Summary

The following describes how to get greater resolution for the QED's 8-bit DAC.

Description

Often greater resolution is needed than that provided by the QED Board's 8-bit DAC. One solution is to combine two DAC channels in hardware to produce a single channel of greater resolution. This solution is implemented by the QED Analog Conditioning Board, on which two pairs of channels are combined with their output calibrated against the 12-bit A/D, or any other pairs combined without direct calibration. This application note provides another solution: The output of any of the DAC channels can be modulated so that it rapidly flickers between two adjacent levels. After the output is averaged with a low pass filter, up to 256 discrete analog voltages can be produced within each step of the 8-bit DAC. Resolutions up to 16 bits can be produced (given sufficient averaging time), but without the true accuracy of 16-bit D/A converter. The absolute accuracy is still limited to that of the 8-bit DAC itself. Even so, this accuracy is generally better than 8-bits; it is approximately 10-11 bits without calibration, and if calibrated against the 12-bit A/D, better than 12-bits resolution with 12-bits of accuracy can be attained.

```
\  ************************************************************************
\  ************************************************************************
\  **********                                                    **********
\  **********      Greater Resolution for the QED's 8-bit DAC    **********
\  **********                                                    **********
\  **********   Copyright January 1998 by Mosaic Industries, Inc.**********
\  **********                 5437 Central Ave. Ste. 1           **********
\  **********                    Newark, CA 94560                **********
\  **********                                                    **********
\  **********  This code is provided to customers of the QED Board **********
\  **********  for use with the QED Board Software Development   **********
\  **********  environment.  The provision of this code is governed **********
\  **********  by the QED software license.                      **********
\  **********                                                    **********
\  **********              For further information               **********
\  **********         contact Paul Clifford 510-790-8222         **********
\  **********                                                    **********
\  ************************************************************************
\  ************************************************************************
```

```
\  ************************************************************************
\  **********                                                    **********
\  **********                    Overview                        **********
\  **********                                                    **********
\  ************************************************************************

\ This program writes 16-bit values to the 8-bit DAC channels.  Intermediate
\ values between the standard 8-bit levels are achieved by rapidly flickering
\ between two adjacent levels (i.e., pulse width modulating).  In this way
\ each step of the 8-bit DAC is divided into 256 smaller steps.  Downstream
\ of the DAC a low pass filter smooths the flickering levels into an average
\ value with a small residual ripple.  The flickering is done using a PWM
\ algorithm that is optimal in the respect that it requires the least averaging
\ time to achieve a given level of resolution.

\  ************************************************************************
\  **********                                                    **********
\  **********          Under the Hood: How It Works              **********
\  **********                                                    **********
\  ************************************************************************

\ For a detailed description of the PWM algorithm see the Mosaic Industries QED
\ Application Note "MI-AP-056: A PWM Algorithm with Optimal Averaging Properties".
\
\ An interrupt service routine services the DACs every t seconds, where t can
\ be adjusted from a minimum of about 3 msec (it takes 2.36 msec to service all
\ the DAC channels) to as much as 131 msec.  Downstream filtering with a time
\ constant of t seconds or less would result in significant ripple, allowing the
\ DACs time to settle to a resolution of only 8-bits.  Each doubling of the
\ downstream filtering time adds an additional bit of resolution to the DAC.
\ For example, with an interrupt service time set to 5 msec and a downstream
\ filter time constant of 80 msec or greater the filter time is 16 times the
\ minimum, and the DAC resolution would be increased by 4 bits (the base 2 log
\ of 16) to 12-bits.  This code sets the interrupt service time to a default
\ value of 5 msec; I recommend that a downstream filtering time constant of
\ 0.1 second be used.
\
\ Although this program successfully divides each 8-bit DAC step into 256
\ smaller steps, the DAC still does not have any better absolute accuracy or
\ linearity than it started with.  It is guaranteed to be monotonic to 8-bits,
\ and to have an integral nonlinearity of typically +/-1/2 lsb, over an ambient
\ temperature range of -40°C to +85°C.  However, if we examine its Data Sheet
\ (for the Analog Devices DAC-8841F) we find that its per step linearity error
\ over the entire temperature range is typically +/- 1/8 lsb.  So its step to
\ step accuracy is really approximately 10 to 11 bits, rather than just 8-bits.
