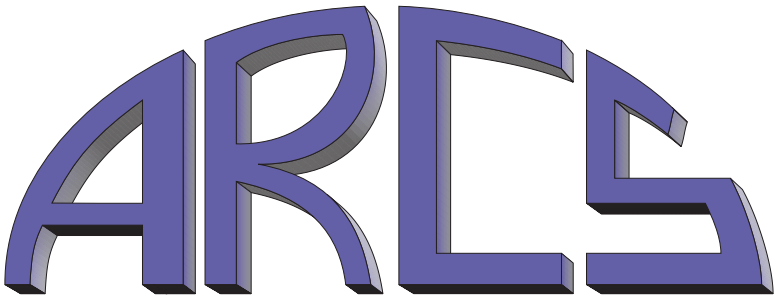

Advanced Realtime Control Systems, Inc.

ARCS LIGHTNING PROGRAMMER'S REFERENCE, VERSION 1.0

3 February 2000
Stephentown, New York

Contents

1 Getting Started

In this section, two simple examples are presented to introduce programming of the DSP system. The next section addresses many of the required elements for implementing a real-time system. Subsequent sections explain the basic structure of the non real-time system which supervises the real-time system, user interface development, extensions to the supervision component and to the communication structure between the supervision and DSP system, and remote processes.

1.1 First Program - *counting*

This first program is analogous to "Hello_World" for embedded systems programming. The program simply increments an integer counter and its listing is provided below.

```
/* counting.c

  Author: Brian R. Tibbetts, A.R.C.S., Inc.  22 July 1999

  This program is an embedded system answer to hello_world.c.  */

#include <hdw_util.h>
#include <host_comm_01x.h>

int cnt = 0;

main()
{
   hdwInit (0, 0, 0, 0, 0, 0, 0);

   while(1)
   {
      communications();
      cnt++;
   }
}
```

Here are some of the important details:

1. *ARCS library*. The include statements provide the minimum elements from the ARCS library which need to be included in an application program. The first header file, *hdw_util.h*, declares all functions needed to initialize the hardware and software plus provides the interrupt support. The second header file, *host_comm_01x.h*, declares all functions required for communications with the host.

2. *Exposing variables*. Any variables which are to be accessible or exposed to the host must be declared in *file scope*. In this example, *cnt* is declared file scope so that the value of *cnt* can be examined from *AIDE* .

3. *Initialization*. Within *main ()*, the first step is to initialize the hardware and software system. This is accomplished with a call to *hdwInit ()*. This function sets the number of each type of I/O device and the number of interrupts. This function also handles the initialization of the host communications subsystem.

4. *Infinite loop*. Typically, embedded systems programs run within a infinite loop. In this program, it is provided by the statement:

   ```
   while () { ... }
   ```

5. *Communications with the host*. Within the function call, *communications ()*, the DSP software system processes any communications requests from the host computer.

It is strongly recommended that you build and run this program before continuing. All of the required files can be found in the *dsp* subdirectory under the root installation directory. The source file is in the *samples* subdirectory. Both header files can be found in the *inc* subdirectory. The library file (for example, *ac30i01a_v1_16.lib*) and the link command file, *applink.cmd*, are both found in the main dsp directory.

The “AIDE User’s Manual” completely documents building and running an application program. Therefore, the typical convention in this manual will

be to assume familiarity with *AIDE* and to not explicitly address use of the interface. However, a typical screen shot for a success first program is shown in Fig. 1.

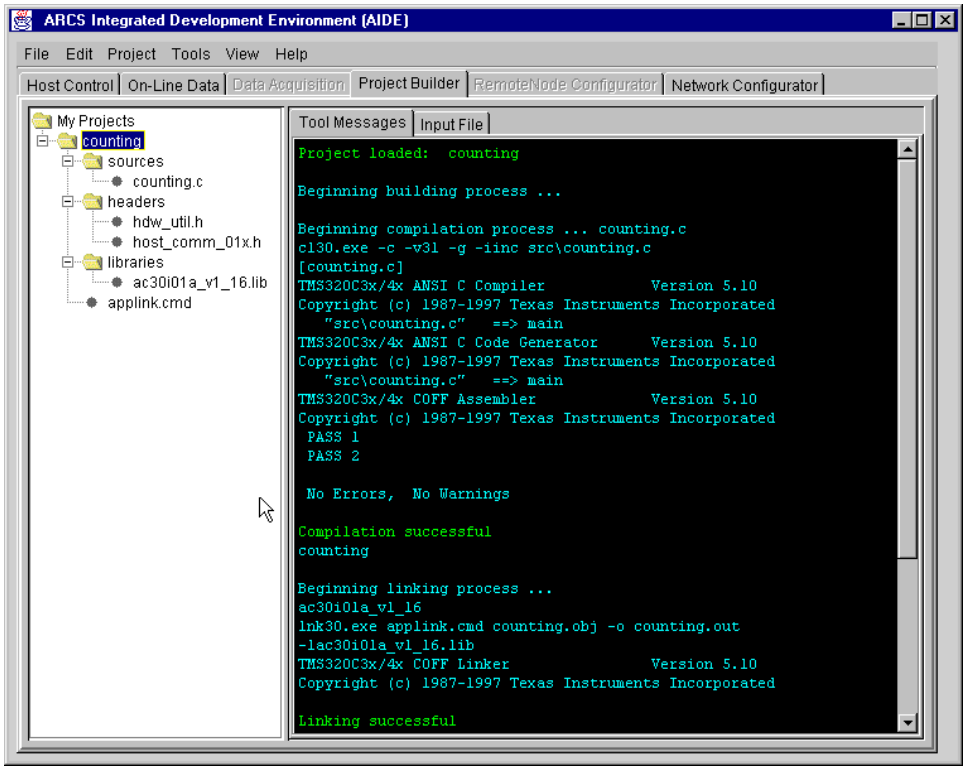

FIGURE 1: Successful First Program Build

After successfully building this program, it should be run and the value of *cnt* should be checked. After starting the program it is wise to use the *Check* prior to examining the value of *cnt*. If the check fails, the most likely problem is that an incorrect memory option was selected during installation (for example, choosing 512k when having purchased the 128k option). This results in an incorrect link command file being built during installation. If this is likely, please see Sec. 2.7 for help.

When you are successful, you will be able to view the *cnt* variable incrementing. Note, if you run for a sufficiently long period the variable with wrap-around to -2,147,483,648 and, then, continue incrementing.

1.2 Second Program - *flippingBit*

The purpose of the second program is to provide an introduction to the I/O elements on the board. The starting point for this program was the first program. The purpose of the program is to use a variable, *cnt*, to toggle the state of a discrete output bit. The listing for this program is provided below.

```
/*  flippingBit.c

    Author:  Brian R. Tibbetts, A.R.C.S., Inc.  22 July 1999

    This program extends counting.c to do a rudimentary i/o
    programming example.
*/

#include <dio.h>
#include <hdw_util.h>
#include <host_comm_01x.h>

int cnt = 0;

void init ()
{
   hdwInit (0, 0, 1, 0, 0, 0, 0);  /* Configure one DIO word */
   setDioWordConfig (0, 0x0001);   /* Make bit #0 an output  */
}


main()
{
   init ();

   while(1)
   {
      communications();
      cnt++;

      /* We'll use the counter least significant bit
         to flip a DO! */
      setDioWord (0, cnt & 0x0001);
   }
}
```

Here are the pertinent changes to the original code:

1. *Header files.* This program uses the discrete input/output (DIO) functionality; therefore, *dio.h* must be included.

2. *Initialization.* There were three changes made to the initialization code. First, it was placed into a separate function. This is strictly an organizational choice. Second, the third arguement in *hdwInit ()* is changed to reflect the use of one word of DIO. Third, the DIO word must be configured. In this example, only bit#0 is a discrete output.

3. *Output device.* Finally, the value of the bit is set according to whether *cnt* is odd or even:

```
setDioWord (0, cnt & 0x0001);
```

After building and running this code, Word #0, Bit #0 should toggle - rapidly. It may be appropriate to change toggle frequency by adjusting the use of *cnt* for driving the toggle action.

2 DSP Programming Elements

This section provides details on various aspects of programming the DSP system with the ARCS libraries. This section is not intended to provide an exhaustive reference on all available functions and API's (application programming interface). This information can be found in the on-line documentation, distributed in HTML format.

2.1 Communications

A great deal of care was taken in the design of the ARCS system to isolate the real-time processes from system crashes on the host computer. This design goal directly impacts the implementation of the communications processing.

The communication scheme is a command-based scheme where the host will issued a command to the DSP system. The command structure has a wide variation of capabilities. This can be as simple as an action command; for example, start application. Much more complex commands are also possible which combine command and data.

These commands are processed within the function, *communications ()*. Therefore, it is possible for the DSP system to ignore an issued command indefinitely. The benefit is that processing commands are not required to be addressed at the expense of the real-time processing. Indefinitely withholding command processing is not advisable since this is the mechanism for getting the current variable values, setting variable values, managing data acquisition, etc.

This approach does place an additional burden on the application developer, namely that the application be designed such that the *communications ()* function be called with adequate frequency. "Adequate frequency"

is relative to the particular application. For example, if an application does not require any on-line data monitoring, change of variable values, or data acquisition, it is reasonable to NOT call *communications ()*. However, it is important to recognize that, in this case, there will be no mechanism available for determining if the DSP system is functioning correctly from the host computer.

Typically, the design issue of greater concern is illustrated as follows:

```
while(1)
{
   communications();
   otherFunction (); /* Requires 10 seconds to execute! */
}
```

In this example, there is another function in background loop. This function requires approximately 10 seconds to execute. Thus, it has the effect of "gating" the communication with the host computer. It is the responsibility of the application developer to design the background processing so that the *communications ()* function is called much more frequently then commands are expected to be issued.

2.2 Exposing Variables

Variables within the DSP system code can be exposed to the host computer (monitored or changed) by declaring them to be file scope. Currently, the system can expose signed integers, floats, characters, and shorts. Primitives, arrays, and pointers can be exposed; however, unions and structures are not currently supported.

If a variable is declared in file scope, it is exposed. Therefore, if the application developer requires a variable to be declared in file scope but wishes

that it not be monitored or adjusted, a specific host application will be need to block access.

2.3 Timer-based Interrupts

This section addresses the crux of real-time system programming with the ARCS Lightning - interrupts and interrupt service routines (or interrupt handlers).

> ✍ **Note:** An **interrupt** is a hardware device. Different hardware peripherals may have conditions where special attention of the processor is required. The peripheral notifies the processor of the need for this attention by the means of a hardware signal, known as an interrupt. The required attention is coded within a special function known as an **interrupt service routine (ISR)** or an **interrupt handler**. When the interrupt is generated the current flow of execution is suspended, the ISR is executed, and processing is resumed where suspension occurred. For example, a timer is set for repetitive count down. When it reaches zero, an interrupt is generated and the counter is preset to its original value and continues to count down. An ISR is run in response to the interrupt.

2.3.1 Main Application Elements with Interrupts

This system will support up to two timer-based interrupts. There are five steps for programming timer-based interrupts on this system:

1. *Number of interrupt channels.* The number of interrupt channels are specified within the *hdwInit ()* function. The last argument is the number of interrupt channels that the application program will be

using. The current hardware design only supports two timer-based interrupts; therefore, only 0, 1, or 2 should be chosen for *x*:

```
hdwInit (a, b, c, d, e, f, g, x)
```

2. *ISR definition.* With one of two exceptions, an ISR definition is identical to a standard C function definition. The two possible exceptions are the required mechanism to inform the compiler that the function is an ISR. The compiler will then add functionality required to implement the context switching. The first possible exception is to include a special `PRAGMA` directive identifying the function as an ISR. In the following code example, this line identifies *timer0 ()* as an ISR:

   ```
   #pragma INTERRUPT (timer0)
   ```

   The other mechanism (not shown in the example) for declaring the ISR function is by use of a very specific naming convention. If the function name is chosen to be *c_initXX ()* where *XX* represents 01 to 99, the TI compile assumes the function to be an interrupt handler.

3. *Link interrupt to the ISR.* There are three elements involved: interrupt channels, interrupts (or interrupt levels), and ISR's. The interrupt channel is part of the application code which establishes the linkage between the interrupt and the ISR. The following line of code from the example illustrates how this linkage is established:

   ```
   initIsr (0, ISR_TIMER0, &timer0, NO_ISR, 0x0000);
   ```

   The first argument is the interrupt channel, the second argument is the interrupt level, the third argument is the address for the appropriate ISR, the forth and fifth auguements are for advanced functionality which will be addressed later.

   The possible interrupt levels are defined in *hdw_util.h*. Each level corresponds to a specific hardware or software device. Note, the priority is established by the level; lower numerical levels have higher priorities.

4. *Initialize timer.* The ARCS library provides a very simple function to establish the time interval: *setSamplingPeriod ().* The first argument is the interrupt channel number and the second is the time interval in seconds. The following line of code is used in the sample application:

```
setSamplingPeriod (0, SAMPLE_TIME);
```

5. *Enable interrupt.* The ARCS library also provides a very simple function to enable the interrupt: *enableIsr ().* The only argument provided is the interrupt channel number as shown in the example code:

```
enableIsr (0);
```

The following code uses a timer-based interrupt to produce an accurate sine wave.

> ◆◆ **Caution:** The purpose of this program is to illustrate interrupts, not to provide a completely correct embedded system program. Specifically, the implementation of the *time* variable will break for long duration use.

```
/*  sinewave1.c

    Author:  Brian R. Tibbetts, A.R.C.S., Inc.  22 July 1999

    This program demonstrates the implementation of a
    timer-based interrupt.  This program generates a sine
    wave which can have variable frequency and amplitude.

    Modified on:  24 October 1999, removed excess variable
    and changed amp value from 1.0 to 10.0.
*/

#include <math.h>

#include <hdw_util.h>
#include <host_comm_01x.h>

#define PI  3.141592
```

```
#define SAMPLE_TIME 0.01

float freq = 0.5;
float amp  = 10.0;
float time = 0.0;
float wave;
float execTime;

#pragma INTERRUPT (timer0)
void timer0 ()
{
   startIsrExecTime (0);

   wave = amp * sin(2.0 * freq * PI * time);
   time += SAMPLE_TIME;

   execTime = finishIsrExecTime (0);
}

void init ()
{
   hdwInit (0, 0, 0, 0, 0, 0, 1);
   initIsr (0, ISR_TIMER0, &timer0, NO_ISR, 0x0000);
   setSamplingPeriod (0, SAMPLE_TIME);
   enableIsr (0);
}

main()
{
   init ();

   while(1)
      communications();
}
```

2.3.2 Structure of the ISR

The required structure of the ISR is none; however, there is a recommended minimum structure. The minimum recommendation is shown below.

```
float execTime;
```

```
#pragma INTERRUPT (timer0)
void timer0 ()
{
   startIsrExecTime (0);
        ⋮
   execTime = finishIsrExecTime (0);
}
```

By implementing the *startIsrExecTime ()*/*finishIsrExecTime ()* pair, the execution time required for the ISR to complete. This time can be reported by the standard method of setting a file scope variable.

