




Chapter 10

Interfacing with the Buttons



Console video game machines such as Game Boy Advance differ greatly from PCs in the realm of input. On a console, there is basically just the stock controller, with the occasional driving wheel or even more esoteric foot pad (for running or dancing games). The PC, however, uses a keyboard and mouse for input, with support for a whole slew of input devices, such as joysticks, gamepads, flight yokes with foot pedals, driving wheels. . . the list goes on. This chapter explains how to program the Game Boy Advance for detecting button input—the sole means of input on this system. You have already seen a sample of how to program the buttons on the GBA, from the ButtonTest program in Chapter 4. This chapter goes much further and explains every aspect of the button hardware on the GBA.

Here are the main topics of this chapter:

- The button architecture
- Detecting button input
- Creating a button handler

The Button Architecture

The Game Boy Advance features 10 buttons that may be used for input by any game. While there are standards by which GBA games have come to follow, such as using the Start and Select buttons exclusively for management-type functions (starting the game, pausing the game, bringing up a menu, and so on), the game designer or programmer is not so limited from a programming point of view, as all of the buttons are treated equally in code. What you do with the buttons is entirely up to you (within the limits of the game design, of course). Figure 10.1 shows the placement of the buttons on a GBA.



Figure 10.1

The Game Boy Advance features 10 distinct gameplay buttons.

Detecting Button Input

Like all other aspects of the GBA, the status of the buttons is placed in a specific location in memory that must be polled by your program. The memory address for the button status is at `0x04000130`. This is simply a number that you must define as a constant, like all other memory address constants in the GBA. If you want to memorize them all, be my guest! But it is always easier to write something down rather than memorize it, particularly when writing games. This is especially true if you get into cross-platform development. For instance, I do a lot of Pocket PC programming, and the Pocket PC devices use primarily the StrongARM processor, which is a close relative of the ARM7 processor used in the GBA! Of course, after using something for a long while, you come to recognize it and even begin to commit many things to memory as a matter of course.

The button status value at `0x04000130` is a 32-bit number. By the way, almost all memory locations are 32-bit simply because the GBA is a 32-bit machine. Since there are 10 buttons for input, and typically only 8 are used for gameplay, it doesn't take much to identify a button press on the part of the hardware—it simply sets a bit on or off based on the status of the button. Now, how about creating a pointer to this memory location so it's easy to check for button input in a game? On the GBA, a 32-bit number is called an *int* or *unsigned*

int. (Remember that a short is 16 bits.) Therefore, to create a pointer to this memory address, you would do this:

```
unsigned int *BUTTONS = (unsigned int *)0x04000130;
```

Throughout this book, I have assumed that you already know C, but just for reference, the `(unsigned int *)` is a typecast that ensures the `*BUTTONS` pointer grabs the whole 32 bits of the memory address. As a precaution, this pointer should be declared with the *volatile* keyword, to tell the compiler that it is a memory address that is changed by another process (not by the program itself):

```
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
```

There you have it—a new pointer to the memory address where button values are stored by the GBA.

Searching for Buttons

Now, suppose you don't know what the various button values are and just want to check to see if any button has been pressed at all. For instance, suppose you are displaying some sort of video on the screen as an introduction to your game and want to just check for any button press to halt the video and go straight to the start screen of the game. You might just check to see if the memory address pointed to by `BUTTONS` contains anything other than 0. Let's write a short code snippet to do just that:

```
while(1)
{
    if(*BUTTONS)
        // yes, a button was pressed!
    else
        // nothing yet
}
```

All this code snippet does is check for any value other than 0 (which is considered a false Boolean value by the `if` statement, as you well know). The only problem with this code is that it always returns a true value, whether a button is being pressed or not! Why is that, do you suppose? I scratched my head about it myself for a while, until I came up with an idea. How about writing a program that performs a loop and scans for possible numbers and then run the program, press a button, and see what number comes up. That's not a bad idea, actually. Let's write a short program to do that! I'll use the `Print` function and the usual `font.h` file to display messages on the screen.

The ScanButtons Program

Now, let's create the ScanButtons project. Fire up Visual HAM and create a new project by opening the File menu and selecting New, New Project. As I've gone over this many times already, I'll assume you are proficient with Visual HAM and no longer need a tutorial on creating the project and so on. Name the project ScanButtons, and then add the font.h file to the project. In case you have forgotten, you can add a file by right-clicking on the project workspace, either the Source Files or Header Files section, selecting Add File(s) to Folder, and then searching for the font.h file from \Sources\Chapter07. I prefer to copy the font.h file to my new project folder, which makes it easier to add the file to the project.

The point is that you don't have to type in the font code again, so whatever works for you, go with it.

Visual HAM automatically adds the main.c file to the project, and this file is where you should type in the

source code for the program. If you want, you may copy the DrawPixel3, Print, PrintT, and DrawChar functions from earlier programs rather than typing them in again. I could put these common functions into a separate code file, but that always leaves room for error for those who are not adept at managing projects in C, including header files and so on. It's just easier to include short and common functions again.