\ The algorithm of this program allows us to increase its resolution to better
\ take advantage of that accuracy.
\
\ In tests of the program over the entire range of D/A output I find that there
\ is a maximum error of 1.5 millivolts for a 3.0 volt full scale output.  That
\ is, the maximum fractional error is one part in two thousand, or there is
\ 11 bits of accuracy.
```

```
\  ************************************************************************
\  **********                                                    **********
\  **********                    How to Use It                   **********
\  **********                                                    **********
\  ************************************************************************

\ To use this higher resolution option perform the following steps:
\
\    1.  Connect the QED Board to an Analog Conditioning Board and insert
\        100 microfarad capacitors in board at locations FD1, FD2...FD8
\        corresponding to the DAC channels you wish to use for greater
\        resolution.  This provides a 0.1 second output filter time constant.
\
\    2.  Insert gain resistors on the Analog Conditioning Board at locations
\        GD1, GD2...GD8 corresponding to the DAC channels you wish to use
\        to scale the maximum DAC output to voltages as great as 10.2 volts while
\        still sourcing up to 10 ma, and about 11.5 volts with little source
\        current capability.  (The maximum voltage is the supply voltage of
\        nominally 13. +/- .13 volts less the op-amp headroom of 1.5 volts.)
\
\    3.  If you wish to continue using some channels as 8-bit DACs rather than
\        higher resolution DACs, and you do not require update times more
\        rapid than 5 msec then you can continue to use this code.  You can just
\        use this software, but pass the desired 8-bit value to >Hi.Res.DAC
\        as the high order byte of its 16-bit integer input and set the low
\        order byte to zero.  This program will then update the DAC channel with
\        the desired 8-bit value within 5 msec, and there will be no flicker on
\        that channel.  You do not need a low pass filter on that channel.
\        If you need updates on an 8-bit channel more rapid than every 5 msec
\        while at the same time requiring high resolution channels then you
\        will need to modify this code appropriately.
\
\    4.  Download this text file.
\
\    5.  Execute Start.DAC  to initialize all DAC outputs to zero, attach the
\        DAC service interrupt, and start up the periodic DAC update every
\        5 msec.
\
\    6.  Send 16-bit integers to the desired DAC channel using the word
\        >Hi.Res.DAC which has the stack picture ( u\n -- ).  This top level word
\        takes as input a 16-bit unsigned value, u, and the DAC channel number, n.
\        The DAC output flickers between the most significant byte (MSB) and the
\        value one greater, MSB+1, with a duty cycle determined by the lower byte
\        (LSB), so that after adequate low-pass filtering an average analog voltage
\        is produced corresponding to one of 256 levels in between MSB and MSB+1.
\        The minimum output code is 0000 hex, and the maximum output code is FF00,
\        with all codes in between represented.  Note that codes between FF00 and
\        FFFF all result in the maximum output.  There are 255*256 = 65280 possible
\        inputs/outputs.  To produce an output voltage, V, given a full scale
\        output, Vmax, the unsigned value that must be sent to >Hi.Res.DAC would
\        be u = 65280 * V / Vmax.
```

```
\   7.  To calibrate a DAC channel send FF00 to the channel using >Hi.Res.DAC and
\       measure the voltage produced.  This voltage is then used to determine
\       the code sent to a DAC channel to produce any voltage.  For example,
\       suppose the voltage measured for channel #3 is 10.31 volts.  The
\       following code sends any desired voltage up to 10.31 volts to DAC
\       channel 3:
\
\               10.31 FCONSTANT Vmax#3    \ a constant to hold the maximum output
\                                         \ voltage
\               : >Channel#3 ( r -- )
\                   \ r is a floating point number representing the voltage to
\                   \ send to channel #3
\                   Vmax#3 F/
\                   0.0 FMAX  \ limit the input voltages to positive values
\                   1.0 FMIN  \ limit the input to less than the maximum
\                   65280. F* UFIXX 3 >Hi.Res.Dac
\                   ;
\
\       You would then send a voltage, for example 5.2 volts to that DAC channel
\       by executing:
\                5.2 >Channel#3
\
\   8.  To stop updating the DACs execute Stop.DAC and the interrupt service
\       routine will stop.  The DACs will be left set to the 8-bit level either
\       just greater or less than the 16-bit value sent to them.
