MASTERs 2012

LAB Manual for 1674 AUD

Creating Audio and DSP applications with 16/32 bit microcontrollers

Table of Contents

MICROCHIP

MASTERs Conference

LAB 1: Digital Filtering

Purpose:

To implement an example of digital filtering in speech and audio applications.

Overview:
This lab exercise will introduce you to digital filtering applications. You will design and use a FIR Digital Notch filter to remove an unwanted tone from an audio signal. You will do this on a PIC32MX 32 bit microcontroller. You are provided with a code setup that adds a 400Hz tone to an input audio/speech signal. Your objective is to design a FIR Digital Notch filter to suppress this tone. You will first use the Digital Filter Design Tool to design the filter. You will then incorporate the filter into your application.

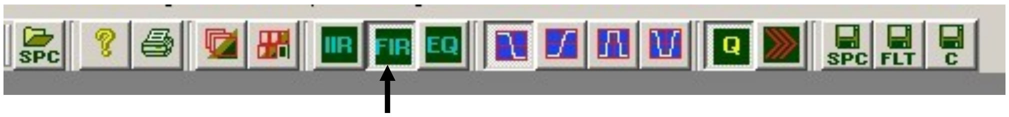
Note: While the Digital Filter Design tool is primarily intended for the Microchip's dsPIC DSC devices, the FIR filters generated by this tool can easily be used with FIR filter function in the XC32 DSP library for PIC32 devices.

Hardware:

You will use the PIC32 Audio Development Board, MPLAB REAL ICE and Headphones for this lab.

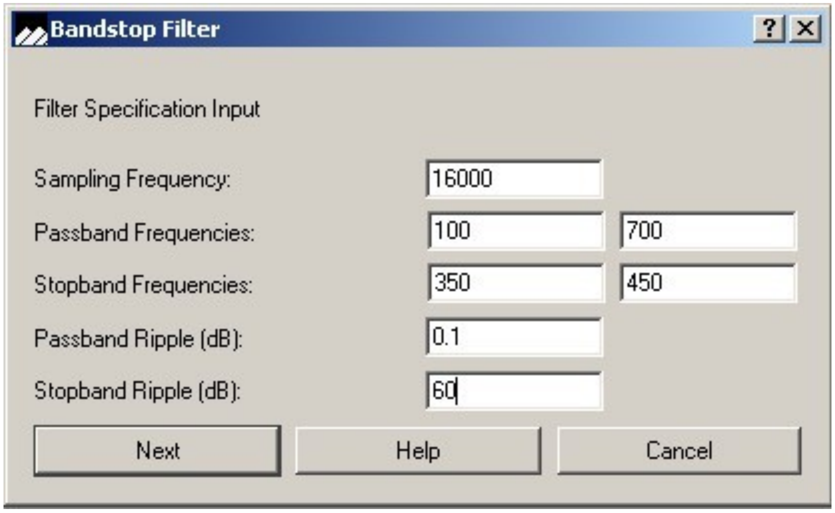
Procedure: Design a FIR Digital Notch Filter

1. On your PC, click on *Start->All Programs->MDS->dsPICFD->dsPICFD.exe.*
2. You will now see the dsPIC FD tool on your screen. Click on the FIR button in the tool menu. See image below.



Choose FIR Filter

3. This will display a filter choice window on the screen. Choose Bandstop filter and click Next.
4. You should now enter the parameters for this filter. The sampling rate for our application is 16KHz. We need to suppress a tone at 400Hz. So choose the lower pass-band frequency at 100 and upper pass-band frequency at 700. Choose the lower stop-band frequency at 350Hz and upper stop-band frequency at 450. Set pass-band ripple at 0.1dB and stop-band ripple at 60dB. See following figure. Click Next.



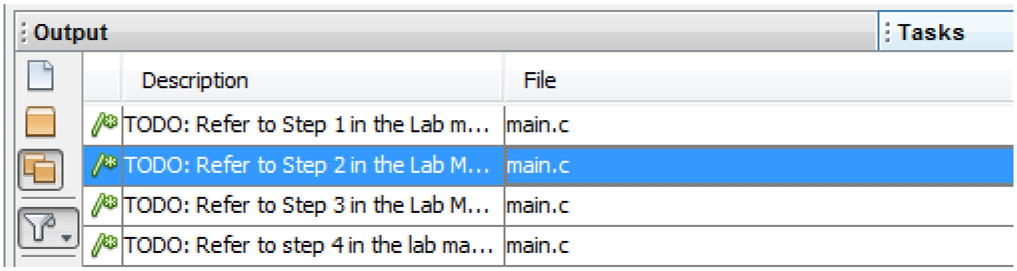
Filter Specification Input

Sampling Frequency: 16000
Passband Frequencies: 100 700
Stopband Frequencies: 350 450
Passband Ripple (dB): 0.1
Stopband Ripple (dB): 60

5. You will now see a dialog box for Filter Window choices. Choose Kaiser and click Next.
6. The designed filter specification will be displayed on the screen. To generate the filter coefficient which can be added to the application, click on *Codegen->Microchip->dsPIC30/33.*
7. You will now see Code Generation Option window. Click OK. This brings up a Save Dialog box. Name the file as `filterCoeffs` and save it at a known location on the PC.
8. Close dsPIC FD.
9. Browse to the location where you saved the filter coefficient file. Open the file `filterCoeffs.s`. Open this file in Notepad and review it. This file is intended for dsPIC30F/33F assembler and needs to be modified for use in PIC32 'C' language application. All assembler directives should be removed and only the filter coefficient data should be retained in a 'C' style array. **You don't have to do this modification for this lab**. The lab already includes the modified C language compatible filter coefficients file. Close the file and proceed to the next step.

Procedure: Integrate Digital FIR Notch Filter into the application.