The `#include <stdio.h>` tells the compiler to include the standard ANSI C input/output library, which includes the uber-cool `sprintf` function.

```
////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ScanButtons Project
// main.c source code file
////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include <stdio.h>
#include "font.h"

//define boolean
#define bool short
#define true 1
```



```

#define false 0

//define some useful colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000

//background 2
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;

//declare some function prototypes
void SetMode(int mode);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);
void PrintT(int, int, char *, unsigned short);
void DrawChar(int, int, char, unsigned short, bool);

////////////////////////////////////
// Function: main()
// Entry point for the program

```

```

////////////////////////////////////
int main()
{
    char str[20];

    SetMode(3);
    Print(1, 1, "SCANBUTTONS PROGRAM", WHITE);
    Print(1, 30, "SCANNING...", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        if (*BUTTONS)
        {
            sprintf(str, "BUTTON CODE = %i  ", (unsigned int)*BUTTONS);
            Print(10, 40, str, BLUE);
        }
    }

    return 0;
}

void SetMode(int mode)
{
    REG_DISPCNT = (mode | BG2_ENABLE);
}

////////////////////////////////////
// Function: DrawPixel3
// Draw a pixel on the mode 3 video buffer
////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{

```

```

        videoBuffer[y * 240 + x] = c;
    }

    ////////////////////////////////////////////////////
    // Function: Print
    // Prints a string using a hard-coded font, must be all caps
    ////////////////////////////////////////////////////
    void Print(int left, int top, char *str, unsigned short color)
    {
        int pos = 0;
        while (*str)
        {
            DrawChar(left + pos, top, *str++, color, false);
            pos += 8;
        }
    }

    ////////////////////////////////////////////////////
    // Function: PrintT
    // Prints a string with transparency
    ////////////////////////////////////////////////////
    void PrintT(int left, int top, char *str, unsigned short color)
    {
        int pos = 0;
        while (*str)
        {
            DrawChar(left + pos, top, *str++, color, true);
            pos += 8;
        }
    }

    ////////////////////////////////////////////////////
    // Function: DrawChar
    // Draws a single character from a hard-coded font

```



```

////////////////////////////////////
void DrawChar(int left, int top, char letter,
              unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}

```

When you finish typing in the code for the ScanButtons program, hit F7 to run the program. If there are no typos in the program, you should see output that looks like Figure 10.2. A glance at the program shows a problem right away. The default value without pressing any buttons is 1,023.

Making Sense of Button Values

What this means is that all the bits are set to 1 by default and are individually set to 0 when a button is pressed. The problem is that the GBA sets status bits based on position rather than just storing an arbitrary value! That's right. So while you might expect to see buttons Start=1, Select=2, and perhaps A=3, B=4, and so on, that is not what is really going on. If you were to just check for a specific value, you would be able to detect only one button press at a time, which is definitely not the way this is supposed to work. Instead of looking at *BUTTONS for a specific value, we need to look at the bitmask. The GBA sets an incremental bit based on the button press, and it looks like Figure 10.3.

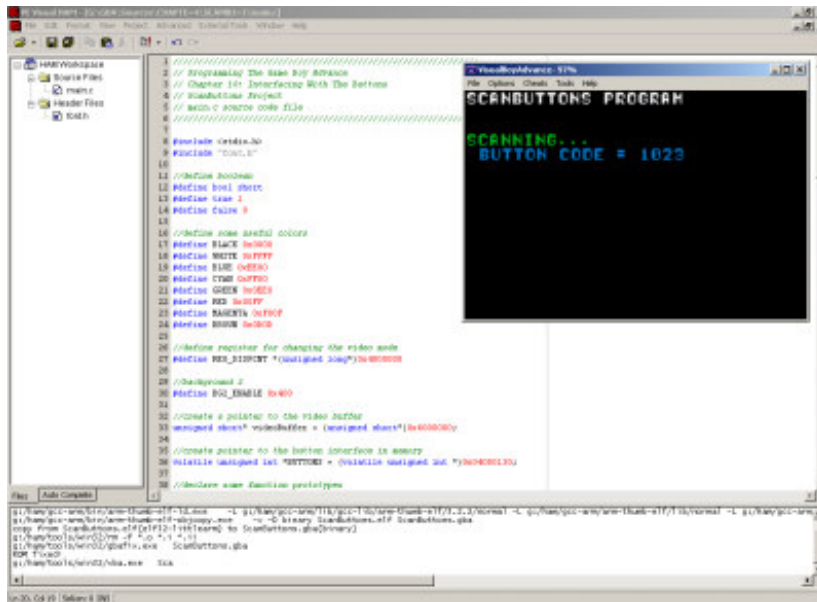


Figure 10.2

The ScanButtons program displays the button input status value.

Memory Address 0x0400130

BITS	VALUES	BUTTONS
0	1	A
1	2	B
2	4	SELECT
3	8	START
4	16	RIGHT
5	32	LEFT
6	64	UP
7	128	DOWN
8	256	R
9	512	L
10	} NOT USED	
·		
·		
·		
·		
·		
·		
·		
·		
·		
31		

Figure 10.3

The bit values of the button status memory location.

This calls for a little change to the program, because the button values will need to be AND'ed to this 1,023 in order to determine which button was pressed (and it's necessary in order to detect multiple button presses).

