APPLICATION NOTES

4D SYSTEMS
TURNING TECHNOLOGY INTO ART

ViSi Quadrature Input

DOCUMENT DATE: 27th APRIL 2019
DOCUMENT REVISION: 1.1

Description

This Application note is intended to demonstrating to the user the set-up, initialization and operation of the built-in quadrature input feature of the Diablo16 display module.

- The target screen can be any of the following Diablo16 touch display modules:

gen4-uLCD-24D series	gen4-uLCD-28D series	gen4-uLCD-32D series
gen4-uLCD-38D series	gen4-uLCD-43D series	gen4-uLCD-50D series
gen4-uLCD-70D series		
uLCD-35DT	uLCD-43D Series	uLCD-70DT

Visit www.4dsystems.com.au/products to see the latest display module products that use the Diablo16 processor.

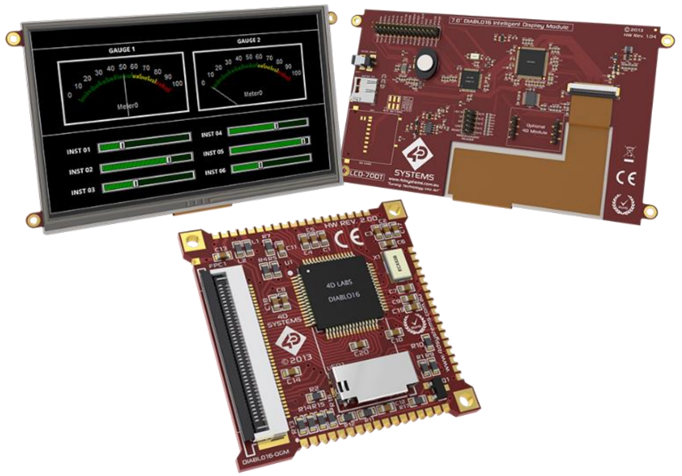
- 4D Programming Cable / µUSB-PA5/uUSBPA5-II for non-gen4 displays (uLCD-xxx)
- 4D Programming Cable & gen4-IB / 4D-UPA / gen4-PA for gen4 displays (gen4-uLCD-xxx)
- Workshop 4 IDE (installed according to the installation document)
- micro-SD (µSD) memory card
- When downloading an application note, a list of recommended application notes is shown. It is assumed that the user has read or has a working knowledge of the topics presented in these recommended application notes.

Content

Application Overview

This document is focused on the fundamental usage of the quadrature input feature of the Diablo16 Embedded Graphics Processor. A quadrature input or encoder, also known as an incremental rotary encoder, can be used to measure the speed and direction of a rotating shaft. Quadrature encoders can use different types of sensors, optical and Hall Effect are both commonly used. Diablo16 OGM and Diablo16 display module has a total of 2 quadrature input channels. These quadrature inputs are supported in several GPIO terminals.

The Diablo16 Embedded Graphics Processor



Driving the display and peripherals is the Diablo16 embedded graphics processor, a very capable and powerful chip which enables stand-alone functionality, programmed using the 4D Systems Workshop 4 IDE Software. The Workshop IDE enables graphic solutions to be constructed rapidly and with ease due to its design being solely for 4D's graphics processors.

The Diablo16 Processor offers considerable FLASH and RAM upgrades over the PICASO processor, and also provides map-able functions such as I2C, SPI, Serial, PWM, Pulse Out, and Quadrature Input, to various GPIO, and also provide up to 4 Analogue Input channels.

Setup Procedure

For instructions on how to launch Workshop 4, how to open a **ViSi** project, and how to change the target display, kindly refer to the section "**Setup Procedure**" of the application note

ViSi Getting Started - First Project for Picaso and Diablo16

Create a New Project

For instructions on how to create a new **ViSi** project, please refer to the section "**Create a New Project**" of the application note

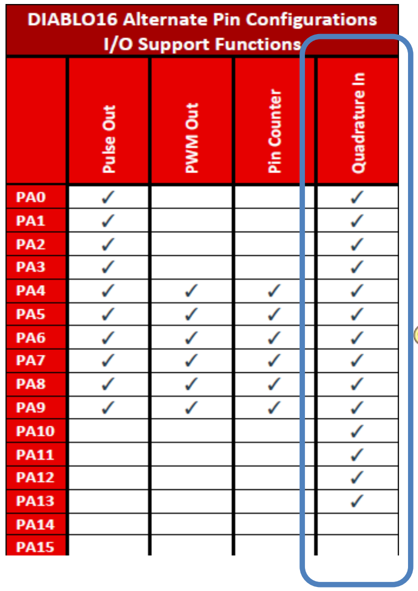
ViSi Getting Started - First Project for Picaso and Diablo16

Design the Project

To create a simple program that will be able to activate and initialize the Diablo16 quadrature input, we will need to use some commands enlisted in the DIABLO 4DGL Internal Functions.

