Gopher2600 Application Architecture

Although `Gopher2600` was designed as a stand-alone executable with a single binary and not as a library package, it is nonetheless possible to use the emulation code in other applications.

The diagram below shows the major packages used by `Gopher2600`. There are many others (and some will be mentioned briefly) but these are main ones we need to understand.

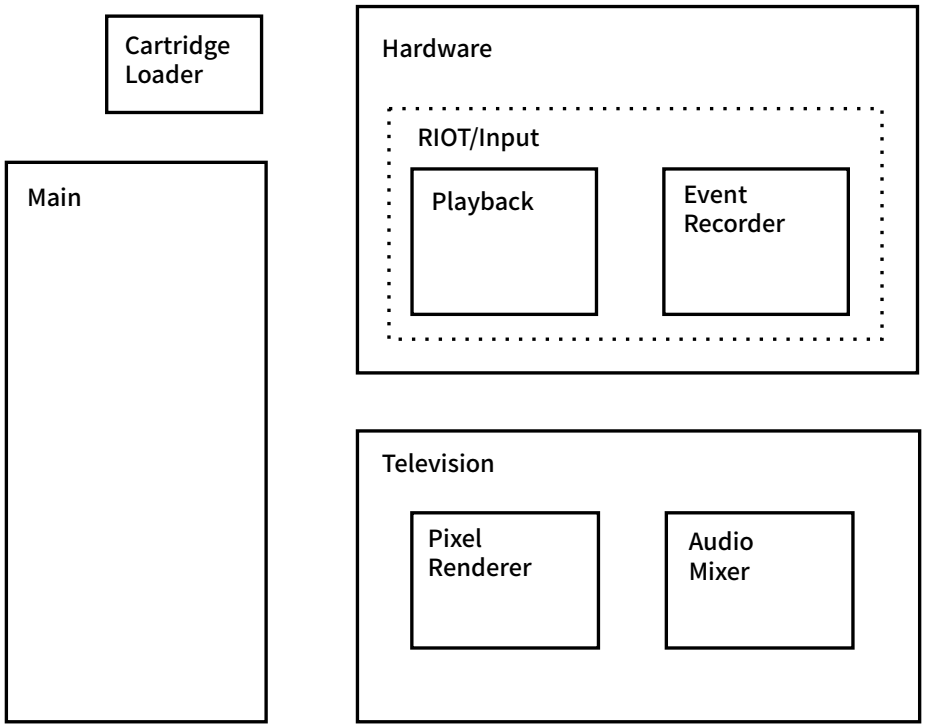

Figure 1: Major architectural features

The remainder of this document will briefly examine each of these packages and will conclude with a short example of a simple utility that can perform a useful audit of a collection of cartridges.

Hardware Package

The `hardware` package contains the meat-and-potatoes of the emulation. It contains the emulation code for the CPU, TIA, memory and everything else to do with the Atari 2600. The most important parts of this package is the `VCS` type.

```
type VCS struct {
    CPU  *cpu.CPU
    Mem  *memory.VCSMemory
    TIA  *tia.TIA
    RIOT *riot.RIOT

    TV television.Television

    Panel           input.Port
    HandController0 input.Port
    HandController1 input.Port
}
```

For most applications, the `CPU`, `Mem`, `TIA` and `RIOT` fields can be ignored and are only required when very low level information is required. For example a VCS debugger would need low level information.

A new instance of the `VCS` type is created with the `NewVCS()` function.

```
func NewVCS(tv television.Television) (*VCS, error)
```

The single argument is an instance of a previously instantiated `Television` the interface for which is found in the `television` package. We'll look at this package in detail later.

The final three fields expose the three methods of user input found in the VCS. The three methods of input are the two `hand controllers`, one for each player; and the VCS's front panel, where we find the reset switch, the difficulty toggles, etc.

These three fields are all instances of the `Port` interface which is found in the `input` package. A `Gopher2600` application does not need to implement this interface but it will need to use it.

```
type Port interface {
    String() string
    Handle(Event, EventValue) error
    AttachPlayback(Playback)
    AttachEventRecorder(EventRecorder)
}
```

The `String()` function implements the standard Go `Stringer` interface. In `Gopher2600`, Stringer implementations tend to return summary information for the underlying component.

The `Handle()` and `AttachPlayback()` functions provide two alternatives for getting input *into* the emulation. `AttachEventRecorder()` meanwhile is a way to get input *out* of the emulation or, more accurately, to *record* input.

`AttachPlayback()` expects an instance of the `Playback` interface.

```
type Playback interface {
    CheckInput(id ID) (Event, EventValue, error)
}
```

The `Playback` interface's single function is called every video cycle (3.49Hz). The single ID argument identifies which of the three `ports` is being polled. This is for convenience - the same implementation can service all three ports. The IDs for the three ports are:

- HandControllerZeroID
- HandControllerOneID
- PanelID

The `CheckInput()` function returns three values. The `Event` value specifies what has happened on the port. The full list can be found in the `input` package documentation but examples of events are `Fire`, `PaddleSet` and `PanelSelect`.