2.4 Data Acquisition

The on-line data monitoring capability is intended to provide gross information about the system, not precise, time-referenced data. This is the role of the data acquisition module in the ARCS system. The data acquisition subsystem is directly linked into one of the timer-based interrupts which provides a “hard” time reference. The following program, *sinewave2.c*, adds the data acquisition capability to the previous example.

There are three required elements to extend the application to include data acquisition capabilities:

1. Add the header file:

   ```
   #include <data_acq.h>
   ```

2. Extend the communications function to support the data acquisition capability:

   ```
   addCmdSeqHandler(0x1010, 0x10, &processDataAcqCmd);
   ```

   Note, this mechanism for extending the communications capability is also available to the application developer and will be detailed in Sec. **??**.

3. Link data acquisition with an ISR:

```
dataAcqUpdate(0);
```

   The chosen file scope variables are logged according to the data acquisition specification. A time reference, relative to start of the data acquisition process, is included in this log. Also, note that the argument is the number of the timer interrupt.

The complete program listing is provided below:

```
/*  sinewave2.c

    Author:  Brian R. Tibbetts, A.R.C.S., Inc.  22 July 1999

    This program makes a minor extension of sinewave1.c to
    demonstrate use of the built-in data acquisition capability.

    Modified on:  24 October 1999, removed excess variable
    and changed amp value from 1.0 to 10.0.
*/

#include <math.h>

#include <data_acq.h>
#include <hdw_util.h>
#include <host_comm_01x.h>

#define PI  3.141592
#define SAMPLE_TIME 0.01

float freq = 0.5;
float amp  = 10.0;
float time = 0.0;
float wave;
float execTime;

#pragma INTERRUPT (timer0)
void timer0 ()
{
   startIsrExecTime (0);

   wave = amp * sin(2.0 * freq * PI * time);
   time += SAMPLE_TIME;
```

```
    dataAcqUpdate(0);
    execTime = finishIsrExecTime (0);
}

void init ()
{
    hdwInit (0, 0, 0, 0, 0, 0, 1);
    addCmdSeqHandler(0x1010, 0x10, &processDataAcqCmd);

    initIsr (0, ISR_TIMER0, &timer0, NO_ISR, 0x0000);
    setSamplingPeriod (0, SAMPLE_TIME);
    enableIsr (0);
}

main()
{
    init ();

    while(1)
        communications();
}
```

2.5 DSP Library Organization

The overall A.R.C.S., Inc. -provided library is called *ac30i01a.lib*. This library includes all of the modules to interface with all of the I/O, the hardware memory map, communications with the PC, interrupt handling, low-level control, etc. So that developers may better control their application, these modules are also provided in a set of “fine-grained” libraries. For example, the library *ac30i01a_io.lib* contains all of the modules associated with the on-board input/output devices. Therefore, the developer can implement a more optimized I/O driver without incurring extra overhead from the A.R.C.S., Inc. software. The complete library structure is shown in Fig. 2.

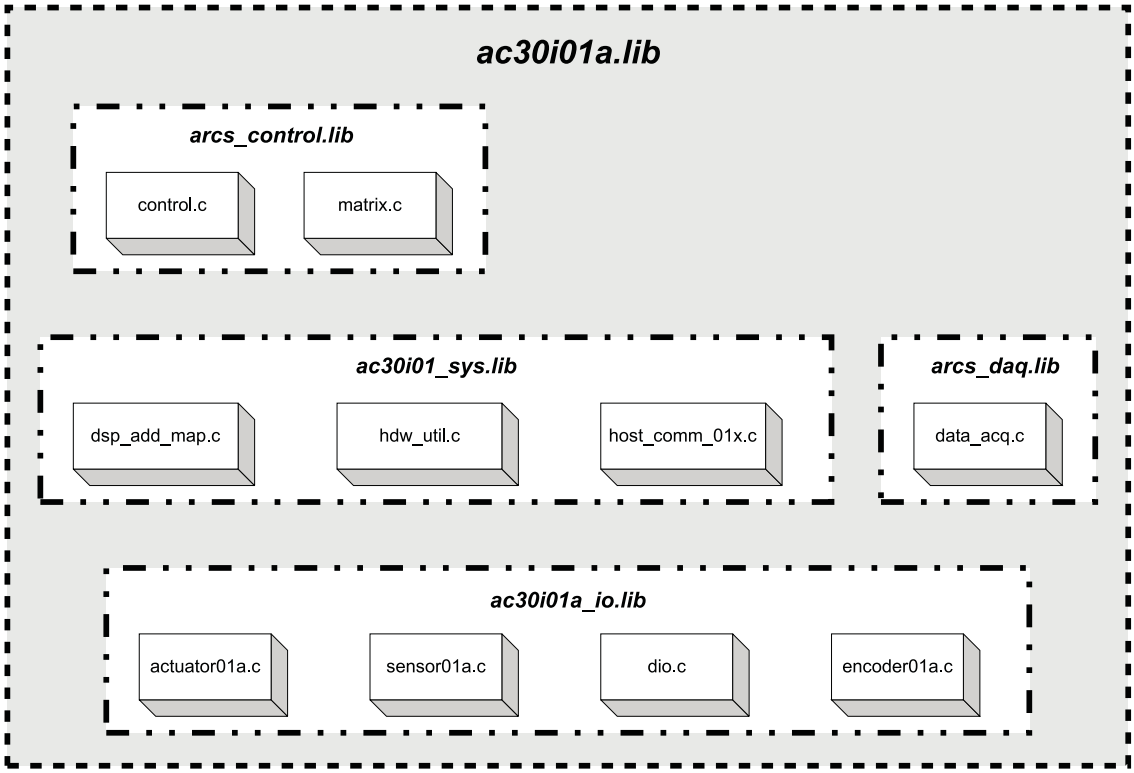

FIGURE 2: DSP Library Organization

2.6 General Application Programming

This section presents a more in-depth example of application programming. In this example, we consider controlling an x-y table to move between two specified points. These points may be hard-coded in the program, specified through *AIDE* , or linked to a variable from an external process.

Both timer based interrupts will be used. The first interrupt service routine (ISR0) will be used for the tight inner loop servo control. The second interrupt service routine (ISR1) will be used by the trajectory generator to provide position set points for ISR0.

The inner loop consists of the following components (running at 1ms loop rate in the sample code):

1. The x and y positions are read in from the encoders and converted into position variables.

2. Each position error (with the set point set by ISR1) are passed through a discrete PID filter to generate the corresponding command signal.

The desired path is a straight line between `x1` and `x2` (which contain preset values but may be modified by the user during run time through *AIDE* or link to variables published by external processes, e.g., machine vision). The outer loop generates the position set point for the x-y table going back and forth between `x1` and `x2` based on a trapezoidal velocity profile.

More detailed description of the outer loop, running at 10ms loop rate in the sample code, is given below:

1. Generate the current set poiont (and desired velocity and acceleration even though they are not used in the inner loop control in this case) based on the initial and final positions (either `x1` or `x2`), maximum velocity and acceleration, and the current traversal time on the trajectory. The initial and final velocities of the trajectory are set to zero. The velocity profile is selected to be trapezoidal, it is also possible to set it to S-curve (so that jerk is limited).

2. If the trajectory is finished, the trajectory timer is reset to zero and the initial and final positions are swapped.

3. The PID gains are reset in the outer loop as well. This allows the user to adjust the gains during run time.

Note that this example draws on the `control.c` and `tgen.c` API's. `control.c` contains general discrete filters in either state space or input/output (transfer function) forms, as well as specific filters such as the PID. `tgen.c` contains trajectory generation functions for one-dimensional straight line with either trapezoidal or S-curve velocity profiles. There are also functions for vector straight line path, Cartesian (x,y,z) straight line or circular

paths, and orientation (using rotation about the equivalent axis). Please refer to the corresponding API documentation (in html format) for detailed description.

We make the following input/output assumptions for this example:

1. The x-y positions are read in from encoders 0 and 1. The encoders have differential inputs and each contains 1024 lines and quadrature decoding. It is also assumed that the 1024 line correspond to 1m physical travel for each axis.
2. Commands to the amplifiers are through the D/A channels 0 and 1, and the range of amplifier input is -10V to 10V.

The initialization portion of the code consists of the following parts:

1. Hardware Initialization: We specify that the application will use the first two encoders, the first two D/A channels, and both timer interrupts.
2. Actuator Initialization: The physical range of the actuator input can be mapped to the D/A output. The full range of the physical D/A output is $\pm$10V. In the example code, we showed that the range is clamped at $\pm 8V$.
3. Encoder Initialization: We assume the starting position is (0,0). Each encoder is set to quadrature counting and 24-bit (due to the on-board encounter chip). The encoder counts are initialize to zero. The read out from the encoder is converted from the counts to physical units; in the sample code, we assume 1024 counts map to 1m travel.
4. Controller Initialization: The PID controllers are initialized.
5. Trajectory Generation Initialization: The initial trajectory time is set to zero. The initial motion is assumed to be from `x1` (the origin) to `x2`.

6. ISR Initialization: Both ISR are started and the periods set to the preset values (1ms for ISR0 and 10ms for ISR1).

7. Command Sequence Addition: Add the additional commands to support data acquisition.

The program consists of the following portions:

1. Include Files: The following include files are needed for this example.

   (a) `hdwutil.h`: Hardware and software initialization and interrupt support

   (b) `host_comm_01x.h`: Host communication

   (c) `data_acq.h`: Data acquition functions

   (d) `actuator.h`: D/A related functions

   (e) `encoder.h`: Encoder related functions

   (f) `control.h`: Control related functions

   (g) `tgen.h`: Trajectory generation related functions

2. Variable Declaration: All global variables (visible in *AIDE*) are declared in this section.

3. ISR0: The first interrupt service routine (running at higher rate) will be used for motor servoing based on the trajectory set point generated in ISR1.

4. ISR1: The second interrupt service routine (running at lower rate) will be used for trajectory generation, providing set point for ISR0.

5. Main program: The main program performs initialization and then enters into an infinite loop.

```
/* x_y.c

   Author: John T. Wen, A.R.C.S., Inc., January 15, 2000

   This program moves the an $x$-$y$ table between two specified
   points to demonstrate the use of both ISRs for control purposes.

*/

/* Include files */

#include <data_acq.h>
#include <hdw_util.h>
#include <host_comm_01x.h>
#include <actuator.h>
#include <encoder.h>
#include <control.h>
#include <tgen.h>

#define PI 3.14159265

/* set the sampling time */
#define SAMPLE_TIME_0 0.001
#define SAMPLE_TIME_1 0.01

/* declaration of global variables */

float execTime0, execTime1; /* execution times for ISR0 and ISR1 */
int actuator_on = 0;        /* flag to turn actuator on and off */
float uout[2] = {0.0,0.0};        /* command voltage */
float position[2];          /* position */
float perror[2];            /* position error */
float velocity[2];          /* estimated position */
float position_des[2];      /* position set point */

LTIsys *pidsys_x, *pidsys_y;/* discrete PID filter for x and y pos */

float kp=1.0,ki=0.2,kd=0.1; /* pre-set PID gains */

float x1[2]={0.0,0.0};      /* two target positions (may be */
float x2[2]={3.0,4.0};      /* modifiable during run-time   */

float v[2], a[2];           /* desired velocity and acceleration -- not used */
float tf, ta, tb;           /* total time, transition times */
float vmax = 10.0;          /* max line velocity */
float accmax = 20.0, decmax = 10; /* max acceleration and deceleration */
float traj_time = 0.0;      /* tgen timer */
float xo[2], xf[2];         /* current initial and final positions */

int return_code;

/* ISR0 */

#pragma INTERRUPT (timer0)
```

```
void timer0 ()
{
   unsigned int i;

   startIsrExecTime (0);

   /* read in x, y positions and form position error */
   for (i=0;i<2;i++)
     { position[i] = getPosition(i);perror[i] = position[i]-position_des[i];}

   /* only run the filter and output the command voltage if the
      actuator flag is set */
   if(actuator_on)
     {
       /* PID control */
       LTIfilter(pidsys_x,perror,uout);
       LTIfilter(pidsys_y,perror+1,uout+1);
       {setActuator ( 0, uout[0] );setActuator ( 1, uout[1] );}
     }
   else
     {setActuator(0, 0.0);setActuator(1, 0.0);}

   /* collect data for data acquisition if necessary
      compute execution time */
   dataAcqUpdate(0);
   execTime0 = finishIsrExecTime (0);
}

#pragma INTERRUPT (timer1)
void timer1 ()
{

  float x_temp[2];            /* temporary array */
  float inparam[7];            /* input parameter array */
  float outparam[7];           /* output parameter array */

  startIsrExecTime (1);

  /* straight line motion between xo and xf */
  inparam[0]=vmax;
  inparam[1]=accmax;
  inparam[2]=decmax;
  inparam[3]=j1;
  inparam[4]=j2;
  inparam[5]=k1;
  inparam[6]=k2;
  return_code=lineTraj(position_des, v, a, outparam, 2, xo, xf,
                       0.0, 0.0, inparam, traj_time, TRAPEZOIDAL);

  traj_time += SAMPLE_TIME_1;
  if (traj_time>tf) /* if time is up, switch xo and xf */
    {
      traj_time = 0;
      x_temp[0] = xo[0];x_temp[1] = xo[1];
```

```
      xo[0] = xf[0]; xo[1] = xf[1];
      xf[0] = x_temp[0]; xf[1] = x_temp[1];
    }

  /* update PID gains if necessary */

  setPID(pidsys_x, kp, ki, kd);
  setPID(pidsys_y, kp, ki, kd);

   /* collect data for data acquisition if necessary
      compute execution time */
   dataAcqUpdate(1);
   execTime1 = finishIsrExecTime (1);
}

void init ()
{
  unsigned int i;

  /* initialize 2 encoders and 2 D/A's */
   hdwInit(2,2,0,0,0,0,2);

   /* initialize actuator actuator channels */
   for (i=0;i<2;i++)
     initAct (i,-8.0,-8.0,8.0,8.0);

   for (i=0;i<2;i++)
     {
       position[i] = 0.0; /* assume starting position is origin */
       initEnc(i, 4, 24); /* quadrature decoding, 24-bit counter */
       setEncoder(i,0);   /* initialize encoder counts */
       initPosition(i,0,0.0,1024,1); /* assume 1024 lines are mapped
                                        into 1m */
     }

   updateAll();

   /* create 2 SISO PID filters */
   pidsys_x = makePID(SAMPLE_TIME_0);
   pidsys_y = makePID(SAMPLE_TIME_0);
   /* initialize trajectory generation timer */
   traj_time = 0.0;
   xo[0]=x1[0];xo[1]=x1[1];
   xf[0]=x2[0];xf[1]=x2[1];

   /* Initialize ISRs */
   initIsr(0, ISR_TIMER0, &timer0, NO_ISR, 0x0000);
   initIsr(1, ISR_TIMER1, &timer1, NO_ISR, 0x0000);
   setSamplingPeriod( 0, SAMPLE_TIME_0);
   setSamplingPeriod( 1, SAMPLE_TIME_1);
   enableIsr(0);
   enableIsr(1);

   addCmdSeqHandler(0x1010, 0x10, &processDataAcqCmd);
```

```
}

main()
{

  init ();
  while(1)
    communications();

}
```

2.7 Link Command File

After the application source code has been developed, there are two basic steps which are done to build an executable application:

1. Each source file is converted from C or assembly language code to machine-level code.
2. The machine-level code from the various source files and libraries are located in memory. This process is called ***linking***.

