Revolution SDK

Nintendo GameCube™ Controller Library (PAD)

Version 1.04

The contents in this document are highly confidential and should be handled accordingly.

RVL-06-0043-001-E
Released: March 27, 2009

Contents

Code Examples

Tables

Revision History

Version	Date Revised	Item	Description
1.04	2008/08/26	2.9	Added description of specifications that do not reset the origin correctly under special conditions.
1.03	2006/09/07	2.7	Added description of the newly added clamp API.
1.02	2006/08/30	2.6 2.7 2.9 3.4	Added notes regarding handling function return values. Added usable value input range and adjustments. Added “Hardware Origin Reset.” Added “Controller Rumble Feature Availability Detection.”
1.01	2006/04/18	2.4	Added description of error codes.
1.00	2006/03/01	-	First release by Nintendo of America, Inc.

1 Introduction

Revolution provides the ability to use Nintendo GameCube Controllers.

This document describes the Nintendo GameCube Controller features and API for the Revolution system. The Nintendo GameCube standard Controller supports the following features:

- Two analog sticks (the Control Stick and the C Stick)
- Two combined analog/digital triggers (the L Button and the R Button). Users will feel a click when the L Button or R Button is fully depressed. Pressing these buttons further activates a digital switch.
- One directional pad (the +Control Pad)
- Six digital buttons (the A Button, B Button, X Button, Y Button, Z Button, and START/PAUSE)
- A built-in Rumble Motor (not available for WaveBird)

The Revolution console has four controller ports. From the hardware perspective, it is not necessary for players to connect Controllers to the controller ports from left to the right. Moreover, players can connect or disconnect Controllers while the Revolution console is turned on.

In Revolution mode, up to eight Controllers can be used simultaneously when both Revolution standard controllers and Nintendo GameCube Controllers are used together.

Important: It is not recommended that the UI tool version 3 (the standard Wii controller type that is inserted into the controller port) and the Nintendo GameCube Controller be used together. Although the WPAD library for the UI tool version 3 can obtain data from both the Nintendo GameCube Controller as well as UI tool version 3, this functionality is limited (see the WPAD library function reference for more information). Avoid using the PAD library in conjunction with the WPAD library for the UI tool version 3.

The Revolution hardware samples the status of every attached Controller automatically at the rate specified by the program. The Video Interface controls the timing of Controller sampling (see the *Video Interface Library (VI)* section in the *Graphics Programmer's Guide* for details). Controller status, stored in the serial interface registers, can be read by the CPU at any time. The Controller library (PAD) provides a set of functions through which game applications can communicate with the Revolution standard controllers.

2 (PAD) API

This chapter describes the constants, data structures, and functions of the basic Controller (PAD) API. (The rumble motor control functions are described in the next chapter.) The following header file defines the PAD API:

Code 2–1 PAD API Header File

```
#include <revolution/pad.h>
```

2.1 PAD_MAX_CONTROLLERS

PAD_MAX_CONTROLLERS identifies the maximum number of Controllers that can be plugged into Revolution in Revolution mode.

Code 2–2 PAD_MAX_CONTROLLERS

```
#define PAD_MAX_CONTROLLERS     4
```

2.2 PADStatus

Code 2–3 PADStatus

```
typedef struct PADStatus
{
    u16 button;                 // Or-ed PAD_BUTTON_* or PAD_TRIGGER_* bits
    s8  stickX;                 // -128 <= stickX       <= 127
    s8  stickY;                 // -128 <= stickY       <= 127
    s8  substickX;              // -128 <= substickX    <= 127
    s8  substickY;              // -128 <= substickY    <= 127
    u8  triggerLeft;            //    0 <= triggerLeft  <= 255
    u8  triggerRight;           //    0 <= triggerRight <= 255
    u8  analogA;                //    0 <= analogA      <= 255
    u8  analogB;                //    0 <= analogB      <= 255
    s8  err;                    // one of PAD_ERR_* number
} PADStatus;
```

The PADStatus data structure represents the status of a Controller and takes the following variables:

