Enhanced source hardware and tank for physical modeling

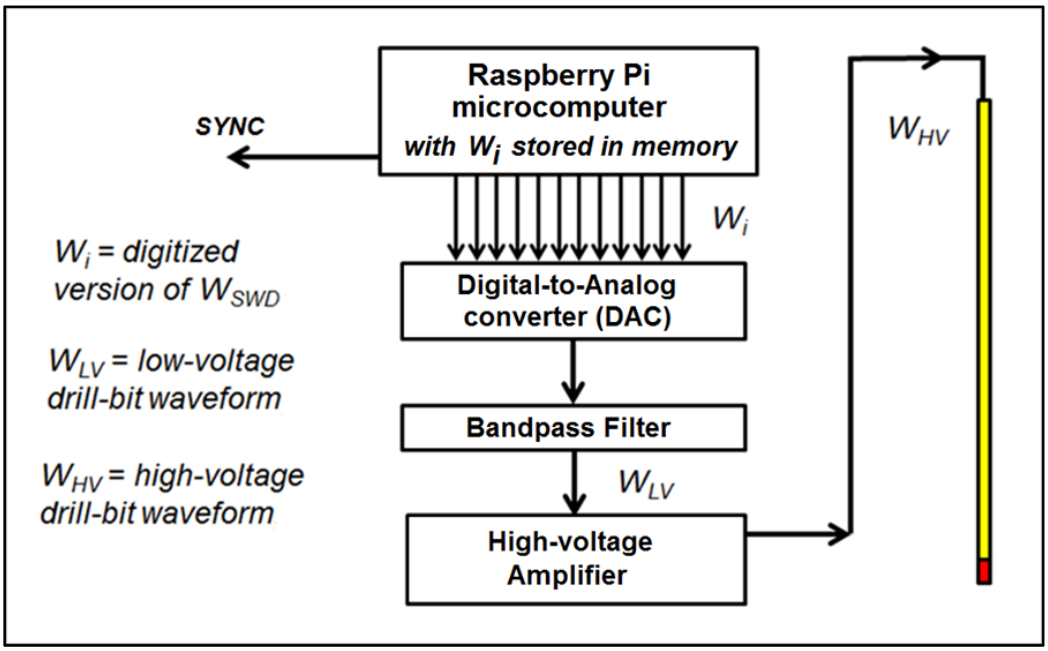
Joe Wong, Kevin Bertram, Hongliang Zhang, Kevin Hall, and Kris Innanen

ABSTRACT

Enhancements made to the University of Calgary Seismic Physical Modeling System include: (1) hardware and software additions that produce arbitrary source waveforms for driving piezoelectric transducers; (2) an improved, larger water tank to support simulations of marine seismic surveys; and (3) data acquisition modules that expand the number of receiver channels to 24. This report describes items (1) and (2) of these enhancements in detail. A companion report by Bertram and Wong (2020) discusses item (3).

1. PRODUCING ARBITRARY SOURCE WAVEFORMS

The ability to produce controlled source-waveforms increases the types of seismic physical-modeling experiments that are possible. Piezoelectric transducers driven by these controlled waveforms can be used in simulations of real-world seismic sources such as swept-frequency vibrators, impulsive sources with controlled delay times, and complex SWD signals produced by drill-bits interacting with subsurface rocks. For this purpose, we have designed and built a prototype Arbitrary Waveform Generator (AWG). The heart of the AWG is a Raspberry Pi 4B microcomputer driving an R2R resistor ladder for digital-to-analogue conversion.



1.1: Block diagram for the design of a high-voltage arbitrary waveform generator (AWG) for driving piezoelectric source transducers.

Figure 1.1 shows the structure of a prototype circuit capable of producing arbitrary high-voltage signal suitable for driving piezopin source transducers. A defined waveform is produced by a C or MATLAB program and saved in a file on a Raspberry Pi 4B microcomputer running the Raspian operating system. A compiled C program reads the file, converts the waveform into 9-bit integers, and transmits the bits of each integer to a digital-to-analogue converter (DAC) via the general purpose input/output (GPIO) pins of the RPi 4B microcomputer. The transmission rate is approximately 1 integer every 250 nanoseconds (limited by the system clock of the RPi 4B). The DAC is either an R2R ladder or a dedicated integrated circuit. The DAC output is filtered and amplified to yield analogue signals that are then amplified to yield peak-to-peak voltages of 100 to 200 volts. The high-voltage signals with bandwidth of approximately 50 kHz to 1.2 MHz can be used to drive piezoelectric source transducers. In elastic or acoustic materials, these will produce ultrasonic signals that scale down by a factor of 10,000 to frequencies of 5 to 120 Hz to match real-world seismic frequencies.

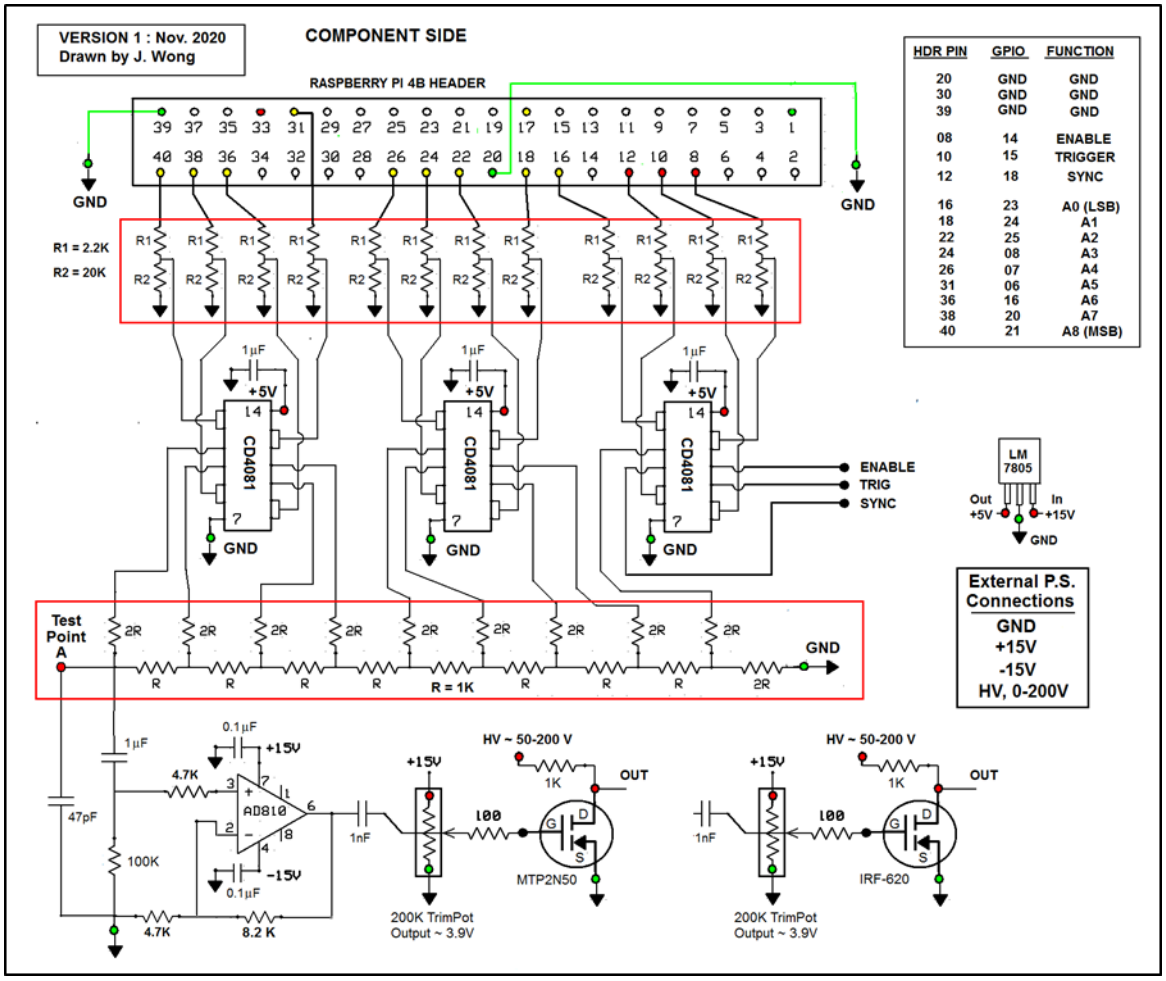


Fig. 1.2: Schematic diagram for an electronics circuit that converts 9-bit integers into an analogue signal. The 40-pin header connects to the GPIO pins of the RPi 4B microcomputer.

