PARAPIN:

A Parallel Port Pin Programming Library for Linux

Jeremy Elson
jelson@circlemud.org

Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292 USA

28 March 2000

This work was supported by DARPA under grant No. DABT63-99-1-0011 as part of the SCADDS project, and was also made possible in part due to support from Cisco Systems.

Contents

1 Introduction

`parapin` makes it easy to write C code under Linux that controls individual pins on a PC parallel port. This kind of control is very useful for electronics projects that use the PC's parallel port as a generic digital I/O interface. Parapin goes to great lengths to insulate the programmer from the somewhat complex parallel port programming interface provided by the PC hardware, making it easy to use the parallel port for digital I/O. By the same token, this abstraction also makes Parapin less useful in applications that need to actually use the parallel port as a *parallel port* (e.g., for talking to a printer).

Parapin has two "personalities": it can either be used as a user-space C library, or linked as part of a Linux kernel module. The user and kernel personalities were both written with efficiency in mind, so that Parapin can be used in time-sensitive applications. Using Parapin should be very nearly as fast as writing directly to the parallel port registers manually.

Parapin provides a simple interface that lets programs use pins of the PC parallel port as digital inputs or outputs. Using this interface, it is easy to assert high or low TTL logic values on output pins or poll the state of input pins. Some pins are bidirectional—that is, they can be switched between input and output modes on the fly.

Parapin was written by Jeremy Elson (`jelson@circlemud.org`) while at the University of Southern California's Information Sciences Institute. This work was supported by DARPA under grant No. DABT63-99-1-0011 as part of the SCADDS project, and was also made possible in part due to support from Cisco Systems. It is freely available under the GNU Public License (GPL). Up-to-date information about Parapin, including the latest version of the software, can be found at the Parapin Home Page[1].

Warning:

Attaching custom electronics to your PC using the parallel port as a digital I/O interface can damage both the PC and the electronics if you make a mistake. **If you're using high voltage electronics, a mistake can also cause serious personal injury. Be careful.**

If possible, use a parallel port that is on an ISA card, instead of one integrated onto the motherboard, to minimize the expense of replacing a parallel port controller that you destroy.

USE THIS SOFTWARE AT YOUR OWN RISK.

2 Parallel Port Specifications

This section will briefly outline some of the basic electrical operating characteristics of a PC parallel port. More detail can be found in the IBM Parallel Port FAQ[2], written by Zhahai Stewart[3], or the PC Parallel Port Mini-FAQ[4] by Kris Heidenstrom[5]. Those documents have a lot of detail about registers, bit numbers, and inverted logic—those topics won't be discussed here, because the entire point of Parapin is to let programmers control the port *without* knowing those gory details.

The PC parallel port usually consists of 25 pins in a DB-25 connector. These pins can interface to the TTL logic of an external device, either as inputs or outputs. Some pins can be used as inputs only, while some can be switched in software, on-the-fly, between input mode and output mode. Note, however, that it can be dangerous for two devices to assert an output value on the same line at the same time. Devices that are using bidirectional pins must agree (somehow) on who is supposed to control the line at any given time.

From the point of view of Parapin, the pinout of the parallel port is as follows:

[1] http://www.circlemud.org/jelson/software/parapin
[2] http://home.rmi.net/~hisys/parport.html
[3] http://home.rmi.net/~hisys/zstewart.html
[4] http://home.clear.net.nz/pages/kheidens/ppmfaq/khppmfaq.htm
[5] http://home.clear.net.nz/pages/kheidens/

Pin	Direction
1	In/Out
2-9	In/Out (see note)
10	Input, Interrupt Generator
11	Input
12	Input
13	Input
14	In/Out
15	Input
16	In/Out
17	In/Out
18-25	Ground

Pins 2-9—called the parallel port's "Data Pins"—are *ganged*. That is, their directions are not individually controllable; they must be either all inputs or all outputs. (The actual *values* of the pins—on or off—*are* individually controllable.) Also, some of the oldest parallel ports do not support switching between inputs and outputs on pins 2-9 at all, so pins 2-9 are always outputs. Many PC motherboards allow the user to select the personality of the parallel port in the BIOS. If you need to use pins 2-9 as bidirectional or input pins, make sure your port is configured as a "PS/2" port or one of the other advanced port types; ports in SPP (Standard Parallel Port) mode may not support direction switching on pins 2-9.