Table 2–1 PADStatus Variables

Variable	Meaning
button	If any button is pressed, the corresponding bit is set to 1.
stickX	Movement data given in terms of the *x*-axis of the main analog stick (the Control Stick).
stickY	Movement data given in terms of the *y*-axis of the main analog stick (the Control Stick).
substickX	Movement data given in terms of the *x*-axis of the second analog stick (the C Stick).
substickY	Movement data given in terms of the *y*-axis of the second analog stick (the C Stick).
triggerLeft	Movement data of the L Button.
triggerRight	Movement data of the R Button.
analogA	Analog input of the A Button. (See note.)
analogB	Analog input of the B Button. (See note.)
err	Controller error code. This holds PAD_ERR_NONE if the pad status is valid; otherwise, it holds another error code.

Note: The Nintendo GameCube Controller does not support analog input values (AnalogA/B) from the A Button or B Button. Furthermore, the Revolution Controller Library does not support analog input from the A Button or B Button.

2.3 Controller Button Bits

The Controller buttons are identified by the following bits:

Table 2–2 Controller Buttons

Buttons	Bits
PAD_BUTTON_LEFT	0x0001
PAD_BUTTON_RIGHT	0x0002
PAD_BUTTON_DOWN	0x0004
PAD_BUTTON_UP	0x0008
PAD_TRIGGER_Z	0x0010
PAD_TRIGGER_R	0x0020

Table 2–2 Controller Buttons

Buttons	Bits
PAD_TRIGGER_L	0x0040
PAD_BUTTON_A	0x0100
PAD_BUTTON_B	0x0200
PAD_BUTTON_X	0x0400
PAD_BUTTON_Y	0x0800
PAD_BUTTON_START	0x1000

2.4 Pad Error Codes

The *err* member of the PADStatus data structure can hold one of the following error codes:

Table 2–3 Error Codes

Definition Name	Code	Description
PAD_ERR_NONE	0	PADStatus contains a valid Controller status.
PAD_ERR_NO_CONTROLLER	-1	No Controller is inserted in the Controller Socket.[a]
PAD_ERR_NOT_READY	-2	The Controller Socket or the Controller is under initialization (for example, just after PADInit or PADReset).
PAD_ERR_TRANSFER	-3	A data transfer error occurred during the last data transfer. PADStatus therefore contains an invalid Controller status.

a. While the Controller is resetting the origin data, the err member has the value PAD_ERR_NO_CONTROLLER. The ability to reset the origin data by simultaneously holding down the X Button, Y Button, and START for three seconds is built into Nintendo GameCube Controller hardware. During the reset (after the process has started and while the three buttons are being pressed), there is no communication with the Controller. Once any of the three buttons is released, communication with the Controller is possible again.

2.5 PADInit

The PADInit function initializes the Controllers and enables the pad sampling performed by the Revolution hardware. You should call this function before invoking any other PAD functions except PADSetAnalogMode.

Note: The Revolution Video Interface (VI) must be initialized via VIInit before calling PADInit since the VI controls the timing of the Controller data sampling. Initialization provides a default sampling rate in which a game program can get the latest Controller status if PADRead is called right after each vertical retrace interrupt.

Code 2–4 PADInit

```
void PADInit(void);
```

This function has no arguments and no return value.

Note: If PADRead is called immediately after PADInit, each pad status may contain a PAD_ERR_NOT_READY or a PAD_ERR_TRANSFER error code.

2.6 PADRead

The PADRead function takes an array from PADStatus to read as a parameter. The number of elements in the array is determined by PAD_MAX_CONTROLLERS. Each *err* member will hold PAD_ERR_NONE if the pad status is valid; otherwise, they hold other error values and all the other PADStatus members are cleared to zero.

Code 2–5 PADRead

```
void PADRead(PADStatus* status);
```

This function reads the status of all Controllers at once. It has no return value.

Note: The pad status error value PAD_ERR_TRANSFER indicates data sampling failure at the corresponding controller port. This error does not indicate that the controller is disconnected. When a plugged-in controller becomes disconnected, the value PAD_ERR_NO_CONTROLLER is returned. Avoid creating an application that determines controller disconnection based on criteria other than the PAD_ERR_NO_CONTROLLER return value.