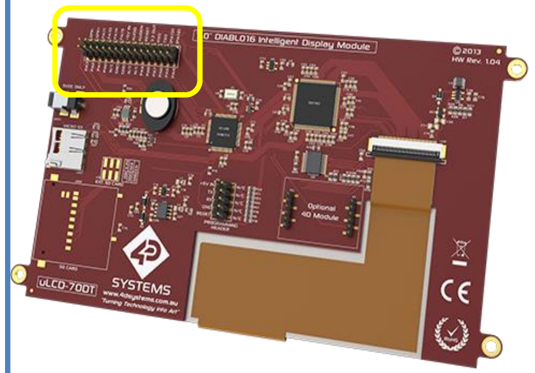
The Diablo16 embedded graphics processor has a total of two quadrature input channels. These quadrature input channels are map-able to the 14 general purpose input-output terminals. Referring to the chart below, we will see that these output pins can be used alternately with other processor I/O function support.

These general purpose I/O pins for the quadrature input does not follow any arrangements. The pins may be used readily, after configuring the function setup, according to the user's needs. These I/O pins are located similar to the image.

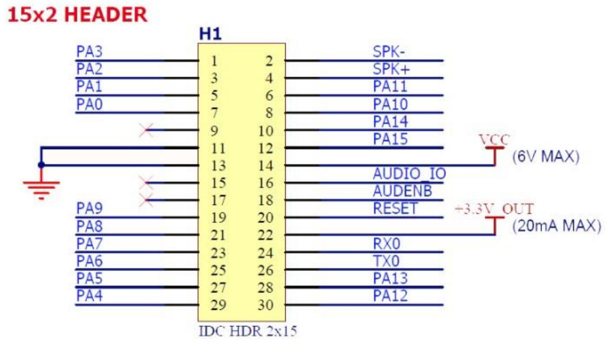


DIABLO16 Alternate Pin Configurations
I/O Support Functions

	Pulse Out	PWM Out	Pin Counter	Quadrature In
PA0	✓			✓
PA1	✓			✓
PA2	✓			✓
PA3	✓			✓
PA4	✓	✓	✓	✓
PA5	✓	✓	✓	✓
PA6	✓	✓	✓	✓
PA7	✓	✓	✓	✓
PA8	✓	✓	✓	✓
PA9	✓	✓	✓	✓
PA10				✓
PA11				✓
PA12				✓
PA13				✓
PA14				
PA15				



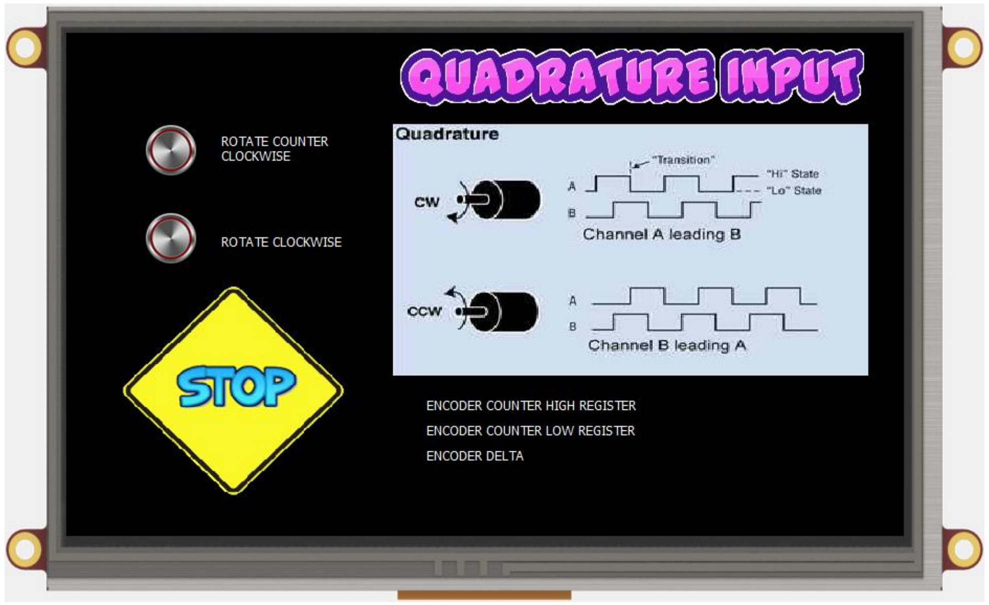
The Diablo16 configurable I/O are a group of 3.3 volts TTL level terminals but these are tolerant to a maximum of 5 volts. Anything greater than or less than the specified operating voltage may prohibit proper communication or even damage the embedded graphics processor.



The 2x15 male header pin assignment of the Diablo16 70DT. The previous table on pin function configuration option are lumped together with several other specific purpose pins.

The ViSi - based application project

For this application project a pair of 4D buttons and images are used. Add objects by navigating to the Layout the objects similar to the one below.

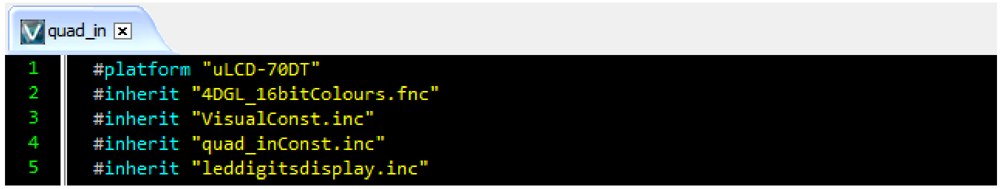


After all the objects have been laid-out, let's continue with the other half which involves the coding of the project. This will be presented in a sectional manner so as not to create confusion with the project.

