Summary

The following software describes how to use peripheral port A (PPA) for PWM or as a DAC.

```
\   ************************************************************************
\   ************************************************************************
\   **********                                                    **********
\   **********           Using Port PPA for PWM or as a 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 pulse width modulates bits of peripheral port A (PPA) so that
\ the apparent pseudo-period of the PWM waveform is as short as possible, and
\ with a resolution of one part in 256.  This enables, among other functions,
\ bits of PPA to be low pass filtered to create 8-bit DAC channels.
\ levels are produced.  Downstream of the digital output a low pass filter
\ smooths the PWM signal into an average value with a small residual ripple.
\ The PWM algorithm is optimal in the respect that it requires the least
\ averaging time to achieve a given level of resolution.  This property makes it
\ good for analog purposes, and it also makes it good for digital signals in
\ which the apparent PWM duty cycle should change as rapidly as possible.  This
\ algorithm gives an apparent duty cycle for times smaller than the full period
```

```
\ that is as good an approximation of the true duty cycle as it can be.  The
\ error in duty cycle (or resolution if used as a DAC) is limited to less than
\ 1/(2*n) where n is the number of consecutive updates of the port bits up to
\ n=256, the full period.  For n=256 the duty cycle error is zero because the
\ duty cycles are produced exactly with a period of 256 updates.


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

\ For a detailed description of the PWM algorithm used see the Mosaic Industries
\ QED Application Note "MI-AP-056: A PWM Algorithm with Optimal Averaging
\ Properties".
\
\ An interrupt service routine services PPA every t seconds, where t can
\ be adjusted from a minimum of about 2 msec (it takes 1.1 msec to service all
\ the PPA bits) to as much as 131 msec.  If used as a DAC downstream filtering
\ is required.  If the filtering time constant, Tau, is less than 256*t
\ there is significant ripple.  For every doubling of Tau beyond Tau=t
\ the ripple is reduced a factor of two and a bit or resolution is gained, up
\ to Tau=256*t.  Beyond that the ripple continues to decrease but no more
\ resolution is gained, and the settling time is needlessly lengthened.
\
\ For example, with an interrupt service time set to 2 msec and a downstream
\ filter time constant of 0.5 sec or greater the filter time is 256 times the
\ minimum, and the DAC resolution is a full 8-bits.  This code sets the
\ interrupt service time to a default value of 2 msec; I recommend that a
\ downstream filtering time constant of 0.5 seconds be used.
\

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


\
\   1.  Download this text file.
\
\   2.  Configure PPA for output using
\           TRUE flag INIT.PIA
\       The TRUE initializes PPA as output.
\       flag = true indicates that upper PPC is output.  Choose upper PPC
\       to be input or output as you need.
```

```
\
\   6.  Execute
\           Start.PPA.PWM
\       to install the interrupt service routine and start periodic updates of
\       the PPA bits.  No PWM waveforms are generated until you specify duty
\       cycles using >PPA.PWM.
\
\   7.  Execute
\           >PPA.PWM    ( u\n -- )
\       to send a PWM duty cycle to any PPA output pin.  This top level word
\       takes as input an 8-bit unsigned value, u, as the PWM duty cycle
\       and the PPA pin number, n, where 0 <= n <= 7.  Other output pins are
\       unaffected.  Values of u from 0 to 256 are allowed, with 256
\       indicating a continuously ON condition.  The duty cycle is given by
\       D.C.= u/256.  The time between updates of the output pins is determined
\       by the value of PERIOD, which holds the time as the number of 2 usec
\       ticks of the TCNT clock.
\
\   8.  If you are filtering the PWM outputs for digital to analog conversion
\       then use the word
\           >PPA.DAC    ( u\n -- )
\       instead of
\           >PPA.PWM    ( u\n -- )
\       This word will cause the PWM signal to over or undershoot appropriately
\       whenever there is a change of duty cycle to compensate for the filter
\       delay so that the DAC output is updated as rapidly as possible.  For this
\       to work optimally the filter time constant should be 256 times the update
\       interval.
\
\       Also, you must install an analog filter with a time constant of 0.5 sec
\       (256 times the update interval of 2 msec) or more on the PPA pin.  A
\       50 Kohm resistor and 10 uF capacitor will provide the proper time
\       constant.  If a low impedance output is needed the filter should be followed
\       by an op-amp voltage follower, or the filter and op-amp may be combined
\       as an active filter.
\
\   9.  To stop the PWM waveforms execute
\               Stop.PPA.PWM
\       and the interrupt service routine will stop.  The PPA output pins that
\       had been used will be left set to zero, except for any that had been set
\       fully on (high) by sending a duty cycle of 256.  They will be left high.
\
\
\ The following are descriptions of all the user words:

\ Start.PPA.PWM ( -- )
    \ Starts up the periodic interrupt service of the PPA bits but doesn't
    \ actually modify any bits until >PPA.PWM is executed.
```

```
\ Stop.PPA.PWM      ( -- )
    \ Stops the interrupt service of the PPA bits, leaving the bits that had
    \ been PWMing set to the zero, and not affecting other bits.

