VS1005 VSOS PROGRAMMER'S GUIDE

VS1005g

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

Revision History			
Rev.	**Date**	**Author**	**Description**
0.24.0	2013-07-22	HH	Chapter 10, *Audio*, updated for VSOS v.024.
0.22.0	2012-11-15	HH&PKP	Initial release for VSOS v0.22.

Contents

List of Figures

1 Introduction

VLSI Solution's flagship audio codec integrated circuit VS1005g has myriads of hardware features. To make them more programmable, VLSI Solution has created a proprietary operating system, called VSOS (VLSI Solution Operating System), to make it as easy as possible for the programmers to access these numerous features.

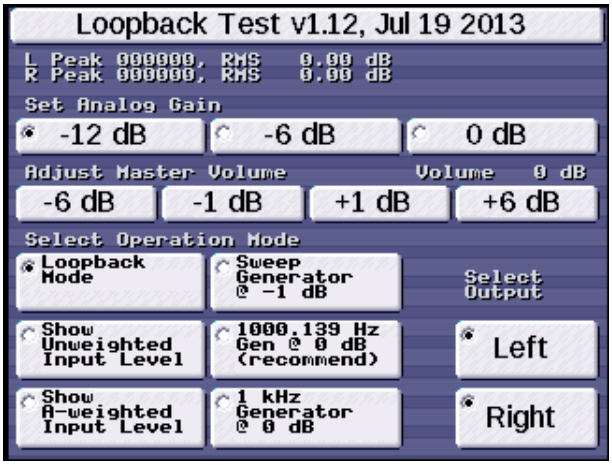


Figure 1: Touch-based LCD user interface example for a VSOS application.

After the disclaimer and definitions in Chapters 2 and 3, an overview of VSOS is given in Chapter 4, *VSOS Overview*. It is followed by a presentation of VSOS basic build blocks in Chapter 5, *VSOS Fundamentals*.

Model-View-Controller-style messaging is explained in Chapter 6, *User Interface Messages*.

VSIDE templates are listed and explained in Chapter 7, *VSOS Templates in VSIDE*.

The VSOS API interfaces are listed in Chapter 8, *THe VSOS API Reference*.

Chapter 9, *VS_DSP4 Architecture*, explains some aspects of the VS_DSP4 processor architecture that are useful for programmers to know.

Chapter 10, *Audio*, tells how to handle audio under VSOS.

Finally, Chapter 11, *How to Write Efficient VS_DSP4 Code*, instructs on how to get your code run faster and smaller.

2 Disclaimer

VLSI Solution makes everything it can to make this documentation as accurate as possible. However, no warranties or guarantees are given for the correctness of this documentation.

At this stage, this document is still very much work-in-progress. It will, however, be updated along with VSOS.

3 Definitions

DSP Digital Signal Processor.

I-mem Instruction Memory (Chapter 9.1).

LSW Least Significant (16-bit) Word.

MSW Most Significant (16-bit) Word.

RISC Reduced Instruction Set Computer.

VS_DSP[4] VLSI Solution's DSP core (Chapter 9).

VSIDE VLSI Solution's Integrated Development Environment.

VSOS VLSI Solution's Operating System.

X-mem X Data Memory (Chapter 9.1).

Y-mem Y Data Memory (Chapter 9.1).

4 VSOS Overview

Traditionally signal processing applications have been monolithic in nature. Each application would take complete control of the hardware and provide all the necessary support functions that interact with the hardware for the application. In essence, it used to be necessary that the application provides its own operating system. This has been very challenging for application programmers, because to make even a simple application for a DSP, the programmer was required to have a lot of special knowledge of the hardware.

VSOS, the VLSI Solution Operating System, aims to provide an application environment, which is familiar to C programmers everywhere. Instead of having to learn platform specific functions for writing to the screen, reading from storage devices, or outputting audio, the VSOS allows the programmer to use familiar functions such as printf(), fopen() or fwrite() to access the hardware or read and write files. If you have ever written a "C" language program that writes to the screen or to a file, you will find VSOS similar.

Like other modern operating systems, VSOS follows the Files, Devices and Device Drivers approach to handling data in the application. Most hardware targets and data streams can be handled as files in VSOS. When you use printf(), the system writes to `stdout` just like in any other OS. `Stdout` is a file, which is by default handled by the `console` device. The `console` device keeps track of where text output from an application should go to and how it should be displayed. Finally it calls a BIOS service `LcdFilledRectangle` to display the text on an LCD screen, on a video monitor output, on a serial terminal, or on whatever display hardware the console happens to be using in your VS1005 based device.

Audio streams are also treated as files in VSOS. An application can play audio by writing to the `stdaudioout` file and input audio by reading from the `stdaudioin` file. VSOS will handle the `stdaudioout` output from your application, directing it to the DAC or digital outputs, if such are present, or to redirect it to some other file or device if the user has so chosen. Or you can force the audio output to a specific device if your application requires it.

Inputting and outputting audio is remarkably easy under VSOS. With just a couple of lines of code, you can read from audio input and write to audio output. In between, you can process the audio to make DSP effects such as filters.

VSOS is also multithreaded. In today's sense it's not multitasking, since only one user application can currently be loaded at one time, but the application can have several threads, usually one for audio input and output and another for the user interface. At the same time, the operating system kernel can take care of I/O and networking tasks using its own threads. VSOS task switching is pre-emptive.

The VSOS kernel provides three interfaces for writing user interfaces. Firstly there is the Standard Output, allowing functions such as printf() to write text output to the screen. Secondly, the display BIOS provides an LcdFilledRectangle() function for displaying bitmaps on the screen and a GetTouchLocation() function to get coordinates from a pointing device such as a touch screen. Thirdly, the kernel implements an inter-

face called StdButtons, which creates and renders controls such as buttons, text areas and custom screen objects on a display device. Depending on the capabilities of the display device, they can be displayed as touch screen bitmaps, text or even hardware I/O tokens such as low level messages on a serial port, making it possible to use a remote link or an external MCU to control applications running under VSOS in VS1005.

Origins of VSOS

VSOS is inspired by a traditional generic model of a POSIX-like operating system, which has files, file descriptors, processes, process states and uses an object oriented approach for describing files, file operations and device drivers.

The VSOS Operating System is original work of VLSI Solution Oy. Panu-Kristian Poiksalo is the main author of the kernel, device drivers, kernel data structures, calling conventions, model-view-controller interface and application loading. Henrik Herranen has created the Standard Audio Interface, Audio Driver and the MP3 decoder model for VSOS. Pasi Ojala has provided the RTOS threads and task switching. Lasse Makkonen has written the lcc C compiler and demo code and built the distribution packages.

5 VSOS Fundamentals

5.1 Files

Files are abstractions of data streams. They can be read and/or written as a series of bytes. There are different kinds of files, but they all have similar operations. Disk files can be stored in a disk device such as SD card or nand flash. Console files can be written to a screen or read from a keyboard. Audio files are written into audio devices such as the DAC for listening on a pair of headphones or line out, or read from an audio device such as the line in ADC.

Basically any kind of data stream which can be read or written can be described as a file. It's often useful to do that to be able to use VSOS's functions that manipulate and work on files.

Some files can be redirected. `Stdout`, for example, is initialized by the kernel to point to a system file called `consoleFile` allowing the console device to take care of it. But let's say that you open a disk file and make `FILE *stdout` point to it, and then load and run an application. The text output from that application would then be written to the disk file.

5.2 File I/O

Files in VSOS are used as they are in standard C. `fopen()` and `fclose()` can be used to open and create files. `fread()`, `fwrite()` and functions such as `printf()`, `fprintf()`, `fgets()` or `fputs()` can be used to read and write data to and from files.

The following program creates a file and writes a line of text to it:

```
#include <vo_stdio.h>

int main(void) {
  FILE *fp = fopen("S:OUT.TXT", "w");
  if (fp) {
    fprintf(fp, "Hello, World!\n");
    fclose(fp);
  }
  return S_OK;
}
```

Notice that the file name starts with "S:" which denotes the "System Disk", or the block storage device from which the system files are loaded. On an unmodified VSOS 0.22 kernel this would mean the SD card.

Under VSOS 0.22, you can only create files with "8.3" style file names (up to 8 characters file name and up to 3 characters extension). Similarly, you must know the "8.3" style

short file name to pass it as a parameter to fopen() to open existing files. Luckily the file searching routines in vo_fatdirops will give you the short file name if you search the directory based on the long file name. Please see the "Main Menu Template" information in Section 7.4 for more info. You can use and access subdirectories that exist on the disk, but not create them under VSOS 0.22.

For more information on File I/O, please refer to a source such as "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie.

5.3 VSDSP Specific File I/O Behaviour

VSOS does its best to provide an interface which is useful and as close to standard C as is practical when working with files. Some idiosyncrasies follow from the fact that the VS_DSP[4] architecture doesn't have 8-bit data types.

Like in some other DSPs and specialized processors, the size of a char is not 8 bits, but 16 bits in VS_DSP. Since, by definition, a char is the smallest addressable unit in a system, it means that if you `fread()` a char from a file, you will get 16 bits, or two eight-bit bytes, instead of one eight-bit byte.

Following the "C" standard to the letter, it would be required to also return 16 bits on a call to `fgetc()` to read a character. But to make it convenient to process standard 8-bit text files in VSOS, `fgetc()`, `fputc()` and the functions `fgets()` and `fputs()` which depend on them, return and use 8-bit characters as you would probably expect.

This means, that although VS_DSP is a 16-bit processor, if you read and write characters and lines of text in files using `fgets()`, `fputs()`, `fgetc()` and `fputc()`, you will be able to read and write 8-bit ("ASCII") text files normally. Internally the strings are stored as 16-bit char arrays with the high 8 bits cleared at file reads and ignored at file writes.

See Chapter 9.3 for details on idiosyncrasies in the VS_DSP architecture.

5.4 Standard Input and Output

As a default, five file handles are open. Three of them are the ones usually expected from a C system, namely *stdin*, *stdout*, and *stderr*. If you have the following line in your code, stdin and stdout will act through the VS3EMU emulator:

```
#include <stdio.h>       /* Not recommended for VS1005 projects */
```

However, this is only practical if you are debugging a kernel and thus have a VS3EMU connection. If you boot the board from a flash, there is no VS3EMU connection and the program will stop, waiting indefinitely for VS3EMU to respond.

VSOS applications should instead use, as their first #include line:

```
#include <vo_stdio.h>    /* Make this your first program line. */
```

With this setup, *stdin* is by default not functional, and *stdout* will output to a display device such as LCD that has been connected to the VS1005g Development Board.

In addition to the standard C input and output file handles, there are two special file handles that are automatically opened for the user, called *stdaudioin* and *stdaudioout*. For more information of these file handles, intended for standard audio input and output, see Chapter 10, *Audio*.

5.5 Devices

Devices are objects that can be targets for *file operations* such as read or write, performed by *filesystems* on files. Block devices provide storage for files. Character devices perform hardware operations on streams of data. An example of a block device is an SD card and and example of a character device is the Audio device.

VSOS has an array of 26 device pointers, which are called "System Devices". They are referred to by letters from "A" to "Z". The system device pointers are stored in array `DEVICE* vo_pdevices[26]`.

The special property of system devices is that they can be accessed using fopen(). The first character of a file name passed to fopen() must be a system device letter and the second character must be a colon (":"). fopen() will find the device descriptor which corresponds with the device by indexing the vo_pdevices array based on the first character of the file name parameter.

VSOS 0.22 enumerates three system devices by default. They are "A" for Audio device, "S" for the System Disk device and "T" for (Text) Console.

To add access to a device not installed by the kernel, instantiate a device and store a pointer to it into the `vo_pdevices[]` array. For an example, you can refer to the UART device driver example.

5.6 Filesystems

Filesystems are collections of *file operation* methods, which are called by the operating system to open, close, read and write files. Each device has one filesystem, which provides the file operations for files opened by fopen() in that device.

For example, when an SD card device is created, VSOS scans an array of installed filesystem drivers in sequence, asking each one to check if it can handle the device. One of the preinstalled filesystem drivers is the FAT filesystem. The FAT filesystem attempts to make sense of the SD card contents. If the FAT filesystem reports that it can understand the file contents, VSOS associates the FAT filesystem with the SD card device.

When a file is opened by fopen(), the system device in which the file resides is determined by the system device letter of the file name. The device descriptor is read and

the file inherits its methods from the filesystem that handles the device. In the case of the SD card with a FAT filesystem, the file inherits its methods from the FAT filesystem driver. Fopen() then calls the file's newly inherited Open() method as f->op->Open(f,...) (which now points to FatFileOpen()) which reads the SD card and tries to find the file. If the method call fails, the file descriptor is zeroed and NULL returned to the caller. Otherwise, proper flags are set based on the capabilities of the device, file and filesystem and a file descriptor pointer is returned to the caller.

Three filesystem drivers are included in VSOS 0.22 kernel:

FAT filesystem provides FAT16 and FAT32 and limited FAT12 access on block devices.

DeviceFS filesystem proviedes raw stream operations on character devices without dedicated filesystems.

AudioFS filesystem provides operations on audio streams. It is implicitly associated with the Audio device using the Audio device driver.

5.7 Displays and Input Devices

Due to their special nature, displays are not considered VSOS standard devices such as those that are defined by a device descriptor. Instead, the display functionality is encapsulated into three functions: InitDisplay(), LcdFilledRectangle() and LcdTextOutXY(). Of these, only the first two need to be rewritten to target a custom display, since the default implementation of LcdTextOutXY() uses LcdFilledRectangle() to render the text output to the screen. The LcdTextOutXY() function is considered a part of the display device driver mainly because on some displays, especially those that have internal font memories or those that are text only and thus cannot support bitmap graphics, it may be impractical or impossible to use the LcdFilledRectangle() to render characters on the screen.

Bitmaps and textures in VSOS are always stored in a fixed RGB565 format, which has 16 bits per one pixel: 5 bits for red, 6 bits for green, 5 bits for blue. Having a standard definition for the texture format allows different device drivers to render same bitmaps in different devices in a compatible way.

Note that if you use bitmaps other than RGB565 in your solution, they are not compatible with display device drivers other than the ones you provide yourself.

System colors are defined in the file <rgb565.h>. If you have an application which uses transparency in bitmaps, you are encouraged to use __RGB565RGB(0,4,0) (near black) as the transparent color.