Pin 10 is special because it can generate interrupts. Interrupt handling is discussed in more detail in Section 9.

Output pins can assert either a TTL high value (between +2.4v and +5.0v), or a TTL low (between 0v and +0.8v). The port can not source much current. Specs for different implementations vary somewhat, but a safe assumption (staying within spec) is that the voltage will be at least 2.5v when drawing up to 2.5mA. In reality you can sometimes get away with using an output pin to power a component that uses 3v or even 5v logic, but the supplied voltage may sag if more than 2.5mA is drawn. Input pins are typically spec'ed to be able to sink up to about 20mA. For a more detailed treatment of the parallel port electronics, see the references above.

3 Parapin Basics

Parapin has two personalities: one for plain user-space operation (i.e., linking Parapin with a regular C program), and one for use with Linux kernel modules. The two personalities differ in how they are compiled (Section 4) and initialized (Section 5). Also, only the kernel version is capable of servicing parallel-port interrupts that are generated from Pin 10. Otherwise, the two personalities are the same.

The user-space library version of Parapin works very much like any other C library. However, kernel programming differs from user-space programming in many important ways; these details are far beyond the scope of this document. The best reference I've ever found is Alessandro Rubini's fantastic book, Linux Device Drivers[6]. Go buy it from your favorite technical bookstore; or, if you don't have a favorite, use my favorite: BookPool[7]. If you don't want to buy the book, a far inferior alternative (but better than nothing) is the free Linux Kernel Hacker's Guide[8].

In Parapin, there are five basic operations possible:

- Initialize the library (Section 5)
- Put pins into input or output mode (Section 6)

[6] http://www.ora.com/catalog/linuxdrive/
[7] http://www.bookpool.com/
[8] http://www.linuxdoc.org/LDP/khg/HyperNews/get/khg.html

- Set the state of output pins—i.e., high/set or low/clear (Section 7)
- Poll the state of input pins (Section 8)
- Handle interrupts (Section 9)

Each of these functions will be discussed in later sections.

Most of the functions take a *pins* argument—a single integer that is actually a bitmap representing the set of pins that are being changed or queried. Parapin's header file, `parapin.h`, defines a number of constants of the form "`LP_PINnn`" that applications should use to specify pins. The `nn` represents the pin number, ranging from 01 to 17. For example, the command

```
set_pin(LP_PIN05);
```

turns on pin number 5 (assuming that pin is configured as an output pin; the exact semantics of `set_pin` are discussed in later sections). C's logical-or operator can be used to combine multiple pins into a single argument, so

```
set_pin(LP_PIN02 | LP_PIN14 | LP_PIN17);
```

turns on pins 2, 14, and 17.

Usually, it is most convenient to use `#define` statements to give pins logical names that have meaning in the context of your application. The documentation of most ICs and other electronics gives names to I/O pins; using those names makes the most sense. For example, a National Semiconductor ADC0831 Analog-to-Digital converter has four I/O pins called `VCC` (the supply voltage), `CS` (chip select), `CLK` (clock), and `DO` (data output 0). A fragment of code to control this chip might look something like this:

```
#include "parapin.h"

#define VCC LP_PIN02
#define CS  LP_PIN03
#define CLK LP_PIN04
#define DO  LP_PIN10 /* input pin */
...
clear_pin(CS);  /* pull Chip Select low, tell it to acquire */
...
set_pin(CLK);   /* clock it */
clear_pin(CLK);
```

This method has a number of advantages. First, it makes the code more readable to someone who has the ADC0831 documentation in hand. Second, the `#define` statements summarize the mappings of IC pins to parallel port pins, making it easier to build the physical interface. Also, if the physical interface changes, the `#define`s make it easy to remap parallel port pins to new IC pins.