\
\   9.  Because this routine uses the kernel word (>DAC) which does not
\       expect SPI resource conflicts, it does not call SPI.RESOURCE GET and
\       RELEASE.  Consequently, this routine should not be used when other
\       hardware (for example the 12-bit A/D) may also require the SPI.
\       If you want to use routine in conjunction with other code that
\       also uses the SPI then you should disable interrupts around the
\       other user of the SPI.  For example, to use the 12-bit A/D, instead of
\       calling the word A/D12.SAMPLE you would execute the sequence:
\
\              DISABLE.INTERRUPTS  (A/D12.SAMPLE) ENABLE.INTERRUPTS
\
\       in which the faster flavor of the A/D word is used ( (A/D12.SAMPLE) instead
\       of A/D12.SAMPLE ) which doesn't call GET or RELEASE because that's
\       not necessary if interrupts are disabled around all SPI using words.
\       For example if you want to measure a signal from a temperature
\       transducer on the A/D12 channel #4 you could write a word like the
\       following:
\
\              : Get.Temperature ( -- u )
\                \ u is an unsigned integer representing the temperature
\                -1      \ put a flag on the stack for single ended,
\                        \ unipolar conversion
\                4       \ the A/D channel number between 0 ad 7
\                DISABLE.INTERRUPTS (A/D12.SAMPLE) ENABLE.INTERRUPTS
\                ;
```

```
\
\ The following are descriptions of all the user words:

\ Start.DAC ( -- )
    \ Initializes DAC outputs to zero and starts up their periodic interrupt
    \ service.

\ Stop.DAC      ( -- )
    \ Stops the interrupt service of the DACs, leaving the DACs set to the
    \ nearest 8-bit approximation of their programmed value.

\ ReStart.DAC   ( -- )
    \ Restarts up the DAC's periodic interrupt service returning them to their
    \ prior 16-bit values.

\ >Hi.Res.DAC   ( u\n -- )
    \ Sends a 16-bit value to a DAC channel.  u is the unsigned 16-bit value to
    \ send to the DAC and n, where 1 <= n <= 8, is the channel number.  The
    \ minimum value for u is 0000 hex, and the maximum is FF00 (65280.), with
    \ all codes in between represented.  Note that codes between FF00 and FFFF
    \ (or greater than 65280.) all result in the maximum output. There are
    \ 255*256 = 65280 possible inputs/outputs.  To produce an output voltage, V,
    \ given a full scale output, Vmax, the unsigned value that must be sent to
    \ >Hi.Res.DAC would be u = 65280 * V / Vmax.  If V and Vmax are floating
    \ point values this would be computed as
    \       V F@ 0.0 FMAX Vmax F@ FMIN Vmax F@ F/ 65280. F* UFIXX

\ >PERIOD   ( u -- )
    \ u is and unsigned integer representing the number of 2 microsecond clock
    \ ticks of TCNT between updates of the PWMed D/A outputs.  A period of 5
    \ milliseconds would require u = 2500.  PERIODs less than 5 msec are not
    \ recommended as the servicing of all 8 DAC channels takes about 2.5 msec.
    \ If the PERIOD is too small interrupts will be missed and full TCNT
    \ rollover periods of 131 msec will be inserted as delays into the interrupt
    \ servicing.  The PERIOD is initialized to 5 msec by Start.DACs, and can
    \ be changed thereafter by >PERIOD.
