GenMon v2.0

ARBITRARY WAVEFORM GENERATION AND MONITORING SOFTWARE
FOR USE WITH THE LAYLA 24/96 SOUND SYSTEM BY
ECHO DIGITAL AUDIO

Programmer’s Reference MANUAL
1/17/2004

PREPARED BY TERRY LEACH
CONTRACT PROGRAMMER UNDER DR. BEN STERNBERG
# Table of Contents

## System Overview

1

## Software Operation Overview

3

### Sound System Control

4

### Waveform Component Definition

5

### Measurement Definition

7

### File Control

8

### Program Control

9

## Software Internals: An Overview

10

## Software Internals

13

### The Cin / Sound Driver Interface

13

#### Data From LabView into the Cin

13

#### Data From the Cin into LabView

15

#### The ASIO Compliant Sound Driver Thread and the Cin

16

### The ASIO.vi and Embedded CIN

18

#### Overview

18

#### Building the Cin: MSVC 6.0 Setup

21

#### Loading the Cin into LabView

26

#### Debugging the Cin

28

#### Within LabView

28

#### Correcting for Latency

28

## Cin Internals

29

### Architecture

29

### Main Modules

29

### AppGenMon

30

### Channel

30

### Layla

33

### Asio

34

### Wave

35

## Known Bugs

36

### Crash After N Activate/DeActivate Cycles

36
CHANNEL DOESN’T “SHUT-OFF” WHEN ALL COMPONENTS ARE DISABLED 36
MEASURED PHASE DOES NOT MATCH TRANSMITTED PHASE 36
SYSTEM OVERVIEW

This manual discusses the internals of the GenMon v2.0 software. The GenMon software was written to support arbitrary waveform generation and monitoring on up to 8 channels. In its current version the software supports arbitrary linear combinations of sine, square, saw-tooth, and triangular components to create waveforms on up to 8 channels. The software also allows for synchronized data acquisition on these channels. Acquired waveforms can be displayed in the time or frequency domain in real-time. FFT bin extraction and logging is also possible. The software will operate in conjunction with the Layla24’s external clocking capabilities ranging from 30kHz to 96kHz.

From a hardware standpoint, the Layla sound box is the only view of the system which the software has. This box should be powered on and externally clocked at a frequency which is a power of 2 if FFT bin extraction is desired. Seeing as the Layla has its own internal clock, you must use the ECHO configuration panel to tell the Layla to make use of external clocking. If you try to do this before you power on and connect the external clock, the ECHO configuration panel will ignore your attempt to setup external clocking. The Layla’s internal clock can clock at 48000hz or 96000hz, however neither of these were convenient FFT bin placement.

From a software standpoint, there are several elements which interact in real time. The first is the ECHO configuration panel (supplied by ECHO AUDIO corporation). Changes made on this panel affect GenMon software operation. GenMon software elements include a user interface written in LabView, a general ASIO interface which is a combination of LabView and a Code Interface Node (CIN) written in C++, and finally, an instantiation of an ASIO compliant sound driver thread used to manipulate the Layla sound box.

In order to use the GenMon software, you must install the ASIO compliant driver available from ECHO digital Audio’s web site. Install LabView 6.1, and install the GenMon .vi set. If you wish to modify the CIN, you will need to install the MicroSoft visual C++ 6.0 compiler followed by the supplied CIN source code. Follow LabView’s instructions related to building a CIN.

In this application, the ASIO sound driver was used in a very minimal manner. The CIN interface to this driver consists of loading or measuring one half of a sound double buffer. This loading or measuring is triggered when the driver thread makes use of a call back routine in the CIN. There is one sound double buffer for each D/A and A/D sound channel in active use on the Layla. While the call back routine in the CIN is loading (D/A channel) or measuring (A/D channel), the ASIO sound driver is manipulating the other ½ of the respective sound buffer via DMA and eventual communication with the Layla sound box. ASIO is a defined protocol for accessing/driving sound equipment in real time whose details are buried many levels into the CIN and need only concern those intent upon a complete re-write of the CIN software. The CIN provides call-back functions to the ASIO compliant thread, of which the main workhorse is
bufSW (buffer switch). Measured sound buffers are processed and new waveforms to be transmitted are handled by bufSW. When the bufSW function returns, this is the signal to the ASIO driver that the refreshed ½ of the sound buffer may now be accessed.

The CIN is meant to be driven by LabView and is packaged into the ASIO.vi. ASIO.vi is completely general and could be used to form the nucleus of alternate Layla24 based systems. The ASIO.vi may be run as a stand-alone program or called as a function in a larger application. The chief mechanism of use is to give a high level command (start, transmit, measure, stop, or query) in conjunction with two bit masks which indicate which A/D and D/A channels should be used.

The start command spawns the ASIO compliant sound driver thread. All other CIN operations are masked out, until a successful start-up has occurred. Other control operations include measurement of active A/D channels, transmission of waveforms on D/A channels, shut-down of the ASIO sound driver, and query of ASIO driver state.