Figure 1.2 shows the circuit diagram for the prototype high-voltage AWG. Figure 1.3 is a photograph of prototype as of November 2020. Details on Figures 1.2 and 1.3 likely will be modified as the AWG design evolves from its present state. However, the overall design shown on Figure 1.1 will remain stable.

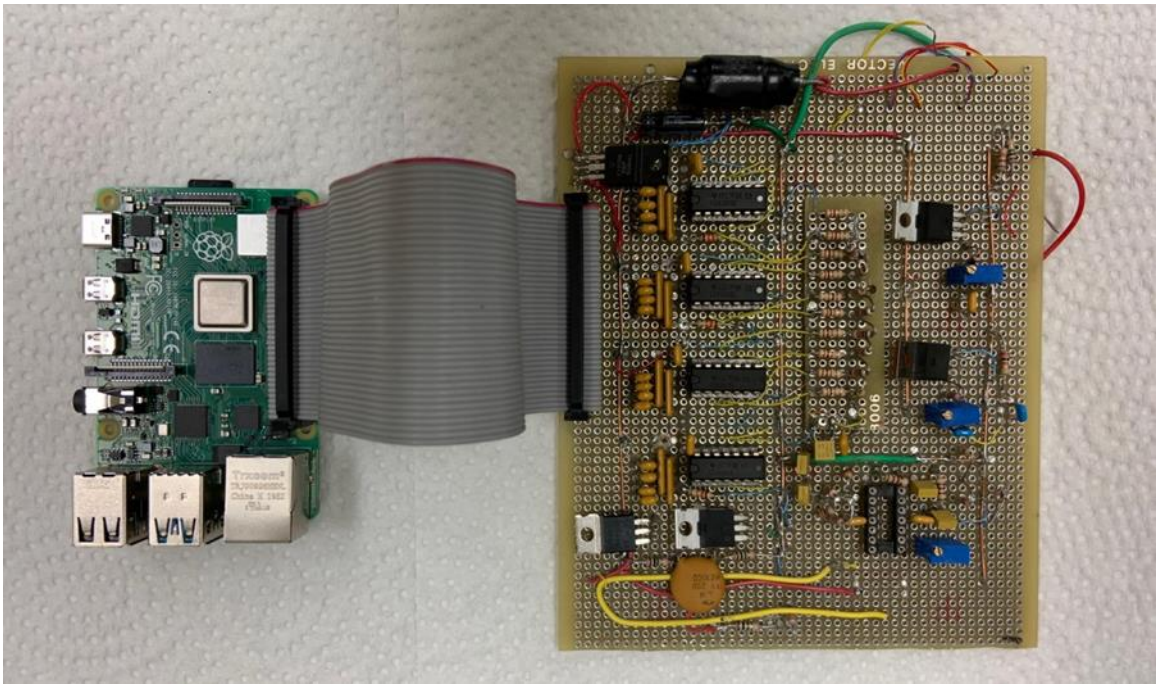


Fig. 1.3: Photograph of the prototype AWG, showing the RPi 4B microcomputer on the left connected to the DAC electronics on the right via 40-pin headers and a ribbon cable (Nov., 2020). Components below the middle blue trimpot on this photograph are for future expansion and have not been included on the schematic Figure 1.1.

The circuit design of the AWG uses 12 of the 16 GPIO pins available on a Raspberry Pi 4B microcomputer for user control. The GPIO pins are accessible to the user via a 40 pin header on the RPi 4B microcomputer. Figure 1.4 shows the assignment of header pin to GPIO functions (pins). C code sends values of 0 or 1 to set or reset the 12 GPIO pins. When a GPIO pin is set HI, it outputs 3.65V; when it is reset LOW, its output is 0V.

We raise the HI outputs at the GPIO pins from 3.65V to 5V by using CD4081 CMOS AND gates powered by a +5V supply. The paired inputs of each of four AND gates on a CD4081 are tied together, and connected to the GPIO pins through the 2.2 kΩ and 20 kΩ resistor combination as shown on Figure 1.2 (the resistor combinations are included for protecting the GPIO pins). The output from each AND gate is 0V or 5V, depending whether the associated GPIO pin value is LOW (0V) or HI (+3.65V).

Three of the gate outputs serve as the control signals SYNC, TRIG, and ENABLE. The remaining nine outputs are fed into a nine-bit R2R ladder to generate an analogue signal. The table on Figure 1.2 gives the bit-order assignment to the GPIO pins driving the R2R ladder. Table 1 lists nominal output analogue voltages measured at Test Point A on Figure 1.2. Nominally, f all nine GPIO pins are LOW (0V), the analog output voltage is 0 volts; if all nine GPIO pins are HI (3,65V), the analog output is +5V*(511/512).

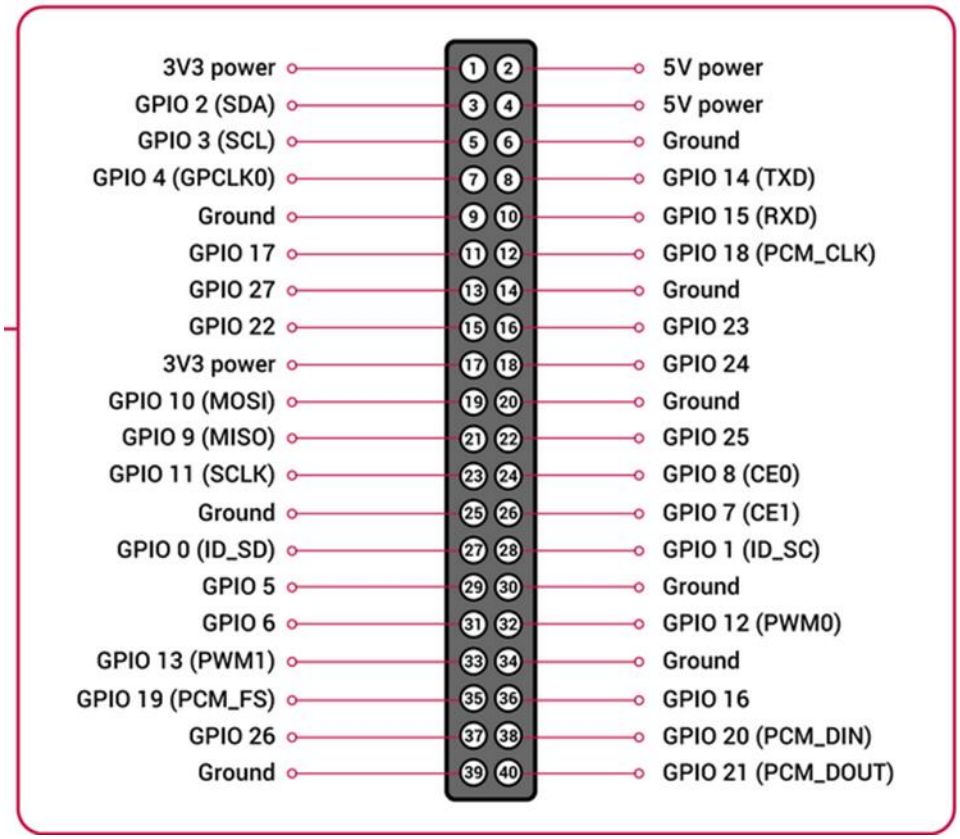


Fig. 1.4: Header pins on the Raspberry Pi 4B microcomputer and their associated GPIO functions (https://www.raspberrypi.org/documentation/usage/gpio/).