Correctly Identifying Buttons

Let's modify the program to fix the problem, so that it only reports a value if one or more buttons are actually pressed. If you want to just load this project off the CD-ROM instead of making the changes by hand, the corrected version is on the CD-ROM under \Sources\Chapter10\ScanButtons2. The only change needed involves the while(1) loop. Here is the new version:

```
// continuous loop
while(1)
{
    //check for button presses
    for (n=1; n < 1000; n++)
    {
        if (!((*BUTTONS) & n))
        {
            sprintf(str, "BUTTON CODE = %i  ", n);
            Print(10, 40, str, BLUE);
            break;
        }
    }
}
```

The first thing you'll likely have noticed is the for loop that goes from 1 to 1,000. That is just an arbitrary number that will accommodate all 10 buttons, because I don't know immediately which bits the GBA uses for each button. I know that it will likely only go up to 512, but this is knowledge gained after the fact. From a fact-seeking point of view, one might not necessarily know this in advance (no pun intended). Now, hurry up and run the program! The output is shown in Figure 10.4.

I was pressing the UP button in the screen shot shown, which indicates a value of 64. Using this little button-scanning program, you can come up with a list of values for each button! Here is the table I came up with after noting the value of each button reported by ScanButtons2 (see Table 10.1).

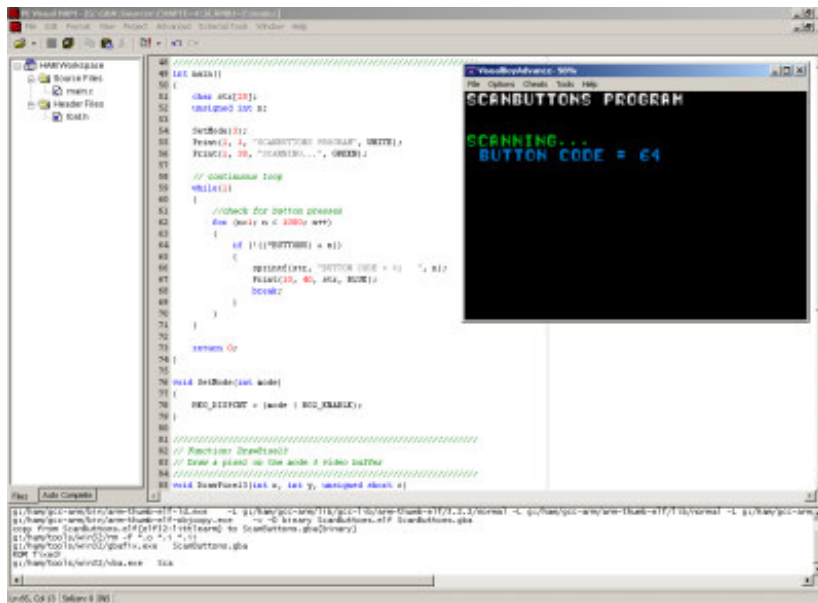


Figure 10.4

The modified ScanButtons2 program displays a value only if a button is pressed.

This is a really good lesson in dealing with low-level hardware interfaces. Being able to figure out something on your own is worth a thousand tutorials on the Web! The reasoning being that these concepts apply to other consoles and computer hardware, not just the GBA.

Table 10.1 Button Input Values

Button	Value
A	1
B	2
SELECT	4
START	8
RIGHT	16
LEFT	32
UP	64
DOWN	128
R	256
L	512

Displaying Button Scan Codes

As you have likely spent some time writing programs for your PC, you probably know all about keyboard scan codes. The GBA's buttons have codes too, although they are not exactly like the ASCII codes on a PC. If you have done any low-level keyboard programming though, where each key is numbered in sequence, it does start to resemble the buttons on a GBA a little more.

Now let's modify the program again so that it shows the name of the button being pressed. Again, this just involves changing the main while loop—but I've used a new function called `strcpy` in order to display the button name, so you will need to add another `#include` statement to the top of the program as follows:

```
#include <string.h>
```

You will also need to add a new variable to the program, called `name`:

```
char name[10];
```

I have saved this modified version as `ScanButtons3`. Here is the new version of the main function:

```
int main()
{
    char str[20];
    unsigned int n;
    char name[10];

    SetMode(3);
    Print(1, 1, "SCANBUTTONS PROGRAM", WHITE);
    Print(1, 30, "SCANNING...", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        for (n=1; n < 1000; n++)
        {
            if (!((*BUTTONS) & n))
            {
                sprintf(str, "BUTTON CODE = %i ", n);
```



```

Print(10, 40, str, BLUE);

//figure out the button name
switch(n)
{
    case 1: strcpy(name, "A"); break;
    case 2: strcpy(name, "B"); break;
    case 4: strcpy(name, "SELECT"); break;
    case 8: strcpy(name, "START"); break;
    case 16: strcpy(name, "RIGHT"); break;
    case 32: strcpy(name, "LEFT"); break;
    case 64: strcpy(name, "UP"); break;
    case 128: strcpy(name, "DOWN"); break;
    case 256: strcpy(name, "R"); break;
    case 512: strcpy(name, "L"); break;
    default: strcpy(name, ""); break;
}
sprintf(str, "BUTTON NAME = %s", name);
Print(10, 50, str, RED);
break;
} //if
} //for
} //while
} //main

```

The output from the ScanButtons3 program is shown in Figure 10.5.