6 User Interface Messages

VSOS encourages modular programming by defining a Model-View-Controller model for passing messages between the user interface (the "View-Controller") and the application's internal operation (the "Model").

For example, VLSI Solution provides a functionality of MP3 decoding in an MP3 decoder *model*. The model is a software module which runs on a separate thread, often using a main function called ModelTask(). Once the model is running, it listens to User Interface Messages ("UiMessages") that instruct it what to do, such as start playing an MP3 or go to Pause mode.

When the model is playing an MP3 file, it also *sends* UiMessages, signaling the user interface for example once per second how many seconds it has been playing or how many percents of a file it has decoded, if such a number can be determined.

At the beginning of the song, the model sends other messages, such as the file name of the song it's currently playing and any ID3 information it can find, such as the Artist, Song title or Album.

Each UiMessage has a message number and a value, which can be a 16- or 32-bit integer, a pointer to a text string or a pointer to binary data.

The user interface message numbers are standardized in the file <uimessages.h>. Below is a list of currently defined message numbers. Many of these message numbers are unimplemented in most models. The idea is that if a message is implemented, the message number in the list should be used, if one exists. If it doesn't exist, email VLSI Solution and we will add it to the list.

For an example of how to use the UiMessages, please see the MP3 decoder templates.

List of currently defined UI messages in VSOS 0.22:

```
//CATEGORY:VSOS RESERVED 0x0000-0x00FF
#define UIMSG_VSOS_END_OF_LIST 0x0000 //No action / End of list
#define UIMSG_VSOS_GET_MSG_LIST 0x0001 //Ask for a list of uiMessageListItems
#define UIMSG_VSOS_SET_CALLBACK_FUNCTION 0x0002 //Pointer to a function which
 //receives ui messages
#define UIMSG_VSOS_MSG_LIST 0x0003 //Pointer to a list of uiMessageListItems
//SUBCATEGORY: HINTS
//These provide extra information which is useful for rendering the UI.
#define UIMSG_HINT 0x0004 //Auxiliry information for user interface generators
#define UIMSG_INPUTS 0x0005 //Start of a category of input (control) messages
#define UIMSG_OUTPUTS 0x0006 //Start of a category of output (status) messages
#define UIMSG_USE_SAME_POSITION 0x0007 //Use the same position on screen
#define UIMSG_USE_SAME_LINE 0x0008 //Use the same line on screen

//CATEGORY:BOOLEAN 0x0100-0x01FF
#define UIMSG_BOOLEAN 0x0100 //Uncategorized boolean value
```

```
//SUBCATEGORY BOOLEAN:BUTTON 0x0101-0x017F
//Buttons are booleans which are resting at value 0 and the transition
//from 0 to 1 triggers a functionality
#define UIMSG_IS_BUTTON(x) (((x)&0xFF80U) == 0x0100)
#define UIMSG_BUTTON 0x0100 //Any kind of button
#define UIMSG_BUT 0x0101 //Unspecified Button
#define UIMSG_BUT_PLAY 0x0102 //Play Button, 1=Start Playing
#define UIMSG_BUT_STOP 0x0103 //Stop Button, 1=Stop Playing
#define UIMSG_BUT_PAUSE 0x0104 //Pause Button, 1=Go to Pause state
#define UIMSG_BUT_PAUSE_TOGGLE 0x0105 //Pause Button, 1=Toggle pause state
#define UIMSG_BUT_RESTART 0x0106 //Restart the currently playing file
#define UIMSG_BUT_PREVIOUS 0x0107 //Switch to the previous track
#define UIMSG_BUT_NEXT 0x0108 //Switch to the next track
#define UIMSG_BUT_GENERIC_TOUCH 0x109 //Touchpad is touched
#define UIMSG_BUT_CLOSE_DECODER 0x10a // Close the decoder application
#define UIMSG_BUT_DECODER_CLOSED 0x10b // Decoder application is closing
#define UIMSG_BUT_FIRST 0x010c //Switch to the first track
#define UIMSG_BUT_LAST 0x010d //Switch to the last track
#define UIMSG_BUT_END_OF_SONG 0x010e //Report end of song has been reached
#define UIMSG_BUT_SAVE_POS 0x010f //Save current audio file play position
#define UIMSG_BUT_RESTORE_POS 0x0110 //Restore previously saved position

//Subcategory BOOLEAN:BOOL 0x0180-01FF
//Switches are booleans which can rest at state 0 or 1
//and the state, 0 or 1 carries a meaning.
#define UIMSG_BOOL 0x0180 //Unspecified Switch
#define UIMSG_BOOL_FAST_FORWARD 0x0181 //Cue forwards
#define UIMSG_BOOL_FAST_BACKWARD 0x0182 //Cue backwards
#define UIMSG_BOOL_SHUFFLE 0x0183 //Shuffle on(1)/off(0)
#define UIMSG_BOOL_FORCE_MONO 0x0184 //Mix audio to mono(1) or don't mix(0)
#define UIMSG_BOOL_DIFFERENTIAL_OUTPUT 0x0185 //Set differential(1)
//or normal(0) output
a#define UIMSG_DISPLAY_ON 0x0186 //Display or user interface on(1) or off(0)

//CATEGORY:NUMERIC 0x0200-0x02FF

//Subcategory NUMERIC:S16/U16 16-bit values 0x0200-0x02FF
#define UIMSG_S16 0x0200 //Unspecified signed 16-bit integer value
#define UIMSG_S16_PERCENTAGE 0x0202 //Unspecified value from 0 to 100
#define UIMSG_S16_LOOP_STATE 0x0203 // See below for values
#define UIMSG_S16_PLAYING 0x0204 // See below for values
#define UIMSG_S16_SONG_NUMBER 0x0205 //Base-1 song # in folder, <0 = fail
#define UIMSG_U16 0x0280 //Unspecified unsigned 16-bit integer value

#define UIMSG_IS_INT(x) (((x)> 0x01ff) && ((x)< 0x0400))

#define UIM_LOOP_NONE 0
```

```
#define UIM_LOOP_FILE 1
#define UIM_LOOP_FOLDER 2

#define UIM_PLAYING_STOPPED 0
#define UIM_PLAYING_PAUSED 1
#define UIM_PLAYING_PLAYING 2

//Subcategory NUMERIC:LONG 32-bit values 0x0300-0x03FF
#define UIMSG_LONG 0x0300 //Unspecified signed 32-bit integer value
#define UIMSG_UNSIGNED_LONG 0x0301 //Unspecified unsigned 32-bit integer
#define UIMSG_SAMPLE_RATE 0x0302 //Sample Rate for unspecified purpose,
//unit: millihertz.
#define UIMSG_DAC_SAMPLE_RATE 0x0303 //Sample Rate for DAC
#define UIMSG_ADC_SAMPLE_RATE 0x0304 //Sample Rate for ADC
#define UIMSG_COORDINATE 0x0305 //Generic coordinate,
//high word is Y, low word is X, left-top is 0,0.
#define UIMSG_POINTER_LOCATION 0x0306 //Location of a mouse pointer
#define UIMSG_CLICK_LOCATION 0x0307 //Location of a mouse click or
//touchscreen touch
#define UIMSG_SELECT_TRACK 0x0308
#define UIMSG_LENGTH_OF_BINARY_DATA 0x0309 //Total length in bytes of a
//(multipart) binary data block which follows
#define UIMSG_U32_PLAY_TIME_SECONDS 0x030a //Play time in seconds
#define UIMSG_U32_PLAY_FILE_PERCENT 0x030b //Percentage of file player


//CATEGORY:TIME 0x0400-0x04FF
//Time is 32-bit signed value, which signifies milliseconds.
#define UIMSG_TIME 0x0400 //Unspecified time, signed 32-bit integer in
//milliseconds with positive values to the future.
#define UIMSG_TIME_SEEK 0x0401 //Set playing position of song in milliseconds

#define UIMSG_IS_TEXT(x) (((x)&0xff00) == UIMSG_TEXT)

//CATEGORY:TEXTUAL 0x0500-0x05FF
//Textual messages value is a __x char* to a string.
//The string can be thrown away when uiCallbackFunction call returns.
#define UIMSG_TEXT 0x0500 //Unspecified text, pointer to a 16-bit string
#define UIMSG_TEXT_OPEN_FILE 0x0501 //Open a file based on its name
#define UIMSG_TEXT_CLOSE_FILE 0x0502 //Close the current file.
//A closed file cannot be reopened without a new "open file" type operation.
#define UIMSG_TEXT_SONG_NAME 0x0503 //Name of song, null-terminated
#define UIMSG_TEXT_ALBUM_NAME 0x0504 //Name of album, null-terminated
#define UIMSG_TEXT_ARTIST_NAME 0x0505 //Name of artist, null-terminated
#define UIMSG_TEXT_YEAR 0x0506 // Year of performance, null-terminated
#define UIMSG_TEXT_LYRICS 0x0507 //Lyrics update
#define UIMSG_TEXT_NAME_OF_BINARY_DATA 0x0508 //Name of binary which follows
#define UIMSG_TEXT_SHORT_FILE_NAME 0x050a // Short file name, if available
```

```
#define UIMSG_TEXT_LONG_FILE_NAME 0x050b // Long file name, if available.
/* Note: If both short and long file names are available, the short file
   name always comes first. If any other textual song metadata is available,
   like artist and song name, it always comes after file name information. */
#define UIMSG_TEXT_ERROR_MSG 0x050c // Error message, NULL = ok

//CATEGORY:BINARY 0x0600-0x06FF
//Value's high word is __x pointer to a binary data block
//Value's low word is length of binary data block in bytes
#define UIMSG_BINARY_DATA 0x0600 //Piece of generic binary data
```

7 VSOS Templates in VSIDE

This chapter gives a brief description of the VSOS 0.22 application templates in VSIDE.

7.1 Hello, World

```
#include <vo_stdio.h>

int main(void) {
  printf("Hello, World!\n");

  return S_OK;
}
```

This is the simplest application template for VSOS, and as such the one where every line is very important.

The first line includes the file <vo_stdio.h>. Each VSOS application should have this include at the top. The <vo_stdio.h> file defines the standard input and output of the application to work with the VSOS operating system, the device's screen and console. To be able to allow the use of those, it also automatically includes the file <vsos.h>, which provides declarations that allow the access to VSOS functions.

The entry point of the application is the main() function. Its return value is int (a signed 16-bit integer). For a VSOS application, it's ok to return a value of EXIT_SUCCESS or S_OK (0) for an OK exit or a negative value such as S_ERROR (-1) for reporting an error to the caller.

When loading the program, the kernel clears the screen and draws a new console frame with the application's title (from the project name) and the application's icon.

printf("Hello, World") writes the words "Hello, World!" and a newline to the console. The call is converted to a call to vo_printf() and works similarly as vo_fprintf(vo_stdout, "Hello, World"). It writes to the VSOS standard output, vo_stdout.

By default, vo_stdout is a pointer to `FILE consoleFile`, which is a directly declared file inside the kernel, handled by the Console device ("devConsole"). If you have not modified the kernel, the Console device uses a bitmap from the VS1005 ROM to render the characters on the LCD screen.

The colors which are used for printing the text are defined by the display device driver. The display device driver holds an important data structure `lcdInfo lcd0`. It is defined in <lcd.h> as

```
/// Info structure of LCD display state
typedef struct lcdInfoStruct {
```

```
  u_int16 width;
  u_int16 height;
  u_int16 x;
  u_int16 y;
  u_int16 textColor;
  u_int16 backgroundColor;
  u_int16 defaultTextColor;
  u_int16 defaultBackgroundColor;
  u_int16 clipx1,clipy1,clipx2,clipy2;
  u_int16 shadowColor, highlightColor, buttonFaceColor,buttonTextColor;
} lcdInfo;
```

The display driver chooses the initial colorset for the display. It holds values for colors which are cleanly renderable in the display.

For example a driver for a black-and-white only display might choose to render colors other than black and white by dithering, which might produce unreadable text in some colorset. That's why the default colors declared by the display device driver should be used for printf in a simple program. Later on, the colorscheme of the product may be changed.

The lcdInfo structure defines some properties of the display.

- `width`: Width of visible area in pixels.
- `height`: Height of visible area in pixels.
- `x`: X coordinate of logical cursor in pixels. Origin is left top corner.
- `y`: Y coordinate of logical cursor in pixels. Origin is left top corner.
- `textColor`: Current text color (RGB565). Used by the console, including printf.
- `backgroundColor`: Current background color. Used as the background color for text output.
- `defaultTextColor`: Defaut text color for this screen. Read by the console to set default console colors.
- `defaultBackgroundColor`: Default text background color for this screen. Read by the console.
- `clipx1, clipx2, clipy1, clipy2`: Current clipping rectangle (used for setting console text area). Also used by CreateStdButton() and SetVirtualResolution() to select an area of the screen for creating new Standard Buttons.
- `shadowColor, highlightColor`: A set of safe colors for highlighting text output.
- `buttonFaceColor, buttonTextColor`: A set of safe colors for rendering a button bitmap on the device.

You can manipulate these values before a call to printf() to control how the text is drawn on screen. If you do change them, it's a good idea to restore the original values afterwards, as in this example, which uses printf() to do a formatted write to the top-left corner of the screen using the shadow color:

```
#include <vo_stdio.h>
#include <string.h>
#include <lcd.h>

int main(void) {
  u_int16 i;
  __y lcdInfo lcdSave;
  memcpyXY(&lcdSave, &lcd0, sizeof(lcd0)); //Save the LCD state
  lcd0.x = 0;
  lcd0.y = 0;
  lcd0.textColor = lcd0.shadowColor;
  i = 1234;
  printf("The value of i is %d. ",i);
  memcpyYX(&lcd0, &lcdSave, sizeof(lcd0)); //Restore the LCD state
}
```

Later on, in the Standard Buttons template (Chapter 7.3), you will learn how to use SetVirtualResolution(), CreateStandardButton() and SetClippingRectangleToButton() to control where and how text and graphics is written on the screen in a screen resolution independent manner, respecting any color schemes the user may have chosen.