Fundamentally the CIN is responsible for generating/acquiring waveforms from user supplied definitions. The CIN accepts input from and returns key parameters to the LabView user interface. Key system status, timing information, and real time waveforms are all returned by the CIN to the LabView user interface.
SOFTWARE OPERATION OVERVIEW

The typical user will see only the LabVeiw user interface, which coordinates the afore mentioned software elements. The main control panel of the user interface is shown below.

The user interface has five main components. These are 1) sound system control, 2) waveform generation definition, 3) measurement definition, 4) File control, and 5) program control. The typical procedure is to define how the sound system will be used, create waveform and measurement definitions and then save your entire configuration in an initialization file (.INI). This is followed by activating the sound system, and making waveform/measurement modifications as desired.

Each of the user interface areas are described below.
Sound System Control

The sound system controls shown below are used to communicate to all software components how the sound system will be used and directly affect system performance in time.

The first control is the rate at which the sound system is being clocked. This is not really a control as it cannot directly affect the clock rate on the Layla. It is used to record the current external (or internal) clock rate. The clock rate “control” basically affects FFT extraction and results display.

The second control lets you operate the sound system dynamically or statically. Running the sound system in Dynamic mode means that all 8 channels of A/D and D/A will be allocated and serviced once the sound system starts running. This option is only practical when you are using synchronized, as opposed to streamed waveforms, and low sample sizes. It’s advantage is dynamic use of all sound system channels, it’s price is that under many desired operating conditions, the software will not be able to keep up with the Layla hardware.

The “Static” option means only those D/A and A/D channels appearing in the waveform and measurement definitions will be activated. You will not be able to change channels while the system is running. Next you must select the sound buffer size. The sound buffer size determines how frequently the ASIO sound driver thread will call the GenMon software for new sound buffer information.
You must select one of the supported sound buffer sizes, the maximum being 16384 samples per ½ buffer. This control allows you to trade off throughput efficiency with speed of channel access. At a clock rate of 32768hz, a 16384 sound buffer can take up to 1 second before changes in your waveform will take effect (remember it’s a double buffer). At the same time, this involves the least number of calls to the ASIO.vi and minimizes the calls the sound driver thread must make to bufSW which is embedded in ASIO.vi of the GenMon software.

The next control lets you select the Sample size in units of sound buffers. For example a sample size of 4 and a sound buffer size of 16384 means that all sampled waveforms and FFT calculations will involve 4*16384 or 65536 points. Your trade off here is once again one of performance. FFT’s have a time complexity of n*ln(n) and for larger sample sizes will take enough time that the software won’t keep up with the Layla hardware. Since this performance varies from machine to machine, the best way to determine your operating limits is via trial and error.

The final control affects how waveforms are handled during generation and subsequent transmission. Static waveforms fill a sound buffer “exactly”. All sound buffers are always filled to capacity because the sound driver interface has no mechanism for handling partial sound buffers, so what is the meaning of “exactly”. Given an N point sound buffer, and the value of the sample to be output on the D/A as Wave(N), then “exactly” means Wave(1) = Wave(N+1). Meaning that each waveform on each active channel must repeat itself without any discontinuous breaks. In other words once the sound buffer has been loaded, it need not be changed unless the waveform definition changes. Streamed waveforms need not be exact (although nothing precludes Wave(1) = Wave(m*N+1)) and require a sound buffer update each time the sound driver thread calls the embedded bufSW function. Streamed waveforms require that the software perform all of its processing fast enough to stay synchronized with the Layla hardware.

**Waveform Component Definition**

The waveform controls shown below define linear combinations of waveform components on one or more D/A channels. Complex waveforms may be built from multiple component definitions on a given channel. These controls will determine which D/A channels are activated and serviced once the sound system is started via the “Activate” button. The exception to this occurs when you have set the sound system controls to “Dynamic” as described in the prior section, in which case ALL D/A channels are activated independent of waveform control settings.
These controls are a scrolling table, for all practical purposes you may define an unlimited number of components, subject only to available machine memory. The box in the upper left corner selects and indicates the row number displayed as the first row in the table. The box just below this indicates the total number of component definitions in the table.

Within the table, the first control is the active indicator which is either green or red. When green it indicates that the component will be included in the waveform for that D/A channel. This control lets you select a sub-set of the currently defined waveform components on any given channel.

The control immediately to the right selects the D/A channel for transmission. The frequency of transmission is next. While you may enter any desired real number in this box, note that some frequencies may not make sense given the configuration of the sound system. If you have your graphical display set to “FFT” such components will be immediately obvious. For example a sound buffer size of 16384, clock rate of 65536, sample size of 1, and synchronized waveforms, frequencies which are not a multiple of 4hz cannot be implemented. To be sure the system will calculate a 3hz sine wave for example, but it is not possible to fit an integral number of waveforms into 16384 points at the given clock rate, so when the next sound buffer is requested by the sound driver thread there will be a discontinuity, making your transmitted waveform something other than a 3hz sine wave. The easiest way to achieve a 3hz waveform given the above parameters would be to select “Streaming” waveforms as opposed to “Synchronized”.