For an in-depth detail of the functions used in this application note please refer to the Diablo16 Internal Functions Reference Manual.

The include section

This project starts with the identification of the platform being used as declared by the #platform function. For the program to be able to function properly files are included herein using the #inherit function.



```
#platform "uLCD-70DT"
#inherit "4DGL_16bitColours.fnc"
#inherit "VisualConst.inc"
#inherit "quad_inConst.inc"
#inherit "leddigitsdisplay.inc"
```

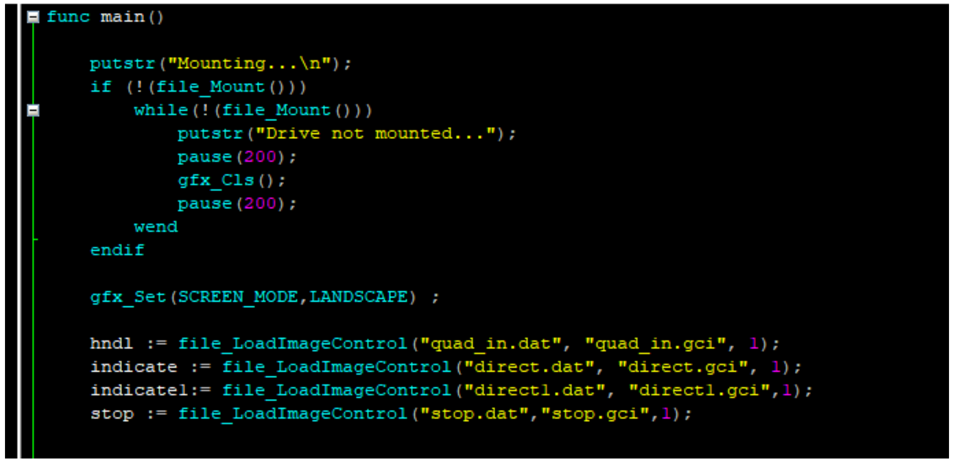
In this application note, quad_inConst.inc, contains all the information about the objects that are used in the project. Meanwhile, the leddigitsdisplay.inc contains the function for the proper operation of the led digits objects.

The main program

The main program for this projects contains several sections: the mounting of the micro-SD card, the initial displaying and image touch setup for objects, the setup for quadrature and input-output control, the repeat-forever loops which contains the continuous registry reading and touch conditions. Also, the main program calls out sub-routine functions that perform a particular functions.

The micro-SD initialization

Let's start with the initialization of the uSD card. The uSD card contains all the image information about the objects used in the project. The object information and data are saved under a *.DAT and a *.GCI filename extension which is copied to the uSD during project compilation. Mounting of the disk in this application note was done using the following set of program statements.

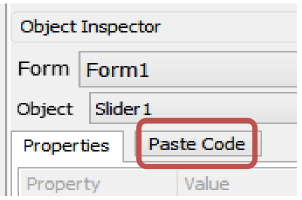


```
func main()

    putstr("Mounting...\n");
    if (!(file_Mount()))
        while(!(file_Mount()))
            putstr("Drive not mounted...");
            pause(200);
            gfx_Cls();
            pause(200);
        wend
    endif

    gfx_Set(SCREEN_MODE,LANDSCAPE) ;

    hndl := file_LoadImageControl("quad_in.dat", "quad_in.gci", 1);
    indicate := file_LoadImageControl("direct.dat", "direct.gci", 1);
    indicate1:= file_LoadImageControl("direct1.dat", "direct1.gci",1);
    stop := file_LoadImageControl("stop.dat","stop.gci",1);
```

When starting a new project in the ViSi environment these set of statements are already included in the coding area. The last part of this set of statements uses a function file_LoadImageControl() to call on the object data/information files on the uSD drive. This initializes the data to be called in using the variable ‘hndl’. Likewise, additional variable were also assigned as image control for some graphics composer generated GCI and DAT files.

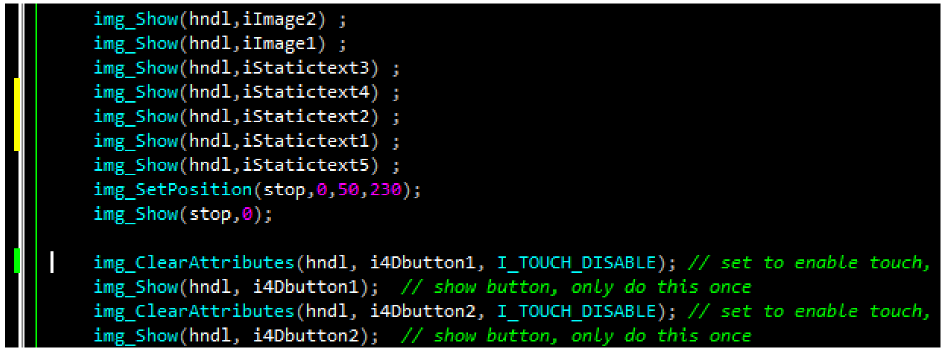
Having been able to load and initialize the uSD drive, the processor is now able to access the information stored therein. As mentioned from the previous section, the filenames with an extension of DAT and GCI has the image data and information.