The *AIDE* uses a “link command file” to specify how the link is configured. An example link command file is shown below, augmented with line numbers.

```
L 1 -c                          /* LINK USING C CONVENTIONS      */
L 2 -stack 0x3c0                /* 960 WORD STACK                */
L 3 -heap  0x400                /* 1024 WORD HEAP                */
L 4 -l rts30.lib                /* GET RUN-TIME SUPPORT          */
L 5 -w                          /* WARN IF NOT IN SECTIONS       */
L 6 -x                          /* ALLOW RE-READING OF LIB       */
L 7
L 8 SECTIONS
L 9 {
L10    .vector: > VECT             /* INTERRUPT VECTOR TABLE     */
L11    .text:   > RAM2             /* CODE                       */
L12    .cinit:  > RAM2             /* C INITIALIZATION TABLES    */
L13    .const:  > RAM2             /* CONSTANTS                  */
L14    .stack:  > RAM1             /* SYSTEM STACK               */
```

```
L15     .sysmem: > RAM0                 /* DYNAMIC MEMORY             */
L16     .bss:    > RAM2, block 0x10000 /* VARIABLES                  */
L17     .data:   > RAM2                 /* DATA - NOT USED FOR C CODE */
L18 }
L19
L20 MEMORY
L21 {
L22     VECT: org = 0x00809fc1 len = 0x0000003f   /* VECTORS IN SRAM */
L23     RAM0: org = 0x00809800 len = 0x00000400   /* RAM BLOCK 0     */
L24     RAM1: org = 0x00809c00 len = 0x000003c0   /* RAM BLOCK 1     */
L25     RAM2: org = 0x00061000 len = 0x0001f000   /* EXTERNAL RAM    */
L26 }
```

There are three main sections to this file. The first section (unlabeled) contains various command-line options. The second section, *SECTIONS*, maps different code and data elements into the available memory areas. The third section, *MEMORY*, describes the available memory in the hardware.

MEMORY:

The areas, *VECT*, *RAM0*, and *RAM1*, are part of the processor. The entry for *VECT* is fixed by the DSP hardware and should never be modified. The division between *RAM0* and *RAM1* is arbitrary and is made based on an assumption about optimization of execution speed. These blocks can be combined, additional blocks added within this space, or the boundary can be moved. The only requirement is that the sum of the space be between `0x00809800` and `0x00809fc0`. This memory is the fastest available on board and should be allocated to heavily accessed elements.

RAM2 is the external (to the DSP) memory. Note, the first `0x1000` words are reserved by ARCS for the kernel. The remaining memory is available for use. Also, this memory space can be partitioned into a number of blocks.

SECTIONS:

Each section corresponds to different functions and have standard definitions within C and assembly language programming. Each line specifies which memory block that a given section will be located.

The areas of greatest interest are the stack and the heap. First, a brief background on these two items is useful. The stack provides the working memory space for the application. For example, when a function is called with arguments and returns a value, the arguments are placed on the stack by the calling routine and used by the routine. Similarly, the called function may return the value via the stack. The stack is also used for local variable storage. The third use of the stack is to save the processor state during an ISR. Thus, the depth of function calls (including recursion), number of local variables, and nesting of ISR's determine the required stack size.

The heap is used for dynamic memory allocation. This function is performed by *malloc ()*. The ARCS library uses dynamic memory during *hdwInit ()* and for data acquisition functions.

The stack will typically have the most frequent access and, therefore, one derives performance benefits having the stack located in the fast, DSP-internal memory. The assumption made in this file is that the heap is also a high access item and, thus, is best in the DSP-internal memory. Note that the stack is configured by a combination of lines 2, 14, and 24. Similarly, the heap is configured by a combination of lines 3, 15, and 23.

One of the more typical changes required will be for data acquisition. The typical system will use 512 words for the base configuration. Therefore, this file leaves only about 450 words for additional application work, such as data acquisition. For example, if two variables are being acquired (plus time which is always acquired), approximately 150 data points could be acquired as this file is configured. Let us supposes that 1000 data points should be acquired. This means that an estimated 3,500 words need to be available to the heap. The following modifications made 4096 words available to the heap, permitting the data to be acquired:

```
L 3 -heap 0x1000 /* 1024 WORD HEAP */
L15 .sysmem:  > RAM2 /* DYNAMIC MEMORY */
```

Texas Instruments has more documentation on issues related to memory allocation.

3 Remote Processes

Many complex systems cannot be easily implemented with a single computer. This difficulty may be due to the physical size of a machine, required interconnection of multiple machines, processing demands, etc. Another aspect is the drive to automate information transfer. For example, an advantage can be gained by issuing commands to machines from central control points - either supervisory persons or factory control systems. Another example is for information to automatically be transferred between a machine and the corporate databases.

A.R.C.S., Inc. has introduced its *RemoteNode* architecture to address these challenges. This architecture supports remote processing of data. This architecture enables rapid, event-level distributed control over standard EtherNet.

*ARCS*ware is shipped with a set of classes that allow a user to set up a *RemoteNode* which can communicate with the controller running on the motion control board. The *RemoteNode* publish data via a standard, open interface for consumption by one or many "agents". These agents could be a standard A.R.C.S., Inc. controller or a third-party/user-developed *RemoteNodeClient*.

For example, Fig. 3 shows a system with three individual nodes - two *RemoteNodes* and one *RemoteNodeClient*. Each of these nodes is an independent process, and could, in general, run on a physically distinct machine or processor. In this example, the three nodes are as follows:

1. Vision Node: The Vision Node is a *RemoteNode* which is responsible for acquiring image data and processing it. This node publishes 4 variables. These variables are the planar position and orientation of an object and the unique identifier on each object. Thus the data that the vision system must provide is a 4-tuple of the form (x, y, θ, j)

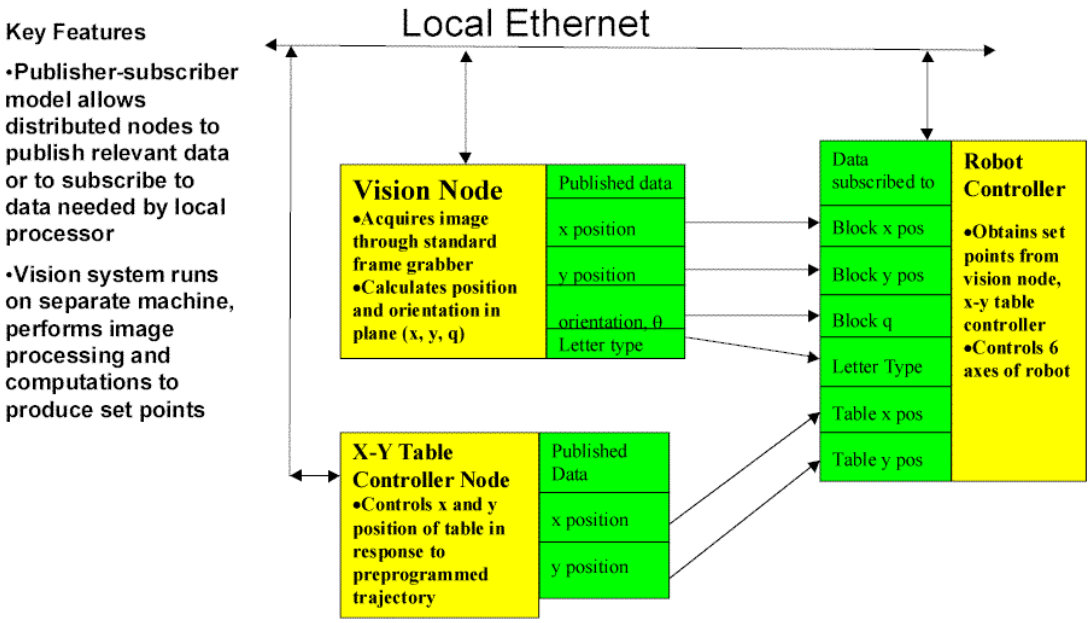

FIGURE 3: Remote Nodes for Distributed Control

where x, y, and θ are floating point numbers while j is the integer value corresponding to the unique letter on the object.

2. X-Y Table Controller: The X-Y Table Controller is also a *RemoteNode* which is responsible for controlling the x-y trajectory of a linear slide. It publishes the (x, y) table location to the robot controller.

3. Robot Controller: The Robot Controller is a *RemoteNodeClient* which is responsible for controlling the six axes of the robot. It has subscribed to the Vision Node. The location and orientation data is used to pick up the block. This node also subscribes to the X-Y Table Controller to determine where the table is located. This information, combined with the object identifier, is used to determine the final trajectory to move the block from the place where the vision identified it to the current position of the table.

3.1 RemoteNode Architecture

*ARCS*ware implements the *RemoteNode* architecture and provides all required base classes and interfaces for application specific systems. Also, *AIDE* has the *RemoteNodeClient* capability built-in. In this section, an overview of this architecture will be provided and the key features will be identified.

This architecture was designed so that multiple consumers or *RemoteNodeClient*'s can get information from a single source. This is done in an environment where connection can be easily and randomly lost. Table 1 provides top-level descriptions of each class.

A server will consist of a source-server object which is sub-classed from *AbsRemoteNodeServer*. Each time a *RemoteNodeClient* requests to be a client, the server will generate a *RemoteNodeImpl* (with its helper, *ClientChecker*. From this point forward, the data transfer and function calls are between the client and its associated *RemoteNode*.

A client consists of an object which extends *AbsRemoteNodeClient*. The client is responsible for initiated the request to receive data from a remote server.

Fig. 4 shows the *RemoteNode* architecture class diagrams in greater detail, including key data items and function calls. Key details are provided below.

The *RemoteNodeServer* interface requires the only the three methods shown in the diagram. The methods for adding and removing *RemoteNodeClients* are implemented by *AbsRemoteNodeServer*. The other method, `getAvailableDataItems()` is not implemented and must be implemented by the application-specific class which extends the server class.

The *AbsRemoteNodeServer* class has three key member data elements. `dataSpec` and `dataVector` are both *Vector* type objects. The element

TABLE 1: *RemoteNode* architecture class summary

Class	Description
Server (or Remote) Process	
AbsRemoteNodeServer	• the source or **producer** of the remote information. • an abstract class which manages the server-side infrastructure. • extended by the developer, adding the application-specific elements. • implements the *RemoteNodeServer* interface.
RemoteNodeImpl	This class is responsible for sending the data to its *RemoteNodeClient*. This class implements the *RemoteNode* interface which the client uses for various actions.
ClientChecker	This object is dedicated to a client and its verifies its health. If a client is not healthy, it causes the associated *RemoteNodeImpl* to be disposed.
Client Process	
AbsRemoteNodeClient	• the **consumer** of the remote information. • an abstract class which manages the client-side infrastructure. • extended by the developer, adding the application-specific elements. • implements the *RemoteNodeClient* interface.

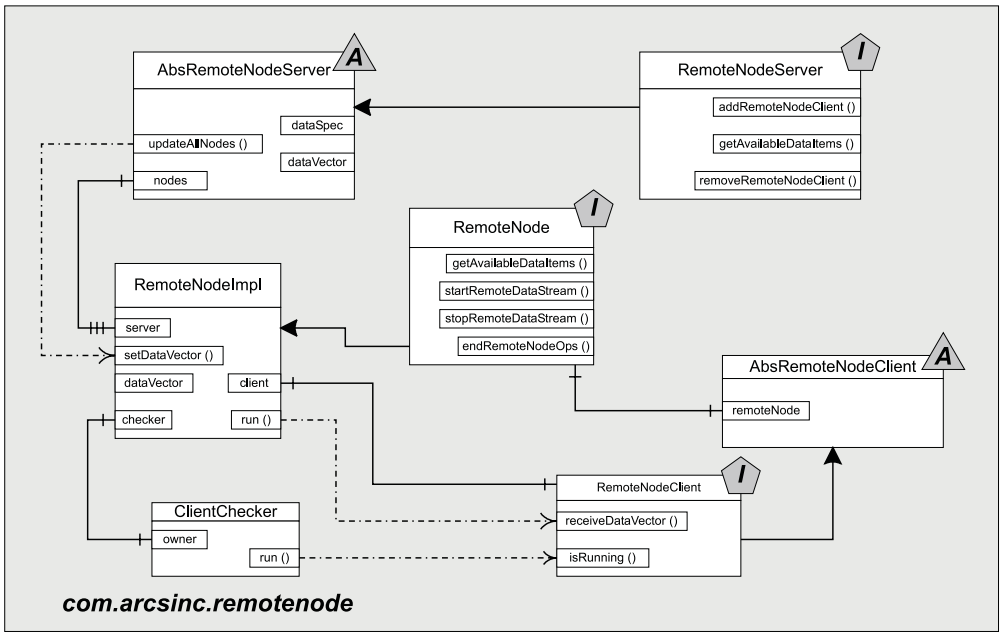

FIGURE 4: Remote Node classes in *ARCS*ware

`nodes` is also a *Vector* type and always contains references to *RemoteNodeImpl* objects currently instanciated.

The application developer must extext *AbsRemoteNodeServer* and do the following:

1. Populate `dataSpec` with serializable objects which describe the data being provided. For use with the standard *ARCS*ware infrastructure, this is *ExtVariableSpec* type.

2. Provide the `dataVector` with serializable objects. For use with the standard *ARCS*ware infrastructure, this is *Number* type, typically *Float* or *Integer*.

3. When the data in the *dataVector* has been modified, a call to `updateAllNodes()` is required.

4. Implement `getAvailableDataItems()`. This should return a copy of the *dataSpec* vector.

The *RemoteNodeImpl* is fully implemented with the member data elements and functions shown. Note that the client can suspend, resume, and end receiving data with standard function calls to this class's interface, *RemoteNode*. Also, its function, `getAvailableDataItems()` provides the same information as the server call.

The *AbsRemoteNodeClient* has one key member data element, a reference to its *RemoteNode*, and two required functions (by the *RemoteNodeClient* interface). The first function is `isRunning()` which is used in the health monitoring and implemented by the *AbsRemoteNodeClient* class. The second function is the `receiveDataVector()`. This function is receives the updated *dataVector* from the associated *RemoteNodeImpl* and its responsible for performing the required actions with this data. This function is not implemented by *AbsRemoteNodeClient* and is the responsibility of the developer.

Fig. 5 illustrates the typical interactions between a client and its *RemoteNodeImpl* on the server.