1. Open MPLAB X IDE. Click on *File->Open Project*. Browse to the class folder. Open the `LAB 1` folder and click on `LAB 1.X`.
2. This will open up the LAB 1 project in MPLAB X IDE.
3. In the Project Explorer window, under Source Files, double click on `main.c`. You will need to add code to this file in order to complete the lab. Look for the "`TODO`" comment to know where code should be added. Every such comment has a coding step associated with it. You can also use the MPLAB X IDE Tasks Tab to locate



Description	File
TODO: Refer to Step 1 in the Lab m...	main.c
TODO: Refer to Step 2 in the Lab M...	main.c
TODO: Refer to Step 3 in the Lab M...	main.c
TODO: Refer to step 4 in the lab ma...	main.c

Note that some coding steps require multiple lines of code to be written.

Step 1: Include the file `dsplib_def.h` and `dsplib_dsp.h` in the application. These files provide access to the digital filtering function in MPLAB XC32 DSP library. Use < > while including these files.

Step 2: Assign the audio sample frame size for our application. We perform processing on audio blocks containing 128 stereo audio samples. Assign this value to `FRAME_SIZE`.

Create an array of type `STEREO_AUDIO_DATA` and size `FRAME_SIZE`. Name it `audio` and align it on a 4 byte boundary. The following code snippet shows an example of how a variable or an array can be aligned at a 4 byte boundary.

```
int myArray[10] __attribute__((aligned(4))); // Example Only
```

Step 3: In this step you will assign values and create arrays needed for the digital filter.

1. The filter order for our application is 236. Assign this value to `FILTER_ORDER`.
2. The size of the audio frame which will be input to the filter is `FRAME_SIZE`. Assign this value to `FILTER_INPUT_SIZE`.
3. Create the arrays as described in the table below.

Array Type	Array Name	Size	Alignment	Comments
`short int`	`delayLine`	`FILTER_ORDER`	4	Used for filter delay line
`short int`	`filterOutput`	`FILTER_INPUT_SIZE`	4	Stores filter output
`short int`	`filterInput`	`FILTER_INPUT_SIZE`	4	Stores filter input
`short int`	`filterCoeffs2x`	`FILTER_ORDER * 2`	4	Needed by XC32 DSP Filter function
`extern short int`	`filterCoeffs`	`FILTER_ORDER`	-	Holds filter coefficients. Defined in `notch.c`

Step 4: Write a for loop to initialize each element of the `delayLine` array to 0. Use the index variable "`i`" to implement the for loop.

Step 5: Call the `mips_fir16_setup()` function to initialize the FIR filter. The function API is given here. The filter order is equal to FILTER_ORDER.

```
mips_fir16_setup(short int * filterCoeffs2x,
 short int * filterCoeffs, int filterOrder);
```

`filterCoeffs2x` - pointer to a `short int` array of size of (2 * `filterOrder`).
`filterCoeffs` - pointer to a `short int` array of size `filterOrder` containing the filter coefficients.
`filterOrder` - Filter order. Should be greater than 4 and a multiple of 4.

Step 6: Call the `mips_fir16()` function to perform filtering operation. The function API is given here. The filter output should be stored in `filterOutput`. The filter input is contained in `filterInput` array. The number of input samples is given by `FRAME_SIZE`. The filter order is equal to `FILTER_ORDER`. Scaling should be 0.

```
mips_fir16(short int * output, short int * input,
               short int * filterCoeffs2x,
               short int * delayLine, int nSamples,
               int filterOrder, int scaling);
```

`output` — pointer to the output array
`input` - pointer to the input array
`filterCoeffs2x` - pointer to a `short int` array of size of (2 * `filterOrder`).
`delayLine` - pointer to the delay line array.
`nSamples` - number of input samples
`filterOrder` - Filter order. Should be greater than 4 and a multiple of 4.
`scaling` - applies a scaling factor to the filter output.

Procedure: Running the code

1. Click on this icon to clean build the code:
2. Ensure that the MPLAB REAL ICE is connected to the PC and is selected in MPLAB X IDE. Refer to Appendix A for more details on how to do this.
3. Ensure that Audio Development Board is powered up and connected to REAL ICE.
4. Click on this icon to program the device.
5. You will see the board display flickers and then show the introductory message along with a filter status message. The message indicates that the filter is currently enabled.
6. Reduce headphone volume to zero . Connect the headphone to jack J6 (labeled HP OUT) on the board. Put on the headphones.

7. Increase the headphone volume gradually and speak into the board microphone from a seated position. You should hear you voice on the headphone.
8. Press switch S1 on the board. This will toggle the filter activation and is indicated by the Filter status message on the display. Toggle the filter activation and note your observations.

Observations:

If the code is working correctly, you will hear a tone along with the speech signal when the filter is OFF. This is a tone at 400Hz. When the filter is ON, the notch filter characteristic suppresses the tone from the input signal and you do not hear it.

Review the rest of the application code.

Summary:

You designed a Digital FIR Notch filter to suppress an unwanted tone in the input audio signal. You implemented the filter in an application and were able to test its performance.

LAB 1 Notes:

LAB 1 Notes:

LAB 2:
Speech Compression Algorithms.

Purpose:

To use the Speex Speech Compression Algorithm for data compression/ decompression in a voice recorder application.

Overview:

In this lab exercise you will build a voice recorder application using Microchip's PIC32MX 32-bit microcontroller. The application will use the PIC32 device RAM to store the compressed speech samples. You will use the Open Source Speex Codec to compress and decompress the speech samples. Your task will involve adding the Speex Codec to the application. You will then evaluate the amount of memory available for storage and the duration of the speech segment stored.

The Open Source Speex algorithm has multiple Narrowband (8Khz sampling) quality levels. You will start with quality level 3 (8kbps) and then try other quality levels.

