# VS1053B / VS1063A EQUALIZER

## VSMPG "VLSI Solution Audio Decoder"

Project Code:
Project Name: VSMPG

**All information in this document is provided as-is without warranty. Features are subject to change without notice.**

| Revision History | | | |
|---|---|---|---|
| **Rev.** | **Date** | **Author** | **Description** |
| 1.50 | 2016-02-26 | HH | Added EarSpeaker and Sine Generator support. WriteUartReg() and ReadUartReg() replaced with more general WriteUart() and ReadUart(). |
| 1.41 | 2016-02-22 | HH | Fixed ReadUartReg() prototype (Chapter 8.2.5). |
| 1.40 | 2016-02-18 | HH | Added UART controls (Chapter 8.2). |
| 1.30 | 2015-08-27 | HH | Added mono microphone input option. |
| 1.20 | 2015-06-18 | HH | Added Chapter 10, Standalone Player. Plugin version is unchanged since v1.10. |
| 1.11 | 2013-12-19 | HH | Added Chapter 11, Frequency Responses. Software is the same as in v1.10. |
| 1.10 | 2013-10-25 | HH | Added 7-Band Equalizer Designer, Chapter 4. |
| 1.01 | 2013-10-24 | HH | Corrected error in Example 2 in Chapter 3.2. |
| 1.00 | 2013-10-23 | HH | Initial release. |

# Contents

# List of Figures

# 1 Introduction

This document is an instruction manual on how to use the VS1053b / VS1063a Equalizer.

Chapter 2 describes the application. Chapter 3 show how to set it up and run it.

The filters can be used in two different ways: either by using a 7-Band Equalizer Designer as shown in Chapter 4, or by designing filters manually as shown in Chapter 5.

The optional UART protocol is presented and explained in Chapter 6.

How to use the Sine Generator and EarSpeaker, is shown in Chapter 7.

Support functions needed to use the application are shown in Chapter 8.

Chapter 9 tells how to load and start the plugin.

A tutorial on how to use the standalone version of the application is told in Chapter 10.

Chapter 11 presents the filter frequency responses.

Chapter 12 tells how to load plugin code to a VS10XX chip.

Finally, Chapter 13 contains VLSI Solution's contact information.

# 2 The VS1053b / VS1063a Equalizer

The *VS1053b / VS1063a Equalizer* application allows the VS10x3 IC to act as a multi-channel equalizer.

When running the application, other functions of VS10x3 is not available.

Some key features of The Equalizer are:

- Runs on VS1053b, VS1063a, and their variants VS8053b, VS8063a, and VS1163a.
- Runs at 48 kHz.
- Audio delay approximately 3 ms.
- Selectable stereo line input / mono microphone input.
- Upto 28 / 22 equalization filters available for VS1063a / VS1053b, respectively. Note: A stereo channel uses two filters.
- Alternatively, a 7-Band Equalizer Designer can be used.
- Can be configured through SPI, UART, or by doing binary patch to image.
- Multi-IC binary, compatible with both VS1053b and VS1063a.
- The application sets VS10x3 to its maximum in-spec clock rate.

The VS1053b / VS1063a Equalizer is available as an application, downloadable at *http://www.vlsi.fi/en/support/software/vs10xxapplications.html* .

## 2.1 Limitations and Requirements

- For the sample rate to be exactly 48 kHz, the crystal needs to be 12.288 MHz. With other clocks, the internal sample rate is $f_s = \frac{48 \times c}{12.288}$ where $c$ is the crystal frequency in MHz.
- If the user tries to define more filters than what can be calculated (28 for VS1063b, 22 for VS1053b), only the ones first in the list will be used.
- Using EarSpeaker will decrease the maximum number of filters by 8.
- Using the Sine Generator will decrease the maximum number of filters by 2.

## 2.2 Use Case Scenarios

The VS1053b / VS1063a Equalizer can be run and controlled in several different ways. It can be controlled through SPI or UART by a microcontroller, or it can operate as a standalone unit with preset and non-modifiable parameters.

### 2.2.1 VS10XX and Microcontroller, SPI Connection

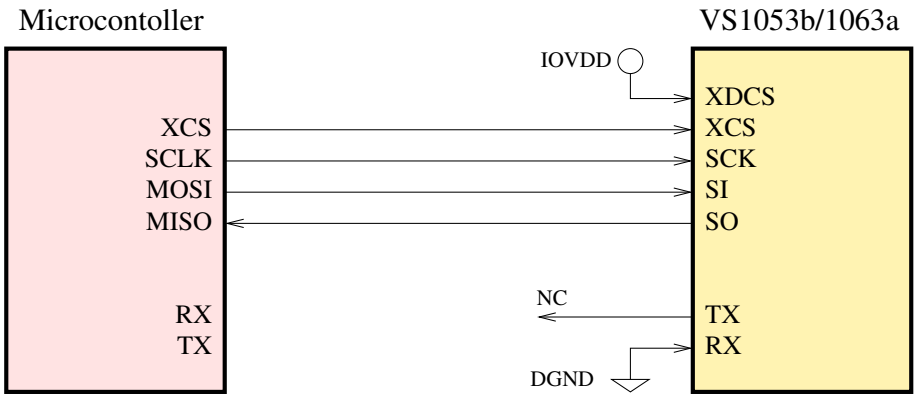


Figure 1: Equalizer loaded and controlled through SPI.

Figure 1 shows a configuration where the Equalizer application is loaded and controlled through SPI by a microcontroller.

To use the Equalizer in this way, read the following:

- Chapter 3, *Running VS1053b / VS1063a Equalizer*
- Chapter 4, *Using the 7-Band Equalizer Designer*
- Chapter 5, *Setting Filters Manually*
- Chapter 8.1, *SPI Functions*
- Chapter 9, *Loading and Starting the Plugin*

### 2.2.2 VS10XX, Microcontroller, and SPI Boot Memory, UART Connection

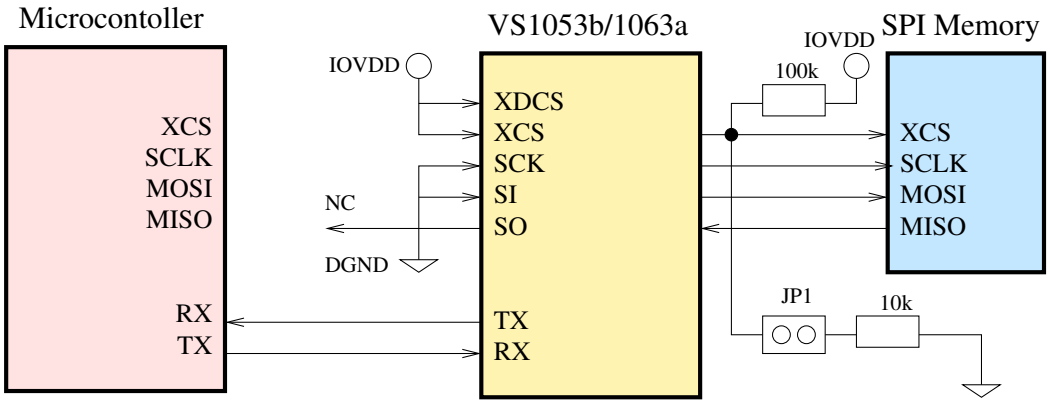


Figure 2: Equalizer boots from SPI memory and is controlled by UART.

Figure 2 shows a configuration where VS10XX boots the Equalizer from external SPI Memory. The memory must be of a type that is compliant with the requirements of VS1053b/VS1063a Datasheet Chapter *SPI Boot*.

To program the SPI Memory you either need a separate programmer, or you can use VS1053b/VS1063a to program it. In the latter case you need to be able to redirect UART TX/RX control to a PC.

To use the Equalizer in this way, read the following:

- Chapter 3, *Running VS1053b / VS1063a Equalizer*
- Chapter 4, *Using the 7-Band Equalizer Designer*
- Chapter 5, *Setting Filters Manually*
- Chapter 8.2, *UART Functions*
- Chapter 10, *Programming and Running the Standalone Version*. If you set all your Equalizer parameters through UART, you may ignore the parts that deal with modifying the boot image.

