UDP and OSX Quartz Event with Go Language

Nintendo DS is a funny little console! I love it's gamepad, which remind me the glorious SNES gamepad. It also sports WIFI capabilities (sadly with no support for WPA). One day, I was thinking:

<< *Hey! Wouldn't be super cool if I can use the DS as a gamepad ?* >>

So I did some research on google and I found this project called DS2Key.

DS2Key is a nice little program which stream Nintendo DS input events to a remote server, the server then decode this packet emitting 'virtual' keystrokes.

DS2Key server run on Linux and Windows, but seems to me that there are no working server for OSX. So I decided to built my own.

Choosing the right tools

First things I did was understanding what the server program has to do, which boils down to 3 things:

- Listening for UDP packets on a port
- Decoding UDP packets into an array of human friendly key codes
- Emitting keystrokes

First two point are very easy to implement on almost every programming language, the third one otherwise is more tricky. Doing some research I found that is possibile to emulate low level input events using Quartz Tap Event, which are parts of the OSX Application Services framework.

After a small tour on XCode, trying to code the whole thing in Objective C, I discovered that Application Services framework is written in C, and GO language can embed C code through CGO package. We'll come back to CGO later.

Understanding DS2Key UDP Packets

DS2Key packet payload is 11 bytes long.

Nintendo DS has 12 buttons, since a single button can be ON/OFF we can represent it's status using a bit for each key, needing a total of 12 bit (3 nibbles). Since a byte is 8 bit, we have to use 2 byte to represent that status of game pad.

This is what a packet payload look like:

```
30 00 41 04 00 00 00 00 00 00 00
```

Starting from leftmost byte, we took the 3rd byte and the 4th byte.

```
41 04
```

Those are HEX values, converting them to binary, we obtain status for all the gamepad buttons:

```
; Byte at index 2
; 30 00 [41] 04 00 00 00 00 00 00 00
; HEX 41 --- BIN 0100 0001 --- INT BASE10 65

0 1 0 0  0 0 0 1
| | | |  | | | |--- KEY_A  [Pressed]  2^0  INT=1     1 << 0
| | | |  | | |----- KEY_B             2^1  INT=2     1 << 1
| | | |  | |------- KEY_SELECT        2^2  INT=4     1 << 2
| | | |  |--------- KEY_START         2^3  INT=8     1 << 3
| | | |------------ KEY_RIGHT         2^4  INT=16    1 << 4
| | |-------------- KEY_LEFT          2^5  INT=32    1 << 5
| |---------------- KEY_UP [Pressed]  2^6  INT=64    1 << 6
|------------------ KEY_DOWN          2^7  INT=128   1 << 7


; Byte at index 3
; 30 00 41 [04] 00 00 00 00 00 00 00
; HEX 04 --- BIN 0000 0100 --- INT BASE10 4

0 0 0 0  0 1 0 0
| | | |  | | | |--- KEY_R             2^0  INT=1     1 << 0
| | | |  | | |----- KEY_L             2^1  INT=2     1 << 1
| | | |  | |------- KEY_X             2^2  INT=4     1 << 2
| | | |  |--------- KEY_Y             2^3  INT=8     2 << 3
| | | |------------ UNUSED
| | |-------------- UNUSED
| |---------------- UNUSED
|------------------ UNUSED
```

For each byte, we can use an array to represent it's bits.

```
byte2 := []string{"KEY_A",        // 1 << 0 = 1
                  "KEY_B",
                  "KEY_SELECT",     // 1 << 2 = 4
                  "KEY_START",
                  "KEY_RIGHT",
                  "KEY_LEFT",
                  "KEY_UP",
                  "KEY_DOWN"}
```

```
byte3 := []string{"KEY_R",
                  "KEY_L",
                  "KEY_X",
                  "KEY_Y"}
```

For each element of the arrays, we can get it's integer value by shifting 1 left `index` times.

