Chapter 1

OMAP-L138 Development System

- OMAP-L138 processor
- Code Composer Studio™ IDE version 4
- Use of the OMAP-L138 eXperimenter
- Programming examples

This chapter gives an overview of the OMAP-L138 processor and Logic PD’s Zoom OMAP-L138 eXperimenter development system. It describes how to install and start using version 4 of Texas Instruments (TI) Code Composer Studio integrated development environment (IDE). Two example programs that demonstrate hardware and software features of the eXperimenter board and of the Code Composer Studio IDE are presented. It is recommended strongly that you review these examples before proceeding to subsequent chapters.

1.1 INTRODUCTION

The Logic PD Zoom OMAP-L138 eXperimenter kit is a low-cost development platform for the Texas Instruments OMAP-L138 processor. This device is a dual-core system on a chip comprising an ARM926EJ-S general-purpose processor (GPP) and a TMS320C6748 digital signal processor. In addition, a number of peripherals and interfaces are built into the OMAP-L138 as shown in Figure 1.1.

The eXperimenter makes a significant number of the OMAP-L138 interfaces available to the user, as shown in Figure 1.2. This book is concerned with the development of real-time digital signal processing (DSP) applications and therefore makes use of the DSP (C6748) side of the device and of the TLC320AIC3106 (AIC3106) analog interface circuit (codec) connected to the OMAP-L138’s

Chapter 1  OMAP-L138 Development System

Figure 1.1 Functional block diagram of OMAP-L138 processor. (courtesy of Texas Instruments)

Figure 1.2 Logic PD Zoom OMAP-L138 eXperimenter baseboard (courtesy of Logic PD).
multichannel audio serial port (McASP). The ARM side of the device is not used by
the examples in this book. Connection to a host PC running the Code Composer
Studio IDE is via XDS100v1 JTAG emulation built in to the eXperimenter. The Code
Composer Studio IDE enables software written in C or assembly language to be
compiled and/or assembled, linked, and downloaded to run on the C6748. Details of
the OMAP-L138, TMS320C6748, TLC320AIC3106, eXperimenter, and Code
Composer Studio IDE can be found in their associated datasheets [1–5] and in the
TI wiki [6]. The purpose of this chapter is to introduce the installation and use of the
eXperimenter for hands-on DSP experiments.

1.1.1 Digital Signal Processors

A digital signal processor is a specialized form of microprocessor. Its architecture and
instruction set are optimized for real-time digital signal processing. Typical optimiza-
tions include hardware multiply accumulate (MAC) provision, hardware circular and
bit-reversed addressing capabilities (for efficient implementation of data buffers and
fast Fourier transform (FFT) computation), and Harvard architecture (independent
program and data memory systems). In many respects, digital signal processors
resemble microcontrollers. Typically, they provide single-chip computer solutions
integrating on-board volatile and nonvolatile memory and a range of peripheral
interfaces, and have a small footprint, making them ideal for embedded applications.
In addition, digital signal processors tend to have low power consumption require-
ments. This attribute has been extremely important in establishing the use of digital
signal processors in cellular handsets. However, the distinctions between digital
signal processors and other more general-purpose microprocessors are blurred. No
strict definition of a digital signal processor exists and semiconductor manufacturers
apply the term to products exhibiting some, but not necessarily all, of the above
characteristics as they see fit.

Digital signal processors are used for a wide range of applications, from
communications and control to speech and image processing. They are found in
cellular phones, disk drives, radios, printers, MP3 players, HDTV, digital cameras,
and so on. Specialized (particularly in terms of their on-board peripherals) DSPs are
used in electric motor drives and in a range of associated automotive and industrial
applications. Overall, digital signal processors are concerned primarily with real-time
signal processing. Real-time processing means that the processing must keep pace
with some external event, whereas non real-time processing has no such timing
constraint. The external event to keep pace with is usually the analog input. While
analog-based systems with discrete electronic components including resistors and
capacitors are sensitive to temperature changes, DSP-based systems are less affected
by environmental conditions such as temperature. Digital signal processors embody
the major advantages of microprocessors. They are easy to use, flexible, and
economical.

Texas Instruments OMAP-L138 device combines a C6748 DSP with an
ARM926EJ-S general-purpose processor to produce a dual-core solution for
handheld and other embedded applications. ARM926EJ-S provides the benefits of a 32-bit RISC processor, well suited to implementing user interfaces and running operating systems.

C6748 is a member of the Texas Instruments TMS320C6000™ DSP family of digital signal processors. Its architecture is very well suited to numerically intensive calculations and it is one of TI’s most powerful digital signal processors. More specifically, as a member of the C674x family, it combines C64x™ DSP fixed-point and C67x™ DSP floating-point architectures in one core, as illustrated in Figure 1.3.

1.2 HARDWARE AND SOFTWARE TOOLS

Most of the examples presented in this book involve the development and testing of short programs intended to demonstrate fundamental DSP concepts in a laboratory setting. To perform the experiments described in the book, a number of hardware and software tools are required.

1) A Logic PD Zoom OMAP-L138 eXperimenter kit This package includes the following:
   (a) Three separate circuit boards, a baseboard, a 4.3″ LCD, and an OMAP-L138 SOM-M1. OMAP-L138 SOM-M1 must be connected to the baseboard, as described in the instructions included in the eXperimenter kit. The LCD is not used in the experiments in this book and therefore need not be connected to the baseboard.
   (b) A universal serial bus (USB) cable that connects the eXperimenter board to a host PC.
   (c) A 5 V universal power supply for the eXperimenter board.

2) A host PC This is used to run the Code Composer Studio IDE. The eXperimenter board is connected to a USB port on the host PC.

(4) **Board support library** Board-level support routines specific to eXperimenter are not supplied with the kit, but may be downloaded from the Logic PD website at [http://www.logicpd.com/product-support](http://www.logicpd.com/product-support). Access to the board support library BSL file 1017292A_OMAP-L138_GEL_BSL_Files_v2.3.zip requires a login and product registration.

