Development Kit For the PIC® MCU

Exercise Book

DSP Analog

dsPIC33FJ128GP706

March 2010

Custom Computer Services, Inc.
Brookfield, Wisconsin, USA
262-522-6500

Recognized Microchip Third-Party Tool Provider

Custom Computer Services, Inc. proudly supports the Microchip brand with highly optimized C compilers and embedded software development tools.

1 UNPACKING AND INSTALLATION

Inventory

- ❑ Use of this kit requires a PC with Windows 95, 98, ME, NT, 2000 or XP. The PC must have a spare 9-Pin Serial or USB port, a CD-ROM drive and 75 MB of disk space.
- ❑ The diagram on the following page shows each component in the DSP Analog kit. Ensure every item is present.

Software

- ❑ Insert the CD into the computer and wait for the installation program to start. If your computer is not set up to auto-run CDs, then select **My Computer** and double-click on the CD drive.
- ❑ Click on **Install** and use the default settings for all subsequent prompts by clicking NEXT, OK, CONTINUE…as required.
- ❑ Identify a directory to be used for the programs in this booklet. The install program will have created an empty directory **c:\program files\picc\projects** that may be used for this purpose.
- ❑ Select the compiler icon on the desktop. In the PCW IDE, click **Help>About** and verify a version number is shown for the IDE and **PCD** to ensure the software was installed properly. Exit the software.

Hardware

- ❑ Connect the PC to the ICD(6) using the USB cable.(1) Connect the prototyping board (10) to the ICD using the modular cable. Plug in the DC adaptor (9) to the power socket and plug it into the prototyping board (10). The first time the ICD-U is connected to the PC, Windows will detect new hardware. Install the USB driver from the CD or website using the new hardware wizard. The driver needs to be installed properly before the device can be used.
- ❑ The LED should be red(2) on the ICD-U to indicate the unit is connected properly.
- ❑ Run the Programmer Control Software by clicking on the CCSLOAD icon on the desktop. Use CCSLOAD Help File for assistance.
- ❑ The software will auto-detect the programmer and target board and the LCD should be illuminated green. If any errors are detected, go to Diagnostic tab. If all tests pass, the hardware is installed properly.
- ❑ Disconnect the hardware until you are ready for Chapter 3. Always disconnect the power to the Prototyping board before connecting/disconnecting the ICD or changing the jumper wires to the Prototyping board.

(1) ICS-S40 can also be used in place of ICD-U. Connect it to an available serial port on the PC using the 9 pin serial cable. There is no driver required for S40.

(2) ICD-U40 units will be dimly illuminated green and may blink while connecting.

① Storage box
② Exercise booklet
③ CD-ROM of C compiler (optional)
④ Serial PC to Prototyping board cable
⑤ Modular ICD to Prototyping board cable
⑥ ICD unit for programming and debugging
⑦ USB (or Serial) PC to ICD cable
⑧ DSP Analog Prototyping Board with the dsPIC33FJ128GP706
(See inside front and back cover for details on the board layout and schematic)
⑨ AC Adaptor (9VDC)

2 USING THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Editor

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>Open>Source File**. Select the file: **c:\program files\picc\examples\ex_stwt.c**
- ❑ Scroll down to the bottom of this file. Notice the editor shows comments, preprocessor directives and C keywords in different colors.
- ❑ Move the cursor over the **Set_timer1** and click. Press the F1 key. Notice a Help file description for set_timer1 appears. The cursor may be placed on any keyword or built-in function and F1 will find help for the item.
- ❑ Review the editor special functions by clicking on **Edit**. The IDE allows various standard cut, paste and copy functions.
- ❑ Review the editor option settings by clicking on **Options>Editor Properties**. The IDE allows selection of the tab size, editor colors, fonts, and many more. Click on **Options>Toolbar>Keyboard>Customize/Setup** to select which icons appear on the toolbars.

Compiler

- ❑ Use the white box on the toolbar to select the compiler. CCS offers different compilers for each family of Microchip parts. All the exercises in this booklet are for the dsPIC33FJ128GP706 chip, a 24-bit opcode part. Make sure **PCD 24-bit** is selected in the white box.
- ❑ The main program compiled is always shown in the lower right corner of the IDE. If this is not the file you want to compile, then click on the tab of the file you want to compile. Right click into editor and select **Make file project**.
- ❑ Click **Options>Project Options>Global Defines** and review the list of directories the compiler uses to search for included files. The install program should have put two directories in this list: devices and drivers
- ❑ Normally the file formats need not be changed and global defines are not used in these exercises. To review these settings, click **Options>Project Options>Output Files** and **Options>Project Options>Global Defines**.
- ❑ Click the compile icon to compile. Notice the compilation box shows the files created and the amount of ROM and RAM used by this program. Compilation box will close automatically when done compiling.

Viewer

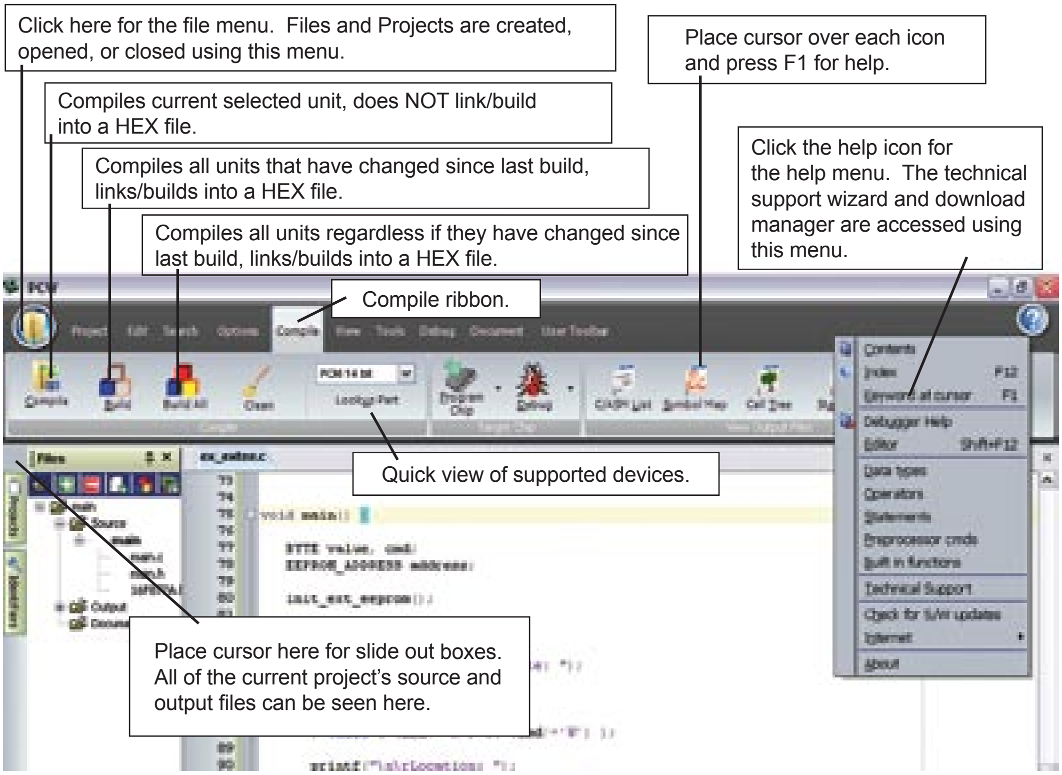
❑ Click **Compile>Symbol Map**. This file shows how the RAM in the microcontroller is used. Identifiers that start with @ are compiler generated variables. Notice some locations are used by more than one item. This is because those variables are not active at the same time.

❑ Click **Compile>C/ASM List.** This file shows the original C code and the assembly code generated for the C. Scroll down to the line:

```
int_count=INTS_PER_SECOND;
```

❑ Notice there are two assembly instructions generated. The first loads 64 into the WO register. INTS_PER_SECOND is #defined in the file to 100. 64 hex is 100 decimal. The second instruction moves WO into a memory location. Switch to the Symbol Map to find the memory location where int_count is located.

❑ Click **View>Data Sheet**, then **View.** This brings up the Microchip data sheet for the microprocessor being used in the current project.



Click here for the file menu. Files and Projects are created, opened, or closed using this menu.

Compiles current selected unit, does NOT link/build into a HEX file.

Compiles all units that have changed since last build, links/builds into a HEX file.

Compiles all units regardless if they have changed since last build, links/builds into a HEX file.

Place cursor over each icon and press F1 for help.

Click the help icon for the help menu. The technical support wizard and download manager are accessed using this menu.

Compile ribbon.

Quick view of supported devices.

Place cursor here for slide out boxes. All of the current project's source and output files can be seen here.

```
void main() {

   BYTE value, cmd;
   EEPROM_ADDRESS address;

   init_ext_eeprom();

      printf("\r\nLocation: ");
```

3 COMPILING AND RUNNING A PROGRAM

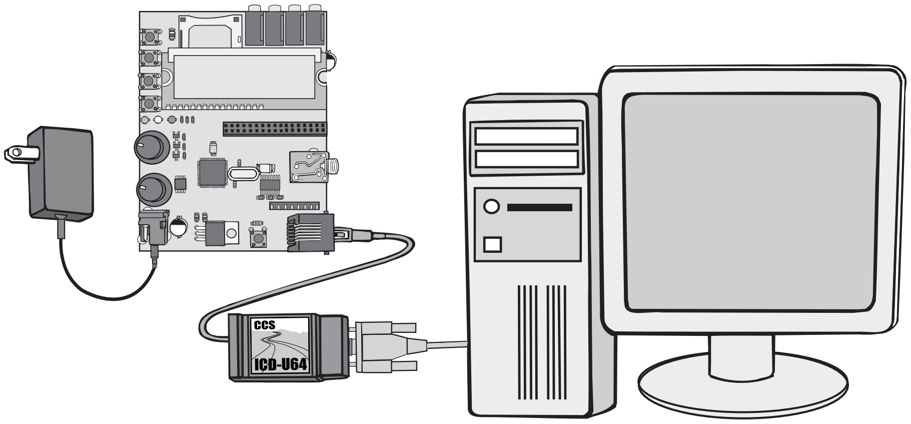
Editor

- ❑ Open the PCW IDE. If any files are open, click **File>Close All**
- ❑ Click **File>New>Source File** and enter the filename **EX3.C**
- ❑ Type in the following program and **Compile.**

```
#include <33fj128gp706.h>
#device ICD=TRUE
#fuses HS, NOWDT, NOCOE, PR
#use delay(clock=12000000)

#define GREEN_LED PIN_C14
void main () {
  while (TRUE) {
    output_low (GREEN_LED);
     delay_ms (1000);
    output_high (GREEN_LED);
    delay_ms (1000);
  }
}
```

NOTES

- The first four lines of this program define the basic hardware environment. The chip being used is the dsPIC33FJ128GP706, running at 12 MHz with the ICD debugger.
- The #define is used to enhance readability by referring to GREEN_LED in the program instead of PIN_C14.
- The “while (TRUE)” is a simple way to create a loop that never stops.
- The “delay_ms(1000)” is a one second delay (1000 milliseconds).