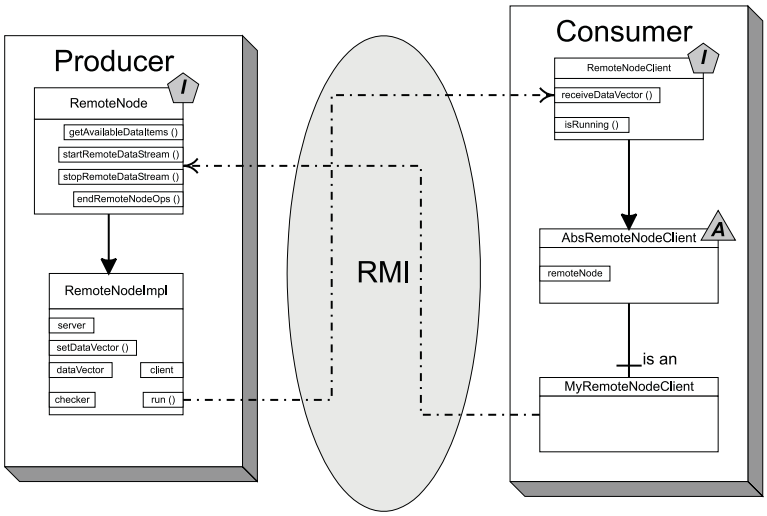

FIGURE 5: *RemoteNode* Data Transfer

3.2 Implementing a RemoteNode

A simple remote node server will be constructed which produces a count and a random number. These numbers are produced within a thread created in this application. The complete code listing can be found in the Appendix (Pg. 59). This program implements all of the steps shown on Pg. 3.1:

1. Populate the `dataSpec`.

```
dataSpec   = new Vector (2);
dataSpec.addElement (new ExtVariableSpec ("Iteration", "Remote",
                                          SymbolParams.T_INT,
                                          SymbolParams.DT_NON));
dataSpec.addElement (new ExtVariableSpec ("Value", "Remote",
                                          SymbolParams.T_FLOAT,
                                          SymbolParams.DT_NON));
```

   A key item to note is the arguments required by the *ExtVariableSpec* constructor. The first argument is the name of the variable being published. The second argument is not currently being used; "*Remote*" is inserted as a placeholder. The third argument is the type being published, i.e. SymbolParams.T_FLOAT for a *Float* in the `dataVector`. The last argument should always be SymbolParams.DT_NON.

2. Populate the `dataVector` and,

3. Call `updateAllNodes()`. These two steps are completed in the `run()` method of this class:

```
Integer tmpInt = (Integer) dataVector.elementAt (0);
tmpInt = new Integer (tmpInt.intValue () + 1);
dataVector.setElementAt (tmpInt, 0);
dataVector.setElementAt (new Float(randomGen.nextFloat ()), 1);

updateAllNodes ();
```

4. Implement `getAvailableDataItems()`.

```
public Vector getAvailableDataItems () throws RemoteException {
   return (Vector) dataSpec.clone (); }
```

Finally, the remote server must be made available. The required code is shown below. Note, various references on RMI will have details on these calls.

```
Registry reg = LocateRegistry.getRegistry (1099);
String remoteNodeName = new String ("SampleRemoteNode");
Naming.rebind(remoteNodeName, new SampleRemoteNode ());
```

This code is run by:

1. Starting the registry: `rmiregistry.exe`.

2. Starting the application: `java SampleRemoteNode`

The *RemoteNode* may be accessed from within *AIDE* or from another *RemoteNodeClient*. In the next section, a simple, stand-alone client is presented which connects to this server.

3.3 Implementing a RemoteNodeClient

The application, *SampeRemoteNodeClient*, is provided in the appendix (Pg. 60). The key elements of this code are:

1. Extend *AbsRemoteNodeClient* class.

2. Implement the function, `receiveDataVector`. For this application, the received data is simply printed out on the command line:

```
public boolean receiveDataVector (Vector dataVec)
{
   Integer tmpInt = (Integer) dataVec.elementAt (0);
   Float   tmpFlt = (Float) dataVec.elementAt (1);
   System.out.println (var1 + " #" + tmpInt.intValue () + "   "
                       + var2 + ":  " + tmpFlt.floatValue ());
   return true;
}
```

3. Initialization:

   (a) Find the server:

   ```
   RemoteNodeServer server
        = (RemoteNodeServer) Naming.lookup (urlString);
   ```

   (b) Add this object as a client:

   ```
   remoteNode = server.addRemoteNodeClient (this);
   ```

   (c) Keep a reference to the *RemoteNode*:

   ```
   setRemoteNode (remoteNode);
   ```

   (d) Get the published data items:

   ```
   Vector specVec = remoteNode.getAvailableDataItems ();
   ```

   (e) Finally, request that data from the *RemoteNode* is sent:

   ```
   thisClient.remoteNode.startRemoteDataStream ();
   ```

This application is run on the command line. For example, if the server has been started on coffee.arcsinc.com, the command line is:

```
java SampleRemoteNodeClient coffee.arcsinc.com
```

4 Java Programming Elements

This section describes the basic elements of programming a custom interface application. These elements include:

- Downloading and running a controller application.
- Monitoring a controller variable.
- Adjusting a controller variable.
- Plotting a controller variable.
- Responding to changing conditions on the controller hardware (using HdwEvent).

The focus in this section will be on using the *ARCS*ware libraries to build custom applications, not on general Java programming. There are a number of good references and tutorials commercially available to address general Java programming questions.

The following controller application was developed for use in this section. It is a modification of the basic sinewave demo program presented previously. The most significant change is the ability to stop the program with the *runComm* flag.

```
/*  sinewave3.c

    Author:  Brian R. Tibbetts, A.R.C.S., Inc.  24 October 1999

    This program makes a minor extension of sinewave1.c to for
    development of application-specific java programs and
    includes a communications termination feature.
*/

#include <math.h>

#include <hdw_util.h>
```

```
#include <host_comm_01x.h>

#define PI  3.141592
#define SAMPLE_TIME 0.01

int   runComm = 1;  // Change to 0 to finish app execution.
float freq    = 0.2;
float amp     = 10.0;
float time    = 0.0;
float wave;
float execTime;

#pragma INTERRUPT (timer0)
void timer0 ()
{
   startIsrExecTime (0);

   wave = amp * sin(2.0 * freq * PI * time);
   time += SAMPLE_TIME;

   execTime = finishIsrExecTime (0);
}


void init ()
{
   hdwInit (0, 0, 0, 0, 0, 0, 1);
   initIsr (0, ISR_TIMER0, &timer0, NO_ISR, 0x0000);
   setSamplingPeriod (0, SAMPLE_TIME);
   enableIsr (0);
}


main()
{
   init ();

   while(runComm)
      communications();
}
```

4.1 HdwHost - Configuring and Controlling

The first capability to address is to simply start a specified application on the controller; the following program accomplishes this (this program is provided with the java sample programs.):

```
/* StartSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  24 October 1999
           Modified:   BRT                    11 January 2000

   This application simply loads and runs a fixed dsp application -
   "sinewave3.c".
*/

import java.io.*;
import java.util.Vector;

import com.arcsinc.hardwarehost.*;
import com.arcsinc.util.*;

public class StartSineWave
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;

   StartSineWave ()
   {
      try
      {
         /* Create all of the hdwHost infrastructure.  We need the
            hdwHost (always needed) and hdwMgr (control the hardware,
            set the application, and modify variables).  Note, this
            is for local operations only.  Also, the
            ControllingClient class can be used for much of this
            functionality. */
         hdwHost       = new HdwHostImpl ();
         hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();

         /* This is an application specific program; therefore, the
            hardware will be initialized, the associated controller
            application will be set, and the application will be
            run. */
         hdwMgr.rebootHdw ();
         registerApp ("sinewave3");

         hdwMgr.downloadApp ();
         hdwMgr.startCntlr ();
      }
      catch (Exception e) { System.exit (0); }
   }

   public static void main (String args[]) throws IOException
   {
      StartSineWave startSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
         ArcsWareProperties.initialize (new File ("arcsware.props"));
```

```
        }
        catch (Exception e)   { System.exit (0); }

        // This line starts all the work.
        startSineWave = new StartSineWave ();

        // Determine if the operation was successful.
        if (startSineWave.hdwMgr.getControllerState ()
              == HdwMgr.RUNNING)
           System.out.println
              ("Successfully started sinewave application!");
        else
           System.out.println ("Start FAILED!!");

        System.out.println ("/n   ... Press Control-C to terminate.");
     }

     void registerApp (String appName)
     {
        byte[] data = getFileData (appName + ".out");
        hdwMgr.registerApp (appName, data, (float) 1.0);
        hdwMgr.setCurrentApp (appName);
     }

     /* Get data from the file into a byte array (binary info). */
     protected byte[] getFileData (String fileName)
     {
        File theFile;
        FileInputStream fis;
        byte[] fileData;

        theFile = new File (fileName);
        fileData = new byte[(int) theFile.length()];

        try
        {
           fis = new FileInputStream (theFile);
           fis.read (fileData);
           fis.close ();
        }
        catch (Exception e) { fileData = null; }

        return fileData;
     }
  }
```

The highest level logic is implemented in these six lines of the constructor:

```
        hdwHost       = new HdwHostImpl ();
        hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();
```

```
hdwMgr.rebootHdw ();
registerApp ("sinewave3");

hdwMgr.downloadApp ();
hdwMgr.startCntlr ();
```

The *hdwHost* object is the main "container" object for the PC interface and supervision of the controller hardware. The *HdwHostImpl* class can be used for either local or remote access; therefore, its constructor can throw a *RemoteException*. This is the reason that this chunk of code was placed into a *try-catch* structure. During the process of constructing this object, the *hdwMgr* object is also created. This object is responsible for the high-level management of the hardware, i.e. rebooting, loading applications, etc.

Once the references to the *hdwHost* and *hdwMgr* objects are available, the following actions are taken:

1. The controller is rebooted so that the kernel software is running on-board.
2. The specific application is registered with the *hdwMgr*:
   (a) The specific application, "sinewave3.out", is read into a `byte[]`,
   (b) The application name, "sinewave3", and its data array are sent to the *hdwMgr* (`hdwMgr.registerApp ("sinewave3")` (it's now in the application library.),
   (c) This application is made current. In other words, this application in the application library will be used by the controller.
3. The application is downloaded to the controller.
4. The controller runs the application.

The `main ()` function ties the application together:

1. Setup *ArcsWareProperties*. This is the mechanism where by the system knows how to find resources. For example, the location of the dynamic library for communication to the controller card is specified in this object.

2. The constructor for this application is invoked, executing the steps given above.

3. Finally, a check to determine if the controller application was started is done.

For an application where no user interface is required and the controller program is the only important part, this example Java application will do the trick. Typically, however, some user interface is required. This capability will be developed in the following sections.

4.2 Interacting with Variables

This section briefly demonstrates methods for monitoring, controlling, and plotting variables which are in the DSP application program. Each topic is demonstrated by extending the previous program. At the end of this section, the completed program is presented. This program, as well as the intermediate programs, are provided with the Java sample programs.

These programs are all based on the Java windowing environment, Swing. Some of these features will be discussed briefly; however, the intent is not to provide a Swing tutorial.

4.2.1 Monitoring

The monitoring application will be built on the application in the previous section, *StartSineWave*, and is called *MonitorSineWave*. This application is provided with the sample Java programs.

The following features are new:

1. **Windowing-specific code:** This application extends *JFrame*. This class provides a basic window for the application.

   Most of the windowing-specific code for this application has been collected in a function, *initMonitoredVarView()*. For this example, a *JSlider* component was chosen to display the data. This component is instantiated and configured. The component is added to a *JPanel* with a *GridBagLayout* layout manager. *ARCS*ware provides a helper class *UtilGui* for controlling the appearance of components being displayed by this layout manager.

```
protected void initMonitoredVarView (JPanel p)
{
   // Use a JSlider to show the sine wave output
   outputView = new JSlider (JSlider.HORIZONTAL, -100, 100, 0);
   outputView.setMinorTickSpacing (5);
   outputView.setMajorTickSpacing (25);
   outputView.setPaintTicks (true);
   outputView.setPaintLabels (true);
   outputView.setEnabled (false);

   // Place the JSlider and its label on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Output "), 0, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.CENTER,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, outputView, 0, 1, 1, 1,
                         GridBagConstraints.HORIZONTAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.45, 0.1);
}
```

   In the constructor, there is new code which initializes the windowing aspects of the application:

```
   //  Let's setup the window here:
   setSize (400, 120);
   // Activate the window widgets!
```

```
addWindowListener (new BasicWindowMonitor () );
JPanel contents = (JPanel) getContentPane ();
contents.setLayout (new GridBagLayout ());

initMonitoredVarView (contents);
setVisible (true);
```

Note the following items:

(a) *ARCS*ware provides a class called *BasicWindowMonitor* to work with the windows control (minimize, full screen, etc.).

(b) The *JPanel* for drawing the application has the appropriate layout manager set.

(c) The application-specific function (*initMonitoredVarView ()*) is called.

(d) The final step is to make the *JFrame* visible, e.g. displayed.

2. **DataClient:** This application implements the *DataClient* interface. Any application which wishes to get data from the on-line monitoring facility must implement this interface. The interface has one member function, *receiveData ()*. The connection between this application and the code which collects and provides the data is accomplished with the following line of code:

   ```
   clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
   ```

   This call registers this object (*MonitorSineWave*) so that the object's *receiveData ()* can be called when fresh data is available.

3. **Specifying the monitoring:** The application must declare which parameters to monitor and how often to monitor. The specific parameters to monitor are contained in a *Vector* and each element is a *ExtVariableSpec*. This class contains the symbol name, its module, and type information. In this example, the following instance is used:

```
new ExtVariableSpec ("_wave", "sinewave3.c", SymbolParams.T_FLOAT,
                     SymbolParams.DT_NON, null, -1)
```

This instance describes the *_wave* symbol in the *sinewave3.c* module. This symbol is a float. Once the vector (*monVec*) has been created, the monitored parameters and monitoring rate are specified:

```
clientDataMgr.setMonitoredParameters (monVec);
clientDataMgr.setOnLineLoggingPeriod (200);
```

These calls are collected in a utility function, `initLinkage()`.

4. **receiveData():** This function is called by a member of the HdwHost, the *clientDataManager*. This function must receive a *Vector* which contains the data and return a boolean, true for success. For this application, only one element is received, the value of the sine wave. The JSlider is set to this value. Note, the code written for this application is not responsible for the mechanics of getting fresh data; this functionality is provided by the basic functionality in *ARCS*ware by calling *receiveData()*.

```
public boolean receiveData (Vector monitoredParams)
{
   Float waveValue = (Float) monitoredParams.elementAt (0);
   outputView.setValue (waveValue.intValue ()); // Set the JSlider.
   return true;
}
```

The resulting display for this intermediate program is shown in Fig. 6.

4.2.2 Controlling

Now, adding the ability to control DSP application variables will be considered. Again, the application being developed in this section, *ControlSineWave* is based on the application from the previous section and is provided with the sample Java programs.

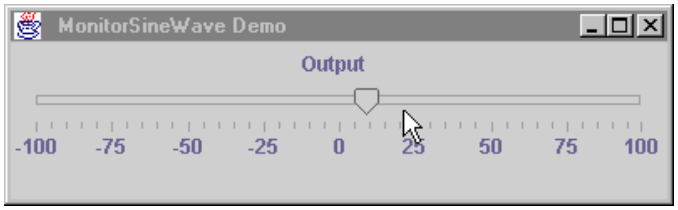

FIGURE 6: Sine wave monitoring GUI

Specifically, controls will be added for the sine wave period and the amplitude. *JSlider* will be used for both of these controls. Here are the major elements which are added to the previous code:

1. **Windowing Display:** There is no change to the structure of the windowing code. For this application, the code required to display the two control sliders is in *initAdjustedVarView()*. This code is analogous to the code for the monitored parameter display (see page 41).