(5) **TMS320C674x DSP library** A number of example programs make use of optimized DSP routines from this library. It can be downloaded from the Texas Instruments wiki at [http://processors.wiki.ti.com/index.php/C674x_DSPLIB](http://processors.wiki.ti.com/index.php/C674x_DSPLIB) or from the Texas Instruments website at [http://focus.ti.com/docs/toolsw/folders/print/sprc900.html](http://focus.ti.com/docs/toolsw/folders/print/sprc900.html).

(6) **DSP/BIOS™ software Kernel Foundation Platform Support Package (PSP)** This is used for examples in Chapter 7. Details of how to download file BIOSPSP_01_30_00_06_Setup.exe, which is part of the OMAP-L138 and C6748 software development kits (SDKs) may be found at [http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software](http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software).

(7) **An oscilloscope, spectrum analyzer, signal generator, headphones, microphone, and loudspeakers** The experiments presented in subsequent chapters of this book are intended to demonstrate digital signal processing concepts in real-time, using audio frequency analog input and output signals. In order to appreciate these concepts and to get the greatest benefit from the experiments, some forms of signal source and sink are required. As a bare minimum, an audio source with line level output and either headphones or loudspeakers are required. Greater benefit will be accrued if a signal generator is used to generate sinusoidal, and other, test signals and if an oscilloscope and spectrum analyzer are used to display, measure, and analyze input and output signals. Many modern digital oscilloscopes incorporate FFT functions, allowing the frequency content of signals to be displayed. Alternatively, a number of software packages that use a PC equipped with a sound card to implement virtual instruments are available. In this book, Goldwave is used, which may be downloaded from [www.goldwave.com](http://www.goldwave.com).

1.2.1 Zoom OMAP-L138 eXperimenter Board

The eXperimenter board is a powerful, yet inexpensive, development system with the necessary hardware and software support tools for real-time signal processing [4]. From the point of view of the example programs in this book, it is a complete DSP system. The board, which measures approximately 5 × 7 inches, includes a 375 MHz OMAP-L138 processor and a 16-bit stereo codec TLV320AIC3106 (AIC3106) for analog input and output. Numerous other interfaces are provided by the eXperimenter but are not used by the example programs in this book.

The onboard codec AIC3106 [3] uses sigma–delta technology that provides analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) functions. It uses a 24.576 MHz system clock and its sampling rate can be selected from a range of alternative settings from 8 to 48 kHz.

The eXperimenter boards each include 128 MB of synchronous dynamic RAM (mDDR SDRAM) and 8 MB of NOR flash memory. Two 3.5 mm jack socket connectors on the boards provide analog input and output: LINE IN for line level input and LINE OUT for line level output. The status of eight user DIP switches on the board can be read from within a program running on the processor and provide a simple means of user interaction. The states of two LEDs on the board can be controlled from within a program running on the processor.

1.2.2 C6748 Processor

The DSP core in the OMAP-L138 device (L138) is a C6748 DSP based on Texas Instruments very long instruction word (VLIW) architecture and is very well suited to numerically intensive algorithms. The internal program memory is structured so that a total of eight instructions can be fetched every cycle. With a clock rate of 375 MHz, the C6748 is capable of fetching eight 32-bit instructions every $\frac{1}{375\text{ MHz}}$ or 2.67 ns. As part of the C674x family, it incorporates both floating-point and fixed-point architectures in one core.

Features of the C6748 include 326 kB of internal memory (32 kB of L1P program RAM/cache, 32 kB of L1D data RAM/cache, and 256 kB of L2 RAM/cache), eight functional or execution units composed of six ALUs and two multiplier units, an external memory interface addressing 256 MB of 16-bit mDDR SDRAM, and 64 32-bit general-purpose registers. In addition, the OMAP-L138 features 128 kB of on-chip RAM shared by its C6748 and ARM9 processor cores [1].

1.2.3 Code Composer Studio IDE

Code Composer Studio software provides an IDE for real-time digital signal processing applications based on the C programming language. It incorporates a C compiler, an assembler, and a linker. It has graphical capabilities and supports real-time debugging. Version 4 of the Code Composer Studio IDE is based on the
open-source Eclipse framework [7], widely used in embedded systems development.

Code Composer Studio software is project based. A Code Composer Studio software project comprises all the files (or links to all the files) required in order to generate an executable file. In addition, a Code Composer Studio software project contains information about exactly how files are to be used in order to generate an executable file. Compiler/linker options can be specified.

A number of debugging features are available in Code Composer Studio software, including setting breakpoints and watching variables, viewing memory, registers, and mixed C and assembly code, graphing results, and monitoring execution time.

Communication between the Code Composer Studio IDE and the eXperimenter is via a USB connection and the XDS100v1 JTAG emulation [8] built into the board. Compared to previous versions, Code Composer Studio software version 4 separates code development and debugging activities through the use of perspectives. Perspectives are sets of windows, views, and menus and as a default Code Composer Studio software provides a default C/C++ perspective, including, among others, Project View, Editor, and Outline windows, and a default Debug perspective, including Debug, Disassembly, and Console views. Users may customize these perspectives or create new ones. The View menu gives an idea of the many other windows available.

### 1.2.4 Installation of Code Composer Studio Software Version 4 and Support Files

The example programs described in this book are intended to be used with Code Composer Studio software version 4 and were tested using version 4.2.1. Code Composer Studio software is supplied on a DVD as part of the eXperimenter kit or alternatively may be downloaded from the Texas Instruments wiki at [http://processors.wiki.ti.com/index.php/Download_CCS](http://processors.wiki.ti.com/index.php/Download_CCS). Installation instructions for Code Composer Studio software are included on the DVD. A typical (default) location for the files is \c:\Program Files\Texas Instruments\ccsv4, but this is not mandatory. The default location for the Code Composer Studio IDE was used during preparation of the example programs in this book. Once installed, an icon with the label Code Composer Studio v4 should appear on the desktop, as shown in Figure 1.4.