### 2.2.3 VS10XX and SPI Boot Memory, No Modifiable Parameters

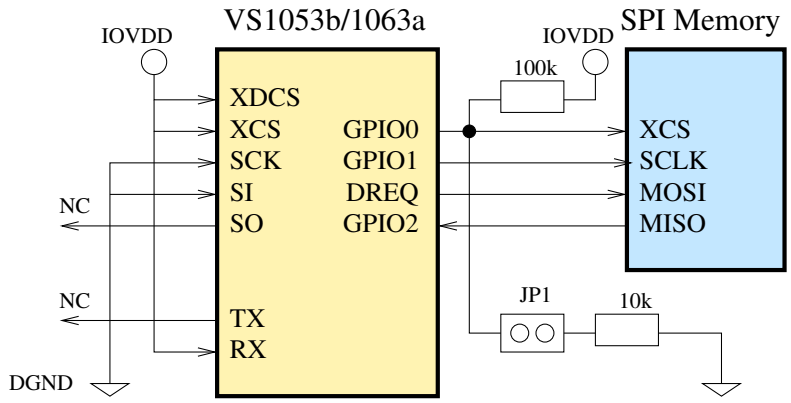


Figure 3: Equalizer boots from SPI memory with preset parameters.

Figure 3 shows how to use Equalizer with preset parameters, booting from external SPI memory.

To program the SPI Memory you either need a separate programmer, or you can use VS1053b/VS1063a to program it. In the latter case you need to be able to redirect UART TX/RX control to a PC.

To use the Equalizer in this way, read the following:

- Chapter 10, *Programming and Running the Standalone Version*

# 3 Running VS1053b / VS1063a Equalizer

There are several SCI (SPI) registers that may be read by the user, presented in Chapter 3.2. To see how to control the SCI registers, read the VS1053b or VS1063a Datasheet, or see the support functions in Chapter 8. Alternatively, the SCI registers may be controlled through the UART (Chapter 6, *UART Protocol*).

The equalizer filters can be set in two ways.

Chapter 4 shows how to set the equalized using the automatic 7-Band Equalizer Designer. The designer uses upto 26 filter channels to create an equalizer that would have the best available audio quality and interband rejection. This is the recommended method for users without a deep signal processing or equalizer background.

Chapter 5 explains how to set individual filters manually. Manual setting is only recommended for users that want to realize a filter system that cannot be created with the 7-Band Equalizer Designer.

## 3.1 Selecting VS1053b / VS1063a Equalizer Input

The equalizer input is selected with register SCI_MODE (0) bit SM_LINE (14).

If SM_LINE is set (default), a stereo line input is read through single-ended input pins LINE1 and LINE2.

If SM_LINE is cleared, a mono microphone input is read through differential input pins MICN and MICP.

### 3.2 VS1053b / VS1063a Equalizer Read/Write Registers

After starting the equalizer, the following registers can be read through SCI by the user:

| Register | R/W | Bits | Description |
|---|---|---|---|
| SCI_AICTRL0 | R | 15:8<br>7:0 | Always 0.<br>Current internal buffer fill state (debug). |
| SCI_AICTRL1 | R | 15:1<br>0 | Always 0.<br>0 = Enough CPU, 1 = internal overflow.<br>Register may be cleared by the user. |
| SCI_AICTRL2 | W | 15:13<br>12:0 | Not used.<br>Sine Generator frequency: $f = 5.859375 \times n$ Hz. |
| SCI_AICTRL3 | R | 15:8<br>7:0 | Maximum number of filters available with this IC.<br>Current number of filters used. |

Note that the current number of filters used may be higher than the maximum number of filters. If this is the case, only the first filters are actually applied to the signal.

To get the register values through the UART interface, send '!'. The reply will be four hexadecimal numbers that contain the contents of SCI_AICTRL0 through SCI_AICTRL3. Issuing this UART command will clear the overflow bit 0 of SCI_AICTRL1.

To write to these registers through UART, issue the command "Xaaaabbbb" where "aaaa" is "c00c" through "c00f" for SCI_AICTRL0 through SCI_AICTRL3, respectively, and bbbb is a four-digit hexadecimal value.

### 3.3 Processing Capability

The table below shows the maximum number of filters available in different conditions:

| Ear-Speaker | Sine Gen. | VS1053b filters | VS1063a filters |
|---|---|---|---|
| | | 22 | 26 |
| | YES | 20 | 24 |
| YES | | 14 | 18 |
| YES | YES | 12[1] | 16 |

[1] Note that if using the 7-Band Equalizer Designer (Chapter 4), the highest band (15600 Hz) can not be equalized in this case.

# 4 Using the 7-Band Equalizer Designer

When the 7-Band Equalizer Designer is used, the memory locations that the application needs to write to are as follows:

| Function | X Mem Address |
|---|---|
| 64 Hz amplification in dB*256, signed two's complement | 0x1800 |
| 160 Hz amplification in dB*256, signed two's complement | 0x1801 |
| 400 Hz amplification in dB*256, signed two's complement | 0x1802 |
| Set to 0 | 0x1803 |
| 1000 Hz amplification in dB*256, signed two's complement | 0x1804 |
| 2500 Hz amplification in dB*256, signed two's complement | 0x1805 |
| 6250 Hz amplification in dB*256, signed two's complement | 0x1806 |
| Set to 0 | 0x1807 |
| 15600 Hz amplification in dB*256, signed two's complement | 0x1808 |
| Set to 0 | 0x1809 |
| Set to 0 | 0x180A |
| Set to 8 (activates designer) | 0x180B |

The user first sets the requested amplifications (recommended range -36...+18 dB), then activates the designer. The designer will create 26 filters to implement the 7-band equalizer as well as possible (14 filters for the main bands, 12 for in-between bands). If the IC in question is incapable of handling the full amount of filters, then some of the less important "in-between" filters are dropped first.

When the 7-Band Equalizer Designer is used, any previous filters designed manually (Chapter 5) will be removed. The filters the designer has created will be located in memory locations 19...31.

Example:
To set a 7-band filter which greatly emphasizes bass and treble, the following code could be used:

```
//                      64   160   400  1000  2500  6250 15600 Hz
u_int16 dBTab[7] = {12.0,  5.0,  1.0, -1.5, -3.0,  2.0,  6.0};

Set7Chan(dBTab);
```

Implementations for Set7Chan() are provided in Chapters 8.1.1 and 8.2.1.

# 5 Setting Filters Manually

There are 32 filter memory locations available to the user. When the application is started, all memory location are cleared. The memory locations are located in data memory as follows:

| Memory Location | X Address |
|---|---|
| 0 | 0x1800 |
| 1 | 0x1804 |
| 2 | 0x1808 |
| 3 | 0x180c |
| ... | ... |
| 31 | 0x187c |

Each filter memory location contains the following fields which may be set by the user:

| Offset | Bits | Description |
|---|---|---|
| 0 | 15:0 | Filter center frequency (Hz).<br>Recommended range is 20...20000 Hz. |
| 1 | 15:0 | Filter amplification (dB * 256, signed two's complement).<br>Recommended range is -36...+18 dB. |
| 2 | 15:0 | Filter Q factor (Q * 256, unsigned).<br>Recommended range is 0.1...10. |
| 3 | 15:4 | Unused |
| 3 | 3 | Reserved, set to 0. |
| 3 | 2 | 1 = Apply filter to left channel. |
| 3 | 1 | 1 = Apply filter to right channel. |
| 3 | 0 | 1 = Update filter. This needs to be clear before new values can be written to, and must be set by the user. See examples below. |

Example 1:
To set a 12.5 dB lifting filter at 2000 Hz with a Q factor of 2 to both channels at filter memory location 3, do the following:

```
//         mem    Hz    dB    Q  l  r
SetFilter(  3, 2000, 12.5, 2.0, 1, 1);
```

Example 2:
To turn the previous filter off, do the following:

```
//         mem  Hz   dB    Q  l  r
SetFilter(  3,  0, 0.0, 0.0, 0, 0); // Only last two zeroes are significant
```

Implementations for SetFilter() are provided in Chapters 8.1.2 and 8.2.2.