\  ************************************************************************
\  **********                                                    **********
\  **********                    Warnings!                       **********
\  **********                                                    **********
\  ************************************************************************
\ Note: The following routine uses the kernel word (>DAC) which does not
\ expect SPI resource conflicts so does not call SPI.RESOURCE GET and
\ RELEASE.  Consequently, this routine should not be used when other hardware
\ (for example the 12-bit A/D) may also require the SPI.  If you want to use
\ this in an multitasking environment, or in conjunction with other code that
\ also uses the SPI then you would generally need to replace the call to (>DAC)
\ in the word Update.DAC.Values with a call to >DAC instead.  However, in this
\ case, that solution is unsufficient.  Because the call occurrs from within an
\ interrupt service routine, the GET would loop forever if the SPI is not free
\ because the interrupts that switch tasks are disabled while the interrupt
\ service routine runs (interrupts are not allowed to nest).  A solution is to
\ continue to use (>DAC) but to disable interrupts around any other uses of the
\ SPI in any tasks so that this interrupt service routine never attempts to use
\ the SPI while it is otherwise in use.  Because (>DAC) is called by this code
\ from within an interrupt service routine other users of the SPI will not be
\ able to interrupt (>DAC)'s use of it so that is not a concern.
```

```
\  ************************************************************************
\  **********                                                    **********
\  **********                      The Code                      **********
\  **********                                                    **********
\  ************************************************************************

ANEW <Hi.Res.DACs>

VARIABLE PERIOD          \ Holds the period between interrups as the number of
                         \ 2 microsecond ticks of TCNT.  A value of 2500
                         \ corresponds to 5 msec.
HEX
801A REGISTER: TOC3              8020 REGISTER: TCTL1
8022 REGISTER: TMSK1             8023 REGISTER: TFLG1

20  CONSTANT OC3.MASK
10  CONSTANT OC3.LEVEL.MASK
20  CONSTANT OC3.MODE.MASK

DECIMAL


Structure.Begin: DAC.Channel.Record
    TYPE.OF:
        INT->        +DAC.Value       \ for the 16-bit unsigned value to be written
        1 RESERVED                    \ for the low order byte of the Target.PWM
    OR.TYPE.OF:
        1 RESERVED                    \ for the MSB to be sent directly to the DAC
        INT->        +Target.PWM      \ for the lower order byte of the DAC value
                                      \ (the high byte of Target.PWM) to be PWMed
    TYPE.END
    2 RESERVED                        \ for use as Average.PWM by ?Update.DAC.PWM
    BYTE->           +Greater.DAC.Value  \ for the MSB+1 to be sent to the DAC
Structure.End


\ Create a single structure containing all eight DAC channel records:

Structure.Begin: All.DAC.Info
    8 DAC.Channel.Record STRUCTS->  +DAC.Info.Start
Structure.End

\ Now we instantiate (reserve space for) the DAC.Info structure in variable
\ space:

All.DAC.Info V.INSTANCE: DAC.Info
```

```
\ If we were not to use the above DAC data structure we would need the following
\ two variables for each DAC channel.  They are shown here only for clarity.
\ The variable Average.PWM must directly follow the variable Target.PWM
\ in memory.

\ VARIABLE Target.PWM   \ Holds the target PWM as an 8-bit number in the high order
\ byte.  The contents of the low order byte are irrelevant. Fetch or store to
\ this variable using C@ and C!.

\ VARIABLE Average.PWM \ Used internally by the algorithm; holds a running
\ average PWM. To update the PWM immediately set both Target.PWM and Average.PWM
\ to the new value.  To update the 256-bit long integral of the output most
\ quickly do not modify Average.PWM when Target.PWM is reset.  The Average.PWM
\ is set by setting its high order byte to the desired PWM (0-255) and setting
\ its low order byte to 255.

\ The following is a high level version of the corresponding assembly language
\ routine.  It is provided here for documentary purposes only:

\ : ?Update.DAC.PWM  ( xaddr -- Flag )
\   \ This word implements as PWM routine that optimally averages.