The example programs make use of the Logic PD BSL which may be downloaded from [http://www.logicpd.com/product-support](http://www.logicpd.com/product-support). During preparation of this book and testing of the example programs, the BSL was installed at \c:\omap1138.

Some example programs make use of the optimized c674x DSPLIB library of digital signal processing routines [9] that may be downloaded from [http://processors.wiki.ti.com/index.php/C674x_DSPLIB](http://processors.wiki.ti.com/index.php/C674x_DSPLIB). During preparation of this book and testing of the example programs, it was installed at \c:\C674x_dsplib_1_03_00_01.
Example programs in Chapter 7 make use of the DSP/BIOS software kernel foundation PSP. During testing of the example programs, the SDK that includes the PSP was installed according to the instructions at http://processors.wiki.ti.com/index.php/GSG_C6748:_Installing_the_SDK_Software, resulting in installation of PSP at \texttt{c:\C6748\_dsp\_1\_00\_00\_11\pspdrivers\_01\_30\_01}.

The files accompanying this book should be copied to \texttt{c:\eXperimenter} so that, for example, files relating to this chapter may be found in folder \texttt{c:\eXperimenter\L138\_chapter1}.

Alternative locations for the various software tools described are possible, but corresponding changes to some of the \textit{Build Properties} within Code Composer Studio software that are shown in this book would have to be made. The path names used and detailed instructions given in this book assume that the software tools have been installed as just described.

### 1.3 INITIAL TEST OF THE EXPERIMENTER USING A PROGRAM SUPPLIED WITH THIS BOOK

Follow these instructions in order to quickly test the correct installation of the software tools and the eXperimenter board:

1. Connect the eXperimenter board to the host PC using the USB cable provided. There are two mini-USB sockets on the eXperimenter. The socket used for connection to the host PC is \textit{Emulation USB port} (J21), located adjacent to the 9-way D-type RS232 \textit{serial debug port} (Figure 1.2).

2. Make sure that all the \textit{Boot mode} DIP switches (S7) are OFF, except for DIP switches #5 and #8 that should be ON (Figure 1.2). This sets the appropriate \textit{Boot Mode} (EMU Debug) for the use made of the eXperimenter in this book.

3. Connect a line level audio source, for example, the output from a PC sound card, to the LINE IN socket on the eXperimenter. Make sure that the output level from the source is sufficiently low that it will not damage the input circuits of the AIC3106 codec.

4. Connect either headphones or loudspeakers to the LINE OUT socket on the eXperimenter.

5. Connect the power supply provided to the eXperimenter board.
(6) Switch on the eXperimenter using the Power Switch. The Power On LED (D5) should light.

(7) Launch Code Composer Studio software by double-clicking on the Code Composer Studio v4 icon on the desktop. You should see a splash screen as shown in Figure 1.5 and then a pop-up window similar to that shown in Figure 1.6.

(8) Enter \c:\eXperimenter\L138_chapter1 as a workspace, as shown in Figure 1.6 and click OK. Next, you should see a welcome screen as shown in Figure 1.7. Click on the Start using CCS icon in the top right-hand corner and Code Composer Studio should start in the C/C++ perspective, but show no projects in the Project View window.

The Code Composer Studio software version 4 debugger makes use of a Target Configuration file containing details of the hardware system being debugged. For the example programs in this book, target configuration file L138_experimenter.ccxml is provided. It is located in folder \c:\eXperimenter\L138_support.

(6) Switch on the eXperimenter using the Power Switch. The Power On LED (D5) should light.

(7) Launch Code Composer Studio software by double-clicking on the Code Composer Studio v4 icon on the desktop. You should see a splash screen as shown in Figure 1.5 and then a pop-up window similar to that shown in Figure 1.6.

(8) Enter \c:\eXperimenter\L138_chapter1 as a workspace, as shown in Figure 1.6 and click OK. Next, you should see a welcome screen as shown in Figure 1.7. Click on the Start using CCS icon in the top right-hand corner and Code Composer Studio should start in the C/C++ perspective, but show no projects in the Project View window.

The Code Composer Studio software version 4 debugger makes use of a Target Configuration file containing details of the hardware system being debugged. For the example programs in this book, target configuration file L138_experimenter.ccxml is provided. It is located in folder \c:\eXperimenter\L138_support.

Figure 1.5 Code Composer Studio software version 4 splash screen.

Figure 1.6 Code Composer Studio software version 4 pop-up window.
(9) Copy the target configuration file L138_eXperimenter.ccxml from c:\eXperimenter\L138_support to the default location used by the Code Composer Studio IDE for target configuration files, that is, c:\Documents and Settings\YOUR_ID\user\CCSTargetConfigurations if you are using Window XP, or c:\User\Your_ID\user\CCSTargetConfigurations if you are using Windows 7.

(10) Copy file C6748.gel from folder c:\eXperimenter\L138_support to c:\Documents and Settings\YOUR_ID\user \CCSTargetConfigurations if you are using Window XP, or c:\User\Your_ID\user\CCSTargetConfigurations if you are using Windows 7. This general extension language (GEL) script is run every time you Connect to Target in the debugger and carries out a number of important initialization procedures on the eXperimenter.

(11) In the C/C++ perspective, select View > Target Configurations, right-click on L138_eXperimenter.ccxml, under User Defined and select Set as Default. The Project View window should then appear as shown in Figure 1.8.

(12) Launch the debugger by selecting Target > Launch TI Debugger. This should cause Code Composer Studio to switch from the C/C++ perspective to the Debug perspective, including a Debug window as shown in Figure 1.9. If Code Composer Studio software does not automatically switch to the Debug perspective, you can do so using the Debug button in the top right-hand corner of the Code Composer Studio IDE window, as shown in Figure 1.10.
(13) Select Target > Connect Target. At this point, the GEL script C6748.gel specified by the target configuration file L138_eXperimenter.ccxml will be run and some diagnostic messages will appear in the console. Following this, the Connect icon in the toolbar should change from grayed out to highlighted, as shown in Figure 1.11. During this step, the error message window shown in Figure 1.12 may appear. If this happens, simply ignore it. It should disappear after a few seconds.