2. **ChangeListener:** This application implements the *ChangeListener* interface. This interface requires that a function, *stateChanged()* be implemented. When the user changes the value of one of the controls, this function is called to respond to the change.

3. **stateChanged():** This function determines which control was changed by the user, retrieves its current value, and sets the associated parameter to the requested value. This function is implemented:

```
public void stateChanged (ChangeEvent e)
{
   Float adjValue;
   String symbolName;
   Vector adjVec = new Vector (1);

   // Determine the symbol name and value by which slider
   if (e.getSource () == ampCntl)
   {
      adjValue = new Float ((float) ampCntl.getValue ());
      symbolName = "_amp";
   }
```

```
        else
        {
           float tmpFlt = 1.0f / ((float) periodCntl.getValue ()
                                  / 1000.0f);
           adjValue = new Float (tmpFlt);
           symbolName = "_freq";
        }

        // Make a Vector with the Symbol info
        adjVec.addElement
           (new ExtVariableSpec (symbolName, "sinewave3.c",
                                 SymbolParams.T_FLOAT,
                                 SymbolParams.DT_NON, adjValue, 0));

        // Set the symbol
        hdwMgr.setParams (adjVec);
    }
```

The resulting display is similar to the display shown in Fig. 8, without the stop button or plot.

4.2.3 Plotting

While maintaining the existing montoring slider and two controlling sliders, a plot of the sinewave output will be added. The following elements are added to the program:

1. **Time:** The application needs to provide a time reference for the plotting. The member variable, *initTime* is added to the class. This value and the current system time will be used to determine the time for the plot.

2. **Create the plot:** The following lines of code within this application's function, *initPlot()*, create the plot.

```
PlotConfigModel pcm = new PlotConfigModel ();
pcm.setBgColorName ("lightGray");
pcm.setAxesColorName ("blue" );
pcm.setDataColorName ("magenta" );
```

```
pcm.setXAxisWidth (25.0);

plot     = new Plot (pcm, null);
```

3. **Add the plot:** A single call within *initPlot()* adds the plot to the panel for display.

4. **Send data to the plot:** The data is sent to the plot by adding a couple of lines of code to the existing *receiveData()* function:

```
DataPoint dp
 = new DataPoint ((System.currentTimeMillis () - initTime)
                  / 1000.0, waveValue.doubleValue ());
plot.receiveData (dp);
```

   First, a new *DataPoint* is created. The datapoint has the time calculated and the output of the sinewave. Next, the plot's *receiveData()* function is called with the *DataPoint* and the plot is updated.

The final code is shown below (it is also included with the sample Java programs):

```
/* PlotSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  26 October 1999
           Modified:   BRT                     8 January 2000
           Modified:   BRT                    11 January 2000
           Modified:   BRT                    30 January 2000

   This application adds plotting capibilities to ControlSineWave.java
*/

import java.io.*;
import java.util.Vector;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
```

```
import com.arcsinc.arcsclient.*;
import com.arcsinc.hardwarehost.*;
import com.arcsinc.hardwarehost.coff.*;
import com.arcsinc.hardwarehost.datamgmt.*;
import com.arcsinc.ui.UtilGUI;
import com.arcsinc.ui.BasicWindowMonitor;
import com.arcsinc.ui.plot.*;
import com.arcsinc.util.ArcsWareProperties;

public class PlotSineWave extends JFrame
   implements DataClient, ChangeListener
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;
   ClientDataMgr clientDataMgr;

   JSlider periodCntl;
   JSlider ampCntl;
   JSlider outputView;
   Plot    plot;
   long    initTime;

   PlotSineWave ()
   {
      super ("  PlotSineWave Demo  "); // call JFrame constructor

      try
      {
       /* Create all of the hdwHost infrastructure.  We need the
          hdwHost (always needed), hdwMgr (control the hardware, set
          the application, and modify variables), and a
          clientDataMgr (get variable values from the hardware).
          Note, this is for local operations only.  Also, the
          ControllingClient class can be used for much of this
          functionality. */
       hdwHost       = new HdwHostImpl ();
       hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();

       /* This is an application specific program; therefore, the
          hardware will be initialized, the associated controller
          application will be set, and the application will be run. */
        hdwMgr.rebootHdw ();
        registerApp ("sinewave3");

        hdwMgr.downloadApp ();
        hdwMgr.startCntlr ();
      }
      catch (Exception e) / System.exit (0); /

      //  Let's setup the window here:
      setSize (600, 300);
      // Activate the window widgets!
      addWindowListener (new BasicWindowMonitor () );
```

```
   JPanel contents = (JPanel) getContentPane ();
   contents.setLayout (new GridBagLayout ());

   // Add the monitoring feature here.
   initMonitoredVarView (contents);
   // Add the controlling features here.
   initAdjustedVarView (contents);
   initPlot (contents);       // Add the plot here.

   setVisible (true);
}

public void initLinkage ()
{
   clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
   specMonitoredParameters ();
   clientDataMgr.enableOnLineLogging (true);
}

public static void main (String args[]) throws IOException
{
   PlotSineWave plotSineWave;

   // The following is required so that the OS interface to the
   // hardware can be found.
   try {
    ArcsWareProperties.initialize (new File ("arcsware.props"));
   }
   catch (Exception e)   { System.exit (0); }

   // This line starts all the work.
   plotSineWave = new PlotSineWave ();
   plotSineWave.initLinkage ();
}

void registerApp (String appName)
{
   byte[] data = getFileData (appName + ".out");
   hdwMgr.registerApp (appName, data, (float) 1.0);
   hdwMgr.setCurrentApp (appName);
}

/* Get data from the file into a byte array (binary info). */
protected byte[] getFileData (String fileName)
{
   File theFile;
   FileInputStream fis;
   byte[] fileData;

   theFile = new File (fileName);
   fileData = new byte[(int) theFile.length()];

   try
   {
```

```
            fis = new FileInputStream (theFile);
            fis.read (fileData);
            fis.close ();
         }
         catch (Exception e) { fileData = null; }

         return fileData;
      }

      public void specMonitoredParameters ()
      {
         // Setup the variable monitor
         Vector monVec = new Vector (1);
         monVec.addElement
            (new ExtVariableSpec ("_wave", "sinewave3.c",
                                  SymbolParams.T_FLOAT,
                                  SymbolParams.DT_NON, null, -1));
         clientDataMgr.setMonitoredParameters (monVec);

         // Desire the data be updated every 200 mSec.
         clientDataMgr.setOnLineLoggingPeriod (200);
      }

      protected void initMonitoredVarView (JPanel p)
      {
         // Use a JSlider to show the sine wave output
         outputView = new JSlider (JSlider.VERTICAL, -100, 100, 0);
         outputView.setMinorTickSpacing (5);
         outputView.setMajorTickSpacing (25);
         outputView.setPaintTicks (true);
         outputView.setPaintLabels (true);
         outputView.setEnabled (false);

         // Place the JSlider and its label on the panel for display.
         UtilGUI.addComponent (p, new JLabel ("Output "), 3, 0, 1, 1,
                               GridBagConstraints.NONE,
                               GridBagConstraints.EAST,
                               3, 3, 3, 0.0, 0.0);

         UtilGUI.addComponent (p, outputView, 3, 1, 1, 1,
                               GridBagConstraints.VERTICAL,
                               GridBagConstraints.NORTHEAST,
                               3, 3, 3, 0.05, 0.1);
      }

      // This is the function which is called as the data is available.
      public boolean receiveData (Vector monitoredParams)
      {
         Float waveValue
          = (Float) monitoredParams.elementAt (0);

         // Set the JSlider here.
         outputView.setValue (waveValue.intValue ());
```

```
      // Update the plot here.
      DataPoint dp
       = new DataPoint ((System.currentTimeMillis () - initTime)
                        / 1000.0, waveValue.doubleValue ());
      plot.receiveData (dp);

      return true;
   }

   protected void initAdjustedVarView (JPanel p)
   {
      // Use a JSlider for control of period and amplitude.
      periodCntl = new JSlider (JSlider.HORIZONTAL, 0, 10000, 5000);
      periodCntl.setMinorTickSpacing (500);
      periodCntl.setMajorTickSpacing (2500);
      periodCntl.setPaintTicks (true);
      periodCntl.setPaintLabels (true);

      ampCntl = new JSlider (JSlider.VERTICAL, 0, 100, 10);
      ampCntl.setMinorTickSpacing (5);
      ampCntl.setMajorTickSpacing (25);
      ampCntl.setPaintTicks (true);
      ampCntl.setPaintLabels (true);

      // These lines link mouse actions to this class
      // ( stateChanged () )
      periodCntl.addChangeListener (this);
      ampCntl.addChangeListener (this);

      // Place the JSliders and its labels on the panel for display.
      UtilGUI.addComponent (p, new JLabel ("Amplitude"), 0, 0, 1, 1,
                            GridBagConstraints.NONE,
                            GridBagConstraints.WEST,
                            3, 3, 3, 0.0, 0.0);

      UtilGUI.addComponent (p, ampCntl, 0, 1, 1, 1,
                            GridBagConstraints.VERTICAL,
                            GridBagConstraints.NORTHWEST,
                            3, 3, 3, 0.05, 0.1);

      UtilGUI.addComponent (p, periodCntl, 0, 2, 4, 1,
                            GridBagConstraints.HORIZONTAL,
                            GridBagConstraints.NORTHEAST,
                            3, 3, 3, 0.45, 0.0);

      UtilGUI.addComponent (p, new JLabel ("Period (mSec)"),
                            0, 3, 4, 1, GridBagConstraints.NONE,
                            GridBagConstraints.CENTER,
                            3, 3, 3, 0.0, 0.0);
   }

   // Called when the use has adjusted one of the two control sliders
   public void stateChanged (ChangeEvent e)
   {
```

```
        Float adjValue;
        String symbolName;
        Vector adjVec = new Vector (1);

        // Determine the symbol name and value by which slider
        if (e.getSource () == ampCntl)
        {
           adjValue = new Float ((float) ampCntl.getValue ());
           symbolName = "_amp";
        }
        else
        {
           float tmpFlt = 1.0f / ((float) periodCntl.getValue () / 1000.0f);
           adjValue = new Float (tmpFlt);
           symbolName = "_freq";
        }

        // Make a Vector with the Symbol info
        adjVec.addElement
           (new ExtVariableSpec (symbolName, "sinewave3.c",
                                 SymbolParams.T_FLOAT,
                                 SymbolParams.DT_NON, adjValue, 0));

        // Set the symbol
        hdwMgr.setParams (adjVec);
     }

     void initPlot (JPanel p)
     {
        // initTime will be used to calculate the elapsed time for
        // display.
        initTime = System.currentTimeMillis ();

        // Configure the plot here.
        PlotConfigModel pcm = new PlotConfigModel ();
        pcm.setBgColorName ("lightGray");
        pcm.setAxesColorName ("blue" );
        pcm.setDataColorName ("magenta" );
        pcm.setXAxisWidth (25.0);

        // Create the plot and add it to the panel.
        plot     = new Plot (pcm, null);
        UtilGUI.addComponent (p, plot, 1, 0, 2, 2,
                              GridBagConstraints.BOTH,
                              GridBagConstraints.WEST,
                              3, 3, 3, 0.8, 0.8);
     }
  }
```

The resulting display is similar to the display shown in Fig. 8, without the stop button.

4.3 Dealing with HdwEvent

This is system uses the Java 1.1 event model to notify different “agents” of changes in the state of the system. These changes or *Events* can range from something very innocuous such as the current application has been updated or changed to a potentially serious issue such as a communications failure. Fig. 7 illustrates the software structure.

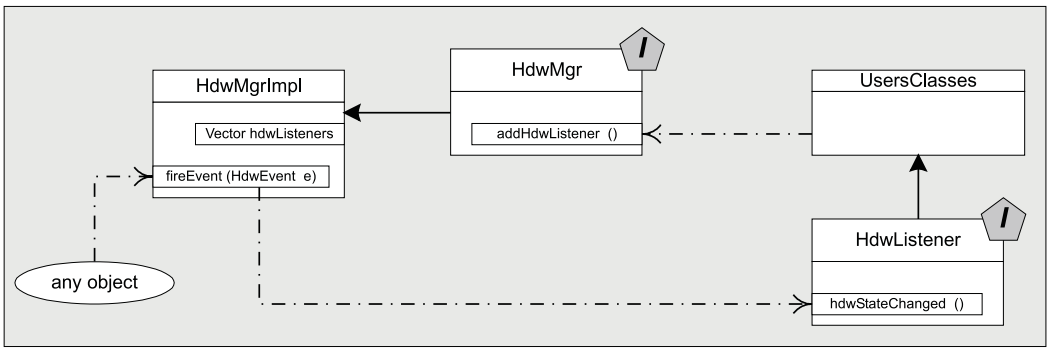

FIGURE 7: HdwEvent structure & Use

There are a number of key points:

1. Any class which wishes to receive these events must implement the *HdwListener* interface.
2. The *HdwListener* interface requires a single method:
   `hdwStateChanged(HdwEvent)`.
3. Once the class is instanciated, it needs to be registered as a *HdwListener* with the *HdwMgrImpl* (via the *HdwMgr* interface). The reference to this object is stored by the *HdwMgrImpl* object in a *Vector* (`hdwListeners`).
4. A *HdwEvent* is generated by an object. This object can be the *HdwMgrImpl* object, some other object which is part of the *ARCS*ware hardware host structure, or other objects. This object calls `fireEvent()` with the generated *HdwEvent*.

5. The `fireEvent ()` function uses the *Vector* of `hdwListeners` to call each listener's `hdwStateChange ()` function.

6. Each listener's `hdwStateChange ()` function contains its specific logic required to respond to the event.

For the example in this chapter, a flag, `runComm` was placed in the *sinewave3.c* code. This flag is initialized to 1; thus, the DSP `communications()` function is continuously run. When this flag is set to 0, the DSP application code terminates and the DSP system enters an undefined state. This is used to create a communications failure.