Therefore, the next part of the main program is to display all the objects that were placed on the Workshop IDE form viewer. To do so, a special button from the Object Inspector can help reduce the time of coding of this part. The ‘Paste Code” simply pastes object code into the coding area.



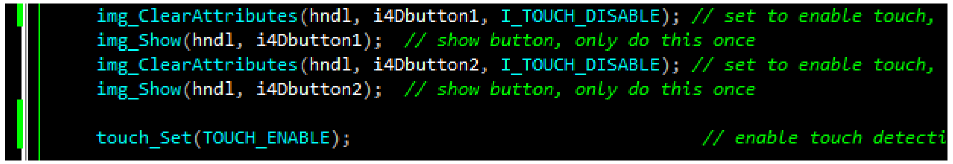
The initial image display and image touch setup segment

In this part of the program, the img_Show() function calls out the object image and information found in the microSD drive. This set of statements displays every object that were included in the application project. The displaying of the images is directly done using the img_Show() function.



```
img_Show(hndl,iImage2) ;
img_Show(hndl,iImage1) ;
img_Show(hndl,iStatictext3) ;
img_Show(hndl,iStatictext4) ;
img_Show(hndl,iStatictext2) ;
img_Show(hndl,iStatictext1) ;
img_Show(hndl,iStatictext5) ;
img_SetPosition(stop,0,50,230);
img_Show(stop,0);

img_ClearAttributes(hndl, i4Dbutton1, I_TOUCH_DISABLE); // set to enable touch,
img_Show(hndl, i4Dbutton1);  // show button, only do this once
img_ClearAttributes(hndl, i4Dbutton2, I_TOUCH_DISABLE); // set to enable touch,
img_Show(hndl, i4Dbutton2);  // show button, only do this once
```

In this segment of the program statements. We have displayed all the static texts, 4D button widgets and an image file. The image file handled with the variable ‘stop’ is directed to be displayed at the 50th of x pixels and 230th of the y pixels. Moving to the next part of the main program, this segment is all related to the image touch detection setup.

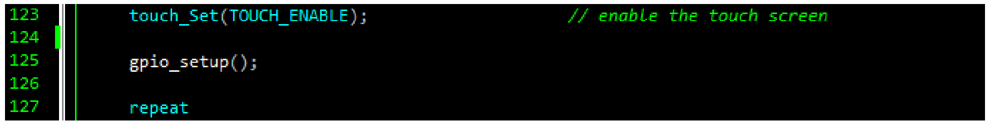


```
img_ClearAttributes(hndl, i4Dbutton1, I_TOUCH_DISABLE); // set to enable touch,
img_Show(hndl, i4Dbutton1);  // show button, only do this once
img_ClearAttributes(hndl, i4Dbutton2, I_TOUCH_DISABLE); // set to enable touch,
img_Show(hndl, i4Dbutton2);  // show button, only do this once

touch_Set(TOUCH_ENABLE);                                  // enable touch detecti
```

This statements uses the img_ClearAttributes() function. The primary objective of this set of statement is to enable the touch detection for the button images. The button images on this project serves as an input objects which utilizes the touch feature of the device. At the end, of this segment we would notice that the touch feature of the device was enabled using the touch_Set(TOUCH_ENABLE) statement.

The GPIO setup sub-routine

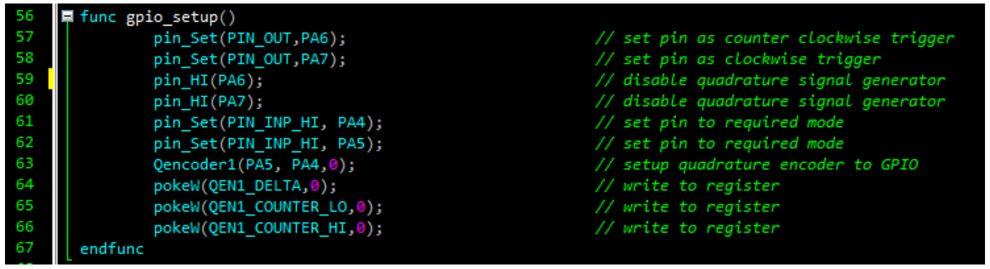
The purpose of this sub-routine is to simplify the presentation of the program statements in this document. When using the GPIO pins for a special function, it is always best that the direction of data is assigned.



```
    touch_Set(TOUCH_ENABLE);                        // enable the touch screen

    gpio_setup();

    repeat
```

Below is the sub-routine being called in by the gpio_setup() function. It can be observed that the GPIO PA6 and PA7 are assigned to control the direction of the encoder generator. These quadrature control signal generator is then temporarily disabled through setting the PA6 and PA7 low.