7.2 Audio In, Audio Out

```
#include <vo_stdio.h>
#include <stdlib.h>

#define BUFSIZE 128

int main(void) {
  // Remember to never allocate buffers from stack space. So, if you
  // allocate the space inside your function, never forget "static"!
  static s_int16 myBuf[BUFSIZE];

  while (1) {
    // Read stereo samples from stdaudioin into myBuf.
    // By default, stdaudioin comes from line in.
    // By default both input and output are 16-bit stereo at 48 kHz.
    fread(myBuf, sizeof(s_int16), BUFSIZE, stdaudioin);

    // Here you may process audio which is in 16-bit L/R stereo format.

    // Write stereo samples from myBuf into stdaudioout.
    // By default, stdaudioout goes to line out.
    fwrite(myBuf, sizeof(s_int16), BUFSIZE, stdaudioout);
  }

  // Not really needed because there was a while(1) before
  return EXIT_SUCCESS;
}
```

The Audio In + Audio Out template is a straightforward example on how to do an application which has audio input and output. The application works by having an internal buffer of 128 16-bit words (64 stereo samples), which it reads from the audio input and writes to the audio output.

Even before the application is loaded, VSOS has created a Standard Audio interface, which is used by accessing the files `stdaudioin` and `stdaudioout`. They use the Line In (ADC) and Line Out (DAC) by default, with a 48 kHz sample rate and 16 bits resolution. These can be changed by `ioctl()` calls to stdaudioin and/or stdaudioout.

The standard audio interface is not discussed here in more detail since it's explained thoroughly in Chapter 10.

One suggestion is to do as few as possible ioctl()'s to stdaudioin and stdaudioout. If possible, don't modify where the input comes from or where the output goes to. That way you allow the OS to decide where to output the audio from your application. If you want to configure the input or output, consider to make another application which configures stdaudioin and/or stdaudioout and then runs your audio application. That way you don't need to recompile the application binary just for changing the audio input

or output port. For example, by not touching stdaudioin or stdaudioout, you could make an audio filter application or an audio effect application, which would be able to work with any audio stream.

The main shortcoming of the template is that there is no user interface and just an endless "while(1)" loop. When this program is run, there is no way for it to exit. You will need to reset the board to kill this application.

Volume Control

The application should not set the system volume. Imagine what would happen if applications would change the system volume. The user might run one application and set a very soft volume. Then the user might change to another application, which might set a very loud volume. The user would unexpectedly hear a very loud sound, which might even damage the user's hearing or equipment.

Please allow the OS to set the system volume. Until we have a better interface for setting application-specific sound volume, you can do effects such as fade in or fade out by scaling the samples.

If you have the choice to do so, or if it's of no matter to you, you could make your system work with audio waves, which use about 50 percent of signed 16-bit sample's amplitude for nominal sound levels, e.g. a 16-bit wave should usually reach values from -16000 to +16000 in normal conditions and only rarely more than that.

7.3 Standard Buttons

```
#include <vo_stdio.h>
#include <stdbuttons.h>

StdButton buttons[10+1]={0}; //Zero-ending list of buttons: the last button's
// result must be zero and it will not be drawn.

int main(void) {
  StdButton *textArea;
  StdButton *currentButton = buttons;
  console.Ioctl(&console, IOCTL_START_FRAME, "Stdbuttons Demo");

  //Divide the lcd's current active area (clip rectangle) into 3x4 units.
  SetVirtualResolution(3,4);

  //Create a set of buttons
  CreateStdButton(currentButton++, 1, BTN_NORMAL, 2,0, 1,1, "Hello");
  CreateStdButton(currentButton++, 2, BTN_NORMAL, 2,1, 1,1, "Jello");
  CreateStdButton(currentButton++, 3, BTN_HIGHLIGHTED, 2,3, 1,1, "Exit");
  CreateStdButton(textArea=currentButton++, BTN_END, BTN_INVISIBLE | BTN_TEXT,
                  0,0, 2,4, "");

  //set the current LCD area to textArea.
  SetClippingRectangleToButton(textArea);

  RenderStdButtons(buttons); //Render the buttons;

  while (1) {
    u_int16 buttonpress = GetStdButtonPress(buttons);

    switch(buttonpress) {
    case 1:
     printf("Hello!\n");
     break;
    case 2:
     printf("Jello! ");
     break;
    case 3:
     return S_OK;
     break;
    }
  }
  return S_OK;
}
```

The StdButtons template makes a simple program that uses Standard Buttons to make a simple user interface.

An StdButton is a very versatile data structure that allows lots of different operations to be made on a rectangular area on a screen. An StdButton can be used to render buttons, radio buttons, text areas or custom controls. It can be used to divide the screen into a grid of elements and further subdivide elements into finer grids of more elements. StdButtons are recipients of touch screen presses or pointer clicks in a system. StdButtons also handle UI color schemes in a VSOS application.

The VSOS kernel handles the StdButtons, thus allowing them to be rendered and operated in a variety of hardware environments, such as those using text or graphic LCD displays, touch screens, video monitors or push-buttons.

The StdButton data structure is defined in <stdbuttons.h> as

```
/// Abstraction of a rectangular area on the (LCD) screen
typedef struct stdButtonStruct {
  u_int16 flags;
  s_int16 result,x1,y1,x2,y2;
  char *caption;
  void *extraRenderInfo;
  void (*render)(register struct stdButtonStruct *button,
                 register u_int16 op, register u_int16 x, register u_int16 y);
} StdButton;
```

An StdButton has

- `flags`, which control the way the button is drawn on screen,
- a `result` value, which is returned on a button press,
- `x1, y1 x2, y2` coordinates, which defines a rectangle into which the button is drawn,
- a pointer to a `caption` which holds text for the button,
- a pointer to `extraRenderInfo` which could hold for example a pointer to a bitmap to be drawn onto the button, or some other info for a custom render, and
- a pointer to a `render` function, which draws the button on a screen.

An StdButton can have the following flags, which may affect its rendering:

- `BTN_NORMAL`: This is actually not a flag, it's a value of zero, which meanst that the button is a default ("simple") button. Setting other flags may make the button into something else than a simple button.
- `BTN_PRESSED`: This means that the button is currently pressed. In case of a touch screen it means that there is a finger currently pushing this button down.
- `BTN_NO_BACKGROUND`: Don't render any background pixels for this button.

- BTN_NO_BEVEL: Don't render any bevel (edges) for this button.
- BTN_LOWERED: This means that the button is currently in a lowered position, but it doesn't necessarily mean that any finger is currently touching the button.
- BTN_NO_FACE: Don't render the button faceplate.
- BTN_HIGHLIGHTED: Render this button in a highlighted way.
- BTN_NO_CAPTION: Don't draw any caption text for the button.
- BTN_TEXT: The button is not a push-button, it's a text area.
- BTN_DISABLED: The button is disabled, it will not respond to any clicks and may be rendered differently to indicate that it's disabled.
- BTN_CHECKABLE: This button has a check mark, which is or is not currently checked.
- BTN_CHECKED: The button's check mark is currently checked.
- BTN_COLORMOD1 and BTN_COLORMOD2: Setting these flags select one of three alternative colorsets for the button.
- additionally, BTN_INVISIBLE is a set of flags BTN_NO_BACKGROUND | BTN_NO_BEVEL | BTN_NO_FACE | BTN_NO_CAPTION | BTN_DISABLED. If a button is set to be invisible and disabled, it should have no effect on the rendering and it should not receive any clicks. Additionally, if the BTN_TEXT flag is set, it is useful for defining text areas on the screen.

To learn how to use StdButtons, it's best to take a look at the template code.

First the code allocates some memory space for holding the button info. The line `StdButton buttons[10+1]=0;` reserves space for 10 buttons plus one "empty" button to mark the end of the buttons. In this case. memory is reserved from the global variables section, but it could just as well be a static local variable of main().

It's important that the last button's `result` value is zero. When you call the library routine RenderStdButtons(), it will start rendering from the first button on a list and continue rendering buttons until it reaches a button with result value 0. (Alternatively you could not use RenderStdButtons() and render the buttons one by one, in which case you would not need the last "empty" button). This makes it convenient to allocate one array for the buttons and then have a varied number of buttons in that array.

In the beginning of main(), two pointers are declared. *currentButton is set to point to the first button in the buttons[] array. *textArea is another pointer to an StdButton, it will later be used to show a convenient way to mark which button is used as a "text area" on the screen.

`console.Ioctl(&console, IOCTL_START_FRAME, "Stdbuttons Demo");` sends a START FRAME io control message to the console device. Upon this ioctl, the console will clear the screen and draw a new title bar on the display with "Stdbuttons Demo" as the title caption. It will also set the screens current usable area, the *clipping rectangle*, to be all area below the title.

The next line `SetVirtualResolution(3,4);` divides the clipping rectangle into three horizontal blocks and four vertical blocks. This "virtual resolution" information is used by

CreateStdButton() function to calculate the physical coordinates for a button for whatever display device happens to be present.

Now that the screen has been divided into a 3 x 4 units logical grid, the next lines create a set of buttons and a text area into that grid. Below you can see the buttons renderd by the ili9345 driver by VSOS 0.22 kernel stdbuttons, using the VSOS 0.22 default colorset. Other renderers might draw the buttons differently, but the general layout should be similar.

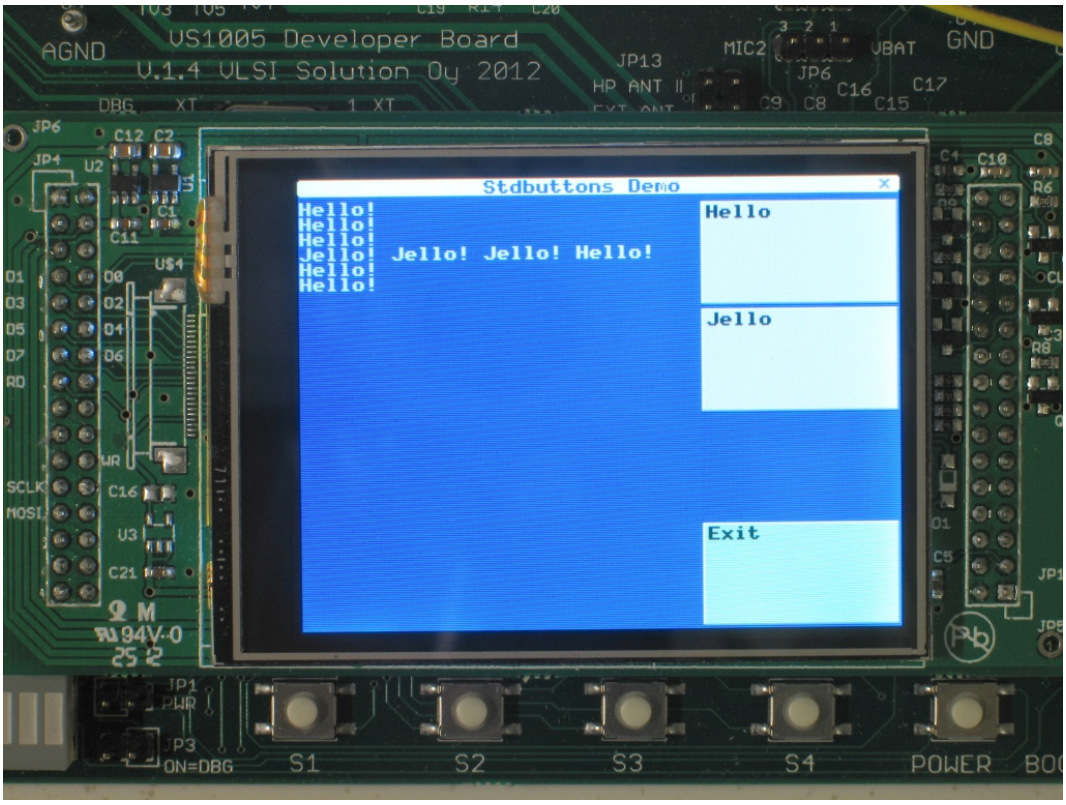


Figure 2: StdButtons Demo running on the VS1005 Developer Board

The "Hello", "Jello" and "Exit" buttons are created by code lines

```
CreateStdButton(currentButton++, 1, BTN_NORMAL, 2,0, 1,1, "Hello");
CreateStdButton(currentButton++, 2, BTN_NORMAL, 2,1, 1,1, "Jello");
CreateStdButton(currentButton++, 3, BTN_HIGHLIGHTED, 2,3, 1,1, "Exit");
```

The "Hello" button is a normal button at logical screen coordinates (2,0) with size of (1,1) logical screen units. CreateStdButton() function calculates physical pixel values based on the grid set by SetVirtualResolution() inside the display's current clipping rectangle. Since the virtual resolution is 3x4, the resuls is a button in the top right corner of the screen under the title, with width of 1/3 of the area's width and height of 1/4 of the area.

The "Hello" button is set to be a "normal" button, it will have the caption "Hello" and it will return the value 1 when pressed. Other buttons are created accordingly. The "Exit" button will be rendered in a highlighted style, e.g. with a slightly different color.

The button is drawn so that two buttons that are edge to edge with each other render in a clean manner within the display's capabilities. The VSOS 0.22 / ili9345 renderer leaves a guard area of two pixels between two adjacent buttons.

Besides the three pushable buttons, a text area is created on the screen with

```
CreateStdButton(textArea=currentButton++, BTN_END, BTN_INVISIBLE | BTN_TEXT,
                0,0, 2,4, "");
```

Several important properties of the declaration must be considered. Firstly the button has a return value of BTN_END, which is zero. This means it is not rendered by a call to RenderStdButtons() and any buttons after this one are also not rendered. It is the end of the `buttons[]` list.

However, it carries meaningful information. It has the physical coordinates of an area on the screen and it defines the colorset of the area. It has the flagset `BTN_INVISIBLE` which should prevent its rendering in case it's explicitly rendered. It has the flag `BTN_TEXT` which means that any text rendered inside the area (caption or some other text) uses `textColor` and `textBackground` colors instead of `buttonText` and `buttonFace`.

Upon the creation of the "button", a pointer to it is also stored into a local pointer `textArea`, which has no special meaning. It's just a note to self that this button will be used as the text area on the screen. That is accomplished by calling

```
SetClippingRectangleToButton(textArea);
```

which sets the current clipping rectangle of the screen to be the area of that button and the current screen colors to be the color of that button. All subsequent printf()'s will then go to that area using that area's color.

Using the console start frame ioctl, SetVirtualResolution(), CreateStdButton() and SetClippingRectangleToButton() functions together is a powerful toolset! It allows you to easily write code which is LCD resolution independent and works without changes on a variety of different display devices with respect to the colorset the user has chosen.