Parapin's header file also provides these constants in an array called `LP_PIN`. This array is useful in some contexts where the pin number is being specified with a variable instead of a constant. For example:

```
/* runway lights: light up pins 1 through 9 in sequence */
while (1) {
        for (i = 1; i <= 9; i++) {
                set_pin(LP_PIN[i]);
                usleep(200);
                clear_pin(LP_PIN[i]);
        }
}
```

Code such as the above fragment would be much more complex if the programmer were forced to use constants. There is a direct correspondence between indices in the array and pin numbers; accordingly, only indices from 1 to 17 should be used. Programs should never reference `LP_PIN[0]` or `LP_PIN[i]` where $i > 17$.

4 Compiling and Linking

4.1 The C Library Version

The user-space version of Parapin is compiled and linked very much like any other C library. If you installed Parapin on your system using "make install", the library (`libparapin.a`) was probably installed in `/usr/local/lib`. The header file with the library's function prototypes and other definitions, `parapin.h`, is probably also in `/usr/local/include`.

Note well: *Parapin must be compiled with gcc using compiler optimization of* `-O` *or better.* This is because Parapin uses the `inb` and `outb` functions that exist only with compiler optimization.

To use the library, first make sure to `#include` parapin.h in your C source file. When linking, add `-lparapin` along with any other libraries you might be using.

4.2 The Kernel Module Version

As with the C library version, kernel source code that uses Parapin must `#include` parapin.h. Make sure that `__KERNEL__` is defined before the include statement.

Note well: *Parapin must be compiled with gcc using compiler optimization of* `-O` *or better.* This is because Parapin uses the `inb` and `outb` functions that exist only with compiler optimization.

Linking is a little more complicated. As with any other interdependent kernel modules, there are actually two ways of using parapin with your Linux device driver:

- Compile Parapin independently, insert it into the kernel as its own module using `insmod`, and then insert your driver separately using a different `insmod` command. In this case, insmod and the kernel resolve the links between your driver and Parapin's functions.
- Compile Parapin right into your device driver, so that the entire thing is inserted with a single `insmod` command. This can be less confusing for users if you're distributing your device driver to other people because it eliminates the extra step of finding, compiling, and inserting a separate module. On the other hand, it will cause conflicts if a user tries to insert two different modules that use Parapin at the same time.

The Makefile that comes with Parapin compiles Parapin into its own independent module, called `kparapin.o`. This module can be inserted into the kernel using `insmod` (as described in the first method). If you'd rather use the second method, and link Parapin directly with your device driver module instead, do the following:

1. Copy parapin.c into your driver's source directory.
2. In your driver's Makefile, compile kparapin.o from parapin.c, *with* the `-D__KERNEL__` directive, but *without* the `-DMODULE` directive.
3. Link kparapin.o with all the other object files that make up your driver.

If your driver only consists of one other source file, and you have never linked multiple .o's together into a single kernel module, it's easy. First, compile each .c file into a .o using `gcc -c` (the same as with normal C programs that span multiple source files); then, link all the .o's together into one big .o using `ld -i -o final-driver.o component1.o component2.o component3.o ...`.

Important note: Whichever method you use, Parapin also requires functions provided by the standard Linux kernel module `parport`. Make sure you insert the `parport` module before inserting any modules that use Parapin; otherwise, you will get unresolved symbols.

5 Initialization and Shutdown

Before any other Parapin functions can be used, the library must be *initialized*. Parapin will fail *ungracefully* if any functions are called before its initialization. This is because the library's other functions do not check that the library is in a sane state before doing their jobs. This was an intentional design decision because the maximum possible efficiency is required in many applications that drive the parallel port at high speeds. My goal was for Parapin to be almost as fast as directly writing to registers.

The exact initialization method varies, depending on if Parapin is being used as a C library or as a kernel module.

5.1 The C Library Version

C library initialization is performed using the function

```
int pin_init_user(int lp_base);
```

whose single argument, `lp_base`, specifies the base I/O address of the parallel port being controlled. Common values of `lp_base` are 0x378 for LPT1, or 0x278 for LPT2; parapin.h defines the constants `LPT1` and `LPT2` to these values for convenience. However, the exact port address may vary depending on the configuration of your computer. If you're unsure, the BIOS status screen displayed while the computer boots usually shows the base addresses of all detected parallel ports.