# 6 UART Protocol

In addition to VS1053b/VS1063a SPI registers, The Equalizer can be controlled through UART, running at 9600/8N1 (9600 bps, 8 data bits, no parity, one stop bit). The UART protocol is as follows.

To write to an X Memory address, send the byte sequence “Xaaaadddd” where “aaaa” is the X Memory address in four-digit hexadecimal format, and “dddd” is the data in four-digit hexadecimal format.

The command for reading the data in an X memory address is “xaaaa”. VS10xx answers with “dddd”.

To write/read Y memory (not required by the Equalizer), replace ’X’/’x’ with ’Y’/’y’.

To set stereo line input (default), send “l” (lowercase L). For mono microphone input, send “m”. To get a report as described in Chapter 3.2, send “!”.

For any unknown command characters, the reply is “?”.

The SCI/SPI registers lie at the following X memory addresses. Note that all registers don’t work as expected when accessed through the UART.

| Register name | X Mem Address |
|---|---|
| SCI_MODE | 0xc000 |
| SCI_STATUS | 0xc001 |
| SCI_BASS | 0xc002 |
| SCI_CLOCKF | 0xc003 |
| SCI_DECODE_TIME | 0xc004 |
| SCI_AUDATA | 0xc005 |
| SCI_WRAM | 0xc006 |
| SCI_WRAMADDR | 0xc007 |
| SCI_HDAT0 | 0xc008 |
| SCI_HDAT1 | 0xc009 |
| SCI_AIADDR | 0xc00a |
| SCI_VOL | 0xc00b |
| SCI_AICTRL0 | 0xc00c |
| SCI_AICTRL1 | 0xc00d |
| SCI_AICTRL2 | 0xc00e |
| SCI_AICTRL3 | 0xc00f |

# 7 Extra Functionality

## 7.1 Sine Generator

For calibration purposes, A Sine Generator is provided with the Equalizer. A sine tone at a level of -18 dB of maximum is put through the Equalizer filters.

To activate the generator, write a non-zero value to register SCI_AICTRL2. The frequency $f$ of the sine is

$f = n \times \frac{48000}{8192}$ Hz

where $n$ is SCI_AICTRL bits 11:0. The usable value range is from 6 through 20000 Hz, or 1 through 3413.

Some useful values can be found in the table below:

| Frequency | Value |
|---|---|
| 64 Hz | 11 |
| 160 Hz | 27 |
| 400 Hz | 68 |
| 1000 Hz | 171 |
| 2500 Hz | 427 |
| 6250 Hz | 1067 |
| 15600 Hz | 2662 |

Example for UART users:
To set the sine generator at 1 kHz (value 171 = 0xab), send the following sequence (SCI_AICTRL2 register is at X address 0xc00e): “Xc00e00ab”

## 7.2 Earspeaker

EarSpeaker spatial processing for earphones can be activated along with the equalizer. To activate it, use the same registers that you would when using the device normally.

With VS1053, Earspeaker is configured with two bits in the SCI_MODE register. For details, see the VS1053b Datasheet. SPI users may write directly to SCI_MODE. UART users can write to SCI_MODE with the sequence “Xc000zzzz” where zzzz is a four-digit hexadecimal value.

With VS1063, Earspeaker is configured with variable earSpeakerLevel that lies in X Memory address 0x1e1e. For details, see the VS1053b Datasheet. SPI users must first write 0x1e1e to SCI_WRAMADDR, then the level value (or 0 to disable) to SCI_WRAM. UART users may use the sequence “X1e1ezzzz” where zzzz is a four-digit hexadecimal value for earSpeakerLevel.

# 8 Support Functions

This chapter provides implementations for functions used in this document.

## 8.1 SPI Functions

These are the support functions needed for SCI / SPI operations.

For details on how the SCI registers work, read the VS1053b or VS1063a Datasheet.

### 8.1.1 Function Set7Chan() SPI Implementation

```
#define SCI_WRAMADDR 7
#define SCI_WRAM 6

// Parameter is a 7-long vector of dB values for
// 64, 160, 400, 1000, 2500, 6250, and 15600 Hz
void Set7Chan(const double *dB) {
  // Check that a previous filter update is not going on,
  // and wait if necessary.
  do {
    WriteSci(SCI_WRAMADDR, 0x180B);
  } while (ReadSci(SCI_WRAM) & 8);

  // Set new equalizer values
  WriteSci(SCI_WRAMADDR, 0x1800);
  WriteSci(SCI_WRAM, (s_int16)(dB[0]*256.0));
  WriteSci(SCI_WRAM, (s_int16)(dB[1]*256.0));
  WriteSci(SCI_WRAM, (s_int16)(dB[2]*256.0));
  WriteSci(SCI_WRAM, 0);
  WriteSci(SCI_WRAM, (s_int16)(dB[3]*256.0));
  WriteSci(SCI_WRAM, (s_int16)(dB[4]*256.0));
  WriteSci(SCI_WRAM, (s_int16)(dB[5]*256.0));
  WriteSci(SCI_WRAM, 0);
  WriteSci(SCI_WRAM, (s_int16)(dB[6]*256.0));
  WriteSci(SCI_WRAM, 0);
  WriteSci(SCI_WRAM, 0);
  WriteSci(SCI_WRAM, 8);  // Activate filter designer
}
```

### 8.1.2 Function SetFilter() SPI Implementation

```
#define SCI_WRAMADDR 7
#define SCI_WRAM 6

// Sets one manual filter.
void SetFilter(u_int16 memoryLocation, u_int16 freqHz, double dB, double q,
               s_int16 left, s_int16 right) {
  // Check that a previous filter update is not going on,
  // and wait if necessary.
  do {
    WriteSci(SCI_WRAMADDR, 0x1800 + memoryLocation*4 + 3);
  } while (ReadSci(SCI_WRAM) & 1);

  // Do our filter update
  WriteSci(SCI_WRAMADDR, 0x1800 + memoryLocation*4);
  WriteSci(SCI_WRAM, freqHz);
  WriteSci(SCI_WRAM, (s_int16)(dB*256.0));
  WriteSci(SCI_WRAM, (u_int16)( q*256.0));
  WriteSci(SCI_WRAM, (left?4:0)|(right?2:0)|1); // Left, right, update
}
```

### 8.1.3 Function ReadFilters() SPI Implementation

```
// This pseudo-code shows how to read VS10xx's filter
// values through the SPI bus.
// Parameter d must be a pointer to a 128-size vector like below:
// u_int16 data[128];
#define SCI_WRAMADDR 7
#define SCI_WRAM 6

void ReadFilters(u_int16 *d) {
  int i;
  WriteSci(SCI_WRAMADDR, 0x1800);
  for (i=0; i<128; i++) {
    d[i] = ReadSci(SCI_WRAM);
  }
}
```

### 8.1.4 Function WriteSci() SPI Implementation

```
// This pseudo-code shows how to write to VS10xx's SCI
// (serial command interface) SPI bus. You need to write
// your own implementation for your own microcontroller.

void WriteSci(u_int16 addr, u_int16 data) {
  AssertSpiChipSelect();
  WriteSpiByte(2);         // 2 = write, 3 = read
  WriteSpiByte(addr);      // SCI register number
  WriteSpiByte(data>>8);   // 8 MSb's of data
  WriteSpiByte(data&0xff); // 8 LSb's of data
  DeassertSpiChipSelect();
}
```

### 8.1.5 Function ReadSci() SPI Implementation

```
// This pseudo-code shows how to read from VS10xx's SCI
// (serial command interface) SPI bus. You need to write
// your own implementation for your own microcontroller.

u_int16 ReadSci(u_int16 addr) {
  u_int16 res;
  AssertSpiChipSelect();
  WriteSpiByte(3);                  // 2 = write, 3 = read
  WriteSpiByte(addr);               // SCI register number
  res = (u_int16)ReadSpiByte()<<8;  // 8 MSb's of data
  res |= ReadSpiByte()&0xff;        // 8 LSb's of data
  DeassertSpiChipSelect();
  return res;
}
```

## 8.2 UART Functions

These are the support functions needed for UART operations. You need to replace WriteUart() and ReadUart() with your own functions.