Hardware:

You will use the PIC32 Audio Development Board, MPLAB REAL ICE and Headphones for this lab.

Procedure : Add Speex Codec to the application

1. Open MPLAB X IDE. If there another project open the in the IDE, close it by right-clicking on the project root folder in the Project Explorer window and then clicking on Close.
2. Click on *File->Open Project*. Browse to the class directory and open `LAB 2` folder. Double-click on `LAB 2.X`. This will open up the LAB 2 project in MPLAB X IDE.
3. In the Project Explorer window, double click on `main.c`. You will need to add code to this file in order to complete the lab. Look for the "`TODO`" comment to know where code should be added. Every such comment has a corresponding coding step associated with it. Note that some coding steps may require multiple lines of code to be written. You can also use the MPLAB X IDE Tasks Tab to locate "`TODO`" sections. The coding steps are outlined here:

Step 1: Include the `speex.h` file in the application. This allows the application to access the Speex Codec API. Add the following line to the code

```
#include "speex/speex.h"
```

Step 2: Set up the application configuration macro values to the ones shown in the table below:

Macro Name	Value	Comment
`SPEEX_DECODER_PERCEPTUAL_ENHANCER`	1	Enables perceptual enhancer in Speex Decoder
`SPEEX_ENCODER_QUALITY`	3	Speex Encoder set for 8kbps operation
`MAX_ENCODED_PACKET_SIZE`	100	Maximum size of encoded packet (in bytes)
`MEMORY_BUFFER_SIZE`	64000	Size of compressed speech memory buffer (in bytes)
`FRAME_SIZE`	160	Size of encoder input frame (in samples)

Step 3: Declare variables needed by the Speex Encoder and Decoder as shown in the table below:

Variable Type	Variable Name	Comment
`void *`	`encoder`	Will point to the Speex encoder data structure
`void *`	`decoder`	Will point to the Speex decoder data structure
`SpeexBits`	`encoderBits`	Output data structure for Speex encoder
`SpeexBits`	`decoderBits`	Output data structure for the Speex decoder.

Step 4: Call the `speex_encoder_init()` function to create the Speex encoder. The API for the function is shown below. The return value of this function should be assigned to the `void` type `encoder` pointer. The mode argument should be set to `&speex_nb_mode`.

```
void * speex_encoder_init(const SpeexMode* mode);
```

`mode` - should be set to `&speex_nb_mode` for narrow band encoder

Step 5: Call the `speex_decoder_init()` function to create the Speex decoder. The API for the function is shown below. The return value of this function should be assigned to the `void` type `decoder` pointer. The mode argument should be set to `&speex_nb_mode`.

```
void * speex_decoder_init(const SpeexMode * mode);
```

`mode` - should be set to `&speex_nb_mode` for narrow band decoder

Step 6: Call the `speex_encode_int()` function to encode the raw audio. The function API is shown here. Set `state` to `encoder`. Set the encoder input to `rawData` and set bits to point to `encoderBits`. Ignore the return value.

```
int speex_encode_int (void * state, short int * input,
                      SpeexBits *bits);
```

`state` - pointer to the initialized encoder state.
`input` - pointer to raw audio data array to be encoded
`bits` - pointer to the speex bits structure used for the encoder.

Step 7: Call the `speex_decode_int()` function to decode the encoded speex frame. The function API is shown here. Set `state` to `decoder`. Set the `bits` input to point to the `decoderBits` input. Set the decoder output to `rawData` array. Ignore the return value.

```
int speex_decode_int (void * state, SpeexBits *bits,
                      short int * output);
```

`state` - pointer to the initialized encoder state.
`bits` - pointer to the speex bits structure used for the decoder.
`output` - pointer to array raw audio data array where the decoded data will be stored.

Procedure: Running the code

1. Click on this icon to clean build the code:
2. Ensure that the MPLAB REAL ICE is connected to the PC and is selected in MPLAB X IDE. Refer to Appendix A for more details on how to do this.
3. Ensure that Audio Development Board is powered up and connected to REAL ICE.

4. Click on this icon to program the device:
5. You will see the board display flicker and then show the introductory message along with a filter status message. The message indicates that the filter is currently enabled.

6. You should have a headphone on your table. If the headphone has a volume control, reduce the volume to zero . Connect the headphone to jack J6 (labeled HP OUT) on the board. Put on the headphones.
7. Increase the headphone volume gradually and speak into the board microphone
8. Press Switch S1 on the Audio Development Board to start recording your voice. Speak into the microphone from a seated position. This speech will be recorded. You will see the 'REC' text on the display will be highlighted.
9. Press Switch S4 to start playback. The recorded voice will be played back.
10. You can press switch S1 again to repeat the record playback experiment.
11. You can now try changing the encoder quality setting (`SPEEX_ENCODER_QUALITY`) in your application to 7 or 10 and note the change in encoding quality. Note that you will have to compile, build, program and run the code.
12. You can also try disabling perceptual enhancer and note the effect this has on the decoding quality. This can be done by setting the `SPEEX_DECODER_PERCEPTUAL_ENHANCER` to 0. Note that you will have to compile, build, program and run the code.

Observations:

Quality Setting	Bit Rate	Input Bit Rate	Compression	Rating (1 to 3)
3	8 kbps	128 kbps		
7	15 kbps	128 kbps		
10	24.6kbps	128 kbps		

1. With the encoding quality set at 3, you can record up to approximately 1 minute of speech in 64K bytes of memory. This is possible due of the compression provided by the Speex algorithm. Record your observations for the other quality level in the table below. Use the rating provided to rate the quality of compression.

(**Rating:** 1– A little noisy but intelligible, 2-Good, 3-Is this really compressed?)

