

# Hello IOP !

## *"Building IRX modules for IOP"*

Version 1.0, 2002/03/07 – Copyright 2002 © David Ryan (Oobles), oobles@hotmail.com

### Introduction

The Playstation 2 contains a two processor architecture containing the Emotion Engine (EE) and IO Processor (IOP). This tutorial and demo code covers all the basics you need to know to create an IOP IRX Module and communicate with it from the EE. In the process of describing this, I will also cover how to import functions and export functions from your module so other modules can call them.

With any luck I'll get through the following:

- ? Introduction to disassembly.
- ? Importing functions from internal IOP modules.
- ? Creating a file driver
- ? Loading Modules from EE
- ? Communicating with the SIF
- ? Exporting function from your module.
- ? Creating Threads and communicating from EE

Before starting, I'd like to thank Dreamtime (Tony) for this great document layout which he used for his PS2 Tutorial #1. Gustavo should also be thanked again and again for his ps2lib, because much of this is an extension of that work. I should also plug my web site at <http://ps2dev.livemedia.com.au>; which is where to find much of the latest and greatest information on the web about PS2DEV. Lastly.. to all the guys on EFNET #ps2dev, keep up the good work.

For those people who will skip the tutorial and jump straight to the code, the end result of this tutorial is the following programs:

- Export – Example of exporting functions from a module.
- FileDriver – Example of a file driver.
- RPC – Example of creating a RPC driver.
- Loader – Main program which loads and calls above modules.

In the processes of doing all this a lot of functions have been added to the IOP section of PS2LIB. To compile the source code you will need the updated version of PS2LIB which you can get hold of via CVS at [ps2dev.livemedia.com.au](http://ps2dev.livemedia.com.au) (I will try to organise a nightly snapshot so a CVS client is not required).

## Development Environment

Before you start, you're going to need to have set up both the EE and IOP compilers; preferably following the guides at [ps2dev.livemedia.com.au](http://ps2dev.livemedia.com.au) or by now3d at [ps2dev.sf.net](http://ps2dev.sf.net). This will ensure you do not need to change the makefile to compile. You will also need Hannimar's disassembler, PS2DIS. This is a great mips disassembler and makes it much easier to look around compiled code.

I used Naplink to develop everything using the demo disk swap method to get it started.

## Introduction to MIPS Disassembly

I thought it might be useful to start with a short tutorial on disassembling code and some things to look out for on MIPS assembler. I was completely new to this at the start of the week and once you know the basics its not too difficult. The main thing I'll discuss is function calls in mips and how to see the parameters. This is important when trying to work out the function declarations of various internal functions.

I found the best place to start with this, was to grab PS2DIS and load up a module such as SNDDR.V.IRX which if you're lucky will have symbols included. You will know if symbols are included because PS2DIS will display the symbol "start" in the left column after the address. PS2DIS actually opens the file and places the cursor at the first function to be called when the module is loaded. Its worth loading up a few different modules to get a feel for how PS2DIS works.

When I first looked at the disassembled code it all looked very confusing and thought it would take forever to understand it. Thankfully that passed reasonably quickly. The first thing you need to know is about MIPS registers. Tony's Tutorial #1 mentions a few MIPS sources which you should also have. He also mentioned the registers (here's a cut and paste).

```
#define zero    $0        // Always 0
#define at     $1        // Assembler temporary
#define v0     $2        // Function return
#define v1     $3        //
#define a0     $4        // Function arguments
#define a1     $5
#define a2     $6
#define a3     $7
#define t0     $8        // Temporaries. No need
#define t1     $9        // to preserve in your
#define t2     $10       // functions.
#define t3     $11
#define t4     $12
#define t5     $13
#define t6     $14
#define t7     $15
#define s0     $16       // Saved Temporaries.
#define s1     $17       // Make sure to restore
#define s2     $18       // to original value
#define s3     $19       // if your function
#define s4     $20       // changes their value.
#define s5     $21
#define s6     $22
#define s7     $23
#define t8     $24       // More Temporaries.
#define t9     $25
#define k0     $26       // Reserved for Kernel
#define k1     $27
#define gp     $28       // Global Pointer
#define sp     $29       // Stack Pointer
#define fp     $30       // Frame Pointer
#define ra     $31       // Function Return Address
```

Just to spell out what's already there. Its important to know that the first four arguments of any function are passed in the the registers a0 to a3 (5<sup>th</sup>+ are placed on the stack), and that values are returned in v0 and v1. One of the other interesting things is that when calling a function the instruction following the jal (jump and link) instruction also gets executed before the jump. So to see what this means this is what a call to void \* strcpy(char \*, char \*) might look like when compiled.

```
li    a0, $00000400 (buffer)
jal   $00000240 (+100 strcpy)
li    a1, $00000500 ("This is a test string\n")
sw    v0, $0010(sp)
```

So the above places an empty buffer into a0, the string to be copied in a1 and then calls strcpy. The return value has been placed on the stack for use later.

