistent.
*** Following each string is an equate (called len<name>) that equals the
*** string's length in bytes ( len<name> = end<name>-str<name>+1 ).
*** Some messages need to have numbers inserted into them, so there is
*** a separate string for each message segment.
*** This naming convention must be followed to use the defined macros.
*** Note that no message is allowed to have more than a total of 256 bytes.
*****
* ASCII control character equates
cr              EQU      $0d          ;Return cursor to beginning of line
lf              EQU      $0a          ;Advance cursor one line
sub            EQU      $1a          ;Clear screen

* Intro string (strint to endint)
strint         db       sub,'Welcome to the DMA demonstration.  The SPI MOSI '
                db       'is being used to generate',cr,lf
                db       'an external clock for the timer which in turn is '
                db       'generating a varying',cr,lf
                db       'PWM on channel 0--both continuously driven '
                db       'by the DMA.  Also, all text',cr,lf
                db       'is sent using the SCI via the DMA.',cr
endint         db       lf
lenint         EQU      endint-strint+1

* Status string begin (strsbg to endsbg)
strsbg         db       cr,lf,lf
                db       'Currently generating a waveform that varies from a '
                db       'duty cycle of'
endsbg         db       ' '
```



```

lensbg          EQU    endsbg-strsbg+1
* Status string middle (strsmd to endsmd)
strsmd          db     '% to'
endsmd         db     ' '
lensmd          EQU    endsmd-strsmd+1
* Status string end (strsed to endsed)
strsed         db     '% at',cr,lf,'a step size of'
endsed        db     ' '
lensed         EQU    endsed-strsed+1
* Status string finish (strsfm to endsfm)
strsfm         db     '. Please choose which you would like to alter'
endsfm        db     '.'
lensfm         EQU    endsfm-strsfm+1
* Function select string (strsel to endsel)
strsel         db     cr,lf,lf,'Would you like to change',cr,lf
              db     ' 0) back to the default values',cr,lf
              db     ' 1) the minimum duty cycle value',cr,lf
              db     ' 2) the maximum duty cycle value',cr,lf
              db     ' 3) the step size of the change in duty cycle',cr,lf
              db     ' '
endsel        db     '?'
lensel         EQU    endsel-strsel+1
* Get minimum duty cycle value (strgmn to endgmn)
* Note, this is the first string of a two string message (goes with strgmf)
strgmn         db     cr,lf,lf,'Please enter the minimum duty cycle '
              db     '[must be an integer between 10',cr,lf
              db     'and'
endgmn        db     ' '
lengmn         EQU    endgmn-strgmn+1

* End of get minimum duty cycle value (strgmf to endgmf)
strgmf         db     '--the current maximum duty cycle]:'
endgmf        db     ' '
lengmf         EQU    endgmf-strgmf+1

* Error string: minimum duty cycle entered too high (stremh to endemh)
* Note, this is the first string of a two string message (goes with strmhf)
stremh         db     cr,lf,'The minimum duty cycle must be less than the '
              db     'current maximum duty cycle',cr,lf,'value of'
endemh        db     ' '
lenemh         EQU    endemh-stremh+1

* Finish of generic error message
strfin         db     '. Please try again'
endfin        db     '.'
lenfin         EQU    endfin-strfin+1

* Error string: minimum duty cycle entered too low (streml to endeml)
streml         db     cr,lf,'The minimum duty cycle must be greater than 9.'
              db     ' Please try again'
endeml        db     '.'
leneml         EQU    endeml-streml+1

```

## Application Note

```
* Get maximum duty cycle value (strgmx to endgmx)
* Note, this is the first string of a two string message (goes with strgmf)
strgmx      db      cr,lf,lf,'Please enter the maximum duty cycle '
            db      '[must be an integer between'
endgmx      db      ' '
lengmx      EQU     endgmx-strgmx+1

* End of get maximum duty cycle value (strgxf to lengxf)
strgxf      db      '--current',cr,lf,'min duty cycle--and 90]:'
endgxf      db      ' '
lengxf      EQU     endgxf-strgxf+1

* Error string: maximum duty cycle entered too low (strex1 to endex1)
* Note, this is the first string of a two string message (goes with strfin)
strex1      db      cr,lf,'The maximum duty cycle must be greater than the '
            db      'current minimum duty cycle',cr,lf,'value of '
endex1      db      ' '
lenex1      EQU     endex1-strex1+1

* Error string: maximum duty cycle entered too high (strexh to endexh)
strexh      db      cr,lf,'The maximum duty cycle must be less than 91.'
            db      ' Please try again'
endexh      db      ' '
lenexh      EQU     endexh-strexh+1

* Get duty cycle step size value (strdcs to enddcs)
strdcs      db      cr,lf,lf,'Please enter the duty cycle step size '
            db      '[must be an integer between 1 and 9]:'
enddcs      db      ' '
lendcs      EQU     enddcs-strdcs+1

* Error string: duty cycle step size entered too low (strlss to endlss)
strlss      db      cr,lf,'The duty cycle step size must be greater than 0.'
            db      ' Please try again'
endlss      db      ' '
lenlss      EQU     endlss-strlss+1