2.7 PADClamp

The PADClamp() function takes an array from PADStatus as a parameter. The number of elements in the array is determined by PAD_MAX_CONTROLLERS. The function clamps the inputs of the Control Stick, C Stick, R Button, and L Button using the algorithms described below. It clamps all of PADStatus at once, and it has no return value.

The Revolution SDK has the functions PADClamp2, PADClampCircle2, and PADClampTrigger, which are capable of receiving a wider range of analog input values compared to the Nintendo GameCube functions PADClamp and PADClampCircle. See the *PAD Library Function Reference Manual* for details.

Code 2–6 PADClamp

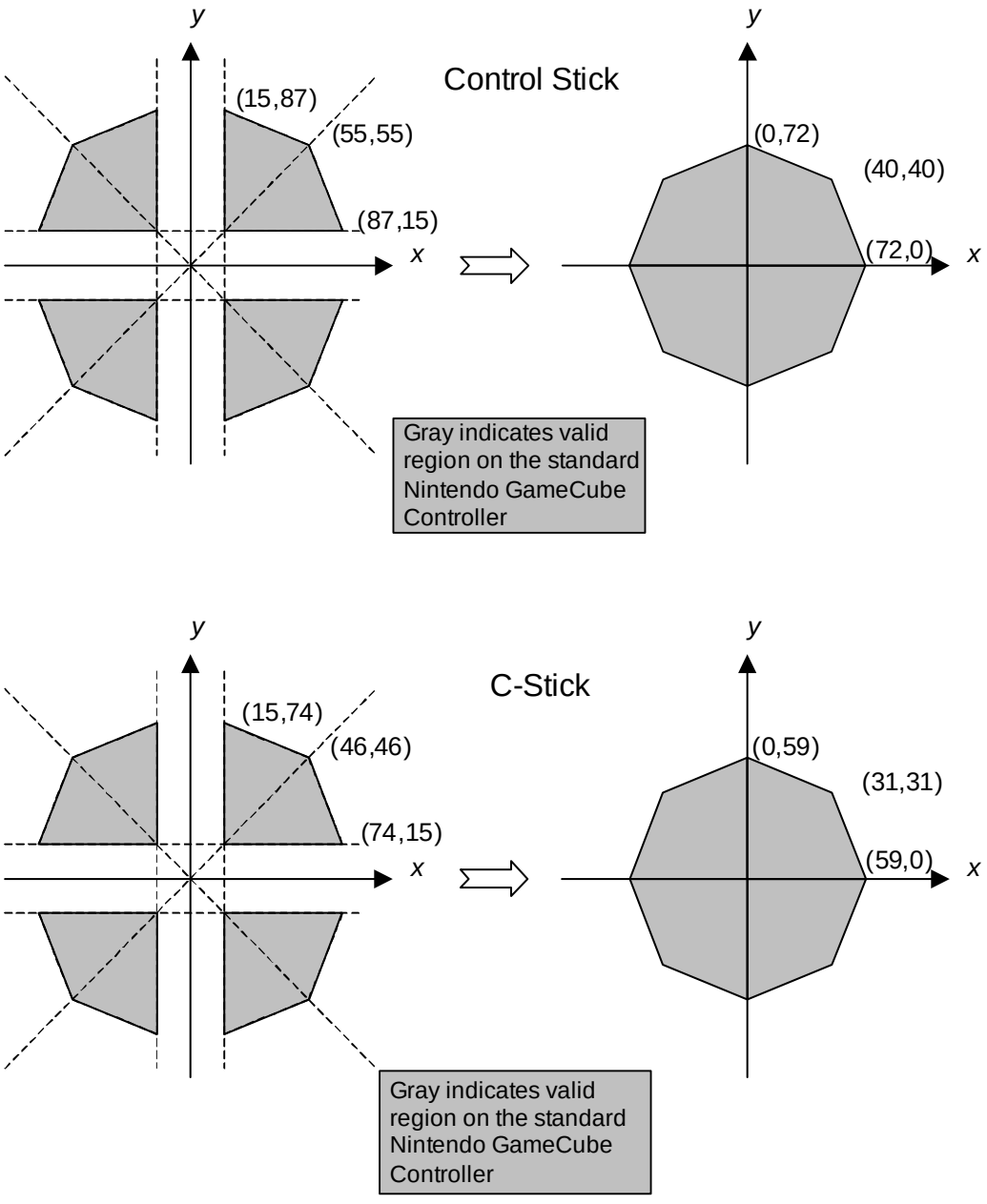
```
void PADClamp(PADStatus* status);
```