The UART parameters are 9600 8N1 (9600 bit/s, 8 data bits, no parity, 1 stop bit).

### 8.2.1 Function Set7Chan() UART Implementation

```
// Parameter is a 7-long vector of dB values for
// 64, 160, 400, 1000, 2500, 6250, and 15600 Hz
void Set7Chan(const double *dB) {
  // Check that a previous filter update is not going on,
  // and wait if necessary.
  while (ReadUart(0x180b) & 8);

  // Set new equalizer values
  WriteUart(0x1800, (s_int16)(dB[0]*256.0));
  WriteUart(0x1801, (s_int16)(dB[1]*256.0));
  WriteUart(0x1802, (s_int16)(dB[2]*256.0));
  WriteUart(0x1803, 0);
  WriteUart(0x1804, (s_int16)(dB[3]*256.0));
  WriteUart(0x1805, (s_int16)(dB[4]*256.0));
  WriteUart(0x1806, (s_int16)(dB[5]*256.0));
  WriteUart(0x1807, 0);
  WriteUart(0x1808, (s_int16)(dB[6]*256.0));
  WriteUart(0x1809, 0);
  WriteUart(0x180a, 0);
  WriteUart(0x180b, 8);  // Activate filter designer
}
```

### 8.2.2 Function SetFilter() UART Implementation

```
// Sets one manual filter.
void SetFilter(u_int16 memoryLocation, u_int16 freqHz, double dB, double q,
               s_int16 left, s_int16 right) {
  u_int16 base = 0x1800+memoryLocation*4;
  // Check that a previous filter update is not going on,
  // and wait if necessary.
  while (ReadUart(base+3) & 1);

  // Do our filter update
  WriteUart(base+0, freqHz);
  WriteUart(base+1, (s_int16)(dB*256.0));
  WriteUart(base+2, (u_int16)( q*256.0));
  WriteUart(base+3, (left?4:0)|(right?2:0)|1); // Left, right, update
}
```

### 8.2.3 Function ReadFilters() UART Implementation

```
// This pseudo-code shows how to read VS10xx's filter
// values through the UART.
// Parameter d must be a pointer to a 128-size vector like below:
// u_int16 data[128];
void ReadFilters(u_int16 *d) {
  int i;
  for (i=0; i<128; i++) {
    d[i] = ReadUart(0x1800+i);
  }
}
```

### 8.2.4 Function WriteUart() UART Implementation

```
// This pseudo-code shows how to write to one register
// using VS1053b / VS1063a Equalizer's UART interface.
// You need to write your own implementation for your
// own microcontroller.
const char tohex[16] = "0123456789abcdef";

void WriteUart(u_int16 addr, u_int16 val) {
  SendUartChar('X');
  SendUartChar(tohex[(addr >> 12) & 0xF]);
  SendUartChar(tohex[(addr >>  8) & 0xF]);
  SendUartChar(tohex[(addr >>  4) & 0xF]);
  SendUartChar(tohex[(addr >>  0) & 0xF]);
  SendUartChar(tohex[(val >> 12) & 0xF]);
  SendUartChar(tohex[(val >>  8) & 0xF]);
  SendUartChar(tohex[(val >>  4) & 0xF]);
  SendUartChar(tohex[(val >>  0) & 0xF]);
}
```

### 8.2.5 Function ReadUart() UART Implementation

```
// This pseudo-code shows how to write to one register
// using VS1053b / VS1063a Equalizer's UART interface.
// You need to write your own implementation for your
// own microcontroller.
#include <ctype.h>
const char tohex[16] = "0123456789abcdef";
s_int16 FromHex(register char c) {
  if (isdigit(c)) return c-'0';
  if (isxdigit(c)) return(tolower(c)-('a'-10));
  return -1;
}

void ReadUart(u_int16 reg) {
  u_int16 res;
  SendUartChar('x');
  SendUartChar(tohex[(reg >> 12) & 0xF]);
  SendUartChar(tohex[(reg >>  8) & 0xF]);
  SendUartChar(tohex[(reg >>  4) & 0xF]);
  SendUartChar(tohex[(reg >>  0) & 0xF]);
  res =  (u_int16)FromHex(GetUartChar()) << 12;
  res |= (u_int16)FromHex(GetUartChar()) <<  8;
  res |= (u_int16)FromHex(GetUartChar()) <<  4;
  res |= (u_int16)FromHex(GetUartChar()) <<  0;
  return res;
}
```

# 9 Loading and Starting the Plugin

To load and start the VS1053b/VS1063a Equalizer, do the following steps:

1. Start up VS1053b/VS1063a in a normal fashion. There is no need to set the clock register. Do not set SCI_BASS (2) to anything else than the default 0.
2. Disable any potential user application by setting SCI_AIADDR (10) to 0.
3. Load the application plugin file *vs1053b_vs1063a_equ.plg* as explained in Chapter 12.
4. Activate the application by writing 0x34 to register SCI_AIADDR (10).
5. Wait until DREQ pin goes high.
6. No filters are on by default, so now you will hear unprocessed sound.
7. Start setting filters as shown in one of the ways shown in Chapters 4 and 5.

# 10 Programming and Running the Standalone Version

There is also a standalone version of the equalizer, which can be used with systems that don't have a microcontroller, or where the microcontroller has so little memory that it cannot store and load the equalizer application to VS1053/VS1063. In this case the filter is automatically loaded at boot time by VS1053/VS1063 from a dedicated SPI EEPROM / FLASH, and initialized with values in the SPI memory. While running, the filters may be adjusted through SCI or UART.

The standalone version image is *vs1053b_vs1063a_equ_sa.img*, which can be directly programmed to a VS1053/VS1063 compatible SPI EEPROM / FLASH. The size of the memory must be at least 16 KiB (128 kbits).

As a default the image contains no signal processing: input is copied directly to the output. To create a preprogrammed set of filters, the following steps need to be taken:

1. For the development phase, run the VS10xx IC with a microcontroller (either the plugin version of the application (Chapter 9), or the standalone version without any preadjustments).
2. Set the filters.
3. Read content of X addresses from 0x1800 through 0x18ff (128 16-bit values), using pseudo-code for ReadFilters() presented in Chapter 8.1.3.
4. Open *vs1053b_vs1063a_equ_sa.img* with a hex editor or a C program.
5. At byte offset 0x10, you will see a two-byte pattern 0x12 0x34 that is repeated 128 times, filling 256 bytes.
6. Copy the data you got earlier to this part of the file. 16-bit numbers are encoded in the big-endian fashion, so number 0xface should be converted to bytes 0xfa 0xce (NOT 0xce 0xfa!).
7. Now you can copy the boot image to the SPI EEPROM / FLASH. If you do this using a VS1053/VS1063 flasher program, remember to pull GPIO0 down before boot with a 10k resistor. This will prevent VS1053/VS1063 from trying to boot from SPI.
8. After programming, pull GPIO0 of VS1053/VS1063 high (e.g. 100k resistor), and reset the IC. Now it will boot from SPI and automatically run the equalizer application with the parameters you set.
9. Optionally, you may now control the Equalizer through SPI or UART.

## 10.1 Standalone Tutorial: Building the Standalone Version

In this tutorial we are going to design a filter set and edit it into the standalone image file.

### 10.1.1 Standalone Tutorial: Making Filters

First we need to make filters, either using the 7-Band Equalizer Designer as shown in Chapter 4, or building the filters manually, as shown in Chapter 5.