Figure 1.8 Target Configurations view following selection of default target configuration file L138_eXperimenter.ccxml.

Figure 1.9 Debug view as it should appear in the Debug perspective after launching the debugger (Connect icon grayed out).
Figure 1.10  Switch between Debug and C/C++ perspectives by clicking these buttons.

Figure 1.11  Debug view as it should appear in the Debug perspective after connecting to target (Connect icon highlighted).

Figure 1.12  Error message window.
1.3 Initial Test of the Experimenter Using a Program Supplied with this Book

(14) Select Target > Load program and Browse to find the executable file L138_loop_poll.out in folder c:\eXperimenter\L138\chapter2\L138_loop_poll\Debug, as shown in Figure 1.13.

(15) Click OK and the Debug view should change to that shown in Figure 1.14, indicating that the program has been loaded.

(16) Execute the program by selecting Target > Run or by clicking on the green Run toolbar button in the Debug window.

Program L138_loop_poll simply reads input samples from the ADC and writes them to the DAC so that an audio signal input to LINE IN, for example, from a PC sound card should be output at LINE OUT.

The program may be halted by selecting Target > Halt or by clicking the yellow Halt toolbar button in the Debug view. The program may be run again after clicking the Restart toolbar button. If this is unsuccessful, select Target > Reset > System Reset and then Target > Reload Program before clicking Run again.

Figure 1.13 Browse to select program L138_loop_poll.out.

Figure 1.14 Debug view as it should appear in the Debug perspective after loading executable file L138_loop_poll.out.
1.4 PROGRAMMING EXAMPLES TO TEST THE EXPERIMENTER

Two programming examples are presented in this chapter to illustrate some of the features of the Code Composer Studio software and the eXperimenter board. The aim of these examples is to enable the reader to become familiar with both the software and hardware tools that will be used throughout this book. It is strongly recommended that you complete these two examples before proceeding to subsequent chapters.

EXAMPLE 1.1: Sine Generation Using 48 Points with Output Recorded in a Buffer for Plotting Using Code Composer Studio Software and MATLAB (L138_sine48_buf_intr)

This example concerns generation of a sinusoidal analog output waveform using a table lookup method. More importantly, it illustrates some of the features of Code Composer Studio software for editing source files, building a project, accessing the code generation tools, and running a program on the OMAP-L138 processor.

Program description

The operation of program L138_sine48_buf_intr.c is as follows. An array, sine_table, of 48 16-bit signed integers is declared and initialized to contain samples of exactly one cycle of a sinusoid. The value of sine_table[i] is equal to

\[10000 \sin \left( \frac{2\pi i}{48} \right), \quad \text{for } i = 0, 1, 2, 3, \ldots, 47.\]

Within function main(), a call to function L138_initialise_intr() initializes the eXperimenter board. The AIC3106 codec is configured to operate at a sampling rate of 48 kHz with 0 dB ADC gain and 0 dB DAC attenuation and the McASP0 interface, used to communicate between the OMAP-L138 processor and the AIC3106 codec, is configured so that an interrupt will occur at each sampling instant. Function L138_initialise_intr() is defined in the file L138_aic3106_init.c, which can be found in the folder eXperimenter\L138_support. Its actions are described in more detail in Chapter 2. The program statement while(1); within the function main() creates an infinite loop. Following initialization, the program does nothing except wait for interrupts.

Each time an interrupt occurs, that is, at each sampling instant, function interrupt4() is called. Within that function, the next sample value read from the array sine_table is output via the AIC3106 codec using function output_left_sample(), and the variable sine_ptr, used as an index into the array, is incremented. When the value of sine_ptr exceeds the allowable range for the array sine_table, that is, it exceeds the value LOOPLENGTH-1, it is reset to zero. The BUFLENGTH sample values most recently output via the codec are stored in the array buffer using variable buf_ptr as an index.

A sinusoidal analog output waveform is generated only on the left channel of the AIC3106 codec and via the LINE OUT socket. One cycle of the sinusoidal analog output waveform
corresponds to 48 output samples and hence the frequency of the sinusoidal analog output waveform is equal to the codec sampling rate (48 kHz) divided by 48, that is 1 kHz.

The C source file L138_sine48_buf_intr.c listed in Figure 1.15 is provided in the folder c:\eXperimenter\L138_chapter1. Before it can be compiled, assembled, linked, downloaded, and executed on the OMAP-L138, a project must be created around it. The following steps assume that the eXperimenter has been connected to the host PC and powered up and that the Code Composer Studio IDE has been launched as described in the previous section.

Create a project

In the C/C++ perspective (switch to the C/C++ perspective by clicking the C/C++ button in the top right corner of Code Composer Studio software), create a new project in the

```c
// L138_sine48_buf_intr.c
//
#include "L138_aic3106_init.h"
#define LOOPLENGTH 48
#define BUFLENGTH 256

int16_t sine_table[LOOPLENGTH] =
{0, 1305, 2588, 3827, 5000, 6088, 7071, 7934,
  8660, 9239, 9659, 9914, 10000, 9914, 9659, 9239,
  8660, 7934, 7071, 6088, 5000, 3827, 2588, 1305,
  0, -1305, -2588, -3827, -5000, -6088, -7071, -7934,
  -8660, -9239, -9659, -9914, -10000, -9914, -9659, -9239,
  -8660, -7934, -7071, -6088, -5000, -3827, -2588, -1305};

int16_t sine_ptr = 0; // pointer into lookup table
int32_t buffer[BUFLENGTH];
int16_t buf_ptr = 0;

interrupt void interrupt4(void) // interrupt service routine
{
    int16_t sample;

    sample = sine_table[sine_ptr];       // read sample from table
    output_left_sample(sample);          // output sample
    sine_ptr = (sine_ptr+1)%LOOPLENGTH;  // increment table index
    buffer[buf_ptr] = (int32_t)(sample); // store sample in buffer
    buf_ptr = (buf_ptr+1)%BUFLENGTH;     // increment buffer index

    return;
}

int main(void)
{
    L138_initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);
    while(1);
}

Figure 1.15 Listing of program L138_sine48_buf_intr.c.
```
workspace L138\chapter1 by selecting File > New > CCS Project. You should see a pop-up window similar to that shown in Figure 1.16.