\   \ xaddr is the address of Target.PWM and xaddr+2 is the address of
\   \ Average.PWM, both as 16-bit unsigned integers.
\   \ Flag is the bit to be outputted, either true for high or false for low.
\   \ Each time Update.PWM is called Flag is set to either true or false
\   \ to maintain the proper average value for the PWM output.
\   [ BASE @ HEX ]
\   XDUP 2 XN+
\   Locals{ x&Average.addr x&Target.addr }
\   x&Target.addr @ FF00 AND x&Average.addr @ U>
\   IF
\       x&Average.addr @ x&Average.addr C@ - 00FF + x&Average.addr !
\       TRUE
\   ELSE
\       x&Average.addr @ x&Average.addr C@ -        x&Average.addr !
\       FALSE
\   ENDIF
\   [ BASE ! ]
\   ;

\ The following code is an assembly language version of the above high level
\ routine.
CODE ?Update.DAC.PWM   ( xaddr -- Flag )
    \ xaddr is the address of Target.PWM and xaddr+2 is the address of
    \ Average.PWM, both as 16-bit unsigned integers.
    \ Flag is the bit to be outputted.
    \ Each time Update.PWM is called Flag is set to either true or false
    \ to maintain the proper average value for the PWM output.
    BASE @ HEX
    02 IND,Y LDD                    \ Get the Target.PWM address
    02 IMM ADDD                     \ and increment by 2 and push it on the stack
    DEY DEY 00 IND,Y STD            \ as the Average.PWM address.
    02 IND,Y LDD                    \ Fetch,
    DEY DEY 00 IND,Y STD            \ and push the page too.
    02 IND,Y LDD                    \ Then XDUP the Average.PWM xaddress
        DEY DEY 00 IND,Y STD
```

```
02 IND,Y LDD
       DEY DEY 00 IND,Y STD
    CALL @                              \ @ Average.PWM and push it
    00 IND,Y LDAB   CLRA                \ get Average.PWM/256
    DEY DEY 00 IND,Y STD                \ push Average.PWM/256
    02 IND,Y LDD 00 IND,Y SUBD          \ replace Average.PWM/256 on tos with
    00 IND,Y STD                        \ Average.PWM - Average.PWM/256
    0A IND,Y LDD DEY DEY 00 IND,Y STD     \ put Target address on tos
    0A IND,Y LDD DEY DEY 00 IND,Y STD     \ put Target page on tos
    CALL @                              \ @ Target.PWM and push it
    00 IND,Y LDD                        \ get target from stack
    CLRB                                \ zero out the low order byte
    04 IND,Y CPD                        \ Target.PWM - Average.PWM
    HI IF,                              \ if Target.PWM > Average.PWM
        TRUE IMM LDD 0C IND,Y STD       \ set output to true and store it in place
                                        \ of target.addr on stack
        CLRA 02 IND,Y ADDD              \ add 255 to Average.PWM - Average.PWM/256
        0A IND,Y STD                    \ and store it in place of the target page on
                                        \ the stack
    ELSE,
        FALSE IMM LDD 0C IND,Y STD      \ else just set output false
        02 IND,Y LDD 0A IND,Y STD       \ and store Average.PWM - Average.PWM/256
                                        \ in place of the target page on the stack
    ENDIF,
    06 IMM LDAB ABY                     \ drop top three stack cells
                                        \ we now have ( flag\new.avg\avg.xaddr )
    CALL !                              \ store to Average.PWM
    RTS
    BASE !
    END.CODE

\ This high level code is provided to help document the following assembly
\ language version:

\ : Update.DAC.Values \ takes 2.36 msec
\    \ Steps through the DAC channels, calling ?Update.DAC.PWM to determine
\    \ which value to send to the DAC, and sends it.