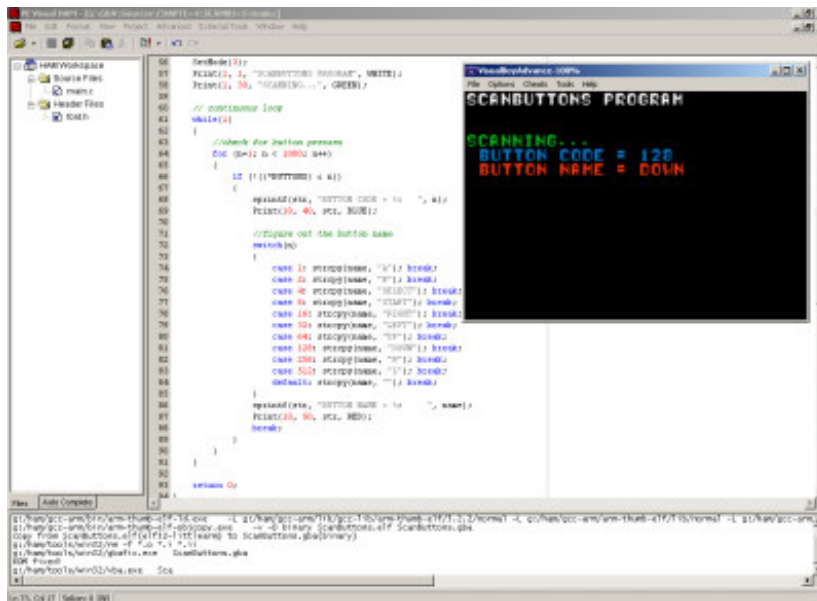


Figure 10.5
The modified ScanButtons3 program now displays the button name.

Getting the Hang of Button Input

Now that you know the value of each button, we can put together a list of `#define` statements to make it easier to use the buttons in your GBA programs. Here is the list that I came up with:

```
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512
```

This list of button constants, along with the `*BUTTONS` pointer, allows you to make robust use of button input in all future GBA programs. Since the code is so meager, it's not a big deal to copy the code from one project to another (in other words, there is no real need for a separate `.lib`). Some folks get all worked up about libraries of code and making everything reusable and so on, but sometimes it's just preferable to see the code directly. Once you have a large collection of GBA code that you use frequently, it is then a good idea to lib it up!

Creating a Button Handler

A button handler takes the actual button code and abstracts it to a second level in order to make the source code more intuitive. Since button input is a very low-level aspect of programming the GBA, it's helpful to move the actual memory reading code into a function and store the results of the buttons in an array. Of course, this is not at all necessary and only reflects my own coding style. If you prefer to put the button code directly in your main loop, feel free to do so. The main benefit for polling all the buttons at the same time is that you are more likely to lose a button if there is a lot of code between each poll. For instance, if your program detects the A button has been pressed, and does some processing based on that input, but then checks to look for a simultaneous B button press, that button may no longer register. By polling the state of all buttons in one quick interval, you may then use up the entire vblank period to handle the input of the buttons without worrying about losing a button.

For instance, if you use the A button to fire a weapon and the D-pad to move a ship on the screen, then your weapon firing code may likely take up so many cycles that you miss the D-pad input. Just think of it in logical terms. It makes more sense to segregate your main loop code into easily identifiable sections, having button input separate from graphics output, sound generation, and game logic. Once buttons are polled at the start of the game loop, it is then an easy matter later on in the loop to check the status of each button as needed.

Handling Multiple Buttons

To abstract the button handler from the low-level button code, an array is needed to store a simple Boolean true or false value (which equates to 1 or 0, respectively):

```
bool buttons[10];
```

Using the button array is simply a matter of polling all the buttons at once and storing the result of each poll in an element of the array as follows. Note that the assignment causes C to evaluate the expression for a true or false. The result is negated because the result of the AND operation is a 0 when the button is pressed, which is the opposite of what we want—a button press should equate to 1.

```
buttons[0] = !((*BUTTONS) & BUTTON_A);  
buttons[1] = !((*BUTTONS) & BUTTON_B);  
buttons[2] = !((*BUTTONS) & BUTTON_LEFT);  
buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);  
buttons[4] = !((*BUTTONS) & BUTTON_UP);  
buttons[5] = !((*BUTTONS) & BUTTON_DOWN);  
buttons[6] = !((*BUTTONS) & BUTTON_START);
```

```
buttons[7] = !((*BUTTONS) & BUTTON_SELECT);
buttons[8] = !((*BUTTONS) & BUTTON_L);
buttons[9] = !((*BUTTONS) & BUTTON_R);
```