After creating the buttons, a RenderStdButtons() call with a pointer to the beginning of the buttons list as a parameter will draw the buttons on the screen. After the call, the program enters a loop where it calls GetStdButtonPress() repeatedly. These calls allow the StdButtons code to read input devices such as the touchscreen's touch panel and to update the screen for example to draw the buttons as lowered or normal based on the touch info.

When a button is pressed, GetStdButtonPress() will return the button's result value. Based on the value, the code does some printf()'s to the screen or finally exits.

7.4 VSOS Main Menu

This section of the document is work in progress and will be appended in the future.

The Main Menu application is a fairly complex template, which demonstrates some aspects of the StdButtons interface. The most interesting piece of the code is the MakeFileButtons function, which searches the system disk for files matchng *.app and generates a list of StdButtons for the files. The file searching routines use malloc() and mallocy() to allocate buffers, so at the beginning of the application, a memory area is set for malloc to use:

```
__x s_int16 mallocX[1024]; // for malloc use
__y s_int16 mallocY[1024]; // for malloc use

int main(void) {
       ... other code here ...
  // Setup malloc area
  __InitMemAlloc(mallocX, sizeof(mallocX), mallocY, sizeof(mallocY));
       ... other code here ...
}
```

The file searching routines return a "double string" for the file name; first there is the short file name ("8.3" style file name) which ends in a zero. This is immediately followed by the long file name or a lower-case copy of the short file name if no long file name is present.

For each of the .app files, an StdButton is created with the long file name as the caption. The position of the button is determined by using a special BTN_SLOT property of StdButtons: if the x coordinate of the button is BTN_SLOT (-1), then the y coordinate is used to set the xy start position of the button so that all logical positions on the current screen area are enumerated from left to right, top to down starting with 0.

```
#define FILE_NAME_CHARS 256
char fileName[FILE_NAME_CHARS]=""; //Temporary storage for a file name

#define NUMBER_OF_BUTTONS 20 //up to this many buttons
StdButton buttons[NUMBER_OF_BUTTONS+1] = {0};
char buttonFileName[NUMBER_OF_BUTTONS][FILE_NAME_CHARS] = {0};

void MakeFileButtons(StdButton *buttons) {
  FILE *fp = NULL;
  volatile int ff;
  volatile int i = 0;
  u_int16 number_of_files;

  InitDirectory(&fp);  // setup file searching
  ResetFindFile(fp);
```

```
  number_of_files = FindFile(fp, fileName, FILE_NAME_CHARS, "app", ofmLast,
                             etFile);
  if (number_of_files < 5) {
    number_of_files = 5;
  }
  if (number_of_files > NUMBER_OF_BUTTONS-3) {
    number_of_files = NUMBER_OF_BUTTONS-3;
  }
  SetVirtualResolution(1,number_of_files); // for calls to CreateStdButton
  ResetFindFile(fp);

  // Find each .app file on the system disk and create a button for them.
  while ((ff = FindFile (fp, fileName, FILE_NAME_CHARS, "app", ofmNext,
          etFile)) >= 0) {
    //FindFile returns a double-string which has both the short and long name.
    char *longName = &buttonFileName[i][0]+strlen(fileName)+1;

    //copy the file name into array buttonFileName for safe keeping...
    memcpy(&buttonFileName[i][0], fileName, FILE_NAME_CHARS);

    CreateStdButton(&buttons[i],i+1,0,BTN_SLOT,i,1,1,longName);

    if (++i>=NUMBER_OF_BUTTONS) {
      goto finally;
    }
  }
  finally:
  EndDirectory(&fp);
}
```

The main code shows these buttons and others that show some information and calibrate the touch screen.

If the user presses one of the buttons that select an application file to run, the code selects the next application to be run by fopening appFile and returning to the kernel. This is done with:

```
  fclose(appFile); //Close the handle to current appFile;
  sprintf(fileName,"S:%s",buttonFileName[i-1]);
  appFile = fopen(fileName,"rb"); //Open a new appFile
  if (appFile) {
    // Exit with appFile pointing to another file.
    // The kernel will then load the next appFile.
    return S_OK;
  }
```

7.5 User Interface Kernel Module

This template implements the AU.SYS kernel module, which can be used to automatically generate a user interface for a solution, which uses the Model-View-Controller approach of VSOS programming. An example that uses this module is presented later in Chapter 7.8, *MP3 Player using User Interface Kernel Module*.

A kernel module is like a simple application, which has the main function declared as

```
int main(int service, void *data)
```

An application can call a kernel module with the function call

```
int CallKernelModule(register u_int16 appIdent, int service, void *data);
```

The kernel resolves the u_int16 appIdent as two ASCII characters (e.g. 'A'*256+'U' => "AU"), loads a corresponding SYS file (e.g. "SYS/AU.SYS") and calls its main function with the `service` and `*data` parameters. The result value of the module's main function is returned to the caller.

Kernel modules can work like shared libraries. It's possible for example to implement some commonly used ui dialogues as kernel modules, so that all applications can call them. The kernel modules are overlaid, so if you call one kernel module and then another kernel module, the first one is overwritten with the second. When you call the first one again, it's loaded again. There's no practical limit to how many kernel modules can exist in a system. Using them is one way to extend the application's code space.

In VSOS 0.22 kernel modules can have 1 kiloword of code and can use 256 words of X and 128 words of Y scratchpad RAM. The RAM content is lost when the module is overlaid, so all persistent data should be accessed through the `*data` pointer parameter. But the scratchpad X and Y areas are useful for declaring constant strings, or for example temporary lists of StdButtons.

The AU.SYS kernel module has three services, which are defined in <uimaker.h>.

UI_CREATE reads a list of uiMessages and creates a set of StdButtons and other screen items based on it.

UI_MESSAGE handles an uiMessage input, for example updates the name of the song, artist or album on the screen.

UI_HANDLE runs one iteration of StdButtons rendering and returns any StdButton press to the caller.

The best way to learn to understand how the module works is to modify the source code, recompile it to AU.SYS, copy it to the system SD card to the SYS subdirectory and try it by running the MP3 player application that uses it. You can be very creative with the AU.SYS module. By changing the AU.SYS module, you can implement a completely new look and feel to the "MP3 Player using AutoUI" MP3 player template.

7.6 Simple MP3 Player

This template code opens TEST.MP3 file from the system disk and plays it.

```
#include <vo_stdio.h>
#include <stdlib.h>
#include <codec.h>
#include <sysmemory.h>
#include "decodeAudio.h"
__x s_int16 xmemPool[7500];   // Minimum amount of X heap for MP3 playing
__y s_int16 ymemPool[3900];   // Minimum amount of Y heap for MP3 playing

int main(void) {
  FILE *inFp = NULL;          // the input file
  AUDIO_DECODER *dec = NULL;  // the decoder
  const char *eStr = NULL;    // decoder's result string
  int eCode = 0;              // decoder's result code
  int result = S_OK;          // application's result code
  __InitMemAlloc(xmemPool, sizeof(xmemPool), ymemPool, sizeof(ymemPool));

  inFp = fopen("S:TEST.MP3", "rb"); // Open the input file
  if (!inFp) {
    result = S_ERROR;
    goto finally;
  }

  /* We could give a hint of the the file format, but we'll leave it to
  the decoder to determine. This is OK if the file is seekable. */
  dec = CreateAudioDecoder(inFp, stdaudioout, NULL, auDecFGuess);
  if (!dec) {
    printf("Couldn't create decoder\n");
    result = S_ERROR;
    goto finally;
  }
  eCode = DecodeAudio(dec, &eStr); //Decode audio

  finally: //Clean-up
  if (dec) {
    DeleteAudioDecoder(dec);
    dec = NULL;
  }
  if (inFp) {
    fclose(inFp);
    inFp = NULL;
  }
  return result;
}
```

The "Simple MP3 Player" template demonstrates the minimum steps to open a file, play it and return. Because it runs in a single thread only, it does not have a user interface and there's no way to control the decoder while it is playing.

Usually the MP3 decoding is best done in a separate thread, and actually this template is something that you might put in that thread and then use another thread to control the decoder. That's covered in the "Simple MP3 Player with Threads" template.

Memory usage considerations

It's a good idea to study this example in detail to get a good idea of what's going on in the VS1005 when MP3 decoding is in progress. You need to setup some heap memory for the MP3 decoder by providing mallocX and mallocY areas and calling __InitMemAlloc() just like in the "VSOS Main Menu" template. Compare the heap sizes with the Main Menu template. The Main Menu used the heap for directory searching, and 1 kiloword of X and 1 kiloword of Y memory was used.

If you make a complex application, you may need to plan your memory usage. As a reminder, the VS1005 has 32 kilowords (64 kilobytes) of X data memory and 32 kilowords (64 kilobytes) of Y data memory. It's shared between the ROM services, the kernel, stack, your application code and the audio decoder. The MP3 decoder requires about 7.5 kilowords X and 3.9 kilowords Y memory. Other decoders might require more memory. VSOS 0.22 kernel allows applications to use 16 kilowords X and 16 kilowords Y by default, and the MP3 decoder heap is allocated from that space. The memory layout may be different in future kernels and it can be tweaked by using a custom kernel.

It's also possible to make a kernel which would use a single heap which is shared between the kernel, drivers, decoders and applications. But in that case the product design and testing becomes critical, as a single badly behaving application, which exits without freeing memory, could eat the memory from other applications.

VS1005 Memory			
Memory Type	Kilowords	Kilobytes	Notes
code RAM	32	128	Kernel and application code
X data RAM	32	64	Kernel and application X data
Y data RAM	32	64	Kernel and application Y data
code ROM	32	128	ROM services, kernel internal functions
X data ROM	32	64	Constants, Sine tables, fonts etc
Y data ROM	32	64	Constants, Sine tables, fonts etc
Total code RAM	32	128	Code wordlength is 32 bits
Total data RAM	64	128	Data wordlength is 16 bits
Total code ROM	32	128	
Total data ROM	64	128	
Total RAM		256	kilobytes
Total ROM		256	kilobytes
Total Memory		512	kilobytes

7.7 Simple MP3 Player with Threads

This template is a combination of the "StdButtons" template and the "Simple MP3 Player" template.

The StdButtons' functions are rewritten to do some simple controls directly to the audio decoder structure. It's just an example to demonstrate that it is possible to affect the functionality of the audio decoder from the user interface thread.

The audio decoding itself is similar to the "Simple MP3 Player" template.

7.8 MP3 Player using User Interface Kernel Module

This template shows the recommended way to make MP3 player applications (and, in the future, players of other formats, too) with user interfaces. The code is highly modular and structural and uses the Model-View-Controller "MVC" style of VSOS programming.

The MP3 decoder and the user interface are running in separate threads. The MP3 decoder comes from the file "mp3model.c" and is implemented as a *model*, which is controlled by specified *user interface messages* or "*uiMessages*". The model also generates uiMessages, which can be received to display a *view* of the state of the model.

The User Interface is drawn by a kernel module named "AU" (after "Automatic UI") and loaded from the system file "S:SYS/AU.SYS". That kernel module is adaptive: it reads a list of input and output uiMessages of the *model* and renders a screen, which shows the values of model outputs and uses StdButtons to create set of buttons that correspond with the model inputs. If you modify the model's list of uiMessages, the user interface changes accordingly.

The default AU.SYS module is useful for quickly testing a model's operation on a variety of displays. For a final product, you may wish to modify the AU.SYS module to render the user interface with a look and feel, which is better suited to your product. For more information on this, see Chapter 7.5, *User Interface Kernel Module*.

7.8.1 The MP3 Decoder Model mp3model.c

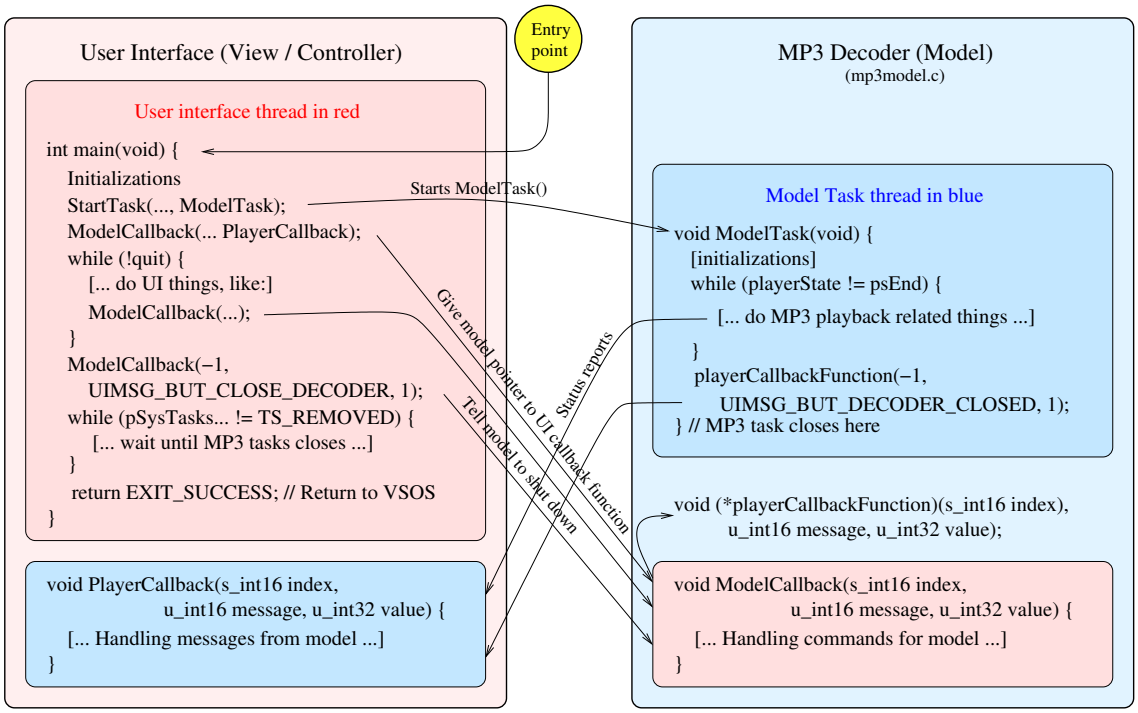


Figure 3: MVC MP3 Player with User Interface and MP3 Decoder threads.

Figure 3 shows how the User Interface main thread starts, controls, receives messages from and finally shuts down the MP3 Player thread.

The MP3 decoder model encapsulates the inner workings of MP3 decoding software functionality into a VSOS *model*. The model is independent from any user interface, so you can use it with any user interface without having to modify the model.