2.7.1 Control Stick and C Stick Clamping Algorithm

`PADClamp()` performs dead-zone and outer-octagon clamping for analog sticks as illustrated in the following figure. For the Control Stick, the function first clamps dead zones along both axes (±15). Then it clamps along the outer octagon, whose lengths from the center to the vertices is 72 ($x = 0$ or $y = 0$) and 56.6 ($x = \pm y$). For the C Stick, the function first clamps dead zones along both axes (±15). Then it clamps along the outer octagon, whose lengths from the center to the vertices are 59 ($x = 0$ or $y = 0$) and 43.8 ($x = \pm y$).

The functions `PADClamp2` and `PADClampCircle2` are capable of receiving a wider range of analog stick input values than the `PADClamp` and `PADClampCircle` functions. See *PAD Library Function Reference Manual* for details.

Figure 2–1 PADClamp Algorithm



2.7.2 R Button and L Button Clamping Algorithm

PADClamp() performs dead and outer zone clamping for the R Button and L Button. It clamps the dead zone at (0 - 30) and the outer zone at (180 - 255). The resulting trigger values are between 0 and 150.

2.8 PADReset

The PADReset function resets the Controllers connected to the specified Controller Socket(s). It takes the argument *mask*, which is the OR-ed bit mask of Controllers (PAD_CHAN*n*_BIT) to reset. The function returns TRUE if the reset sequence starts successfully; otherwise, it returns FALSE.

PADReset may return a value of FALSE if the Controller Socket interface was busy performing other transactions. In that case, PADReset should be called again until PADRead detects something other than the PAD_ERR_NO_CONTROLLER error.

If PADRead is called immediately after the successful PADReset, the returned PADStatus of the specified ports may contain a PAD_ERR_NOT_READY or a PAD_ERR_TRANSFER error code.

Code 2–7 PADReset

```
#define PAD_CHAN0_BIT           0x80000000    // Controller 1
#define PAD_CHAN1_BIT           0x40000000    // Controller 2
#define PAD_CHAN2_BIT           0x20000000    // Controller 3
#define PAD_CHAN3_BIT           0x10000000    // Controller 4

BOOL PADReset(u32 mask);
```

2.9 PADRecalibrate

The PADRecalibrate function recalibrates the specified Controllers. Otherwise, the PADRecalibrate function behaves like PADReset. Controllers are automatically calibrated when power to Revolution is turned on.

Note: PADRecalibrate should be called when the Revolution RESET button is pressed. With WaveBird, calling PADRecalibrate does not cause calibration to occur. However, the game programmer does not need to be concerned about this. Call PADRecalibrate whenever the RESET button is pressed. Also, as described in the specifications, the origin will not be reset correctly if a controller is inserted while the control stick is pushed to the right.

Code 2–8 PADRecalibrate

```
#define PAD_CHAN0_BIT           0x80000000
#define PAD_CHAN1_BIT           0x40000000
#define PAD_CHAN2_BIT           0x20000000
#define PAD_CHAN3_BIT           0x10000000

BOOL PADRecalibrate(u32 mask);
```

If PADRead is called immediately after the successful PADRecalibrate, the returned PADStatus of the specified ports may contain a PAD_ERR_NOT_READY or a PAD_ERR_TRANSFER error code. In this case, continue to call PADReset (not PADRecalibrate) until PAD_ERR_NONE is returned as the error code for PADRead.

2.9.1 Hardware Origin Reset

The timing at which a hardware origin reset is performed will be different between the Nintendo GameCube Controller and WaveBird.

Nintendo GameCube Controller

- Initial power-on

  In general, the Nintendo GameCube Controller performs an origin reset at initial power-on (for example: when a Nintendo GameCube is turned on with the Controller plugged in; when a Controller is plugged in to a port of a Nintendo GameCube that is already on).