Polling the buttons is a very fast process, so there really is no need to strip out buttons that you don't use, because this code could end up in a library eventually. If you plan to have a single code listing with this function in your main.c file, for instance, then you may want to comment out any buttons that you don't plan to use in the game. While the timing is negligible, every clock cycle does help. When it comes time to check for the button presses (presumably, that would be later on in the game loop), you can use a function such as the following to return a true or false value based on the button that is passed to the function:

```
bool Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons[1];
        case BUTTON_LEFT: return buttons[2];
        case BUTTON_RIGHT: return buttons[3];
        case BUTTON_UP: return buttons[4];
        case BUTTON_DOWN: return buttons[5];
        case BUTTON_START: return buttons[6];
        case BUTTON_SELECT: return buttons[7];
        case BUTTON_L: return buttons[8];
        case BUTTON_R: return buttons[9];
    }
}
```

The ButtonHandler Program

As you have seen, the button handler need not be complicated unless you want to detect button *releases* separately from button *presses*. Then it requires a little more thought and will most likely require another array to keep track of which buttons have been pressed. I have never needed to check for button releases, because when it comes to GBA coding, all that really matters is responding to a button press.

I have written a complete program to demonstrate how to write your own general-purpose button handler. The project is called ButtonHandler and is located on the CD-ROM under

\Source\Chapter10\ButtonHandler. As usual, this project requires the font.h file, so if you are creating this project from scratch and typing it in, be sure to add the font.h file to the folder for this project. Note that adding a file to the actual workspace in Visual HAM is not necessary—and actually, such files are not automatically compiled. Adding them to the file list simply makes it easier to edit the files in the project. The compiler only looks at header files that are included in a main source file with the #include directive. So it is sufficient to simply copy the font.h file to the project folder for each program that needs to display output. When you aren't using the built-in Hamlib functionality in your program, a font subsystem like this is essential, as you have seen over the last few chapters.

Now, here is the source code for the ButtonHandler project. Type it into the main.c file, and feel free to copy duplicate functions from other programs you have already typed in (such as DrawPixel3, Print, and so on).

```
////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ButtonHandler Project
// main.c source code file
////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include "font.h"

//define boolean
#define bool short
#define true 1
#define false 0

//declare some function prototypes
void CheckButtons();
bool Pressed(int);
void SetMode(int);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);
void DrawChar(int, int, char, unsigned short, bool);
```



```
//define some colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define the buttons
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512

//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000

//use background 2
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//create pointer to the button interface in memory
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
```

```

//keep track of the status of each button
bool buttons[10];

////////////////////////////////////
// Function: main()
// Entry point for the program
////////////////////////////////////
int main()
{
    SetMode(3);
    Print(1, 1, "BUTTONHANDLER PROGRAM", WHITE);
    Print(1, 30, "PRESS A BUTTON:", GREEN);

    // continuous loop
    while(1)
    {
        //check for button presses
        CheckButtons();

        //display the status of each button
        if (Pressed(BUTTON_A))
            Print(10, 40, "A PRESSED", BLUE);
        else
            Print(10, 40, "          ", 0);
        if (Pressed(BUTTON_B))
            Print(10, 50, "B PRESSED", BLUE);
        else
            Print(10, 50, "          ", 0);
        if (Pressed(BUTTON_SELECT))
            Print(10, 60, "SELECT PRESSED", BLUE);
        else
            Print(10, 60, "          ", 0);
        if (Pressed(BUTTON_START))

```

```

        Print(10, 70, "START PRESSED", BLUE);
    else
        Print(10, 70, "          ", 0);
    if (Pressed(BUTTON_LEFT))
        Print(10, 80, "LEFT PRESSED", BLUE);
    else
        Print(10, 80, "          ", 0);
    if (Pressed(BUTTON_RIGHT))
        Print(10, 90, "RIGHT PRESSED", BLUE);
    else
        Print(10, 90, "          ", 0);
    if (Pressed(BUTTON_UP))
        Print(10, 100, "UP PRESSED", BLUE);
    else
        Print(10, 100, "          ", 0);
    if (Pressed(BUTTON_DOWN))
        Print(10, 110, "DOWN PRESSED", BLUE);
    else
        Print(10, 110, "          ", 0);
    if (Pressed(BUTTON_R))
        Print(10, 120, "R PRESSED", BLUE);
    else
        Print(10, 120, "          ", 0);
    if (Pressed(BUTTON_L))
        Print(10, 130, "L PRESSED", BLUE);
    else
        Print(10, 130, "          ", 0);
}
return 0;
}

////////////////////////////////////
// Function: CheckButtons
// Polls the status of all the buttons