- ❑ Connect the ICD to the Prototyping board using the modular cable, and connect the ICD to the PC. Power up the Prototyping board.
- ❑ Click **Debug>Enable Debugger** and wait for the program to load.
- ❑ If you are using the ICD-U40 and the debugger cannot communicate to the ICD unit go to the debug configure tab and make sure ICD-USB from the list box is selected.
- ❑ Click the green go icon:
- ❑ Expect the debugger window status block to turn yellow indicating the program is running.
- ❑ The green LED on the Prototyping board should be flashing. One second on and one second off.
- ❑ The program can be stopped by clicking on the stop icon:

FURTHER STUDY

A *Modify the program to light the green LED for 5 seconds, then the yellow for 1 second and the red for 5 seconds.*

B *Add to the program a #define macro called "delay_seconds" so the delay_ms(1000) can be replaced with : delay_seconds(1); and delay_ms(5000) can be: delay_seconds(5);.*

Note: *Name these new programs EX3A.c and EX3B.c and follow the same naming convention throughout this booklet.*

4 HANDLING INPUT

- ❑ Type in the following program, named **EX4.C, Compile and Run**:

```
#include <33fj128gp706.h>
#device ICD=TRUE
#fuses HS, NOWDT, NOCOE, PR
#use delay(clock=12000000)

#define GREEN_LED   PIN_C14
#define YELLOW_LED  PIN_G9
#define RED_LED        PIN_C2
#define PUSH_BUTTON PIN_B2
//
light_one_led(int led) {
  output_low(GREEN_LED);
  output_low(YELLOW_LED);
  output_low(RED_LED);
  switch(led) {
   case 0 : output_high(GREEN_LED);  break;
   case 1 : output_high(YELLOW_LED); break;
   case 2 : output_high(RED_LED);    break;
  }
}
wait_for_one_press() {
  while(input(PUSH_BUTTON)) ;
  while(!input(PUSH_BUTTON)) ;
}
void main() {
  while(TRUE) {
     light_one_led(0);
     wait_for_one_press();
     light_one_led(1);
     wait_for_one_press();
     light_one_led(2);
     wait_for_one_press();
  }
}
```

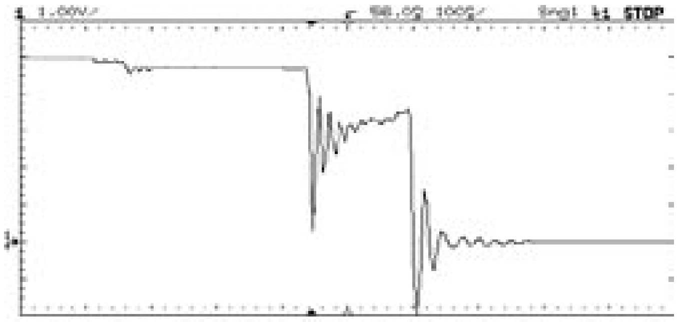
- ❑ Click **Compiler.**
- ❑ Click **GO** in Debugger Window
- ❑ The green LED should come on. Press the top button and the yellow LED should light and then the red LED when pressed again. Add the following new typebelow the // lines:

```
    typedef enum  {GREEN,YELLOW,RED}  colors;
```

- ❑ Change the parameter to light_one_led to colors instead of int.
- ❑ Change the 0, 1, 2 in the call to GREEN, YELLOW, RED.

NOTES

- The Prototyping board has one momentary push-button that may be used as an input to the program. The input pin is connected to a 4.7K pull-up resistor to +5V. The button, when pressed, shorts the input pin to ground. The pin is normally high while in this configuration, but it is low while the button is pressed.
- This program shows how to use simple C functions. The function wait_for_one_press() will first get stuck in a loop while the input pin is high (not pressed). It then waits in another loop while the pin is low. The function returns as soon as the pin goes high again. Note that the loops, since they do not do anything while waiting, do not look like much-they are a simple ; (do nothing).
- When the button is pressed once, it is common for several very quick connect disconnect cycles to occur. This can cause the LEDs to advance more than once for each press. A simple debounce algorithm can fix the problem. Add the following line between the two while loops: delay_ms(100); The following scope picture of a button press depicts the problem:



FURTHER STUDY

A *Modify the program so that while the button is held down the LEDs alternate as fast as possible. When the button is not pressed the LED state freezes. This creates a random color program.*

5 PROGRAM STRUCTURE

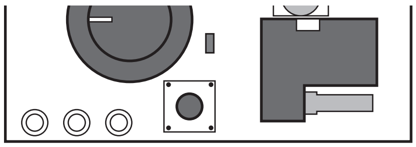
- ❑ It is good practice to put all the hardware definitions for a given design into a common file that can be reused by all programs for that board. Open EX4.C and drag the cursor over (highlight) the first 9 lines of the file. Click **Edit>Paste to file** and give it the name **prototype.h**.
- ❑ It is also helpful to collect a library of utility functions to use as needed for future programs. Note that just because a function is part of a program does not mean it takes up memory. The compiler deletes functions that are not used. Highlight the wait_for_one_press() function, light_one_led function and the typedef line (if added from Chapter 4 Notes section) and save as a new file named utility.c. Open **utility.c** and add the following new function to the file:

```
show_binary_on_leds(int n) {
  output_low(GREEN_LED);
  output_low(YELLOW_LED);
  output_low(RED_LED);
  if( bit_test(n,0) )
    output_high(GREEN_LED);
  if( bit_test(n,1) )
    output_high(YELLOW_LED);
  if( bit_test(n,2) )
    output_high(RED_LED);
}
```

- ❑ Close all files and start a new file named **EX5.C** as follows:

```
#include <prototype.h>
#include <utility.c>
void main() {
  int count = 0;
  while(TRUE) {
    show_binary_on_leds(count);
    wait_for_one_press();
    count++;
  }
}
```

- [ ] Compile and Run the program. Check that with each button press, the LEDs increment in a binary number 0-7 as shown here.



```
0  0  0 ───► 0
0  0  1 ───► 1
0  1  0 ───► 2
0  1  1 ───► 3
1  0  0 ───► 4
1  0  1 ───► 5
1  1  0 ───► 6
1  1  1 ───► 7
```

NOTES

- In C, a function must either appear in the input stream before it is used OR it must have a prototype. A prototype is the part of the function definition before the "{". In a program where main calls function A and function A calls function B, the order in the file must be B, A, MAIN. As an alternative, have Ap, Bp, MAIN, A, B where Ap and Bp are prototypes. Frequently, prototypes are put into a header file with a .h extension.
- The scope, initialization, and life of C variables depend on where and how they are declared. The following is a non-inclusive summary of the common variable scopes. Note that if a variable has an initialization (like **int a=1**;) the assignment happens each time the variable comes to life.

Where it is defined	Can be accessed	Life of the variable
Inside a function	Only in that function	While function is alive
Inside a function with STATIC	Only in that function	During the entire run of the program
Outside all functions	In any function defined afterwards in the file	During the entire run of the program
After "{" inside a function	Only between the "{" and corresponding "}"	Only up to the corresponding "}"

FURTHER STUDY

A *Modify the program to increment the binary number 1 every second (the button is not used).*

B *Instead of the built-in function BIT_TEST use the standard C operators (such as & and ==) to test the bits.*

6 DEBUGGING

- ❑ Open **EX5.C** and start the debugger **Debug>Enable Debugger**.
- ❑ Click the reset icon to ensure the target is ready.
- ❑ Click the step-over icon twice. This is the step over command. Each click causes a line of C code to be executed. The highlighted line has not been executed, but is the next line to be executed.
- ❑ Step over the `show _ binary _ on _ leds(count);` line and notice that one click executed the entire function. This is the way step over works. Click step over on `wait _ for _ one _ press();`. Press the prototype button and notice the debugger now stops since the function terminates.
- ❑ Click the **Watches** tab, then the add icon to add a watch. Enter **count or choose count the variables from list**, then click **Add Watch**. Notice the value shown. Continue to step over through the loop a few more times (press the button as required) and notice the count watch increments.
- ❑ Step over until the call to `show _ binary _ on _ leds(count);` is highlighted. This time, instead of step over, use the standard step icon several times and notice the debugger is now stepping into the function.
- ❑ Click the GO icon to allow the program to run. Press the prototype button a couple of times to verify that the program is running normally. Click the stop icon to halt execution. Notice the C source line that the program stopped on. This is the line were the program is waiting for a button press.
- ❑ In the editor, click on `show _ binary _ on _ leds(count);` to move the editor cursor to that line. Then click the Breaks tab and click the add icon to set a breakpoint. The debugger will now stop every time that line is reached in the code. Click the GO icon and then press the prototype button. The debugger should now stop on the breakpoint. Repeat this a couple of times to see how the breakpoint works.
- ❑ Click **Compile>C/ASM list**. Scroll down to the highlighted line. Notice that one assembly instruction was already executed for the next line. This is another side effect of the ICD debugger. Sometimes breakpoints slip by one ASM instruction.
- ❑ Click the step over icon a few times and note that when the list file is the selected window, the debugger has executed one assembly instruction per click instead of one entire C line.
- ❑ Close all files and start a new file **EX6.C** as follows:

```
#include <prototype.h>

void main() {
   int8 a,b,c;

   a=11;
   b=5;
   c=a+b;
   c=b-a;
   while(TRUE);
}
```

- ❑ Compile the program and step-over until the c=a+b is executed. Add a watch for c and the expected value is 16.
- ❑ Step-over the subtraction and notice the value of c. The **int** data type by default is signed, so c will be the expected –6. The modular arithmetic works like a car odometer when the car is in reverse only in binary. For example, 00000001 minus 1 is 00000000, subtract another 1 and you get 11111111. When using signed binary a 1 in the most significant bit signifies that the number is negative. Therefore, 11111010 represents $^{-}6$
- ❑ Reset and again step up to the c=a+b. Click the **Eval** tab. This pane allows a one time expression evaluation. Type in a+b and click **Eval** to see the debugger and calculate the result. The complete expression may also be put in the watch pane as well. Now enter b=10 and click **Eval**. This expression will actually change the value of B if the “keep side effects” check box of the evaluation tab is checked. Check it and click **Eval** again. Step over the addition line and click the **Watches** tab to observe the c value was calculated with the new value of b.