Packet parsing

We now associate the previously defined array to an offset, using a map.

```
var KEYS = map[uint32] []string {
    2: []string{"KEY_A", "KEY_B",
                "KEY_SELECT", "KEY_START",
                "KEY_RIGHT", "KEY_LEFT", "KEY_UP", "KEY_DOWN"},

    3: []string{"KEY_R", "KEY_L", "KEY_X", "KEY_Y"},
}
```

Pseudo code for parsing:

- iterate KEYS map; we got the `offset` and the `keys` represented by this offset
- get `byte integer value` at offset
- iterate throught keys, we have an `index` and the `string value`
    - get `integer value` of key using a left shift: `1 << index`
    - bitmask the `integer value` with `byte integer value`
        - if result is equal to key integer value, the **key is pressed**

A practical example

If, we press START and SELECT key on the nintendo ds, we get:

```
0000 1100
```

Now, we iterate as usual our array of KEYS, we find the integer value by shifting left, but before comparing the current item value, we apply a `bitmask` using a `bitwise and`.

A bitwise and took binary value of two integers variable and apply a `logical and` on them. Since logical and return 1 only if both bit are 1, we can easily isolate the bit we need to check.

```
---iteration 0
KEY_A,
0000 0001 AND
0000 1100
===========
0000 0000

---iteration 1
KEY_B,
0000 0010 AND
0000 1100
===========
0000 0000

--- iteration 2
KEY_SELECT,
0000 0100 AND
0000 1100
===========
0000 0100   KEY_SELECT is pressed!

--- iteration 3
KEY_START,
0000 1000 AND
0000 1100
===========
0000 1000   KEY_START is pressed!

...
```

Show me the code!

```
package parser

// Key is offset of the byte we need to parse
// Value is an array of strings, we use the index to calculate
// the binary value, by applying a left shift 'index' times.
var KEYS = map[uint32] []string {
    2: []string{"KEY_A", "KEY_B",
                "KEY_SELECT", "KEY_START",
                "KEY_RIGHT", "KEY_LEFT", "KEY_UP", "KEY_DOWN"},

    3: []string{"KEY_R", "KEY_L", "KEY_X", "KEY_Y"},
}

func DetectKeys(payload []byte) []string {
    // for each pressed key, we push it's string value to this array
    pressedKeys := []string{}

    for offset, keys := range KEYS {
        value := uint32(payload[offset])
```

```
        for n, keyStr := range keys {
            mask := uint32(1 << uint32(n))

            if (value & mask) == mask {
                pressedKeys = append(pressedKeys, keyStr)
            }
        }
    }

    return pressedKeys
}
```

Easy.

Defining key binding

Our tiny parser return an array of strings, each string is an human readable label of a key. Since I am a lazy programmer, at the moment binding with keyboard keys will be hard coded, no fancy configuration files. The biggest drawnback is that letters keycodes depends on your keyboard layout. So, excluding modifier and special keys (RETURN, SPACE, etc) other keys will be different for each layout.

```
; US Layout
KEY_START    -> RETURN
KEY_SELECT   -> SPACE

KEY_A        -> a
KEY_B        -> s

KEY_X        -> z
KEY_Y        -> x

KEY_L        -> q
KEY_R        -> e

KEY_UP       -> up arrow
KEY_DOWN     -> down arrow
KEY_LEFT     -> left arrow
KEY_RIGHT    -> right arrow
```

To generate a virtual keystroke, we have to use Quartz Tap Event, through C call. We'll use CGO.

```
# file: ds2key-srv/kbd/kbd.go
package kbd

/*
```

```
#cgo CFLAGS: -Qunused-arguments
#cgo LDFLAGS: -framework ApplicationServices
#include <ApplicationServices/ApplicationServices.h>
#include <Carbon/Carbon.h>

void keyevt(int keycode, bool isdown) {
    CGEventRef evt;
    evt = CGEventCreateKeyboardEvent(NULL, (CGKeyCode)keycode, isdown);
    CGEventPost(kCGSessionEventTap, evt);
}
*/
import "C"

// 'Carbon.h' define some useful constants
// to deal with KeyCodes.
// Remember that letters keycodes are layout specific.
var KEYS = map[string] int32{
    "KEY_UP"    : C.kVK_UpArrow,
    "KEY_DOWN"  : C.kVK_DownArrow,
    "KEY_LEFT"  : C.kVK_LeftArrow,
    "KEY_RIGHT" : C.kVK_RightArrow,

    "KEY_L"     : C.kVK_ANSI_Q,
    "KEY_R"     : C.kVK_ANSI_E,

    "KEY_A"     : C.kVK_ANSI_A,
    "KEY_B"     : C.kVK_ANSI_S,

    "KEY_X"     : C.kVK_ANSI_Z,
    "KEY_Y"     : C.kVK_ANSI_X,

    "KEY_START" : C.kVK_Return,
    "KEY_SELECT": C.kVK_Space,
}

func KeyDown(key string) {
    C.keyevt(C.int(KEYS[key]), C.bool(true))
}

func KeyUp(key string) {
    C.keyevt(C.int(KEYS[key]), C.bool(false))
}
```

The comment block before `import "C"` statement is C code we can call using the `C` package.