Table 1: Nominal output voltage at Test Point A on Figure 2 for gate supply voltage of +5V, when one of nine GPIO pins is set HI (3.6V) while the rest are set LOW (0V). When multiple GPIO pins are HI, the output voltage is the sum of the output voltages associated with those pins.

GPIO HI	Bit	Output (volts)
21	0	0.00976
20	1	0.01953
16	2	0.03907
12	3	0.07813
07	4	0/15625
08	5	0.31250
25	6	0.62500
24	7	1.25000
23	8	2.50000

Disabling system interrupts on the Raspberry Pi 4B

The Raspberry PI 4B m9crocomputer functions under the Raspian operating system. Compiled C code with timing loops have accurate timing disrupted by the system interrupts which monitor such essential tasks such as keyboard strokes, mouse operations, and printing requests. In order for the timing of sending digital data to the R2R ladder to be accurate, the system interrupts must be disabled before sending. After sending is completed, the interrupts must be enabled again so that requests from the input and output devices are will be recognized by the operating system. Disabling and enabling of system interrupts can be done using open source C functions found on the Raspberry.org website (see Appendix B).

Amplifying arbitrary source waveforms

In addition to the DAC circuit, the AWG require a high-voltage linear amplifier capable of raising low-level AC signals (about 100mV peak-to peak) up to 50V to 200V with bandwidths of about 50 kHz to 1.2 MHz. An appropriate design for the high-voltage amplifier is shown on Figure 1.2, and uses an integrated circuit operational amplifier (AD810) and power MOSFETS (MTP50N and/or IRF620).

In our design, an arbitrary waveform is defined by a sequence of 9-bit integers with values in the range 0 to 511. Each bit is assigned to a single GPIO pin on the Raspberry Pi. The GPIO pins are made accessible to the outside world via a 40-pin header connector. Figure 5 show the correspondence of the GPIO pins to the header pins, and Table 1 lists the correspondence of header pins to GPIO pins and the bits of the integers defining the arbitrary waveform.

Important limits for the Raspberry Pi 4B GPIO pins

The system clock of the RPi 4B is 14 GHz. This is fast enough to send out 9-bit words via the GPIO interface as fast as one word every 200 ns using current C code. The output voltage of each GPIO pin is 3.6 volts, but each pin must be limited to less than 13 milliamps, and if more than one GPIO pin is used as an output, the total current must be limited to less than 60 milliamps. If these limits are exceeded, the GPIO pins will be destroyed. We protect each GPIO pin by using the 2.2 kohm and 20 kohm resister combinations shown on Figure 1.2.

Initial testing of the AWG

Using the setup shown on the photograph of Figure 1.2, we produced analog signals by using compiled C software on the RPi 4B to read a file and produce an analogue signal defined by a list of integers on that file.

The times on all figures showing waveforms have been expressed using a real-world seismic exploration scale. This scale is 10000 times larger than the times measured in the Physical Modelling Laboratory. Thus, a time shown as 1 ms is actually equal to 100 ns (0.1 μs), and a time shown as 1 second is actually 100 μs.

Figure 1.5 shows the results of the first test. Figure 1.5(a) is a plot of the impulse response (measured in the Physical modeling Laboratory and saved on file) of a pair of piezopins (CA-1106) immersed in water and used as source and receiver. Figure 1(b) is the plot of the output waveform produced by the AWG as measured at Test Point A on Figure 1.5(b) when the impulse response data are fed through the AWG.

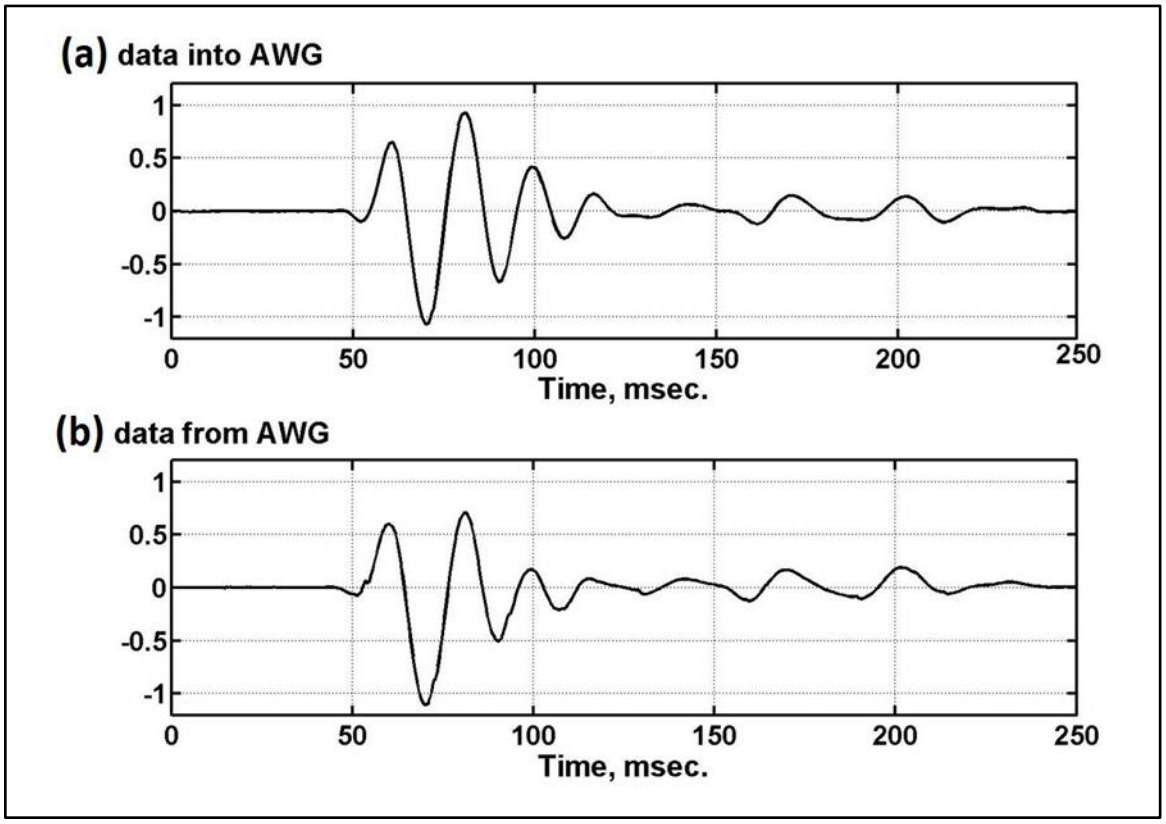


Fig. 1.5: Normalized plots. (a) Input data to AWG: piezopin impulse response, stored on file as digitized numeric data; (b) analogue output from the AWG taken at Test Point A on Figure 1.2, as digitized by a Gage 4424 ADC module.

The input data for Figure 1.5 are floating-point numbers separated by a sampling interval of 1 ms. To satisfy the speed and amplitude limitations of the RPi-based AWG, the input data are down-sampled to about 1.25 words per ms, and each word must be normalized, level-adjusted, and converted to 9-bit integers so that the minimum and maximum amplitudes fall in the range of 0 to 511.