Also, some user interfaces can work with several different models, which can be very useful for testing and developing new functionality in products.

To make use of a model, you need to include the file `<mp3model.h>`.

Before you start the model, you should initialize a memory pool for it as shown below:

```
static s_int16 __x xmemPool[8000];
static s_int16 __y ymemPool[4000];

__InitMemAlloc(xmemPool, sizeof(xmemPool), ymemPool, sizeof(ymemPool));
```

This makes it possible to for the model to dynamically allocate memory areas for the decoder.

You can start the model by calling

```
// Start the MP3 decoder task to run in the background
StartTask(TASK_DECODER, ModelTask); // ModelTask() in mp3model.c/.h
Yield(); // Release the CPU to allow the MP3 decoder task to initialize
```

This starts a new thread for the model, running at a priority higher than the user interface. The model has a higher priority to make it sure that audio is uninterrupted even in the case where there is a shortage of processing time for all threads.

You can control the model's operation by sending uiMessages to the model. Messages are sent by calling the `ModelCallback()` function. The first call should be

```
ModelCallback(-1, UIMSG_VSOS_SET_CALLBACK_FUNCTION,
                (u_int32)PlayerCallback);
```

where you give the model your User Interface's callback function (in this example called `PlayerCallback()`), and which has the prototype

```
void *PlayerCallback(s_int16 index, u_int16 message, u_int32 value);
```

This will set a function pointer `playerCallbackFunction` in the model. After this call, whenever the model needs to send a message, like a file name listing, playback time, or just a message that it has finished playing a song, it will report that by calling the function. As an example, each second when a file is being played, the model will update playback time by calling the User Interface callback function like this:

```
playerCallbackFunction(-1, UIMSG_U32_PLAY_TIME_SECONDS, seconds);
```

The model is closed by calling

```
ModelCallback(-1, UIMSG_BUT_CLOSE_DECODER, 1);
```

When you have asked to model to close, it will first close down everything it is doing. It will then respond by calling the UI callback:

```
playerCallbackFunction(-1, UIMSG_BUT_DECODER_CLOSED, 1);
```

and after that it will close its thread.

You should wait for its thread to stop by running a waiting loop in your application's main thread such as

```
while (pSysTasks[TASK_DECODER].task.tc_State
       && pSysTasks[TASK_DECODER].task.tc_State != TS_REMOVED) {
  Delay(1); //Release the CPU for 1 milliseconds
  UiHandler(); //Call the user interface handler
}
```

Only after the model thread is terminated are you free to exit your main thread.

8 The VSOS API Reference

This chapter defines the VSOS Application Programming Interface. The description of each API call should be looked up from the kernel header files. This chapter lists the API items and the header files from where you can look for more information.

8.1 API Function Calls and system variables

This is a list of VSOS API functions, which the application can call, and the system variables, which are shared between the kernel and the application.

__nextDeviceInstance
vsos.h extern u_int16 __nextDeviceInstance;

appFile
vsos.h extern VO_FILE *appFile;

BiosService
vsos.h auto u_int16 BiosService(u_int16 service,...);

CallKernelModule
vsos.h int CallKernelModule(register u_int16 ident, int service, void *data);

CommonErrorResultFunction
vsos.h ioresult CommonErrorResultFunction();

CommonOkResultFunction
vsos.h ioresult CommonOkResultFunction();

console
vsos.h extern DEVICE console;

currentTime
vsos.h extern TIMESTRUCT currentTime;

get_time_from_rtc_hook_ptr
vsos.h extern void *get_time_from_rtc_hook_ptr;

GetTouchLocation
touch.h u_int16 GetTouchLocation (u_int16 *x, u_int16 *y);

InitDisplay
lcd.h u_int16 InitDisplay (u_int16 display_mode);

KernelService
kernelservices.h auto ioresult KernelService (u_int16 service,...);

lastErrorMessagePtr
vsos.h extern const char *lastErrorMessagePtr;

lcd0
lcd.h extern lcdInfo lcd0; ///< Information for the main display.

LcdFilledRectangle
lcd.h u_int16 LcdFilledRectangle (u_int16 x1, u_int16 y1, u_int16 x2, u_int16 y2, u_int16 *texture, u_int16 color);

LcdTextOutXY
lcd.h u_int16 LcdTextOutXY (u_int16 x, u_int16 y, char *s);

osVersion
vsos.h extern u_int16 osVersion;

pColors

pSysTasks
vsos.h extern struct SysTask *pSysTasks;

pTouchInfo
vsos.h extern s_int16 *pTouchInfo[];

SetPower

SetUartSpeed
uartSpeed.h auto void SetUartSpeed(u_int32 bitsPerSecond);

StartFileSystem
vsos.h FILESYSTEM *StartFileSystem(DEVICE *dev, char *name);

stdaudioin
vsos.h extern VO_FILE *stdaudioin;

stdaudioout
vsos.h extern VO_FILE *stdaudioout;

sys_error_hook_ptr
vsos.h extern void *sys_error_hook_ptr;

sys_report_hook_ptr
vsos.h extern void *sys_report_hook_ptr;

SysError

	vsos.h	ioresult SysError (const char *errorMsg);
SysReport		
	vsos.h	ioresult SysReport (const char *msg);
vo_fat_allocationSizeClusters		
	vo_fat.h	extern u_int16 vo_fat_allocationSizeClusters;
fclose		
	vo_stdio.h	#define fclose vo_fclose
	vsos.h	ioresult vo_fclose(register __i0 VO_FILE *stream);
feof		
	vo_stdio.h	#define feof vo_feof
	vsos.h	ioresult vo_feof(VO_FILE *stream);
ferror		
	vo_stdio.h	#define ferror vo_ferror
	vsos.h	ioresult vo_ferror(VO_FILE *stream);
fflush		
	vo_stdio.h	#define fflush vo_fflush
	vsos.h	ioresult vo_fflush(VO_FILE *stream);
fgetc		
	vo_stdio.h	#define fgetc vo_fgetc
	vsos.h	int vo_fgetc(VO_FILE *stream);
fgets		
	vo_stdio.h	#define fgets vo_fgets
	vsos.h	char *vo_fgets(char *str, int n, VO_FILE *stream);
vo_filesystems		
	vsos.h	extern FILESYSTEM *vo_filesystems[];
fopen		
	vo_stdio.h	#define fopen vo_fopen
	vsos.h	VO_FILE *vo_fopen(const char *filename, const char *mode);
fputc		
	vo_stdio.h	#define fputc vo_fputc
	vsos.h	int vo_fputc(int ch, VO_FILE *stream);
fputs		
	vo_stdio.h	#define fputs vo_fputs
	vsos.h	int vo_fputs(const char *str, VO_FILE *stream);
fread		

vo_stdio.h #define fread vo_fread
vsos.h u_int16 vo_fread(void *ptr, u_int16 size, u_int16 nobj,
VO_FILE *stream);

fseek
vo_stdio.h #define fseek vo_fseek
vsos.h ioresult vo_fseek(VO_FILE *stream, s_int32 offset, s_int16 origin);

ftell
vo_stdio.h #define ftell vo_ftell
vsos.h u_int32 vo_ftell(VO_FILE *stream);

fwrite
vo_stdio.h #define fwrite vo_fwrite
vsos.h u_int16 vo_fwrite(const void *ptr, u_int16 size, u_int16 nobj,
VO_FILE *stream);

vo_max_num_files
vsos.h extern u_int16 vo_max_num_files;

vo_osMemorySize
vsos.h extern u_int16 vo_osMemorySize;

vo_osMemoryStart
vsos.h extern u_int16 *vo_osMemoryStart;

vo_pdevices
vsos.h extern DEVICE *vo_pdevices[26];

vo_pfiles

vo_StartOS
vsos.h ioresult vo_StartOS(u_int16 *osMemPtr, u_int16 osMemSizeWords);

vo_stderr
vsos.h extern VO_FILE *vo_stderr;

vo_stdin
vsos.h extern VO_FILE *vo_stdin;

vo_stdout
vsos.h extern VO_FILE *vo_stdout;

vo_unget
vo_stdio.h #define ungetc vo_ungetc
vsos.h int vo_ungetc(int ch, VO_FILE *stream);

8.2 VS1005 Low Level Calls

These are the calls to VS1005 ROM and RTOS services, which the application may be able to do. Be careful when using these services.

Forbid
exec.h __near void Forbid(void); Disables task switching.

Permit
exec.h __near void Permit(void); Enables task switching.

Yield
exec.h __near void Yield(void); Causes a reschedule.

StartTask
vsostasks.h int StartTask(int taskId, void(*func)(void));

Delay
timers.h __near void Delay(u_int16 milliseconds);

CodMpgCreate
codecmpg.h struct Codec *CodMpgCreate(void);

SetVolume
audio.h auto void SetVolume(void);

malloc
sysmemory.h void *malloc(size_t size);

mallocy
sysmemory.h __y void *mallocy(size_t size);

calloc
sysmemory.h void *calloc(size_t number, size_t size);

callocy
sysmemory.h __y void *callocy(size_t number, size_t size);

realloc
sysmemory.h void *realloc(void *ptr, size_t size);

reallocy
sysmemory.h __y void *reallocy(__y void *ptr, size_t size);

free
sysmemory.h void free(void *ptr);

freey
sysmemory.h void freey(__y void *ptr);

AllocMemX
sysmemory.h void *AllocMemX(size_t size, size_t align);

AllocMemY
sysmemory.h __y void *AllocMemY(size_t size, size_t align);

AllocMemXY
sysmemory.h void *AllocMemXY(size_t size, size_t align);

FreeMemX
sysmemory.h void FreeMemX(void *ptr, size_t size);

FreeMemY
sysmemory.h void FreeMemY(__y void *ptr, size_t size);

FreeMemXY
sysmemory.h void FreeMemXY(void *ptr, size_t size);

ReAllocMemX
sysmemory.h void *ReAllocMemX(void *ptr, size_t oldsize, size_t newsize,
size_t align);

ReAllocMemY
sysmemory.h __y void *ReAllocMemY(__y void *ptr, size_t oldsize,
size_t newsize, size_t align);

ReAllocMemXY
sysmemory.h void *ReAllocMemXY(void *ptr, size_t oldsize, size_t newsize,
size_t align);

AllocMemAbsX
sysmemory.h void *AllocMemAbsX(u_int16 addr, size_t size);

AllocMemAbsY
sysmemory.h __y void *AllocMemAbsY(u_int16 addr, size_t size);

AllocMemAbsXY
sysmemory.h void *AllocMemAbsXY(u_int16 addr, size_t size);

InitMemAlloc
sysmemory.h void InitMemAlloc(void *xstart, size_t xchars,__y void *ystart,
size_t ychars);

ReadTimeCount

	audio.h	u_int32 ReadTimeCount(void);
InitClockSpeed		
	clockspeed.h	auto s_int16 InitClockSpeed(register u_int16 extClockKHz, register u_int32 uartByteSpeed);
SetClockSpeedLimit		
	clockspeed.h	auto s_int16 SetClockSpeedLimit(register u_int32 speedLimitHz);
SetClockSpeed		
	clockspeed.h	auto s_int16 SetClockSpeed(register u_int32 clkHz);
SetClockDividers		
	clockspeed.h	auto u_int16 SetClockDividers(register u_int16 clockDividerMask);
DelayMicroSec		
	clockspeed.h	auto void DelayMicroSec(u_int32 microSec);
ForceClockMultiplier		
	clockspeed.h	auto u_int16 ForceClockMultiplier(register u_int16 clockX);
GetClockMultiplier		
	clockspeed.h	auto u_int16 GetClockMultiplier(void);

9 VS_DSP^4 Architecture

VS_DSP^4 is a RISC / DSP hybrid processor 40-/32-/16-bit core. While it is a processor capable of running general-purpose code, it has specialized Digital Signal Processing (DSP) features.

Understanding some basics of the VS_DSP^4 architecture helps the programmers of VSOS to understand why certain things are the way they are, particularly the C language extensions or limitations that have to do with VS_DSP^4 memory types data datatypes.

9.1 VS_DSP^4 Memory Types

Typically for DSP's, VS_DSP^4's memory isn't in one linearly addressable chunk. Instead it is divided into three sections, called Instruction (I-mem), X Data (X-mem), and Y Data (Y-mem), which all have their separate buses, and which are accessed with separate assembly language instructions.

Instruction memory is 32-bit memory where the executable code is loaded and executed.

There are two kinds of data memory in VSDSP: X and Y Data. There exists a 16-bit bus for each of the memories so there can be one read from or write to each of the memories in a given clock cycle. As a C convention, static variables are by default stored in X data memory.

With a single exception of JMPI, all VSDSP instructions are executed in one clock cycle.

With certain limitations an instruction can be loaded with upto three operations + two pointer updates. When hardware looping and ring buffer capability is added to this, VSDSP can at its most perform two 16-bit data reads, one 16-bit $\times$ 16-bit to 32-bit multiplication (with automatic boundary checking and saturation), one 40-bit sum (with boundary checking and saturation), two pointer updates with ring buffer boundary checks, and a loop update and loop end condition check, all of these in one single clock cycle. This capability makes it much more powerful than most other processors with similar clock ratings.

9.2 Basic VS_DSP^4 Datatypes

The basic VS_DSP^4 datatypes are 16-bit and 40-bit arithmetic data words, and 16-bit data pointers.

16-bit words map directly into the C datatypes s_int16 and u_int16. All of the arithmetic operations are available for 16-bit registers, including binary operations, addition, subtraction, and multiplication. VS_DSP^4 has 8 symmetrical 16-bit data registers.

40-bit datatypes don't have a direct C equivalent. However, limited to 32 bits they are used to implement basic long C datatypes s_int32 and u_int32. All arithmetic operations,

except for multiplication, is available for 32-bit data registers. The 8 16-bit data registers and 4 8-bit guard registers can be combined to form upto 4 40-bit data registers. As a C convention, the 32-bit words are stored in little endian fashion: the LSW is stored in the lower address, and MSW in the higher address.

VS_DSP^4 data pointers are 16 bits. Depending on the instruction, the same address can reference to either I-mem, X-mem or Y-mem.

VS_DSP^4 doesn't have floating point numbers as such, but a 48-bit non-IEEE compliant double datatype is supported by the C compiler.

