Cryptic Allusion KallistiOS / Programmer's Manual

July 29, 2002

Contents

Legal

Version Info

This documentation is for KallistiOS version 1.1.8 . If the version code says "##version##", then this is an interim in-progress copy from CVS rather than from a numbered release.

Chapter 1

Overview

Welcome to KallistiOS! If this documentation looks familiar to users of libdream, that's because it is – this is basically a "port" of the documentation, with new sections added as neccessary, and other sections modified. You'll find some entire sections marked "***UNPORTED FROM LIBDREAM***" and these haven't been updated to KOS yet. Some other sections are simply unfinished as yet. Bear with us here, this is a *lot* of documentation to write.

The following is copied from the README file:

> KallistiOS is a pseudo-real-time operating system for gaming consoles, licensed under the terms of the **new** BSD license (the one without the advertising clause). It has currently been ported to the Dreamcast (tm) and Gameboy Advance (tm) platforms.
>
> Depending on whether you wish to see the fnords, you can pronounce it "kallisti-o's" (like a cereal) or "kallisti o s" (like an operating system). We like the former since it's silly to be so serious most of the time =). "Kallisti" means "to the fairest" in Greek. This was the word (so the story goes) that was inscribed on the golden apple that Eris threw into the banquet of the gods to start the Trojan war. This somehow landed her the title of matriarch of a silly religion called Discordianism, which is what the name pays homage to. If you want a short abbreviation, you can also refer to it as "KOS", which can be pronounced "k-os" (chaos =) or just plain old "k-o-s".
>
> Note that this name is *not* to be confused or associated with either the professional development company Kalisto Software(tm) or the cracking group "Kalisto".
>
> Now that that is cleared up... =)
>
> Like most embedded kernels, KallistiOS is designed to be linked directly into a program. Unlike some simple libraries, however, KallistiOS (abbreviated elsewhere as KOS) also includes a full thread scheduler, virtual file system, etc. The only real difference is that your program is linked directly into the kernel instead of being run in a "userland" state.

Note that the only active port of KOS is for the Dreamcast(tm), so the rest of this document will refer to that port unless otherwise stated.

Chapter 2

Getting Started

We assume that you have gotten a working compiler set up. The compiler you need depends entirely on the platform you wish to use KOS with. At the moment, only the DC platform is very well supported. The others may or may not compile at this time. So this document will focus on that aspect of KOS.

The first thing you'll want to do is get a working binary image of the KOS libraries. You may download KOS in binary or source form.

If you downloaded KOS in binary form, then you'll still need a binary for **genromfs** if you want to use the ROMFS file system (which will probably come in quite handy for development). Linux tends to come with this binary, BSD users can compile one using the copy included with the KOS sources, and Cygwin users will likewise need to compile it or obtain a binary from the same place you got the KOS package.

The first thing you'll want to do in either case is to customize an "environ" script. This script is run inside your shell to set various environment variables so that the KOS build scripts and tools know where to find KOS and the compiler toolchains (for DC, an SH-4 compiler, and optionally, an ARM7 compiler). Look in the **doc** tree for samples. If you are using **bash** (the default for Cygwin) then you should start with one ending in ".sh" and if you are using **tcsh** then you should start with one ending in ".tcsh". Generally it shouldn't take more than changing a few base directory variables at the top of the file, and the environ script will be ready to go. You then need to either execute this script before working ("`source environ.sh`" or "`source environ.tcsh`"), or put the appropriate command in your shell's startup script. The instructions on how to do basic shell things like that is beyond the scope of this document, so look to Cygwin and/or *nix help groups if you want to find it.

Now that you think you've got your environ script setup correctly, type **set** (under **bash**) or **setenv** (under **tcsh**) and look at the environment variables. You should see a whole bunch of them starting with "KOS". If that's true then this step is probably complete.

If you downloaded the source package for KOS, then you'll want to compile it now. Just change the directory to the root of the KOS tree (e.g., inside "kos- 1.1.8 ") and type **make**. After a few minutes everything should have completed successfully. If not, then you might need to recheck the above steps and make sure everything is set up properly. If all else fails, try asking on the *dcdev* or *cadcdev-kallistios* mailing lists.

Well, now that that is (hopefully) out of the way, we'll proceed on to writing a KallistiOS program.

Chapter 3

A Basic KOS Program

A KOS program is pretty simple and straightforward. Each C module should include **kos.h** as a system header (e.g. "`#include <kos.h>`").[1] You should have a C-standard "main", which is called `main`. It is also possible to use several directives outside your `main` function that modify the initialization and shutdown behavior of KOS. It should be instructive to follow along in one of the Hello World examples for your platform to see how a basic KOS program should look.

You can have KOS automatically initialize subsystems if you want them to be active during your program. For example, you will probably want IRQs to be active in your program. On the other hand, you might not want threading active.[2] A basic KOS init flags line looks like this:

```
KOS_INIT_FLAGS(INIT_DEFAULT);
```

The parameter is a bitmask describing what you want to be initialized by default. The constant INIT_DEFAULT is available on every platform as a generic way to initialize as much of KOS at once as is feasible. For example, on the DC, this constant enables IRQs and pre-emptive threads.

Another directive available to you is the romdisk directive. It looks like this:

```
KOS_INIT_ROMDISK(romdisk);
```

The parameter is a pointer to a ROMFS image created by **genromfs**. ROMFS was taken from Linux (the specification, not any code) and the Linux and KOS implementations are 100% compatible, except that KOS doesn't implement a few ROMFS features such as hard links and soft links. You can, for example, mount the same ROMFS image in KOS and under a loopback device in Linux. If you don't want a ROMFS, then you should use the constant `ROMDISK_NONE`. ROMFS images are generally included in the binary by using the KOS utility **bin2o**, which is included in the "utils" tree. The Hello World Makefiles show how to do this.

Note that leaving out either of the above init lines will cause KOS to use defaults. `KOS_INIT_FLAGS` defaults to using `INIT_DEFAULT` for its flags, and `KOS_INIT_ROMDISK` defaults to `NULL` (no romdisk).

By the time your `main` function is called, the basic KOS systems should be ready to go! If you want to use any of the extra/add-on libraries that come aggregated with KOS, then you will need to initialize them seperately. These might include 3D support, an MP3 or OGG playback library, KGL, etc.

For more info on the different subsystems, please see the chapter devoted to that system. For examples, see the "examples" directory off the KOS root.

[1] Alternatively it is possible to include individual headers as you use them; but with the speed of modern C compilers there is generally no good reason to do this unless you have a namespace conflict.

[2] In KOS 1.1.6 and above, threading is *always* active; it is not a matter of threaded or not, but cooperative vs pre-emptive threading. Cooperative threading is the equivalent of the old KOS non-threaded mode, it just adds a little bit to the binary size.

Now that you know what a basic KOS program looks like, you'll probably want to start figuring out how to use the various parts of the library. The following chapters are more in the way of a reference manual than a tutorial. The next chapter describes the portable parts of KOS which are present on every platform (though in some cases there are minor differences). The chapters after that describe the various platform-specific parts of KOS, and the final chapters describe the add-on libraries which are aggregated with the KOS distribution.

3.1 A Note on C++ Compatibility

KOS supports compiling with C++ support "out of the box". The headers are guarded with `extern ''C''` and we have avoided the silly new keywords in stdc++ such as "or" and "and". So you should be able to use KOS from a C++ program exactly as if you were using it from a C program, except that you must make sure you include all of the headers you need.

Chapter 4

The Fully Portable Subsystems

All of the following subsystems are written in a portable fashion so that the code should compile on any new platform for which a hardware abstraction layer is created. The HAL (located under **kernel/arch** for each platform) will be discussed in the next section.

4.1 Data Types

Before we get started on the individual systems, I figured a small word about data types is in order. Almost everything in KOS uses the following conventions for data types. KOS' architecture specific type header (**arch/type.h**) defines types *int8*, *int16*, *int32*, and *int64* to match signed integers of the appropriate number of bits. *uint8*, *uint16*, *uint32*, and *uint64* are the same, but unsigned. All of those types have corresponding volatile versions with a 'v' prefix (*vint8*, *vuint8*, etc). Booleans or countsare often stored as *int*, but counts are also often *uint32*, depending on the circumstance. Functions that can return errors generally return them as a negative *int*, and success as a zero *int*. Strings are passed around as *const char **. Individual characters are generally *int*. Several other misc types are defined, including some ANSI C compatibility types (*u_short*, etc) but those are the main ones.