Enter L138\sine48\buf\intr as a project name. This will create a project folder in the c:sources\L138\chapter1 workspace. Click Next and the window shown in Figure 1.17 should appear.

Select C6000 as Project Type and leave Debug and Release configuration options checked. Click Next.

Do not reference any other projects. In the Additional Project Settings window (Figure 1.18), click Next. Figure 1.19 shows the Project settings you should use: Output type Executable, Device Variant Generic C674x Device, Device Endianness Little, Code Generation Tools TI v7.0.3, and Runtime Support Library rts6740.lib.

Click Finish. At this point, the C/C++ Project View in Code Composer Studio software should appear, as shown in Figure 1.20. A new project has been created but it does not contain any source files. Add source file L138\sine48\buf\intr\c to the new project, using Windows Explorer to copy the source file L138\sine48\buf\intr\c from folder c:sources\L138\chapter1 to the newly created project folder c:sources\L138\chapter1\L138\sine48\buf\intr. After a few seconds, the file should appear in the C/C++ Project View window, as shown in Figure 1.21.

**Link support files to project**

A number of support files are required in order to build an executable version of the program and these are provided in folder c:sources\L138\support. Links to

![Figure 1.16 New CCS Project window.](image)
1.4 Programming Examples to Test the Experimenter

Figure 1.17 Select Project window.

Figure 1.18 Additional Project Settings window.
these files must be made from the L138_sine48_buf_intr project. Select Project > Link Files to Active Project and Open the following files.

1. L138_aic3016_init.c
2. L138_aic3016_init.h
3. linker_dsp.cmd
4. vectors_intr.asm

These should appear in the C/C++ Project View window as shown in Figure 1.22.
Set build options

Right-click on the project name L138_sine48_buf_intr in the C/C++ Project View window and select Build Properties. A new Properties for L138_sine48_buf_intr (Filtered) window should appear. In the Tool Settings tab under Configuration Settings, click C6000 Compiler Include Options. The compiler needs to know the locations of the BSL and of the other support files linked to the project. Click the Add button next to Add dir to #include search path (--include_path, -I) and add the paths to the required folders. Assuming that the BSL has been installed at c:\omap138 and the files accompanying this book copied to
c:\eXperimenter, the appropriate paths will be c:\omapl138\bsl\inc and c:\eXperimenter\L138_support, respectively. Enter these directly or browse for them by clicking the File System button in the Add directory path pop-up window. After adding these paths, the Configuration Settings in the Build Properties window should appear as shown in Figure 1.23.

Next, click C6000 Linker > File Search Path and Add the library c:\omapl138\bsl\lib\evmomap138_bsl.lib. The Build Properties window should then appear as shown in Figure 1.24.

Click Apply and then OK to close the Build Properties window.
Building loading and running the program

If you have not specified a default target configuration file as part of the previous example, copy files C6748.gel and L138_eXperimenter.cxml from folder c:\L138_support to folder c:\Documents and Settings\YOUR_ID\user\CCSTargetConfigurations. Select View > Target Configurations, right-click on L138_eXperimenter.cxml, and select Set as Default.

Select Target > Debug Active Project or click the Debug Active Project toolbar button. This will build the active project, launch the debugger, and load the program. Once the project has been built and the program downloaded onto the eXperimenter, the Code Composer Studio software should have switched to the Debug perspective and a Debug window similar to that shown in Figure 1.25 should have appeared. If you still see the C/C++ perspective, then click the Debug button in the top right corner of the Code Composer Studio IDE window.

To run the program, select Target > Run or click the Run toolbar button. The program should generate a 1 kHz tone via the left channel of LINE OUT. Verify this using headphones or loudspeakers connected to LINE OUT.

The program may be halted by selecting Target > Halt or by clicking the Halt toolbar button. The program may be run again after clicking the Restart toolbar button. If this is unsuccessful, select Target > Reset > System Reset and Target > Reload Program before clicking Run again.

Editing source files within Code Composer Studio software

Edit the source file L138_sine48_buf_intr.c in an Editor window. If you have carried out the steps described above, then there should already be a tabbed Editor window open in the Debug perspective containing file L138_sine48_buf_intr.c there. If not, then select File > Open File and open file c:\eXperimenter\L138_chapter1\L138_sine48_buf_intr\L138_sine48_buf_intr
Changing the frequency of the generated sinusoid

There are several different means by which the frequency of the sinusoid generated by program L138_sine48_buf_intr.c can be altered.

1. Change the AIC3106 codec sampling frequency from 48 to 24 kHz by changing the line that reads

   initialise_intr(FS_48000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);

   to read

   initialise_intr(FS_24000_HZ, ADC_GAIN_0DB, DAC_ATTEN_0DB);

Select Target > Reset > CPU Reset and then Project > Build Active Project. Run the program and verify that the frequency of the sinusoid generated is 500 Hz. The different sampling frequencies supported by the AIC3106 codec are 8, 9.6, 11.025, 12, 16, 19.2, 22.05, 24, 32, 44.1, and 48 kHz.