- Origin reset command

  The Nintendo GameCube Controller has the function to perform an origin reset by pressing the X Button, Y Button, and START/PAUSE simultaneously for three seconds.

WaveBird

- WaveBird power-on

  The origin is set when the power is turned on for the WaveBird. Power cycling the WaveBird will reset the origin. However, these setting will not be reflected to the Nintendo GameCube until the receiver accepts the signal from the WaveBird.

The reset of values following these operations will be performed automatically by the library, so the application (programmer) does not need to be aware of the origin reset.

Note: As indicated in the specifications, the origin is not correctly reset when the control stick is left pushed to the right when the origin reset command (X + Y + START) has been given.

2.10 SISetSamplingRate

The `SISetSamplingRate` function sets the Controller data sampling rate *msec* in milliseconds (from 1 millisecond to 11 milliseconds). All of the Controllers are sampled at the rate specified by this function. If *msec* = 0, the function sets the default sampling rate, which allows the game program to get the latest Controller status if `PADRead` is called after each vertical retrace interrupt. There is no return value.

Code 2–9 SISetSamplingRate

```
void SISetSamplingRate(u32 msec);
```

Note: If `PADRead` is called at a faster rate than the one specified by `SISetSamplingRate`, only the first `PADRead` returns the valid input. The following `PADRead`s return `PAD_ERR_TRANSFER` until the next period.

The transfer cycle for controller status is about 2.2 msecs for the WaveBird. Accordingly, even if a sample rate higher than this transfer cycle is set for these controllers, the same input values will be obtained. Even for the same input values, when `PADRead` is called with a longer cycle than the set sample frequency, `PADREad` will fail

2.11 PADSetAnalogMode

Note: This functions sets analog mode for the Nintendo GameCube Controller being used. This function was kept to expand the type of controllers that can be connected to the controller ports. As of March 1, 2006, this function is not required.

The `PADSetAnalogMode` function specifies the analog mode of the Controllers to use. The analog mode controls the resolution of the Controller's analog inputs stored in `PADStatus` as shown below. The default mode is `PAD_MODE_3`.

Note: The Nintendo GameCube Controller does not support analog input values (AnalogA/B) from the A Button or B Button. Furthermore, the Revolution Controller Library does not support analog input from the A Button or B Button.

Table 2–4 Analog Modes 0, 5, 6, 7

Modes 0, 5, 6, 7	
StickX/Y	All 8 bits are valid.
SubstickX	All 8 bits are valid.
TriggerLeft/Right	Only left-most 4 bits are valid. Other bits are set to zero.
AnalogA/B (See note)	Only left-most 4 bits are valid. Other bits are set to zero.

Table 2–5 Analog Mode 1

Mode 1	
StickX/Y	All 8 bits are valid.
SubstickX/Y	Only left-most 4 bits are valid. Other bits are set to zero.
TriggerLeft/Right	All 8 bits are valid.
AnalogA/B (See note)	Only left-most 4 bits are valid. Other bits are set to zero.

Table 2–6 Analog Mode 2

Mode 2	
StickX/Y	All 8 bits are valid.
SubstickX/Y	Only left-most 4 bits are valid. Other bits are set to zero.
TriggerLeft/Right	Only left-most 4 bits are valid. Other bits are set to zero.
AnalogA/B (See note)	All 8 bits are valid.

Table 2–7 Analog Mode 3

Mode 3	
StickX/Y	All 8 bits are valid.
SubstickX/Y	All 8 bits are valid.
TriggerLeft/Right	All 8 bits are valid.
AnalogA/B (See note)	All 8 bits are always zero.

Table 2–8 Analog Mode 4

Mode 4	
StickX/Y	All 8 bits are valid.
SubstickX/Y	All 8 bits are valid.
TriggerLeft/Right	All 8 bits are always zero.
AnalogA/B (See note)	All 8 bits are valid.