9.3 VS_DSP^4, VS1005g and VSIDE Idiosyncrasies

This section describes a few idiosyncracies that a VS1005g programmer needs to know so as not to fall into some common traps of the processor architecture or the tools.

9.3.1 No 8-bit Data Types or Byte Addressing

As shown in Chapter 9.2, VS_DSP^4 doesn't have any 8-bit datatypes. It also doesn't have 8-bit addressing: I-mem is addressed in 32-bit, and X-mem and Y-mem in 16-bit words. Because of this the C compiler for VS_DSP^4 doesn't support any 8-bit datatypes. There exists a type called *char*, but that datatype is 16 bits.

Because 16 bits is the smallest addressable memory entity,
*sizeof(char) = sizeof(u_int16) = sizeof(s_int16) = sizeof(void *) = 1*,
and
sizeof(u_int32) = sizeof(s_int32) = 2.

As a result of this, also some file operations are affected, as explained in Chapter 5.3, *VSDSP Specific File I/O Beheviour*.

9.3.2 No Large Stack Memory Allocations

Unlike modern PC architectures, VS1005g does not have gigabytes of memory to spend – not even megabytes. Because of this, there is also a very limited amount of stack space. While it is ok to allocate single variables inside functions, you should not write code like this:

```
#include <vstypes.h>

auto u_int16 MyBadFunc(register u_int16 param) {
  u_int16 workSpace[256]; // BAD! This will explode our stack!
  [... some code here ...]
}
```

If you need to have a function that uses temporary space, eitherăallocate it statically, as shown here:

```
#include <vstypes.h>

auto u_int16 MyStaticFunc(register u_int16 param) {
  static u_int16 workSpace[256]; // Static allocation
  [... some code here ...]
}
```

Or, if for space and/or other reasons you need allocation to be dynamic, use malloc():

```
#include <vstypes.h>

auto u_int16 MyDynamicFunc(register u_int16 param) {
  static u_int16 *workSpace = malloc(sizeof(u_int16)*256); // Dynamic
  if (!workSpace) {
    return some_error_code;
  }
  [... function code here ...]
  free(workSpace);
  return ok_code;
}
```

9.3.3 Using Standard Input/Output on the VS1005 Development Board

As a default, VLSI Solution's standard input/output libraries and the Integrated Development Environment VSIDE use the VS3EMU emulator to handle standard input and output. The most useful way to debug output, however, is through the LCD of the development board. To activate the redirected feature, write the following line as the first program line in your code:

```
#include <vo_stdio.h>    /* Make this your first program line. */
```

For additional information, see Chapter 5.2.

10 Audio

VSOS makes it very easy to produce and record sound with its standard-C-like interfaces to audio. Instead of being forced to use audio-specific I/O routines, VSOS makes audio look just like files: you open sound files, read from and write to them, then finally close them. And even the opening and closing parts are optional for those who need just one sound source and destination. Actually, for the moment the audio open and close commands are not needed at all because only one audio input/output stream can be open at any given time.

10.1 Standard Audio

VSOS offers the user a standard audio source and destination. Called *stdaudioin* and *stdaudioout*, they are to sound much like *stdin* and *stdout* are to standard input and output in standard C. It is not allowed for the user to close standard audio input or output files, but the user may modify their parameters.

VSOS offers standard audio input and output in the form of *stdaudioin* and *stdaudioout*. Initially, the input is connected to analog line input pins LINE1_1 (left) and LINE1_3 (right), and output is connected to analog output pins LEFT and RIGHT.

Both standard audio input and output open in stereo, 16-bit, 48 kHz mode. These parameters can be changed by the user, with some hardware-dependent limitations.

The user may use all standard read and write operations to read from and write to standard audio. For performance reasons it is, however, recommended that fread() / fwrite() functions are used to handle many samples at the time instead of character-based operations like fgetc() and fprintf().

Standard audio may not be closed or reopened by the user. However, input and output may be redirected (output redirecting not supported yet). For how to redirect the audio input to come from other input pins than the defaults, see Chapter 10.3.1.

If there are more than one channel of sound, samples from the audio channels are stored in an interleaved fashion. In 32-bit mode, the least significant bits are stored first. This is the same as the native VSDSP 32-bit word order.

Audio sample buffer word order				
Audio format	**Word 0**	**Word 1**	**Word 2**	**Word 3**
16-bit mono	Mono 0	Mono 1	Mono 2	Mono 3
16-bit stereo	Left 0	Right 0	Left 1	Right 1
32-bit mono	Mono 0 LSW	Mono 0 MSW	Mono 1 LSW	Mono 1 MSW
32-bit stereo	Left 0 LSW	Left 0 MSW	Right 0 LSW	Right 0 MSW

10.2 Simple Audio Example Program

The following audio program example reads samples, adds a low-intensity sine wave to the left channel, then outputs the samples.

```
#include <vo_stdio.h>
#include <stdlib.h>
#include <math.h>
#include <saturate.h>

#define SIN_TAB_SIZE 96
#define SIN_AMPLITUDE 1000 /* Max 32767 */

static const s_int16 __y sinTab[SIN_TAB_SIZE];

int main(void) {
  // Remember to never allocate buffers from stack space. So, if you
  // allocate the space inside your function, never forget "static"!
  static s_int16 myBuf[2*SIN_TAB_SIZE];
  int i;

  /* Build sine table */
  for (i=0; i<SIN_TAB_SIZE; i++) {
    sinTab[i] = (s_int16)(sin(i*2.0*M_PI/SIN_TAB_SIZE)*SIN_AMPLITUDE);
  }

  while (1) {
    // Read 16 stereo sample from stdaudioin.
    // By default both input and output are 16-bit stereo at 48 kHz.
    fread(myBuf, sizeof(s_int16), 2*SIN_TAB_SIZE, stdaudioin);

    // Add sine wave to the left channel.
    // Note that before addition, the buffer value is casted to s_int32
    // so that the addition will not overflow. Finally the result is
    // saturated to 16 bits.
    for (i=0; i<SIN_TAB_SIZE; i++) {
      myBuf[i*2] = Sat32To16((s_int32)myBuf[i*2] + sinTab[i]);
    }

    // Write result
    fwrite(myBuf, sizeof(s_int16), 2*SIN_TAB_SIZE, stdaudioout);
  }

  // Not really needed because there was a while(1) before
  return EXIT_SUCCESS;
}
```

10.3 Audio Input

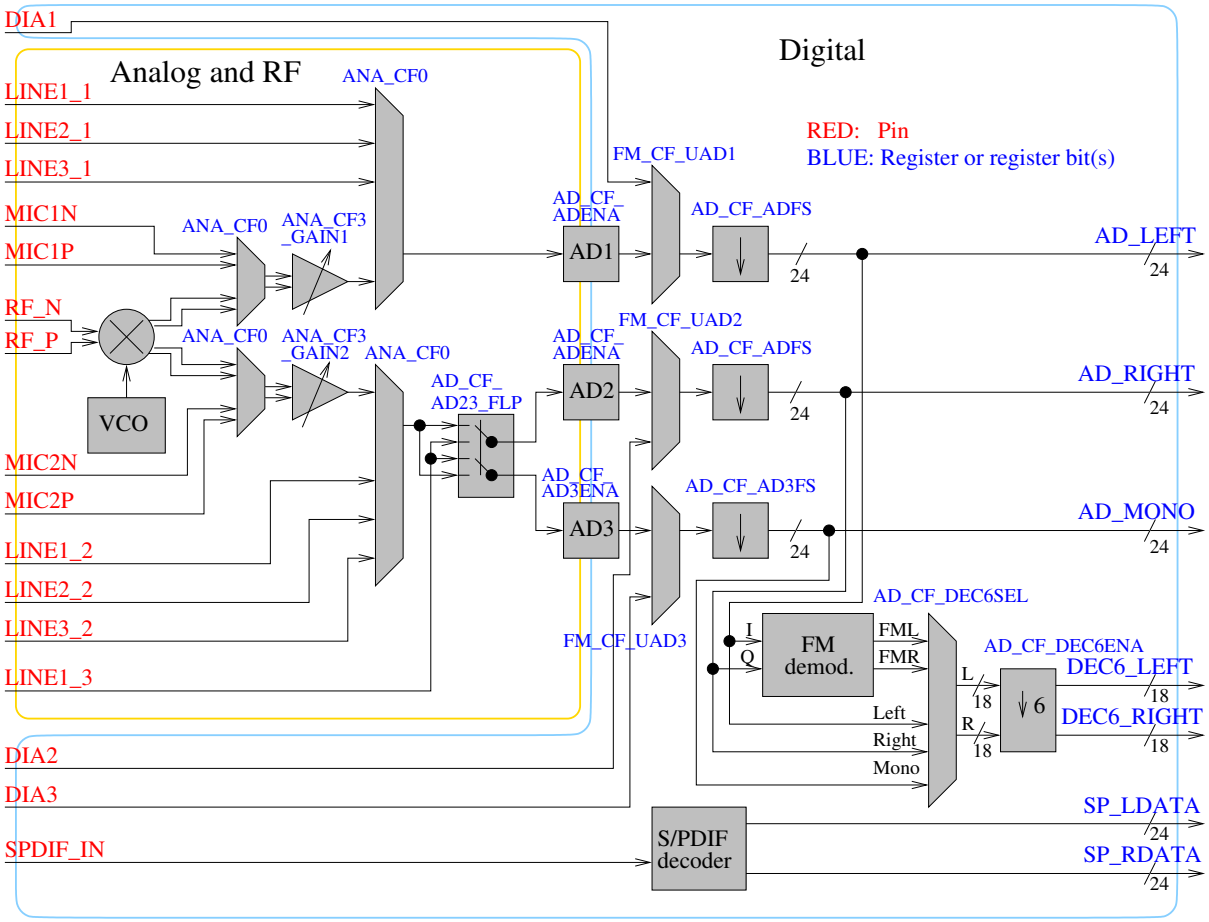


Figure 4: VS1005g recording (AD and FM) signal paths

Figure 4 presents VS1005g's input audio signal paths. In addition to paths shown in the Figure, also an I2S input is available.

The following sections present how to change an audio file that you have opened yourself, or *stdaudioin*.

10.3.1 Redirecting an Audio Input

Audio input is changed by using ioctl() methods.

In the long run any mono/stereo combination that can be implemented with the internal hardware is going to be supported, but for now only the stereo input paths listed in this section are supported.

Any analog or DIA input selection can be connected through the down-by-six decimator. In the VS1005g hardware, this will mean that audio is sent through the down-by-6 decimator to the DEC6_LEFT and DEC6_RIGHT registers instead of the default AD_LEFT and AD_RIGHT (or AD_MONO) registers. It will thus allow some sample rates not normally available. For further discussion on available input sample rates, see Chapter 10.3.4.

Set input to Line1_1 + Line1_3 stereo:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SELECT_INPUT,
                        (void *)(AID_LINE1_1|AID_LINE1_3));
```

Set input to Line1_1 + Line1_3 stereo and down-by-six decimator:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SELECT_INPUT,
                        (void *)(AID_LINE1_1|AID_LINE1_3|AID_DEC6));
```

Set input to stereo Mic1 + Mic2:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SELECT_INPUT,
                        (void *)(AID_MIC1|AID_MIC2));
```

Set input to Mic1 + Line1_3 stereo:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SELECT_INPUT,
                        (void *)(AID_MIC1|AID_LINE1_3));
```

Set input to Line1_1 + Mic2 stereo:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SELECT_INPUT,
                        (void *)(AID_LINE1_1|AID_MIC2));
```

10.3.2 Controlling Audio Input Sample Rate

To read the current intput sample rate:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int32 sampleRate;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_GET_IRATE, (void *)(&sampleRate));
printf("Current sample rate is %ld Hz\n", sampleRate);
```

To set the input sample rate, do the following. Notice that sample rate doesn't necessarily fit into 16 bits, so it needs to be passed as a pointer. Note also that in many cases there are only a limited amount of different input sample rates available. So don't assume you get an exact sample rate. Instead, check your real sample rate by rereading the sample rate immediately after writing it.

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int32 sampleRate = 48000;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_IRATE, (void *)(&sampleRate));
if (ior == S_OK) {
  ior = ioctl(fp, IOCTL_AUDIO_GET_IRATE, (void *)(&sampleRate));
  if (ior == S_OK) {
    printf("We got a sample rate of %ld Hz\n", sampleRate);
  }
}
```

10.3.3 Miscellaneous Audio Input Settings

To get the current number of intput audio channels:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 channels;
channels = ioctl(fp, IOCTL_AUDIO_GET_ICHANNELS, NULL);
if (channels >= 0) {
  printf("There are %d channels\n", channels);
}
```

To get the size of the input buffer in 16-bit words:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 bufSize;
bufSize = ioctl(fp, IOCTL_AUDIO_GET_INPUT_BUFFER_SIZE, NULL);
if (bufSize >= 0) {
  printf("Input buffer size is %d words\n", bufSize);
}
```

To get the amount of data that can be read from the input buffer without blocking the process. The result is in 16-bit words:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 bufFill;
s_int16 *buf;
bufFill = ioctl(fp, IOCTL_AUDIO_GET_INPUT_BUFFER_FILL, NULL);
if (bufFill >= 0) {
  printf("There are %d words of data in the input buffer\n", bufFill);
  fread(buf, sizeof(s_int16), bufFill, fp);  // This will never block
}
```