A great function of PS2DIS is that Hanimar has implemented an analyse function. This allows you to track down where a function has been called from. Simply find the function you want to track, press "space" on the keyboard and then hit F3. It will automatically track down where the function is used. Using this method its not too difficult to work out how to call any function in the PS2 internal modules.

I used these basic principles to create all the C header files used in this code.

## Hello IOP

Now, back to the point of this tutorial; getting to know the IO Processor. The IOP is actually the PS1 on a single chip, complete with 2MB of its own memory. As the name suggests it handles all input/output devices on the PS2. This is a very smart design as it offloads device interrupts and device handling off the main processor. This smart design does however add to the complexity of software.

The problem with having a separate CPU is that its not a trivial to communicate with it. Sony have combined DMA and a Serial Interface (SIF) to solve the problem. The SIF is like a very close/fast wire between the EE and IOP. Any function in EE that needs to access an IO device must use the SIF/DMA combination to send messages to the IOP where the function is performed and a message is returned.

To ease the pain of using this method Sony have built two SIF libraries. A simple command interface, and a Remote Procedure Call (RPC) interface. The RPC interface is built on top of the command interface. This tutorial explains and provides an example of the RPC method; I'll leave it up to someone else to do the command method.

Additional to programming directly to the SIF interfaces, there is another way of communicating with the IOP. The file driver (ioman) on the IOP already provides a interfaces to open/read/write/close files from the EE. By adding your own file driver sub-system you can get a quick and dirty way of communicating with the IOP from EE. This method is also how Naplink allow you to open a file from the host pc.

## Importing Functions in IRX Modules

One of the great things about the PS2 is that it has an abundance of functions already available to you in the BIOS. Functions like creating and managing threads and even a standard c library with functions like strcpy,strlen,etc. It even has printf which when using naplink will automatically print directly to your Naplink console on the PC! This Naplink feature is great and I'm still wondering how Napalm hooked into this standard function (I haven't disassembled their source yet).

The way the PS2 imports functions in IRX files is via a data block which gets linked during load. Each module to be linked has the following format.

```

0x41e00000 // Magic number marking start of module import.
0x00000000 // Unknown?
0x00000102 // Version Number. This Case is 1.2

“stdio\0\0” // 8 byte module name.

```

Each function is listed as an offset into the modules function list. In this case below we are importing the printf function which is at position four in the stdio module:

```

jr ra
li zero, $00000004 // Where four is the function offset into the Module.

```

During loadtime these references are replaced with the true function call of the module. The great thing about this is that you can search through various IRX files and find functions to call then work out its parameters and create a C declaration. The hard bit is finding an IRX with symbols still included so you can simply find where its called and re-create the function call in C.

In the process of developing this demo and tutorial I have created many new import libraries in PS2LIB. Each module has been compiled into separate libraries which you can link with individually. Many functions are still missing and even more still don't have C declarations for them. If you either find a new function or create a C declaration for something not already done, please send me the changes. To give you an idea here are the various import libraries available

**INTRMAN:** Interrupt manager functions.  
**IOMAN:** File driver functions.  
**LIBSD:** Sound Processor functions  
**LOADCORE:** Core misc functions  
**SIFCMD:** Serial Interface functions  
**SIFMAN:** Serial Interface manager functions  
**STDIO:** Printf function.  
**SYSCLIB:** Standard C library functions  
**SYSTEMEM:** Memory management functions  
**THBASE:** Threading functions  
**THSEMAP:** Semaphore functions.

## Loading Modules from EE

After you've created your first module. Maybe something as simple as:

```

#include <stdlib.h>

void start( int argc, char ** argv )
{
    printf( “Hello IRX Modules!\n”);
}

```

You will need to take the compiled ELF and run it through Gustavo's ELF2IRX tool. There are some problems in this tool which will hopefully be fixed soon. I found for basic importing of functions that the current version is working fine.

Once you have created the IRX you will need some way of loading the module. Naplink does include a function called `execiop` which should allow you to load it directly. I found however that this did not work. Included in the source is a program called `loader` which is simply a small loading and test program with a hard coded filenames. Simply change the filename and path to the format:

```
host:<drive>:<path><filename>
```

After you've compiled your basic IOP module and your loader you can use Naplink to load both using the normal execee command.

```
execee host:<drive>:<path>loader.elf
```

I'm not going to explain the various other helper files that were included in building the basic elf and irx. You can look through the makefile and sources to see how it all comes together. Just remember that the loader is built using EE compilers and IRX is built using IOP compilers. This makes for a slightly more complex makefile, but nothing you can't handle! :)

If you were to execute the and load the module described above you would get the message "Hello IRX Modules!" printed directly to your PC on the Naplink Console.

Notice that the start function also has argc and argv parameters. This allows you to send various arguments to the module when loading. The arguments are passed to the module from the loadModule function on the EE. This can be useful for configuring your module on startup.