2. Toll quality speech is processed at 128kbps. How much toll quality speech can you store in 64K bytes of RAM?

3. How much compressed speech can you store in 64Kb RAM (specify in time duration for different quality setting).

Summary:

The Speex Codec was used to compress and decompress speech and increase the effective speech data storage capacity in a voice recorder application. The impact of speech compression and decompression was outlined.

LAB 3:
Audio Coding

Purpose:

To learn how to use the Open Source Helix MP3 Decoder to decode MP3 files while using Run Time Library Loading Technique on PIC32MX Devices.

Overview:

In this lab exercise you will use the Open Source Helix MP3 Decoder to build a USB Thumb Drive based MP3 player. You will access the Helix MP3 Decoder API through the Run Time Library Loading (RTLL) Technique. The MP3 player you will build will not be a full featured MP3 player. This was done to keep the lab exercise as simple as possible.

Given the complex nature of the involved concepts, some of the coding steps in this lab exercise are review steps. In the steps where actual coding is required, the solutions have been provided.

Concepts:

The information in this section will be needed to understand some of the steps that you will perform. In this lab, you will create two hex files. One will be for LAB 3 and the other for the Helix MP3 Decoder Library. The LAB 3 project places its code starting from program memory location `0x9D000000` to `0x9D06DFFF`. The Helix MP3 Decoder project places its code starting from location `0x9D06E000` to `0x9D07FFFF`. Refer to the Appendix B for linker script changes needed to implement this. Appendix B also contains information on the application state diagram and program memory design for this lab exercise.

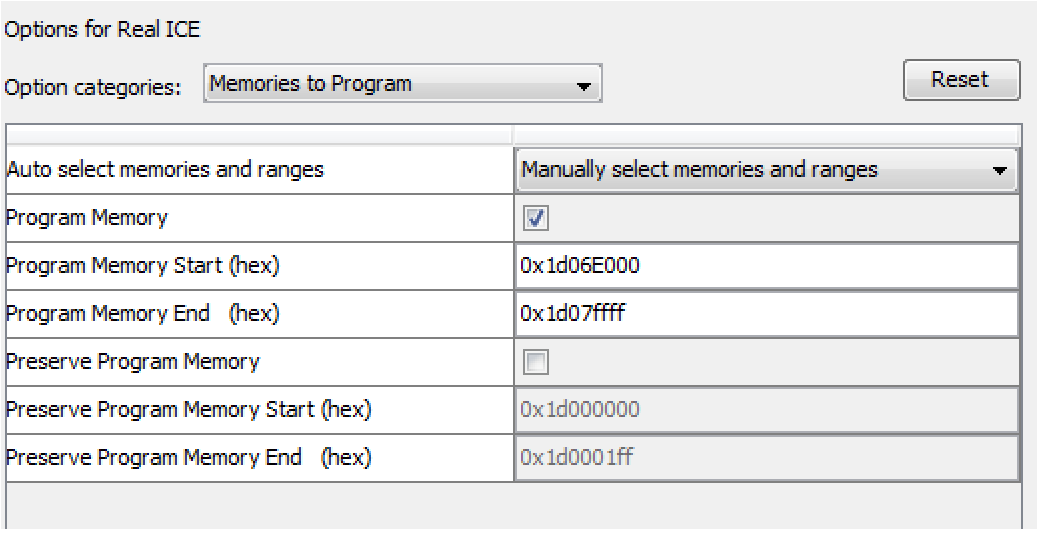
Hardware:

You will use the PIC32 Audio Development Board, MPLAB REAL ICE and Headphones for this lab.

Procedure : Build MP3 Decoder Library project and program device.

1. Open MPLAB X IDE. If a project is already open, close it by right clicking on the project root node in the Project Explorer window and click on close.
2. Click on *<u>File->Open Project</u>*. Browse to the `LAB 3\MP3` folder. Double click on `MP3.X`. This will open the MP3 project in the MPLAB X IDE.

3. The project is already configured to be built. You must configure REAL ICE to program a certain section of program memory that is relevant to the MP3 project. Go to project properties (right click on the project root node in the project explorer window and click on properties).
4. In the project properties, select REAL ICE. In Option categories, select “Memories to Program”. Choose the “Manually Select memories and ranges” option and enter data as shown in the figure below. Note that you are entering the physical (and not virtual) memory addresses. Click apply to save changes.



Options for Real ICE

Option categories: Memories to Program

Auto select memories and ranges	Manually select memories and ranges
Program Memory	☑
Program Memory Start (hex)	0x1d06E000
Program Memory End (hex)	0x1d07ffff
Preserve Program Memory	☐
Preserve Program Memory Start (hex)	0x1d000000
Preserve Program Memory End (hex)	0x1d0001ff

5. Click on this icon to clean build the project:
6. Note that you may see a linker warning about the reset symbol not being found. You can safely ignore this warning.
7. Ensure that the Audio Development Board is connected to the REAL ICE
8. Click on this icon to program the device:
9. When the programming is complete, close the project. (right-click on the project root code and select Close).

Procedure: Complete MP3 player application and program.

1. Click on *File->Open*. Browse to the `LAB 3` folder. Double click on `LAB 3.X`. This will open the LAB 3 project in the MPLAB X IDE.
2. In the Project Explorer window, double click on `AudioUSBTasks.c`. You will need to add code to this file in order to complete the lab. Look for the “`TODO`” comment to know where code should be added. Every such comment has a corresponding coding or review step associated with it. Note that some coding steps may require multiple lines of code to be written. The coding steps are outlined here:

Step 1: Assign value and declare some of the variables needed by the Helix MP3 Decoder.

Declare a macro named `READBUF_SIZE`. This defines the maximum data chunk size (in bytes) that should be read from the MP3 file. Assign a value of 1940 to it.

Declare an array of type `BYTE` and name it `readBuf`. Its size should be `READBUF_SIZE`. This array will hold the MP3 data read from the MP3 file.