The next control selects the functional form for waveform generation. Choices of Sine, Triangular, Square, and Saw Tooth have been implemented. Other choices have been allowed for but not yet implemented, if selected, they will default to Sine wave generation.

The next control is not yet implemented.

The next pair of controls, defines the magnitude and phase of the transmitted waveform via a complex number pair. The number on the left is the real component.
The final control is not yet implemented, and will allow selection of a file containing a stream of binary bit patterns to be sent to the sound system.

**Measurement Definition**

These controls define the A/D channels to take measurements from. At this point in time it is only possible to take one type of measurement. One or more extracted bins from an FFT of the measured waveform on the desired A/D channel is the only measurement possible. Also, included in this control set, is selection of type of real time display. Each active A/D channel will have an associated real-time display. You can choose FFT or time domain waveform display, or you can turn all real-time displays off. When generating streamed waveforms, or calculating large FFT’s, you may not want to pay the time overhead of the real time displays. These can be shut off by holding the shift key while clicking the active real-time display off.

The down sample button assures that no matter how many points have been collected in your measured waveform, a maximum of 65536 points will be sent to the FFT calculation. This data reduction is achieved through down sampling and is only effective if initial FFT’s indicate there are no high frequency components which will be aliased. Future development may include re-sampling of an FIR filter output prior to FFT calculation. The rate of down sampling is determined by the ratio of your acquired waveform size divided by 65536.

Like the waveform definition control, the measurement definition control is a scrolling table whose upper left corner box selects the top row to be displayed in the table. Within the table, the leftmost control selects the A/D channel on which to acquire a waveform. Next is the frequency to be extracted from the FFT of that channel’s waveform. Next is a selection of units for display of the extracted bin. The next two values are derived from the extracted FFT bin. The last component is a late flag indicating if the GenMon software got behind in collecting...
measured waveforms from the sound driver thread. This control will be red each time a measurement is late. Red indicates that at least 1 sound buffer was lost in the waveform measurement.

**File Control**

The file controls allow you to load and save the settings and values of all GenMon controls. They also allow logging of data collected and displayed in the measurement control section.

Upon program start up, the software will automatically load the GenMon.INI configuration file. On program exit, it will ask the user if GenMon.INI should be updated to reflect the most recent operating parameters of the software. Once the program is started, the user is free to load/save initialization files one or more times.

To log data, a data file name is first entered or browsed for. A block of data will be written to the file each time the “Write Data” button is pressed. All active rows in the measurement definition table will be written to the file. There are two additional control components (currently hidden) which will allow a once per file comment and a once per data block comment.

The “Write Data” button may be pressed at any time whether or not the sound system is active.
Program Control

The program control section consists of making the sound system active/inactive and exiting the GenMon software. Once all control banks have been configured, latching the “Activate” button will start and run the sound system until this button is un-latched.

Once you have completed a data taking session, use the “Stop Program” button to exit the GenMon application. This button may be pressed whether or not the sound system is currently active. When pressed, the software will deactivate the sound system thread automatically, and ask the user for confirmation of an overwrite of the GenMon.INI file. The program will then exit closing all active application windows.
SOFTWARE INTERNALS; AN OVERVIEW

The GenMon software consists of three main components. An ASIO compliant sound driver thread, a code interface node (CIN) integrated directly into the LabView ASIO VI, and a collection of LabView modules providing a high level user interface.
The application is started when the user runs GenMon2 via any of the normal Windows mechanisms. This starts the main LabView module GenMon2.vi which displays the main application panel. A default initialization file (GenMon.INI) is loaded, which restores all controls and tables to their prior operating state. At this point, the sound system is not operating.

The application is now waiting for user input which typically involves loading an initialization file of choice. Alternatively the user may decide to complete a waveform transmission/acquisition definitions from scratch. The sound system will be activated when the “Activate” button is pressed. This activation involves reading and interpretation of the Sound System, Waveform Definition and Measurement definition controls. Among other things, these controls will determine how many A/D and D/A channels will be active during sound system operation. Upon successful sound system activation, the GenMon software will begin servicing the sound buffers, transmitting new waveforms and measuring waveforms as timed by the sound driver.

More specifically, the “Activate” button results in a call to the ASIO.vi which in turn calls the embedded CIN passing a control code which tells the CIN to spawn the ASIO sound driver thread. This spawning involves sound buffer allocation for active D/A, A/D channels and preparation of a list of call-back functions to be used by the ASIO driver thread in communicating with the CIN. The ASIO driver immediately makes several 1 time call backs for configuration information and then calls bufSW for the first time. bufSW is a function inside the CIN which has been embedded into ASIO.vi. bufSW loads zeros into all active sound driver transmission buffers and then returns to the ASIO driver thread. During this same block of time the CIN has returned status codes to higher level GenMon software indicating if the sound system was successfully started.