Programs using Parapin must be running as root when they are initialized. Initialization of the library will fail if the process is owned by any user other than the super-user because Parapin has to request the right to write directly to hardware I/O registers using the `ioperm` function. The security-conscious programmer is encouraged to drop root privileges using `setuid` after a successful call to `pin_init_user`.

The return value of `pin_init_user` will be 0 if initialization is successful, or -1 if there is an error. Applications must not call other Parapin functions if initialization fails. Failed initialization is usually because the program is not running as root.

No shutdown function needs to be called in the C library version of Parapin.

5.2 The Kernel Module Version

Initialization and shutdown in the kernel flavor of Parapin is done using the following two functions:

```
int pin_init_kernel(int lpt,
                    void (*irq_func)(int, void *, struct pt_regs *));
void pin_release();
```

The first function is not as intimidating as it looks. Its first argument, `lpt`, is the parallel port number that you want to control. The number references the kernel's table of detected parallel ports; 0 is the first parallel port and is a safe guess as a default.

The second argument, `irq_func`, is a pointer to a callback function to be used for servicing interrupts generated by the parallel port. This argument may be NULL if the driver does not need to handle interrupts. Details about interrupt handling are discussed in Section 9.

`pin_init_kernel` will return 0 on success, or a number less than 0 on error. The return value will be a standard `errno` value such as `-ENODEV`, suitable for passing up to higher layers. If Parapin initialization fails, the driver *must not* call Parapin's other functions. As described earlier, this requirement is not enforced for efficiency reasons.

When a driver is finished controlling the parallel port using Parapin, it must call `pin_release`. The state of the parallel port's interrupt-enable bit will be restored to the state it was in at the time `pin_init_kernel` was originally called.

Parapin's `pin_init_kernel` and `pin_release` functions work with the Linux kernel's standard `parport` facility; as noted above, the `parport` module must be loaded along with any module that uses Parapin.

When initialized, Parapin will register itself as a user of the parallel port, and claim *exclusive access* to that port. This means no other processes will be allowed to use the parallel port until `pin_release` is called.

6 Configuring Pins as Inputs or Outputs

As shown in the table in Section 2, most of the pins on modern parallel ports can be configured as either inputs or outputs. When Parapin is initialized, it is safest to assume that the state of all these switchable pins is *undefined*. That is, they must be explicitly configured as outputs before values are asserted, or configured as inputs before they are queried.

As mentioned earlier, Pins 2-9 can not be configured independently. They are always in the same mode: either all input or all output. Configuring any pin in that range has the effect of configuring all of them. The constant `LP_DATA_PINS` is an alias that refers to all the pins in this range (2-9).

The other switchable pins (Pins 1, 14, 16, and 17) may be independently configured. Pins 10, 11, 12, 13, and 15 are always inputs and can not be configured.

It is also worth reiterating that having two devices both try to assert (output) a value on the same pin at the same time can be dangerous. The results are undefined and are often dependent on the exact hardware implementation of your particular parallel port. This situation should be avoided. The protocol spoken between the PC and the external device you're controlling to should always agree who is asserting a value on a pin and who is reading that pin as an input.

Pins are configured using one of the following three functions:

```
void pin_input_mode(int pins);
void pin_output_mode(int pins);
void pin_mode(int pins, int mode);
```

The `pins` argument of all three functions accepts the `LP_PINnn` constants described in Section 3. The `pin_mode` function is just provided for convenience; it does the same thing as the `pin_input` and `pin_output` functions. Its `mode` argument takes one of the two constants `LP_INPUT` or `LP_OUTPUT`. Calling `pin_mode(pins, LP_INPUT)` is exactly the same as calling `pin_input(pins)`.