In this example, we are going to use the 7-Band Equalizer Designer, as follows (Note: implementation for Set7Chan() is in Chapter 8.1.1):

```
// This filter will greatly add to bass and treble
//                      64   160   400  1000  2500  6250 15600 Hz
u_int16 dBTab[7] = {12.0,  5.0,  1.0, -1.5, -3.0,  2.0,  6.0};

Set7Chan(dBTab);
```

The vector Set7Chan() writes to VS1053/VS1063 through SCI_WRAM should look very close to this:

```
0xc00, 0x500, 0x100, 0x0 0xfe80, 0xfd00, 0x200, 0x0, 0x600, 0x0, 0x0, 0x8
```

### 10.1.2 Standalone Tutorial: Reading Filter Vector

Now we need to read the resulting filter values. This can be done as follows:

```
u_int16 filterVector[128];
ReadFilters(filterVector);
```

After this, filterVector should be very close to this:

```
filterVector[0x00..0x07]: 0000 0000 0000 0000 0000 0000 0000 0000
filterVector[0x08..0x0f]: 0000 0000 0000 0000 0000 0000 0000 0000
filterVector[0x10..0x17]: 0000 0000 0000 0000 0000 0000 0000 0000
[... lots of similar lines deleted ...]
filterVector[0x40..0x47]: 0000 0000 0000 0000 0000 0000 0000 0000
filterVector[0x48..0x4f]: 0000 0000 0000 0000 0040 0b4e 0200 0006
filterVector[0x50..0x57]: 0065 0276 0300 0006 00a0 0379 0200 0006
filterVector[0x58..0x5f]: 00fd 0092 0300 0006 0190 fff7 0200 0006
filterVector[0x60..0x67]: 0278 ffb6 0300 0006 03e8 fe49 0200 0006
filterVector[0x68..0x6f]: 062d ff21 0300 0006 09c4 fc79 0200 0006
filterVector[0x70..0x77]: 0e08 ffb8 0300 0000 186a 01d3 0200 0006
filterVector[0x78..0x7f]: 2692 014b 0300 0000 3cf0 05f3 0200 0006
```

### 10.1.3 Standalone Tutorial: Updating Filters to the Image File

Before editing, the first 0x110 bytes of vs1053b_vs1063a_equ_sa.img should look like this:

```
00000000  50 26 48 01 01 08 17 fc  00 00 00 00 00 00 00 00  |P&H....ü........|
00000010  12 34 12 34 12 34 12 34  12 34 12 34 12 34 12 34  |.4.4.4.4.4.4.4.4|
00000020  12 34 12 34 12 34 12 34  12 34 12 34 12 34 12 34  |.4.4.4.4.4.4.4.4|
[... lots of similar lines deleted ...]
00000100  12 34 12 34 12 34 12 34  12 34 12 34 12 34 12 34  |.4.4.4.4.4.4.4.4|
```

You can either drop the resulting vector to the image file with a hex editor to byte offset 0x10, or you can copy the values using the following C code:

```
FILE *fp = fopen("vs1053b_vs1063a_equ_sa.img", "r+b");
fseek(fp, 0x10, SEEK_SET);
for (i=0; i<128; i++) {
  fputc((filterVector[i]>>8)&0xFF);
  fputc((filterVector[i]   )&0xFF);
}
fclose(fp);
```

After editing in the new contents of filterVector, the first 0x110 bytes of the image file vs1053b_vs1063a_equ_sa.img should look like this:

```
00000000  50 26 48 01 01 08 17 fc  00 00 00 00 00 00 00 00  |P&H....ü........|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
[... lots of similar lines deleted ...]
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 40 0b 4e 02 00 00 06  |.........@.N....|
000000b0  00 65 02 76 03 00 00 06  00 a0 03 79 02 00 00 06  |.e.v.....ă.y....|
000000c0  00 fd 00 92 03 00 00 06  01 90 ff f7 02 00 00 06  |.ý........ßœ....|
000000d0  02 78 ff b6 03 00 00 06  03 e8 fe 49 02 00 00 06  |.xßű.....èţI....|
000000e0  06 2d ff 21 03 00 00 06  09 c4 fc 79 02 00 00 06  |.-ß!.....Äüy....|
000000f0  0e 08 ff b8 03 00 00 00  18 6a 01 d3 02 00 00 06  |..ßÿ.....j.Ó....|
00000100  26 92 01 4b 03 00 00 00  3c f0 05 f3 02 00 00 06  |&..K....<ð.ó....|
```

If you want to use the mono microphone input instead of the stereo line input, set bit 0 of word 0xf of the image file, so it begins like this:

```
00000000  50 26 48 01 01 08 17 fc  00 00 00 00 00 00 00 01  |P&H....ü........|
```

### 10.1.4 Standalone Tutorial: Writing Image File to SPI Memory

An image file may be copied to the SPI EEPROM or FLASH either using a programmer, or any VS1053 / VS1063 board with a UART connection.

Open the Command Prompt. Using *cd*, go to the software folder in this package. Then, depending on whether you are using VS1053 or VS1063, run either *uniprom1053.bat*, or *uniprom1063.bat*, respectively. As a parameter, you need to give the COM port number of your RS232 port or USB UART converter (see example below).

Example:
Below is a successful run of programming the image on a VS1063, using port COM4. Note that only the first command is issued by the user; the others come from the .bat file.

```
G:\software>uniprom1063.bat 4

G:\software>copy vs1053b_vs1063a_equ_sa.img eeprom.img
        1 file(s) copied.

G:\software>vs3emu -chip vs1002b -e 0x50 -s 9600 -x 12288 -ts 38400 -l vs1063-un
iprom2.coff -c run.cmd -p 4
VSEMU 2.2 Nov 12 2010 16:48:42(c)1995-2010 VLSI Solution Oy
Using serial port 4, COM speed 9600
Waiting for a connection to the board...

Caused interrupt
Interrupted at 0x400d
Chip version "1063"
Stack pointer 0x1920, bpTable 0x4a2f
User program entry address 0x50
Speed changed to 38400
vs1063-uniprom2.coff: includes optional header, 24 sections, 930 symbols
Section 1: code        page:0 start:80 size:7 relocs:2 fixed
Section 2: puthex      page:0 start:87 size:50 relocs:2
Section 3: puthex8     page:0 start:137 size:36 relocs:2
Section 4: SpiPrivDelay  page:0 start:173 size:17 relocs:1
Section 5: SpiSendClocks  page:0 start:190 size:24 relocs:1
Section 6: SpiPrivSendReceive  page:0 start:214 size:51 relocs:4
Section 7: SpiPrivInit  page:0 start:265 size:14 relocs:0
Section 8: SpiPrivRead  page:0 start:279 size:52 relocs:11
Section 9: SpiPrivStatus  page:0 start:331 size:33 relocs:4
Section 10: SpiPrivWrite  page:0 start:364 size:127 relocs:31
Section 11: SpiStatus   page:0 start:491 size:33 relocs:4
Section 12: SpiEraseBlock  page:0 start:524 size:102 relocs:22
Section 13: SpiBlockWrite  page:0 start:626 size:70 relocs:13
Section 14: SpiVerify   page:0 start:696 size:68 relocs:15
Section 15: SpiReadBlock  page:0 start:764 size:59 relocs:13
Section 16: BlockProgram  page:0 start:823 size:62 relocs:10
Section 17: ReadSignature  page:0 start:885 size:120 relocs:38
Section 18: PrintFirst16  page:0 start:1005 size:87 relocs:20
Section 19: main        page:0 start:1092 size:287 relocs:121
Section 20: init_x      page:1 start:7168 size:21 relocs:0
```

```
Section 21: const_x     page:1 start:7189 size:792 relocs:0
Section 22: bss_x       page:1 start:7981 size:274 relocs:0
Section 23: VS_stdiolib  page:0 start:1379 size:74 relocs:18
Section 24: VS_stdiolib$0  page:0 start:1453 size:110 relocs:32
VS1053/VS1063 16/24 BIT SPI EEPROMMER / FLASHER V1.2 (UNIPROM2)
(can prom up to 64K bytes)
Detecting connected SPI Flash or EEPROM type...
Read electronic signature (RDES (0xAB) method): 29 29
Read manufacturer and product id (RDID (0x90) method): 00 00
First 16 bytes of the chip are now:
50264801010817fc0000000000000000 P&H.............
Sending a Flash Erase Block 0 (64KB) Command (0xD8)...
Erase took time, probably this is a Flash chip: Using 24-bit address.
First 16 bytes of the chip are now:
ffffffffffffffffffffffffffffffff ................
Opened file eeprom.img.
Programming blocks of data... "." means OK, "X" means error.
. 0x01 KB.. 0x02 KB.. 0x03 KB.. 0x04 KB.. 0x05 KB.. 0x06 KB.. 0x07 KB.. 0x08 KB.
. 0x09 KB.. 0x0a KB.. 0x0b KB.. 0x0c KB.. 0x0d KB... 0x0e KB..
Verifying...
.. 0x01 KB.. 0x02 KB.. 0x03 KB.. 0x04 KB.. 0x05 KB.. 0x06 KB.. 0x07 KB.. 0x08 KB
.. 0x09 KB.. 0x0a KB.. 0x0b KB.. 0x0c KB.. 0x0d KB.. 0x0e KB..
Verify OK.
Finished!
First 16 bytes of the chip are now:
50264801010817fc0000000000000000 P&H.............
Resetting chip.
  A2  : 0x10  A1  : 0x1010  A0  : 0x1010
  B2  : 0x10  B1  : 0x1010  B0  : 0x1010
  C2  : 0x10  C1  : 0x1010  C0  : 0x1010
  D2  : 0x10  D1  : 0x1010  D0  : 0x1010
  LR0 : 0x1010  LR1 : 0x1010  MR0 : 0x1010  MR1 : 0x1010
  LC  : 0x1010  LS  : 0x1010  LE  : 0x1010
  I0  : 0x1010  I1  : 0x1010  I2  : 0x1010  I3  : 0x1010
  I4  : 0x1010  I5  : 0x1010  I6  : 0x1010  I7  : 0x1010
  P   : 0x0010101010 =~ 269488144
  A   : 0x0010101010 =~ 269488144
  B   : 0x0010101010 =~ 269488144
  C   : 0x0010101010 =~ 269488144
  D   : 0x0010101010 =~ 269488144
  PC  : 0x00000000
Next Exec: 0x0000 LDC 0x0,A0
_audio_buffer + 0x000000


G:\software>
```