4.2 ANSI C library

KOS' built-in **libc** supports most of the standard ANSI C functions, ported mostly from the BSD library. KOS also includes an integrated copy of the Newlib **libm** library so that you are guaranteed that your libraries all match up.

The libc includes the stdio file functions as well, to wrap its native VFS functions, so that you do not have to port that functionality.

`memcpy2`, `memset2`, `memcpy4` and `memset4` are hardware helper functions which work like `memset` and `memcpy`, but they copy the specified number of bits at a time. `memalign` is a memory helper function which assures alignment on a certain byte boundary for hardware buffers.

Many people have commented that KOS' libc is somewhat spartan compared to many modern libc's such as glibc, BSD, and Newlib. However, we feel like the KOS libc is probably more than sufficient for most console efforts (games, demos, etc). If you want a more complete portability layer then you might be looking in the wrong place – perhaps DC Linux or NetBSD would be better suited to your application. KOS makes no bones about being pretty bare when it comes to standard POSIX style functionality. Still, there's more than enough here to drive the console-intensive parts of your application as long as you're willing to do a bit of porting work.

4.3 VFS (Virtual File System)

KOS contains a reasonably capable VFS (virtual file system), much like a *nix. It provides the basic file operations which you have come to expect as a programmer: `open` (with path translation, thus the "virtual" part of VFS), `read`, `write`, `seek`, `tell`, `readdir`, `ioctl`, `rename`, `unlink`, `mmap`. VFS also provides an unconventional but very helpful function, `total`, which returns the total file size if possible. Some of these functions may not work on all file systems. For example, you cannot `write` to a CD or a ROMFS image. But you can `write` to a VMU on a DC. You also cannot `mmap` on most file systems, the notable exceptions being fs_romdisk and fs_vmu. Failure in these functions is indicated (as mentioned above) by a negative return value, except in the case of `open`, which may eventually be changed to conform anyway.

4.3.1 file_t fs_open(const char * fn, int mode)

Opens a file on the VFS. This function searches its VFS table for the longest match against the first of *fn*, and then passes off the rest of the functionality to the given VFS with the remainder of the path. The returned value should be considered opaque and will be zero (or `NULL` if you prefer) on error, or a valid file handle otherwise. The *mode* parameter should be one of `O_RDONLY`, `O_RDWR`, `O_APPEND`, or `O_WRONLY`, potentially logically or'd with `O_TRUNC`, `O_DIR`, or `O_META`.

4.3.2 void fs_close(file_t hnd)

Closes an opened VFS file. Handle values passed to this function (and all other VFS functions below) are *not* checked for validity! If you try to pass an already closed handle (or a random value) to this function, the result will be undefined.

4.3.3 ssize_t fs_read(file_t hnd, void * buffer, size_t cnt)

Reads up to *cnt* bytes from the file handle *hnd*, and stores them into *buffer*. The actual number of read bytes will be returned.

4.3.4 ssize_t fs_write(file_t hnd, const void * buffer, size_t cnt)

Writes up to *cnt* bytes from *buffer* and stores them to the file handle *hnd*. The actual number of written bytes will be returned.

4.3.5 off_t fs_seek(file_t hnd, off_t offset, int whence)

Works like the ANSI `fseek` function. The file pointed to by *hnd* will be seeked to *offset*, where *offset* is interpreted as specified in *whence*. The actual file position will be returned. The *whence* parameter is one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.

4.3.6 off_t fs_tell(file_t hnd)

Returns the file position of the handle *hnd*.

4.3.7 size_t fs_total(file_t hnd)

Returns the total byte size of the handle *hnd*, if applicable.

4.3.8 dirent_t * fs_readdir(file_t hnd)

Reads the next directory entry from the handle *hnd*. The file handle is assumed to have been opened using `fs_open(..., O_DIR)`. The returned pointer may be queried for values unless it is `NULL`, which signifies the end of the directory listing.

4.3.9 int fs_ioctl(file_t hnd, void * data, size_t size)

Perform a device/fs specific I/O Control message on the given file handle. This is entirely fs dependent. The only existing fs handler which supports this ioctl is fs_iso9660, which uses it to hint that the data/index cache should be flushed and the CD TOC should be rescanned.

4.3.10 int fs_rename(const char * fn1, const char * fn2)

Move/rename a file, if possible. The two files *must* be located on the same file system. Returns a negative value on failure.

4.3.11 int fs_unlink(const char * fn)

Delete a file, if possible. Returns a negative value on failure.

4.3.12 int fs_chdir(const char * fn)

Change the working directory for the current thread. Any relative path (i.e., a path not beginning with "/") will be treated as relative to this directory. This value will be inherited by any thread created by the thread which called this function. Returns a negative value on failure.

4.3.13 void * fs_mmap(file_t hnd)

Returns a pointer to a buffer containing the full data of the handle *hnd*. Depending on how the file was opened, the buffer may or may not be writable. This call may not succeed (limited memory, inability of the file system to perform the call, etc). In this case, the function will return `NULL`.

4.3.14 const char * fs_getwd()

Returns the working directory for the current thread.

4.3.15 int fs_handler_add(const char * prefix, vfs_handler * hnd)

Adds a new VFS handler to the VFS system. Each of the built-in file systems performs this call on startup. The file system will be placed on the file heirarchy at *prefix*, and will be accessed using the functions pointed to by *hnd*. Since this is considered an arcane and fairly internal piece of the VFS, please look at the source code for an existing handler if you wish to write a new one. It is fairly simple but beyond the scope of this document.

4.3.16 int fs_handler_remove(const vfs_handler * hnd)

Removes a previously added VFS handler. Each built-in file system performs this call on shutdown.

4.4 File System: fs_builtin

This file system module is somewhat deprecated, but it is still included and maintained because it has some interesting properties that could be useful to a programmer.

The basic concept of fs_builtin is that you pass it a table of "files", which really consist of pointers to data blocks and file sizes. It then manages file handles on these files and all of the normal VFS functions work on them.

This used to be used for building files into program images, but that function has been relocated to the much more able fs_romdisk. The interesting property of fs_builtin is that the pointers are arbitrary and point to blocks of data that can be changed on the fly. This could, for example, be used to implement a /proc like filesystem. In practice, it hasn't been used for anything like that.

fs_builtin mounts itself on **/bi** on the tree.

4.5 File System: fs_romdisk

fs_romdisk is the primary method by which one builds files into a program image in KOS (faster loading, easy for debugging, etc). A romdisk image is generated using the program **genromfs**; this image is converted to a binary object and linked with the program. The KOS initialization sets up fs_romdisk to point to this image. Any number of arbitrarily structured directories and files are also possible in fs_romdisk.

fs_romdisk mounts itself on **/rd** on the tree.

4.6 Memory Allocation System

KOS contains a full set of ANSI-standard memory allocation functions, implemented with the very popular **dlmalloc** library. This library is public domain and is very efficient for quickly allocating and freeing a large number of memory blocks, such as one might do from a C++ program.

The KOS kernel library also contains the GCC stub functions required for the use of the `new` and `delete` operators in C++. Thus you may write basic C++ programs without including any extra libraries to do so. In some cases, you may want to include libk++ as well, which is a very minimalistic stdc++ library for KOS. This avoids the necessity of building libstdc++ from GCC for about 99% of the cases.

Finally, if you feel comfortable with building your own KOS libraries and you are also comfortable with Makefiles, you may use the debug mallocator included with KOS. This is done by replacing "malloc.o" in **kernel/mm/Makefile** with **malloc_debug.o** and rebuilding. The debug mallocator never frees memory, so be aware of this if you use a lot of memory blocks. However, it does do buffer checking to look for overrun and underrun conditions, doubly freed blocks, leaked pointers, and other anomalies. If you enable the debug mallocator, you will also want to set `INIT_MALLOCSTATS` in your `KOS_INIT_FLAGS` line. This will tell the kernel to print out malloc stats when the program has exited since this is where all the extra checking is performed.

4.7 Network System

In addition to low-level drivers on applicable platforms, KOS supports a fully abstracted network datagram system. This system is capable of enumerating devices attached to the system and letting the program send and receive datagrams on those devices. Usually the network system is attached to something more high level, such as a TCP/IP stack. To enable networking, you must use the `INIT_NET` flag on the `KOS_INIT_FLAGS` line.