Comparing Figures 1.5(a) and 1.5(b) indicates that, in this example, the AWG results in an output signal that preserves both amplitude and timing quite well. In this example, the distortion between the input and output waveforms are relatively minor, so that the analogue output can be used to drive a source piezoelectric transducers with little or no concern.

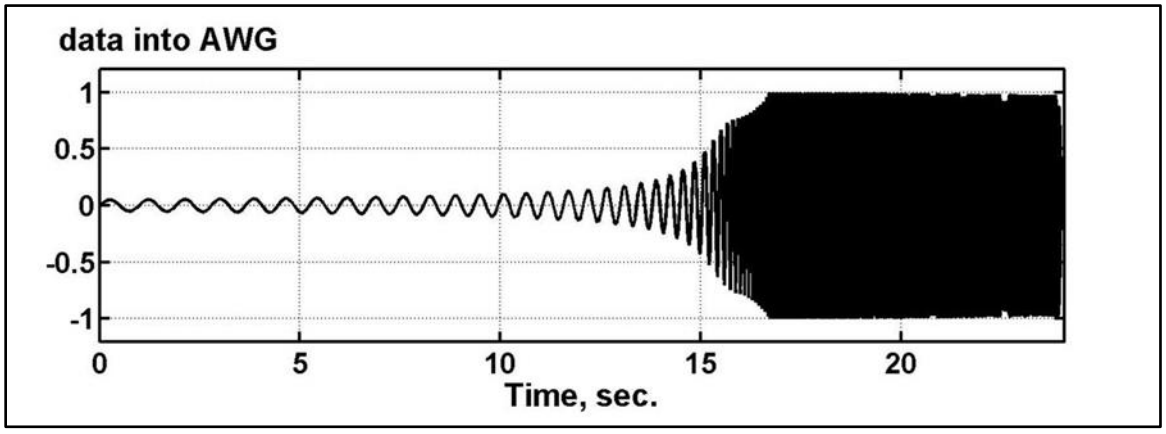


Fig. 1.6: Scaled amplitude plot of a low-dwell Vibroseis sweep. The sweep is slightly over 24 seconds long, and was used in a field experiment in Alberta to drive a heavy land vibrator. Start frequency is 1 Hz; end frequency is 100 Hz. The sweep is cosine tapered with tapers lengths of 0.2 seconds at the start and end. The sampling interval is 2 ms,

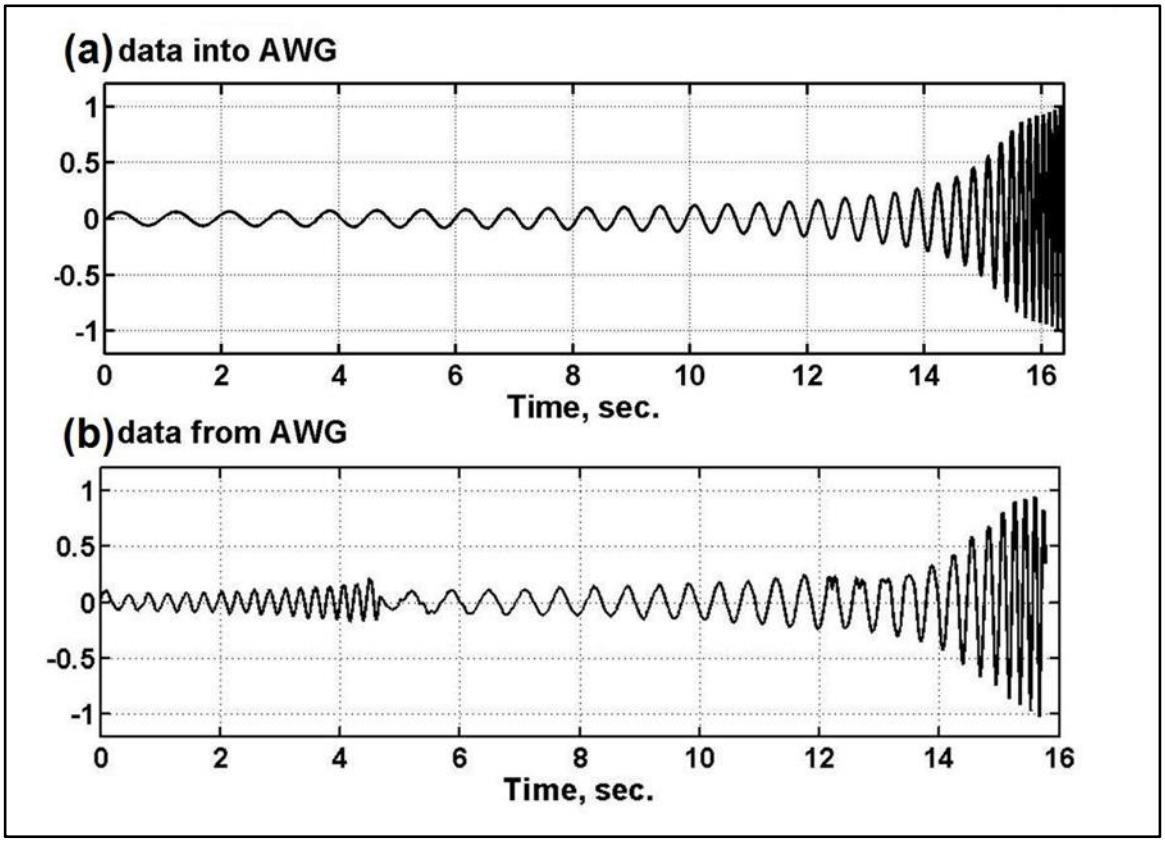


Fig. 1.7: Scaled amplitude plots. (a) Input data to AWG: 16 seconds of the low-dwell frequency sweep, stored on file as digitized numeric data; (b) analogue output from the AWG taken at Test Point A on Figure 1.2, as digitized by a Gage 4424 ADC module. The sampling time for Gage output shown on Figure 1.7(b) was 2 ms, but the time axis has been adjusted to match that of Figure 1.7(a).

Figures 1.6 and 1.7 show the results of the second test. Figure 1.6 is a plot of a low-dwell Vibroseis sweep used to drive a Failing low-frequency vibrator in a field experiment near Hussar, Alberta (Isaac and Margrave, 2011).

Figure 1.7(a) shows the first 16 seconds of the low-dwell sweep. These data were used as input to the AWG. Figure 17(b) is the plot of the output waveform produced by the AWG as measured at Test Point A on Figure 1.2. The data of Figure 1.7(a) are floating-point numbers separated by a sampling interval of 2 ms. To satisfy the speed and amplitude limitations of the RPi-based AWG, the input data are down-sampled to about 1.25 words per ms, and each word must be normalized, level-adjusted, and converted to 9-bit integers so that the minimum and maximum amplitudes fall in the range of 0 to 511.

Figure 1.7 indicates that, for this example with a more complex waveform, the DAC gives results that don not preserve both amplitude and timing as well as for the previous example. In the previous example, the distortion between the input and output waveforms were relatively minor. But in this example, we see severe disruptions in in both timing and amplitudes at certain portions of the waveform. These disruptions are due to issues related to bit-transition times as specific amplitudes in the input data change from one value to another. It is also likely that aliasing causes some trouble when the sampling time set for the Gage A/D module somehow is not in synchronization with the bit-transition times.

The severe distortions in the analogue output waveform from this example means that it cannot be used to in a physically-modeled experiment to accurately represent a real-world Vibroseis survey. A priority in the continue development of the AWG will be to eliminate these distortions or reduce them to an acceptable level.