# 11 Frequency Responses

This Chapter shows example frequency responses for the filters designed either with the 7-Band Equalizer Designer (Chapter 11.1), or manually (Chapter 11.2).

All examples have been generated by running the equalizer software on the VSDSP processor from a digital input to a digital output.

## 11.1 Frequency Responses: 7-Band Equalizer

All of the figures in this Chapter are for the VS1063.

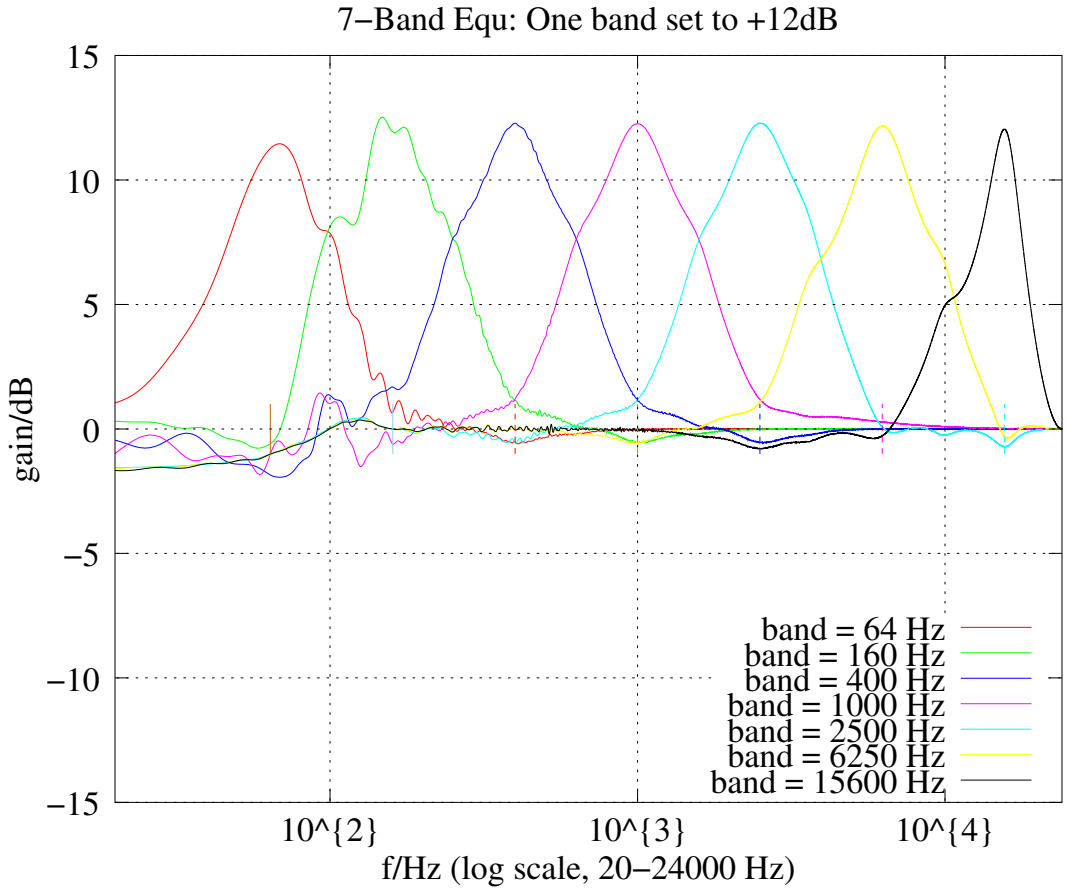


Figure 4: 7-Band Equalizer: One band set to +12 dB, others at 0 dB.

Figure 4 shows the frequency responses for the 7-band equalizer when each of the seven bands are set to +12 dB, while all others are set to 0 dB.

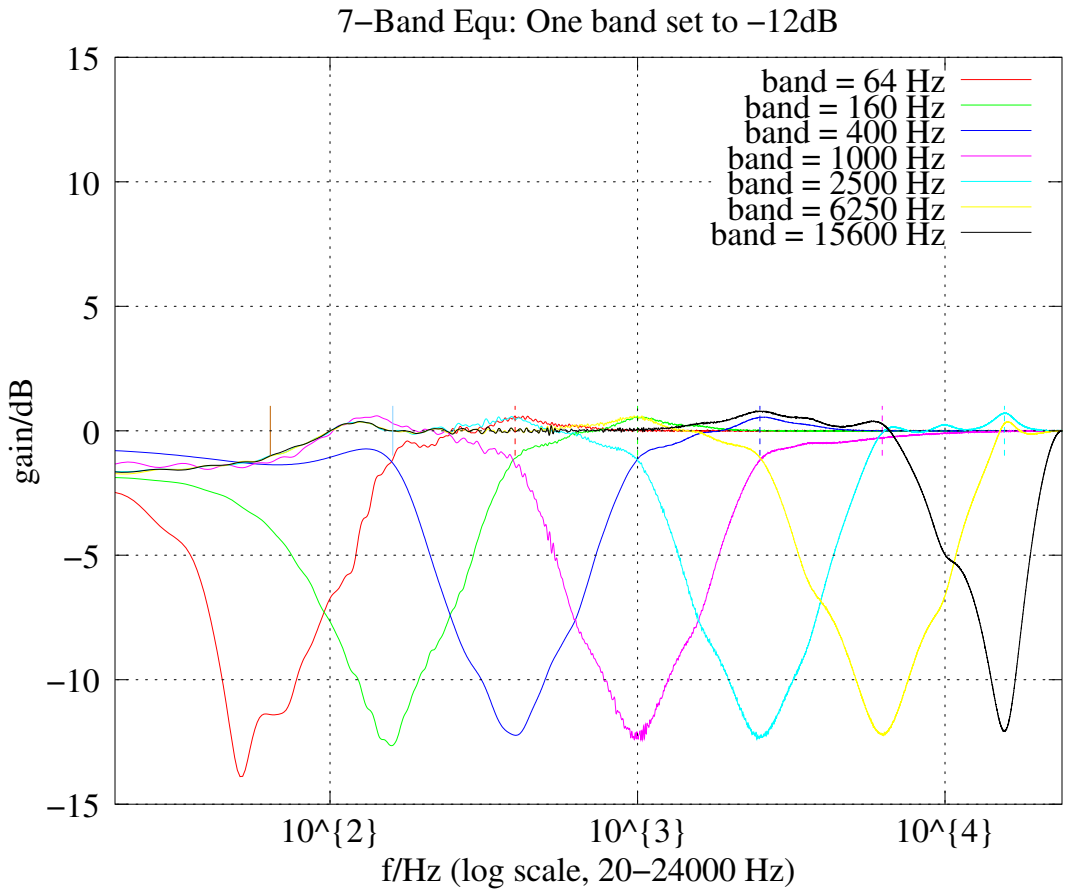


Figure 5: 7-Band Equalizer: One band set to -12 dB, others at 0 dB.