FURTHER STUDY

A *Modify the program to include the following C operators to see how they work:*
** / % & ^*
Then, with b=2 try these operators: >> <<
Finally, try the unary complement operator with: c=~a;

B *Design a program to test the results of the relational operators:*
<   >   = =   !=
by exercising them with b as 10, 11, and 12.
Then, try the logical operators || and && with the four combinations of a=0,1 and b=0,1.
Finally, try the unary not operator with: c=!a; when a is 0 and 1.

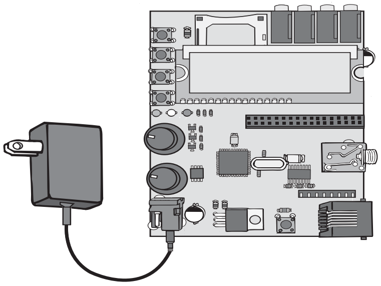
7 STAND-ALONE PROGRAMS AND EXTERNAL EEPROM

The example in chapter 5 always begins counting at 0, but in some applications, the ability to remember past a power cycle is necessary. EEPROM is used for this purpose. The microcontroller on this board does not have internal EEPROM, so to save long term data, an AT25256A external EEPROM is used.

- ❑ Click **File>New>Source File** and name the file **EX7.C**

```
#include <prototype.h>
#include <utility.c>

#define EEPROM_SELECT PIN_D9
#define EEPROM_DI     PIN_F2
#define EEPROM_DO     PIN_F3
#define EEPROM_CLK    PIN_F6
#include <at25256.c>

#define EEPROM_ADDR 0x002

void main()
{
   unsigned int8 count = 0;

   init_ext_eeprom();
   count = read_ext_eeprom(EEPROM_ADDR);

   while(1)
   {
      show_binary_on_leds(count);
      wait_for_one_press();
      count++;
      write_ext_eeprom(EEPROM_ADDR, count);
   }
}
```

- ❑ Click **Compiler** and Run the program from within debugger window. Verify that when the program is halted, reset and restarted that the count continues where it left off.
- ❑ Close the debug window.
- ❑ Copy the prototype.h file to a new file called protoalone.h. Remove from this file the line:
  `#device ICD=TRUE`
- ❑ This makes a program that uses the new include file a stand alone program which does not need the ICD to run.
- ❑ Modify **EX7.C** to use protoalone.h. Compile the program, then click **Tools>CCSLOAD** to load the program onto the Prototyping board.



- ❑ Disconnect the power from the Prototyping board, then disconnect the ICD from the Prototyping board.
- ❑ Power up only the Prototyping board and verify the program runs correctly.
- ❑ Press the reset button on the Prototyping board and release. The LEDs should go off while in reset, then the program will restart.

NOTES

- The AT25256 uses SPI, a common 3 or 4 wire protocol. The protocol uses two data lines for full duplex communication. One is outgoing data (SDO) and the other is incoming (SDI) The name of the data line is given with respect to the current device; this means that the microcontroller's SDO is sending the data to the EEPROM, which receives the data on its SDI. A third line clocks the serial data on both the SDI and SDO, and is generated by the master device.
- An optional fourth line called a chip select may be used. This allows multiple slaves to utilize the same physical bus. When the master device wishes to communicate with a particular slave, it drives the chip select low to enable the slave. Each slave requires an independent chip select.
- There is a limit as to how many times a given location in the data EEPROM can be written to. For this reason, a program should be designed not to write any more often than necessary. For example, one might wait until there are no changes to a system setting for 5 seconds before saving a new value to EEPROM. Some system designs can give early warning on power down and the program can only save to EEPROM at power down.

FURTHER STUDY

A *Modify the EX7.c program so that 10 EEPROM locations are used and each time the button is pressed only one of the 10 locations is written to and the location changes with each press. This will extend the life of this unit by 10 times, if it were a real product.*

Hint: The count value could be the sum of all 10 locations %8.

8 ANALOG TO DIGITAL CONVERSION AND LIQUID CRYSTAL DISPLAYS

Digital signal processors process analog signals that have been digitized. An Analog to Digital Converter (ADC) is a device that digitizes the signal from an analog source. Many microcontrollers now integrate ADC's.

LCD displays are a common way of displaying textual and increasingly, graphical data. CCS provides a driver library for easy use of most standard alpha-numeric displays

- ❑ Click **File > New > Source File** and name the file **EX8.C**
- ❑ Type in the following code:

```
#include <protoalone.h>

#define  LCD_RS_PIN     PIN_D1
#define  LCD_RW_PIN     PIN_D2
#define  LCD_ENABLE_PIN PIN_D3

#include <lcd.c>

#define BOTTOM_POT_PORT sAN9
#define BOTTOM_POT_CHANNEL 9

#define TOP_POT_PORT sAN8
#define TOP_POT_CHANNEL 8

void main(){
  unsigned int16 adc1 = 0;
  unsigned int16 adc2 = 0;

  lcd_init();
  setup_adc_ports(BOTTOM_POT_PORT);
  setup_adc_ports2(TOP_POT_PORT);

  setup_adc(ADC_CLOCK_INTERNAL);
  setup_adc2(ADC_CLOCK_INTERNAL);

  set_adc_channel(BOTTOM_POT_CHANNEL);
  set_adc_channel2(TOP_POT_CHANNEL);
```

(continued...)

(...continued)

```
  while(TRUE){
    adc1 = read_adc();
    adc2 = read_adc2();

    printf(LCD_PUTC, "\fTop Pot=%lu\nBottom Pot=%lu",adc2,adc1);
    delay_ms(100);
  }
}
```

- ☐ Compile the program, then click **Tools>ICD** to load the program onto the Prototyping board.
- ☐ Turn the top and bottom pots and verify that the LCD updates with the new ADC readings.

NOTES

- A delay between LCD updates is necessary or the LCD will update too quickly, and the refresh lines will blank out the text. A delay of 100 ms or more is recommended.
- The ADC output is left aligned; this means that even though the output of the ADC is 10 bits, if a 16 bit variable is used, the value left in the variable will be shifted to the right 6 bits. This makes 64 the smallest interval between ADC readings. In this format, the range of possible values stretches from 0 to 65535, but only 10 bits of resolution exist.
- If wanting to use the ADC in an 8 bit mode, use #device adc=8 at the beginning of code.
- '\f' clears the LCD and '\n' writes a new line.

FURTHER STUDY

A *Read both potentiometers with a single ADC module. OR together both ports in `setup _ adc _ ports()` and change the channel between readings with `set _ adc _ channel()`. Make sure to pause at least 15 us between setting and reading the channel.*

B *Use the LCD to display which buttons are currently depressed. Find the button state with input() and use %u in printf() to output the state.*

9 AUDIO CODEC

A primary feature of this development kit is the ability to generate tones and other sounds while driving speakers and headphones.

WARNING: Do not wear headphones directly over your ears, or use large, amplified outputs for these examples. Some examples, depending on default settings or settings you have changed, will probably be uncomfortably or even dangerously loud.

For the examples presented in this booklet and for you own experimentation, we suggest that you use inexpensive headphones or speakers. During your experimentation, you may generate signals that are harmful to audio equipment.

In general, you should avoid signals that:

- Have sharp points or quickly change such as some saw-tooth wave forms and square waveforms; these signals may be dangerous to both your hearing and your equipment.
- Signals that are in a frequency range beyond the range of your device. Generally, staying within the human hearing range (20-20,000 Hz) will ensure the equipment's safety.

A codec chip provides hardware support for multimedia tasks, generally audio or video. Using a codec chip can significantly reduce the load on the microprocessor, but at the expense of an additional hardware package.

The TLV320 codec chip is an audio ADC and DAC (Digital to Analog Converter) that can sample incoming signals with 32 bit resolution at 96 kHz; allowing high fidelity sound to be sampled, digitized, processed and reproduced. It connects to the microcontroller through two separate interfaces. A control interface utilizes SPI, a three or four wire interface, to configure the codec. A second interface communicates with the audio module through a protocol called Multichannel.

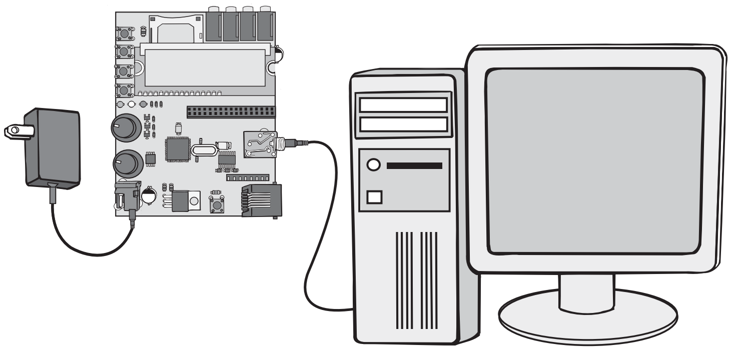
The TLV320 codec has the capability for several transmission protocols. In this device, the Multichannel protocol is used. The Multichannel Protocol allows for left and right channel (stereo) sound to be transmitted over 4 lines. Multichannel, like many other PCM protocols, divide their transmissions into frames. Each frame is representative of a single sample. The data is in a 2's complement, signed integer (of the same size as your sample size) format. The clock signal that accompanies the data clocks in individual bits on its rising edge.