The pieces of the network system which are actually functional and well-specified are described below.

4.7.1 int net_input(netif_t * device, const uint8 * data, int len)

Low-level device drivers should call this function to register datagram input to the network system. The low-level driver's handle should be passed as *device*, while *data* and *len* are the data and length of the datagram.

4.7.2 net_input_func net_input_set_target(net_input_func t)

Sets the output port for datagram inputs. The function *t* will be called any time a datagram is available from a call to `net_input`. This function returns the address of the old function. The default target is a simple ICMP echo (ping) handler.

4.7.3 int net_reg_device(netif_t * device)

Registers a potentially available device driver with the network system. This should be called by any low-level driver which wants to make itself available. Note that the device need not be physically present on the system – the network system will scan for available drivers at init time.

4.7.4 int net_unreg_device(netif_t * device)

Unregisters a previously registered network device.

4.8 Thread System

KOS contains a fully functional thread system, including priority based scheduling, thread sync primitives with their own queues, zero-CPU-usage sleeping, and the choice between cooperative and pre-emptive thread models.

In the cooperative thread model, a thread will only ever be interrupted if it manually places itself on a wait queue. This can be accomplished by calling the `thd_sleep` function or waiting on an unavailable thread sync primitive. Use of cooperative threading mode with multiple threads has not really been tested very well, so use it at your own risk. We currently recommend sticking with a single thread if you want to use cooperative mode (this is what it was mainly designed for).

In pre-emptive mode, a timer interrupt is called HZ times per second (a constant in **arch.h**) to re-schedule. The current thread is placed back on the ready queue and another thread is pulled from it. This may be the same thread if no other threads are running. The effect is what you usually see in modern operating systems when running more than one program at once – all of the programs seem to be running simultaneously. The very big difference between those schedulers and the KOS pre-emptive scheduler is that KOS' scheduler does exactly what you tell it to do. There are no temporary priority boosts or decay, or anything of the like. This means that, like most real-time embedded kernels, if you place a thread at a high priority, *no other thread will execute until it is sleeping, has had its priority lowered, or quits.* I highlight the previous statement because thread priorities may be very useful for your application, but you must be careful not to cause a situation where your high-priority thread is always the one running. You also have to watch for "priority inversion", but discussion of these topics is beyond the scope of this document – go get a CS operating systems textbook. =)

Pre-emptive thread mode is the default (for `INIT_DEFAULTS`). You can cancel this out by using (`INIT_DEFAULTS & ~INIT_THD_PREEMPT`) in your `KOS_INIT_FLAGS` line, or by creating your own bitmask manually and leaving that part out.

On to the function reference...

4.8.1 kthread_t * thd_create(void (* routine)(void * param), void * param)

Creates a new thread and returns the *kthread_t* structure representing it. The thread will begin at *routine*, and its single and only parameter will be *param*. In case of failure, `NULL` will be returned.

4.8.2 int thd_destroy(kthread_t * thd)

Destroys a previously created thread. *This is not recommended.* Instead, if at all possible, a thread should quit on its own in a graceful manner. However, if you must kill a thread which hasn't died, then you can use this function. Any resources

the thread had allocated or locked (such as memory, thread sync primitives, etc) will not be freed.

4.8.3 void thd_exit()

Gracefully exit the current thread. This function will never return.

4.8.4 void thd_schedule()

Manually re-schedule. *This function is to be called only inside an interrupt/syscall context.* It assumes that interrupts are disabled and that we have entered the routine by saving the previous context into the appropriate *kthread_t*.

4.8.5 void thd_schedule_next(kthread_t * thd)

Same basic thing as `thd_schedule`, but you may specify a thread to be manually scheduled next. This can be useful if, for example, a device driver needs to pass control back to some code which was placed on a wait queue as quickly as possible.

4.8.6 void thd_pass()

The caller voluntarily gives up the rest of its timeslice. Any remaining timeslice will be given to the next scheduled thread. Note that the timeslice does *not* start over when this happens, so used incorrectly it could theoretically cause starvation issues.

4.8.7 void thd_sleep(int ms)

The caller will sleep the requested number of milliseconds. This function will return at a time equal to or greater than the requested delay. This is not guaranteed to be precise, and will certainly not be more precise than `HZ` allows. While the caller is sleeping, zero CPU time is spent (it is not scheduled at all).

4.8.8 int thd_set_prio(kthread_t * thd, prio_t prio)

Sets the priority of the given thread. The default priority is `PRIO_DEFAULT` and the maximum priority is `PRIO_MAX`. A larger value means lower priority.

4.8.9 kthread_t * thd_get_current()

Retrive the *kthread_t* for the caller.

4.8.10 const char * thd_get_label(kthread_t * thd)

Retrieve the "thread label" for the given thread. This is an arbitrary free-form string which identifies the thread for debugging purposes.

4.8.11 void thd_set_label(kthread_t * thd, const char * label)

Sets the "thread label" for the given thread.

4.8.12 const char * thd_get_pwd(kthread_t * thd)

Retrieve the current working directory for the given thread. This is used by the VFS for relative paths.

4.8.13 void thd_set_pwd(kthread_t * thd, const char * pwd)

Sets the current working directory for the given thread.

4.9 Sync Objects: Semaphore

Semaphores should be familiar to most computer science people and programmers. A semaphore is a counted sync primitive. The best analogy for this is a bank. In a bank, you have N tellers and M customers. Each customer can either go directly to an open window, or must take a ticket and wait until a teller is available.

In concise terms, each semaphore contains a count value. When a thread wants to use the limited resource represented by the semaphore, it must call `sem_wait` or `sem_wait_timed`. If the count is greater than zero, then the count is decreased and the thread is allowed to continue, all in an atomic and thread-safe manner. If the count is already zero, then the thread is placed on a queue and must sleep until another thread performs a `sem_signal`. That function checks to see if anyone is waiting and wakes them. In this way, control is very efficiently passed between waiting threads.

A very simple condition variable can be created by using a semaphore with an initial count of zero. A mutex can be created by using an initial count of one. For this reason, semaphores are mostly considered to be the fundamental thread sync primitive.

4.9.1 semaphore_t * sem_create(int value)

Create and return a new semaphore with an initial value of *value*.

4.9.2 void sem_destroy(semaphore_t * sem)

Destroy a previously created semaphore.

4.9.3 void sem_wait(semaphore_t * sem)

"Wait" on the given semaphore. The operation is described above.

4.9.4 int sem_wait_timed(semaphore_t * sem, int timeout)

"Wait" on the given semaphore for a maximum of *timeout* milliseconds. If the semaphore has not been signaled in that time, then return with an error code (negative value).

4.9.5 void sem_signal(semaphore_t * sem)

"Signal" on the given semaphore. Any waiting thread will be placed back on the scheduler's ready queue.

4.9.6 int sem_count(semaphore_t * sem)

Return the current count of the semaphore. Note that although this may be used to check if it's likely that the thread will be put to sleep, this is not an entirely reliable way to check this because of race condition issues.

4.10 Sync Objects: Condition Variable

Condition variables are like a pulsed one-count semaphore. N threads may wait on a condition variable. When a thread calls signal, one waiting thread is released. This is like a Win32 "Event" sync primitive.

Note however, that there is no function to release all waiting threads simultaneously, so this is not really very useful above a semaphore at the moment. Will be documented in more detail later.

4.10.1 condvar_t * cond_create()

4.10.2 void cond_destroy(condvar_t * cv)

4.10.3 void cond_wait(condvar_t * cv)

4.10.4 void cond_signal(condvar_t * cv)

4.11 Platform-Independent Images

Most people would argue that this has no business in the main kernel library, and they may very well be right. However, I wanted this to be a standard part of KOS on any platform it got ported to so that you're guaranteed some platform-independet, device-dependent representation of images. This way, any image loader library that is ported to KOS can convert to this data format and it can be used by anything in the kernel that understands it (drivers, etc).

KOS doesn't try to put any limitations on what format the image may take; instead it tries to encapsulate most of the common formats you'd want to use in writing a game or demo and then provides an "out" for platform-specific things that are just too strange to be used elsewhere. A good example of this (though one which isn't actually used currently) would be the PVR2's "twiddled" textures. Some somewhat strange things that it *does* cover are inverted axis textures (e.g., for GL) and YUV formats for MPEG (tm) .