Code 2–10 PADSetAnalogMode

```
#define PAD_MODE_0              0
#define PAD_MODE_1              1
#define PAD_MODE_2              2
#define PAD_MODE_3              3
#define PAD_MODE_4              4
#define PAD_MODE_5              5
#define PAD_MODE_6              6
#define PAD_MODE_7              7

void PADSetAnalogMode(u32 mode);
```

Note: PADSetAnalogMode suspends the hardware Controller sampling. The next PADRead returns PAD_ERR_NO_CONTROLLER errors for the currently attached Controllers. The specified analog mode takes effect the next time PADReset or PADInit is called. PADSetAnalogMode may be called before PADInit.

2.12 PADButtonDown

The PADButtonDown macro identifies which button(s) have just been pressed.

Code 2–11 PADButtonDown

```
#define PADButtonDown(buttonLast, button)   \
    (((buttonLast) ^ (button)) & (button))
```

The argument buttonLast is the previous button status and button is the current one. Both are returned in PADStatus by PADRead(). PADButtonDown returns the OR-ed Controller button bits of the pressed button(s).

2.13 PADButtonUp

The PADButtonUp macro identifies which button(s) have just been released.

Code 2–12 PADButtonUp

```
#define PADButtonUp(buttonLast, button)     \
    (((buttonLast) ^ (button)) & (buttonLast))
```

The argument buttonLast indicates the previous button status and button is the current one. Both are returned in PADStatus by PADRead(). PADButtonUp returns the OR-ed pad button bits of the released button(s).

3 Rumble Motor Control API

This chapter describes the Wii controller's rumble motor control functions, which are described in the header file below.

Code 3–1 PAD API header file

```
#include <revolution/pad.h>
```

3.1 Motor State

The standard Wii controller has a single rumble motor which can be in one of the three states shown in the following table:

Table 3–1 Rumble Motor Status

Defined Name	Code	Description
PAD_MOTOR_STOP	0	The motor remains stopped, or stops naturally if it is rumbling.
PAD_MOTOR_RUMBLE	1	The motor keeps rumbling, or starts rumbling.
PAD_MOTOR_STOP_HARD	2	The motor stops hard if it is rumbling.

You can stop the motor by force, or by terminating the motor power supply. Although the motor can be programmed in several ways, we expect the following motor state transitions to be common:

PAD_MOTOR_STOP ⇒ PAD_MOTOR_RUMBLE ⇒ PAD_MOTOR_STOP

PAD_MOTOR_STOP ⇒ PAD_MOTOR_RUMBLE ⇒ PAD_MOTOR_STOP_HARD ⇒ PAD_MOTOR_STOP

3.2 PADControlMotor and Utility Macro Functions

PADControlMotor controls the specified Controller motor state. It takes the arguments chan, a value of PAD_CHAN*n*, and command, a value of PAD_MOTOR_* (the default motor state is PAD_MOTOR_STOP). PADControlMotor has no return value.

Code 3–2 PADControlMotor

```
#define PAD_CHAN0             0  // Controller 1
#define PAD_CHAN1             1  // Controller 2
#define PAD_CHAN2             2  // Controller 3
#define PAD_CHAN3             3  // Controller 4

#define PAD_MOTOR_STOP        0
#define PAD_MOTOR_RUMBLE      1
#define PAD_MOTOR_STOP_HARD   2

void PADControlMotor(int chan, u32 command);
```

Note: Controllers must be initialized first via PADInit.

We provide three utility macros with PADControlMotor for ease of use:

Code 3–3 PADControlMotor utility macros

```
#define PADStartMotor(chan)      PADControlMotor((chan), PAD_MOTOR_RUMBLE)
#define PADStopMotorHard(chan)   PADControlMotor((chan), PAD_MOTOR_STOP_HARD)
#define PADStopMotor(chan)       PADControlMotor((chan), PAD_MOTOR_STOP)
```

3.3 PADControlAllMotors

The PADControlAllMotors function takes the parameter *CommandArray*, an array[PAD_MAX_CONTROLLERS] of PAD_MOTOR_*, and sets every Controller motor state at once. It has no return value.

Code 3–4 PADControlMotor

```
#define PAD_MOTOR_STOP          0
#define PAD_MOTOR_RUMBLE        1
#define PAD_MOTOR_STOP_HARD     2

void PADControlAllMotors(u32* commandArray);
```

Note: The Revolution hardware always sets the four motor states by a single operation; therefore, calling PADControlAllMotors is rather more efficient than calling PADControlMotor four times.