Examples:

```
pin_input_mode(LP_PIN01);
/* Pin 1 is now an input */

pin_output_mode(LP_PIN14 | LP_PIN16);
/* Pins 14 and 16 are now outputs */

pin_mode(LP_PIN02, LP_INPUT);
/* Pins 2 through 9 are now ALL inputs */

pin_mode(LP_PIN01 | LP_PIN02, LP_OUTPUT);
/* Pin 1, and Pins 2-9 are ALL outputs */

pin_input_mode(LP_DATA_PINS);
/* The constant LP_DATA_PINS is an alias for Pins 2-9 */
```

7 Setting Output Pin States

Once Parapin has been initialized (Section 5), and pins have been configured as output pins (Section 6), values can be asserted on those pins using the following functions:

```
void set_pin(int pins);
void clear_pin(int pins);
void change_pin(int pins, int state);
```

The `pins` argument of all three functions accepts the `LP_PINnn` constants described in Section 3. `set_pin` turns pins *on*, electrically asserting high TTL values. `clear_pin` turns pins *off*, electrically asserting low TTL values. The convenience function `change_pin` does the same thing as `set_pin` and `clear_pin`; its `state` argument takes one of the constants `LP_SET` or `LP_CLEAR`. Calling `change_pin(pins, LP_SET)` has exactly the same effect as calling `set_pin(pins)`.

These three functions will *only* have effects on pins that were previously configured as output pins as described in Section 6. Attempting to assert a value on an input pin will have no effect.

Note that while the *direction* of Pins 2-9 must be the same at all times (i.e., either all input or all output), the actual *values* of these pins are individually controllable when they are in output mode.

Examples:

```
pin_output_mode(LP_PIN01|LP_DATA_PINS|LP_PIN14|LP_PIN16|LP_PIN17);
/* All these pins are now in output mode */

set_pin(LP_PIN01 | LP_PIN04 | LP_PIN07 | LP_PIN14 | LP_PIN16);
/* Pins 1, 4, 7, 14, and 16 are all on */

clear_pin(LP_PIN01 | LP_PIN16);
/* Pins 1 and 16 are now off */

change_pin(LP_PIN01 | LP_PIN02, some_integer ? LP_SET : LP_CLEAR);
/* Pins 1 and 2 are now off if and only if some_integer == 0 */
```

8 Polling Input Pins

Once Parapin has been initialized (Section 5), and pins have been configured as input pins (Section 6), the value being asserted by the "far end" can be queried using the following function:

```
int pin_is_set(int pins);
```

Any number of pins can be queried simultaneously. The `pins` argument accepts the `LP_PINnn` constants described in Section 3. The return value is an integer in the same format. Any pins that are set (electrically high) will be set in the return value; pins that are clear (electrically low) will be clear in the return value.

Pins may only be queried if:

- They are permanent input pins (Pins 10, 11, 12, 13, and 15), as described in Section 2, **or**
- They are bidirectional pins previously configured as input pins, as described in Section 6.

Any query to an output pin will always return a value indicating that the pin is clear. In other words, this function *can not* be used to determine what value was previously asserted to an output pin.

Examples:

```
pin_input_mode(LP_PIN01 | LP_DATA_PINS | LP_PIN17);
/* Pins 1, 2-9, and 17 are now in input mode, along with
   Pins 10, 11, 12, 13, and 15, which are always inputs */

/* check the state of pin 1 */
printf("Pin 1 is %s!\n", pin_is_set(LP_PIN01) ? "on" : "off");
```

```
/* check pins 2, 5, 10, and 17 - demonstrating a multiple query */
int result = pin_is_set(LP_PIN02 | LP_PIN05 | LP_PIN10 | LP_PIN17);

if (!result)
        printf("Pins 2, 5, 10 and 17 are all off\n");
else {
        if (result & LP_PIN02)
                printf("Pin 2 is high!\n");
        if (result & LP_PIN05)
                printf("Pin 5 is high!\n");
        if (result & LP_PIN10)
                printf("Pin 10 is high!\n");
        if (result & LP_PIN17)
                printf("Pin 17 is high!\n");
}
```

9 Interrupt (IRQ) Handling

The kernel-module version of Parapin lets parallel port drivers catch interrupts generated by devices connected to the parallel port. Most hardware generates interrupts on the rising edge[9] of the input to Pin 10.