2. ACRYLIC WATER TANK

We design and built an acrylic (Plexiglas) water tank to replace the wood used from 2007 to 2019. The outside dimensions of the new tank are 20”H by 40”W by 48”L. Detailed drawings are presented in Appendix A. The new tank is more durable, and is larger so that wall reflections are diminished when the tank is used for simulating marine seismic surveys.

Fig. 2.1: Acrylic water tank. Note the black square steel tubing along the right side of the tank. A similar tubing runs along the opposite side. The two square tubings are held together by threaded steel rods at the front and back ends (the front rod is visible). The two square tubings are clamped tight against the tank sides with nuts and washers on the ends of the rods to prevent excessive bowing of the tank sides under water pressure.

In general terms, the acrylic tank was built in the following manner. Side and bottom panels were cut to size and assembled using stainless steel screws, them loosened slightly so acrylic glue could be applied to all edges for mechanical reinforcement. The screws were tightened again, and after the glue set, silicone sealant was applied to all inside corners and edges to prevent water leakage. Repeated filling of the tank to full capacity and left unattended for weeks revealed only small leaks at the bottom seams. We have fixed these leaks by resealing with acrylic cement and silicone sealant. We clamped rectangular steel tubing to the exterior of the tank to prevent bowing of the sides when it is filled with water (see Figure 2.1).

DISCUSSION AND CONCLUSION

A prototype Arbitrary Waveform Generator has been constructed and tested. Initial results are reasonably good. At present, the AWG is able to convert integers to an analogue signal at an approximate rate of one 9-bit word every 250 ns. Also, unwanted glitches are present in the output analogue signal when the input digital words assume some specific values. We will investigate software and hardware modifications in order (1) to increase the rate and its precision, and (2) to eliminate or mitigate the amplitude glitches.

A new, bigger water tank has been designed and built. The increased dimensions of the tank should lessen the presence and amplitudes of tank-boundary reflections observed in physically-modeled marine seismic surveys.

ACKNOWLEDGEMENTS

We thank the sponsors of CREWES for continued support. This work was funded by CREWES industrial sponsors, NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 461179-13, and in part by the Canada First Research Excellence Fund.

REFERENCES

Bertram, K. and Wong, J., Enhanced receiver hardware for physical modelling: *CREWES Research Report*, this volume.

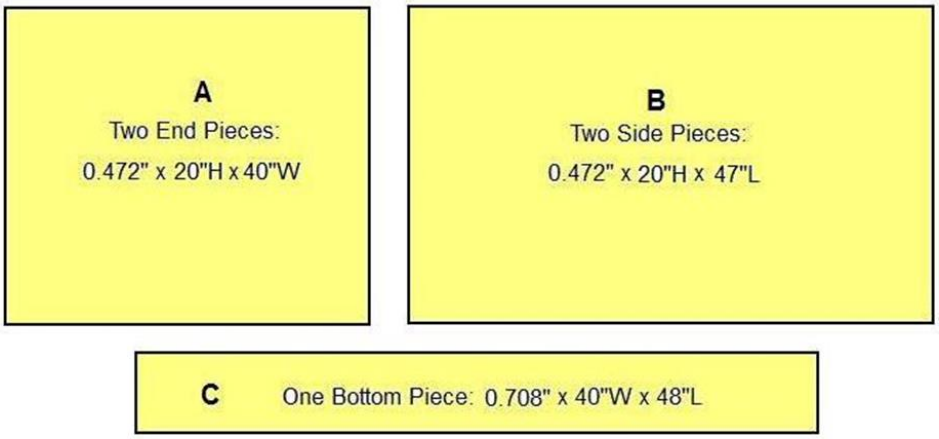
Isaac, J. H., and Margrave, G. F., 2011, Hurrah for Hussar! Comparisons of stacked data: *CREWES Research Report*, **23**, 55.1–55.23.

Wong, J., 2013, Multiple simultaneous vibrators controlled by m-sequences: 83rd Annual International Meeting, SEG, Expanded Abstracts, 109-113.

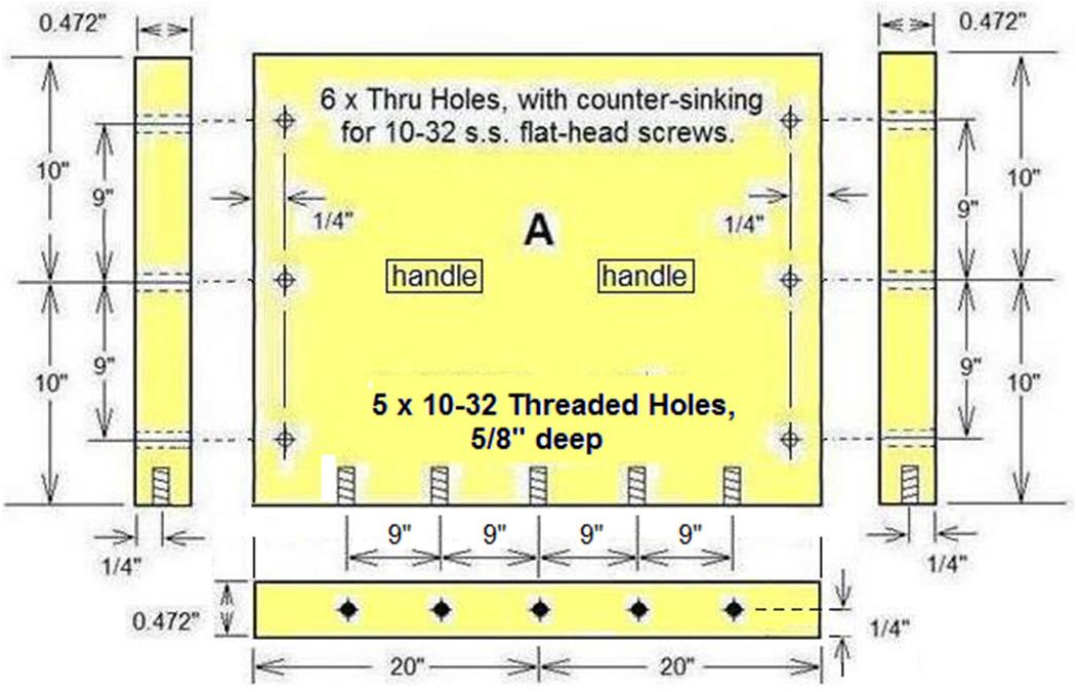
Wong, J., 2012, Simultaneous multi-source acquisition using m-sequences: *CREWES Research Report*, **25**, 81.1-81.16.

(https://www.raspberrypi.org/documentation/usage/gpio/).