The ASIO sound driver thread is now running asynchronously, calling bufSW repeatedly each time a D/A is ready for new waveform transmission and/or an A/D channel has been measured. Initially bufSW satisfies these requests with “silence” or zeros on active D/A channels of the Layla sound box, and measures all active A/D channels although nothing is done with these measurements. This timing of the sound driver thread is communicated through status bits to higher level GenMon code. Once GenMon receives status bits indicating that the sound system is now running, it will begin polling other status bits indicating timing of the sound driver thread. One transmission requested bit is set for each active D/A channel immediately after bufSW has sent the most current buffer to the sound driver thread. Likewise a measurement requested bit is set for each active A/D channel just recorded by bufSW. Upon seeing one or more transmission request bits, the GenMon software will pass updated
waveforms to the CIN, so they will be ready for the next time the asio driver thread calls bufSW. Similarly, the GenMon software will collect current waveform measurement segments via a call to the CIN embedded in the ASIO vi.

In addition to calling ASIO.vi and starting the sound driver, the “Activate” button results in creation of graphical displays for each measured channel. These graphical displays are brought up as asynchronous vi’s permitting updates while simultaneously servicing the controls of the User interface. Once the results windows are displayed, the LabView side of the application handles all user interaction through its GUI. Initialization files can be loaded, edited and saved. Data storage files can be selected and updated. Real time FFT’s or Waveforms can be printed and viewed. None of these operations cause the CIN to be called. They all take place completely within the LabView environment. Meanwhile the ASIO sound driver thread is in communication with the CIN via the bufSW callback.

When the “Activate” button is latched, the GenMon software polls the ASIO vi to determine if it is time to measure and/or transmit a waveform. It is also through this polling that GenMon determines if it is late making a measurement. **The ASIO vi can be run stand alone.** Its front panel can also be displayed in real-time during GenMon operation. This is usually very helpful if making sound system related modifications to the software or developing a new application around the ASIO vi.

An FFT is computed for each measured waveform associated with an A/D channel and the results are displayed in the measurement table.

The cycle of transmission, measurement, results calculation and display continues in real-time until the user un-latches the “Activate” button. Upon un-latching, GenMon tells the ASIO vi to stop the sound driver thread. The ASIO vi calls the embedded CIN, at which point sound buffer memory is released and the sound driver thread terminated.
SOFTWARE INTERNALS

The CIN / Sound Driver interface

Data from LabView into the CIN

The LabView user interface passes the following elements into the CIN.

```c
// INPUTS...
CIN MgErr CINRun( uInt32 *Control    // Control bits telling the CIN which operation to perform
   , uInt16 *SoundSize  // Sound buffer size in bytes. Size of ½ double buffer
   , hWAVESDA wavesDA   // List of waveforms to be transmitted
   , uInt16 *WaveChanged // Bits indicating active D/A and A/D channels
   , uInt16 *DigitalDA  // Unimplemented: values for LAYLA digital outputs
```

For absolute details concerning these structures you are referred to the commented source code (GenMonCIN.c). However I’ll briefly discuss the functional purpose of each of these structures. The Control element points to a structure telling the CIN which operation to perform (start, stop, transmit, measure, or query). The CIN enforces a desired command sequence by passing the Control element through a mask. For example, before any commands can be accepted, the sound system must be started, so any commands other than start do not pass through the mask. Upon a successful start the mask is changed to allow any command EXCEPT a start command which would be redundant. The stop command once again changes the mask to accept only the start command.

The SoundSize parameter, controls the size of the sound buffer allocation when the sound system is being started. If the requested sound buffer size does not match the current ECHO panel setting, then the ECHO panel is brought up automatically so the user can reconcile the difference.

The wavesDA is a list of handles of waveforms to be transmitted on active D/A channels. The respective handles of waveforms to be transmitted are stored *contiguously* in memory. NOTE WELL, the position of a given physical channel waveform is NOT fixed, but instead is a function of what the user has done with the Waveform Definition Control. For example the user may generate waveforms for D/A channels 1,2, and 4. Upon starting the sound system, 3 sound buffers are allocated. Later the user may decide to stop transmitting on channel 2. Waveforms for channels 1 and 4 will be sent to the CIN. There will be no waveform for channel 2. Hence the position of channel 4’s waveform handle moves from 3 to 2.