Before Parapin's interrupt-handling can be used, the Linux kernel itself must be configured to handle parallel port interrupts. Unlike most other hardware devices, the kernel does not detect or claim the parallel port's interrupts by default. It is possible to manually enable kernel IRQ handling for the parallel port by writing the interrupt number into the special file `/proc/parport/`n`/irq`, where n is the parallel port number. For example, the following command tells the kernel that `parport0` is using IRQ 7:

```
echo 7 > /proc/parport/0/irq
```

If parallel port support is being provided to the kernel through modules, it is also possible to configure the IRQ number as an argument to the `parport_pc` module when it is loaded. For example:

```
insmod parport
insmod parport_pc io=0x378 irq=7
```

Note that both the `io` and `irq` arguments are required, even if the parallel port is using the default I/O base address of 0x378.

The actual interrupt number used by the kernel (7 in the examples above) must, of course, match the interrupt line being used by the hardware. The IRQ used by the parallel port hardware is usually configured in the BIOS setup screen on modern motherboards that have built-in parallel ports. Older motherboards or stand-alone ISA cards usually have jumpers or DIP switches for configuring the interrupt number. The typical assignment of interrupts to parallel ports is as follows:

Port	Interrupt
LPT1	7
LPT2	5

These are reasonable defaults if the actual hardware configuration is not known.

As described in Section 5, the `pin_init_kernel()` function allows installation of an interrupt handler function by passing a pointer to the handler as the second argument. A `NULL` value for this parameter means that interrupts are disabled. A non-NULL value should be a pointer to a callback function that has the following prototype:

[9] Legend has it that some very old parallel port hardware generates interrupts on the *falling* edge.

```
void my_interrupt_handler(int irq, void *dev_id, struct pt_regs *regs);
```

If and only if a pointer to such a handler function is passed to `pin_init_kernel`, the following functions can then be used to turn interrupts on and off:

```
pin_enable_irq();
pin_disable_irq();
```

Parapin turns parallel port interrupts off by default. That is, no interrupts will be generated until after a call to `pin_init_kernel()` to install the interrupt handler, *and* a subsequent call to `pin_enable_irq();`

Interrupts must not be enabled and disabled from within the interrupt handler itself. In other words, the interrupt handling function passed to `pin_init_kernel()` *must not* call `pin_enable_irq()` or `pin_disable_irq()`. However, the handler function *does not* need to be reentrant: the IRQ line that causes an interrupt handler to run is automatically disabled by the Linux kernel while the interrupt's handler is running. There are other important restrictions on kernel programming in general and interrupt-handler writing in particular; these issues are beyond the scope of this document. For more details, the reader is referred to the Linux kernel programming guides mentioned in Section 3.

Some PC hardware generates a spurious parallel port interrupt immediately after the parallel port's interrupts are enabled (perhaps to help auto-detection of the IRQ number). Parapin includes a workaround that prevents this interrupt from being delivered. This is done to ensure consistent interrupt behavior across all platforms.

10 Examples

Example code using Parapin is included in the Parapin distribution. Eventually, this section will have links to it...

11 Limitations and Future Work

Currently, Parapin only supports a single parallel port at a time. It is not possible to have a single instance of the library manage multiple instances of a parallel port. This may be a problem for software that is simultaneously trying to control multiple parallel ports. Someday, I may fix this, but it will make the interface messier (a port handle will have to be passed to every function along with pin specifiers).

The C-library version of Parapin should probably do better probing of the parallel port, but my desire to do this was limited because it replicates what is already done by the Linux kernel (and I usually use the kernel version of Parapin these days anyway).

If you have bug reports, patches, suggestions, or any other comments, please feel free to contact me at jelson@circlemud.org. I love getting feedback.

For the parallel port to work properly in a guest, it must first be configured properly on the host. Most issues involving parallel port functionality are a result of the host configuration. Check these areas of concern: the version of your Linux kernel, your device access permissions and the required modules. Parallel Ports and Linux 2.2.x Kernels. Parallel Ports and Linux 2.4.x Kernels. Parallel Ports and Linux 2.6.x Kernels. Device Permissions. Parallel Ports and Linux 2.2.x Kernels. The 2.2.x kernels that support parallel ports use the parport , parport_pc and vmppuser modules. Be sure tha