The majority of the PII system is just format specifications so that pieces have a common language to talk about the images. This includes a structure (*kos_img_t*) that contains a pointer to the raw data, width and height in pixels, and a format descriptor. The first three pieces are pretty self-explanatory, so I won't waste any time on those. However, the format descriptor deserves some attention.

The format descriptor is composed of two parts: the platform independent part, and the platform dependent part. Each of these parts is 16 bits wide. The platform independent part is further divided into a format specifier and standard flags. The format specifier may be one of:

- `KOS_IMG_FMT_NONE` – Undefined / Unknown
- `KOS_IMG_FMT_RGB888` – Interleaved r/g/b/ bytes (24-bit)
- `KOS_IMG_FMT_ARGB8888` – Interleaved a/r/g/b bytes (32-bit)
- `KOS_IMG_FMT_RGB565` – r/g/b 5/6/5 (16-bit)
- `KOS_IMG_FMT_ARGB4444` – a/r/g/b 4/4/4/4 (16-bit)
- `KOS_IMG_FMT_ARGB1555` – a/r/g/b 1/5/5/5 (16-bit)
- `KOS_IMG_FMT_PAL4BPP` – Paletted (4-bits per pixel)
- `KOS_IMG_FMT_PAL8BPP` – Paletted (8-bits per pixel)

- KOS_IMG_FMT_YUV422 – y/u/v 4/2/2 (8-bit)
- KOS_IMG_FMT_BGR565 – b/g/r 5/6/5 (16-bit)
- KOS_IMG_FMT_RGBA8888 – r/g/b/a bytes (32-bit)

The attributes will be a bitmask with the following:

- KOS_IMG_INVERTED_X – The image is flipped along the X axis
- KOS_IMG_INVERTED_Y – The image is flipped along the Y axis (usually used with GL)

There is one function associated with the PII system currently:

- void kos_img_free(kos_img_t * img, int struct_also)

  Frees the image pointed to by *img*. If *struct_also* is non-zero, then the *kos_img_t* struct itself will also be freed. This function is to ensure future compatibility in code that calls libraries to allocate images and then needs to free them later itself.

Several macros are also available for tearing apart and putting together the format descriptor:

- KOS_IMG_FMT_I(x) – given a format descriptor x, return the platform independent part
- KOS_IMG_FMT_D(x) – given x, return the platform dependent part
- KOS_IMG_FMT(i,d) – given an independent part i and a dependent part d, return a full format descriptor

Note that KOS_IMG_FMT(KOS_IMG_FMT_I(x), KOS_IMG_FMT_D(x)) === x.

In the future, the PII system may also include functions to convert between image formats, but this hasn't been done yet.

Chapter 5

Hardware Abstraction Layer: Portable Parts

Underneath the platform independent pieces of KOS is a comprehensive hardware abstraction layer (HAL). This piece of code implements a standard API (plus a number of other things that provide ways to get directly to hardware functionality when necessary) which may be used by the platform independent pieces and also by user programs. The HAL includes the following pieces (and maybe more, depending on the needs of the platform):

- C runtime startup / shutdown, including `atexit`
- C++ runtime setup / shutdown, *not* including RTTI and exceptions
- A basic "bootstrap" debug I/O system (e.g., serial port)
- Setup of various KOS systems and hardware drivers from the init bitmasks
- Misc program control calls
- IRQ management / Context management (for threading)
- Realtime Clock (RTC) management
- Basic memory management (free pool for the system dlmalloc)
- MMU management if applicable/desired
- Timer management
- Cache coherency management (O/I-Cache flushing/writeback/invalidation, ITLB flushing, etc)
- Program image replacement (aka `exec`)
- Kernel/Interrupt mode-User mode transitioning (syscall)
- Low-level contention prevention (spinlocks)
- Platform-independent API for streaming sound and playing sound effects (optional)
- Platform-independent API for basic device input (optional, not yet specified)

- Platform-independent API for basic frame-buffer access (optional, not yet specified)
- Platform-specific file systems (e.g., for memory cards) *
- Platform-specific hardware drivers / APIs (3D, SPU, etc) *
- Misc platform-specific utility functions *

The three items above marked with a star (*) are part of the non-portable section of the HAL, and thus will be covered in the next chapter. However, they are listed here for completeness. The parts marked "not yet specified" have been potentially planned, but not yet designed or implemented, so they will not be specified here at all.

Note also that while we try to keep fairly generic pieces (like timer handling) either to a generic interface, or providing a generic interface, there are many cases where it's simply not possible. For example, every CPU has different sets of exceptions you can catch in software. Some are similar across CPUs, some aren't. This is a low-level enough thing, though, where making it portable would have probably caused more trouble than it would have saved. In the sections below where that is the case, this document will not expound further upon the subject in a platform-specific manner. You will need to refer to the headers and source code for the platform in question when writing something that uses these low-level interfaces.

Most users of KOS who are simply writing programs to use it will never need to look through most of this chapter of the manual. You will be much more interested in the next chapter.

But now, without further delay...

5.1 C/C++ Runtime

All of this stuff is completely under the covers for the vast majority of programs out there (and/or conforms to the ANSI C spec, such as `atexit`). If you need to know more about this part, then please visit your friendly local source code. Other sections which have no visible interface (or which have already been described, like `KOS_INIT_FLAGS`) will also not be discussed in this document.

5.2 Misc Program Control Calls

These pieces are part of the HAL but don't really fit anywhere else. They are just generic, and mainly related to startup/shutdown.

5.3 Bootstrap Debug Input/Output

Every architecture supported by KOS supports the concept of a debug I/O console. Sometimes this may not be connected to anything (especially in the really embedded ports like GBA) but this is where all debug and printf output goes by default, and this is where any pieces (such as DC's libconio) will grab their key input if configured to do so from the dbgio channel.

5.3.1 dbgio_printk_func dbgio_set_printk(dbgio_printk_func func)

Set a function to capture all debug output that comes through `dbgio_printf`, and return the old function. This is used by, e.g., dc-tool. The type for the function is:

```
typedef void (* dbgio_printk_func)(const char *);
```

5.3.2 void dbgio_set_parameters(int baud, int fifo)

Set serial parameters; this is not as platform independent as I want it to be, but it should be generic enough to be useful. The baud rate will be set to *baud* and if *fifo* is non-zero, the serial FIFO will be enabled.

5.3.3 void dbgio_write(int c)

Write one char to the debug port. You must call `dbgio_flush()` to actually ensure that the output was sent.

5.3.4 void dbgio_flush()

Flush all buffered bytes out of the port buffer, if any.

5.3.5 void dbgio_write_buffer(const uint8 * data, int len)

Send an entire buffer of data. This is just a shortcut for sending each byte of data with `dbgio_write`.

5.3.6 void dbgio_read_buffer(uint8 * data, int len)

Read an entire block of data. This is just a shortcut for reading each byte of data with `dbgio_read`. Note that this function will block until it has *len* bytes of data.

5.3.7 void dbgio_write_str(const char * str)

Send a C string (null-terminated).

5.3.8 int dbgio_read()

Read one char from the debug port (-1 if nothing to read).

5.3.9 void dbgio_disable()

Disable debug I/O globally; all debug output will be discarded and debug input will always return failure. This can be used as a quick way to disable all of your debug output before burning a final copy of your game/demo, etc.

5.3.10 void dbgio_enable()

Enable debug I/O globally (after calling `dbgio_disable`).

5.3.11 int dbgio_printf(const char * fmt, ...)

Works like `dbgio_write_str`, but provides formatting capabilities. There is a limit to the number of characters that can be in the resulting string, so don't pass arbitrarily long strings to this function.

5.4 IRQ/Context Management

IRQ management requires a few data structures and macros which are portable across platforms, and which will be described up front:

- *irq_context_t* – the basic structure describing an IRQ context; this is used both for saving the state of a program during IRQs and for switching contexts during threading
- *irq_t* – the type used to uniquely identify an interrupt/exception
- *irq_handler* – the function type for an interrupt/exception handler
- `REG_BYTE_CNT` – the number of bytes required to store the context on this platform
- `CONTEXT_PC(x)` – macro to retrieve the current "program counter" or "instruction pointer" from a context
- `CONTEXT_FP(x)` – macro to retrieve the current "frame pointer" from a context
- `CONTEXT_SP(x)` – macro to retrieve the current "stack pointer" from a context
- `TIMER_IRQ` – the interrupt/execption identifier (of type *irq_t*) to be used to hook the timer interrupt
- Various other values of type *irq_t* for interrupts/exceptions available on this platform

The rest is handled through the function interface.