As there is no *order* information passed to the CIN, it assumes that waveforms to be transmitted and buffers to be filled with measurements are in order of increasing active
channel number. So if it is desired to transmit on D/A channels 2, 4, and 8, the CIN will assume that the TX waveform handle for D/A channel 2 is first, followed by the TX waveform handle for D/A channel 4. There are no such restrictions on GenMon’s Waveform Definition table, so this table must be sorted prior to transmission to ASIO.vi. The same holds true for waveform measurement buffers.
Data from the CIN into LabView

The following elements are passed to the LabView user interface from the CIN.

```c
// OUTPUTS... REMEMBER this code must size the outputs, LabView can't
HWAVESAD wavesAD    // Measured waveforms
uint16  *ServiceBuf  // Status bits indicating service timing
uint32  *Status      // Status bits including late buffers, and overall system status
```

The first element returned to the LabView side of the application, is the wavesAD element. This element is a handle to a list of pointers to waveform measurement arrays. The CIN will create and dimension 1 floating point measurement structure for each active A/D channel indicated in the WaveChanged bits. The measurement structure will be sized according to the current sound buffer size.

The ServiceBuf bits are used to tell the LabView based application that a D/A channel is ready for a new waveform, and/or an A/D channel has acquired a new buffer of data. The high 8 bits are used for A/D channels.

The 32 bits Status indicator contains late bits for the A/D and D/A channels as well as status concerning the operational state of the sound driver and the CIN code. See the ASIO.vi diagram for the details of each bit.
The ASIO compliant Sound Driver thread and the CIN

The ASIO specification for writing an ASIO compliant sound driver can be found in the file ASIO SDK 2.pdf. This is a specification for creating an ASIO compliant sound driver not for using an existing driver. The development problem then became one of simplification, making use of the minimum functionality in support of our application.

In our application the Sound Driver Thread (SDT) is supplied with buffer space, some configuration information (shutting off fancy stuff), a workhorse call-back routine bufSW, and is then started. The user is free to transmit and modify waveforms taking measurements until done with the system OR a change in physical channel configuration is desired. At this point the SDT must be stopped and re-started, as resources such as sound buffers cannot be shuffled while the SDT is active.

Our application is atypical as far as sound generation is concerned because it is not necessary for us to “stream” sound information in an un-broken contiguous time stream. When you play a CD or record music, a time contiguous flow of digital sound is necessary for practical use. In this application, a waveform is defined and then generated on a D/A channel; the waveform typically remains static for many passages of the sound buffer. Likewise it is only necessary to measure an A/D channel on an event driven basis, a very short amount of streaming is necessary, after which changing A/D information is ignored.

When operating in this “static” mode bufSW simply modulo’s a pointer and returns immediately to the ASIO sound driver thread, having done nothing outside of the pointer toggle. The whole architecture of a “sound” system is not required and simply adds un-needed complexity. However short of becoming a Windows XP architecture guru (if this is even possible outside of a M.S. development team) to gain direct access to the Layla hardware (ECHO themselves complain that XP won’t let this happen), this seemed like the most effective approach in terms of development time. A brief attempt to use M.S. Direct X was made prior to going with the ASIO compliant driver. However the M.S. obfuscation philosophy of documentation prevailed, and I happily went with a competitor/alternative seeing as use of 24 bit sound under Direct X was left to the reader’s deduction.

The ASIO interface to the CIN is contained in the file Layla.CPP. Within this file, there are three functional groups of routines. The LAYLA_xxx routines provide high level functionality to the CIN so it doesn’t have to worry about low level details associated with the ASIO driver. The ASIO_xxx routines encapsulate those low level details for the LAYLA_xxx routines. Finally routines without either the LAYLA or ASIO prefix are call backs used by the ASIO sound driver thread.
Complete details of the Layla module is discussed latter in this manual. See the “CIN Internals” section of this manual.
THE ASIO.vi and embedded cin

Overview

Of the application software components, the CIN is really the only component which has to be “built”. The other components must only be configured and/or accessed. The CIN is built under MSVC 6.0 as a .DLL which is then subject to a post-build process in which a LabView supplied utility is called to change the .DLL into an .LSB file. The .LSB file can then be embedded directly into a LabView .VI (in this case ASIO.vi).

CIN entry points, CIN entry pre-defined calling order, use of data structures within the LabView context, passing of CIN parameters and their dynamic sizing are covered in the file Program Files\National Instruments\LabVIEW 6.1\manuals\lvexcode.pdf.

In a nutshell the CIN entry point calling order is as follows...

- **CINLoad**  Called when you load the .LSB into your LabView .vi
- **CINRun**  Called anytime you access the CIN block in your .vi diagram
- **CINUnLoad**  Called when the LabView application terminates
In our application, the CINRun entry point is the only entry point of any significance. LabView will actually generate a CIN.c file for you from within the LabView diagram. In the GenMon application, open the ASIO.vi file and access its diagram. In that diagram you will see the element shown below...

Right click the code interface node element....
to see the following dialog…

This dialog will create a CIN interface for you if you select the “Create .c File” option. The file created will depend on the items you have wired into the CIN diagram node. The names of the items will be very general and not very readable, so, in the GenMon project, new names for CIN interface structures were created manually.