To set the size of the input buffer in 16-bit words:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 bufSize = 1024; /* Must be power of two */
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_INPUT_BUFFER_SIZE, (void *)bufSize);
```

10.3.4 Audio Input Sample Rate Considerations

The limitations on input sample rates are listed below. All values are calculated with the nominal XTALI = 12.288 MHz crystal.

FM:
XTALI/64/6 = 32 kHz, using down-by-six decimator.

Other analog inputs:
XTALI/{64,128,256,512} = 192, 96, 48, 24 kHz.
XTALI/6/{64,128,256,512} = 32, 16, 8, 4 kHz, using down-by-six decimator.

10.4 Audio Output

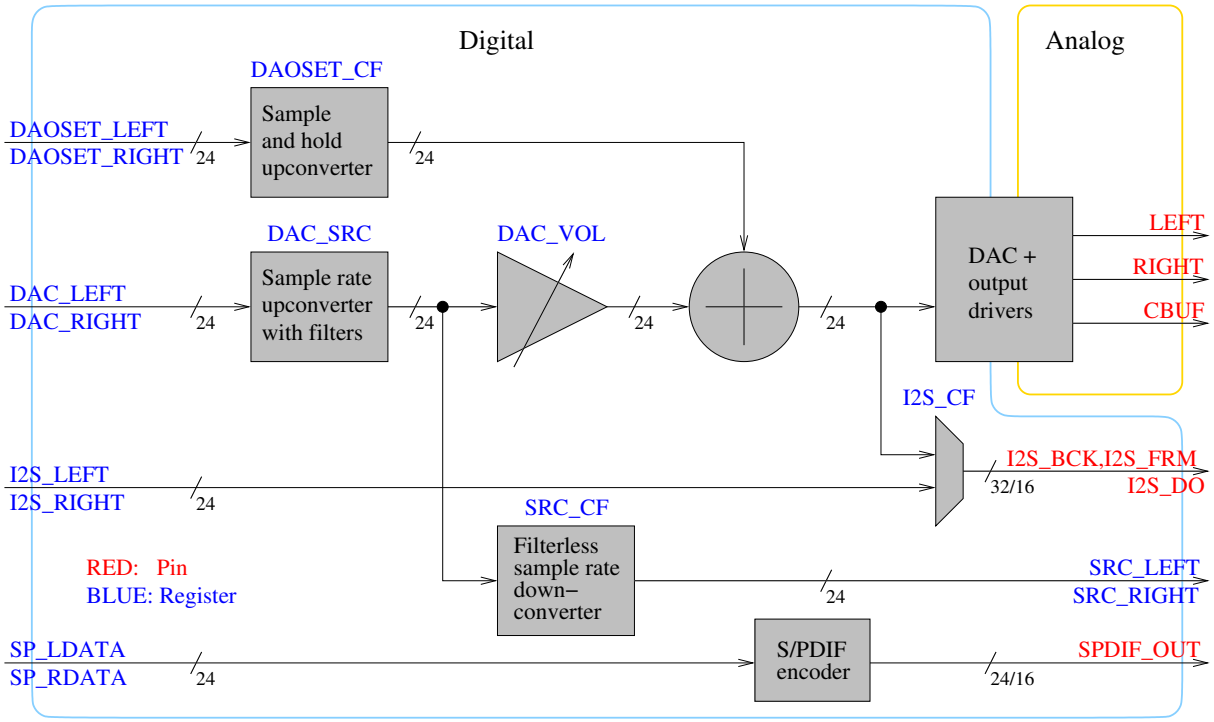


Figure 5: VS1005g playback (DA) audio paths.

Figure 5 shows the VS1005g audio playback paths. Currently the main audio path through DAC_LEFT and DAC_RIGHT registers is supported, as well as redirecting the same data to I2S and S/PDIF. As a default, all of these options are turned on.

The following sections present how to change an audio file that you have opened yourself, or *stdaudioout*.

10.4.1 Redirecting an Audio Output (new for v0.24)

Audio output that is being sent to the DAC can also be copied to either I2S, S/PDIF, or both. To redirect audio, use one or more of the following flags:

Audio Output Redirection Options	
Name	**Description**
AOR_DAC	Use DAC (ALWAYS REQUIRED)
AOR_I2S	Use I2S, 96 kHz / 32 bits, master mode
AOR_SPDIF	Use S/PDIF, 48 kHz / 24 bits, master mode NOTE! Hi-Speed USB cannot be used with S/PDIF!

To turn all outputs on, run the following code:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_OUTPUTS,
            (void *)(AOR_DAC | AOR_I2S | AOR_SPDIF));
if (ior == S_OK) {
  printf("I2S and S/PDIF activated\n");
}
```

10.4.2 Controlling Audio Output Sample Rate

To set the sample rate, do the following. Notice that sample rate doesn't necessarily fit into 16 bits, so it needs to be passed as a pointer. Note also that in some cases there are only a limited amount of different output sample rates available. So, it may be wise to check your real output sample rate by rereading it immediately after writing it.

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int32 sampleRate = 48000;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_ORATE, (void *)(&sampleRate));
if (ior == S_OK) {
  ior = ioctl(fp, IOCTL_AUDIO_GET_ORATE, (void *)(&sampleRate));
  if (ior == S_OK) {
    printf("We got a sample rate of %ld Hz\n", sampleRate);
  }
}
```

To read the current output sample rate:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int32 sampleRate;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_GET_ORATE, (void *)(&sampleRate));
if (ior == S_OK) {
  printf("Current sample rate is %ld Hz\n", sampleRate);
}
```

10.4.3 Controlling Number of Output Audio Channels

To get the current number of output audio channels:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 channels;
channels = ioctl(fp, IOCTL_AUDIO_GET_OCHANNELS, NULL);
if (channels >= 0) {
  printf("There are %d channels\n", channels);
}
```

To set the number of output audio channels:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 channels = 2;
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_OCHANNELS, (void *)channels);
```

10.4.4 I2S Output Controls (new for v0.24)

I2S is switched automatically on, at 96 kHz and 32 bits.

Turning I2S on and off is presented in Chapter 10.4.1. Currently there are no other controls for I2S output.

10.4.5 S/PDIF Output Controls (new for v0.24)

S/PDIF is switched automatically on, at 48 kHz and 24 bits.

Turning S/PDIF on and off is presented in Chapter 10.4.1.

S/PDIF volume can be controlled in 0.5 dB steps. Full volume is presented with integer value 0, and attenuation from that is presented with 0.5 dB steps (e.g. value 20 corresponds to -10.0 dB). The default value is 4 (-2.0 dB). At the default volume setting, using S/PDIF at 48 kHz requires 0.9 MIPS processing power. 2.4 MIPS is consumed at any other volume settings.

Volume settings are less accurate when attenuation $\geq$60 dB. For complete silence, set volume to 0x7FFF.

To read the current volume setting:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 vol;  // Attenuation from maximum in 0.5 dB steps; 20 = -10.0 dB
vol = ioctl(fp, IOCTL_AUDIO_GET_SPDIF_VOLUME, NULL);
printf("S/PDIF volume is at %3.1f dB of maximum\n", -0.5*vol);
```

To set S/PDIF volume to maximum:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 vol = 0;  // Attenuation from maximum in 0.5 dB steps; 20 = -10.0 dB
ioresult ior;
ior = ioctl(fp, IOCTL_AUDIO_SET_SPDIF_VOLUME, (void *)vol);
```

10.4.6 Miscellaneous Audio Output Settings

To get the size of the output buffer in 16-bit words:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 bufSize;
bufSize = ioctl(fp, IOCTL_AUDIO_GET_OUTPUT_BUFFER_SIZE, NULL);
if (bufSize >= 0) {
  printf("Output buffer size is %d words\n", bufSize);
}
```

To get the amount of data that can be written to the output buffer without blocking the process. The result is in 16-bit words:

```
#include <vo_stdio.h>
#include <audiofs.h>
#include <devAudio.h>
FILE *fp;
s_int16 bufFree;
s_int16 *buf;
bufFree = ioctl(fp, IOCTL_AUDIO_GET_OUTPUT_BUFFER_FREE, NULL);
if (bufFree >= 0) {
  printf("%d words can be written to the output buffer\n", bufFree);
  fwrite(buf, sizeof(s_int16), bufFree, fp);  // This will never block
}
```

10.4.7 Audio Output Sample Rate Considerations

The limitations on output sample rates are listed below. All values are calculated for when used with the nominal XTALI = 12.288 MHz crystal.

Analog outputs:
$0 \ldots XTALI/2^{27} \times 1048575$ = 0...95999.9 kHz, sample rate can be adjusted with an accuracy of approximately 0.09 Hz. (Driver limits adjustment low limit to 100 Hz to avoid hanging up the system, and its resolution is only 1 Hz.)

10.5 Decoding a Compressed Audio File

VS1005g is capable of decoding compressed audio files.

The following is an example of a very simple MP3 player. By using the code in this player, you can easily add MP3 playback functionality to your own programs.

As of VSOS 0.22 the decodeAudio library only supports MP3 files, but this will be expanded in the future.

```
#include <vo_stdio.h>
#include <stdlib.h>
#include <codec.h>
#include <sysmemory.h>
#include <decodeAudio.h>

/* Following are the minimum amounts of Malloc Area for MP3 decoding to work.*/
__x s_int16 xmemPool[7300];
__y s_int16 ymemPool[3700];

int main(void) {
  FILE *inFp = NULL;
  AUDIO_DECODER *dec = NULL;
  const char *eStr = NULL;
  int eCode = 0;

  /* Initialize memory allocation pools for the decoder.
  __InitMemAlloc(xmemPool, sizeof(xmemPool), ymemPool, sizeof(ymemPool));

  /* Replace the file name you want to play here. Don't forget to begin
     the name with "S:". And make sure your SD card contains the file! */
  inFp = fopen("S:TEST.MP3", "rb");
  if (!inFp) {
    printf("Couldn't open read file\n");
    goto end;
  }

  /* We could give a hint of the the file format, but we'll leave it to
     the decoder to determine. This is OK if the file is seekable. */
  dec = CreateAudioDecoder(inFp, stdaudioout, NULL, auDecFGuess);
  if (!dec) {
    printf("Couldn't create decoder\n");
    goto end;
  }

  printf("Decoding\n");
  eCode = DecodeAudio(dec, &eStr);
  printf("Decoder returns %d, \"%s\"\n", eCode, eStr ? eStr : "(null)");

 end:
  printf("Clean-up.\n");

  if (dec) {
    DeleteAudioDecoder(dec);
    dec = NULL;
  }
  if (inFp) {
    fclose(inFp);
    inFp = NULL;
  }
  return EXIT_SUCCESS;
}
```

11 How to Write Efficient VS_DSP^4 Code

VS_DSP^4 is a RISC / DSP core. While VS_DSP^4 has some idiosyncracies typical to DSP cores, it also have some features not often available in general-purpose processors that still can be utilized even through C language, like zero overhead loops (Chapter 11.1) and ring buffer hardware (Chapter 11.3).

This chapter will explain some ways that you can use to make your code faster or smaller, or how to make it use less data space.

11.1 How to Utilize Zero Overhead Loop Hardware

VS_DSP^4 contains zero ovearhead loop hardware, which can be utilized by the C compiler if the circumstances are right.

For the compiler to be able to use the loop hardware, the compiler must be able to determine the loop count before the loop is started. To make sure of this, the following restrictions must be followed:

- The initial statement must set the loop variable to a known quantity (e.g. `i=0`).
- Loop variable may not be changed inside the loop.
- Loop end condition must be a simple expression and only dependable on the loop variable (e.g. `i<5` or `i<j`, but *not* `i<2*n+7 && j<i`.)
- Loop modifier statement must be a simple statement modifying the loop counter with a non-changing value (e.g. `i+=5`).

An example of a simple for() loop utilizing the loop hardware is presented below:

```
#include <vstypes.h>

auto void SetMemGood(register u_int16 *d, register u_int16 value,
                     register u_int16 n) {
  u_int16 i;              // Loop variable
  for (i=0; i<n; i++) {  // Simple loop
    *d++ = n;
  }
}
```

```
        .sect code,SetMemGood // 16 words
_SetMemGood:
        STX A0,(I6) ; STY A1,(I6)+1
        STX LC,(I6) ; STY LS,(I6)+1
        AND NULL,NULL,A1 ; STX LE,(I6)
        SUB A1,C1,A0
        NOP
        JCS $1                      // If n=0, goto exit code
```

```
        NOP
        SUB ONES,A0,A0              // a0 = n-1
        LOOP A0,$1-1                // Hardware loop a0+1 (=n) times
        NOP
$0:     STX C1,(I2)+1               // L1: First and only cmd of hardware loop
$1:
        LDX (I6)-1,LE
        LDX (I6)-1,LC ; LDY (I6),LS
        LDX (I6)-1,A0 ; LDY (I6),A1
        JR
        NOP
```

Compare this with another way of doing the same thing, which on surface might seem like the logical thing to do, but which doesn't allow for the C compiler and VS_DSP[4] to use its loop hardware.

```
#include <vstypes.h>

auto void SetMemSlow(register u_int16 *d, register u_int16 value,
                     register u_int16 n) {
  while (n--) {  // While loop not as good as for() with LCC
    *d++ = n;
  }
}
```

```
        .sect code,SetMemSlow // 11 words
_SetMemSlow:
        ADD C1,NULL,A0 ; STX A0,(I6) ; STY A1,(I6)

$1:     ADD A0,ONES,A1                  // L1: First cmd of loop, n--
        SUB A0,NULL,A0                  // L2: Compare old value of n with 0
        NOP                             // L3: delay slot
        JZS $8                          // L4: If n=0, end loop
        NOP                             // L5: delay slot
        J $1                            // L6: Go to beginning of loop
        ADD A1,NULL,A0 ; STX A1,(I2)+1 // L7: delay slot, update n + write val

$8:     LDX (I6)-1,A0 ; LDY (I6),A1
        JR
        NOP
```

11.2 Using Pointers Instead of Table Indexes

From the compiler's point of view it is more efficient to use pointer instead of indexing tables. Because of this, if code size and speed is essential to you, avoid writing code as follows:

```
#include <vstypes.h>

/* Table indexing loop. Not the most efficient way for VSDSP compilers */
auto void SetMemSlow(register u_int16 *d, register u_int16 value,
```

```
                    register u_int16 n) {
  u_int16 i;              // Loop variable
  for (i=0; i<n; i++) {  // Simple loop
    d[i] = n;
  }
}
```

As can be seen from the following disassembly listing, although hardware loops are used by the compiler to make the code faster, indexing requires three clock cycles per loop:

```
        .sect code,SetMemSlow // 20 words
_SetMemSlow:
        STX A0,(I6) ; STY A1,(I6)+1
        AND NULL,NULL,A0 ; MV I2,A1
        STX B0,(I6) ; STY I0,(I6)+1
        STX LC,(I6) ; STY LS,(I6)+1
        SUB A0,C1,B0 ; STX LE,(I6)
        NOP
        JCS $1                      // If n=0, goto exit code
        NOP
        SUB ONES,B0,B0              // b0 = n-1
        LOOP B0,$1-1                // Hardware loop b0+1 (=n) times
        NOP
$0:     ADD A1,A0,B0                // L1: First cmd of hardware loop
        SUB A0,ONES,A0 ; MV B0,I0   // L2:
        STX C1,(I0)                 // L3: Last cmd of hardware loop
$1:
        LDX (I6)-1,LE
        LDX (I6)-1,LC ; LDY (I6),LS
        LDX (I6)-1,B0 ; LDY (I6),I0
        LDX (I6)-1,A0 ; LDY (I6),A1
        JR
        NOP
```

Using a pointer and modifying it, as shown below, is much more efficient:

```
#include <vstypes.h>

auto void SetMemGood(register u_int16 *d, register u_int16 value,
                    register u_int16 n) {
  u_int16 i;            // Loop variable
  for (i=0; i<n; i++)  // Simple loop
    *d++ = n;
}
```

As can be seen from the following disassembled code snippet, the loop is now just as powerful as a hand optimized assembly loop: it sets one memory element each and every clock cycle (for the full listing, see SetMemGood() in Chapter 11.1).