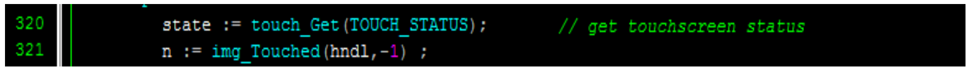
The quadrature input channel was initialized using the Qencoder() function. The pokeW() function at the end of this routine writes to registers related to the quadrature encoder channel.



```
func gpio_setup()
        pin_Set(PIN_OUT,PA6);                               // set pin as counter clockwise trigger
        pin_Set(PIN_OUT,PA7);                               // set pin as clockwise trigger
        pin_HI(PA6);                                        // disable quadrature signal generator
        pin_HI(PA7);                                        // disable quadrature signal generator
        pin_Set(PIN_INP_HI, PA4);                           // set pin to required mode
        pin_Set(PIN_INP_HI, PA5);                           // set pin to required mode
        Qencoder1(PA5, PA4,0);                              // setup quadrature encoder to GPIO
        pokeW(QEN1_DELTA,0);                                // write to register
        pokeW(QEN1_COUNTER_LO,0);                           // write to register
        pokeW(QEN1_COUNTER_HI,0);                           // write to register
endfunc
```

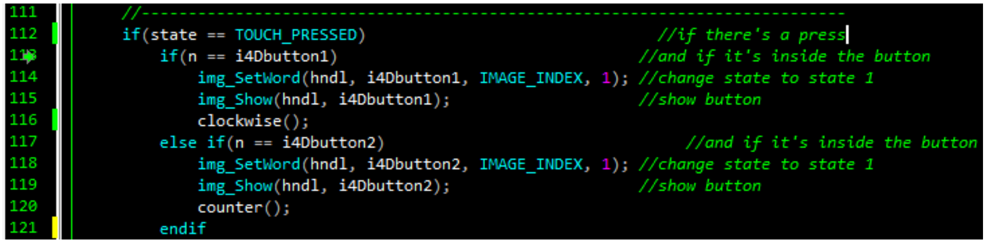
The repeat-forever image touch detect loop

At this end part of the main program, the routine was to detect any activity on the touch screen. Three touch states were included in this repetitive routine: the detection for a pressed state, a released state, and a moving state. Prior to the touch detection, a variable 'n' is assigned to store temporary image touch detection result. The img_Touched() function checks the object being touched and return the name of the object enlisted in the variable 'hndl'.



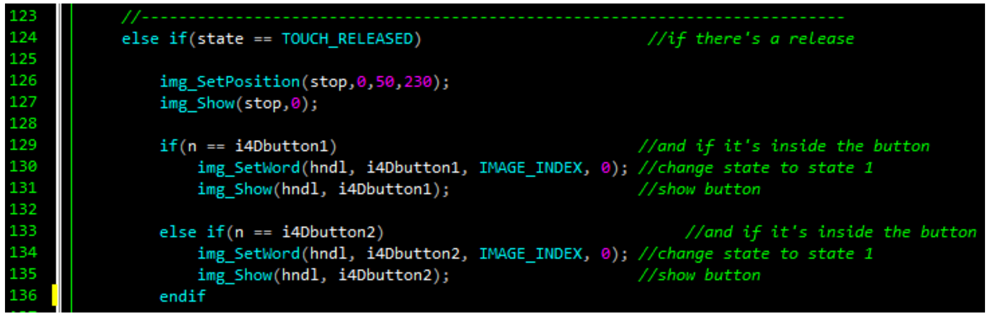
```
        state := touch_Get(TOUCH_STATUS);       // get touchscreen status
        n := img_Touched(hndl,-1) ;
```

Moving to the touch detection routines, two touch states were utilized in this document. The touch state are detected through the touch_Get() function. This function returns the status into a variable 'state'. If one of the conditions is met, the processor immediately executes the statements included therein.



```
        //----------------------------------------------------------------------
        if(state == TOUCH_PRESSED)                                  //if there's a press
            if(n == i4Dbutton1)                                    //and if it's inside the button
                img_SetWord(hndl, i4Dbutton1, IMAGE_INDEX, 1); //change state to state 1
                img_Show(hndl, i4Dbutton1);                    //show button
                clockwise();
            else if(n == i4Dbutton2)                                 //and if it's inside the button
                img_SetWord(hndl, i4Dbutton2, IMAGE_INDEX, 1); //change state to state 1
                img_Show(hndl, i4Dbutton2);                    //show button
                counter();
            endif
```

The repeat-forever loop includes a condition for a 'touch released' state. If the released condition for touch is satisfied, the index '0' of the image handler 'stop' is displayed. Included in this segment is a pair of if-conditions that results to displaying of the 4Dbuttons in their '1' index. For the if-condition statements, two subroutines are called – the clockwise() and counter(). These sub-routines will be further explained in details in the next part of this document.