The event handler for this example is given below. Notice that only two of many possible events generate any action. For this example, if either a `HdwEvent.COMM_FAIL` or`HdwEvent.HDW_FAIL` is reported, the user is queried. The user can choose to exit or to restart the system.

```
public void  hdwStateChanged (HdwEvent e)
{
   switch (e.getEventType ())
   {
      case HdwEvent.COMM_FAIL:
      case HdwEvent.HDW_FAIL:
         int option
            = JOptionPane.showOptionDialog
            (this, "What would you like to do?", "Application stopped!",
             JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE,
             null, new Object[] "Restart", "Exit", "Restart");

         if (option != 0)
            System.exit (0);

         // Okay, restart the application and its plot.  Note, a
         // couple of corners were cut here.  To be complete,
         // either the sliders should be reset or the values should
         // be set in the application.
         plot.reinitialize ();
         hdwMgr.rebootHdw ();
         hdwMgr.downloadApp ();
         hdwMgr.startCntlr ();
         initTime = System.currentTimeMillis ();
         break;
```

```
        case HdwEvent.HDW_RESTARTED:
        case HdwEvent.CUR_APP_REMOVED:
        case HdwEvent.CUR_APP_CHANGED:
        case HdwEvent.CUR_APP_UPDATED:
        case HdwEvent.KERNEL_RUNNING:
        case HdwEvent.CUR_APP_RUNNING:
           break;
    }
}
```

As stated previously, this class must implement the *HdwListener* interface as indicated in this class statement:

```
public class EventSineWave extends JFrame
   implements DataClient, ChangeListener, HdwListener
```

After this class is instanciated, the object is registered as a listener. In this example, this is executed from the `initLinkage ()` function:

```
hdwMgr.addHdwListener (this);
```

In order to modify the `runComm` flag, a **Stop** button was added as well as its event handler to the `initAdjustedVarView()` function. The resulting application is shown in Fig. 8. The complete code listing can be found on Page 70.