**The key point** is that the variables wired into the CIN on your diagram and the formal parameter list of the CINRun entry point MUST match.

Note that a blank Code Interface Node diagram element looks like this…
Building the CIN: MSVC 6.0 setup

This document section assumes you are familiar with the MSVC 6.0 development environment. This section summarizes the key setup elements in the project which depart from a “typical” or default project setup. The list of files in your project is as follows…
Your CIN is built as a .DLL which is then post processed into an .LSB file. You can automate this post processing by accessing your project settings as follows…

The lvsbutil program supplied by LabView performs the necessary conversion of your .DLL file.
Your structure alignment must be 1 byte, and your run-time library must be set to Multithreaded DLL.
In addition include file search directories for this project are:

```
Options

Editor  |  Tabs  |  Debug  |  Compatibility  |  Build  |  Directories

Platform: Win32
Show directories for: Include files

Directories:
- C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
- C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
- C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
- C:\AUDIOPROJECT\AUDIO
- C:\AUDIOPROJECT\COMMON
- C:\PROGRAM FILES\NATIONAL INSTRUMENTS\LABVIEW 6.1\CINTOOL
```

OK  Cancel
If you rebuild the entire project, your output window should look like this...

```
--------------------Configuration: GenMon - Win32 Release--------------------

COMPILING...
  _COMPLEX.cpp
  appGenMon2.cpp
  appGenMon3.cpp
  CHANNEL.cpp
  ERROR.cpp
  GENMONWIN.C

C:\audio\system\GenMon2\GenMonCIN.c(23) : warning C4273: '_lctype' : inconsistent dll linkage, dlexport assumed.
C:\audio\system\GenMon2\GenMonCIN.c(32) : warning C4273: '__malloc_max' : inconsistent dll linkage, dlexport assumed.

Laying.cpp
matrix.cpp
MEM.cpp
waves3.cpp

LINUXING...
  Creating library Release/GenMon.lib and object Release/GenMon.exp
  Performing Custom Build step on .\Release\GenMon.d11
  LabVIEW resource file, type 'CIN', name 'GenMon.lib', created properly.
  Creating browse info file...

GenMon.d11 - 0 error(s), 2 warning(s)
```
Loading the CIN into LabView

After building the CIN, load the CIN into the LabView environment for use in the GenMon application. Open the ASIO.vi file from within LabView. Activate the diagram associated with this .VI (Ctl-E is a quick way). Find the diagram elements shown below…

Right click the CIN diagram element to see the following menu, and select the “Reload Resource From…” option.

Select either the debug or release version of the CIN, and hit the “Open” button.
Note that the CINLoad entry point will be called at this time. So you should either keep this part of your CIN very simple, or test it stand alone before loading the CIN into LabView. You can create a deadlock condition which could potentially cause the loss of the ASIO.vi file.
Debugging the CIN

There are two basic approaches to debugging the CIN software. Each requires that the sound hardware be powered up and software initialized. The CIN software can be debugged in conjunction with an operating LabView interface, or as a stand alone console application.

Within LabView

The simplest form of debugging within the LabView environment is to delete the file “nuller.log” prior to running the program. Start the GenMon application as described in this manual. Perform the operations you would like to monitor (button sequences of interest), and exit the program.

Open the “Nuller.log” file with any text editor and view the results. Note this is a very large file, having the advantage of recording all program operations in excruciating detail. Of course the disadvantage is filtering through to find the segment of interest in the file.

It is possible to invoke the MSVC++ debugger for the CIN while running in LabView. Place the following line of code in your CINRun entry point…

```
DebugBreak();
```

Recompile and rebuild the debug version of your project. Load the debug CIN into the ASIO module of the LabView application as described in this manual. Run the LabView application. You will get an error message… simply click OK. You can now perform normal debugging operations in the MSVC environment. The downside is of course non-real time performance, and the fact that MSVC will not remember your debugging environment.

Correcting for Latency
CIN INTERNALS

Architecture
The figure below diagrams the relationship between the main modules of the GenMon application. All relationships are fairly classic between the main objects, meaning that all encapsulations are observed. The CHANNEL object however violates the WAVES object encapsulation by reaching directly into memory devoted to waveform generation. This was done for speed purposes. The GenMonCIN (called by LabView), calls only entry points in the APP module. The APP (short for appgenmon2) module makes use of only two objects; CHANNEL and WAVES.
Main Modules

This document section discusses each of the main modules in the GenMon application. A strong effort was made to modularize and encapsulate key functions of the system. The idea was to allow top level source code to be readable and easy to maintain. At the same time, it was desirable to be able to manipulate or configure the software at a high level of abstraction which was related to the functions we needed to perform in the physical system. This level of abstraction would have nothing to do with implementation details of the sound system, the windows environment, or any specific details related to how the solution to the problem was coded.