2. Change the number of samples stored in the lookup table to 24. By changing the statements

   #define LOOPLENGTH 48
   int16_t sinetable[LOOPLENGTH] =
   {0, 1305, 2588, 3827, 5000, 6088, 7071, 7934, 8660, 9239, 9659, 9914, 10000, 9914, 9659, 9239, 8660, 7934, 7071, 6088, 5000, 3827, 2588, 1305, 0, -1305, -2588, -3827, -5000, -6088, -7071, -7934, -8660, -9239, -9659, -9914, -10000, -9914, -9659, -9239, -8660, -7934, -7071, -6088, -5000, -3827, -2588, -1305};

   to

   #define LOOPLENGTH 24
   int16_t sinetable[LOOPLENGTH] =
   {0, 2588, 5000, 7071, 8660, 9659, 10000, 9659, 8660, 7071, 5000, 2588, 0, -2588, -5000, -7071, -8660, -9659, -10000, -9659, -8660, -7071, -5000, -2588};

Select Target > Reset > System Reset and then Project > Build Active Project. Run the program and verify that the frequency of the sinusoid generated is 2 kHz (assuming a 48 kHz sampling frequency).

Plotting memory contents using the Code Composer Studio IDE

In addition to generating an analog voltage waveform, program L138_sine48_buf_intr.c uses array buffer to store the BUFLENGTH most recent output sample
values. Once the program has been halted, Code Composer Studio software may be used to plot data in both time and frequency domains.

Halt program execution by clicking the yellow **Halt** toolbar button in the **Debug** window. In order to plot a graph of the contents of the array **buffer**, select **Tools > Graph > Single Time** and set the **Graph Properties**, as shown in Figure 1.26. You should then see a graph of 256 buffered output samples, as shown in Figure 1.27. This assumes that the original version of the program has been run.

A frequency domain representation of the signal generated can be produced by selecting **Tools > Graph > FFT Magnitude** and setting the **Graph Properties**, as shown in Figure 1.28. Because the 256 buffered output samples do not represent an integer number of cycles of the 1 kHz sinusoid, spectral leakage is evident in Figure 1.29. Nonetheless, the 1 kHz tone is represented clearly in the resulting graph.

![Graph Properties for plotting contents of array buffer.](image)

**Figure 1.26** Graph Properties for plotting contents of array buffer.

![Contents of array buffer following execution of program L138_sine48_buf_intr.c.](image)

**Figure 1.27** Contents of array buffer following execution of program L138_sine48_buf_intr.c.
Plotting memory contents using MATLAB

An alternative and more flexible method of plotting the buffered output samples is to save the buffer contents in a file and use MATLAB as a graph plotting tool.

In order to save the buffered samples to a file, select View > Memory and enter buffer as the Address Text and 32 Bit Signed Integer as the Data Type, as shown in Figure 1.30.

Click the Save button in the Memory view, choose a filename, and fill in the details, as shown in Figure 1.31.

Although the values stored in array sine_table and passed to function output_left_sample() are 16-bit signed integers, array buffer is declared as type

![Graph Properties](image)

**Figure 1.28** Graph properties used to plot magnitude of FFT of contents of array buffer.

![Graph](image)

**Figure 1.29** Magnitude of FFT of contents of array buffer following execution of program L138_sine48_buf_intr.c.
int32_t (32-bit signed integer) in program L138_sine48_buf_intr.c to allow for the fact that there is no 16-bit Signed Integer data type option in the Save Memory window, as shown in Figure 1.31.

The TI Data Format (.dat) file saved can be opened and its contents plotted using the MATLAB function L138_logfft().

Type L138_logfft at the MATLAB command line and enter the filename, sampling frequency, frequency scale type, and the number of sample values to be plotted when prompted. Figures 1.32 and 1.33 show the resultant plots. A listing of L138_logfft.m is given in Chapter 2.
EXAMPLE 1.2: Dot Product of Two Arrays (L138_dotp4)

This example illustrates the use of breakpoints and Watch windows within the Code Composer Studio IDE. In addition, it illustrates how to use the Code Composer Studio software Profile Clock in order to estimate the time taken to execute a section of code.
The C source file L138_dotp4.c, listed in Figure 1.34, calculates the dot product of two arrays of integer values. The first array is initialized using the four values 1, 2, 3, and 4, and the second array using the four values 0, 2, 4, and 6. The dot product is equal to 
\[ (1 \times 0) + (2 \times 2) + (3 \times 4) + (4 \times 6) = 40. \]

The program can readily be modified to handle larger arrays. No real-time input or output is used in this example, and so the real-time support files L138_aic3106_init.c, L138_aic3106_init.h, and vectors_intr.asm are not needed.

In the C/C++ perspective, click Project > Import Existing CCS/CCE Eclipse Project and Browse for search-directory c:\eXperimenter\L138_chapter1\L138_dotp4. The Import CCS Eclipse Projects window should appear, as shown in Figure 1.35. Make sure that Discovered Project L138_dotp4 is checked.

Click Finish and the L138_dotp4 project should appear in the C/C++ Project View window, along with the L138_sine48_buf_intr project. The L138_dotp4 project should be Active. If it is not, right-click on the project name L138_dotp4 in the C/C++ Project View window and select Set as Active Project (Figure 1.36).

Select Target > Debug Active Project or click the Debug Active Project toolbar button. The Debug window in the Debug perspective should then appear, as shown in Figure 1.37.

Click on the Run toolbar button in the Debug window and you should see the text
\[ result = 40 \text{ (decimal)} \]
appear in the console. The program can be run again, without reloading, after clicking the Restart toolbar button in the Debug window.
Use of breakpoints and watch window

Open a Watch window in the Debug perspective by selecting View > Watch. Then, enter the variable names `sum` and `i` in the Watch window, as shown in Figure 1.38.

At this point, the Watch window will report that it cannot find the identifiers `sum` and `i`. This is because they are declared locally in function `dotp()`. When the program is halted within that function, their values will be displayed in the Watch window. Set a breakpoint at line

![Image](image-url)
sum += a[i] * b[i];

by double-clicking in the left margin beside that line in the Editor window in which a listing of the file L138_dotp4.c appears. A symbol should appear in the left margin, as shown in Figure 1.39.