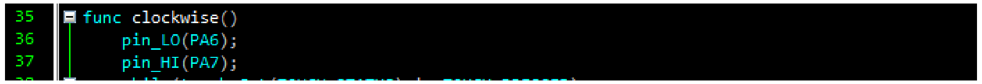


```
    //-----------------------------------------------------------------------------
    else if(state == TOUCH_RELEASED)                        //if there's a release

        img_SetPosition(stop,0,50,230);
        img_Show(stop,0);

        if(n == i4Dbutton1)                                 //and if it's inside the button
            img_SetWord(hndl, i4Dbutton1, IMAGE_INDEX, 0); //change state to state 1
            img_Show(hndl, i4Dbutton1);                     //show button

        else if(n == i4Dbutton2)                                //and if it's inside the button
            img_SetWord(hndl, i4Dbutton2, IMAGE_INDEX, 0); //change state to state 1
            img_Show(hndl, i4Dbutton2);                     //show button
        endif
```

Let us take the above statements under the 'touch released' if-condition. From the start of the repeat-forever loop, the img_Touched() function saves the result of an image touch to a variable 'n'. Subsequently, when the touch on the image is released the image name is returned. This result is then checked on an if-conditional loop. Whenever a condition is satisfied, the statements results to the image buttons being displayed with their '0' index.

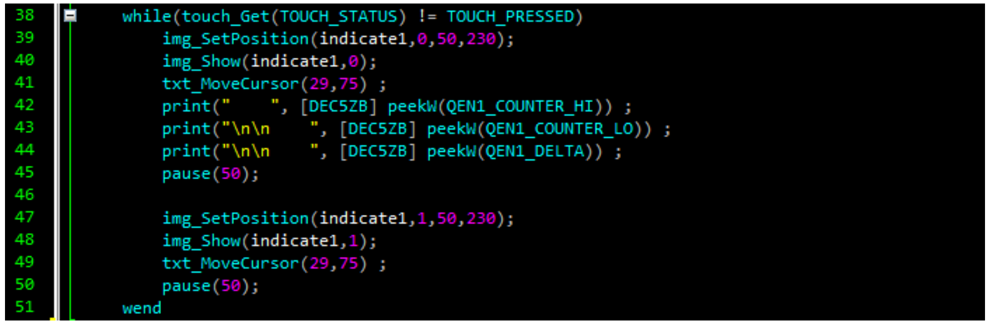
The clockwise() sub-routine

Whenever the touch detection results to the 4Dbutton1 image being touched, the processor is directed to run the statements contained in the clockwise() sub-routine.



```
func clockwise()
    pin_LO(PA6);
    pin_HI(PA7);
```

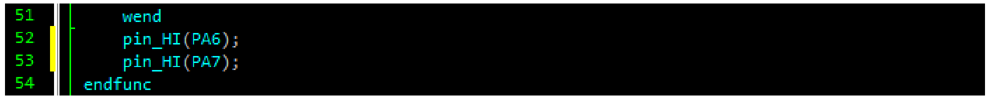
The first part of the sub-routine sets the PA6 and PA7 with a LOW and a HIGH logic, respectively. This enable the 'going positive' turn for the quadrature signal generator. The signal generator is continuously run while the 4Dbutton image is not pressed.



```
    while(touch_Get(TOUCH_STATUS) != TOUCH_PRESSED)
        img_SetPosition(indicate1,0,50,230);
        img_Show(indicate1,0);
        txt_MoveCursor(29,75) ;
        print("    ", [DEC5ZB] peekW(QEN1_COUNTER_HI)) ;
        print("\n\n    ", [DEC5ZB] peekW(QEN1_COUNTER_LO)) ;
        print("\n\n    ", [DEC5ZB] peekW(QEN1_DELTA)) ;
        pause(50);

        img_SetPosition(indicate1,1,50,230);
        img_Show(indicate1,1);
        txt_MoveCursor(29,75) ;
        pause(50);
    wend
```

While the touch related condition is not achieved, the processor runs the statements inside the loop. Referring to the while-condition statements above, the registers for the quadrature encoder channel 1 are repetitively read using the peekW() function. Also, it is repetitively displayed using the print function. For the purpose of demonstration, the index of the image handler variable 'indicate1' is also changed every 50 milliseconds. This produces the visual graphics to represent the direction of the turn.

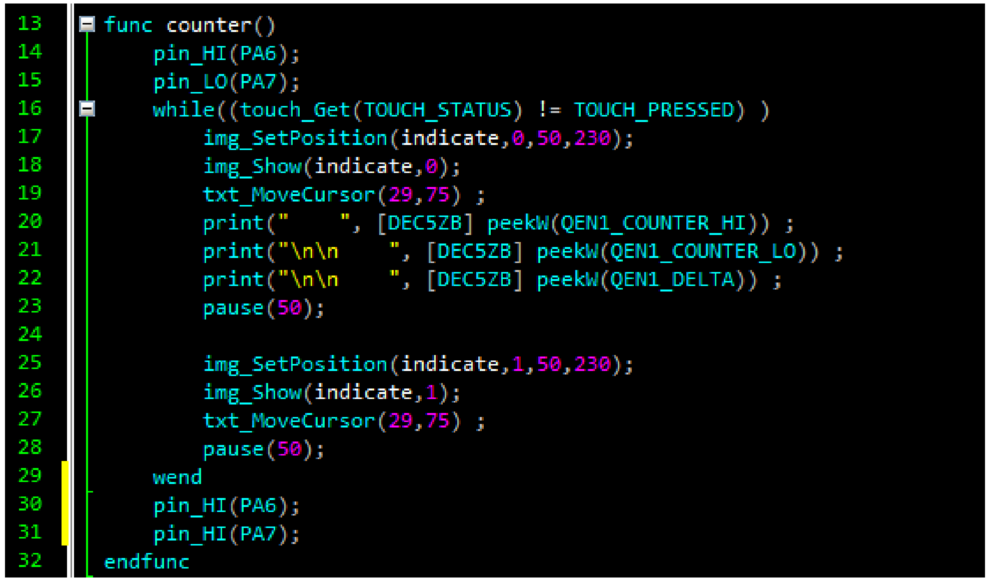
After the while condition loop is satisfied, the next statements in the sub-routine simply puts the signal generator to a stop.



```
    wend
    pin_HI(PA6);
    pin_HI(PA7);
endfunc
```