```

```
////////////////////////////////////
```

```
void CheckButtons()
{
    //store the status of the buttons in an array
    buttons[0] = !((*BUTTONS) & BUTTON_A);
    buttons[1] = !((*BUTTONS) & BUTTON_B);
    buttons[2] = !((*BUTTONS) & BUTTON_LEFT);
    buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);
    buttons[4] = !((*BUTTONS) & BUTTON_UP);
    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);
    buttons[6] = !((*BUTTONS) & BUTTON_START);
    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);
    buttons[8] = !((*BUTTONS) & BUTTON_L);
    buttons[9] = !((*BUTTONS) & BUTTON_R);
}
```

```
////////////////////////////////////
```

```
// Function: Pressed
// Returns the status of a button
////////////////////////////////////
```

```
bool Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons[1];
        case BUTTON_LEFT: return buttons[2];
        case BUTTON_RIGHT: return buttons[3];
        case BUTTON_UP: return buttons[4];
        case BUTTON_DOWN: return buttons[5];
        case BUTTON_START: return buttons[6];
        case BUTTON_SELECT: return buttons[7];
        case BUTTON_L: return buttons[8];
        case BUTTON_R: return buttons[9];
    }
}
```

```

    }
    return false;
}

////////////////////////////////////
// Function: SetMode
// Changes the video mode
////////////////////////////////////
void SetMode(int mode)
{
    REG_DISPCNT = (mode | BG2_ENABLE);
}

////////////////////////////////////
// Function: DrawPixel3
// Draw a pixel on the mode 3 video buffer
////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}

////////////////////////////////////
// Function: Print
// Prints a string using a hard-coded font, must be all caps
////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color, false);
        pos += 8;
    }
}

```



```

}

////////////////////////////////////
// Function: DrawChar
// Draws a single character from a hard-coded font
////////////////////////////////////
void DrawChar(int left, int top, char letter,
              unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}

```

Now if you run the ButtonHandler program, you should see output that looks like the screen shot in Figure 10.6.

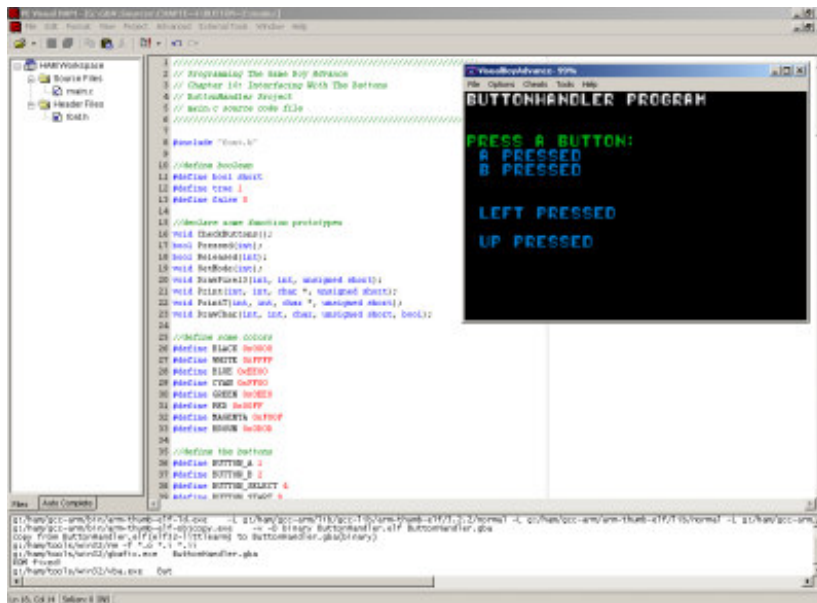


Figure 10.6

The ButtonHandler program shows how to write a reusable button handler.

Detecting Button Combos

Combos are a big aspect of many games, primarily fighting games like *Soul Calibur*, *Dead or Alive 3*, *Street Fighter* series, and so on. A combo is a series of button pushes in a specific order that gives the player's character some kind of power-up or super attack. For example, the infamous "sho'riuken!" dragon punch from *Street Fighter II*'s Ken character is almost infamous in videogame legend. Another popular combo for Ken (and Ryu) was the fireball, which required a combo of Down, Down+Right, Right, A to unleash a powerful uppercut (and the combo is reversed depending on the direction that the player is facing). What this means is that you must press D, DR, R, A in rapid succession to unleash the combo attack.

In a real game, timing is a critical issue, as you learned back in Chapter 8, "Using Interrupts and Timers." But to keep this example program short, I have cheated a little and just put sort of a hard-coded delay into the program to slow it down (interrupt and timer code is somewhat lengthy, as you'll recall). Button input occurs so quickly that it's impossible to detect combos without a delay; as soon as you touch a key, several input events fire off! The problem with this, when trying to detect a combo, is that the first button is registered several times on first press. So I'll just use a small loop to slow the program a bit. Following is a program called ComboTest, which demonstrates how to put combo support into your game.

In order to detect a combo, it is necessary to first identify how many buttons are being pressed and only start counting combinations if a single button is being pressed. To do this, I wrote a function called ButtonsPressed:

```

int ButtonsPressed()
{

```

```

    int n;
    int total = 0;
    for (n=0; n < 10; n++)
        total += buttons[n];
    return total;
}

```

Here's the source code for the complete ComboTest program. Just type this into the main.c file for a new project, and be sure to copy the font.h file to the project folder as usual. Again, there's some duplicated source code here, so feel free to copy and paste as needed.

```

////////////////////////////////////
// Programming The Game Boy Advance
// Chapter 10: Interfacing With The Buttons
// ComboTest Project
// main.c source code file
////////////////////////////////////

#define MULTIBOOT int __gba_multiboot;
MULTIBOOT

#include <stdio.h>
#include "font.h"

//define boolean
#define bool short
#define true 1
#define false 0

//declare some function prototypes
int ButtonsPressed();
void CheckButtons();
bool Pressed(int);
void SetMode(int);
void DrawPixel3(int, int, unsigned short);
void Print(int, int, char *, unsigned short);