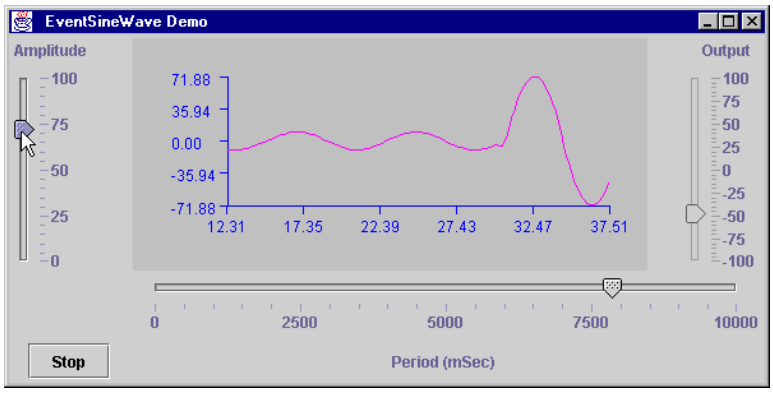

FIGURE 8: Sine wave application GUI with events

5 Java-based Architecture Reference

5.1 Hardware Host

5.2 Remote Node

5.3 Controlling Client

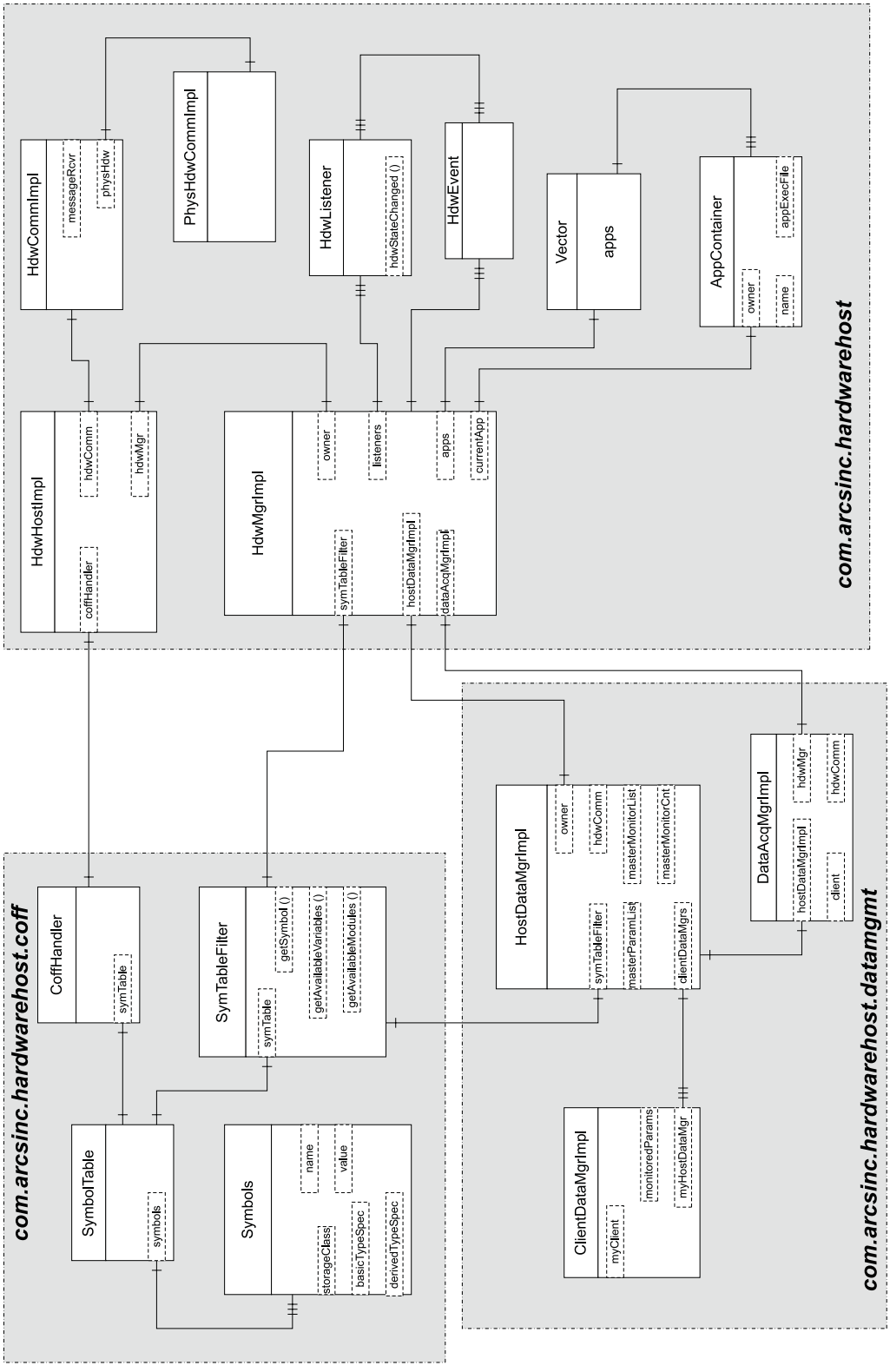

FIGURE 9: Object-oriented Diagram for HdwHost

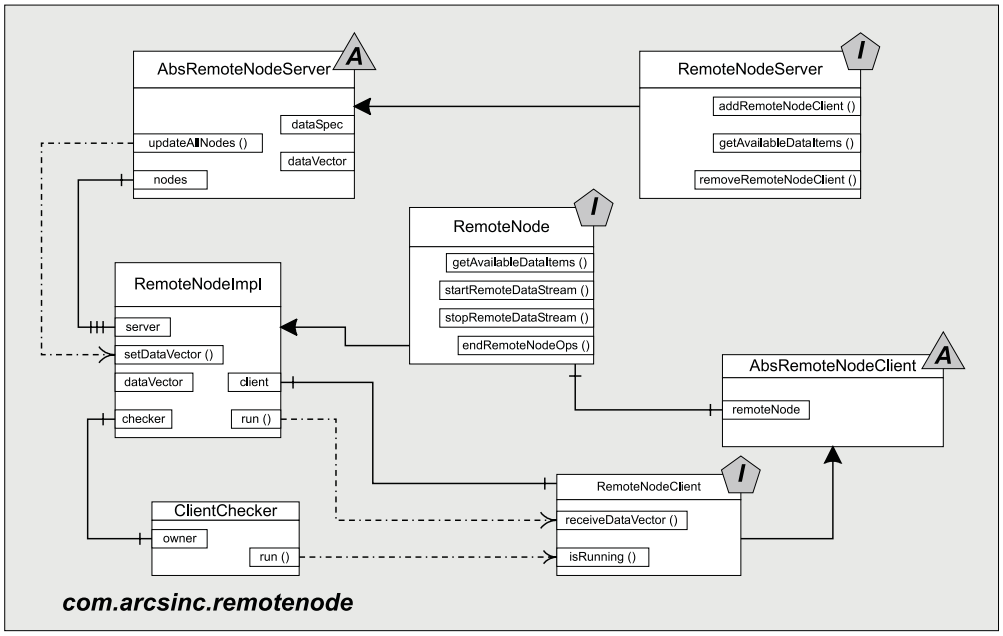

FIGURE 10: Object-oriented Diagram for Remote Node

A Java Sample Code

```
/* SampleRemoteNode.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  8 January 2000

   This application demonstrates the A.R.C.S. Remote Node capability.
   This node publishes:
      "Iteration" # of times a random number has been produced (int)
      "Value"     the random number (float)
*/

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Random;
import java.util.Vector;

import com.arcsinc.hardwarehost.coff.ExtVariableSpec;
import com.arcsinc.hardwarehost.coff.SymbolParams;
import com.arcsinc.remotenode.AbsRemoteNodeServer;

public class SampleRemoteNode extends AbsRemoteNodeServer implements Runnable
{
   Random randomGen;
   Thread thread;

   public SampleRemoteNode () throws RemoteException
   {
      super ();

      dataSpec   = new Vector (2);
      dataSpec.addElement (new ExtVariableSpec ("Iteration", "Remote",
                                                SymbolParams.T_INT,
                                                SymbolParams.DT_NON));
      dataSpec.addElement (new ExtVariableSpec ("Value", "Remote",
                                                SymbolParams.T_FLOAT,
                                                SymbolParams.DT_NON));

      randomGen    = new Random ( (long) 12454343);

      dataVector = new Vector (2);
      dataVector.addElement (new Integer (0));
      dataVector.addElement (new Float (randomGen.nextFloat ()));

      thread = new Thread (this);
      thread.setPriority ((Thread.currentThread ()).getPriority () - 1);
      thread.start ();
   }

   public void run ()
   {
      while (true)
      {
         try {
            Thread.sleep (5000); /* 5 sec between new values*/ }
         catch (InterruptedException ie) { }

         Integer tmpInt = (Integer) dataVector.elementAt (0);
         tmpInt = new Integer (tmpInt.intValue () + 1);
         dataVector.setElementAt (tmpInt, 0);
         dataVector.setElementAt (new Float(randomGen.nextFloat ()), 1);

         updateAllNodes ();
      }
   }

   public Vector getAvailableDataItems () throws RemoteException {
      return (Vector) dataSpec.clone (); }

   public static void main (String args[])
   {
      System.setSecurityManager (new RMISecurityManager());
      try
      {
         Registry reg = LocateRegistry.getRegistry (1099);
         String remoteNodeName = new String ("SampleRemoteNode");
         Naming.rebind(remoteNodeName, new SampleRemoteNode ());

         System.out.println ("This server object has been bound to"
                             + " the registry: " + remoteNodeName);
      }
      catch (Exception e) { e.printStackTrace(); }
   }
}
```

```
/* SampleRemoteNodeClient.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  8 January 2000

   This application demonstrates how to build a stand alone program to
   receive data from an A.R.C.S. Remote Node.  It is designed to
   receive data from SampleRemoteNode.  It is run from the cmd line in
   one of two ways:

        java SampleRemoteNodeClient host.name.com
        java SampleRemoteNodeClient 10.0.0.1
*/

import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.rmi.NotBoundException;
import java.rmi.UnknownHostException;
import java.rmi.Naming;
import java.util.Vector;

import com.arcsinc.hardwarehost.coff.ExtVariableSpec;
import com.arcsinc.remotenode.AbsRemoteNodeClient;
import com.arcsinc.remotenode.RemoteNodeServer;

public class SampleRemoteNodeClient extends AbsRemoteNodeClient
{
   String var1, var2;

   public SampleRemoteNodeClient () throws RemoteException
   {
      super ();
   }

   public void init (String hostName)
   {
      String urlString = "rmi://" + hostName + "/SampleRemoteNode";
      System.out.println ("Checking URL:  " + urlString);

      try
      {
         RemoteNodeServer server
            = (RemoteNodeServer) Naming.lookup (urlString);
         remoteNode = server.addRemoteNodeClient (this);
         setRemoteNode (remoteNode);

         Vector specVec = remoteNode.getAvailableDataItems ();
         ExtVariableSpec spec;

         spec = (ExtVariableSpec) specVec.elementAt (0);
         var1 = spec.getName ();
         spec = (ExtVariableSpec) specVec.elementAt (1);
         var2 = spec.getName ();
      }
      catch (NotBoundException re) {
         System.out.println ("Object not found!! Exiting now!");
         System.exit (0);
      }
      catch (UnknownHostException re) {
         System.out.println ("I can't find the server!! Exiting now!");
         System.exit (0);
      }
      catch (MalformedURLException re) {
         System.out.println ("Bad URL!! Exiting now!");
         System.exit (0);
      }
      catch (RemoteException re) {
         re.printStackTrace ();
         System.out.println ("Unknown RemoteException!! Exiting now!");
         System.exit (0);
      }
   }

   public boolean receiveDataVector (Vector dataVec)
   {
      Integer tmpInt = (Integer) dataVec.elementAt (0);
      Float   tmpFlt = (Float) dataVec.elementAt (1);
      System.out.println (var1 + " #" + tmpInt.intValue () + "   "
                          + var2 + ":  " + tmpFlt.floatValue ());
      return true;
   }

   public static void main (String args[])
   {
      SampleRemoteNodeClient thisClient;
      boolean toggle = false;

      try
      {
         thisClient = new SampleRemoteNodeClient ();
         thisClient.init (args[0]);
         thisClient.remoteNode.startRemoteDataStream ();

         while (true)
         {
            try {
               Thread.sleep (30000);  }
            catch (InterruptedException ie) { }
         }
      }
      catch (Exception e) { e.printStackTrace (); }
   }
}
```

```
/* StartSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  24 October 1999
           Modified:   BRT                    11 January 2000

   This application simply loads and runs a fixed dsp application -
   "sinewave3.c".
*/

import java.io.*;
import java.util.Vector;

import com.arcsinc.hardwarehost.*;
import com.arcsinc.util.*;

public class StartSineWave
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;

   StartSineWave ()
   {
      try
      {
         /* Create all of the hdwHost infrastructure.  We need the
            hdwHost (always needed) and hdwMgr (control the hardware,
            set the application, and modify variables).  Note, this
            is for local operations only.  Also, the
            ControllingClient class can be used for much of this
            functionality. */
         hdwHost       = new HdwHostImpl ();
         hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();

         /* This is an application specific program; therefore, the
            hardware will be initialized, the associated controller
            application will be set, and the application will be
            run. */
         hdwMgr.rebootHdw ();
         registerApp ("sinewave3");

         hdwMgr.downloadApp ();
         hdwMgr.startCntlr ();
      }
      catch (Exception e) { System.exit (0); }
   }

   public static void main (String args[]) throws IOException
   {
      StartSineWave startSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
         ArcsWareProperties.initialize (new File ("arcsware.props"));
      }
      catch (Exception e)   { System.exit (0); }

      // This line starts all the work.
      startSineWave = new StartSineWave ();

      // Determine if the operation was successful.
      if (startSineWave.hdwMgr.getControllerState ()
            == HdwMgr.RUNNING)
         System.out.println
            ("Successfully started sinewave application!");
      else
         System.out.println ("Start FAILED!!");

      System.out.println ("/n   ... Press Control-C to terminate.");
   }

   void registerApp (String appName)
   {
      byte[] data = getFileData (appName + ".out");
      hdwMgr.registerApp (appName, data, (float) 1.0);
      hdwMgr.setCurrentApp (appName);
   }

   /* Get data from the file into a byte array (binary info). */
   protected byte[] getFileData (String fileName)
   {
      File theFile;
      FileInputStream fis;
      byte[] fileData;

      theFile = new File (fileName);
      fileData = new byte[(int) theFile.length()];

      try
      {
         fis = new FileInputStream (theFile);
         fis.read (fileData);
         fis.close ();
      }
      catch (Exception e) { fileData = null; }

      return fileData;
   }

}
```

```
/* MonitorSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  24 October 1999
           Modified:   BRT                     8 January 2000
           Modified:   BRT                    11 January 2000
           Modified:   BRT                    22 January 2000

 This application loads and runs a fixed dsp application -
 "sinewave3.c".  The sine wave output is displayed with a JSlider.
*/

import java.io.*;
import java.util.Vector;

import java.awt.*;
import javax.swing.*;

import com.arcsinc.arcsclient.*;
import com.arcsinc.hardwarehost.*;
import com.arcsinc.hardwarehost.coff.*;
import com.arcsinc.hardwarehost.datamgmt.*;
import com.arcsinc.util.ArcsWareProperties;
import com.arcsinc.ui.UtilGUI;
import com.arcsinc.ui.BasicWindowMonitor;

public class MonitorSineWave extends JFrame implements DataClient
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;
   ClientDataMgr clientDataMgr;

   JSlider outputView;

   MonitorSineWave ()
   {
      super ("  MonitorSineWave Demo  "); // call JFrame constructor

      try
      {
       /* Create all of the hdwHost infrastructure.  We need the
          hdwHost (always needed), hdwMgr (control the hardware, set
          the application, and modify variables), and a
          clientDataMgr (get variable values from the hardware).
          Note, this is for local operations only.  Also, the
          ControllingClient class can be used for much of this
          functionality. */
       hdwHost       = new HdwHostImpl ();
       hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();
       clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);

       /* This is an application specific program; therefore, the
          hardware will be initialized, the associated controller
          application will be set, and the application will be run. */
       hdwMgr.rebootHdw ();
       registerApp ("sinewave3");

       hdwMgr.downloadApp ();
       hdwMgr.startCntlr ();
      }
      catch (Exception e) / System.exit (0); /

      //  Let's setup the window here:
      setSize (600, 300);
      // Activate the window widgets!
      addWindowListener (new BasicWindowMonitor () );
      JPanel contents = (JPanel) getContentPane ();
      contents.setLayout (new GridBagLayout ());

      initMonitoredVarView (contents);
      setVisible (true);
   }

   public void initLinkage ()
   {
      clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
      specMonitoredParameters ();
      clientDataMgr.enableOnLineLogging (true);
   }

   public static void main (String args[]) throws IOException
   {
      MonitorSineWave monitorSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
       ArcsWareProperties.initialize (new File ("arcsware.props"));
      }
      catch (Exception e)   { System.exit (0); }

      // This line starts all the work.
      monitorSineWave = new MonitorSineWave ();
      monitorSineWave.initLinkage ();
   }

   void registerApp (String appName)
   {
      byte[] data = getFileData (appName + ".out");
      hdwMgr.registerApp (appName, data, (float) 1.0);
      hdwMgr.setCurrentApp (appName);
   }

   /* Get data from the file into a byte array (binary info). */
   protected byte[] getFileData (String fileName)
   {
      File theFile;
```

```
   FileInputStream fis;
   byte[] fileData;

   theFile = new File (fileName);
   fileData = new byte[(int) theFile.length()];

   try
   {
      fis = new FileInputStream (theFile);
      fis.read (fileData);
      fis.close ();
   }
   catch (Exception e) { fileData = null; }

   return fileData;
}

public void specMonitoredParameters ()
{
   // Setup the variable monitor
   Vector monVec = new Vector (1);
   monVec.addElement
      (new ExtVariableSpec ("_wave", "sinewave3.c",
                            SymbolParams.T_FLOAT,
                            SymbolParams.DT_NON, null, -1));
   clientDataMgr.setMonitoredParameters (monVec);

   // Desire the data be updated every 200 mSec.
   clientDataMgr.setOnLineLoggingPeriod (200);
}

protected void initMonitoredVarView (JPanel p)
{
   // Use a JSlider to show the sine wave output
   outputView = new JSlider (JSlider.VERTICAL, -100, 100, 0);
   outputView.setMinorTickSpacing (5);
   outputView.setMajorTickSpacing (25);
   outputView.setPaintTicks (true);
   outputView.setPaintLabels (true);
   outputView.setEnabled (false);

   // Place the JSlider and its label on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Output "), 3, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.EAST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, outputView, 3, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.05, 0.1);
}

// This is the function which is called as the data is available.
public boolean receiveData (Vector monitoredParams)
{
   Float waveValue = (Float) monitoredParams.elementAt (0);
   outputView.setValue (waveValue.intValue ()); // Set the JSlider.
   return true;
}


}
```

```
/* ControlSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  24 October 1999
           Modified:   BRT                     8 January 2000
           Modified:   BRT                    11 January 2000
           Modified:   BRT                    22 January 2000

 This application loads and runs a fixed dsp application -
 "sinewave3.c".  The sine wave output is displayed with a JSlider
 and JSliders are provided for adjustment of the amplitude and
 period of the wave form.
*/

import java.io.*;
import java.util.Vector;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import com.arcsinc.arcsclient.*;
import com.arcsinc.hardwarehost.*;
import com.arcsinc.hardwarehost.coff.*;
import com.arcsinc.hardwarehost.datamgmt.*;
import com.arcsinc.util.ArcsWareProperties;
import com.arcsinc.ui.UtilGUI;
import com.arcsinc.ui.BasicWindowMonitor;

public class ControlSineWave extends JFrame
   implements DataClient, ChangeListener
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;
   ClientDataMgr clientDataMgr;

   JSlider periodCntl;
   JSlider ampCntl;
   JSlider outputView;

   ControlSineWave ()
   {
      super ("  ControlSineWave Demo  "); // call JFrame constructor

      try
      {
       /* Create all of the hdwHost infrastructure.  We need the
          hdwHost (always needed), hdwMgr (control the hardware, set
          the application, and modify variables), and a
          clientDataMgr (get variable values from the hardware).
          Note, this is for local operations only.  Also, the
          ControllingClient class can be used for much of this
          functionality. */
       hdwHost       = new HdwHostImpl ();
       hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();
       clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);

       /* This is an application specific program; therefore, the
          hardware will be initialized, the associated controller
          application will be set, and the application will be run. */
        hdwMgr.rebootHdw ();
        registerApp ("sinewave3");

        hdwMgr.downloadApp ();
        hdwMgr.startCntlr ();
      }
      catch (Exception e) / System.exit (0); /

      //  Let's setup the window here:
      setSize (600, 300);
      // Activate the window widgets!
      addWindowListener (new BasicWindowMonitor () );
      JPanel contents = (JPanel) getContentPane ();
      contents.setLayout (new GridBagLayout ());

      // Add the monitoring feature here.
      initMonitoredVarView (contents);
      // Add the controlling features here.
      initAdjustedVarView (contents);

      setVisible (true);
   }

   public void initLinkage ()
   {
      clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
      specMonitoredParameters ();
      clientDataMgr.enableOnLineLogging (true);
   }

   public static void main (String args[]) throws IOException
   {
      ControlSineWave controlSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
       ArcsWareProperties.initialize (new File ("arcsware.props"));
      }
      catch (Exception e)   { System.exit (0); }

      // This line starts all the work.
      controlSineWave = new ControlSineWave ();
      controlSineWave.initLinkage ();
   }
```

```
void registerApp (String appName)
{
   byte[] data = getFileData (appName + ".out");
   hdwMgr.registerApp (appName, data, (float) 1.0);
   hdwMgr.setCurrentApp (appName);
}

/* Get data from the file into a byte array (binary info). */
protected byte[] getFileData (String fileName)
{
   File theFile;
   FileInputStream fis;
   byte[] fileData;

   theFile = new File (fileName);
   fileData = new byte[(int) theFile.length()];

   try
   {
      fis = new FileInputStream (theFile);
      fis.read (fileData);
      fis.close ();
   }
   catch (Exception e) { fileData = null; }

   return fileData;
}

public void specMonitoredParameters ()
{
   // Setup the variable monitor
   Vector monVec = new Vector (1);
   monVec.addElement
      (new ExtVariableSpec ("_wave", "sinewave3.c",
                            SymbolParams.T_FLOAT,
                            SymbolParams.DT_NON, null, -1));
   clientDataMgr.setMonitoredParameters (monVec);

   // Desire the data be updated every 200 mSec.
   clientDataMgr.setOnLineLoggingPeriod (200);
}

protected void initMonitoredVarView (JPanel p)
{
   // Use a JSlider to show the sine wave output
   outputView = new JSlider (JSlider.VERTICAL, -100, 100, 0);
   outputView.setMinorTickSpacing (5);
   outputView.setMajorTickSpacing (25);
   outputView.setPaintTicks (true);
   outputView.setPaintLabels (true);
   outputView.setEnabled (false);

   // Place the JSlider and its label on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Output "), 3, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.EAST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, outputView, 3, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.05, 0.1);
}

// This is the function which is called as the data is available.
public boolean receiveData (Vector monitoredParams)
{
   Float waveValue = (Float) monitoredParams.elementAt (0);
   outputView.setValue (waveValue.intValue ()); // Set the JSlider.
   return true;
}

protected void initAdjustedVarView (JPanel p)
{
   // Use a JSlider for control of period and amplitude.
   periodCntl = new JSlider (JSlider.HORIZONTAL, 0, 10000, 5000);
   periodCntl.setMinorTickSpacing (500);
   periodCntl.setMajorTickSpacing (2500);
   periodCntl.setPaintTicks (true);
   periodCntl.setPaintLabels (true);

   ampCntl = new JSlider (JSlider.VERTICAL, 0, 100, 10);
   ampCntl.setMinorTickSpacing (5);
   ampCntl.setMajorTickSpacing (25);
   ampCntl.setPaintTicks (true);
   ampCntl.setPaintLabels (true);

   // These lines link mouse actions to this class
   // ( stateChanged () )
   periodCntl.addChangeListener (this);
   ampCntl.addChangeListener (this);

   // Place the JSliders and its labels on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Amplitude"), 0, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.WEST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, ampCntl, 0, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHWEST,
                         3, 3, 3, 0.05, 0.1);

   UtilGUI.addComponent (p, periodCntl, 0, 2, 4, 1,