The counter() sub-routine

This sub-routine is almost identical to that of the previous sub-routine. The only difference between the two is the logic level for the GPIOs PA6 and PA7. This difference produces the switching of signals between the quadrature signal generators. This switching results to the reversal of the phase of the quadrature input.



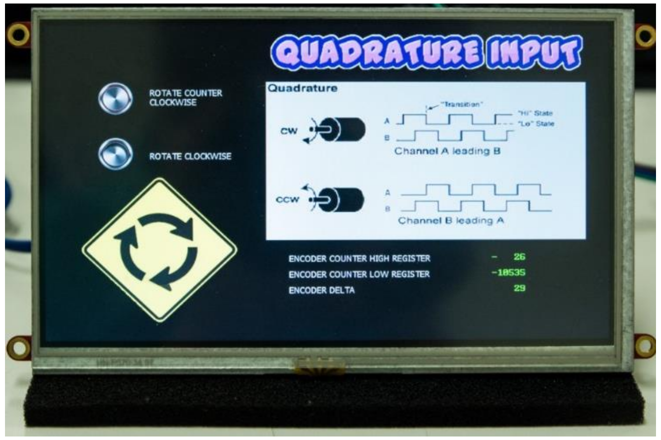
```
func counter()
    pin_HI(PA6);
    pin_LO(PA7);
    while((touch_Get(TOUCH_STATUS) != TOUCH_PRESSED) )
        img_SetPosition(indicate,0,50,230);
        img_Show(indicate,0);
        txt_MoveCursor(29,75) ;
        print("    ", [DEC5ZB] peekW(QEN1_COUNTER_HI)) ;
        print("\n\n    ", [DEC5ZB] peekW(QEN1_COUNTER_LO)) ;
        print("\n\n    ", [DEC5ZB] peekW(QEN1_DELTA)) ;
        pause(50);

        img_SetPosition(indicate,1,50,230);
        img_Show(indicate,1);
        txt_MoveCursor(29,75) ;
        pause(50);
    wend
    pin_HI(PA6);
    pin_HI(PA7);
endfunc
```

Running the project

Compile and download the program to the display module. Having been able to complete this step, the next step that needs to be done is to provide the wire up for the quadrature input generator and the Diablo16 70DT.

For the sole purpose of demonstration, a digital arbitrary waveform generator is used to provide the logic transitions. Furthermore, the quadrature encoder input used was produced using a set of D FLIP-FLOPS and XOR gates. The input represents a set of feedback and control signals that are commonly present with servo motors.

In this document, the servo motor feedback is mimicked using the quadrature generator and in addition, the rotation direction is controlled using two GPIO pins. In addition, a pulse generator was used to vary the clock pulse of the flip-flops. This pulse in turn represents the speed of the input. The pulse generator used in this application is an Agilent Arbitrary Waveform Generator set to a frequency of 10 Hz. Set with an output voltage of 0-5 volt dc peak voltage square wave output.
This application is fairly simple. A quadrature signal is fed to the DIABLO-70DT quadrature input channel. The display module then displays the read-out from the

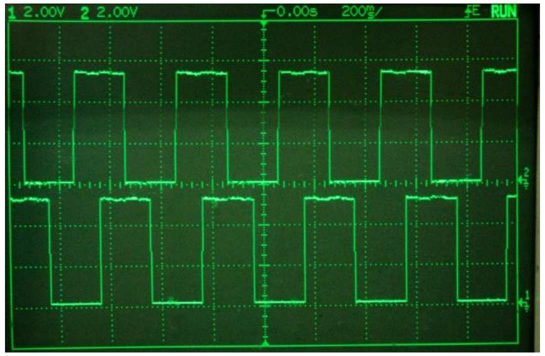


quadrature input registers. The registers that were displayed includes the encoder counter HIGH and LOW registers alongside the encoder DELTA register.

The quadrature channel of the Diablo16 -70DT detects the change in phase difference and identifies this a ‘going negative’ change or a “going positive” change. There are no pre-assigned sign convention for the results of the registers. Below are the view of the signals with an oscilloscope.