## The File Driver

As mentioned earlier its possible to add your own file driver sub-system onto the file system of the PS2. This means that you can add your own handler for different file devices on the IOP. The program specifies the handler using a URL style file naming convention. To load from the CDROM an EE program would open

```
CDROM:<path><filename>
```

The CDROM tells the base file driver which backend module will need to actually perform the open and all other subsequent operations. Naplink have added in their own "HOST" file driver which is why you specify to load your programs using

```
execee host:<path><filename>
```

The execee command could just as easily load programs from the CDROM, or other installed file sub-system.

The file driver example provided is very simple and is called the "FD" driver. Any request to read from a file opened on the FD driver will return the string "Hello IOP!". Any write to a file opened with the FD driver will call the printf function and write the contents to the Naplink console. I have used this method in the loader to print messages from the EE back to the Naplink console. This is not how you should write messages from EE. Look at the nmpmprintf function to see how to do this correctly.

Something else to note about the Naplink host file driver. The Naplink program on the PC treats the directory it was run from as the base directory. In the loader code I have loaded modules using "HOST:<module.IRX>" with no path. So if you run naplink on the PC from the path you build your IRX modules to, you will not need to changes the source code.

I have noticed that there is an MCMAN (Memory Card Manager) library on IOP which creates a file driver subsystem. This should allow any program to simply open a file on a memory card using a standard fio\_open command on EE. If anyone does this, let me know.

## Remote Procedure Call

As described earlier, the Remote Procedure Call(RPC) interface provides an easier method of communicating with IOP modules from EE than using the SIF directly. The RPC method is exactly the same as many other RPC technologies like CORBA. Function parameters are marshalled into a data buffer, sent to the server where the parameters are passed to the real function. Building this type of communication system is usually tedious work, however the library makes it a bit easier.

When you set up an RPC server in your IOP module it needs to be running on a separate thread. This is because each RPC server calls a function `sceSifRpcLoop` which never returns. It receives the RPC requests and dispatches them to your handler function. Look in the `rpc/driver.c` file to see how the RPC is setup.

The RPC uses a magic number when registering the handler. This magic number is then used by the RPC client to bind to the correct handler. This number must be unique on the system. Look in the `loader.c` file to see how the bind operation is called. The magic number used when registering the RPC server and binding from the client is `0x80000120`.

After the client binds to the RPC server you are ready to start sending messages. The actual data buffers are sent via DMA transfer. In pukko's `padlib` code I found buffers aligned to 16 byte boundaries. This will be to make sure the DMA works correctly. The `loader.c` code demonstrates how to do this. The `sif_call_rpc` function is passed a command as the second parameter which is provided as the first parameter of the `rpc` handler function. The buffers for input and output and size of those buffers are also passed to the function. The `rpc` handler function receives the input buffer and size. It must return a pointer to the return buffer. The size returned is set by the client. The `rpc` handler function has no control of the size of this buffer.

The example provided simply prints out the input buffer to the Naplink console. I suggest that if you are doing a real RPC handler that you set up structures to hold your various function parameters.

## Exporting function from your module.

At this point you can create and load modules and send commands to your module via the SIF `rpc` method. The last thing that you might want to do is allow other modules on the IOP to call functions directly in your module. This is done in a very similar way to importing functions. The export part of the file looks as follows:

```
0x41c00000 // Magic number marking start of module export.
0x00000000 // nop.
0x00000101 // Version Number. This Case is 1.1

"mytest\0\0" // 8 byte module name.
```

Then each function looks like this when disassembled:

```
.word $000000f8 (Initialise)
.word $00000110 (someFunction)
.word 0
```

Refer to `exp.s` in the `export` directory to see how this has been setup. The file `exp_imp.s` in the `rpc` directory is importing the functions from the export module. The only other thing you must remember if that you must export the function table when your module is loaded. Look at the `export.c` to see how this is done.

Your functions should begin in slot four of the export table. Slot zero is called automatically on startup. Slots one, two and three are possibly used for other purposes.

All module functions I have seen so far begin at slot four, so there's no reason to be any different! You should also remember to put place holder functions in any empty slots that simply return when called. Have a look at LIBSD.IRX (the sound driver) for some good examples of an export library.

The only problem with the above description and code provided is that it doesn't work! Two things are causing problems. The exp.s file does not correctly fill in the location of the functions when linked. The values recorded to the export table are 0. The second problem is that the elf2irx currently available does not correctly setup an .iopmod segment with the export table. If anyone can shed any light on what I'm doing wrong please let me know so I can fix this up!

As a side note and something interesting to note. The LIBSD module is a pure set of export functions which can only be accessed from other IOP modules. Other modules such as SNDDRV are designed to be a server to the EE, allowing functions to be called through the Command SIF interfaces.

After you have created the module you will then need to create another file with the import definitions for the modules you wish to use the functions. Make sure you load the modules in the correct order. The module will fail to load if one of its import modules isn't available.

## Conclusion

Well that's what I've discovered about programming on the IOP in the last couple of weeks. Its a starting point to do useful things on the IOP. Hopefully people will be able to use this to start programming the Sound devices, Memory Cards, or anything else they can put their minds too. Let me know what you think by sending any comments, corrections and/or suggestions to [Oobles@hotmail.com](mailto:Oobles@hotmail.com)