\    8 0
\    DO
\        \ Call ?Update.DAC.PWM for a channel:
\        DAC.Info DAC.Channel.Record I * XN+ XDUP +Target.PWM ?Update.DAC.PWM
\        \ Depending on flag returned by ?Update.DAC.PWM send either the DAC's
\        \ eight bit value or a value one greater:
\        IF +Greater.DAC.Value ELSE +DAC.Value ENDIF
\        C@ I 1+ (>DAC)
\    LOOP
\    ;
```

```
CODE Update.DAC.Values ( -- ) \ This version takes 1.6 milliseconds.
    \ Steps through the DAC channels, calling ?Update.DAC.PWM to determine
    \ which value to send to the DAC, and sends it.
    CLRA
    01 IMM LDAB                         \ load counter value into B
    DEY DEY 00 IND,Y STD                \ and put counter on stack  ( counter )
    DAC.Info +Target.PWM SWAP           \ get xaddress of first target.pwm
    IMM LDD DEY DEY 00 IND,Y STD        \ put address on the stack
    IMM LDD DEY DEY 00 IND,Y STD        \ put page on the stack ( counter\xaddr )
    BEGIN,                              \ ( counter\xaddr )
        02 IND,Y LDD                    \ XDUP the xaddress
            DEY DEY 00 IND,Y STD
            02 IND,Y LDD
            DEY DEY 00 IND,Y STD
        02 IND,Y LDD                    \ XDUP the xaddress
            DEY DEY 00 IND,Y STD
            02 IND,Y LDD
            DEY DEY 00 IND,Y STD        \ ( counter\xaddr\xaddr\xaddr )
        CALL ?Update.DAC.PWM            \ ( counter\xaddr\xaddr\flag )
        00 IND,Y LDD                    \ test the flag
        EQ IF,
            04 IND,Y LDD                \ modify address to point to +DAC.Value
                01 IMM SUBD
                04 IND,Y STD
        ELSE,
            04 IND,Y LDD                \ modify address to point to +Greater.DAC.Value
                04 IMM ADDD
                04 IND,Y STD
        ENDIF,
        02 IMM LDAB ABY                 \ drop the flag
        CALL C@                         \ ( counter\xaddr\value )
        06 IND,Y LDD                    \ get the channel#, the counter
        DEY DEY 00 IND,Y STD            \ and push it and call (>DAC)
        CALL (>DAC)                     \ ( counter\xaddr )
        02 IND,Y LDD                    \ increment the target address
            06 IMM ADDD                 \ to point to next DAC channel record
            02 IND,Y STD
        05 IND,Y INC                    \ increment the counter
        09 IMM LDAA                     \ load terminal count into A
        05 IND,Y CMPA                   \ and compare it to the counter
    EQ UNTIL,                           \ to see if we're done
    06 IMM LDAB ABY                     \ drop the stack
    RTS
END.CODE
```

```
: >Hi.Res.DAC ( u\n -- )
    \ u is an unsigned 16-bit value to send to the DAC and 1 <= n <= 8 is
    \ the channel number.  The minimum value for u is 0000 hex, and
    \ the maximum is FF00, with all codes in between represented.  Note
    \ that codes between FF00 and FFFF all result in the maximum output.
    \ There are 255*256 = 65280 possible inputs/outputs.  To produce an output
    \ voltage, V, given a full scale output, Vmax, the unsigned value that must
    \ be sent to >Hi.Res.DAC would be u = 65280 * V / Vmax.
    1-  \ convert channel number to 0...7 range
    Locals{ &channel &value }

    \ First store zero to the Target.PWM and Average.PWM
    0\0 DAC.Info DAC.Channel.Record &channel * XN+ +Target.PWM 2!

    \ Store the full 16-bit value in the DAC.Info data structure
    \ This puts the MSB in the +DAC.Value location and the LSB in
    \ the +Target.PWM location
    &value
        DAC.Info DAC.Channel.Record &channel * XN+ +DAC.Value !

    \ Store the lower byte in the Average.PWM
    &value
        DAC.Info DAC.Channel.Record &channel * XN+ +Target.PWM 2XN+ C!