Ideally, this means that object/module interfaces should be simple (take a few number of simple data types as arguments), and if possible should have most of the needed information transferred at definition/construction.

As you can see from the preceding diagram, the APP module requires only the use of CHANNEL and WAVE objects.

We must be able to start and stop the system, and to perform operations requested by the User, such as measurement, transmission, or status query. The APP module provides these high level services to the LabView interface via the GenMonCIN code which is called by LabView. APP is insulated from LabView details (how user controls are processed as well as the allocation of LabView variables) by the GenMonCIN code.

**APPGENMON**

The APP module contains all high level functions of our GenMon application. Basically there is a supporting APP entry point for each high level function in the main application.

**CHANNEL**

The CANNEL object encapsulates the interface to a Layla sound channel via the ASIO sound driver. It presents only the logical functions necessary to send and receive data. You must instantiate a “master” CHANNEL object which is NOT mapped to any Layla channel, but initializes and manages a common data structure to all future instantiations of the CHANNEL object. It is through the master CHANNEL object that the sound system is started.
The key methods of this object are...

```c
CHANNEL ( void );
void go ( void );
void stop ( void );
```

This constructor builds the master channel object. The master channel object initializes global data structures to all channel instantiations. It also explicitly starts and stops the ASIO sound driver thread. The “go” method must be used to start the sound driver thread after all channels have been constructed. The “stop” method will stop the ASIO sound driver (although this is done automatically once all non-master channel objects have been destructed).

```c
CHANNEL ( WAVE *pwave ,int Channel );
```

Use the above constructor for defining D/A channels.

```c
CHANNEL ( void *pv ,ulong WVsize ,int Channel );
```

Use the above constructor for defining A/D channels.

```c
void lock ( void );
void unlock ( void );
void change ( void );
```

These methods coordinate use of a sound buffer on the application side. The ‘lock’ method prevents the sound driver thread from reading (D/A) or writing (A/D) information into the application side sound buffer (the application side sound buffer, is that ½ of the double buffer currently not in use by the ASIO driver). Incoming sound information (A/D) is simply lost should a write be attempted while a lock is in effect. The ‘unlock’ method frees a lock. The ‘change’ method, lets the sound driver know that a D/A sound buffer has changed. Because D/A buffers spend most of their time un-changed, they are not blindly copied each time the double buffer is cycled, instead the double buffer pointer is simply cycled. However, when a D/A sound buffer has changed (a tone definition has changed, or a null has been calculated), then the ‘change’ method is used to let the sound driver interface (bufSW) know that new data must be copied into the sound buffer.

```c
void convert ( void );
void convert ( WAVEPTR pWV );
void cvrtSeg ( WAVEPTR wpToSegment );
```

The above methods are used to convert an entire sound buffer from binary left justified 24 bit sound in a 32 bit field to a single precision floating point value. Each sample in the sound buffer is converted. Naturally these methods only work on A/D channels. Calling one of these methods on a D/A channel will simply be ignored. The ‘cvrtSeg’ method can be used to convert only a segment of a sound buffer to floating point representation (for short 4hz
resolution FFT’s used in nulling calculations for example). NOTE: always lock the sound buffer you intend to convert both during the conversion and during access to the converted values.

```c
int   done  ( void );
int   busy  ( void );
```

Here are some more methods for controlling access to a sound buffer. The ‘done’ method will return true when the sound driver has processed the application side of the sound buffer. For D/A channels this means that the updated sound buffer has been transferred from the application side to the ASIO sound driver side. For A/D channels this means that ALL segments of the A/D sound buffer have been loaded. The CHANNEL object allows the A/D sound buffer to be an arbitrary multiple of the ASIO sound driver side (in our case this multiple is 4). The ‘busy’ method is identical to !done().

```c
int   seg   ( int whichSegment );
int   NextSeg  ( void );
int   NextSeg  ( int SegmentOffset );
int   CurSeg  ( void );
WAVEPTR SegAddr  ( int whichSegment );
```

These methods coordinate access to the channel’s sound buffer on a segment basis. The ASIO sound driver thread works with a given sound buffer size for all channels. In our application actual size of a measurement may involve 1 to n successive sound buffers (although it is currently 4 for all A/D channels). Each acquired channel sound buffer then becomes a segment in the total measurement. The ASIO sound driver thread via the bufSW call back function continuously fills the measurement buffer which was linked to the CHANNEL object at construction.
LAYLA

The LAYLA object is not a C++ object (... requirement of the ASIO interface), but has a similar structure. The job of the LAYLA object is to provide top level functionality for accessing the LAYLA sound hardware. The LAYLA object is aware of the physical channel limits on the hardware, and some details of the ASIO sound driver. LAYLA “stops the buck” as far as ASIO driver details are concerned. No layers of code above the LAYLA object have to deal with any ASIO driver considerations.

NOTE WELL: the LAYLA ‘object’ is really for convenience of the CHANNEL object. Higher level application code should use a CHANNEL object, never call the LAYLA object directly.