Restart and Run the program. Execution should halt at the breakpoint and when this happens, the Watch window should display the values of variables sum and i (Figure 1.40). Click on the Run toolbar button several more times and you should see the values change as execution progresses (and function dotp() is called repeatedly). Once execution of the program is complete and the message

\[\text{sum} = 40 \text{ (decimal)}\]

is displayed in the console, the Watch window will once again report that identifiers sum and i cannot be found.
If the program is edited so that the variables \texttt{sum} and \texttt{i} are declared as global variables, then their values can be displayed in the \textit{Watch} window at all times. Cut the lines that read

\begin{verbatim}
    int i;
    int sum = 0;
\end{verbatim}

from the definition of function \texttt{dotp()} and paste them immediately following the function declaration

\begin{verbatim}
    int dotp(short *a, short *b, int ncount);
\end{verbatim}

Rebuild and reload the program (by clicking \textit{Build Active Project} or by selecting \textit{Project > Build Active Project}) and repeat the previously described steps involving the breakpoint and \textit{Watch} window in order to verify that the values of \texttt{sum} and \texttt{i} can be displayed at all times.

![Figure 1.39](image1.jpg) \textbf{Figure 1.39} Breakpoint indicated in left margin of program listing.

![Figure 1.40](image2.jpg) \textbf{Figure 1.40} \textit{Watch} window showing the values of variables \texttt{sum} and \texttt{i} during execution of program \texttt{L138_dotp4.c}.
Estimating execution time for function `dotp()` using the profile clock

The time taken to execute function `dotp()` can be estimated using the Code Composer software `Profile Clock`. In the `Debug` perspective,

1. Select `Target > Clock > Enable`.
2. Click the `Restart` toolbar button in the `Debug` window.
3. Clear all existing breakpoints (by double-clicking on the breakpoint symbols in the left-hand margin of the Editor window) and then set breakpoints at the lines
4. `result = dotp(x, y, count);` and `printf("result = %d (decimal) \n", result);`
5. Run the program. It should halt at the first breakpoint.
6. Reset the `Profile Clock` by double-clicking on its icon in the bottom right-hand corner of the Code Composer Studio IDE window.
7. Run the program. It should halt at the second breakpoint.

The number of instruction cycles counted by the profile clock between the two breakpoints, that is, during the execution of function `dotp()`, should be displayed next to the `Profile Clock` icon. On a 375 MHz C6748 processor, each instruction cycle takes 2.67 ns. With the default `Build Properties` setting of no compiler optimization, the function should take around 112 instruction cycles to execute.

To change the compiler optimization level, right-click on the project name `L138_dotp4` in the `C/C++` perspective `Project View` window and select `Build Properties`. In the `Tool Settings` tab under `Configuration Settings`, click on `C6000 Compiler > Basic Options` and set the `Optimization Level`.

Repeat the experiment having set the compiler `Optimization Level` to 2 and you should see a significant reduction in the number of instruction cycles used by function `dotp()`. Using breakpoints and the profile clock can give an indication of the execution times of sections of program, but it does not always work with higher levels of compiler optimization. More detailed profiling of program execution can be achieved using a simulator.

1.5 SUPPORT FILES

The support files `L138_aic3106_init.c`, `L138_aic3106_init.h`, `vectors_intr.asm`, `vectors_poll.asm`, `L138_eXperimenter.ccxml`, `C6748.gel`, and `linker_dsp.cmd` are used by nearly all the examples in this book. These files are stored in folder `c:\eXperimenter\L138_support`.

1.5.1 Initialization and Configuration File (`L138_aic3106_init.c`)  

Source file `L138_aic3106_init.c` contains definitions for a number of functions that may be called from C source files and which are used in many of the example programs in the following chapters.
Functions `L138_initialise_poll()`, `L138_initialise_intr()`, and `L138_initialise_edma()` initialize communications between the OMAP-L138 processor and the AIC3106 codec for polling-, interrupt-, or DMA-based input and output. In each case, McASP0 is configured slightly differently, using functions `L138_mcasp_init_poll()`, `L138_mcasp_init_intr()`, and `L138_mcasp_init_edma()`, respectively. In the case of DMA-based input and output, the parameters of the EDMA3 controller are configured using function `EDMA3_PARAMETER_setup()`. More details of the three different I/O methods are given in Chapter 2.

Functions `input_sample()`, `input_left_sample()`, `input_right_sample()`, `output_sample()`, `output_left_sample()`, and `input_right_sample()` are used to read and write data to and from the codec. These functions make calls to lower level functions provided by the BSL evm omapmapl138_bsl.lib.

Functions `prand()` and `prbs()` are concerned with generating pseudorandom noise sequences. `prand()` uses a Parks–Miller method to generate a pseudorandom sequence of integer values. `prbs()` uses a shift register to generate a PRBS sequence.

### 1.5.2 Header File (L138_aic3106_init.h)

The corresponding header support file `L138_aic3106_init.h` contains function prototypes as well as definitions of several constants.

### 1.5.3 Vector Files (vectors_intr.asm and vectors_poll.asm)

To make use of CPU interrupt INT4, a branch instruction (jump) to the interrupt service routine (ISR) `interrupt4()` defined in a C program, for example, `L138_sine48_buf_intr.c`, must be placed at the appropriate point, as part of a fetch packet, in the interrupt service table (IST). Assembly language file `vectors_intr.asm`, which sets up the IST, is listed in Figure 1.41. Note the underscore preceding the name of the routine or function being called. By convention, this indicates a C function.