*** Hex to packed BCD lookup table
h2pbcd      db      $00,$01,$02,$03,$04,$05,$06,$07,$08,$09
            db      $10,$11,$12,$13,$14,$15,$16,$17,$18,$19
            db      $20,$21,$22,$23,$24,$25,$26,$27,$28,$29
            db      $30,$31,$32,$33,$34,$35,$36,$37,$38,$39
            db      $40,$41,$42,$43,$44,$45,$46,$47,$48,$49
            db      $50,$51,$52,$53,$54,$55,$56,$57,$58,$59
            db      $60,$61,$62,$63,$64,$65,$66,$67,$68,$69
            db      $70,$71,$72,$73,$74,$75,$76,$77,$78,$79
            db      $80,$81,$82,$83,$84,$85,$86,$87,$88,$89
            db      $90,$91,$92,$93,$94,$95,$96,$97,$98,$99

*****
* Vector equates
*****

ORG         SCIRec_INT
fdb         SCIRec_SVR

ORG         DMA_INT
fdb         DMA_SVR

ORG         RESET
fdb         prog_body
```

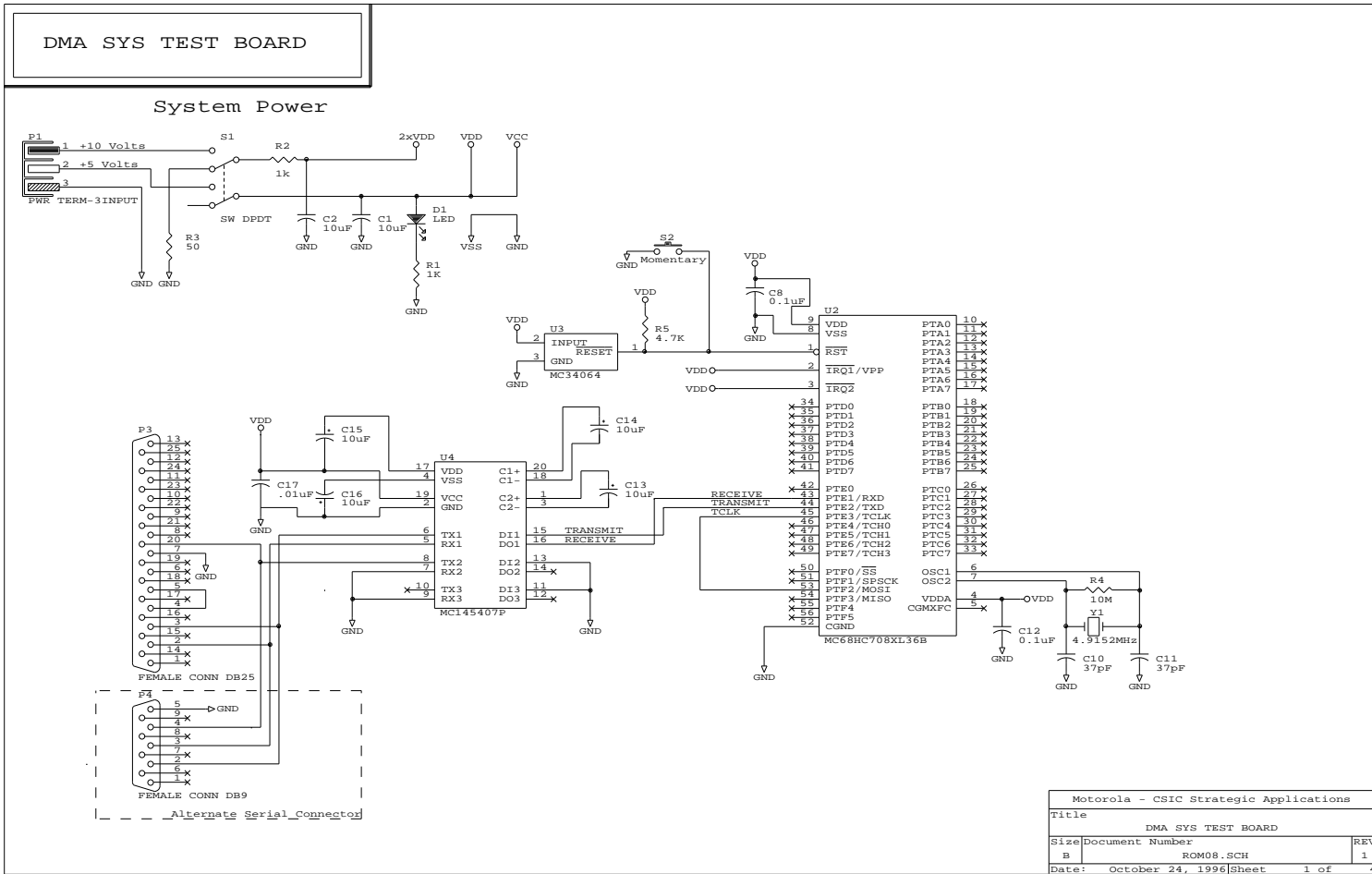



Figure 6. DMA System Test Board

# Application Note

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

## How to reach us:

**USA/EUROPE/Locations Not Listed:** Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1-800-441-2447 or 303-675-2140

**Mfax™:** RMFAX0@email.sps.mot.com – TOUCHTONE 602-244-6609, US & Canada ONLY 1-800-774-1848

**INTERNET:** <http://www.mot.com/SPS/>

**JAPAN:** Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, 6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 81-3-3521-8315

**ASIA/PACIFIC:** Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park, 51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

Mfax is a trademark of Motorola, Inc.



**MOTOROLA**

© Motorola, Inc., 1997

**AN1711/D**