

PSP HACKS™

*Tips & Tools for Your Mobile Gaming
and Entertainment Handheld*



HACK
#39

Create an Infrared Peripheral Interface

In this article, you will learn how to get a microcontroller to communicate with the PSP and how to use it to make an interface for gamepads and more. Although it won't work with existing games out of the box, you can write your own homebrew games that use these controllers.

If you're lucky enough to own a Sony PSP capable of running homebrew software [Hack #40], you may have tried out some of the countless emulators and ports of older games out there already. The bad thing is that you can't play all the old games in two-player mode. Finding a solution for this was actually my motivation to start with this hack. First, it had to avoid any hardware modifications on the PSP itself, because I don't like to open such an expensive device. Since the PSP already provides infrared and USB ports, this wasn't much of a problem. Secondly, it had to be cheap. Being a student, I was looking for a cheap solution, and so I chose infrared. Aside from IR being cheaper, USB-capable controllers are hard to get.

As the interface consists of parts you can get at most electronic part distributors and doesn't require any special equipment, this project can be handled even by people with little programming and/or soldering experience.

Things You'll Need

- An ATmega8 microcontroller
- A programmer for the AVR (for loading your program onto the microcontroller)
- Two male SUB-D connectors (9-pin)
- 13 resistors, 1 k Ω
- One resistor, 470 Ω
- One infrared diode

If you build your own power supply:

- One LM7805 Voltage Regulator
- One electrolytic capacitor, 10 μF
- Two ceramic capacitors, 100 nFm
- One diode 1N4001

I also recommend downloading the ATmega8 Data Sheet (http://www.atmel.com/dyn/products/product_card.asp?part_id=2004) because it provides useful information about the microcontroller.

To create the software for the AVR, I have used AVR-GCC, which is included in the WinAVR Package (<http://winavr.sourceforge.net>). A programmer like

Create an Infrared Peripheral Interface

the AVR-PG2 from Olimex (<http://www.olimex.com>) will fit your needs. You can either buy them there or build your own one easily with the schematics available for download.

You will also need the newest version of the PSPSDK (<http://www.pspdev.org>). If you haven't already, you have to install Cygwin (<http://www.cygwin.com/>) in order to run PSPSDK on Windows [Hack #47].

Types of Gamepads

With every generation of game console, the gamepads have changed. Until the early 1990s, most gamepads consisted only of some buttons directly connected to the console. Since gamepads were designed with more and more buttons, new ways for reading out the data had to be found. Original PlayStation gamepads, for example, have a small microcontroller inside that handles all the button input, converts the analog stick data to binary values, and communicates with the console over a serial protocol.

For this project, I will use a Sega Master System gamepad, which consists of only six buttons and uses a 9-pin SUB-D connector. Figure 4-8 shows the schematic.

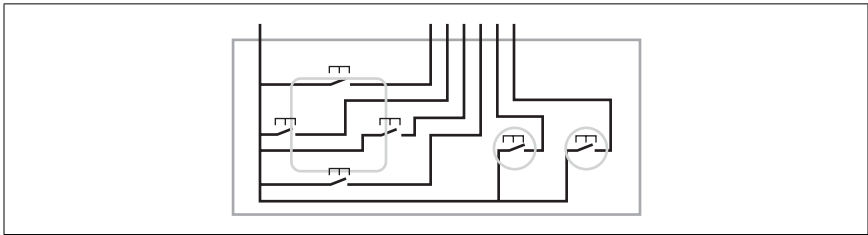


Figure 4-8. Schematic of the Sega Master System gamepad



If your eyes are starting to glaze over right now, don't fret. You just need a crash course. Dan O'Sullivan and Tom Igoe's *Physical Computing* (Course Technology PTR) explains everything involved with projects such as this, from the individual components up to the power supply, to all the skills you need to breadboard up your circuits.

Setting up the AVR

Before you can start with the fun, you have to get the AVR up and running. This isn't really difficult, since you only have to connect the supply voltage and ground, and then upload your program. Being a microcontroller, the AVR doesn't need any periphery.