5.4.1 int irq_inside_int()

Returns non-zero if the caller is executing inside an interrupt handler.

5.4.2 void irq_force_return()

This is kind of a weird one. This assumes that you have interrupts disabled, but are not actually running inside an interrupt handler, and you want to pretend that you were in fact inside an interrupt handler and return from that interrupt. This can be used to, for example, force a context switch without actually switching into "kernel" mode. We don't really use this anymore (deprecated in favor of the syscall interface).

5.4.3 int irq_set_handler(irq_t source, irq_handler hnd)

Set a handler for the given interrupt. The definition of *irq_handler* may platform dependent, but is generally:

```
typedef void (*irq_handler)(irq_t source, irq_context_t * context);
```

5.4.4 int irq_set_global_handler(irq_handler hnd)

Set a global interrupt handler. This routine will receive all interrupts and exceptions in favor of any individual handlers. Use with caution.

5.4.5 void irq_set_context(irq_context_t *regbank)

Change our "IRQ context". This should **never** be done with interrupts enabled. Basically whenever an interrupt occurs, the program context is saved into the current IRQ context. This is usually part of the *kthread_t* structure. When the interrupt handler returns, the IRQ context is re-established and the program continues. You can use this function to swap contexts in an interrupt handler (the thread system does this).

5.4.6 irq_context_t * irq_get_context()

Return a pointer to the current IRQ context. Note that it's not valid behavior to get this pointer and then use `CURRENT_xx()` from above on it because the values aren't valid until an interrupt occurs. Thus it is mainly used inside an interrupt handler to see who got interrupted.

5.4.7 void irq_create_context(irq_context_t * context, uint32 stack_pointer, uint32 routine, uint32 * args, int usermode)

Creates an *irq_context_t* from scratch. This is used mainly for creating new threads. The new values will be written to *context*, which must be pre-allocated. The initialo stack location will be *stack_pointer*, the initial PC will be *routine*, and *args* will be available as arguments to the routine (though the number of arguments is platform dependent). The *usermode* flag is a leftover from an earlier adaptation of KOS and may be removed later, but if non-zero it essentially tells us to create a context which will run with reduced privelges and in a user-capable address space.

5.4.8 int irq_disable()

Disable all IRQs and return the original IRQ state.

5.4.9 void irq_enable()

Enable all IRQs.

5.4.10 void irq_restore(int v)

Restore an old IRQ state from what was returned from `irq_disable`.

5.5 Realtime Clock (RTC) Management

Most systems contain a realtime clock of some sort which keeps track of the date and time in the outside world. This set of functions is designed to access that.

5.6 Basic Memory Management

On most systems, you simply have a flat chunk of RAM available which dlmalloc and other higher level pieces will manage. This very basic low-level memory management simply tracks that and provides a standard Unix(tm) style sbrk() function call. This effectively abstracts the address space.

Eventually we'll probably have functions for both getting and setting the time, but for now there's just one to retrieve it.

5.6.1 time_t rtc_unix_secs()

Returns the current day/time in *nix "seconds since 1970" format.

5.7 MMU Management

Most users of KOS will have no need of a memory management/mapping unit, but in case you do, this set of functions should provide for it. This is mostly abstracted as much as possible so as to not be platform-dependent, but there are limitations.

Nothing currently uses this and it probably doesn't even fully work, so this part isn't filled in right now. See **mmu.h** under the arch of your choice if you want more info.

5.8 Timer Management

Every platform should contain at least one available timer interrupt. These functions manage that timer interrupt and any others that may be available.

The DC part is relevant enough here that I'm going to go ahead and document it as well for this part.

5.8.1 Totally Portable Parts

void timer_ms_gettime(uint32 * secs, uint32 * msecs)

Returns the number of seconds and milliseconds since the program was started. This is for fine timing where the time of day doesn't really matter.

int timer_ints_enabled(int which)

Returns non-zero if the requested timer is enabled. The value of *which* is entirely platform dependent, but you can always use the constant `TIMER_ID` to refer to the primary timer that is used for threading and such.

void timer_disable_ints(int which)

Disables interrupts for the requested timer.

void timer_enable_ints(int which)

Enables interrupts for the requested timer.

void timer_spin_sleep(int ms)

Precise spin-loop sleep function. On the DC, this function does not interfere in any way with threading or other timer usage because it uses TMU1. On other platforms, this may be different.

void timer_primary_enable()

Enable the "primary" timer – start it counting and enable any associated interrupts. This is generally the timer used by pre-emptive threading.

void timer_primary_disable()

Disable the "primary" timer.

5.8.2 DC Parts

The SH-4 contains four usable timers, and we support three. These are `TMU0`, `TMU1`, and `TMU2`. `TMU0` is the "primary" timer, `TMU1` is used by `timer_spin_sleep`, and the other is available for programmer usage.

int timer_prime(int which, uint32 speed, int interrupts)

Pre-initialize a timer; setup the values but don't actually start it running. Timer *which* will be set to expire in *speed* milliseconds, and if *interrupts* is non-zero, interrupts will be enabled for it.

int timer_start(int which)

Starts the requested timer actually running.

int timer_stop(int which)

Stops the requested timer from running.

uint32 timer_count(int which)

Returns the count value of the requested timer.

int timer_clear(int which)

Clears the underflow bit of the requested timer and returns what its count value was.

5.9 Cache Coherency Management

Some processors have very good interal cache coherency, while some processors (notably embedded RISC chips) force you to be more careful about cache coherency. These functions are provided to ensure that you can maintain that coherency when doing things like loading new code into RAM.

5.9.1 void icache_flush_range(uint32 start, uint32 count)

Flush a range of instruction cache starting at the physical address *start* and going for *count* bytes. The value of *count* may be rounded up to the next closest possible size.

5.9.2 void dcache_inval_range(uint32 start, uint32 count)

Invalidate a range of data/operand cache starting at the given physical address and going for the given number of bytes. If the processor is using write-back cache, then the contents of the write-back cache will be *lost*.

5.9.3 void dcache_flush_range(uint32 start, uint32 count)

Flush a range of data/operand cache starting at the given physical address and going for the given number of bytes. If the processor is using write-back cache, then the contents of the cache will be written out to RAM before being invalidated.

5.10 Program Image Replacement

Kiosk/menu type programs (and others) often have the need to load a new image in place of the running image. This isn't the same as loading and executing a userland program, but instead replaces the running KOS image with an entirely new image (which may or may not be KOS based). This set of functions helps you with that task.

5.10.1 void arch_exec_at(const void * image, uint32 length, uint32 address)

Loads the program image located at *image* with a length of *length* bytes to the physical address *address*, and executes it. Note that *address* may overlap the existing program, and this is just fine. If necessary, this function will build a trampoline to copy and run the new image.

This function will never return.

5.10.2 void arch_exec(const void * image, uint32 length)

Same as `arch_exec_at`, but it loads the program at the default program location for the platform before running it.

This function will never return.

5.11 Syscall

There are times when you want to transition into a "kernel mode" state as if an interrupt had occured, but you don't want to have to wait for one. Most processors have a "trap" or "int" instruction which will manually cause this processor state. These functions wrap that functionality.

5.11.1 void syscall_set_return(irq_context_t * context, int value)

Sets the "return value" for *context* to *value*. If a user performed a syscall, then the target routine might call this function to set the value that the program will think it was returned.

This generally just pokes a value into one of the registers in the context, but it's platform dependent.

5.11.2 SET_RETURN(thread, value)

Macro that sets a return value for the named thread. This is just a wrapper for `syscall_set_return`.

5.11.3 SET_MY_RETURN(value)

Like `SET_RETURN`, but it sets the value on the "current" thread, which would generally be the one who invoked the syscall target.

5.11.4 RETURN(value

Sets the return value on the current thread and then causes a thread schedule.

5.11.5 SYSCALL(routine)

Use this macro to insert the platform-dependent code for invoking a syscall at the given address. *routine* is generally a function pointer.