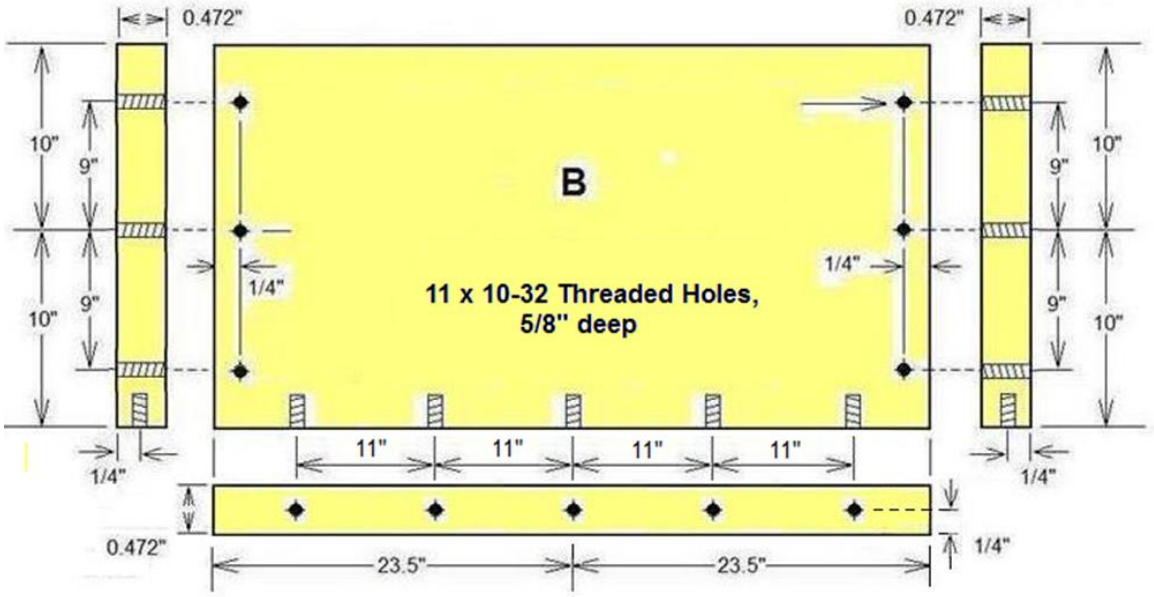
APPENDIX A: BUILDING AN ACRYLIC WATER TANK



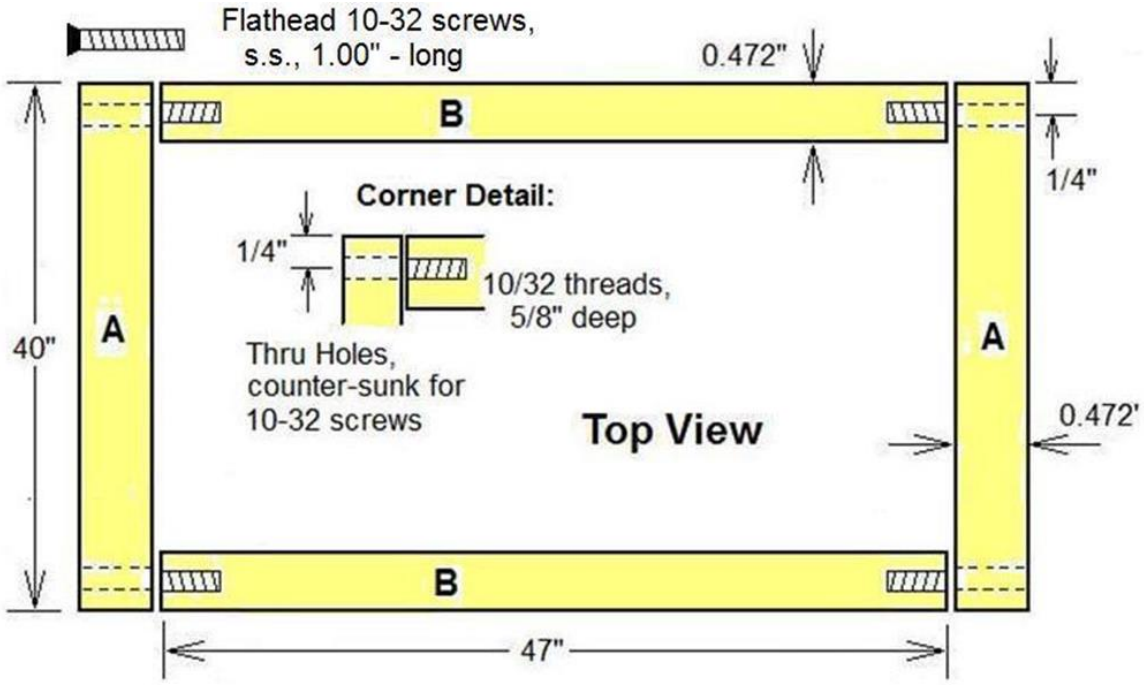
STEP 1**:** Cut five pieces from stock acrylic material. Save extra material.



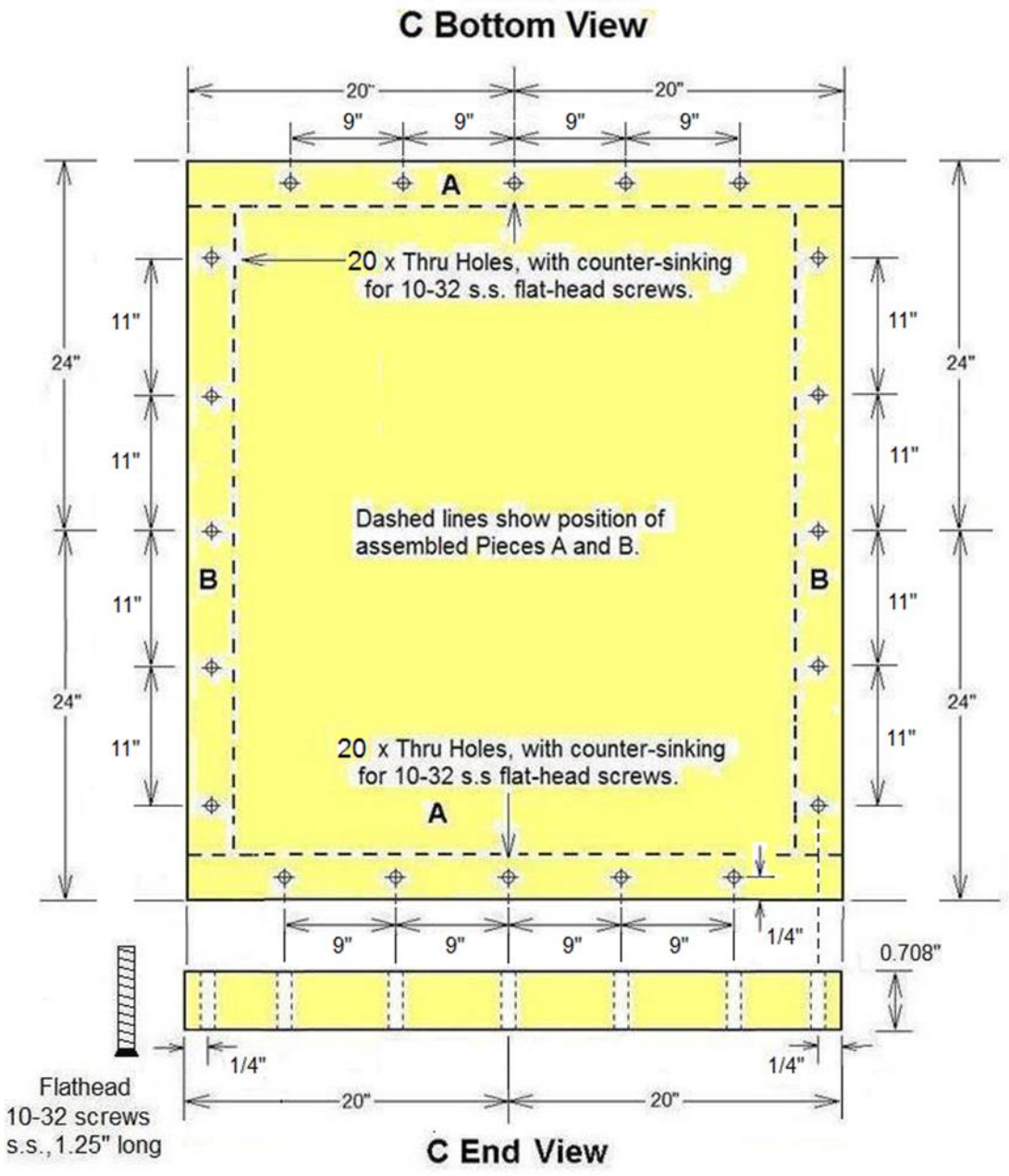
STEP 2: Drill 10-32 threaded holes and through holes on two pieces of Piece A.