- ❑ Click **File>New>Source File** and name the file **EX9.C**
- ❑ Type the following code to demonstrate the use of the codec to sample and record sound with a microphone and loop it back out to a headphone output.

```
#include <protoalone.h>
#include <TLV320AIC23B.c>

void main()
{
  signed int16 leftChannel = 0x0000;
  signed int16 rightChannel = 0x0000;

  delay_ms(600);


  setup_dci((MULTICHANNEL_MODE | MULTI_DEVICE_BUS | UNDERFLOW_LAST
             | DCI_SLAVE |DCI_CLOCK_INPUT | SAMPLE_RISING_EDGE),
            DCI_2WORD_FRAME | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT,
             RECEIVE_SLOT0 | RECEIVE_SLOT1, TRANSMIT_SLOT0 |
             TRANSMIT_SLOT1, 0);

   dci_start();

  codec_initialize();
  codec_setup_analog_path(DAC_SELECT | ADC_MIC_INPUT);
  codec_setup_hp_output(HP_NO_GAIN, HP_NO_GAIN);

  while(TRUE){
    codec_read_data16(&leftChannel, &rightChannel);
    codec_write_data16(&leftChannel, &rightChannel);
  }
}
```

- ❑ Connect a microphone to J8 and a speaker to J7.
- ❑ Click **Compile** and then click **Tools>ICD** to load the program
- ❑ Verify that the microphone loops sound to the speaker.

9 AUDIO CODEC (CONT.)



NOTES

- The clock is generally clocking a data in and data out line, however depending on the application circuit, one of these may not be present (if the device is only recording, or only reproducing sound). The final signal is a frame sync signal. This signal outputs a pulse at the selected sampling frequency. Data is transmitted and/or received 1 clock cycle after the rising edge of the frame sync.
- `codec_setup_analog_path()` sets up the internal pathway sound data is obtained from. By default, sound is sampled from the Line input. If you wish to obtain audio data from this source, leave out the ADC_MIC_INPUT argument.
- Because a microphone is a mono audio source, the codec sends the same values for both the left and right channel.

FURTHER STUDY

A *Use `codec_setup_analog_path(DAC_SELECT)` and `codec_setup_line_input(...)` to utilize the Line Input instead of the microphone to loop stereo sound.*
Note: *The microphone cannot be connected to Line Input because it does not produce a Line Output. Line Outputs are common on CD players and sound cards.*

B *Generate a triangular waveform by incrementing and ouput using `codec_write_data16()`*

C *Negate one audio channel when a button is depressed passing a dummy variable to one parameter codec_write_data16(...).*

10 PULSE CODE MODULATION

Pulse code modulation (PCM) is the most common way to convey raw sound data. Raw samples of the signal are taken in the time domain, or sequentially. The sample values represent the displacement of the signal, or the sound pressure at a given point in time.

PCM is popular because it is the raw output of a sampling device like an ADC and it is the direct input to a reconstruction device like a DAC.

Many codec chips provide means to obtain and output PCM data. Other codecs may also include digital filters that refine the signals. Others still may even include hardware to decode popular compression formats like .mp3.

Several chips in the dsPIC family have an on-chip peripheral that supports buffered transmission of PCM data. This peripheral is called the *Data Converter Interface* or *DCI*.

- ❑ Click **File>New** and name the file **EX10.C**
- ❑ Type in the following code that creates a table of values that represent a sine wave encoded as PCM data.

```
#include <33FJ128GP706.h>
#fuses HS, NOWDT, NOCOE, NODEBUG, PR
#use delay(clock=12000000)

#include <math.h>
#include <TLV320AIC23B.c>

#define PCM_TABLE_SIZE 256
unsigned int8 pcm_pos;
signed int16 pcm_table[PCM_TABLE_SIZE];
const float INCREMENT = ((2*PI)/PCM_TABLE_SIZE);

signed int16 leftChannel = 0x0000;
signed int16 rightChannel = 0x0000;

void main(){
   unsigned int8 i;
   pcm_pos = 0;

   for(i = 0; i < 255; i++){
      pcm_table[i] = 127 * sin(INCREMENT * i) + 128;
   }
   //make sure to init i = 255
   pcm_table[i] = 127 * sin(INCREMENT * i) + 128;

   delay_ms(600);
```

(continued...)

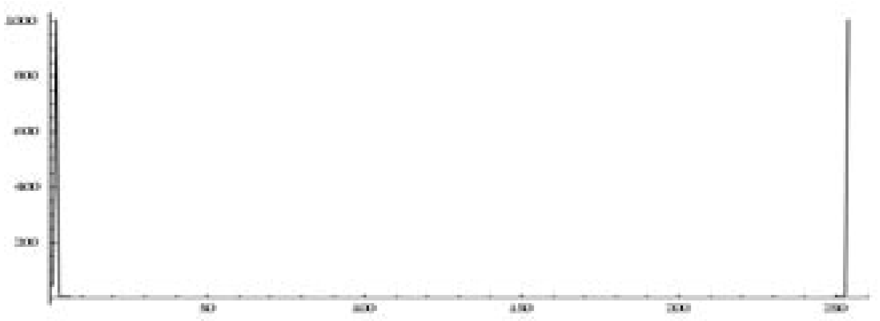
(...continued)

```
   //Starting the DCI peripheral

   setup_dci((MULTICHANNEL_MODE | MULTI_DEVICE_BUS | UNDERFLOW_LAST |
            DCI_SLAVE | DCI_CLOCK_INPUT | SAMPLE_RISING_EDGE),
            DCI_2WORD_FRAME | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT,
            RECEIVE_SLOT0 | RECEIVE_SLOT1, TRANSMIT_SLOT0 | TRANS-
MIT_SLOT1,0);

   dci_start();

   //Initializing the Codec
   codec_initialize();

   //Setting the timer interrupt
   setup_timer1(TMR_INTERNAL | TMR_DIV_BY_8, 10);
   enable_interrupts(INT_TIMER1);
   enable_interrupts(INTR_GLOBAL);

   while(TRUE){
     codec_write_data16(&leftChannel, &rightChannel);
   }
}

#int_timer1
void timer1_isr(){
  pcm_pos = (pcm_pos + 1) % PCM_TABLE_SIZE;

  leftChannel = pcm_table[pcm_pos];
  rightChannel = leftChannel;
}
```

- ❑ Click **Compile** and then click **Tools>ICD** to load the program
- ❑ Connect a speaker to J7 and verify the output is a steady tone.

NOTES

- This example uses timer overflow interrupts to create the intervals at which the next waveform value is output. If using this method in the future, do not forget to enable_interrupts().
- Be cautious when using this method to send data to the codec. This will work when the signal is readily available with no other processing loads on the dsPIC. See the Chapter 12 for another method of timing data sent to the codec.
- To save memory, this PCM signal is only 256 samples long. This produces a nearly pure tone; however some side tones and beat frequencies can be heard. The following Fourier analysis of the samples shows that some energy is present in the signal at relatively low frequencies (5-10 Hz), which accounts for the tapping or thumping noise you may hear.



- Varying the speed at which the data is changed varies the frequency of the output tone

FURTHER STUDY

A *Change the period of the timer to change the frequency of the output data.*

B *Change the phase of the left and right channel outputs.*

C *Use another timer to generate different frequency on the left and right channels.*

11 SAMPLE SIZE

Sample size is resolution to which the original signal was sampled. An increased sample size results in greater reproduction fidelity.

Sample frequency is how quickly the samples are taken. Mathematically, two samples of a waveform are required to reproduce the signal perfectly via a Fourier series. Unfortunately, PCM does not have the capability to perfectly reproduce a signal, and it still requires two samples per waveform to reproduce a recognizable tone. The higher the sample rate, the less aliasing (distortion) of a waveform will occur, resulting in higher quality. 44.1 kHz is the CD standard sampling rate for this reason. The highest tone a human (with excellent hearing) can hear is around 22,000 Hz. Since two samples are required per waveform to create a recognizable tone, the sample frequency must be twice the highest frequency to be reproduced. 44.1 kHz is just slightly more than double the highest frequency humans hear.

- ❑ Click **File>New>Source** and name the file **EX11.C**
- ❑ Type the following code to show the effects of sample size on a sine wave.

```
#include <33fj128gp706.h>
#fuses HS, NOWDT, NOCOE, NODEBUG, PR
#use delay(clock=12000000)
#use rs232(baud=9600, UART2)

#include <math.h>
#include <TLV320AIC23B.c>

#define PCM_TABLE_SIZE 256
#define MAXIMUM_VAL 32768

unsigned int8 pcm_pos;
signed int16 pcm_table[PCM_TABLE_SIZE];
const float INCREMENT = ((2*PI)/PCM_TABLE_SIZE);

signed int16 leftChannel = 0x0000;
signed int16 rightChannel = 0x0000;

unsigned int8 sample_resolution = 16; // 1 to 16
signed int32 sample_step;
signed int32 base;
signed int32 multiplier;
int8 down_sample_period = 1;

#define RED_LED       PIN_C2
#define YELLOW_LED    PIN_G9
#define GREEN_LED     PIN_C14
```

(continued...)