                         GridBagConstraints.HORIZONTAL,
                         GridBagConstraints.NORTHEAST,
```

```
                        3, 3, 3, 0.45, 0.0);

   UtilGUI.addComponent (p, new JLabel ("Period (mSec)"),
                        0, 3, 4, 1, GridBagConstraints.NONE,
                        GridBagConstraints.CENTER,
                        3, 3, 3, 0.0, 0.0);
}

// Called when the use has adjusted one of the two control sliders
public void stateChanged (ChangeEvent e)
{
   Float adjValue;
   String symbolName;
   Vector adjVec = new Vector (1);

   // Determine the symbol name and value by which slider
   if (e.getSource () == ampCntl)
   {
      adjValue = new Float ((float) ampCntl.getValue ());
      symbolName = "_amp";
   }
   else
   {
      float tmpFlt = 1.0f / ((float) periodCntl.getValue () / 1000.0f);
      adjValue = new Float (tmpFlt);
      symbolName = "_freq";
   }

   // Make a Vector with the Symbol info
   adjVec.addElement
      (new ExtVariableSpec (symbolName, "sinewave3.c",
                            SymbolParams.T_FLOAT,
                            SymbolParams.DT_NON, adjValue, 0));

   // Set the symbol
   hdwMgr.setParams (adjVec);
  }

}
```

```
/* PlotSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  26 October 1999
           Modified:   BRT                     8 January 2000
           Modified:   BRT                    11 January 2000
           Modified:   BRT                    30 January 2000

   This application adds plotting capibilities to ControlSineWave.java
*/

import java.io.*;
import java.util.Vector;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import com.arcsinc.arcsclient.*;
import com.arcsinc.hardwarehost.*;
import com.arcsinc.hardwarehost.coff.*;
import com.arcsinc.hardwarehost.datamgmt.*;
import com.arcsinc.ui.UtilGUI;
import com.arcsinc.ui.BasicWindowMonitor;
import com.arcsinc.ui.plot.*;
import com.arcsinc.util.ArcsWareProperties;

public class PlotSineWave extends JFrame
   implements DataClient, ChangeListener
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;
   ClientDataMgr clientDataMgr;

   JSlider periodCntl;
   JSlider ampCntl;
   JSlider outputView;
   Plot    plot;
   long    initTime;

   PlotSineWave ()
   {
      super ("  PlotSineWave Demo  "); // call JFrame constructor

      try
      {
       /* Create all of the hdwHost infrastructure.  We need the
          hdwHost (always needed), hdwMgr (control the hardware, set
          the application, and modify variables), and a
          clientDataMgr (get variable values from the hardware).
          Note, this is for local operations only.  Also, the
          ControllingClient class can be used for much of this
          functionality. */
       hdwHost       = new HdwHostImpl ();
       hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();

       /* This is an application specific program; therefore, the
          hardware will be initialized, the associated controller
          application will be set, and the application will be run. */
        hdwMgr.rebootHdw ();
        registerApp ("sinewave3");

        hdwMgr.downloadApp ();
        hdwMgr.startCntlr ();
      }
      catch (Exception e) / System.exit (0); /

      //  Let's setup the window here:
      setSize (600, 300);
      // Activate the window widgets!
      addWindowListener (new BasicWindowMonitor () );
      JPanel contents = (JPanel) getContentPane ();
      contents.setLayout (new GridBagLayout ());

      // Add the monitoring feature here.
      initMonitoredVarView (contents);
      // Add the controlling features here.
      initAdjustedVarView (contents);
      initPlot (contents);       // Add the plot here.

      setVisible (true);
   }

   public void initLinkage ()
   {
      clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
      specMonitoredParameters ();
      clientDataMgr.enableOnLineLogging (true);
   }

   public static void main (String args[]) throws IOException
   {
      PlotSineWave plotSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
       ArcsWareProperties.initialize (new File ("arcsware.props"));
      }
      catch (Exception e)   { System.exit (0); }

      // This line starts all the work.
      plotSineWave = new PlotSineWave ();
      plotSineWave.initLinkage ();
   }
```

```
void registerApp (String appName)
{
   byte[] data = getFileData (appName + ".out");
   hdwMgr.registerApp (appName, data, (float) 1.0);
   hdwMgr.setCurrentApp (appName);
}

/* Get data from the file into a byte array (binary info). */
protected byte[] getFileData (String fileName)
{
   File theFile;
   FileInputStream fis;
   byte[] fileData;

   theFile = new File (fileName);
   fileData = new byte[(int) theFile.length()];

   try
   {
      fis = new FileInputStream (theFile);
      fis.read (fileData);
      fis.close ();
   }
   catch (Exception e) { fileData = null; }

   return fileData;
}

public void specMonitoredParameters ()
{
   // Setup the variable monitor
   Vector monVec = new Vector (1);
   monVec.addElement
      (new ExtVariableSpec ("_wave", "sinewave3.c",
                            SymbolParams.T_FLOAT,
                            SymbolParams.DT_NON, null, -1));
   clientDataMgr.setMonitoredParameters (monVec);

   // Desire the data be updated every 200 mSec.
   clientDataMgr.setOnLineLoggingPeriod (200);
}

protected void initMonitoredVarView (JPanel p)
{
   // Use a JSlider to show the sine wave output
   outputView = new JSlider (JSlider.VERTICAL, -100, 100, 0);
   outputView.setMinorTickSpacing (5);
   outputView.setMajorTickSpacing (25);
   outputView.setPaintTicks (true);
   outputView.setPaintLabels (true);
   outputView.setEnabled (false);

   // Place the JSlider and its label on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Output "), 3, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.EAST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, outputView, 3, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.05, 0.1);
}

// This is the function which is called as the data is available.
public boolean receiveData (Vector monitoredParams)
{
   Float waveValue = (Float) monitoredParams.elementAt (0);
   outputView.setValue (waveValue.intValue ()); // Set the JSlider.

   double time  = (System.currentTimeMillis () - initTime) / 1000.0;
   DataPoint dp = new DataPoint (time, waveValue.doubleValue ());
   plot.receiveData (dp);

   return true;
}

protected void initAdjustedVarView (JPanel p)
{
   // Use a JSlider for control of period and amplitude.
   periodCntl = new JSlider (JSlider.HORIZONTAL, 0, 10000, 5000);
   periodCntl.setMinorTickSpacing (500);
   periodCntl.setMajorTickSpacing (2500);
   periodCntl.setPaintTicks (true);
   periodCntl.setPaintLabels (true);

   ampCntl = new JSlider (JSlider.VERTICAL, 0, 100, 10);
   ampCntl.setMinorTickSpacing (5);
   ampCntl.setMajorTickSpacing (25);
   ampCntl.setPaintTicks (true);
   ampCntl.setPaintLabels (true);

   // These lines link mouse actions to this class
   // ( stateChanged () )
   periodCntl.addChangeListener (this);
   ampCntl.addChangeListener (this);

   // Place the JSliders and its labels on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Amplitude"), 0, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.WEST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, ampCntl, 0, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHWEST,
```

```
                          3, 3, 3, 0.05, 0.1);

   UtilGUI.addComponent (p, periodCntl, 0, 2, 4, 1,
                         GridBagConstraints.HORIZONTAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.45, 0.0);

   UtilGUI.addComponent (p, new JLabel ("Period (mSec)"),
                         0, 3, 4, 1, GridBagConstraints.NONE,
                         GridBagConstraints.CENTER,
                         3, 3, 3, 0.0, 0.0);
}

// Called when the use has adjusted one of the two control sliders
public void stateChanged (ChangeEvent e)
{
   Float adjValue;
   String symbolName;
   Vector adjVec = new Vector (1);

   // Determine the symbol name and value by which slider
   if (e.getSource () == ampCntl)
   {
      adjValue = new Float ((float) ampCntl.getValue ());
      symbolName = "_amp";
   }
   else
   {
      float tmpFlt = 1.0f / ((float) periodCntl.getValue () / 1000.0f);
      adjValue = new Float (tmpFlt);
      symbolName = "_freq";
   }

   // Make a Vector with the Symbol info
   adjVec.addElement
      (new ExtVariableSpec (symbolName, "sinewave3.c",
                            SymbolParams.T_FLOAT,
                            SymbolParams.DT_NON, adjValue, 0));

   // Set the symbol
   hdwMgr.setParams (adjVec);
}

void initPlot (JPanel p)
{
   // initTime will be used to calculate the elapsed time for
   // display.
   initTime = System.currentTimeMillis ();

   // Configure the plot here.
   PlotConfigModel pcm = new PlotConfigModel ();
   pcm.setBgColorName ("lightGray");
   pcm.setAxesColorName ("blue" );
   pcm.setDataColorName ("magenta" );
   pcm.setXAxisWidth (25.0);

   // Create the plot and add it to the panel.
   plot     = new Plot (pcm, null);
   UtilGUI.addComponent (p, plot, 1, 0, 2, 2,
                         GridBagConstraints.BOTH,
                         GridBagConstraints.WEST,
                         3, 3, 3, 0.8, 0.8);
}
}
```

```
/* EventSineWave.java

   Author: Brian R. Tibbetts, A.R.C.S., Inc.  26 October 1999
           Modified:   BRT                    11 January 2000
           Modified:   BRT                    22 January 2000

   This application expands on PlotSineWave.java to demonstrate the
   use of HdwEvent.
*/

import java.io.*;
import java.util.Vector;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

import com.arcsinc.arcsclient.*;
import com.arcsinc.hardwarehost.*;
import com.arcsinc.hardwarehost.coff.*;
import com.arcsinc.hardwarehost.datamgmt.*;
import com.arcsinc.ui.UtilGUI;
import com.arcsinc.ui.BasicWindowMonitor;
import com.arcsinc.ui.plot.*;
import com.arcsinc.util.ArcsWareProperties;

public class EventSineWave extends JFrame
   implements DataClient, ChangeListener, HdwListener
{
   HdwHostImpl hdwHost;
   HdwMgr hdwMgr;
   ClientDataMgr clientDataMgr;

   JSlider periodCntl;
   JSlider ampCntl;
   JSlider outputView;
   JButton stopButton;
   Plot    plot;
   long    initTime;

   EventSineWave ()
   {
      super ("  EventSineWave Demo  "); // call JFrame constructor

      try
      {
         /* Create all of the hdwHost infrastructure.  We need the
            hdwHost (always needed), hdwMgr (control the hardware, set
            the application, and modify variables), and a
            clientDataMgr (get variable values from the hardware).
            Note, this is for local operations only.  Also, the
            ControllingClient class can be used for much of this
            functionality. */
         hdwHost       = new HdwHostImpl ();
         hdwMgr        = ((HdwHostImpl) hdwHost).getHdwMgr ();

         /* This is an application specific program; therefore, the
            hardware will be initialized, the associated controller
            application will be set, and the application will be run. */
         hdwMgr.rebootHdw ();
         registerApp ("sinewave3");

         hdwMgr.downloadApp ();
         hdwMgr.startCntlr ();
      }
      catch (Exception e) { System.exit (0); }

      //  Let's setup the window here:
      setSize (600, 300);
      // Activate the window widgets!
      addWindowListener (new BasicWindowMonitor () );
      JPanel contents = (JPanel) getContentPane ();
      contents.setLayout (new GridBagLayout ());

      // Add the monitoring feature here.
      initMonitoredVarView (contents);
      // Add the controlling features here.
      initAdjustedVarView (contents);
      initPlot (contents);

      setVisible (true);
   }

   public void initLinkage ()
   {
      clientDataMgr = ((HdwHostImpl) hdwHost).addDataClient (this);
      hdwMgr.addHdwListener (this);
      specMonitoredParameters ();
      clientDataMgr.enableOnLineLogging (true);
   }

   public void  hdwStateChanged (HdwEvent e)
   {
      switch (e.getEventType ())
      {
         case HdwEvent.COMM_FAIL:
         case HdwEvent.HDW_FAIL:
            int option
               = JOptionPane.showOptionDialog
               (this, "What would you like to do?", "Application stopped!",
                JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE,
                null, new Object[] "Restart", "Exit", "Restart");

            if (option != 0)
               System.exit (0);
```

```
            // Okay, restart the application and its plot.  Note, a
            // couple of corners were cut here.  To be complete,
            // either the sliders should be reset or the values should
            // be set in the application.
            plot.reinitialize ();
            hdwMgr.rebootHdw ();
            hdwMgr.downloadApp ();
            hdwMgr.startCntlr ();
            initTime = System.currentTimeMillis ();
            break;

         case HdwEvent.HDW_RESTARTED:
         case HdwEvent.CUR_APP_REMOVED:
         case HdwEvent.CUR_APP_CHANGED:
         case HdwEvent.CUR_APP_UPDATED:
         case HdwEvent.KERNEL_RUNNING:
         case HdwEvent.CUR_APP_RUNNING:
            break;
      }
   }

public static void main (String args[]) throws IOException
   {
      EventSineWave eventSineWave;

      // The following is required so that the OS interface to the
      // hardware can be found.
      try {
       ArcsWareProperties.initialize (new File ("arcsware.props"));
      }
      catch (Exception e)   { System.exit (0); }

      // This line starts all the work.
      eventSineWave = new EventSineWave ();
      eventSineWave.initLinkage ();
   }

   void registerApp (String appName)
   {
      byte[] data = getFileData (appName + ".out");
      hdwMgr.registerApp (appName, data, (float) 1.0);
      hdwMgr.setCurrentApp (appName);
   }

   /* Get data from the file into a byte array (binary info). */
   protected byte[] getFileData (String fileName)
   {
      File theFile;
      FileInputStream fis;
      byte[] fileData;

      theFile = new File (fileName);
      fileData = new byte[(int) theFile.length()];

      try
      {
       fis = new FileInputStream (theFile);
       fis.read (fileData);
       fis.close ();
      }
      catch (Exception e) { fileData = null; }

      return fileData;
   }

   public void specMonitoredParameters ()
   {
      // Setup the variable monitor
      Vector monVec = new Vector (1);
      monVec.addElement
       (new ExtVariableSpec ("_wave", "sinewave3.c",
                             SymbolParams.T_FLOAT,
                             SymbolParams.DT_NON, null, -1));
      clientDataMgr.setMonitoredParameters (monVec);

      // Desire the data be updated every 200 mSec.
      clientDataMgr.setOnLineLoggingPeriod (200);
   }

   protected void initMonitoredVarView (JPanel p)
   {
      // Use a JSlider to show the sine wave output
      outputView = new JSlider (JSlider.VERTICAL, -100, 100, 0);
      outputView.setMinorTickSpacing (5);
      outputView.setMajorTickSpacing (25);
      outputView.setPaintTicks (true);
      outputView.setPaintLabels (true);
      outputView.setEnabled (false);

      // Place the JSlider and its label on the panel for display.
      UtilGUI.addComponent (p, new JLabel ("Output "), 3, 0, 1, 1,
                            GridBagConstraints.NONE,
                            GridBagConstraints.EAST,
                            3, 3, 3, 0.0, 0.0);

      UtilGUI.addComponent (p, outputView, 3, 1, 1, 1,
                            GridBagConstraints.VERTICAL,
                            GridBagConstraints.NORTHEAST,
                            3, 3, 3, 0.05, 0.1);
   }

   // This is the function which is called as the data is available.
   public boolean receiveData (Vector monitoredParams)
   {
      Float waveValue = (Float) monitoredParams.elementAt (0);
```

```
   outputView.setValue (waveValue.intValue ()); // Set the JSlider.

   double time  = (System.currentTimeMillis () - initTime) / 1000.0;
   DataPoint dp = new DataPoint (time, waveValue.doubleValue ());
   plot.receiveData (dp);  // Update the plot here.

   return true;
}

protected void initAdjustedVarView (JPanel p)
{
   // Use a JSlider for control of period and amplitude.
   periodCntl = new JSlider (JSlider.HORIZONTAL, 0, 10000, 5000);
   periodCntl.setMinorTickSpacing (500);
   periodCntl.setMajorTickSpacing (2500);
   periodCntl.setPaintTicks (true);
   periodCntl.setPaintLabels (true);

   ampCntl = new JSlider (JSlider.VERTICAL, 0, 100, 10);
   ampCntl.setMinorTickSpacing (5);
   ampCntl.setMajorTickSpacing (25);
   ampCntl.setPaintTicks (true);
   ampCntl.setPaintLabels (true);

   stopButton = new JButton ("Stop");
   stopButton.addActionListener (new ActionListener () {
    public void actionPerformed (ActionEvent e)
       {
          Vector adjVec = new Vector (1);
          adjVec.addElement
             (new ExtVariableSpec ("_runComm", "sinewave3.c",
                                   SymbolParams.T_INT,
                                   SymbolParams.DT_NON,
                                   new Integer (0), 0));

          hdwMgr.setParams (adjVec);

       }
   });

   // These lines link mouse actions to this class
   periodCntl.addChangeListener (this);
   ampCntl.addChangeListener (this);

   // Place the JSliders and its labels on the panel for display.
   UtilGUI.addComponent (p, new JLabel ("Amplitude"), 0, 0, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.WEST,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, ampCntl, 0, 1, 1, 1,
                         GridBagConstraints.VERTICAL,
                         GridBagConstraints.NORTHWEST,
                         3, 3, 3, 0.05, 0.1);

   UtilGUI.addComponent (p, periodCntl, 1, 2, 3, 1,
                         GridBagConstraints.HORIZONTAL,
                         GridBagConstraints.NORTHEAST,
                         3, 3, 3, 0.45, 0.0);

   UtilGUI.addComponent (p, new JLabel ("Period (mSec)"),
                         1, 3, 3, 1, GridBagConstraints.NONE,
                         GridBagConstraints.CENTER,
                         3, 3, 3, 0.0, 0.0);

   UtilGUI.addComponent (p, stopButton, 0, 3, 1, 1,
                         GridBagConstraints.NONE,
                         GridBagConstraints.CENTER,
                         3, 3, 3, 0.0, 0.0);
}

// Called when the use has adjusted one of the two control sliders
public void stateChanged (ChangeEvent e)
{
   Float adjValue;
   String symbolName;
   Vector adjVec = new Vector (1);

   // Determine the symbol name and value by which slider
   if (e.getSource () == ampCntl)
   {
    adjValue = new Float ((float) ampCntl.getValue ());
    symbolName = "_amp";
   }
   else
   {
    float tmpFlt = 1.0f / ((float) periodCntl.getValue () / 1000.0f);
    adjValue = new Float (tmpFlt);
    symbolName = "_freq";
   }

   // Make a Vector with the Symbol info
   adjVec.addElement
    (new ExtVariableSpec (symbolName, "sinewave3.c",
                          SymbolParams.T_FLOAT,
                          SymbolParams.DT_NON, adjValue, 0));

   // Set the symbol
   hdwMgr.setParams (adjVec);
}

void initPlot (JPanel p)
{
   // initTime will be used to calculate the elapsed time for
   // display.
   initTime = System.currentTimeMillis ();
```

```
        // Configure the plot here.
        PlotConfigModel pcm = new PlotConfigModel ();
        pcm.setBgColorName ("lightGray");
        pcm.setAxesColorName ("blue" );
        pcm.setDataColorName ("magenta" );
        pcm.setXAxisWidth (25.0);

        // Create the plot and add it to the panel.
        plot    = new Plot (pcm, null);

    UtilGUI.addComponent (p, plot, 1, 0, 2, 2,
                          GridBagConstraints.BOTH,
                          GridBagConstraints.WEST,
                          3, 3, 3, 0.8, 0.8);
            }
        }
```

Index