Declare a pointer of type `BYTE` and name it `readPtr`.

Declare an array of type `INT16` and name `outBuf`. Its size should (1152 * 2). This array will hold the decoded audio data. The multiply by two indicates space allocation for stereo operation. The value 1152 holds good for operation at 44.1KHz

Step 2: Declare a `void` pointer and name it `hMP3Decoder`. This would serve as the pointer to the Helix MP3 Decoder state data structure.

Step 3: Create a pointer to the MP3 library. Declare a `void` pointer and name it `hMP3DecoderLibrary`. This will actually point to a data structure (called the Module Header) which contains information about the Helix MP3 Library.

Step 4: This is a code review step. You don't have to add code in this step. As described in Step 3, This section contains prototype declarations of the exported Helix library functions. Each prototype declaration declares a function pointer.

Step 5: This is a coding step. Use the RTLL function, `dlopen()` to obtain a handle to the MP3 decoder library. Store the return value (the library handle) in `hMP3DecoderLibrary`. The API description is given here. The library string for the Helix MP3 library is “`Helix Library`” (case-sensitive). The library module header address is `0x9D06E000`. Type cast the library module header address using `void *` type to avoid compiler warning. Set the data parameter to zero.

```
 handle = dlopen(const char * libraryString,
void * libModuleHeaderAddress, void * data);
```

`libraryString`	- Library identifier string contained in the library module header
`libModuleHeaderAddress`	- Absolute address of the library module header in program memory.
`data`	- Optional data as specified by the library.

The solution for this step in available on page 18

Step 6: This is a coding step. Use the RTLL `dlsym()` function to obtain a handle to each function exported by the library. The API description is given here. The library handle should be set to `hMP3DecoderLibrary`. The function description string depends on the function to which you are trying to obtain the handle.

```
functionHandle = dlsym(void * libraryHandle,
     const char * functionDescString);
```

`libraryHandle`	- Library handle obtained from the `dlopen()` function
`functionDescString`	- String key of the desired function (defined in the library module header)

For example, the function description string for MP3InitDecoder function is "`HelixMPInitDecoder`". The coding step already has some examples of how the handles are obtained. Use the table below to obtain handles to the rest of exported functions.

Return Value Variable	String Description	Comment
`MP3FreeDecoder`	`"HelixMP3FreeDecoder"`	Will free the MP3 Decoder memory
`MP3Decode`	`"HelixMP3Decode"`	Decodes a MP3 frame
`MP3GetLastFrameInfo`	`"HelixMP3GetLastFrameInfo"`	Get information about the last frame

The solution to this step is given on page 18

Step 7: This is a coding step. Call the `MP3InitDecoder()` function. The function takes zero arguments and returns a handle to the initialized MP3 decoder state data structure. Store the return value in `hMP3Decoder`.

Step 8: This is a coding step. Call the `MP3FindSyncWord()` function . The function takes 2 arguments and returns an `int` type offset of the byte location where the synchronization word was found in the current frame. The first argument is `readPtr`. The second argument is `bytesLeft`. Store the return value in `offset`.

```
int MP3FindSyncWord(unsigned char *buf, int nBytes);
```

`buf`	- pointer to the input frame
`nBytes`	- size of the frame.

<u>Step 9</u>: This is a coding step. Call the `MP3Decode()` function. The function takes 5 arguments and returns a `int` type error value. The first argument should be set to `hMP3Decoder`. The second argument should be set to `&readPtr`. The third argument is pointer to `&bytesLeft`. The fourth argument is the `outBuf` array where the decoded data will be stored. The fifth argument should be `0`. Store the return value in `err`.

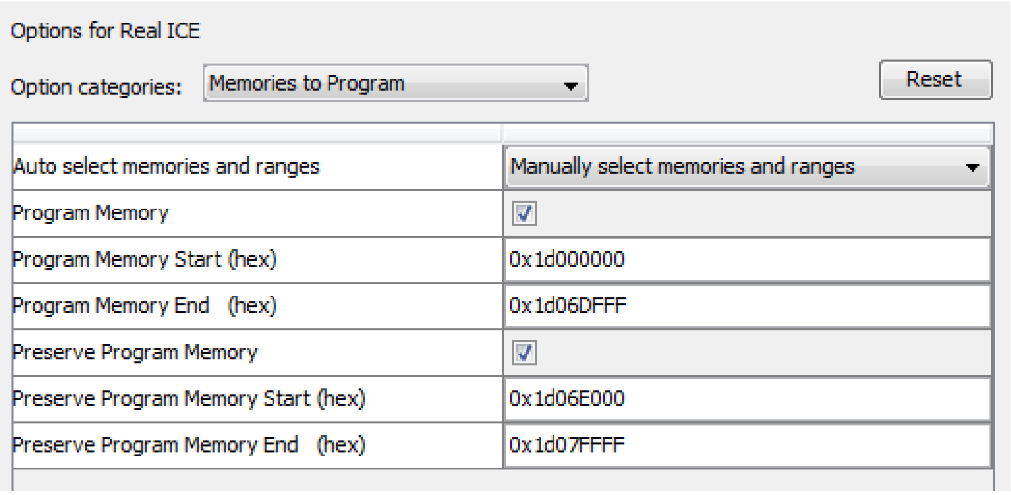
```
int MP3Decode(HMP3Decoder hMP3Decoder, unsigned char **inbuf,
        int *bytesLeft, short *outbuf, int useSize);
```

`hMP3Decoder`	- pointer to the MP3 decoder state data structure obtained by using the `MP3InitDecoder()` function
`inbuff`	- modifiable pointer to the input frame
`bytesLeft`	- indicates the number of un-processed bytes in the current frame
`outbuf`	- pointer to the output raw audio buffer.
`useSize`	- should be 0

The solution for this step is given on Page 18

The code is now ready to be built. Perform the following steps to program the device:

1. You must configure REAL ICE to program a certain section of program memory that is relevant to the LAB 3 project and not erase the program memory that contains the MP3 library code. Go to project properties (right click on the project root node in the project explorer window and click on properties).
2. In the project properties, select REAL ICE. In Option categories, select Memories to Program. Choose the Manually Select memories and ranges option and enter data and shown in the following figure. Note that you are entering the physical (and not virtual) memory addresses. Click apply to save changes.