(...continued)

```
#define BUTTON_1 PIN_B2
#define BUTTON_2 PIN_B3
#define BUTTON_3 PIN_B4
#define BUTTON_4 PIN_B5

void main()
{
   unsigned int8 i;

   setup_adc_ports(NO_ANALOGS);
   setup_adc_ports2(NO_ANALOGS);

   pcm_pos = 0;

   sample_step = 1;
   for(i = 0;i < sample_resolution - 1;i++){
      sample_step *= 2;
   }
   base = MAXIMUM_VAL / sample_step;

   delay_ms(600);

   /* Start up the dci peripheral */
   setup_dci((MULTICHANNEL_MODE | MULTI_DEVICE_BUS | UNDERFLOW_LAST
              | DCI_SLAVE | DCI_CLOCK_INPUT | SAMPLE_RISING_EDGE),
              DCI_2WORD_FRAME | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT,
              RECEIVE_SLOT0 | RECEIVE_SLOT1, TRANSMIT_SLOT0 |
              TRANSMIT_SLOT1, 0);

   /* Initialize the codec */
   codec_initialize();
   codec_setup_analog_path(DAC_SELECT | ADC_MIC_INPUT);

   dci_start();

   /*generate sine wave PCM table */
   for(i = 0; i < 255; i++)
   {
      pcm_table[i] = 127 * sin(INCREMENT * i) + 128;
   }
   //make sure to init i = 255
   pcm_table[i] = 127 * sin(INCREMENT * i) + 128;
```

(continued...)

(...continued)

```
 setup_timer1(TMR_INTERNAL | TMR_DIV_BY_8, 10);
   setup_timer2(TMR_INTERNAL | TMR_DIV_BY_64, 0x4000);
   enable_interrupts(INT_TIMER1);
   enable_interrupts(INT_TIMER2);
   enable_interrupts(INTR_GLOBAL);

   while(1)
   {
      codec_write_data16(&leftChannel, 0);
   }
}

#int_timer1
void timer1_isr()
{
   /* Choose the next waveform value */
   pcm_pos = (pcm_pos + 1) % PCM_TABLE_SIZE;

   /* Generate and format the value */
   leftChannel = pcm_table[pcm_pos];

   multiplier = leftChannel / base;
   leftChannel = base * multiplier;

   //rightChannel = leftChannel;
   rightChannel = 0;
}

#int_timer2
void timer2_isr()
{
   int8 i;

   if(!input(BUTTON_4))
   {
      output_toggle(GREEN_LED);

      if(sample_resolution > 1)
         sample_resolution--;

      sample_step = 1;
      for(i = 0;i < sample_resolution - 1;i++)
         sample_step *= 2;
```

(continued...)

(...continued)

```
      base = MAXIMUM_VAL / sample_step;

         return;
      }

      if(!input(BUTTON_3))
      {
         output_toggle(YELLOW_LED);

         if(sample_resolution < 16)
            sample_resolution++;

         sample_step = 1;
         for(i = 0;i < sample_resolution - 1;i++)
            sample_step *= 2;

         base = MAXIMUM_VAL / sample_step;

         return;
      }
   }
```

- ☐ Click **Compile** and then click **Tools>ICD** to load the program.
- ☐ Connect a speaker to J7 and verify the output is a steady tone.
- ☐ Press Button 3 (B4) to increase the sample size and Button 4 (B5) to decrease the sample size.

NOTES

- Sampling frequency and sample size are two major factors in audio quality. However, saturation may also significantly affect the quality of the signal. Most standard audio equipment works over a 1 Volt RMS range. If the amplitude of the signal is pushed too high, generally by hardware amplifiers, parts of the signal may be pushed outside of the 1 V RMS range; creating a plateau on the waveform as if it was simply chopped off. This will produce a rattling sound. It is the rattling sound heard when a stereo is turned up to maximum volume.
- Down-sampling the signal is achieved by clipping off the least significant bits of the PCM sign and scaling it appropriately. 16 bits per channel is the lowest sample size the codec can produce, however the codec may sample as high as 32 bits per channel.
- Timer interrupts are effective for obtaining user input at preset intervals. If you choose to use interrupts for any purpose, don't forget to `enable _ interrupts(INTR _ GLOBAL)`. Failing to do so will allow no interrupts to run, regardless of their source.

FURTHER STUDY

A *Apply the down-sampling and frequency reductions to only one audio channel and compare the two channels.*
Significantly increase the amplitude of the output signal to understand the effects of saturation; the effects will be more obvious if you do so at a large sample size. Try multiplying the leftChannel value.

B *Significantly increase the amplitude of the output signal to understand the effects of saturation; the effects will be more obvious if you do so at a large sample size. Try multiplying the* `leftChannel` *value.*

C ***Advanced:*** *Calculate the Root-Mean-Square volume of the signal and use it normalize the signal (keep the volume constant) as it is down-sampled.*

D *Down sample input from the microphone instead of a sine wave input.*

12 DIRECT MEMORY ACCESS AND SD/MMC

A nonvolatile storage medium can be used to save and playback data from the codec. However, the SD/MMC card can not be accessed continuously due to buffer write delays which would create gaps in the waveform.

Direct Memory Access (DMA) solves this issue by allowing the processor to asynchronously write to the codec. This is done by writing the data to send to a buffer in RAM and then configuring the DMA module to use that buffer to write data to the codec without the intervention of the CPU. When it is finished, it will interrupt the CPU for more data. DMA can be used to transfer data while reading more data from the SD card, process incoming data and operate other peripherals or devices.

DMA is not specific to only the DCI (codec) peripheral and can be used for many peripherals on the microcontroller including SPI and the Analog to Digital converter for both reading and writing from and to the peripheral.

The DMA peripheral can be configured to both read and write from the codec in a double buffered fashion. While one buffer is sent to the codec, the CPU will fill the other

- ❑ Click **File>New>Source File** and name the file **EX12.C**
- ❑ Type the following program:

```
#include <33fj128gp706.h>
#fuses HS, NOWDT, NOCOE, NODEBUG, PR_PLL
#use delay(clock=80000000)
#use rs232(baud=9600, UART2)

#define MMCSD_PIN_SCL     PIN_G6 //o
#define MMCSD_PIN_SDI     PIN_G7 //i
#define MMCSD_PIN_SDO     PIN_G8 //o
#define MMCSD_PIN_SELECT PIN_D8

#include <mmcsd.c>
#include <TLV320AIC23B.c>

#define RED_LED      PIN_C2
#define YELLOW_LED   PIN_G9
#define GREEN_LED    PIN_C14

signed int16 TxBufferA[256];
signed int16 TxBufferB[256];
signed int16 RxBufferA[256];
signed int16 RxBufferB[256];
```

(continued...)

(...continued)

```
/* DMA RAM begins at location 0x4000 and extends for 2Kbytes after,
 this allows us to reserve this area for our exclusive use;
 This location changes with the type of chip used, check your
 datasheet for DMA RAM locations. */
#define DMA_BASE 0x4000
#locate TxBufferA=0x4000
#locate TxBufferB=0x4200
#locate RxBufferA=0x4400
#locate RxBufferB=0x4600

#define DMA_OUT_DCI (0x3C0000|DMA_OUT|getenv("sfr:TXBUF0"))
#define DMA_IN_DCI  (0x3C0000|DMA_IN |getenv("sfr:RXBUF0"))
#define RECORD_LENGTH (1024*128) // divide by 1024 for kB

unsigned int32 mmcsd_address=0;
char           mode=0;
int16*         bufferPtr;
int1           bufferIsLoaded;

#word CLKDIV = getenv("sfr:CLKDIV")
#bit PLLPOST0 = CLKDIV.6
#bit PLLPOST1 = CLKDIV.7
#word PLLFBD = getenv("sfr:PLLFBD")
#word OSCCON = getenv("sfr:OSCCON")
#bit LOCK = OSCCON.5

void setup_pll()
{
   //M = 40, N1 = 3, N2 = 2

   //N1
   CLKDIV = 1; //PLLPRE

   //N2
   PLLPOST0 = 0;
   PLLPOST1 = 0;

   //M
   PLLFBD = 38;
   while(!lock);
}
```

(continued...)

(...continued)

```
void main()
{
   setup_adc_ports(NO_ANALOGS);
   setup_adc_ports2(NO_ANALOGS);

   setup_pll();
   delay_ms(10);

   if(mmcsd_init() != MMCSD_GOODEC)
   {
      output_high(RED_LED);
      printf("Fails init op");
   }

   delay_ms(600);

   /* Setup the dci peripheral */
   setup_dci((MULTICHANNEL_MODE | MULTI_DEVICE_BUS | UNDERFLOW_LAST |
   DCI_SLAVE | DCI_CLOCK_INPUT | SAMPLE_RISING_EDGE),
   DCI_1WORD_FRAME  | DCI_16BIT_WORD | DCI_2WORD_INTERRUPT,
   RECEIVE_SLOT0 | RECEIVE_SLOT1,
   TRANSMIT_SLOT0 | TRANSMIT_SLOT1, 0);

   dci_start();

   /* Initialize the codec */
   codec_initialize();
   codec_setup_analog_path(DAC_SELECT | ADC_MIC_INPUT);

    /* Setup DMA */
   setup_dma(1, DMA_OUT_DCI, DMA_WORD);
   setup_dma(2, DMA_IN_DCI,  DMA_WORD);

   clear_interrupt(INT_DMA1);
   clear_interrupt(INT_DMA2);
   enable_interrupts(INTR_GLOBAL);

   delay_ms(200);
   printf("\r\nBegin\r\n");
```

(continued...)

(...continued)

```
while(1){
      printf("\r\nRecord or playback? (r, p)");
      mode = getc();
      delay_ms(10);

      if(mode == 'r')
      {
         printf("\r\nRecording\r\n");
         mmcsd_address=0;
         bufferPtr = RxBufferA;
         bufferIsLoaded = TRUE;
         delay_ms(10);

         enable_interrupts(INT_DMA2);
         dma_start(2, DMA_ONE_SHOT, RxBufferA,0x0100);

         while(mmcsd_address<RECORD_LENGTH){

            while(bufferIsLoaded);

            if(bufferPtr == RxBufferA){
               dma_start(2, DMA_ONE_SHOT, RxBufferA, 0x0100);
               bufferPtr = RxBufferB;
            }
            else{
               dma_start(2, DMA_ONE_SHOT, RxBufferB, 0x0100);
               bufferPtr = RxBufferA;
            }

            disable_interrupts(INT_DMA2);
            mmcsd_write_block(mmcsd_address,512,(int8*)bufferPtr)
            mmcsd_address+=512;
            bufferIsLoaded = TRUE;
            enable_interrupts(INT_DMA2);
         }
         disable_interrupts(INT_DMA1);
         disable_interrupts(INT_DMA2);

         printf("\r\nDone");
      }
```

(continued...)