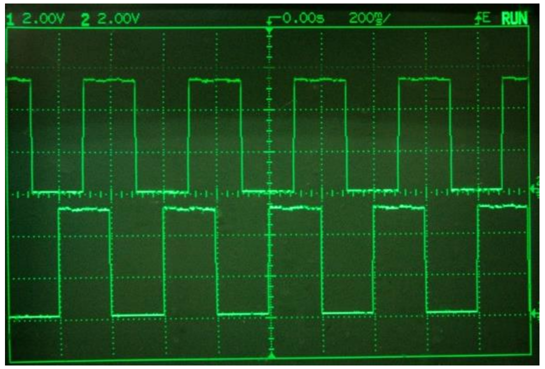
CLOCKWISE ROTATION QUADRATURE SIGNAL

The upper waveform is leading the lower waveform with 90 degrees.

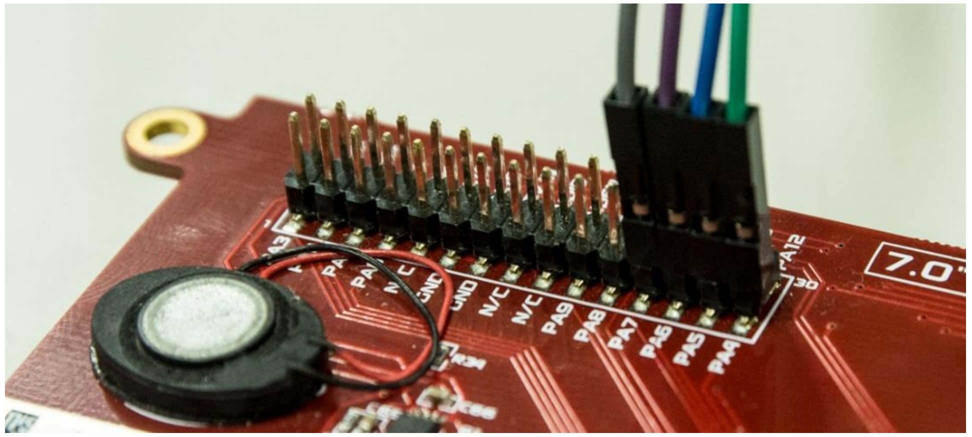


COUNTER-CLOCKWISE QUADRATURE SIGNAL

The lower waveform is leading the upper waveform with 90 degrees.



The quadrature signal is multiplexed using two general purpose input-output pins, namely: PA6 and PA7. These two pins enable or disable the signal generators. While on the other hand, PA4 and PA5 served as the input for the quadrature signals.



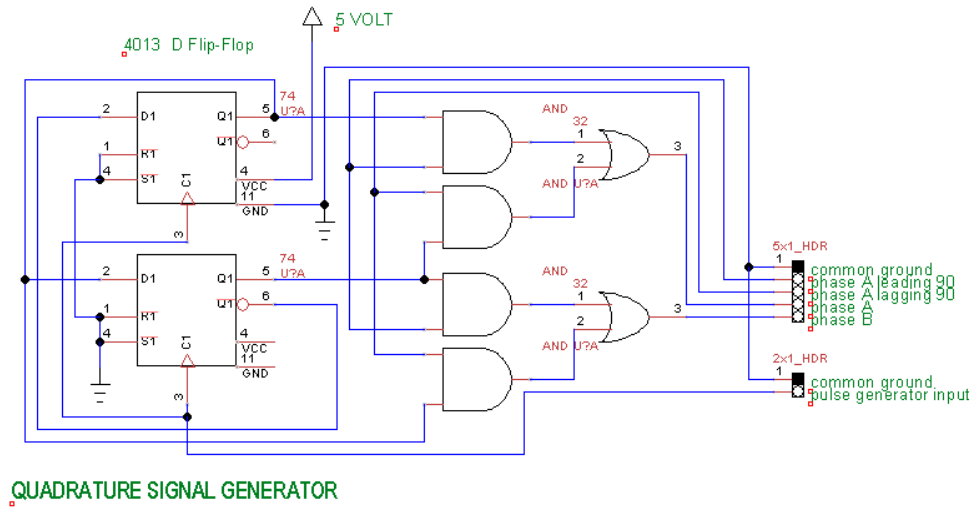
Pin number	purpose
PA4	Input for the PHASE A waveform
PA5	Input for the PHASE B waveform
PA6	Select pin for PHASE A leading with 90 degrees.
PA7	Select pin for PHASE A lagging with 90 degrees.

Note: If you’re using a gen4-Display, please consult the gen4-PA/4D-UPA Datasheet for the proper configuration of the GPIO pins.

The quadrature signal generator

D flip-flops are one of the simplest way to create a pair of signals with a 90 degree phase difference. It can be arranged in a manner wherein a single clock pulse can be used with two D flip-flops. Coupled to a few AND gates and OR gates, the phase select can be made to switch between two choices, that is – PHASE A leads with 90 degrees or PHASE A lags with 90 degrees.

Below is the schematic for the D Flip-flop based Quadrature Signal Generator.



The phase A lagging and phase B leading are used to switch the output. The quad in A and quad in B are to be connected with the PA4 and PA5, respectively.