For programs that use polling-based I/O, file `vectors_poll.asm` is used in place of `vectors_intr.asm`. The main difference between these files is that there is no branch to function `interrupt4()` in the IST set up by `vectors_poll.asm`. Common to both files is a branch to `c_int00()`, the start of a C program, associated with the reset interrupt (Figures 1.41 and 1.42).
; vectors_intr.asm
;
; interrupt service table for interrupt- and DMA-based i/o
; example programs

.global _vectors
.global _c_int00
.global _vector1
.global _vector2
.global _vector3
.global _interrupt4 ; interrupt service routine
.global _vector5
.global _vector6
.global _vector7
.global _vector8
.global _vector9
.global _vector10
.global _vector11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00 ; C program entry address

; macro creates ISFP entries for IST
VEC_ENTRY .macro addr
    STW   B0,*--B15
    MVKL  addr,B0
    MVKH  addr,B0
    B     B0
    LDW   *B15++,B0
    NOP   2
    NOP
    NOP
    .endm

; dummy interrupt service fetch packet
_vec_dummy:
        B     B3
        NOP   5

; This is the actual interrupt service table (IST).
.sect ".vecs"
.align 1024

_vectors:
_vector0:   VEC_ENTRY _c_int00 ;RESET

Figure 1.41 Listing of interrupt vector file vectors_intr.asm.
1.5.4 Linker Command File \texttt{(linker_dsp.cmd)}

Linker command file \texttt{linker_dsp.cmd} is listed in Figure 1.43. It specifies the memory configuration of the internal and external memory available on the Xper- imenter board and the placing of sections of code and data to absolute addresses in that memory. For example, the .text section, produced by the C compiler, is placed into level 2 RAM/cache, that is, the internal memory of the C6748 digital signal processor, starting at address 0x11800000. The section .vecs created by vectors_intr.asm or by vectors_poll.asm is mapped into IVECS, that is, internal memory starting at address 0x00000000 (the interrupt service table). Chapter 2 contains an example illustrating the use of the \texttt{pragma} directive to create a section named EXT_RAM to be mapped into external memory starting at address 0xC0000000 (SDRAM).

\begin{verbatim}
  ; vectors_poll.asm
  ; interrupt service table for polling-based i/o
  ; example programs
  .global _vectors
  .global _c_int00
  .global _vector1
  .global _vector2
  .global _vector3
  .global _vector4
  .global _vector5
  .global _vector6
  .global _vector7
  .global _vector8

  _vector1:   VEC_ENTRY _vec_dummy ;NMI
  _vector2:   VEC_ENTRY _vec_dummy ;RSVD
  _vector3:   VEC_ENTRY _vec_dummy ;RSVD
  _vector4:   VEC_ENTRY _interrupt4 ;Interrupt4 ISR
  _vector5:   VEC_ENTRY _vec_dummy
  _vector6:   VEC_ENTRY _vec_dummy
  _vector7:   VEC_ENTRY _vec_dummy
  _vector8:   VEC_ENTRY _vec_dummy
  _vector9:   VEC_ENTRY _vec_dummy
  _vector10:  VEC_ENTRY _vec_dummy
  _vector11:  VEC_ENTRY _vec_dummy
  _vector12:  VEC_ENTRY _vec_dummy
  _vector13:  VEC_ENTRY _vec_dummy
  _vector14:  VEC_ENTRY _vec_dummy
  _vector15:  VEC_ENTRY _vec_dummy

  Figure 1.41 (Continued)
\end{verbatim}
1.5 Support Files

.global _vector9
.global _vector10
.global _vector11
.global _vector12
.global _vector13
.global _vector14
.global _vector15

.ref _c_int00 ; C program entry address

; macro creates ISFP entries for IST
VEC_ENTRY .macro addr
  STW   B0, -- B15
  MVKL  addr, B0
  MVKH  addr, B0
  B     B0
  LDW   *B15++, B0
  NOP   2
  NOP
  NOP
.endm

; dummy interrupt service fetch packet
_vec_dummy:
  B     B3
  NOP   5

; This is the actual interrupt service table (IST).
.sect ".vecs"
.align 1024

_vectors:
  _vector0:   VEC_ENTRY _c_int00 ; RESET
  _vector1:   VEC_ENTRY _vec_dummy ; NMI
  _vector2:   VEC_ENTRY _vec_dummy ; RSVD
  _vector3:   VEC_ENTRY _vec_dummy ; RSVD
  _vector4:   VEC_ENTRY _vec_dummy
  _vector5:   VEC_ENTRY _vec_dummy
  _vector6:   VEC_ENTRY _vec_dummy
  _vector7:   VEC_ENTRY _vec_dummy
  _vector8:   VEC_ENTRY _vec_dummy
  _vector9:   VEC_ENTRY _vec_dummy
  _vector10:  VEC_ENTRY _vec_dummy
  _vector11:  VEC_ENTRY _vec_dummy
  _vector12:  VEC_ENTRY _vec_dummy
  _vector13:  VEC_ENTRY _vec_dummy
  _vector14:  VEC_ENTRY _vec_dummy
  _vector15:  VEC_ENTRY _vec_dummy

Figure 1.42 (Continued)
EXERCISES

1. Modify program L138_sine48_buf_intr.c to generate a sine wave with a frequency of 3000 Hz. Verify your result using an oscilloscope connected to the LINE OUT socket on the eXperimenter as well as by using Code Composer to plot the 256 most recently output samples in both time and frequency domains.

2. Write an interrupt-driven program that maintains a buffer containing the 128 most recent input samples read at a sampling frequency of 16 kHz from the AIC3106 codec. Halt the program and plot the buffer contents using Code Composer.

3. Write a program that reads input samples from the left channel of the AIC3106 codec ADC at a sampling frequency of 32 kHz using function input_left_sample() and writes each sample value to the right channel of the AIC3106 codec DAC using function output_right_sample(). Verify the effective connection of the left-hand channel of the LINE IN socket to the right-hand channel of the LINE OUT socket using a signal generator and an oscilloscope. Gradually increase the frequency of the input signal until the amplitude of the output signal is reduced drastically. This frequency corresponds to the bandwidth of the DSP system (discussed in more detail in Chapter 2).

Figure 1.43 Listing of linker command file linker_dsp.cmd.
REFERENCES

1. OMAP-L138 C6-Integra™ DSP + ARM® Processor, SPRS586C, Texas Instruments, Dallas, TX, 2011.
2. TMS320C6748 Fixed/Floating-Point DSP, SPRS590C, Texas Instruments, Dallas, TX, 2011.
3. Low-Power Stereo Audio Codec for Portable Audio/Telephony, SLAS509E, Texas Instruments, Dallas, TX, 2008.