Figure 5 shows the frequency responses for the 7-band equalizer when each of the seven bands are set to -12 dB, while all others are set to 0 dB.

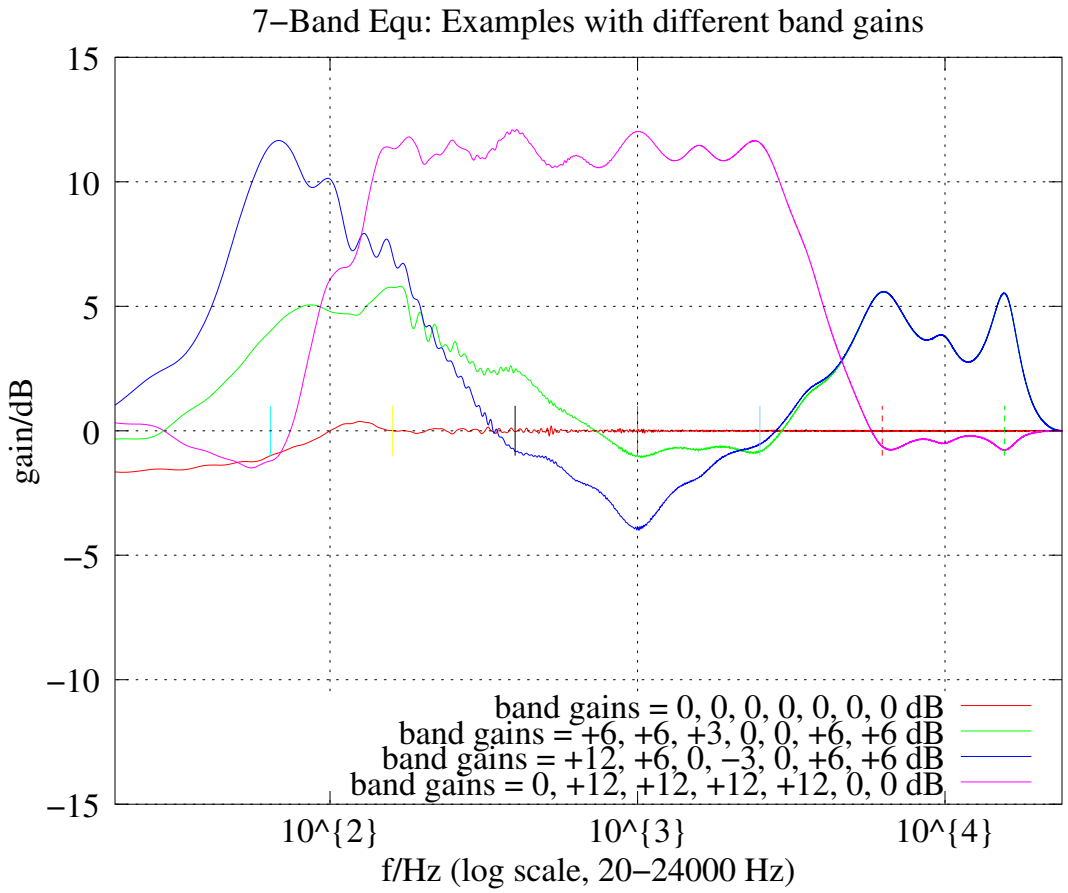


Figure 6: 7-Band Equalizer: Examples with different band gains.

Figure 6 shows the frequency responses for the 7-band equalizer with four different settings. The first setting is an “all-neutral” setting with all bands set to 0 dB.

The two next ones are different “loudness” curves.

The last example shows how the 7-band filter uses additional filters to smooth its frequency response: while four adjacent bands are set to +12 dB, you can see seven peaks in the frequency response: the four main bands, as well as an extra band between each of them.

### 11.2 Frequency Responses: Single Filters

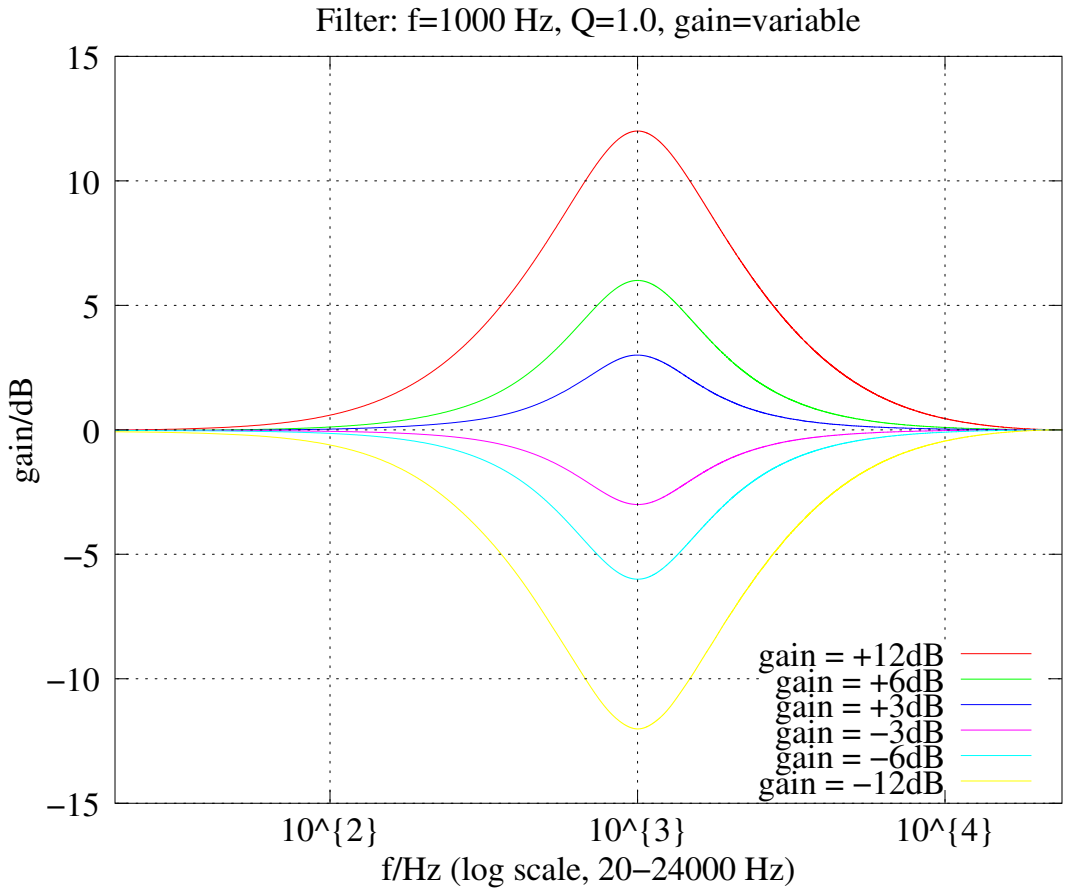


Figure 7: Single filter: Effect of gain.

Figure 7 shows how gain affects a single filter set to 1 kHz, Q=1.0.