```

```

void DrawChar(int, int, char, unsigned short, bool);

//define some colors
#define BLACK 0x0000
#define WHITE 0xFFFF
#define BLUE 0xEE00
#define CYAN 0xFF00
#define GREEN 0x0EE0
#define RED 0x00FF
#define MAGENTA 0xF00F
#define BROWN 0x0D0D

//define the buttons
#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512

//define register for changing the video mode
#define REG_DISPCNT *(unsigned long*)0x4000000

//use background 2
#define BG2_ENABLE 0x400

//create a pointer to the video buffer
unsigned short* videoBuffer = (unsigned short*)0x6000000;

//create pointer to the button interface in memory

```

```
volatile unsigned int *BUTTONS = (volatile unsigned int *)0x04000130;
```

```
//keep track of the status of each button
```

```
bool buttons[10];
```

```
////////////////////////////////////////////////////////////////
```

```
// Function: main()
```

```
// Entry point for the program
```

```
////////////////////////////////////////////////////////////////
```

```
int main()
```

```
{
```

```
    int counter = 0;
```

```
    char str[20];
```

```
    int delay;
```

```
    int combo[10] = {BUTTON_UP,BUTTON_UP,BUTTON_DOWN,BUTTON_DOWN,
```

```
                    BUTTON_LEFT,BUTTON_RIGHT,BUTTON_LEFT,BUTTON_RIGHT,
```

```
                    BUTTON_B,BUTTON_A};
```

```
    SetMode(3);
```

```
    Print(1, 1, "COMBOTEST PROGRAM", WHITE);
```

```
    Print(1, 30, "PRESS U,U,D,D,L,R,L,R,B,A:", GREEN);
```

```
    // continuous loop
```

```
    while(1)
```

```
    {
```

```
        //check for button presses
```

```
        CheckButtons();
```

```
        if (ButtonsPressed() == 1)
```

```
        {
```

```
            sprintf(str, "COUNTER = %i ", counter + 1);
```

```
            Print(10, 50, str, BLUE);
```

```
            if (Pressed(combo[counter]))
```



```

        {
            counter++;

            if (counter == 10)
            {
                Print(10, 70, "CHEAT MODE ACTIVATED!", RED);
                counter = 0;
            }

            //slow down the program
            delay = 500000;
            while (delay--);
        }
    else
        counter = 0;
}
}
return 0;
}

```

```

////////////////////////////////////
// Function: ButtonsPressed
// Returns the number of buttons being pressed
////////////////////////////////////
int ButtonsPressed()
{
    int n;
    int total = 0;
    for (n=0; n < 10; n++)
        total += buttons[n];
    return total;
}

```

```

////////////////////////////////////

```

```

// Function: CheckButtons
// Polls the status of all the buttons
////////////////////////////////////
void CheckButtons()
{
    //store the status of the buttons in an array
    buttons[0] = !((*BUTTONS) & BUTTON_A);
    buttons[1] = !((*BUTTONS) & BUTTON_B);
    buttons[2] = !((*BUTTONS) & BUTTON_LEFT);
    buttons[3] = !((*BUTTONS) & BUTTON_RIGHT);
    buttons[4] = !((*BUTTONS) & BUTTON_UP);
    buttons[5] = !((*BUTTONS) & BUTTON_DOWN);
    buttons[6] = !((*BUTTONS) & BUTTON_START);
    buttons[7] = !((*BUTTONS) & BUTTON_SELECT);
    buttons[8] = !((*BUTTONS) & BUTTON_L);
    buttons[9] = !((*BUTTONS) & BUTTON_R);
}

////////////////////////////////////
// Function: Pressed
// Returns the status of a button
////////////////////////////////////
bool Pressed(int button)
{
    switch(button)
    {
        case BUTTON_A: return buttons[0];
        case BUTTON_B: return buttons[1];
        case BUTTON_LEFT: return buttons[2];
        case BUTTON_RIGHT: return buttons[3];
        case BUTTON_UP: return buttons[4];
        case BUTTON_DOWN: return buttons[5];
        case BUTTON_START: return buttons[6];
        case BUTTON_SELECT: return buttons[7];
    }
}

```

```

        case BUTTON_L: return buttons[8];
        case BUTTON_R: return buttons[9];
    }
    return false;
}

////////////////////////////////////
// Function: SetMode
// Changes the video mode
////////////////////////////////////
void SetMode(int mode)
{
    REG_DISPCNT = (mode | BG2_ENABLE);
}

////////////////////////////////////
// Function: DrawPixel3
// Draw a pixel on the mode 3 video buffer
////////////////////////////////////
void DrawPixel3(int x, int y, unsigned short c)
{
    videoBuffer[y * 240 + x] = c;
}

////////////////////////////////////
// Function: Print
// Prints a string using a hard-coded font, must be all caps
////////////////////////////////////
void Print(int left, int top, char *str, unsigned short color)
{
    int pos = 0;
    while (*str)
    {
        DrawChar(left + pos, top, *str++, color, false);
    }
}