Power supply. If you take a look at the data sheet, you will see that the ATmega8 accepts any value between 4.5 V and 5.5 V as supply voltage. This means you could either use a battery pack as power supply or build one using an LM7805 Voltage Regulator, which will provide a more stable 5 V supply voltage. Figure 4-9 shows a basic power supply design.

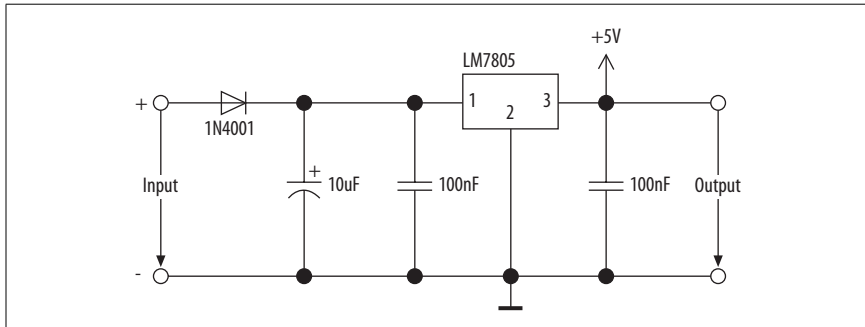


Figure 4-9. A power supply using the 7805 Voltage Regulator

If you decide to build a power supply using the 7805, the only thing you have to do is connect the input voltage, which can be any value between 8V and 35V, between the input and the ground pin. Even though it will work on its own, I strongly recommend that you connect a diode between input and pin 1, a 1 μ F and a 100 nF capacitor between Pin 1 and ground, and a 1 μ F capacitor between output and ground.

Having built your power supply, connect the output to the VCC pin of the AVR and tie the GND and AGND pins to ground. Connect a 1 k Ω resistor between VCC and RESET to make sure that the AVR isn't in reset mode, if no programmer is connected.

If you don't feel up to building the power supply, you can always fall back on the battery pack, or purchase a power supply that delivers 5 volts DC and use it in this project.

Now you can connect your programmer to the AVR.

Creating the Circuit

You can connect the buttons directly to the microcontroller using pull-down resistors. Pull-down resistors are necessary because each input pin of the microcontroller needs either a low or high state. If you choose the pins where you connect the buttons, you have to be careful about which ones you can use and which you cannot. For example, the RESET pin is necessary for programming the microcontroller and should not be disabled.

I connected the gamepads to the microcontroller, as shown in Figure 4-10.

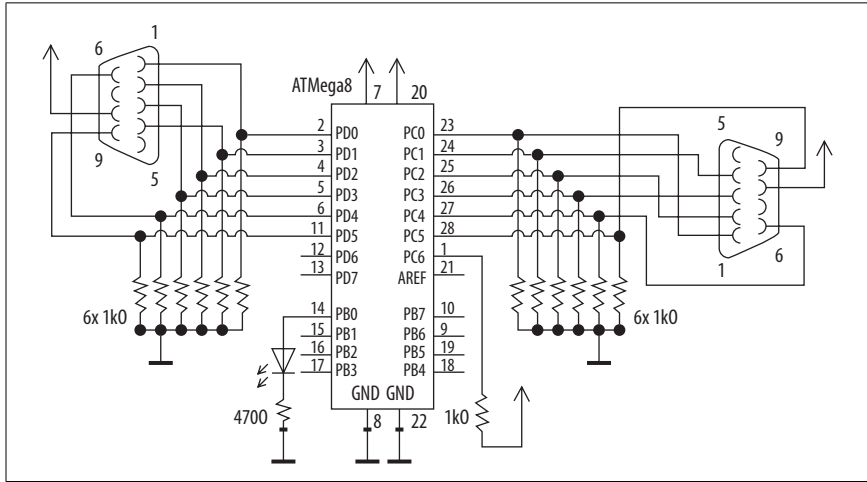


Figure 4-10. A schematic of the interface

The IR diode is directly connected to an I/O pin of the microcontroller with a resistor. The resistance is lower than the one you would use for a standard LED, since the way of using pulses enables the diode to shine brighter (explained previously).

Sending Data to the PSP

Now that the AVR is set up and ready to run your programs, you can start writing the program that sends data to the PSP. To do this, you need to understand how IRDA Data Transfer works.