5.12 Spinlocks

Spinlocks are a much lighter weight (though less efficient) way to prevent contention between threads. In KOS they can also allow you to prevent contention between interrupt handlers and normal user code while still providing performance by allowing critical sections that still allow interrupts to happen.

The data type for a spinlock is *spinlock_t*. This value can either be initialized statically with `SPINLOCK_INITIALIZER`, or it can be initialized programmatically with `spinlock_init`.

These are usually implemented as macros for speed.

5.12.1 void spinlock_init(spinlock_t * lock)

Initializes the given spinlock. It will be unlocked initially.

5.12.2 void spinlock_lock(spinlock_t * lock)

Attempts to lock the spinlock. If it is already locked then the function will not return until the caller owns the lock.

5.12.3 void spinlock_unlock(spinlock_t * lock)

Unlocks the given spinlock, if it's locked. Note that any thread can unlock any spinlock – there is no lock ownership.

5.12.4 int spinlock_is_locked(spinlock_t * lock)

Returns non-zero if the given lock is locked. This is helpful for, e.g., allocating memory inside an interrupt.

5.13 PI API for Sound

The output of streaming sound and sound effects is one of the basic pieces that you will want for any demo/game program you write, and basically every target platform of KOS will have some method of doing that. This set of functions helps you interact with that hardware without necessarily knowing how it works underneath. Note that initializing and using this API will usually prevent lower-level access (or will contend with it in bad ways), though this is usually reversable by shutting it down.

Note that although this API is designed to be platform independent eventually, it is currently only implemented on the DC and its headers are located with the DC headers. This may change in the future, though, and if you include **kos.h** then you shouldn't have troubles.

5.13.1 int snd_mem_init(uint32 reserve)

Setup the sound allocation system. Generally allocates a buffer for sound usage and reserves *reserve* bytes at the front for system overhead. The value for *reserve* should never be zero because this would allow for valid sound mem handles to indicate an error condition in `snd_mem_malloc`.

5.13.2 void snd_mem_shutdown()

Shut down the sound allocation system, and free any structures related to it.

5.13.3 uint32 snd_mem_malloc(size_t size)

Allocate a chunk of sound RAM and return a handle to it. This will generally be an offset into a pre-allocated sound buffer (or in the case of the DC, the SPU RAM). A zero return value signifies failure.

5.13.4 void snd_mem_free(uint32 addr)

Frees a previously allocated chunk of sound RAM.

5.13.5 uint32 snd_mem_available()

Returns the largest chunk of sound RAM available.

5.13.6 int snd_init()

Initialize the overall sound system. This is generally preferred to calling `snd_mem_init` directly.

5.13.7 void snd_shutdown()

Shut down the overall sound system. This is generally preferred to calling `snd_mem_shutdown` directly.

5.13.8 int snd_sfx_load(const char * fn)

Load the sound sample located on the VFS at *fn*. Currently this function only knows about RIFF WAV files, but this may change later. Returns a handle to the loaded effect.

5.13.9 void snd_sfx_unload(int idx)

Unload a single loaded sample. Call with the handle you got back above.

5.13.10 void snd_sfx_unload_all()

Unload all samples loaded with `snd_sfx_load`. This function is deprecated in favor of freeing individual samples.

5.13.11 void snd_sfx_play(int idx, int vol, int pan)

Play the given sound effect at volume *vol* (0-255) and with panning *pan* (0 - 255, 128 is center).

5.13.12 int snd_stream_init(void* (* callback)(int, int *))

Setup the sound system for streaming sound data. *callback* will be called whenever the system needs more data for output (see below for more info about the callback).

This function calls `snd_init` implicitly, so there is no need to call it beforehand.

5.13.13 void snd_stream_shutdown()

Shuts down the sound streaming system.

This function calls `snd_shutdown` implicitly, so there is no need to call it afterwards.

5.13.14 void snd_stream_set_callback(void *(*func)(int req, int *ret))

Sets the callback function for the streaming mechanism. When we are running low on data in the circular output buffers, *func* will be called. It will receive a request for the number of samples we'd like to get. The function should return a pointer to where those samples are located and return via *ret* how many we actually got back. If the return value is NULL, then the stream is considered finished.

5.13.15 void snd_stream_queue_enable()

Enable sound stream queueing. This allows you to very finely tune when the playback will start in relation to other events in your program (for example, if you need music exactly timed to the graphics).

5.13.16 void snd_stream_queue_disable()

Disable queueing.

5.13.17 void snd_stream_queue_go()

If queueing is enabled, calling this function will make it actually start playing instantly.

5.13.18 void snd_stream_start(uint32 freq, int st)

Start stream playback with the given frequency. If *st* is non-zero, then we are playing a stereo stream. If queueing is enabled, then the stream will not actually start but will just fill its buffers and get ready.

5.13.19 void snd_stream_stop()

Stops any playing stream.

5.13.20 int snd_stream_poll()

Polls the stream driver to see if we need more data. This may not be required on all platforms; on the DC it is currently required.

If the return value is -1, then there was an internal error (including a `NULL` callback. If -3, then the stream has ended. If zero, then everything is hunky dory.

5.13.21 void snd_stream_volume(int vol)

Set the volume of the streaming output (0-255).

Chapter 6

Hardware Abstraction Layer: DC Parts; OUT OF DATE

UNCONVERTED FROM LIBDREAM

6.1 Basic Video (video.c)

The basic video subsystem of libdream is designed to facilitate frame buffer setup and access. It does not handle 3D acceleration, that is handled in the "TA" module. The basic usage of the video system is to call vid_init with the desired parameters, and then use one of the "vram" pointers to access video memory.

The following variables and functions are available in basic video:

6.1.1 uint32 *vram_l

Pointer to video memory (0xa5000000) as a 32-bit unsigned integer array. Use this to access video memory when in RGB888 mode, or when copying large regions (not recommended =).

6.1.2 uint32 *vram_s

Similar to vram_l, but accesses as a 16-bit unsigned integer array. Use this to access video memory when in RGB555 or RGB565 mode.

6.1.3 int vram_config

Stores the first parameter to vid_init, which is the pixel format.

6.1.4 int vram_size

Stores the total size (in pixels) of one page of video frame buffer.

6.1.5 int vid_cable_type

Stores the cable_type parameter to the video init function below.

6.1.6 int vid_check_cable()

Checks and returns the attached video cable type; the three constants matching the return values are CT_VGA, CT_RGB, and CT_COMPOSITE.

6.1.7 int vid_init(int cable_type, int disp_mode, int pixel_mode)

Does full frame buffer initialization with the requested cable type, display mode, and pixel mode. You should pass the return value from vid_check_cable() as the first parameter. dc_setup() does this for you. disp_mode constants are DM_320x240, DM_640x480 and DM_800x608. pixel_mode constants are PM_RGB555, PM_RGB565, and PM_RGB888.

6.1.8 void vid_set_start(uint32 start)

Set the "start address" register, which governs where in the frame buffer the output picture comes from. This can be used for "double buffering", but it will most commonly be used during 3D acceleration.

6.1.9 void vid_border_color(int r, int g, int b)

Set the border color. The border is the area outside the standard viewing area. On NTSC this is mostly above and below the picture. I'm not sure what it is on PAL. Generally unless you're doing some odd debugging you'll want to set this to black (vid_init does this for you).

6.1.10 void vid_clear(int r, int g, int b)

Clears the first page of frame buffer based on the current video mode, with the given color. This is most useful when using frame buffer access, not 3D access.

6.1.11 void vid_empty()

Clear the entirety of video memory using zeros. This is done using longwords so it's fairly quick. Once again, mainly used with 3D setup.

6.1.12 void vid_waitvbl()

Wait for a vertical blank period. Vertical blank is the period between the time that the scan beam reaches the bottom of the screen and the time that it reaches the top and starts drawing again. This is relevant because this is the best time to draw to the frame buffer without causing "tearing" or other artifacts. It's also generally when you want to switch start addresses.

6.2 BIOS Fonts (biosfont.c)

BIOS fonts are the ones you see in the boot manager on the Dreamcast. These are stored in ROM and so are available to any program. You will probably recognize them immidiately since they are used all over the place in official productions. The BIOS font contains European Latin-1 characters (which we support) and Kanji (which we don't support yet but will eventually). The Latin-1 characters are bit masks of size 12x24, so each character uses 36 bytes. I suspect that the Kanji characters are 24x24, but I haven't tested this yet. These functions are frame-buffer agnostic except that they expect a 16-bit pixel size.