```

```

        pos += 8;
    }
}

////////////////////////////////////
// Function: DrawChar
// Draws a single character from a hard-coded font
////////////////////////////////////
void DrawChar(int left, int top, char letter, unsigned short color, bool trans)
{
    int x, y;
    int draw;

    for(y = 0; y < 8; y++)
        for (x = 0; x < 8; x++)
        {
            //grab a pixel from the font char
            draw = font[(letter-32) * 64 + y * 8 + x];
            //if pixel = 1, then draw it
            if (draw)
                DrawPixel3(left + x, top + y, color);
            else
                //fill in black pixel if no transparency
                if (!trans)
                    DrawPixel3(left + x, top + y, BLACK);
        }
}
}

```

The output from the ComboTest program is shown in Figure 10.7. In case you were wondering, the combo used in this program was inspired by the infamous NES version of *Contra*: U-U-D-D-L-R-L-R-B-A-B-A. I think the game actually required you to hit Select or Start at the end of the combo, which became known as the "Konami code" over the years.

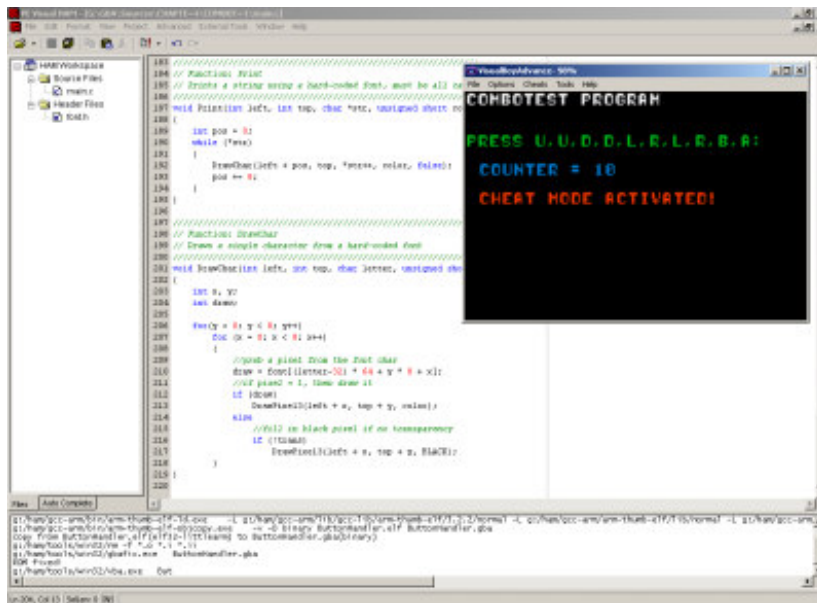


Figure 10.7

The ComboTest program shows how to use button combos.

Summary

This chapter has covered the absolutely critical subject of button input, and it wasn't too rough of a ride after all. Adding a button handler to a GBA program is probably even more important than the graphics or sound—in fact, all three are critical for a good game, so there really is no comparison for priority here. As you saw in this chapter, there's a lot you can do with a button handler, and I have only touched on the major points, such as detecting multiple button presses, displaying the button codes, and even handling combos.

Challenges

The following challenges will help to reinforce the material you have learned in this chapter. .

Challenge 1: The ScanButtons3 program does a pretty good job of demonstrating button input, but it would be more interesting with more color. Modify the program so that a different color is used for the name of each button.

Challenge 2: The ButtonHandler program displays button press events on separate lines. Modify the program so it keeps a counter of each button pressed and displays the counter value with each button message on the screen.

Challenge 3: The ComboTest program currently only supports a single combo. Modify the program so that it can support many different combos and display the name of the combo when activated.

Chapter Quiz

The following quiz will help to further reinforce the information covered in this chapter. The quiz consists of 10 multiple-choice questions with up to four possible answers. The key to the quiz may be found in the appendix.

1. How many buttons does the GBA have?
 - A. 6
 - B. 8
 - C. 10
 - D. 12

2. What is the memory address used to check the status of button input?
 - A. 0x04000130
 - B. 0x05020100
 - C. 0x01000900
 - D. 0x60000000

3. What is the type of processor used in the GBA?
 - A. MIPS
 - B. StrongARM
 - C. SH3
 - D. ARM7

4. True/False: Can the GBA access 32 bits of memory at a time?
 - A. True
 - B. False

5. What method is used to check for button input on the GBA?
 - A. Hardware interrupt
 - B. Callback function
 - C. Memory polling



D. Meditation

6. What is the largest value returned by any button press event?
 - A. 128
 - B. 256
 - C. 512
 - D. 1,024

7. How many total bits (out of 32) are used by the GBA to report button input status?
 - A. 8
 - B. 10
 - C. 16
 - D. 6

8. Which video mode is used by the sample programs in this chapter?
 - A. Mode 0
 - B. Mode 2
 - C. Mode 5
 - D. Mode 3

9. What bitwise operation is used to evaluate the status of a single button?
 - A. AND
 - B. OR
 - C. NOT
 - D. XOR

10. What bit value is set by the GBA to identify that a button press has occurred?
 - A. 1
 - B. 0
 - C. -1
 - D. 2