3.4 Controller Rumble Feature Availability Detection

There are instances where the controller Rumble Feature is disabled or a controller without a Rumble Feature is being used when the presentation requires the feature. In such case, an alternative presentation may become necessary.

The application should be able to detect a disabled Rumble Feature. The PADRead function can determine whether the controller has a Rumble Feature.

4 Coding Sample

4.1 Simple Demo

The following program demonstrates a simple use of the PAD API.

Code 4–1 Simple Demo

```
#include <revolution.h>

PADStatus Pads[PAD_MAX_CONTROLLERS];

void main(void)
{
    u16 button = 0; // Previous button status
    u16 down;       // Buttons just pressed down
    u16 up;         // Buttons just released

    VIInit();       // VI must be initialized before PAD
    PADInit();

    do {
        PADRead(Pads);

        if (Pads[0].err != PAD_ERR_NONE)
            continue;

        down = PADButtonDown(button, Pads[0].button);
        up   = PADButtonUp  (button, Pads[0].button);
        button = Pads[0].button;

        PADClamp(Pads);

     OSReport("Buttons: %c%c%c%c %c Stick: (%4d, %4d) SubStick: (%4d, %4d) Trigger (%3d, %3d)
Down: %c%c Up: %c%c\n",
            (Pads[0].button & PAD_BUTTON_A) ? 'A' : '_',
            (Pads[0].button & PAD_BUTTON_B) ? 'B' : '_',
            (Pads[0].button & PAD_BUTTON_X) ? 'X' : '_',
            (Pads[0].button & PAD_BUTTON_Y) ? 'Y' : '_',
            (Pads[0].button & PAD_BUTTON_START) ? 'S' : '_',
            Pads[0].stickX,
            Pads[0].stickY,
            Pads[0].substickX,
            Pads[0].substickY,
            Pads[0].triggerLeft,
            Pads[0].triggerRight,
            (down & PAD_BUTTON_A) ? 'A' : '_',
            (down & PAD_BUTTON_B) ? 'B' : '_',
            (up & PAD_BUTTON_A) ? 'A' : '_',
            (up & PAD_BUTTON_B) ? 'B' : '_');
    } while (!(button & PAD_BUTTON_MENU));
}
```

The Video Interface must first be initialized via `VIInit`. Then a call to `PADInit` initializes the Controllers.

In the body of the `do` loop, the `PADRead` function reads the status of all the Controllers, then `PADClamp` clamps the analog input data.

Note: Both `PADRead` and `PADClamp` take an array[`PAD_MAX_CONTROLLERS`] of `PADStatus`.

The program then checks the error code of the Controller connected to the first Controller Socket. If the error code of the first Controller is not `PAD_ERR_NONE`, the program ignores the current input. Otherwise, the program prints out the current `PADStatus` of the first Controller, as well as an indication of whether the A Button or B Button has just been pressed and/or released. The program terminates when the START/PAUSE Button of the first Controller is pressed.

4.2 Handling Controller-Related Errors

The next program illustrates how to handle Controller-related errors such as may occur when no Controller is attached to the Revolution console at the start of game play, or when one or more Controllers are disconnected and then reconnected to the console during game play.

As always, this program initializes the VI, then the Controllers, in that order. The `for` loop of the program checks each returned error code returned by `PADRead`.

The variable *connectedBits* holds the bits of the Controllers recognized by the program. The program recognizes an attached game by the error code `PAD_ERR_NONE` or `PAD_ERR_TRANSFER`.

The variable *resetBits* holds the bits of empty Controller ports by checking all the *err* members. The program calls `PADReset` for those Controller ports indicated by *resetBits*.

Note: Once the program recognizes a set of plugged-in Controllers (i.e., *connectedBits* is not zero), the variable *resetBits* is OR-ed with *connectedBits*. Thus the program does not try to reset the unused Controller ports to minimize CPU overhead.

Finally, the program shows the current status of the Controllers.