The key “methods” in this object are:

```c
void LAYLA_init ( void );
```
This method initializes the main mapping data structure which associates a physical LAYLA sound channel with a CHANNEL object for managing that channel.

```c
int LAYLA_go ( void );
int LAYLA_stop ( void );
```
Starts and stops sound production/measurement. The ‘_go’ method configures the ASIO driver (sound buffer base size, # sound buffers, format of the sound data, clock source etc.) and then starts the sound driver thread. The ‘_stop’ method stops the sound driver thread. NOTE: The channel object will call LAYLA_stop at the appropriate time (when there are no more active CHANNEL objects). If you attempt to call LAYLA_stop from higher levels of application code, your application will become un-stable if there are still any active CHANNEL objects.

```c
int LAYLA_map ( CHANNEL *pCH );
int LAYLA_update ( int TblIdx , CHANNEL *pCH );
```
These methods are for manipulating the mapping table. This table maps physical LAYLA sound channels to CHANNEL objects. The ‘_map’ method makes a new entry in the table. The ‘_update’ method changes an existing entry in the table.
The ASIO “object” is not a C++ object, merely a collection of C functions whose name begins with “ASIO_”. However conceptually this group of functions serves to isolate higher code layers from the details of the ASIO sound interface. **PLEASE NOTE** that the ASIO set of routines is for use by Layla only, you will likely crash the system if you attempt to call these routines out of context. This is why their function definitions do not appear in Layla.H.

The ASIO interface contains a great deal of power and sophistication which we don’t need in our application, hence the task was one of simplification. The routines below are a result of that simplification.

```c
void ASIO_link ( char *pszDriverName );
void ASIO_config ( void );
void ASIO_buffers ( void );
void ASIO_start ( void );
void ASIO_stop ( void );
void ASIO_err ( ASIOError asioErr );
```

Initial development was complicated by the fact that the only documentation for the ASIO interface was a manual describing development of a native driver for OEM hardware which would be compliant with the ASIO specification. This manual was translated from a foreign language and came with several huge example programs.

Ultimately, the system was reduced to the above routines.

**ASIO_link** is called first. It accesses the system registry using the name ‘ASIO Echo WDM’ which is what ECHO decided to register their ASIO compliant driver as. If the system ever uses different sound hardware which is ASIO compliant, you will have to find out the name under which they registered their driver. Once the registry is accessed, a copy of the driver can be instantiated in the form of a separate thread which is running by the time ASIO_link returns to the caller.

Next, the driver must be configured at runtime. The first step in this configuration is a call to **ASIO_config**. This routine sets system clock source, sample rate and sound buffer size. If any of these parameters cannot be set due to overrides from the ECHO driver panel, an error message is displayed and the ECHO driver panel is started; the user is asked to set clock source, sample rate access, and sound buffer size to be consistent with the application’s needs.

The second step in the runtime configuration of the driver involves definition of the number of sound buffers which will be attached to D/A and A/D channels on the sound equipment. This is performed by the **ASIO_buffers** routine. The **ASIO_buffers** routine also has the job of handing the list of call-back functions to the running sound driver thread. It is these call back
routines which form the core of the runtime interface between our application and the separate sound driver thread.

Once the driver has been runtime configured, it must be “started” which tells the driver to start moving sound data to/from the sound hardware. This is performed by the ASIO_start routine. Even though the driver has been “started” its runtime configuration is not yet complete. The final step in runtime configuration is a call to the asioMsg call back routine describing the instantiating application sophistication. In our application, I chose the minimum sophistication. Finally, the sound driver thread will call the bufSW callback for the first time giving it a chance to initialize double sound buffer management.

Following this sequence, the sound system will actually be running making calls to bufSW when it either has 1 or more new A/D sound buffers acquired or is ready for 1 or more fresh D/A buffers. When called, bufSW will copy new sound information from the application side of the A/D sound double buffers and load the application side of D/A sound buffers with new waveforms to be output to the sound hardware. Upon returning to the calling sound driver thread, the application side of the sound double buffer will become the sound driver side, at which point the sound driver is free to read or write into the sound buffer.

**WAVE**

The WAVE object is used for D/A channels and basically manages the conversion from a floating point waveform to binary sound buffer format. Note that a WAVE object does not actually hold a sound buffer, but contains a chunk of memory which is the same size and whose data is in the same format as an actual sound buffer. The WAVE object lets us manipulate signals at our leisure without having to worry about the time constraints associated with a sound buffer. When a WAVE has been calculated then this key chunk of memory can be copied directly into a sound buffer for ultimate consumption by the ASIO driver. This copy is managed by bufSW using a set of global tables filled out by CHANNEL.
KNOWN BUGS

Crash after N Activate/DeActivate Cycles

Channel doesn’t “shut-off” when all components are disabled

Measured phase does not match transmitted phase