\ ReStart.PPA.PWM   ( -- )
    \ Restarts up the PIA's periodic interrupt service returning the PIA bits
    \ that had been PWMing before Stop.PPA.PWM had been executed back to their
    \ PWM action.

\ >PPA.PWM  ( u\n -- )
    \ u is an unsigned 8-bit PWM value to send to PPA and 0 <= n <= 7 is
    \ the PPA output bit number.  Values of u from 0 to 256 are allowed, with 256
    \ indicating a continuously ON condition.  The time between updates of the
    \ output pins is determined by the value of PERIOD, which holds the time
    \ as the number of 2 usec ticks of the TCNT clock.

\ >PERIOD   ( u -- )
    \ u is and unsigned integer representing the number of 2 microsecond clock
    \ ticks of TCNT between updates of the PWMed PPA outputs.  A period of 2
    \ milliseconds would require u = 1000.  PERIODs less than 2 msec are not
    \ recommended as the servicing of PPA takes about 1.1 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 2 msec by Start.PPA.PWM, and can
    \ be changed thereafter by >PERIOD.


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

ANEW <PPA.PWM>
6 WIDTH !

VARIABLE PERIOD         \ Holds the period between interrups as the number of
                        \ 2 microsecond ticks of TCNT.  A value of 1000
                        \ corresponds to 2 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: PPA.Record
    BYTE->  +PWM.Value  \ Only a single byte is used for the 8-bit
                        \ unsigned value to be written to the output pin
                        \ as a PWM signal for values from 0 to 255.
    2 RESERVED          \ For use as a running average PWM for this channel.
Structure.End


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

Structure.Begin: Info.for.PPA.PWM
    8 PPA.Record * RESERVED
Structure.End

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

Info.for.PPA.PWM V.INSTANCE: PPA.PWM.Data

Variable PPA.Bits.Used  \ a byte mask with bits set corresponding to PPA
                        \ output bits used for PWM.  Other bits are unaffected.

\ If we were not to use the above data structure we would need the following
\ two variables for each PWM 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.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.PWM   ( xaddr -- Flag )
    \ xaddr is the address of Target.PWM as a single byte and xaddr+1 is the
    \ address of Average.PWM as a 16-bit unsigned integer.  (In the high level
    \ example above they were both 16-bit integers.  In this version the Target
    \ value is assumed to take only one byte in memory. )
    \ Flag is the bit to be outputted, true indicates one or high.
    \ 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
    01 IMM ADDD                     \ and increment by 1 and push it on the stack
    DEY DEY 00 IND,Y STD            \ as the Average.PWM address.
    02 IND,Y LDD                    \ Fetch and push the page too.
        DEY DEY 00 IND,Y STD        \
    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 C@                         \ C@ Target.PWM and push it
    01 IND,Y LDAA                   \ get target from stack into high byte of D
    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.PPA.Bits ( -- ) \ Takes about 2.1 msec to execute
\   \ Steps through the PPA bits, calling ?Update.PWM to determine
\   \ which value to send to the bit, and accumulating them in PPA.Value, then
\   \ sending PPA.Value to PPA all at once.
\   0 Locals{ &PPA.Value }
\   1       \ bit place counter, gets doubled each iteration
\   8 0     \ for each bit from lsb to msb
\   DO
\       \ Call ?Update.PWM for each channel:
\       PPA.PWM.Data PPA.Record I * XN+ +PWM.Value ?Update.PWM
\       \ Depending on flag returned by ?Update.PWM send either a one or a zero:
\       IF DUP &PPA.Value + TO &PPA.Value ENDIF
\       2*  \ shift place counter
\   LOOP
\   DROP
\   &PPA.Value PPA.Bits.Used C@ PPA PIA.CHANGE.BITS
\   ;

CODE Update.PPA.Bits ( -- ) \ Takes about 1.1 msec to execute
    \ Steps through the PPA bits, calling ?Update.PWM to determine
    \ which value to send to the bit, and sends it.
    01 IMM LDAB                     \ load counter value into B
    DEY 00 IND,Y STAB               \ and put it on stack  ( counter )
    00 IMM LDAB                     \ initialize PPA.Value
    DEY 00 IND,Y STAB               \ and put it on stack  ( counter\ppa.value )

    PPA.PWM.Data +PWM.Value SWAP    \ get xaddress of first PWM value
    IMM LDD DEY DEY 00 IND,Y STD    \ put address on the stack
    IMM LDD DEY DEY 00 IND,Y STD    \ put page on stack ( counter\ppa.value\xaddr )
```

```
BEGIN,                              \ ( counter\ppa.value\xaddr )
        02 IND,Y LDD                  \ XDUP the xaddress
            DEY DEY 00 IND,Y STD
            02 IND,Y LDD
            DEY DEY 00 IND,Y STD      \ ( counter\ppa.value\xaddr\xaddr )
        CALL ?Update.PWM              \ ( counter\ppa.value\xaddr\flag )
        00 IND,Y LDD                  \ test the flag
        NE IF,
            06 IND,Y LDAA             \ add counter to ppa.value
            07 IND,Y ADDA
            06 IND,Y STAA
        ENDIF,
        02 IMM LDAB ABY               \ drop the flag ( counter\ppa.value\xaddr )
        PPA.Record IMM LDD            \ get offset
        02 IND,Y ADDD                 \ increment the address to point to the
        02 IND,Y STD                  \ next desired PWM
        05 IND,Y ASL                  \ increment the counter
    CS UNTIL,                         \ are we done?