The following functions are available:

6.2.1 void* bfont_find_char(int ch)

Returns the address in ROM of the given character, after being mapped to the BIOS font.

6.2.2 void bfont_draw(uint16 *buffer, int bufwidth, int c)

Draws Latin-1 character 'c' at the given location in 'buffer', and assumes that 'buffer' is 'bufwidth' pixels wide. For example, to draw an 'a' at 20,20 in a 640x480 framebuffer, you'd use bfont_draw(vram_s+20*640+20, 640, 'a').

6.2.3 void bfont_draw_str(uint16 *buffer, int bufwidth, char *str)

Exactly like bfont_draw, but it takes a string and draws each character in turn.

6.3 PC Fonts (font.c)

The PC font system handles bitmapped fonts that are 8 pixels wide, and any number of pixels tall. The module is being deprecated in favor of the BIOS font module, so I won't describe it here. If you want more information, please reference font.c itself.

6.4 Sound Processor Unit (spu.c)

The sound processor unit (as mentioned in the README) is a Yamaha(tm) AICA sound system. To use the processor you will need to write a seperate program that runs on the ARM7 RISC core and uses the AICA's own registers. This isn't covered in this document (or anywhere, to my knowledge). For some decent examples, though, take a look at "s3mplay" on the Cryptic Allusion DCDev site (see README).

The following defines and functions are available to assist in using the sound processor:

6.4.1 SMP_BASE

All samples loaded to the AICA should proceed from this location relative to sound RAM (which maps to 0xa0810000 in the SH-4). This is mainly used in the S3M player but it's a good guideline to follow in general since it gives you 64k of space for the sound program (which is generally plenty).

6.4.2 void snd_ram_write_wait()

The AICA's RAM is attached to the chip itself rather than the SH-4, and so access proceeds through an ASIC. You must call this function every 8 long-words of written sound memory so that the ASIC's FIFO can catch up. If you don't, the data won't be written accurately.

6.4.3 void snd_load_arm(void *src, int size)

Loads an ARM7 program and starts it executing. The program will be loaded at offset 0, so it needs to begin with reset/exception vectors.

6.4.4 void snd_stop_arm()

Stops execution in the ARM7, and disables all AICA synthesizer channels. This insures that whatever was going on in the SPU is stopped completely.

6.4.5 void snd_init()

Initialize the SPU: disable the ARM7 and clear sound memory.

6.4.6 void snd_load(void *src, int dest, int len)

Load miscellaneous data into the SPU's RAM. 'src' is where to load from, and 'dest' is relative to the SPU based (so you could pass, e.g., SMP_BASE here). 'len' is in bytes but will be rounded up to long-words.

6.5 CD-Rom Access (cdfs.c)

Libdream provides the capability to use CD and CDR discs in the GD-Rom drive using this module.

Note that this file has been specifically crippled (or rather, we just never wrote it in) so that it can't access Sega's GD discs. This means that you can't access the data on any commercial game. There are really only a few legitimate reasons for doing this so we've disabled the feature to avoid coming under fire for assisting with copyright infringement. If you really want to know how, I'm sure you can figure it out =).

The following functions are available:

6.5.1 uint32 iso_open(const char *path, int oflag)

Open a file on the CD, using absolute path "path", with open flags "oflag". Note that in the current system, "path" must use forward slashes for path seperators (but can mix upper and lower case freely with no troubles), and "oflag" must be O_RDONLY or (O_RDONLY — O_DIR). A file descriptor will be returned, or a zero on error.

6.5.2 void iso_close(uint32 fd)

Close the file referenced by the given file descriptor.

6.5.3 int iso_read(uint32 fd, void *buf, int count)

Read "nbyte" bytes from the file referenced by "fd", into buffer "buf". Note that this function will always read the surrounding 2048 byte sector before extracting the parts you want, so you should never read less than 2048 bytes unless that's all you want. Reading more than 2048 does work. The number of bytes read will be returned.

6.5.4 long iso_lseek(uint32 fd, long offset, int whence)

Seek in file "fd" by "offset" bytes, relative to "whence". "whence" is one of the standard STDIO constants: SEEK_SET, SEEK_CUR, SEEK_END. "offset" may be positive or negative depending on the usage. The new file location will be returned.

6.5.5 long iso_tell(uint32 fd)

Returns the current file pointer within "fd".

6.5.6 dirent_t *iso_readdir(uint32 dirfd)

Read the next entry (if any) from the opened directory. Returns a pointer to a dirent_t on success (see fs_iso9660.h for more info on dirent_t) or `NULL` if nothing is left.

6.5.7 int cdrom_init() int iso_init()

Initialize the GD-Rom drive for reading CD/CDR media, and initialize the file system driver.

6.6 Timer Counters (timer.c)

This module supports the SH-4's internal timer perhipherals. Support is provided for TMU0 through TMU2. WDT (watchdog) is defined but not supported yet. TMU0 through TMU2 may all be used independently and count at different rates.

The following defines and functions are available:

6.6.1 TMU0, TMU1, TMU2, WDT

These are constants used to identify which timer you wish to operate on.

6.6.2 int timer_prime(int which, uint32 speed)

Primes a timer, but does not start it. "which" is one of the timer constants, and "speed" is a times per second rate which the counter will bottom out. So if you set speed to "1", then the timer will hit bottom after one second, and start counting again. Returns 0 for success.

6.6.3 int timer_start(int which)

Starts the requested timer counting (after priming it).

6.6.4 int timer_stop(int which)

Stops the requested timer.

6.6.5 uint32 timer_count(int which)

Returns the current timer count. The only way you can really make use of this externally is to get the timer count after priming but before starting, and scale the real-time results.

6.6.6 int timer_clear(int which)

Clears the timer underflow bit and returns what its value was. Underflow is set when the timer counts down. So for example, you could start a timer on a 1HZ cycle and poll this function until it returns true. At that point you'd have waited for a second, and the timer is already counting down again.

6.6.7 void timer_sleep(int ms)

Uses TMU0 to sleep for the given number of milliseconds.

6.6.8 int timer_init()

Setup timers (enable and stop all).

6.7 Maple Access (maple.c)

Libdream 0.7 includes Jordan DeLong's rewritten maple access code. This is a lot more modular and it is setup for future expansion with queueing multiple frames and DMA completion interrupts. For now it basically does the same as Marcus' old maple routines but with cleaner code.

In general using the maple bus consists of finding your peripheral (using DEVINFO queries), and storing this address; when you want to use the peripheral, you send it a condition query message and get a frame back describing the state of the peripheral. Most of the exported functions in maple.c won't be useful to mere mortals =) but that's a good thing since there are specific support modules for each of the major peripherals we have had access to.

The following functions are available:

6.7.1 void maple_init(int quiet)

Initialize the maple bus; if "quiet" is non-zero, then the bus scan will not produce any output.

6.7.2 void maple_shutdown()

Shut down all maple bus operations.

6.7.3 uint8 maple_create_addr(uint8 port, uint8 unit)

Create a maple address for the given port and unit.

6.7.4 int maple_docmd_block(...)

Parameters omitted for topic brevity: int8 cmd, uint8 addr, uint8 datalen, void *data, maple_frame_t retframe. This is the main "work horse" of the maple system. "cmd" should be one of the maple command constants in maple.h; "addr" should be created with maple_create_addr (or one of the maple_*_addr functions below); "datalen" is the length of the extra data (beyond what's in the frame header), "data" is a pointer to extra data (if any) that goes after the frame header; and "retframe" is a maple_frame_t that you should pass in to be filled in with return data. Zero is returned on success, and -1 returned on error. For some examples of using docmd_block directly, please check one of the maple peripheral modules.

6.7.5 int maple_rescan_bus(int quiet)

Rescans the maple bus. This will be neccessary if the user swaps out any controllers or memory cards. It also determines what is where and stores that info for later usage. If "quiet" is non-zero, it produces no output.

6.7.6 uint8 maple_device_addr(int code)

Pass a maple function code, and it returns the address of the first one that matches it.

6.7.7 uint8 maple_*_addr()

These include controller, mouse, kb, lcd, and vmu currently. Each one searches the maple bus to find the first matching type of peripheral and returns an address.