STEP 3: Drill 10-32 threaded holes on two pieces of Piece B.



Step 4: Assemble Pieces A and B to form sides of tank using 1.00”-long 10-32 stainless steel flat-head screws



Step 5: (a) Drill through holes for 10-32 screws on Piece C (bottom of tank). (b) Attached assembled sides from Step 4 to Piece C with 10-32 screws. (c) Loosen screws and apply acrylic cement to all edges to be joined. (d) When cement has set, apply silicone sealant to all inside edges and corners to make the tank water tight.

APPENDIX B

C code for disabling system interrupts on Raspberry Pi 4b microcomputer

The following listing contains C code functions for disabling and interrupts on the Raspberry Pi 4B computer. These function are used in application C codes for controlling the DAC circuitry on Figures 1.2 and 1.3.

```
/* ---------------------------------------------------------------
 Open source code by petzval, Jan.23, 2019, for RPi
https://www.raspberrypi.org/forums/viewtopic.php?f=29&t=52393&p=1420461#p1420461
---------------------------------------------------------------
/*

/*************************************************
timtest.c

This C program illustrates accurate timing on a
Raspberry Pi. Works on ARMv6 and ARMv7 (Pi3) systems.
It disables interrupts then runs a test to determine how
many times interrupts disrupt the timer functions. The
count should be zero.
It then enables interrupts and runs the test again,
and the result should be a large error count.

Compiled from console with gcc
gcc timtest.c -o timtest

To use the functions in your own code:
1. Call setup() once
2. Call interrupt(0) to disable interrupts
3. Use the timer and gpio code as documented below.
   While interrupts are disabled, the keyboard and
   screen will not work - so no printf calls
4. Call interrupt(1) to re-enable interrupts

Interrupts can be disabled/re-enabled any number of
times. Interrupts can be left disabled for long
periods if necessary.

The function hardwaretest() is included,
but is not called by this main() code.
Puts a 50kHz signal on GPIO pin 2
-------------------------------------------------------------------------------
*/
#include <stdio.h>
```

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int setup(void);
int interrupts(int flag);
void hardwaretest(void);
int measureints(void);

// ARMv6
#define ADDR_BASE6  0x20000000
// ARMv7
#define ADDR_BASE7  0x3F000000

#define GPIO_BASE 0x00200000
#define TIMER_BASE 0x00003000
#define INT_BASE 0x0000B000

volatile unsigned *gpio,*gpset,*gpclr,*gpin,*timer,*intrupt,*intquad;
enum pitypes {NOTSET,ARM6,ARM7};
enum pitypes pitype = NOTSET;  // set by setup()  0=not set 1=ARMv6  2=ARMv7

unsigned int timend;
#define TIMERLOOP  while((((*timer)-timend) & 0x80000000) != 0)
#define TIMEOUT          (((*timer-timend) & 0x80000000) == 0)
#define NOTIMEOUT        (((*timer-timend) & 0x80000000) != 0)


/********** TIMER CODE example *******

int msdelay, getout;

setup()               // initialise system
                    // call only once

interrupts(0);          // disable interrupts

timend = *timer + 200;   // Set up 200 microsecond delay
                    // Maximum possible delay
                    // is 7FFFFFF or about 35 minutes
TIMERLOOP;              // 200 us delay


msdelay = 23;
timend = *timer + msdelay*1000;  // set up 23ms delay
TIMERLOOP;                    // 23ms delay
```

```
      // time out examples

timend = *timer + 5000;   // set up 5ms time out
do
  {
  }
while(NOTIMEOUT);         // exits loop after 5ms


timend = *timer + 45000;  // set up 45ms time out
getout = 0;
do
  {
  if(TIMEOUT)
    getout = 1;    // exit loop after 45ms
  }
while(getout == 0);

interrupts(1);       // re-enable interrupts

********** END TIMER EXAMPLES ******************************/
*/

//********************* GPIO read/write ********************/

#define GP2_HI *gpset = (1 << 2)  // GPIO 2 output hi
#define GP2_LO *gpclr = (1 << 2)  // GPIO 2 output lo

#define GP3_IN (*gpin & (1 << 3))   // GPIO 3 input - zero or non-zero, not 1

#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7 << (((g)%10)*3))
#define OUT_GPIO(g) *(gpio+((g)/10)) |=  (1 << (((g)%10)*3))

/********** GPIO code examples ****************

set GPIO 3 as input
  INP_GPIO(3);

read GPIO 3    GP3_IN is zero or non-zero (do not test for == 1)
  if(GP3_IN == 0)    // test lo
  if(GP3_IN != 0)    // test hi

set GPIO 2 as output
  INP_GPIO(2);   // must do this first to zero control bits
  OUT_GPIO(2);   // set as output
```

```
set GPIO 2 level
  GP2_HI;    // GPIO 2 hi
  GP2_LO;    // GPIO 2 lo

******************** END GPIO ****************/

int main()
  {
  int n;

  printf("Setting up...\n");

  sleep(3);    // 3 second delay
               // When the program starts, the interrupt
               // system may still be dealing with the
               // last Enter keystroke. This gives it
               // time to finish.

  if(setup() == 0)      // setup GPIO, timer and interrupt pointers
    return(0);

  printf("Testing interrupts - will take 1 minute\n");
  printf("Disabling interrupts and testing...\n");

  sleep(2);

  if(interrupts(0) == 0)
    return(0);

  n = measureints();

  interrupts(1);

  printf("Count of interrupts that disrupt TIMERLOOP = %d\n",n);
  printf("Enabling interrupts and testing again...\n");

  n = measureints();

  printf("Count of interrupts that disrupt TIMERLOOP = %d\n",n);

  return(0);
  }


/********** MEASURE INTERRUPT ERRORS *******
```

Runs for 30 seconds.

```
Returns the number of times an interrupt disrupts
the 10 microsecond delay causing a timing error

**************************/

int measureints()
  {
  int n,count;

  n = 0;
  count = 0;
  do
    {
    timend = *timer + 10;  // 10 us delay
    TIMERLOOP;
    if( (*timer - timend) != 0 )
      ++count;            // missed 10us timend

    ++n;
    }
  while(n < 3e6);   // 30 seconds

  return(count);
  }

/************ HARDWARE TEST **************
sets up GPIO 2 as output
disables interrupts
puts a 50kHz signal on GPIO 2 for 20 seconds
enables interrupts
*****************************************/

void hardwaretest()
  {
  int n;

                // set GPIO 2 as output
  INP_GPIO(2);   // must do this first to zero control bits
  OUT_GPIO(2);   // set as output

  if(interrupts(0) == 0)  // disable interrupts
    return;

  timend = *timer;   // current timer value
  for(n = 0 ;  n < 1000000  ; ++n)
    {
    timend += 10;    // 10us
```

```
   TIMERLOOP;

   GP2_HI;         // output GPIO 2 hi

   timend += 10;    // 10us
   TIMERLOOP;

   GP2_LO;         // output GPIO 2 lo
   }

  interrupts(1);    // re-enable interrupts
  }


/********************* INTERRUPTS **************

interrupts(0)   disable interrupts
interrupts(1)   re-enable interrupts

return 1 = OK
      0 = error with message print

Uses pitype and intrupt/intquad pointers set by setup()

Avoid calling immediately after keyboard input
or key strokes will not be dealt with properly

*****************************************/

int interrupts(int flag)
  {
  int n;
  unsigned int temp131;
  static unsigned int sav132 = 0;
  static unsigned int sav133 = 0;
  static unsigned int sav134 = 0;
  static unsigned int sav4 = 0;
  static unsigned int sav16 = 0;
  static unsigned int sav17 = 0;
  static unsigned int sav18 = 0;
  static unsigned int sav19 = 0;
  static unsigned int sav20 = 0;
  static unsigned int sav21 = 0;
  static unsigned int sav22 = 0;
  static unsigned int sav23 = 0;
  static int disflag = 0;
```