Option categories: Memories to Program

Auto select memories and ranges	Manually select memories and ranges
Program Memory	☑
Program Memory Start (hex)	0x1d000000
Program Memory End (hex)	0x1d06DFFF
Preserve Program Memory	☑
Preserve Program Memory Start (hex)	0x1d06E000
Preserve Program Memory End (hex)	0x1d07FFFF

3. Click on this icon to clean build the project:
4. Note that you may see a warning regarding MDD-FS read attribute. This can be safely ignored.
5. Ensure that the Audio Development Board is connected to the REAL ICE
6. Click on this icon to program the code:
7. Copy the file `test.mp3` from the `LAB 3` folder on to your Thumb Drive. Then insert the Thumb Drive onto the Host connector on the PIC32 Audio Development Board.
8. Connect the headphone to jack J6 (labeled HP OUT) on the board. Put on the headphones. Increase the headphone volume gradually . You should hear the MP3 file playback on the headphone.

Summary:

The Helix MP3 Decoder Library was integrated into the application via the RTLL framework. The library API was invoked using RTLL framework. Some of the key library function were coded and the application was tested. Microchip's AN1367 - "Porting the Helix MP3 Decoder onto Microchip's PIC32MX 32-bit Microcontroller" provides more details on Helix MP3 Decoder API and Run Time Library Loading Technique.

Solutions:

The solutions for the different coding steps are given here

Step 5:

```
  hMP3DecoderLibrary = dlopen("Helix Library",
            (void*)HELIX_LIB_MODULE_HEADER_ADDRESS,(void *)0);
```

Step 6:

```
MP3InitDecoder      = dlsym(hMP3DecoderLibrary, "HelixMP3InitDecoder");
MP3FreeDecoder      = dlsym(hMP3DecoderLibrary, "HelixMP3FreeDecoder");
MP3Decode           = dlsym(hMP3DecoderLibrary, "HelixMP3Decode");
MP3GetLastFrameInfo = dlsym(hMP3DecoderLibrary, "HelixMP3GetLastFrameInfo");
MP3GetNextFrameInfo = dlsym(hMP3DecoderLibrary, "HelixMP3GetNextFrameInfo");
MP3FindSyncWord     = dlsym(hMP3DecoderLibrary, "HelixMP3FindSyncWord");
RegisterMalloc      = dlsym(hMP3DecoderLibrary, "HelixMP3RegMalloc");
RegisterFree        = dlsym(hMP3DecoderLibrary, "HelixMP3RegFree");
```

Step 9:

```
err = MP3Decode(hMP3Decoder, &readPtr, &bytesLeft, outBuf, 0);
```

Helix MP3 Library API Summary:

Decoder Initialization:

```
HMP3Decoder MP3InitDecoder(void);
```

This function returns a pointer to the initialized MP3 Decoder State Data Structure.

Decoder de-allocation:

```
void MP3FreeDecoder(HMP3Decoder hMP3Decoder);
```

This function de-allocates memory allocated to the `hMP3Decoder` state memory.

MP3 Decode:

```
int MP3Decode(HMP3Decoder hMP3Decoder, unsigned char **inbuf,
         int *bytesLeft, short *outbuf, int useSize);
```

This function decodes an MP3 frame. `inbuf` points to the input frame and is modified to point to the last consumed byte in the input frame. `bytesLeft` indicates the number of bytes left to be consumed in the input frame. `outBuf` will contain the decoded audio samples. `useSize` should be 0. This function returns a non-zero error code.

Get Last Frame Information:

```
void MP3GetLastFrameInfo(HMP3Decoder hMP3Decoder, MP3FrameInfo
                        *mp3FrameInfo);
```

This function returns the MP3FrameInfo frame information data for the last decoded frame.

Get Next Frame Information:

```
int MP3GetNextFrameInfo(HMP3Decoder hMP3Decoder, MP3FrameInfo
             *mp3FrameInfo, unsigned char *buf);
```

This function returns the MP3FrameInfo frame information data for the frame pointed to by `buf`.

Find MP3 Frame Synchronization word:

```
int MP3FindSyncWord(unsigned char *buf, int nBytes);
```

This function return the byte offset location of the frame synchronization word if it is found in the current frame. Return -1 otherwise.

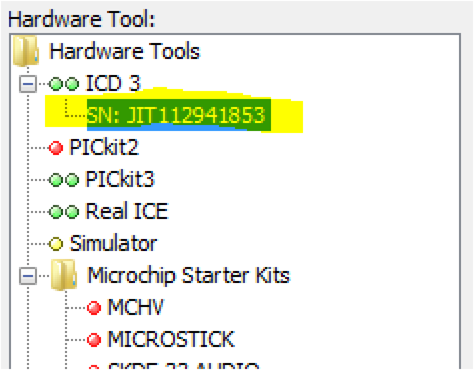
LAB 3 Notes:

Appendix A:

Enabling REAL ICE or ICD3 in MPLAX X IDE

This section describes the steps to be followed to enable REAL ICE or ICD3 in MPLAB X IDE project. You need to have a open project and the REAL ICE or ICD3 should be connected to the PC.