6.8 Maple Peripheral Support Modules

Support modules are included for standard controllers, keyboards, VMUs, and mice. Most peripherals fit into these molds. Eventually we'll probably add support for more things like the purupuru pack (force feedback) but we don't have one yet, so we can't. =) Notable among this list is the mouse since it just came out. Wow your friends by writing software that uses it before Sega gets a chance! =)

Since these are mostly the same (except for names and structure values) I won't go over them in detail. Each module generally contains a poll function that checks the state of the peripheral and fills in a device-specific structure. See the header files for the specific structure information. I'll list out the poll functions here for convienence though.

6.8.1 int cont_get_cond(uint8 addr, cont_cond_t *cond)

Check controller status. Returns which buttons are pressed and the state of the various analog controls.

6.8.2 int kbd_get_cond(uint8 addr, kbd_cond_t *cond)

Check keyboard status. Returns up to six keys being pressed at once. There are other support functions for the keyboard that you should look up in keyboard.h if you want to use it seriously. These do queueing and buffering for you. If you want this functionality, you should use kbd_poll(uint8 addr) and then kbd_get_key() to get key presses.

6.8.3 int vmu_draw_lcd(uint8 addr, void *bitmap)

Draws the given bitmap to the LCD screen. Generally these are on VMUs (which is why it's part of vmu.c) but it's not required. The bitmap should be a 48x32 bit array. The picture will show up right side up on the VMU itself, so when it's inserted in a controller you'll need to flip it in each direction.

6.8.4 int vmu_block_read(uint8 addr, uint16 blocknum, uint8 *buffer)

Read the requested block of the VMU's flash ram and put it in "buffer".

6.8.5 int vmu_block_write(uint8 addr, uint16 blocknum, uint8 *buffer)

Take what's in "buffer" and write it to the requested block of the VMU's flash ram.

6.8.6 int mouse_get_cond(uint8 addr, mouse_cond_t *cond)

Gets the condition of the mouse peripheral specified. Returns button states and delta x, y, and z (roller).

6.9 Tile Accelerator (ta.c)

The Tile Accelerator (3D acceleration) really deserves its own book, but for completeness (and my hands are getting tired =) I'm just going to go over the basics of setting it up and the functions you use to do so. For more specific information,

look around on the web for various documents describing the TA, and look in the examples. Hopefully this section can be more fleshed out in future versions.

The TA is exactly what it says: the screen in the PVR 3D chip is broken up into 32x32 pixel tiles. So in 640x480, you'd really have a 20x15 tile field, not a 640x480 pixel field. The PVR's 3D magic happens by taking each of these tiles along with a "display list" describing what is to be displayed on the screen, and doing internal z-buffering. This means that each polygon is drawn only once, so even though there is not a standard z-buffer present, the end result looks like there is one. Opaque polygons, opaque volume modifiers (fog, etc), translucent polygons, translucent modifiers, and punch-through polygons (which can "cut" pieces of other polygons out, I think) must be sent to the TA, in that order. Each list is rendered in that order as well, for each tile, and so the more lists you send, the slower the rendering process gets. Opaque polygons are the fastest obviously, followed by punch-throughs, translucent polygons, and then the volume modifiers.

Because of the tile system, there is no user clipping neccessary: the TA works backwards by intersecting polygons and volumes with each tile before rendering. The end result of all of this is that all you have to do as a user is cull out the completely useless polygons (if you feel like it), arrange things in polygon "strips" as much as possible, and then throw the various lists to the TA. Then sit back and wait for it to do its work.

The PVR chip is not magic: it is powerful and can accelerate the drawing process to an amazing degree, but it still draws in terms of screen coordinates. So it is really a fancy 2D accelerator with perspective correction support for textures, and z-buffering.

Coordinates in the PVR start at 0,0 (all coordinates are floating point numbers) and go to 640,480, in the normal mode. Any coordinates outside this will work but will be clipped. Z coordinates start at 0 and move out of the screen towards the viewer. As far as I can tell, in normal mode, it wants Z and not 1/Z (W). I may be wrong of course. I'm no 3D hardware expert.

All that being said, the basic operation goes something like this:

1. Setup the TA (ta_init); initialize the background plane structure
2. Load any textures you may want to use
3. For each frame:

- Call ta_begin_render to initialize the rendering process.
- Construct and send one or more polygon headers for opaque polygons, each followed by zero or more verteces; each vertex strip must end with an "end of list" marker.
- Call ta_commit_eol to finish the opaque list.
- Construct and send one or more polygon headers for translucent polygons (same process as above).
- Call ta_commit_eol to finish the translucent list.
- Call ta_finish_frame to finish the rendering process and wait for a vertical blank to flip pages.

Here are the structures and functions needed to do these things:

6.9.1 struct pv_str ta_page_values[2]

Holds all the internal rendering data for the page flipper and renderer. This is useful mainly if you want to do something like take a screen shot (you can find the current frame buffer).

6.9.2 bkg_poly ta_bkg

The background plane polygon. The background plane is currently automatically a 640x480, three-point opaque polygon. I'm not even sure if you can change this. For the values to load into this, take a look at one of the 3D example programs. If you want to do color shifting you can change this on the fly.

6.9.3 poly_hdr_t

A polygon header; this is always four flag long-words and four dummy words. The four dummy words are actually used with different types of shading and volume modifiers, but these are not supported yet in libdream. You should fill this structure directly (if you know what you're doing) or use ta_build_poly_hdr.

6.9.4 vertex_oc_t

Represents a single opaque/colored vertex with floating point coordinates and ARGB values. Actually it works fine for translucent polygons also but the naming convention stuck.

6.9.5 vertex_ot_t

Represents a single opaque/textured vertex with floating point coordinates and ARGB values. Actually it works fine for translucent polygons also.

6.9.6 int ta_curpage

The current working page (out of ta_page_values above).

6.9.7 void ta_init()

Initializes the TA and prepares for page flipped 3D.

6.9.8 void ta_send_queue(void *sql, int size)

Sends one (or two) store queue(s) full of data to the TA.

6.9.9 void ta_begin_render()

Call before you start drawing a frame.

6.9.10 void ta_commit_poly_hdr(poly_hdr_t *polyhdr)

Sends one polygon header to the TA. This needs to be done when you want to change drawing modes; e.g., opaque color, opaque textured, translucent color, translucent textured.

6.9.11 void ta_commit_vertex(void *vertex, int size)

Sends one vertex to the TA; this can be a vertex_oc_t or vertex_ot_t. Pass along the result of sizeof() on the vertex.

6.9.12 void ta_commit_eol()

Sends the "end of list" marker to the TA. This ought to be used after all opaque polygons are sent, and again after all translucent polygons are sent.

6.9.13 void ta_finish_frame()

Call after you've finished sending all data. This completes the rendering process in the alternate screen buffer and then waits for a vertical blank to switch to the new page.

6.9.14 void ta_build_poly_hdr(poly_hdr_t *target, ...)

Parameters omitted for brevity: int translucent, int textureformat, int tw, int th, uint32 textureaddr, int filtering. This builds a polygon header for you so you don't have to diddle with bitfields. Translucent should be one of TA_OPAQUE or TA_TRANSLUCENT. Textureformat needs to be one of the texture format constants or TA_NO_TEXTURE. This includes whether it's twiddled or not (for info on twiddled textures, look for the PVR E3 presentation online). The rest of the parameters are only relevant if textureformat is not TA_NO_TEXTURE. tw and th are the texture width and height, and must be powers of two between 8 and 1024. Textureaddr is the address within the PVR RAM that you loaded the texture, and it must be aligned on an 8-byte boundary. Filtering should be TA_NO_FILTER or TA_BILINEAR_FILTER. Note that bi-linear filtering is a fairly expensive operation unless you store your textures in the PVR RAM in twiddled format, in which case it's free.

6.9.15 void ta_load_texture(uint32 dest, void *src, int size)

Loads a texture into PVR ram at the given offset. "size" must be a multiple of 4 and will be rounded up if it's not already. A seperate function is required because the PVR requires you to send all texture data to 0xa4000000, not 0xa5000000. This must also be done after ta_init.

6.9.16 void *ta_texture_map(uint32 loc)

Maps a given PVR offset to a texture space. You should use this if you want to write directly into texture ram. Once again, it must be done after ta_init.

Chapter 7

DC Add-On Libraries

Chapter 8

About this document

This document was originally written in a stock LyX 1.1.6fix4 distribution, and has since been converted to raw LaTeX2e.