If you already know a bit about serial data transmission, this should be easy for you to understand, because IRDA is very similar to RS232 transmission. Each frame has a start bit and a stop bit, and the standard baud rate of the PSP is set to 9600 bps. The difference to the standard serial protocol is that bits are determined as short pulses, and the byte to send has to be inverted. Figure 4-11 illustrates this.

Usually the length of the pulses is 3/16 of the bit time. The receiver gets a clearer signal when using pulses because the output power of the diode can be higher over such a short duration. With this information, you can write a function that sends one byte of data to the PSP. First, you have to include the necessary libraries:

```
#include <avr/delay.h>
#include <avr/io.h>

#define F_CPU 1000000UL
```

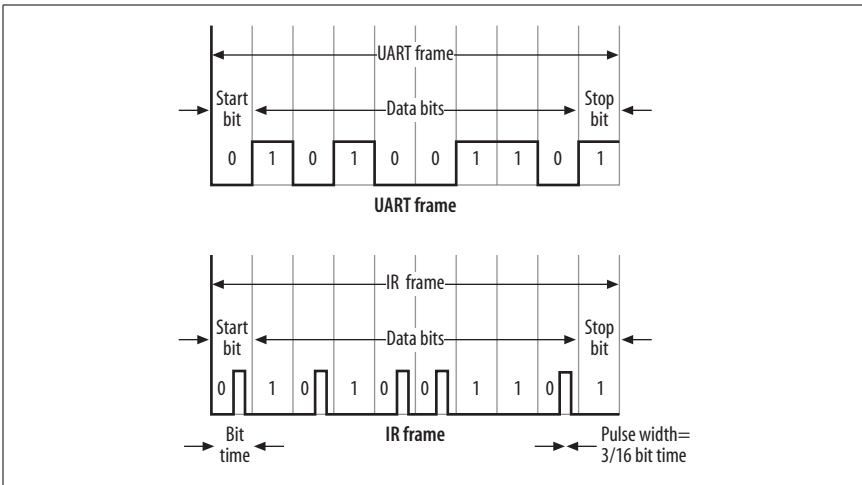


Figure 4-11. An IRDA Data Frame compared to an RS232 Data Frame

The file *delay.h* provides functions for software delays, and *io.h* includes the I/O Port definitions. The delay functions need the clock frequency defined as `F_CPU` in Hertz in order to work. You must write 1 MHz, since this is the default clock speed of the ATmega8.

Next, you need to define some constants:

```
#define OUT_PORT    PORTB
#define OUT_PIN     PB1
#define OUT_DDR     DDRB

#define DOWN_TIME   78.125
#define PULSE_TIME  18.0288
```

The `#defines` with the prefix `OUT` determine which pin of which port is used for the output of the IRDA Signal. This is the pin the IR LED should be connected to.

Next, you need a function to send one bit. To make it more accurate, define it as *inline*. It checks whether the bit to send is 0 and, if so, creates a pulse by setting `PB1` high for a short period of time:

```
__inline__ void ir_send_bit(char bit){
    _delay_us(DOWN_TIME);
    if(!bit) {
        PORTB |= 0x01;
    }
    _delay_us(PULSE_TIME);
    PORTB &= !0x01;
}
```

Create an Infrared Peripheral Interface

For sending a byte to the PSP, you need to call the `ir_send_bit` function 10 times. First the start bit, then a loop that sends the byte and, at the end, the stop bit:

```
void ir_send_byte(char byte)
{
    char i = 8;
    ir_send_bit(0); // Start-Bit
    while(i)
    {
        ir_send_bit(byte&0x01);
        byte >>= 1;
        i--;
    }
    ir_send_bit(1); // Stop-Bit
}
```

Now that you're able to send bytes over infrared, the only thing left to do is to detect button changes and send them to the PSP. In each byte you send to the PSP, a byte is used to indicate the state of the gamepad controls (see Figure 4-12).