(...continued)

```
      else if(mode == ‘p’)
      {
         printf(“\r\nPlayback”);
         delay_ms(100);

         mmcsd_address=0x0000;
         bufferIsLoaded = FALSE;
         bufferPtr = TxBufferA;
         enable_interrupts(INT_DMA1);

         mmcsd_read_block(mmcsd_address,512,(int8*)bufferPtr);
         mmcsd_address+=512;

         while(mmcsd_address<RECORD_LENGTH){


            if(bufferPtr == TxBufferA){
               dma_start(1, DMA_ONE_SHOT, TxBufferA, 0x0100);

               bufferPtr = TxBufferB;
            }
            else{
               dma_start(1, DMA_ONE_SHOT, TxBufferB, 0x0100);
               bufferPtr = TxBufferA;
            }

            mmcsd_read_block(mmcsd_address,512,(int8*)bufferPtr);
            mmcsd_address+=512;
            bufferIsLoaded = TRUE;

            while(bufferIsLoaded);
         }
         disable_interrupts(INT_DMA1);
         printf(“\r\ndone\r\n”);
      }
   }
}
}
```

(continued...)

(...continued)

```
#INT_DMA1
void dma1_isr()
{
   bufferIsLoaded = FALSE;
   clear_interrupt(INT_DMA1);
}

#INT_DMA2
void dma2_isr()
{
   bufferIsLoaded = FALSE;
   clear_interrupt(INT_DMA2);
}
```

- ❑ Use the included serial cable to connect the PC and prototyping board and open the program **Tools>Serial Port Monitor**. Set the correct COMM port if necessary.
- ❑ Click Compile and then **Tools>ICD** to load the program.
- ❑ Check the red LED is not lit to verify that the MMC/SD card was recognized. Try pressing the reset button once if the initialization fails.
- ❑ At the prompt, press ‘r’ or ‘p’ on the keyboard to record and play microphone samples.

NOTES

- It is a good idea to check that the mmcsd_xxxx functions return MMCSD_GOODEC. If they do not, the media may not be of the correct type, or it may have been removed.
- DMA is comprised of 8 channels. Each channel is one way (read or write) and provides an interrupt when it is finished transferring data.
- DMA is used as a double buffer here, but it is possible to use a single buffer to conserve DMA RAM
- There are 2 Kbytes of DMA RAM available. This example, uses all of it. 512 bytes for each buffer on both receive and transmit channels. 512 bytes was chosen because it matches the size of an SD Card block. However, a smaller, or arbitrary number could be chosen.

13 MIGRATING TO YOUR OWN HARDWARE

Setting up the DCI module on your own hardware involves:

- Choosing a protocol (Multi-channel, I2S, or AC’97 (16 or 20 bit)
- Configuring the size and number of your sample frames
- Calculating your sample rate if master mode is selected

DCI Protocols

The DCI peripheral supports multiple audio PCM protocols. Each of them may be utilized by many different devices and each may find their individual use. The following is a summary of each that may be used with this development kit.

Multi-channel

Multi-channel DCI is an adaptable, general purpose protocol. It consists of data frames which are delineated by a pulse of the Frame sync signal (COFS). Following the frame sync signal are anywhere between 1 and 16 data words. Each data word is comprised of 4 to 16 bits (user configurable). The direction (input or output) of each word is configurable by the user.

This bus is generally used when multiple devices are being used on the same bus. For example, a codec controller (a dsPIC) may be on a bus with a modem codec in a fax machine. On the same bus, we have a voice codec to receive input from the telephone. We can use a multi-channel protocol with an input word from each device followed by an output data word to each codec for a total of 4 data words. Keep in mind when designing such a bus that most codec chips can allow you to select a range of possible data words and when in the frame they are transmitted. The range is generally limited though, so you should be sure that you can use the multi-channel bus without any collisions (which would result in contention on this protocol).

I2S, Inter-Integrated Sound

I2S is a protocol designed for stereo sound transmission between two devices. This is a relatively common protocol and provides for full duplex sound transmission. Unlike I2C however, there is no addressing system which usually limits the number of devices to master and slave. Also four wires are required, frame sync, input, output and a clock. Depending on your codec hardware, you may be able to add data words to each half of the transmission, similar to a multi-channel protocol. Refer to your codec datasheet for possible configurations.

I2S uses the COFS signal on the dsPIC as a frame sync that has a 50% duty cycle over the course of a frame. The particulars about which half is what are configurable; normally the left channel data is transferred one clock cycle after the rising edge of the COFS signal, followed by the right channel data after the falling edge of the signal.

Other sound transmission types

AC'97 both 16 and 20 bit protocols are supported by the DCI module on the dsPIC. The codec used on this board (the Texas Instruments TLV320AIC23B) does not utilize this protocol, though you may use it on your own hardware. This transmission consists of a set protocol with fixed data size. It is recommended that you manually configure the DCI module for use with this transmission protocol, especially the 20 bit configuration.

Troubleshooting

AC'97 both 16 and 20 bit protocols are supported by the DCI module on the dsPIC. The codec used on this board (the Texas Instruments TLV320AIC23B) does not utilize this protocol, though you may use it on your own hardware. This transmission consists of a set protocol with fixed data size. It is recommended that you manually configure the DCI module for use with this transmission protocol, especially the 20 bit configuration.

- ❑ The MCLR pin must be in a high state for the chip to run. Note the Prototyping board schematic uses a pushbutton to ground this pin and to reset the chip.
- ❑ Most problems involve the clock. Make sure the configuration fuses are set to the proper oscillator setting. In the above case, for a 12MHz crystal HS (High Speed) and PR (Primary Oscillator) is the proper setting. In the above circuit, the size of the resistor may need adjustment depending upon the crystal.
- ❑ If the program does not seem to be running, verify 3.3 Volts on the MCLR pin and the two power pins.
- ❑ Isolate hardware problems from firmware problems by running a program with the following at the start of main ( ) and check B0 with a logic probe or scope:

```
while(TRUE)   {
    output_low (PIN_B0);
    delay_ms (1000);
    output_high (PIN_B0);
    delay_ms (1000);
  }
```

References

This booklet is not intended to be a tutorial for the dsPIC33FJ128GP706 or the C programming language. It does attempt to cover the basic use and operation of the development tools. There are some helpful tips and techniques covered, however, this is far from complete instruction on C programming. For the reader not using this as a part of a class and without prior C experience the following references should help.

On The Web

Comprehensive list of PICmicro® Development tools and information	www.mcuspace.com
Microchip Home Page	www.microchip.com
CCS Compiler/Tools Home Page	www.ccsinfo.com
CCS Compiler/Tools Software Update Page	www.ccsinfo.com click: Support → Downloads
C Compiler User Message Exchange	www.ccsinfo.com/forum
Device Datasheets List	www.ccsinfo.com click: Support → Device Datasheets
C Compiler Technical Support	support@ccsinfo.com
Texas Instruments Data Sheet TLV320AIC23BPW	http://focus.ti.com/docs/prod/folders/print/tlv320aic23b.html

Other Development Tools

EMULATORS

The ICD used in this booklet uses two I/O pins on the chip to communicate with a small debug program in the chip. This is a basic debug tool that takes up some of the chip's resources (I/O pins and memory). An emulator replaces the chip with a special connector that connects to a unit that emulates the chip. The debugging works in a simulator manner except that the chip has all of its normal resources, the debugger runs faster and there are more debug features. For example an emulator typically will allow any number of breakpoints. Some of the emulators can break on an external event like some signal on the target board changing. Some emulators can break on an external event like some that were executed before a breakpoint was reached. Emulators cost between $500 and $3000 depending on the chips they cover and the features.

DEVICE PROGRAMMERS

The ICD can be used to program FLASH chips as was done in these exercises. A stand alone device programmer may be used to program all the chips. These programmers will use the .HEX file output from the compiler to do the programming. Many standard EEPROM programmers do know how to program the Microchip parts. There are a large number of Microchip only device programmers in the $100-$200 price range. Note that some chips can be programmed once (OTP) and some parts need to be erased under a UV light before they can be re-programmed (Windowed). CCS offers the Mach X which is a stand-alone programmer and can be used as an in-circuit debugger.

PROTOTYPING BOARDS

There are a large number of Prototyping boards available from a number of sources. Some have an ICD interface and others simply have a socket for a chip that is externally programmed. Some boards have some advanced functionality on the board to help design complex software. For example, CCS has a Prototyping board with a full 56K modem on board and a TCP/IP stack chip ready to run internet applications such as an e-mail sending program or a mini web server. Another Prototyping board from CCS has a USB interface chip, making it easy to start developing USB application programs.

SIMULATORS

A simulator is a program that runs on the PC and pretends to be a microcontroller chip. A simulator offers all the normal debug capability such as single stepping and looking at variables, however there is no interaction with real hardware. This works well if you want to test a math function but not so good if you want to test an interface to another chip. With the availability of low cost tools, such as the ICD in this kit, there is less interest in simulators. Microchip offers a free simulator that can be downloaded from their web site. Some other vendors offer simulators as a part of their development packages.

CCS Programmer Control Software

The CCSLOAD software will work for all the CCS device programmers and replaces the older ICD.EXE and MACHX.EXE software. The CCSLOAD software is stand-alone and does not require any other software on the PC. CCSLOAD supports ICD-Sxx, ICD-Uxx, Mach X, Load-n-Go, and PRIME8.

