Wii

Interestingly-named home Nintendo console that everyone's aunt bought for Christmas 2007

Getting Started

Wii homebrew is compiled using devkitpro, specifically devkitPPC. It compiles to a `boot.dol` and a `boot.elf` file, we're more interested in the first.

Repository Setup

Use the wii-template template repository to create your git repo. You can either click the "Use this template" button, or select "alex/wii-template" from the "Template" dropdown when making a repository. The template includes:

- The example hello world application from devkitpro.
- The devkitpro example Makefile, modified to always output a file named `boot.dol` for the Homebrew Channel.
- A Jenkinsfile that will build your code using the Makefile in the devkitppc Docker container.
- The icon.png and meta.xml files needed for the Homebrew Channel.

Local Compilation

Most sources online recommend you install devkitpro locally, so that's definitely an option. Nobody really mentions the Docker container, but if you want you can use that locally too, using the following command from the project root directory:

```
docker run -it --rm -v $(pwd):/src devkitpro/devkitppc bash -c 'cd /src; make'
```

This will output your `boot.dol` and `boot.elf` to the same folder.

Emulating Your Code

You can quite easily run your code in the Dolphin emulator, just go to "Open" and select your `boot.dol` file.

Homebrew Channel Installation

To get the code running on an actual Wii, you need to install it to the Homebrew Channel. Applications are installed to the `apps` folder in the root of your SD card, each application has its own folder. Just put your `boot.dol`, `meta.xml` and `icon.png` in a new folder in the apps folder, and you should be good to go.

The `meta.xml` describes what appears in the menu for your application in the Homebrew Channel. Its fields are documented on wiibrew, here.

The `icon.png` is the application's icon in the Homebrew Channel. Wiibrew says it should be 128x48 pixels (see here), and provides a seperate page with a lot of templates.

How do I ...?

Check controller input?

First you need to called `WPAD_Init()`! It is defined in `wiiuse/wpad.h`

Wii Remote

WPAD stores information on Wii Remote controllers in `WPADData` structs, which you can get a pointer to with the `WPAD_Data(chan)` function. The `chan` parameter is presumably the player number? But zero-indexed. The structure is defined as follows:

```
typedef struct _wpad_data
{
	s16 err;

	u32 data_present;
	u8 battery_level;

	u32 btns_h;
	u32 btns_l;
	u32 btns_d;
	u32 btns_u;

	struct ir_t ir;
	struct vec3w_t accel;
	struct orient_t orient;
	struct gforce_t gforce;
	struct expansion_t exp;
} WPADData;
```

Rather then read from the struct directly, I think the WPAD developers would prefer you use helper functions for pulling out individual fields, such as button states. These functions are:

- `u8 WPAD_BatteryLevel(int chan)`
- `u32 WPAD_ButtonsUp(int chan)`
- `u32 WPAD_ButtonsDown(int chan)`
- `u32 WPAD_ButtonsHeld(int chan)`
- `void WPAD_IR(int chan, struct ir_t *ir)`

- `void WPAD_Orientation(int chan, struct orient_t *orient)`
- `void WPAD_GForce(int chan, struct gforce_t *gforce)`
- `void WPAD_Accel(int chan, struct vec3w_t *accel)`
- `void WPAD_Expansion(int chan, struct expansion_t *exp)`

Buttons

Buttons can either be up, down, or held. Presumably buttons transition from being down when initially pressed, to held, to up when released. The `u32` values returned from the button functions contain the state of each button on the controller, and access masks are provided to query individual values. For example, to test if the HOME button is pressed:

```
#include <wiiuse/wpad.h>


bool home_button_pressed(int chan)
{
  return (WPAD_ButtonsDown(chan) & WPAD_BUTTON_HOME > 0);
}
```

The full list of available buttons includes everything on the Wii Remote, the two additional buttons on the Nunchuck, all of the Classic Controller buttons, and all of the Guitar Hero buttons. They are all defined here.

References

- wpad.h and wpad.c.

Do networking?

Work-in-progress page, just wanted to capture some trickiness Jack came across.

- In `net_bind()`, `IPPROTO_TCP` as the third parameter gives the cryptic error -81, which is not really defined anywhere. This really needs to be `IPPROTO_IP` instead.
- Get your port number right!

Example

```
int32_t listen_tcp(uint16_t port)
{
    // Create server socket.
    int32_t sock = net_socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    if (sock == INVALID_SOCKET)
        return -1;
    struct sockaddr_in server;
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = INADDR_ANY;

    // Bind server socket.
    int32_t ret = net_bind(sock, (struct sockaddr *)&server, sizeof(server));
    if (ret < 0)
        return ret;

    // Listen on server socket.
    ret = net_listen(sock, 10);
    if (ret < 0)
        return ret;

    // Create client socket.
    struct sockaddr_in client;
    memset(&server, 0, sizeof(server));
```

```
    uint32_t client_len = sizeof(client);

    // Create receive buffer.
    char buffer[1024];

    while (true)
    {
        printf("\x1b[%d;%dH", 4, 0);
        printf("Waiting for data...");
        int32_t csock = net_accept(sock, (struct sockaddr *)&client, &client_len);
        if (csock < 0)
            return ret;

        // Reset buffer and receive contents.
        memset(buffer, 0, 1024);
        int32_t bytes = net_recv(csock, buffer, 1024, 0);
        printf("\x1b[%d;%dH", 5, 0);
        printf("Received %d bytes.", bytes);

        // Print buffer to screen.
        printf("\x1b[%d;%dH", 6, 0);
        printf(buffer);

        net_close(csock);
    }

    return 1;
}
```

Draw to the screen?