    \ Store the MSB+1 to the +Greater.DAC.Value
    &value -8 SCALE \ shift value over 8 places to get high order byte
    255 AND         \ blank out new top byte
    254 MIN 1+      \ increment by one but don't allow rollover from 255 to 256
        DAC.Info DAC.Channel.Record &channel * XN+ +Greater.DAC.Value C!
    ;

: >PERIOD ( n -- )
    \ n is the number of 2 microsecond clock ticks of TCNT between updates of
    \ the PWMed D/A outputs.  A period of 1/2 millisecond or 500 microseconds
    \ would require n = 250
    PERIOD !
    ;
: D/A.Update.Interrupt.Service
    OC3.MASK TFLG1 C!   \ Reset the OC3 interrupt flag so that new
    \ OC3 interrupts will be recognized. Because the flag is cleared by writing
    \ a one to it we can use a C! command without affecting the other bits.
    PERIOD @ TOC3 +!    \ Add the PERIOD to TOC3 to set the time at which
    \ the next interrupt occurrs.
    Update.DAC.Values   \ Update the 8 DAC channels
    ;
```

```
: Install.D/A.Update.Interrupt.Service
    OC3.MASK TMSK1 CLEAR.BITS           \ First we disable OC3 interrupts.
    OC3.MODE.MASK  TCTL1 CLEAR.BITS     \ Set the OC3 mode and level bits so that
    OC3.LEVEL.MASK TCTL1 CLEAR.BITS     \ the timer is disconnected from output pin.
    CFA.FOR D/A.Update.Interrupt.Service \ Attach the service routine.
        OC3.ID ATTACH
    OC3.MASK TFLG1 C!                   \ Clear the OC3 interrupt flag.
    \ We clear the OC3 interrupt flag by writing a one to it.  This seems counter-
    \ intuitive but that's the way the hardware works! It makes sense when
    \ we realize that we can just use a C! and not affect the other bits.
    \ OC3.MASK TMSK1 SET.BITS           \ Finally, we enable OC3 interrupts.
    \ Interrupts won't start until interrupts are also globally enabled by
    \ ENABLE.INTERRUPTS. Locally enabling the interrupts here is commented out
    \ because, although it's a good idea for some applications, for this
    \ application we don't want the interrupts starting until a separate
    \ word, called Start.Periodic.D/A.Update, is executed.
    ;

: Stop.DAC ( -- )
    OC3.MASK TMSK1 CLEAR.BITS   \ Disables the OC3 interrupts.
    ;

: Start.Periodic.D/A.Update
    Install.D/A.Update.Interrupt.Service
    2500 >PERIOD
    OC3.MASK TMSK1 SET.BITS     \ Enables the OC3 interrupts
    ENABLE.INTERRUPTS           \ and globally enables interrupts.
    ;

: Start.DAC ( -- )
    9 1
    DO
        0 I >Hi.Res.DAC
    LOOP
    Init.A/D12&DAC
    Start.Periodic.D/A.Update
    ;

: ReStart.DAC ( -- )
    Start.Periodic.D/A.Update
    ;

\ AXE out all words that the user doesn't need:

AXE DAC.Channel.Record      AXE +DAC.Value          AXE +Target.PWM
AXE +Greater.DAC.Value      AXE All.DAC.Info        AXE +DAC.Info.Start
AXE DAC.Info                AXE ?Update.DAC.PWM     AXE Update.DAC.Values
AXE PERIOD                  AXE D/A.Update.Interrupt.Service
AXE TOC3        AXE TCTL1           AXE TMSK1       AXE TFLG1
AXE OC3.MASK    AXE OC3.LEVEL.MASK  AXE OC3.MODE.MASK
AXE Install.D/A.Update.Interrupt.Service
AXE Start.Periodic.D/A.Update

\  ***************************************************************************
\  **********                                                       **********
\  **********                    End of Code                        **********
\  **********                                                       **********
\  ***************************************************************************
```

Mosaic Industries

5437 Central Ave Suite 1, Newark, CA 94560 Telephone: (510) 790-8222 Fax: (510) 790-0925