Powerful Command Line Options in Windows and Linux

- Specify operational settings at the execution level
- Set-up software to perform, tasks like save, set target Vdd
- Preset with operational or control settings for user

Easy to use Production Interface

- Simply point, click and program
- Additions to HEX file organization include associating comments or a graphic image to a file to better ensure proper file selection for programming
- Hands-Free mode auto programs each time a new target is connected to the programmer
- PC audio cues indicate success and fail

Extensive Diagnostics

- Each target pin connection can be individually tested
- Programming and debugging is tested with known good programs
- Various PC driver tests to identify specific driver installation problems

Enhanced Security Options

- Erase chips that failed programming
- Verify protected code cannot be read after programming
- File wide CRC checking

Automatic Serial Numbering Options

- Program memory or Data EEPROM
- Incremented, from a file list or by user prompt
- Binary, ASCII string or UNICODE string

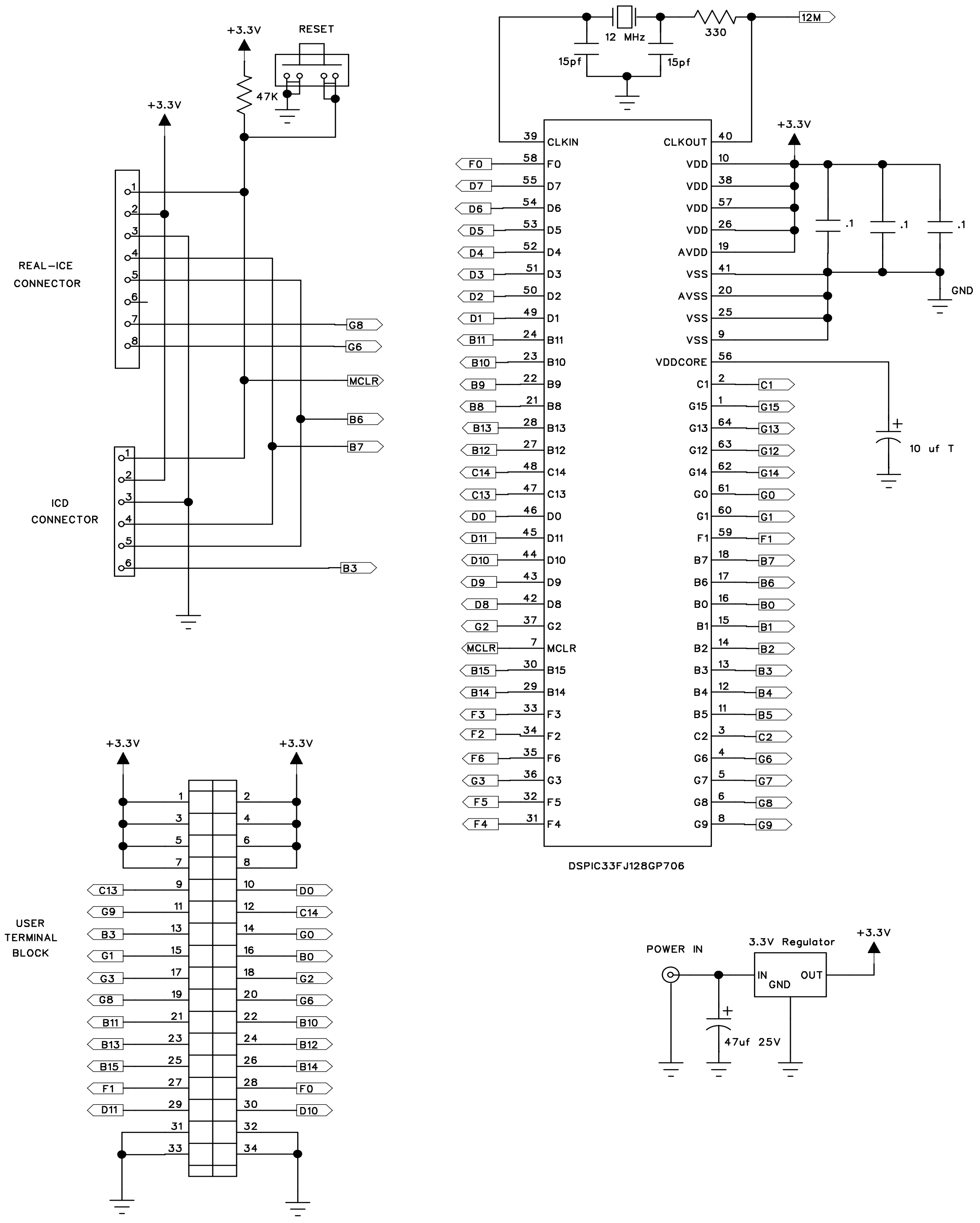
CCS IDE owners can use the CCSLOAD program with:

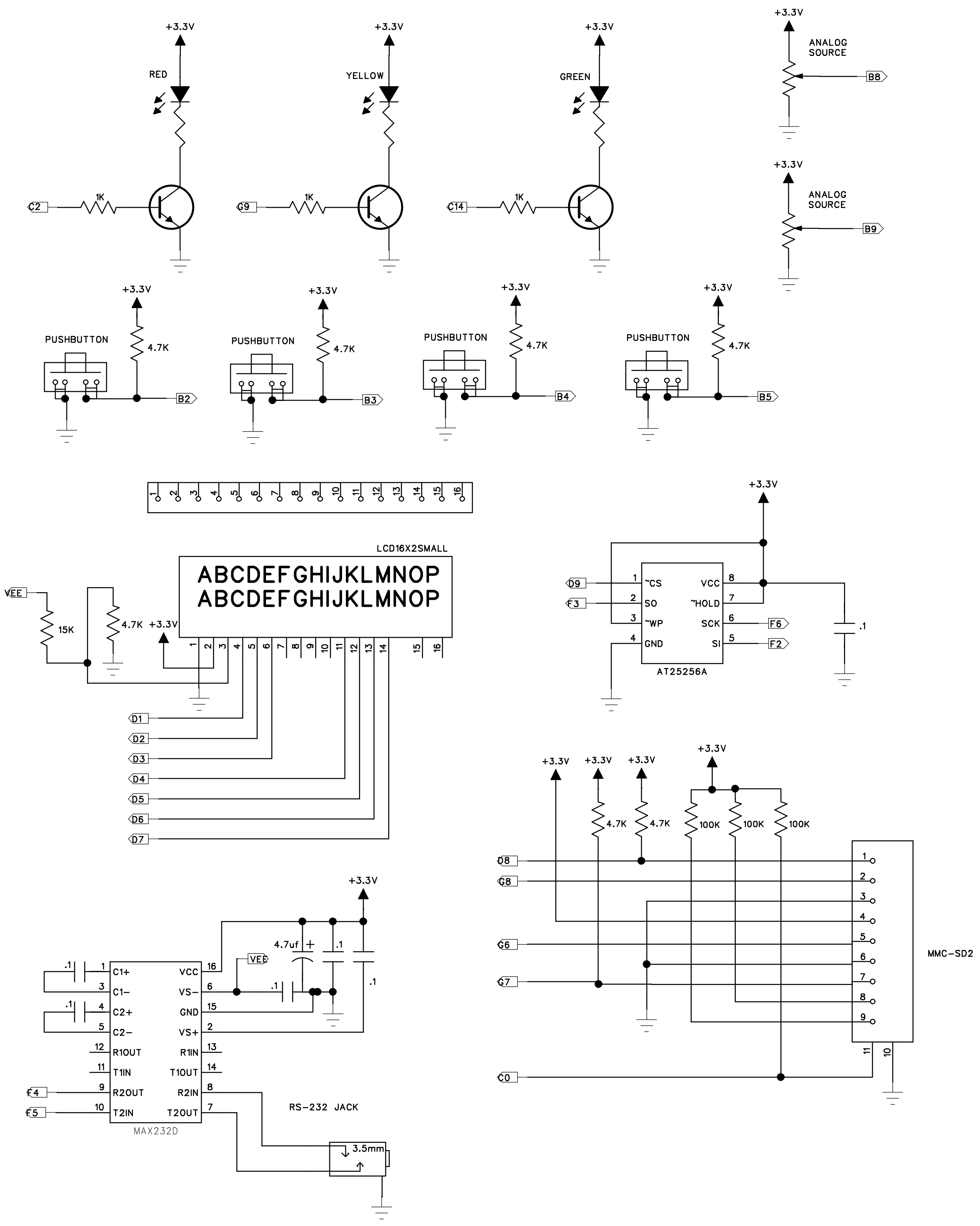
- MPLAB®ICD 2/ICD 3
- MPLAB®REAL ICE™
- **All CCS programmers and debuggers**