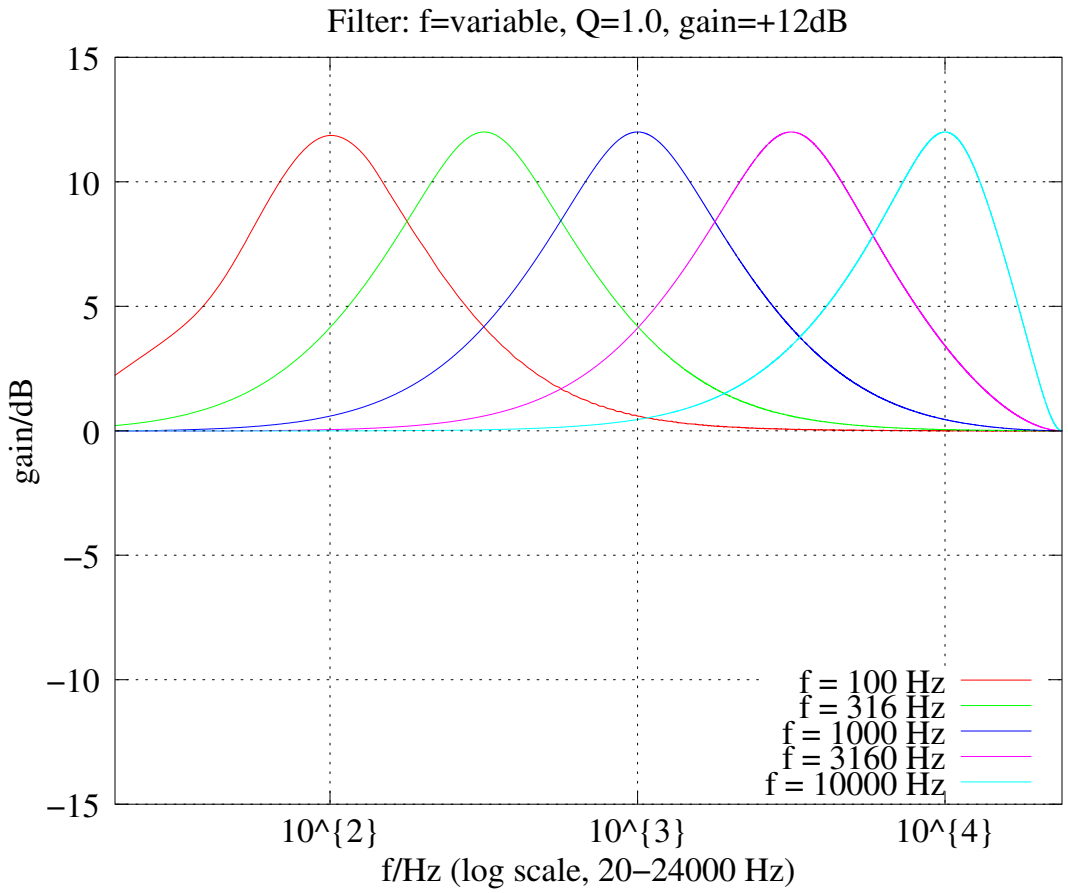


Figure 8: Single filter: Effect of center frequency.

Figure 8 shows how center frequency affects a single filter set to Q=1.0, gain=+12 dB.

Note that filter frequency responses are symmetrical on a logarithmic frequency scale. A slight exception to this are the filters that are close to zero frequency or fs/2 (24000 Hz).

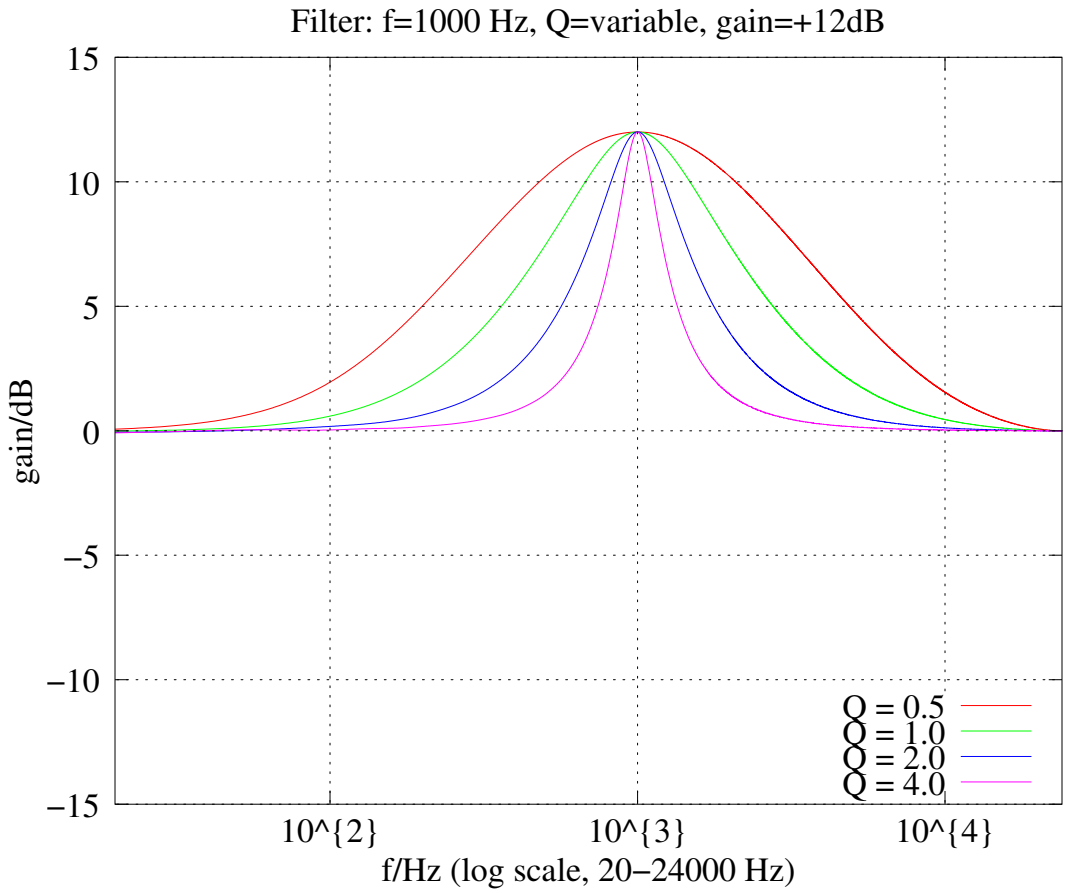


Figure 9: Single filter: Effect of Q factor.

Figure 9 shows how Q factor affects a single filter set to 1 kHz, gain=+12 dB.

# 12 How to Load a Plugin

A plugin file (.plg) contains a data file that contains one unsigned 16-bit vector called plugin. The file is in an interleaved and RLE compressed format. An example of a plugin vector is:

```
const unsigned short plugin[10] = { /* Compressed plugin */
  0x0007, 0x0001, 0x8260,
  0x0006, 0x0002, 0x1234, 0x5678,
  0x0006, 0x8004, 0xabcd,
};
```

The vector is decoded as follows:

1. Read register address number `addr` and repeat number `n`.
2. If `n` & 0x8000U, write the next word `n` times to register `addr`.
3. Else write next `n` words to register `addr`.
4. Continue until table has been exhausted.

The example vector first tells to write 0x8260 to register 7. Then write 2 words, 0x1234 and 0x5678, to register 6. Finally, write 0xabcd 4 times to register 6.

Assuming the vector is in vector `plugin[]`, a full decoder in C language is provided below:

```
void WriteSci(unsigned short addr, unsigned short value);

void LoadUserCode(void) {
  int i = 0;

  while (i<sizeof(plugin)/sizeof(plugin[0])) {
    unsigned short addr, n, val;
    addr = plugin[i++];
    n = plugin[i++];
    if (n & 0x8000U) { /* RLE run, replicate n samples */
      n &= 0x7FFF;
      val = plugin[i++];
      while (n--) {
        WriteSci(addr, val);
      }
    } else {           /* Copy run, copy n samples */
      while (n--) {
        val = plugin[i++];
        WriteSci(addr, val);
      }
    }
  }
}
```

# 13 Contact Information

VLSI Solution Oy
Entrance G, 2nd floor
Hermiankatu 8
FI-33720 Tampere
FINLAND

URL: http://www.vlsi.fi/
Phone: +358-50-462-3200
Commercial e-mail: sales@vlsi.fi

For technical support or suggestions regarding this document, please participate at
http://www.vsdsp-forum.com/
For confidential technical discussions, contact
support@vlsi.fi