```
if(pitype == NOTSET)
  {
  printf("Setup not done\n");
  return(0);
  }

if(flag == 0)    // disable
  {
  if(disflag != 0)
    {
    // Interrupts already disabled so avoid printf
    return(0);
    }

        // Note on register offsets
        // If a register is described in the documentation as offset 0x20C = 524 bytes
        // The pointers such as intrupt are 4 bytes long,
        // so intrupt+131 points to intrupt + 4x131 = offset 0x20C

        // save current interrupt settings

  if(pitype == ARM7)   // Pi3 only
    {
    sav4 = *(intquad+4);     // Performance Monitor Interrupts set  register 0x0010
    sav16 = *(intquad+16);   // Core0 timers Interrupt control  register 0x0040
    sav17 = *(intquad+17);   // Core1 timers Interrupt control  register 0x0044
    sav18 = *(intquad+18);   // Core2 timers Interrupt control  register 0x0048
    sav19 = *(intquad+19);   // Core3 timers Interrupt control  register 0x004C
          // the Mailbox interrupts are probably disabled anyway - but to be safe:
    sav20 = *(intquad+20);   // Core0 Mailbox Interrupt control  register 0x0050
    sav21 = *(intquad+21);   // Core1 Mailbox Interrupt control  register 0x0054
    sav22 = *(intquad+22);   // Core2 Mailbox Interrupt control  register 0x0058
    sav23 = *(intquad+23);   // Core3 Mailbox Interrupt control  register 0x005C
    }

  sav134 = *(intrupt+134);  // Enable basic IRQs register 0x218
  sav132 = *(intrupt+132);  // Enable IRQs 1 register 0x210
  sav133 = *(intrupt+133);  // Enable IRQs 2 register 0x214

        // Wait for pending interrupts to clear
        // Seems to work OK without this, but it does no harm
        // Limit to 100 tries to avoid infinite loop
  n = 0;
  while( (*(intrupt+128) | *(intrupt+129) | *(intrupt+130)) != 0 && n < 100)
    ++n;
```

```
      // disable all interrupts

   if(pitype == ARM7)   // Pi3 only
     {
     *(intquad+5) = sav4;   // disable via Performance Monitor Interrupts clear  register
0x0014
     *(intquad+16) = 0;     // disable by direct write
     *(intquad+17) = 0;
     *(intquad+18) = 0;
     *(intquad+19) = 0;
     *(intquad+20) = 0;
     *(intquad+21) = 0;
     *(intquad+22) = 0;
     *(intquad+23) = 0;
     }


   temp131 = *(intrupt+131);  // read FIQ control register 0x20C
   temp131 &= ~(1 << 7);      // zero FIQ enable bit 7
   *(intrupt+131) = temp131;  // write back to register
                       // attempting to clear bit 7 of *(intrupt+131) directly
                       // will crash the system

   *(intrupt+135) = sav132;  // disable by writing to Disable IRQs 1 register 0x21C
   *(intrupt+136) = sav133;  // disable by writing to Disable IRQs 2 register 0x220
   *(intrupt+137) = sav134;  // disable by writing to Disable basic IRQs register 0x224

   disflag = 1;   // interrupts disabled
   }
  else           // flag = 1 enable
   {
   if(disflag == 0)
     {
     printf("Interrupts not disabled\n");
     return(0);
     }

          // restore all saved interrupts

   *(intrupt+134) = sav134;
   *(intrupt+133) = sav133;
   *(intrupt+132) = sav132;

   temp131 = *(intrupt+131);     // read FIQ control register 0x20C
   temp131 |= (1 << 7);          // set FIQ enable bit
   *(intrupt+131) = temp131;     // write back to register
```

```
   if(pitype == ARM7)  // Pi3 only
     {
     *(intquad+4) = sav4;
     *(intquad+16) = sav16;
     *(intquad+17) = sav17;
     *(intquad+18) = sav18;
     *(intquad+19) = sav19;
     *(intquad+20) = sav20;
     *(intquad+21) = sav21;
     *(intquad+22) = sav22;
     *(intquad+23) = sav23;
     }

   disflag = 0;         // indicates interrupts enabled
   }
  return(1);
  }

/****************** SETUP ****************
Determines Pi type ARMv6 or ARMv7:
  pitype
Sets timer, gpio and interrupt pointers:
  timer,gpio,gpset,gpclr,gpin,intrupt,intquad
Does not disable interrupts
return 1 = OK
      0 = error with message print
*************************************/

int setup()
  {
  int memfd;
  void *gpio_map,*timer_map,*int_map,*quad_map;
  unsigned int baseadd;
  FILE *stream;
  static char arms[6] = {"ARMv"};
  int n,c,getout;

      // read file /proc/cpuinfo to determine PI type
      // look for "ARMv6" or "ARMv7"

  pitype = NOTSET;  // in case fails

  stream = fopen("/proc/cpuinfo","rb");
  if(stream == NULL)
    {
    printf("Failed to open /proc/cpuinfo\n");
    return(0);
```

```
  }

getout = 0;
n = 0;  // compare string arms[n] offset
do
  {
  c=getc(stream);
  if(c == '6' && n == 4)
    {
    pitype = ARM6;      // old Pis ARMv6
    baseadd = ADDR_BASE6;
    getout = 1;
    printf("Pi type = ARMv6\n");
    }
  else if(c == '7' && n == 4)
    {
    pitype = ARM7;      // Pi 3 ARMv7
    baseadd = ADDR_BASE7;
    getout = 1;
    printf("Pi type = ARMv7\n");
    }
  else if(c == EOF)
    {
    printf("Failed to determine Pi type from cpuinfo\n");
    fclose(stream);
    return(0);
    }
  else
    {
    if(c == arms[n])
      ++n;
    else
      n = 0;
    }
  }
while(getout == 0);

fclose(stream);

memfd = open("/dev/mem",O_RDWR|O_SYNC);
if(memfd < 0)
  {
  printf("Mem open error\n");
  pitype = NOTSET;
  return(0);
  }
```

```
gpio_map = mmap(NULL,4096,PROT_READ|PROT_WRITE,
            MAP_SHARED,memfd,baseadd+GPIO_BASE);

timer_map = mmap(NULL,4096,PROT_READ|PROT_WRITE,
            MAP_SHARED,memfd,baseadd+TIMER_BASE);

int_map = mmap(NULL,4096,PROT_READ|PROT_WRITE,
            MAP_SHARED,memfd,baseadd+INT_BASE);

if(pitype == ARM7)  // Pi3
  quad_map = mmap(NULL,4096,PROT_READ|PROT_WRITE,
            MAP_SHARED,memfd,0x40000000);
else
  quad_map = MAP_FAILED;

close(memfd);

if(gpio_map == MAP_FAILED ||
   timer_map == MAP_FAILED ||
   int_map == MAP_FAILED ||
   (pitype == ARM7 && quad_map == MAP_FAILED) )
  {
  printf("Map failed\n");
  pitype = NOTSET;
  return(0);
  }

if(pitype == ARM7)
  intquad = (volatile unsigned *)quad_map;

          // interrupt pointer
intrupt = (volatile unsigned *)int_map;
          // timer pointer
timer = (volatile unsigned *)timer_map;
++timer;    // timer lo 4 bytes
          // timer hi 4 bytes available via *(timer+1)

          // GPIO pointers
gpio = (volatile unsigned *)gpio_map;
gpset = gpio + 7;     // set bit register offset 28
gpclr = gpio + 10;    // clr bit register
gpin = gpio + 13;     // read all bits register
return(1);
}
```