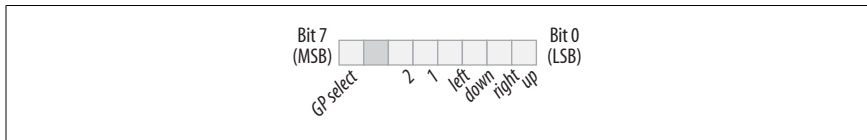


Figure 4-12. A byte containing button data

The first thing the main function does is to set the Data Direction Registers. All pins, except the first one of PortB, will be defined as input pins. After that, an infinite loop follows, which checks whether any button states have changed:

```
int main(void) {
    DDRB = 0x01;
    DDRD = 0x00;
    DDRC = 0x00;
    unsigned char gp1_old, gp2_old, gp1_new, gp2_new;
    while(1) {
        gp1_new = PINC;
        gp2_new = PIND;
        if(gp1_new != gp1_old){
            ir_send_byte(gp1_new);
        }
        if(gp2_new != gp2_old){
            /* set the GP select bit */
            ir_send_byte(gp2_new | 0b10000000);
        }
    }
}
```

```

        _delay_us(10);
        gp1_old = gp1_new;
        gp2_old = gp2_new;
    }
}

```

Writing the PSP Application

Now that you are able to send data to the PSP, you just have to write a little library that provides functions for decoding the data and making applications with support for external gamepads.

First, some declarations and imports are necessary:

```

#include <pspkernel.h>
#include <pspdebug.h>
#include <pspctrl.h>
#include <stdlib.h>

/* Define the module info section */
PSP_MODULE_INFO("IRDA Example", 0, 1, 1);

#define printf      pspDebugScreenPrintf

#define          GP_UP          0x0001
#define          GP_RIGHT      0x0002
#define          GP_DOWN      0x0004
#define          GP_LEFT      0x0008
#define          GP_1          0x0010
#define          GP_2          0x0020

SceUID          f;
unsigned char    odata = 0,
                ndata = 0,
                gamepad1,
                gamepad2;

```

Next, initialize the infrared port of the PSP. Write a function that does this work and that has to be called once the application is started. It opens a stream for reading data from the IR port:

```

int GP_Init()
{
    return (f = sceIoOpen("irda0:", PSP_O_RDWR, 0));
}

```

Now you just need a function that the application can use for checking the button states. As long as there is new data available, the loop assigns it to the variables `gamepad1` or `gamepad2`, depending on the state of the GP select bit:

```

void GP_Update()
{

```

```

do
{
    odata = ndata;
    sceIoRead(f, &ndata, 1);
    if(ndata & 0b10000000)
    {
        gamepad2 = ndata;
    } else {
        gamepad1 = ndata;
    }
} while( odata != ndata );
}

```

Having written all the necessary code for receiving data from the interface, you can test to see whether everything works with this simple test application. It will check for updates every frame and directly print the states on the screen:

```

void printStates(unsigned char data)
{
    printf("UP    : %u\n", (data&GP_UP) != 0);
    printf("DOWN  : %u\n", (data&GP_DOWN) != 0);
    printf("LEFT  : %u\n", (data&GP_LEFT) != 0);
    printf("RIGHT : %u\n", (data&GP_RIGHT) != 0);
    printf("1    : %u\n", (data&GP_1) != 0);
    printf("2    : %u\n", (data&GP_2) != 0);
}

int main(void)
{
    SceCtrlData pad;
    pspDebugScreenInit();
    printf("\nIRDA Test Application\n");
    GP_Init();
    while (1)
    {
        GP_Update();
        pspDebugScreenSetXY(0,3);

        printf("--- Gamepad 1 ---\n");
        printStates(gamepad1);

        printf("\n--- Gamepad 2 ---\n");
        printStates(gamepad2);

        sceCtrlReadBufferPositive(&pad, 1); // Refresh the PSP
                                           // Button States
        if(pad.Buttons & PSP_CTRL_CIRCLE) // If Circle is pressed...
        {
            sceKernelExitGame(); // ...exit the Application
        }
    }
}

```

This application provides a good base for creating your own gamepad-enabled applications or implementing gamepad support into existing applications, such as emulators.

Hacking the Hack

This is a very simple hack that shows only a small example of what is possible with the PSP, a microcontroller, and infrared. You can connect virtually everything to the microcontroller, send the data to the PSP, and do whatever you want with it inside of your own homebrew applications. Atmel provides useful application notes on their site that show you, for example, how to connect a keyboard to an AVR.

—*Thomas Novotny*