With `C package` we can also call variables, and make type conversion from GO types to C types.

Tie up everything

We now have two files:

- `parser/parser.go`
- `kbd/kbd.go`

We need to create a main file, which with a great work of fantasy we will call `main.go`.

Pseudo code for this file is:

- init an empty map of type `map[string] bool` to holds the status of keys, so we can determine if a key has been released or is still pressed. We call this variable `status`
- setup an UDP Listener on a customizable port (9501 by default)
- when we receive a packet we pass it to the `parser.DetectKeys`
- parser return back an array of strings
- iterate through the array of strings
    - if key is not pressed, we emulate a key down, by calling `kbd.KeyDown`, then we create an entry on `status` with value `true`
    - if key is pressed, we do nothing
- compare keys in the status array with keys in parsed packet
    - if a keys was pressed but is not in the parsed packet array, we release it by emulating a key up (`kbd.KeyUp`)
- loop forever

Source code:

```
# main.go
package main

import (
    "fmt"
    "log"

    "strings"

    "flag"

    "net"

    "github.com/andreadipersio/ds2key-srv/parser"
    "github.com/andreadipersio/ds2key-srv/kbd"
)
```

```
var (
    port int
    verbose bool

    status map[string] bool // used to determine when to release keys
)

func init() {
    flag.IntVar(&port, "port", 9501, "DS2KEY Port")
    flag.BoolVar(&verbose, "verbose", false, "Enable logging of keystrokes on stderr")

    flag.Parse()

    status = make(map[string] bool)
}

func releaseAll() {
    for key, isPressed := range status {
        if isPressed {
            kbd.KeyUp(key)
            status[key] = false
        }
    }
}

func released(keys []string, key string) bool {
    for _, newKey := range keys {
        if newKey == key {
            return false
        }
    }

    return true
}

func logKeyStatus() {
    var s = []string{}

    for key, isPressed := range status {
        if !isPressed {
            continue
        }

        s = append(s, fmt.Sprintf("[%v]", key))
    }

    log.Print(strings.Join(s, " --- "))
}


func main() {
    fullAddr := fmt.Sprintf(":%d", port)
```

```
    addr, err := net.ResolveUDPAddr("udp", fullAddr)

    log.Print(addr)

    if err != nil {
        log.Panicf("Wrong address %v: %v", fullAddr, err);
    }

    sock, err := net.ListenUDP("udp", addr)

    if err != nil {
        log.Panicf("Cannot listen from %v: %v", fullAddr, err);
    }

    buf := [11]byte{}

    for {
        if _, err := sock.Read(buf[0:]); err != nil {
            log.Printf("ERROR::%v", err)
            continue
        }

        // first 4 bytes contains status of pad buttons
        payload := buf[:4]

        keys := parser.DetectKeys(payload);

        // all buttons on gamepad released
        if len(keys) == 0 {
            releaseAll()
            continue
        }

        for _, key := range keys {
            stillDown, wasPressed := status[key]

            if wasPressed && stillDown {
                continue
            } else {
                kbd.KeyDown(key)
            }

            status[key] = true
        }

        for key, wasPressed := range status {
            if wasPressed && released(keys, key) {
                kbd.KeyUp(key)
                status[key] = false
            }
        }

        if verbose {
```

```
            logKeyStatus()
        }
    }
}
```

Source

Source code for this program can be found on Github.

Conclusion

I spent some days making this little program, at the beginning I tried writing it using *Objective C / Cocoa / XCode*, but after writing 200 lines of code only to implement a UDP listener, I was really missing *GO*. When I discovered that ApplicationService framework is written in C I did some research on how to call C code from GO and found *CGO*. Then everything went downhill!

After some hours I had a working program, and I was finally able to play my retro games with a proper gamepad, Nintendo DS.

A programming language that make you able to create useful software while having fun doing that, is a keeper.

Long live GO.