1. Right click on the root node of the project tree in the Project Explorer in MPLAB X IDE and click on Properties.
2. In the Project Properties window, select the desired tool in the Hardware Tool section by clicking on the tool serial number. For example, the figure below shows an ICD3 with serial no JIT112941853 being selected. Note that serial number on your REAL ICE or ICD3 will be different.



3. Click on Apply

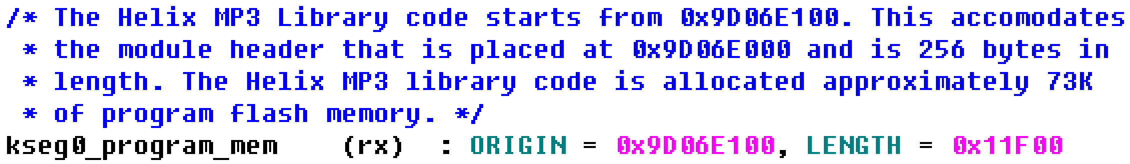
Appendix B:

LAB3: Linker Script Changes, Program Memory Layout and Application State Diagram

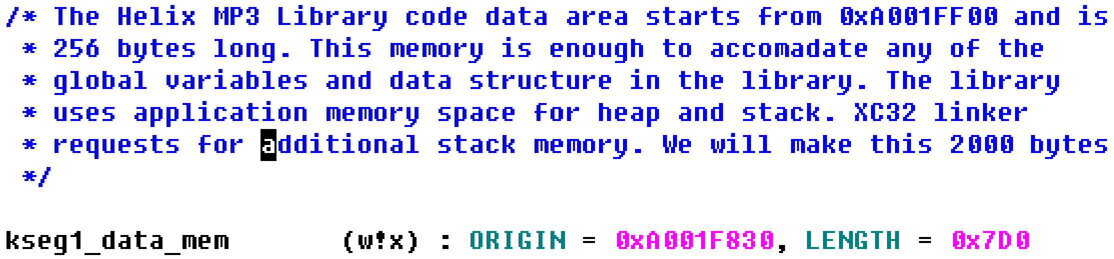
Linker Script Changes:

LAB 3 contains two projects, the Helix MP3 Decoder Project and the LAB 3 application project. Each project uses it's own linker script file. The linker script files have been modified to implement the program memory layout described in the Program Memory Layout section.

The linker script for the Helix MP3 Decoder Project is located at `\LAB 3\MP3\MP3.X\procdefs.ld`. A review of this file will show that the program memory start location (`kseg0_program_memory`) has been modified to start at location `0x9D06E100`. This provides approximately 73KB of program memory for the library code and accommodates the library module header that is located at 0x9D06E000. The data memory (`kseg1_data_mem`) is configured to start at location 0xA001F830. The length of the data memory is 2000 bytes. Note that the library uses the application heap and stack. The code snippet below show the library linker script snippet.



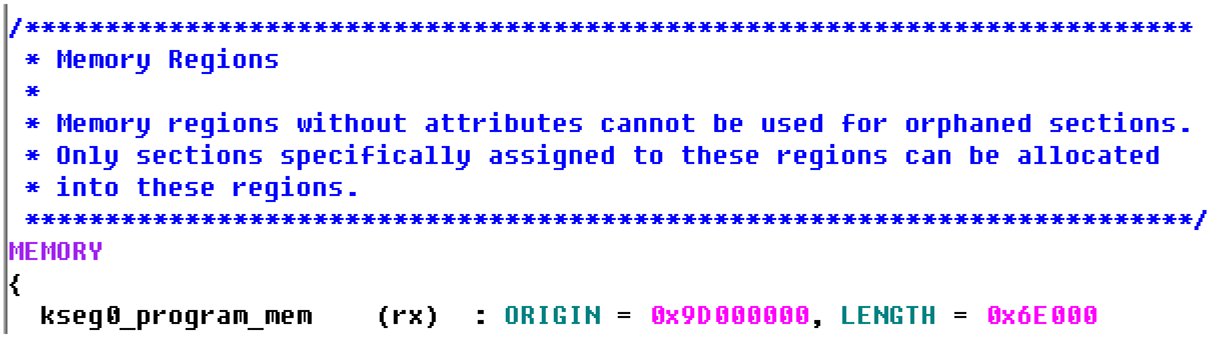
```
/* The Helix MP3 Library code starts from 0x9D06E100. This accomodates
 * the module header that is placed at 0x9D06E000 and is 256 bytes in
 * length. The Helix MP3 library code is allocated approximately 73K
 * of program flash memory. */
kseg0_program_mem    (rx)  : ORIGIN = 0x9D06E100, LENGTH = 0x11F00
```



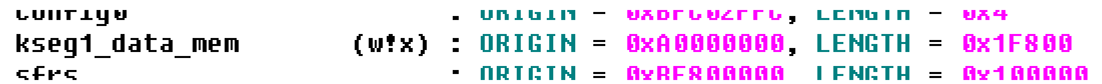
```
/* The Helix MP3 Library code data area starts from 0xA001FF00 and is
 * 256 bytes long. This memory is enough to accomadate any of the
 * global variables and data structure in the library. The library
 * uses application memory space for heap and stack. XC32 linker
 * requests for additional stack memory. We will make this 2000 bytes
 */

kseg1_data_mem       (w!x) : ORIGIN = 0xA001F830, LENGTH = 0x7D0
```

The linker script for the LAB 3 Application Project is located at `\LAB 3\MP3\LAB 3.X\procdefs.ld`. A review of this file will show that the program memory length (`kseg0_program_memory`) has been modified to end 0X9D06DFFF. This accommodates the library header that is located at 0x9D06E000. The data memory (`kseg1_data_mem`) is configured to end at 0xA001F800. This provides memory for the library data section.

The following code snippet shows the relevant sections of the LAB3 application project Linker script.