Setup

There's quite a bit going on in the setup, but it thankfully is mostly boilerplate.

The functions you want, in order, are:

- `VIDEO_Init()`: This needs to be called before any other VIDEO functions. Documentation says it should be done "in the early stages of your `main()`, but we've had success in moving it to a different function that is called from main.
- `VIDEO_GetPreferredMode()`: This returns you your `GXRModeObj` struct pointer (a "rendermode object") defining the screen layout and mode. What you get from this depends on the console settings, and if it's PAL or NTSC, things like that. The parameter also takes a `GRXModeObj` and seems to overwrite it if present? Not sure why you'd want that, just pass `NULL`.
- `SYS_AllocateFramebuffer()`: Allocate cacheline aligned memory for the external framebuffer based on the passed rendermode object. Returns a 32-byte-aligned pointer to the framebuffer's start address. This allocates the memory for our framebuffer, but we cannot use it yet.
- `MEM_K0_TO_K1()`: This is a macro to cast a cached virtual address to an uncached virtual address. In this case, we use this to convert the address of the allocated framebuffer to one we can use to write to it.
- `VIDEO_Configure()`: Configures the "VI" (video interface?) with the given rendermode object. Internally, this is setting a lot of video registers with our rendermode settings
- `VIDEO_SetNextFramebuffer()`: This sets some video register to point to our allocated framebuffer memory. In effect, we are telling the video hardware where the framebuffer is.
- `VIDEO_SetBlack()`: TRUE to black out the screen, FALSE not to.
- `VIDEO_Flush()`: When we make changes to the video hardware registers in the above functions, we do not *actually* make those changes to the hardware until we flush them with this function. This in effect commits our changes.
- `VIDEO_WaitVSync()`: Waits for the next vertical retrace.

Putting all that together, a simple (minimal) video setup would look like:

```
#include <gccore.h>
```

```
static void* xfb = nullptr;
static GXRModeObj* rmode = nullptr;

void video_initialise()
{
	// Initialise the video system
	VIDEO_Init();

	// Obtain the preferred video mode from the system
	// This will correspond to the settings in the Wii menu
	rmode = VIDEO_GetPreferredMode(NULL);

	// Allocate memory for the display in the uncached region
	xfb = MEM_K0_TO_K1(SYS_AllocateFramebuffer(rmode));

	// Set up the video registers with the chosen mode
	VIDEO_Configure(rmode);

	// Tell the video hardware where our display memory is
	VIDEO_SetNextFramebuffer(xfb);

	// Make the display visible
	VIDEO_SetBlack(FALSE);

	// Flush the video register changes to the hardware
	VIDEO_Flush();

	// Wait for Video setup to complete
	VIDEO_WaitVSync();
	if (rmode->viTVMode & VI_NON_INTERLACE)
		VIDEO_WaitVSync();
}
```

Text output

Getting console output text out to the screen is quite easy. We need an additional setup function, `CON_Init()` (sometimes seen with the old name `console_init()`). This takes as input:

- The framebuffer (the same parameter we gave to `VIDEO_SetNextFramebuffer()`, `xfb` in examples).
- The x, then y-coordinate to start output from. This allows for a border around the edge of the screen.
- The full width, then height, of the screen. Get these from the rendermode object (`rmode` in the example).
- The *stride*, which is the size of one line of the framebuffer in bytes. This is the width, multiplied by the constant `VI_DISPLAY_PIX_SZ`, which is the size of one pixel in bytes.

And then that's it! You can then use `printf()` (from `<stdio.h>`) normally and it prints to the screen.

The console also accepts VT terminal escape codes, notably the `cursorpos` one to set the console position. The statement `printf("\x1b[2;0H");` sets the cursor to row 2, character 0. Remember the padding border in the init function, a pos of 0,0 is not going to be the top-left of the screen if you have a border set.

An example of printing to the console looks like:

```
int main()
{
	// Initialise the video interface.
	video_initialise()

	// Initialise the console.
	CON_init(xfb, 20, 20, rmode->fbWidth, rmode->xfbHeight, rmode->fbWidth * VI_DISPLAY_PIX_SZ)

	// Set the cursor to row 2.
	printf("\x1b[%d;%dH", 2, 0);

	// Print the text.
	printf("Hello World!");
}
```

Drawing to the framebuffer

Drawing to the framebuffer directly is the simplest method of drawing to the screen. All that's required is writing the pixel values to the array directly inbetween clears each frame.

RFCs

Suggestions for how all our different components should talk to one another

In game client to server controls

Scope

THis document describes the messages clients will send to the server to inform the server about inputs on their end.

The following assumptions are made:

- UDP
- Packet length up to 512 bytes total (https://stackoverflow.com/questions/1098897/what-is-the-largest-safe-udp-packet-size-on-the-internet ?)
- No need for auth, ect, players ips will be known from the server so will recognize senders like that

Packet structure

Field	Length	Description	Notes
Message Type	1 byte	Indicates the type of data of the remainder of the packet	I really hope we never get past 255 different types....
Content	Up to 511 bytes	The actual data of the request	

Message Type

The following values are accepted for the Message Type field.

Value	Description

0	Basic Controls
1-255	Reserved for future use

Content

Basic Controls

A message of this type is 1 byte long. Each bit has the following meaning:

Value	Description
0	↑
1	↓
2	←
3	→
4	□
5	□
6	Select
7	Start

TODO

Soemthing to say the client is leaving the game? IDK?