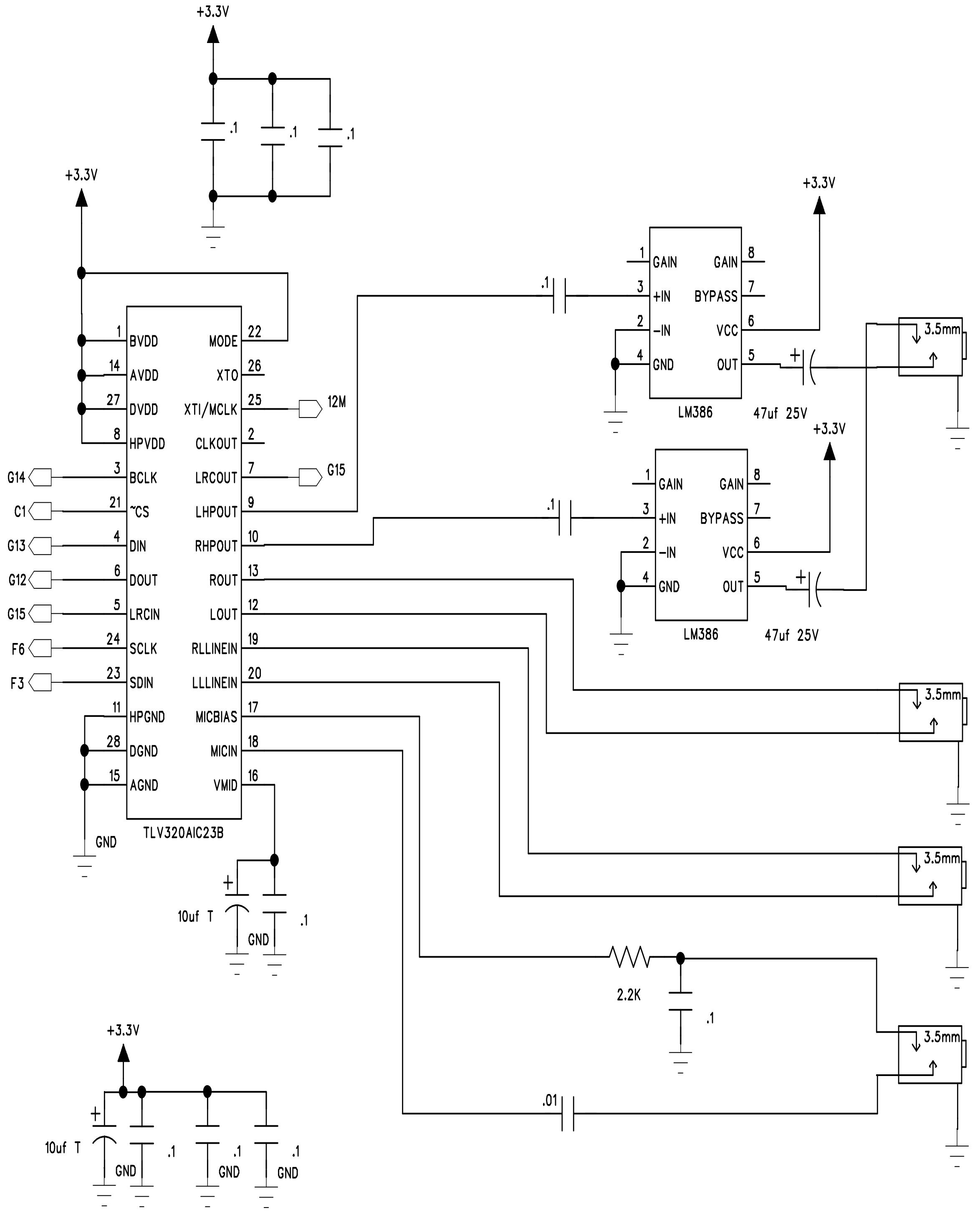
How to Get Started:

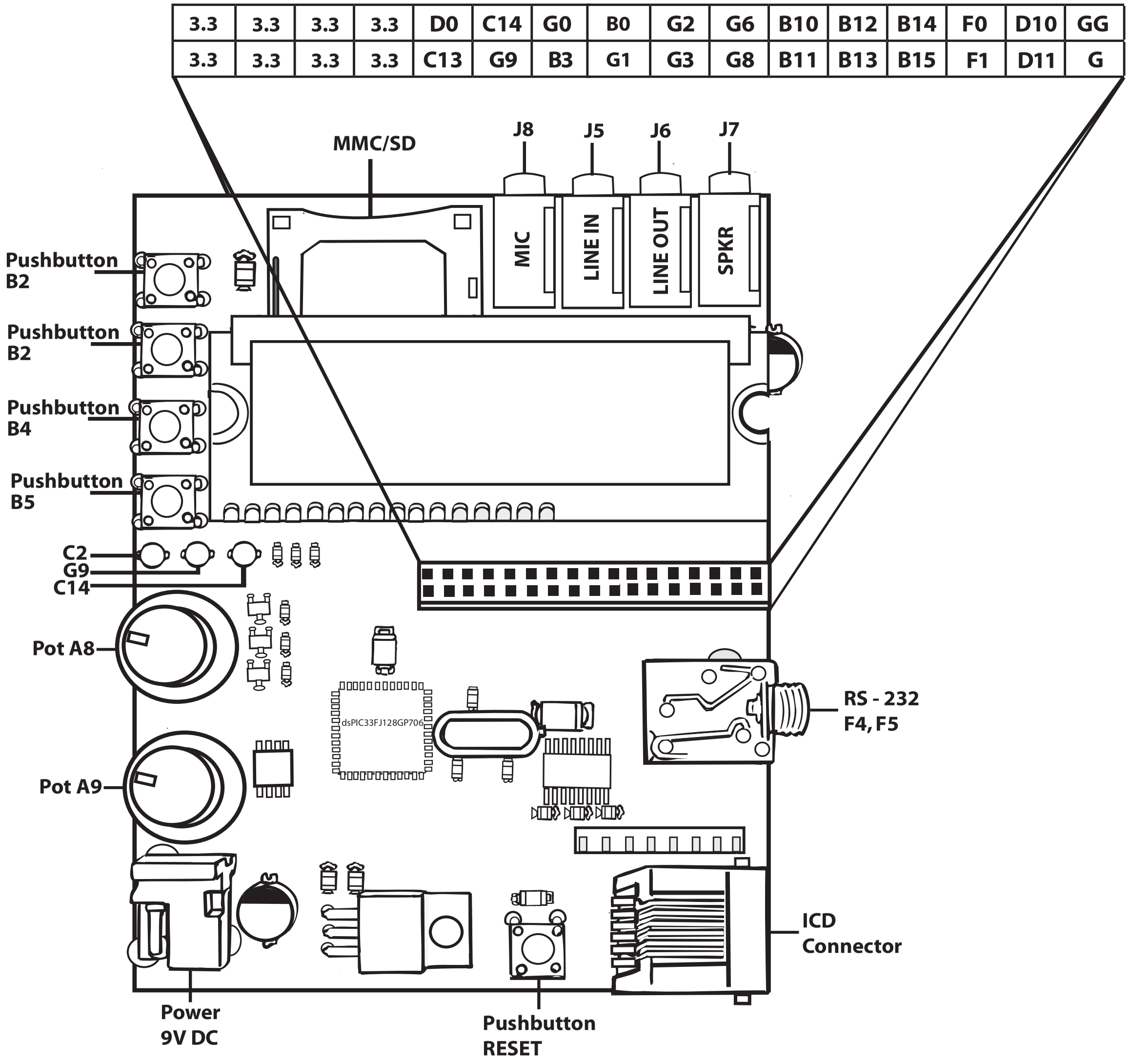
Step 1: *Connect Programmer to PC and target board. Software will auto-detect the programmer and device.*
Step 2: *Select Hex File for target board.*
Step 3: *Select Test Target. Status bar will show current progress of the operation.*
Step 4: *Click “Write to Chip” to program the device.*

Use the Diagnostics tab for troubleshooting or the ccsload.chm help file for additional assistance.


+3.3V
RESET
47K
REAL-ICE CONNECTOR
ICD CONNECTOR
G8
G6
MCLR
B6
B7
B3
12 MHz
15pf
15pf
330
12M
39 CLKIN
40 CLKOUT
58 F0
55 D7
54 D6
53 D5
52 D4
51 D3
50 D2
49 D1
24 B11
23 B10
22 B9
21 B8
28 B13
27 B12
48 C14
47 C13
46 D0
45 D11
44 D10
43 D9
42 D8
37 G2
7 MCLR
30 B15
29 B14
33 F3
34 F2
35 F6
36 G3
32 F5
31 F4
10 VDD
38 VDD
57 VDD
26 VDD
19 AVDD
41 VSS
20 AVSS
25 VSS
9 VSS
56 VDDCORE
2 C1
1 G15
64 G13
63 G12
62 G14
61 G0
60 G1
59 F1
18 B7
17 B6
16 B0
15 B1
14 B2
13 B3
12 B4
11 B5
3 C2
4 G6
5 G7
6 G8
8 G9
.1
.1
.1
GND
10 uf T
DSPIC33FJ128GP706
USER TERMINAL BLOCK
C13, G9, B3, G1, G3, G8, B11, B13, B15, F1, D11
D0, C14, G0, B0, G2, G6, B10, B12, B14, F0, D10
POWER IN
3.3V Regulator
IN
GND
OUT
47uf 25V


+3.3V
RED
YELLOW
GREEN
1K
C2
G9
C14
ANALOG SOURCE
B8
B9
PUSHBUTTON
4.7K
B2
B3
B4
B5
LCD16X2SMALL
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
VEE
15K
D1
D2
D3
D4
D5
D6
D7
~CS
SO
~WP
GND
VCC
~HOLD
SCK
SI
D9
F3
F6
F2
AT25256A
.1
100K
D8
G8
G6
G7
C0
MMC-SD2
MAX232D
C1+
C1-
C2+
C2-
R1OUT
T1IN
R2OUT
T2IN
VS-
VS+
R1IN
T1OUT
R2IN
T2OUT
4.7uf
F4
F5
RS-232 JACK
3.5mm


+3.3V
.1
.1
.1
+3.3V
1 BVDD
14 AVDD
27 DVDD
8 HPVDD
3 BCLK
21 ~CS
4 DIN
6 DOUT
5 LRCIN
24 SCLK
23 SDIN
11 HPGND
28 DGND
15 AGND
22 MODE
26 XTO
25 XTI/MCLK
2 CLKOUT
7 LRCOUT
9 LHPOUT
10 RHPOUT
13 ROUT
12 LOUT
19 RLLINEIN
20 LLLINEIN
17 MICBIAS
18 MICIN
16 VMID
TLV320AIC23B
G14
C1
G13
G12
G15
F6
F3
12M
G15
GND
10uf T
.1
GND
+3.3V
1 GAIN
3 +IN
2 -IN
4 GND
8 GAIN
7 BYPASS
6 VCC
5 OUT
LM386
47uf 25V
3.5mm
2.2K
.1
.01
10uf T
.1
.1
.1
GND
GND
GND


3.3 3.3 3.3 3.3 D0 C14 G0 B0 G2 G6 B10 B12 B14 F0 D10 GG
3.3 3.3 3.3 3.3 C13 G9 B3 G1 G3 G8 B11 B13 B15 F1 D11 G
MMC/SD
J8
J5
J6
J7
MIC
LINE IN
LINE OUT
SPKR
Pushbutton B2
Pushbutton B2
Pushbutton B4
Pushbutton B5
C2
G9
C14
Pot A8
Pot A9
dsPIC33FJ128GP706
RS - 232 F4, F5
ICD Connector
Power 9V DC
Pushbutton RESET