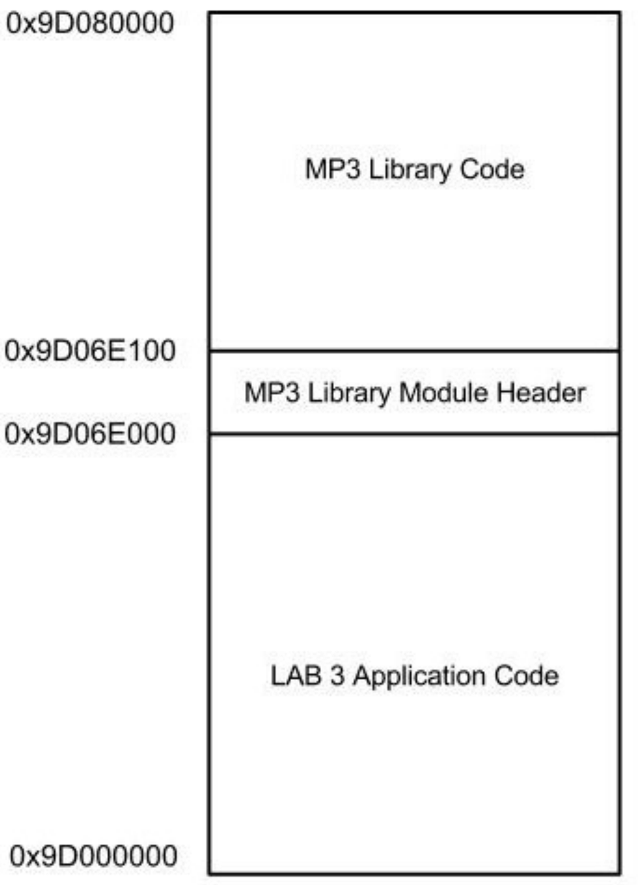
```
/*************************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for orphaned sections.
 * Only sections specifically assigned to these regions can be allocated
 * into these regions.
 *************************************************************************/
MEMORY
{
  kseg0_program_mem    (rx)  : ORIGIN = 0x9D000000, LENGTH = 0x6E000
```



```
 config0                   : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
 kseg1_data_mem       (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x1F800
 sfrs                      : ORIGIN = 0xBF800000, LENGTH = 0x100000
```

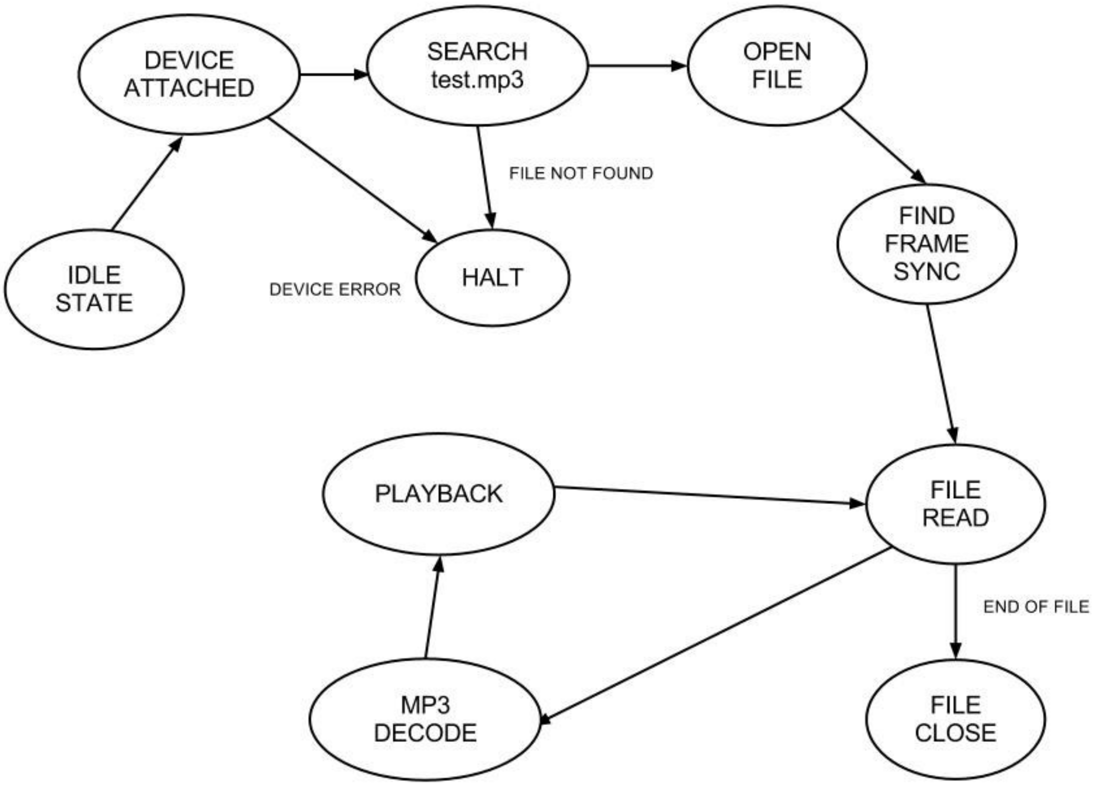
Program Memory Design:

The following figure show the allocation of different program sections by the LAB 3 project and the Helix MP3 Decoder Project



Application State Diagram:

The following diagram shows the various states of the application. This state machine is implemented in the `AudioUSBTasks()` function in `AudioUSBTasks.c` file contained in the LAB 3 project.


DEVICE ATTACHED
SEARCH test.mp3
OPEN FILE
FILE NOT FOUND
FIND FRAME SYNC
IDLE STATE
DEVICE ERROR
HALT
PLAYBACK
FILE READ
END OF FILE
MP3 DECODE
FILE CLOSE