Code 4–2 Handling Errors

```
#include <revolution.h>

PADStatus Pads[PAD_MAX_CONTROLLERS];

static void PrintPads(void)
{
    int chan;

    OSReport("Port  A      B      XY M ZLR +Pad Left         Right        Trigger\n");
    for (chan = 0; chan < PAD_MAX_CONTROLLERS; ++chan)
    {
        OSReport("%d[%-2d] %c[%3d] %c[%3d] %c%c %c %c%c%c %c%c%c%c (%4d, %4d) (%4d, %4d) (%3d,
%3d)\n",
            chan,
            Pads[chan].err,
            (Pads[chan].button & PAD_BUTTON_A) ? 'O' : '_',
            Pads[chan].analogA,
            (Pads[chan].button & PAD_BUTTON_B) ? 'O' : '_',
            Pads[chan].analogB,
            (Pads[chan].button & PAD_BUTTON_X) ? 'O' : '_',
            (Pads[chan].button & PAD_BUTTON_Y) ? 'O' : '_',
            (Pads[chan].button & PAD_BUTTON_START) ? 'O' : '_',
            (Pads[chan].button & PAD_TRIGGER_Z) ? 'O' : '_',
            (Pads[chan].button & PAD_TRIGGER_L) ? 'O' : '_',
            (Pads[chan].button & PAD_TRIGGER_R) ? 'O' : '_',

            (Pads[chan].button & PAD_BUTTON_LEFT)  ? '<' : '_',
            (Pads[chan].button & PAD_BUTTON_RIGHT) ? '>' : '_',
            (Pads[chan].button & PAD_BUTTON_UP)    ? '^' : '_',
            (Pads[chan].button & PAD_BUTTON_DOWN)  ? 'v' : '_',

            Pads[chan].stickX,
            Pads[chan].stickY,
            Pads[chan].substickX,
            Pads[chan].substickY,
            Pads[chan].triggerLeft,
            Pads[chan].triggerRight);
    }
}

void main(void)
{
    u32 padBit;
    u32 resetBits;
    u32 connectedBits;
    int chan;

    VIInit();
    PADInit();

    connectedBits = 0x0;

    for (;;)
    {
        PADRead(Pads);

        resetBits = 0x0;
        for (chan = 0; chan < PAD_MAX_CONTROLLERS; ++chan)
        {
            padBit = PAD_CHAN0_BIT >> chan;
```

```
            switch (Pads[chan].err)
            {
              case PAD_ERR_NONE:
              case PAD_ERR_TRANSFER:
                connectedBits |= padBit;
                break;
              case PAD_ERR_NO_CONTROLLER:
                resetBits |= padBit;
                break;
              case PAD_ERR_NOT_READY:
              default:
                break;
            }
        }
        if (connectedBits)
        {
            resetBits &= connectedBits;
        }
        if (resetBits)
        {
            PADReset(resetBits);
        }

        if (connectedBits)
        {
            OSReport("\033c");   // Resets the terminal
            OSReport("\nAttached Controllers: 0x%1x.\n", connectedBits);
            PrintPads();
            PADClamp(Pads);
            OSReport("\nClamped\n");
            PrintPads();
        }
        else
        {
            OSReport("Please connect Controllers\n");
        }

        VIWaitForRetrace();
    }
}
```

Notes:

- All Revolution game programs must support Controller live plug-in/out.

4.3 Further Examples

You can find more Controller examples under `/revolution/build/demos/paddemo/src`. The `cont.c` file implements some Controller utility functions which you might want to use in your game.

`ReadCont` performs the following functions:

- Generates key repeat inputs that are independent of the current TV format, and thus not affected by different refresh rates
- Emulates +Control Pad inputs from Control Stick input
- Keeps the previous Controller input if `PAD_ERR_TRANSFER` is returned so that the game main loop can ignore the error code returned by the `PADRead`
- Mixes up four Controller inputs and generates the pseudo fifth Controller input to support single play games more easily

The function `InitCont` can direct the main loop to check only the specified Controller ports, or to check the initial set of Controller ports (i.e., the ports where Controllers were originally attached).

The `contdemo.c` file illustrates the use of `InitCont` and `ReadCont`. It also shows how to recalibrate and change the attached Controller ports at the soft reset.

The `motordemo.c` file illustrates the use of `PADControlAllMotors` and how to generate various strengths of rumble motor effects.