    04 IND,Y LDAB CLRA                \ set up the stack for PIA.CHANGE.BITS
        04 IND,Y STD                  \ put the value on the stack
    PPA.Bits.Used                     \ get xaddress of PPA.Bits.Used
    IMM LDD 00 IND,Y STD              \ put page on the stack
    IMM LDD 02 IND,Y STD              \ put addr on the stack ( ppa.value\xaddr )
    CALL C@                           \ get PPA.Bits.Used ( ppa.value\ppa.bit.mask )
    PPA SWAP                          \ get xaddress of PPA
    IMM LDD DEY DEY 00 IND,Y STD      \ put address on the stack
    IMM LDD DEY DEY 00 IND,Y STD      \ put page on stack
                                      \ stack now: (ppa.value\ppa.bit.mask\xaddr)
    CALL PIA.CHANGE.BITS              \ send out the bits
    RTS
END.CODE
: >PPA.PWM ( u\n -- )
    \ u is an unsigned 8-bit PWM value to send to PPA and 0 <= n <= 7 is
    \ the PPA output bit number.  Values of u from 0 to 256 are allowed, with 256
    \ indicating a continuously ON condition.  The time between updates of the
    \ output pins is determined by the value of PERIOD, which holds the time
    \ as the number of 2 usec ticks of the TCNT clock.
    Locals{ &channel &value }
    &channel 0 MAX 7 MIN TO &channel
    &value 256 =
    IF
        1 &channel SCALE PPA.Bits.Used CLEAR.BITS
        1 &channel SCALE PPA PIA.SET.BITS
    ELSE
        &value PPA.PWM.Data PPA.Record &channel * XN+ +PWM.Value C!
        \ The following two lines are used if PWM update must be immediate
        \ rather than allowing over/undershoot for downstream averaging:
        &value PPA.PWM.Data PPA.Record &channel * XN+ +PWM.Value 1XN+ C!
             0 PPA.PWM.Data PPA.Record &channel * XN+ +PWM.Value 2XN+ C!
        \ we set PPA.Bits.Used after setting the value so that we don't have
        \ transient update problems
        1 &channel SCALE PPA.Bits.Used SET.BITS
    ENDIF
    ;
```

```
: >PERIOD ( n -- )
    \ n is the number of 2 microsecond clock ticks of TCNT between updates of
    \ the PWMed D/A outputs.  A period of 2 millisecond or 2000 microseconds
    \ would require n = 1000
    PERIOD ! ;

: PWM.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.PPA.Bits      \ Update the PPA output bits
    ;

: Install.PWM.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 PWM.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.PPA.Update is executed.
    ;
: Stop.PPA.PWM ( -- )
    \ Stops the PWMing outputs and sets them to zero.
    OC3.MASK TMSK1 CLEAR.BITS   \ Disables the OC3 interrupts.
    0 PPA.Bits.Used C@ PPA PIA.CHANGE.BITS
    8 0
    DO
        0 PPA.PWM.Data PPA.Record I * XN+ +PWM.Value      C!
        0 PPA.PWM.Data PPA.Record I * XN+ +PWM.Value 1XN+  !
    LOOP
    ;

: (Start.PPA.Update)
    Install.PWM.Update.Interrupt.Service
    1000 >PERIOD
    OC3.MASK TMSK1 SET.BITS     \ Enables the OC3 interrupts
    ENABLE.INTERRUPTS           \ and globally enables interrupts.
    ;
```

```
: Start.PPA.PWM ( -- )
    \ Sets no bits for PWM output and initializes all bits for zero duty cycle.
    0 PPA.Bits.Used C!
    8 0
    DO
        0 PPA.PWM.Data PPA.Record I * XN+ +PWM.Value      C!
        0 PPA.PWM.Data PPA.Record I * XN+ +PWM.Value 1XN+  !
    LOOP
    (Start.PPA.Update)
    ;

: ReStart.PPA.PWM ( -- )
    \ Restarts the PWM output updates but does not send them any duty cycles.
    \ That must be done with >PPA.PWM or >PPA.DAC
    (Start.PPA.Update)
    ;

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

AXE PERIOD
AXE TOC3         AXE TCTL1            AXE TMSK1       AXE TFLG1
AXE OC3.MASK     AXE OC3.LEVEL.MASK   AXE OC3.MODE.MASK

AXE PPA.Record           AXE +PWM.Value
AXE Info.for.PPA.PWM     AXE PPA.PWM.Data        AXE PPA.Bits.Used
AXE ?Update.PWM          AXE Update.PPA.Bits
AXE Install.PWM.Update.Interrupt.Service
AXE PWM.Update.Interrupt.Service
AXE (Start.PPA.Update)

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


Mosaic Industries

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