```
        .sect code,SetMemGood // 16 words
_SetMemGood:
        [code snipped]
        LOOP A0,$1-1                // Hardware loop a0+1 (=n) times
        NOP
```

```
$0:     STX C1,(I2)+1               // L1: First and only cmd of hardware loop
$1:
        [code snipped]
```

11.3 How to Utilize Ring Buffer Hardware

On unique DSP feature VS_DSP[4] has to offer is hardware ring buffer pointer handling.

The standard VS1005g library offers ring buffer manipulation functions, presented in <ringbuf.h>.

The traditional C way of implementing a function that copies data from a linear buffer to a ring buffer could be written like this. The function takes the current ring buffer pointer as a parameter, and returns the new ring buffer pointer, which has to be stored by the caller.

```
#include <ringbuf.h>
#define RING_BUF_SIZE 80
u_int16 __y myRingBuf[RING_BUF_SIZE];

auto u_int16 __y *CopyToRingBufferSlow(register const u_int16 *s,
                                       register u_int16 n,
                                       register u_int16 __y *ringBufPtr) {
  int i;
  for (i=0; i<n; i++) {
    *ringBufPtr++ = *s++;
    if (ringBufPtr >= myRingBuf+RING_BUF_SIZE) {  // If at end of ring buffer
      ringBufPtr = myRingBuf;                     // return pointer to start
    }
  }
  return ringBufPtr;
}
```

The code above does the copying as efficiently as it can, but although it compiles to a manageable 27 instruction words, it will use a little over 8 clock cycles for each copied word.

The code may be slightly optimized by using functions that modify the ring buffer pointer using VSDSP hardware. The code below is 34 instruction words in size, and uses 8 clock cycles for each copied word (five in the loop, and three in the function ringmodY()).

```
#include <ringbuf.h>
#define RING_BUF_SIZE 80
__align u_int16 __y myRingBuf[RING_BUF_SIZE]; // Ring buffers need __align

auto u_int16 __y *CopyToRingBufferMedium(register const u_int16 *s,
                                         register u_int16 n,
                                         register u_int16 __y *ringBufPtr) {
  int i;
```

```
  u_int16 ringMod = GetRingModifier(1, RING_BUF_SIZE); // Get ring buf mod
  for (i=0; i<n; i++) {
    *ringBufPtr = *s++;
    ringBufPtr = ringmodY(ringBufPtr, ringMod); // Modify ring buf pointer
  }
  return ringBufPtr;
}
```

So, using ring buffer hardware for a simple pointer update didn't necessarily save us much processing time. But there are better ways to optimize the function.

The example that follows uses functions that use the VSDSP hardware to copy the data to the ring buffer. It compiles to 31 words, which is still larger than the original slow function. But instead of a littler over 8 clock cycles for each copied data word, this one uses only one clock cycle for each word copy:

```
#include <ringbuf.h>
#define RING_BUF_SIZE 80
__align u_int16 __y myRingBuf[RING_BUF_SIZE]; // Ring buffers need __align

auto u_int16 __y *CopyToRingBufferFast(register const u_int16 *s,
                                       register u_int16 n,
                                       register u_int16 __y *ringBufPtr) {
  static u_int16 ringMod = 0;
  if (!ringMod) {
    ringMod = GetRingModifier(1, RING_BUF_SIZE); // Get ring buf modifier
  }
  ringBufPtr = RING_XY_D(ringcpyXY(ringBufPtr, ringMod, s, 1, n));
  return ringBufPtr;
}
```

Note that in the examples above, the ring buffer was placed in __y data memory while the linear buffer data was in "normal" __x memory. This was made to make copying between ring and linear buffers faster: if both buffers had been in __x memory, it would take two clock cycles to copy one memory element. However, when they are in different memories, a copy can be performed in one clock cycle (See Chapter 9.1 for details on VS_DSP[4] memory types).

11.4 Using Register Parameters in Functions

With functions that don't have too many parameters (upto two or three pointers, plus upto three to four 16-bit parameters or upto two 32-bit parameters), the most efficient way to pass parameters are usually in registers. So, consider the following examples, written withour implicit register parameters:

```
// This would be the default way to write a function
#include <vstypes.h>
void AddValueThroughPointerSlow(s_int16 *d, s_int16 m) {
```

```
  *d += m;
}
```

The code above compiles as follows using LCC v1.42:

```
        .sect code,AddValueThroughPointerSlow // 15 words
_AddValueThroughPointerSlow:
        LDX (I6)+2,NULL
        STX I6,(I6) ; STY I4,(I6)
        LDX (I6)+1,I4
        STX A0,(I6) ; STY A1,(I6)+1
        STX I0,(I6) ; LDY (I4)-2,NULL
        LDX (I4)-1,I0
        LDX (I0),A1
        LDX (I4)+3,A0
        ADD A1,A0,A0
        STX A0,(I0)
        LDX (I6)-1,I0
        LDX (I6)-1,A0 ; LDY (I6),A1
        LDX (I4),I6 ; LDY (I4),I4
        JR
        LDX (I6)-2,NULL
```

To make the previous example more efficient and smaller, we can declare both parameters to be passed in registers instead of the stack:

```
#include <vstypes.h>
void AddValueThroughPointerOk1(register s_int16 *d, register s_int16 m) {
        *d += m;
}
```

The code above compiles into the following using LCC v1.42. As you can see, we saved 7 words of instruction space from the original function's 15:

```
        .sect code,AddValueThroughPointerOk1 // 8 words
_AddValueThroughPointerOk1:
        LDX (I6)+1,NULL
        STX A0,(I6)
        LDX (I2),A0
        ADD A0,C0,A0
        STX A0,(I2)
        LDX (I6)-1,A0
        JR
        NOP
```

It is also possible to specifically specify in which registers you want to pass your parameter in, but unless you intend to build assembly language interfaces, that rarely helps.

```
#include <vstypes.h>
void AddValueThroughPointerOk2(register __i0 s_int16 *d,
                               register __a0 s_int16 m) {
        *d += m;
}
```

The code above compiles into the following using LCC v1.42, and is the same size as the previous example:

```
        .sect code,AddValueThroughPointerOk2 // 8 words
_AddValueThroughPointerOk2:
        LDX (I6)+1,NULL
        STX A1,(I6)
        LDX (I0),A1
        ADD A1,A0,A0
        STX A0,(I0)
        LDX (I6)-1,A1
        JR
        NOP
```

A very small saving, usually one code word per function, can be obtained by adding an *auto* declaration to the function return type. This will instruct the VS_DSP[4] compiler to handle the function stack in a slightly more efficient way. So the typical best way to write this function would be:

```
#include <vstypes.h>
auto void AddValueThroughPointerGood(register s_int16 *d,
                                     register s_int16 m) {
        *d += m;
}
```

The code above compiles into the following using LCC v1.42, giving us an ultimate saving of 8 words, or little over 50 % of the original function's 15 words.

```
        .sect code,AddValueThroughPointerGood // 7 words
_AddValueThroughPointerGood:
        STX A0,(I6)
        LDX (I2),A0
        ADD A0,C0,A0
        STX A0,(I2)
        LDX (I6)-1,A0
        JR
        NOP
```

11.5 Avoid Large Char Tables

VS_DSP[4] doesn't have native 8-bit datatypes (Chapter 9.3.1). Because of this, character and short table allocations take up equal amounts of space. Thus, the following three declarations all take up the exact same amount of data space:

```
char tab1[4096];    // Char is a 16-bit entity in VSDSP4
u_int16 tab2[4096]; // 16-bit entities
u_int32 tab3[2048]; // 32-bit entities
```

So, if it fits your application, consider packing two bytes into your large 8-bit data table. There are support routines to help you move data around in this packed format.

11.6 How to Handle 8-Bit Data Packed into 16-bit Tables

Because of the missing 8-bit data types, *string.h*'s memory copying routines can only address data at 16-bit word boundaries:

```
#include <string.h>
const u_int16 s[5] = {0x0102, 0x0304, 0x0506, 0x0708, 0x090a};
u_int16 d[5];

memset(d, 0, sizeof(d)); // Clear destination
memcpy(d, s, 3);         // Copy three 16-bit words
// Now d = {0x0102, 0x0304, 0x0506, 0x0000, 0x0000}

memset(d, 0, sizeof(d)); // Clear destination
memcpy(&d[1], &s[2], 3); // Copy three 16-bit words
// Now d = {0x0000, 0x0506, 0x0708, 0x090a, 0x0000}
```

To get byte-accurate access, you can use function MemCopyPackedBigEndian(). Its protytype is:

```
void MemCopyPackedBigEndian(register __i0 __near unsigned short *dst,
                            register __a0 unsigned short dstIdx,
                            register __i1 __near unsigned short *src,
                            register __a1 unsigned short srcIdx,
                            register __b0 unsigned short byteSize);
```

The two index parameters, dstIdx and srcIdx are byte offsets to the first bytes of data to be copied. Modifying the previous example, here is what the function does:

```
#include <string.h>
const u_int16 s[5] = {0x0102, 0x0304, 0x0506, 0x0708, 0x090a};
u_int16 d[5];

memset(d, 0, sizeof(d));                        // Clear destination
MemCopyPackedBigEndian(d, 0, s, 0, 6);          // Simple copy
// Now d = {0x0102, 0x0304, 0x0506, 0x0000, 0x0000}

memset(d, 0, sizeof(d));                        // Clear destination
MemCopyPackedBigEndian(&d[1], 0, &s[2], 0, 6); // Simple copy
// Now d = {0x0000, 0x0506, 0x0708, 0x090A, 0x0000}

memset(d, 0, sizeof(d));                        // Clear destination
MemCopyPackedBigEndian(d, 2, s, 4, 6);          // Same as previous copy
// Now d = {0x0000, 0x0506, 0x0708, 0x090A, 0x0000}

memset(d, 0, sizeof(d));                        // Clear destination
MemCopyPackedBigEndian(d, 1, s, 4, 6);          // Odd destination, 6 bytes
// Now d = {0x0005, 0x0607, 0x0809, 0x0A00, 0x0000}
```

```
memset(d, 0, sizeof(d));                        // Clear destination
MemCopyPackedBigEndian(d, 2, s, 3, 6);          // Odd source, 5 bytes
// Now d = {0x0000, 0x0405, 0x0607, 0x0800, 0x0000}
```

There also exists a variant that copies Y data memory, called MemCopyPackedBigEndianYY().

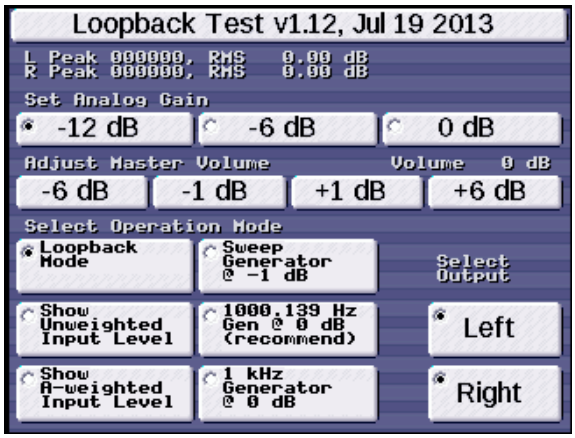
12 VSOS History

This chapter shows the version history for the last VSOS releases.

12.1 Version 0.24, 2013-07-22

VSOS Kernel 0.24 is downwards compatible with previous 0.2x versions. Its highlight features are:

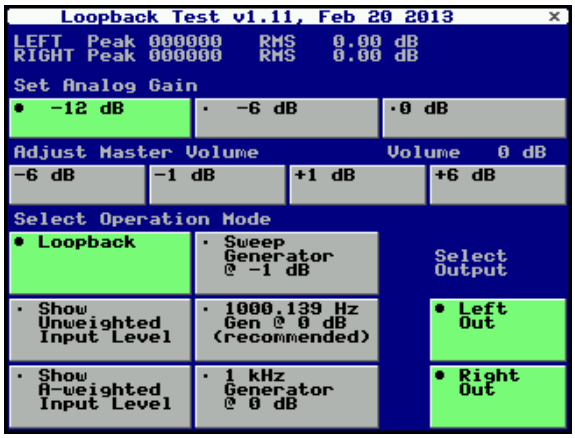
- Updated audio driver with I2S and S/PDIF output.
- Updated FAT16/FAT32 filesystem driver.
- Hardware locks and resource allocation, required for fast multithreading with new drivers.
- Improved visual style of StdButtons.



12.2 Version 0.23, 2012-12-17

This is a minor update to the Kernel. Applications are compatible with 0.22 versions.

- Improved the real-time scheduler, fixed *AllocMemY()*.
- Improved kernel module calling (added *SysCall()*).
- Fixed power-button delayed reset issue.
- Added colorscheme support also for console applications.
- Added the capability to prom the OS kernel into VS1005g internal flash.



12.3 Version 0.22, 2012-10-12

This major release of VSOS contains:

- *stdaudin* and *stdaudout* created, along with a framework to control and read from / write to audio devices just like ordinary files. While incompatible with earlier ways of handling audio, this is the way audio is intended to be from now on in VSOS.
- Most user applications are compatible with VSOS 0.21.

13 Latest Document Version Changes

This chapter describes the most important changes to this document.

Version 0.24.0, 2012-07-22

First public release for VSOS 0.24. So far, only Chapter 10.4, *Audio Output*, has been updated to show new functionality.

Version 0.22.0, 2012-11-15

First public release for VSOS v0.22.

14 Contact Information

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

Fax: +358-3-3140-8288
Phone: +358-3-3140-8200
Commercial e-mail: sales@vlsi.fi
URL: http://www.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

For technical questions or suggestions regarding this documentation, please contact
support@vlsi.fi.