The `EventValue` value is the actual value associated with the event. For example, for the `Fire` event the value can be `true` or `false`.

The `Port` interface also requires an `AttachEventRecorder()` function. Similar to the `AttachPlayback()` function this requires an instance of the `EventRecorder` interface.

```
type EventRecorder interface {
    RecordEvent(ID, Event, EventValue) error
}
```

From what we now know about the `Playback` interface the arguments to the `RecordEvent` function should be obvious. The `ID` identifies which port the event is from, the `Event` describes what has happened (`Up`, `Down`, `NoEvent`, etc.) and `EventValue` is the actual value of the `Event`.

The Handle() function

The `Handle()` function in the `Port` interface is a more immediate alternative to the `Playback` interface and depending on your use-case it might be more convenient to use. For example, the main `Gopher2600` application uses the `Handle()` function to submit live input from a hardware controller (joypad, keyboard, etc.)

The advantage of the `Handle()` function is that your code can submit the `Event` and continue more-or-less straight-away. With a `Playback` implementation, having to respond to a `CheckInput()` may be tricky to coordinate.

Can you call `Handle()` and attach a `Playback` implementation in the same application? There's no reason why not but obviously results may not be as expected.

Television Package

Returning to the package diagram from the start of the document: the `Television` package defines the operation of a television connected to the console. Crucially, this package does not directly implement the visual or audible representation of the TV signal. Instead, it provides two interfaces. The `PixelRenderer` interface and the `AudioMixer` interface.

`PixelRenderer` simplifies the three major operations that a television must perform: when to start a new television frame, a new scanline and it must be able to apply color to a pixel. Respectively, the three functions related to these operations are `NewFrame()`, `NewScanline()` and `SetPixel()`.

```
type PixelRenderer interface {
    NewFrame(frameNum int) error
    NewScanline(scanline int) error
    SetPixel(x, y int, red, green, blue byte, vblank bool) error
    SetAltPixel(x, y int, red, green, blue byte, vblank bool) error
    EndRendering() error
}
```

In addition the three principle functions, there are other two required functions. The first allow a pixel to have an *alternative* color applied to it. This `SetAltPixel()` function is there to support debugging functionality and while it must be implemented by PixelRenderer implementations, be aware that it might be removed in the future.

The final `PixelRenderer` function allows an implementation to be notified when rendering is to end. We can think of this as being when the television is turned off and is an opportunity for implmentations to close or dispose of any resources that may be in use.

The `AudioMixer` interface does for sound what `PixelRenderer` does for video.

```
type AudioMixer interface {
    SetAudio(audioData uint8) error
    EndMixing() error
}
```

`SetAudio()` is called every Atari 2600 color clock. If you don't know what that is, don't worry, it is sufficient to understand that the bitrate of the VCS audio stream is `31403Hz`. The `EndMixing()` function, like the `EndRendering()` function, is called when a television session is completed and is an opportunity to cleanup any resources used.

Novel Uses for `PixelRender` & `AudioMixer`

Implementations of the `PixelRenderer` and `AudioMixer` interfaces are not limited to the production of a moving image or a sound that can be immediately seen or heard. For example, in addition to the expected purposes, the `Gopher2600` project uses the `PixelRenderer` interface to create screen digests. Screen digests are produced by SHA-1 hashing and are the key ingredient to the regression database system.[1] Without going into detail, the regression database system works by calculating a hash for a given cartridge and storing it for future comparison.

Cartridge Loader

The final package shown in the diagram at the top of the document is the `cartidgeloader` package. the `loader` type contained in this package is used to specify which cartridge is to be loaded into the emulator. A new loader can be initialised in the usual go fashion:

```
ld := cartridgeloader.Loader{Filename: "pitfall.bin"}
```

The other public fields are the `Format` field and the `Hash` field. The `Hash` field can be used to specify what the expected hash of the file should be.[2] It can be left empty if the hash is unknown.

The `Format` field meanwhile, can be used to specify the cartridge format. The cartridge emulation is pretty good at detecting the format but it may need over-riding on occasion. The list of cartridge and valid `Format` strings can be found in the `cartridge` package documentation.[3]

The other elements of the `cartridgeloader` package are used by the emulation. For example, the `Load()` function is used to load the cartridge data into the emulation. The neat thing about the `cartridgeloader` mechanism is that it transparently handles the loading of files over `HTTP` as well as files stored locally.

Example: Building a Gopher2600 Application

The following is a small example of a utility that uses the packages described above. The utility we'll be creating will perform a simple audit of a collection of Atari 2600 cartridges and output the results to the console.

The full source code for this example can be found at:

`http://github.com/JetSetIlly/Gopher2600-Utils/Example-Audit/`

[1] Keen computer scientists will say that SHA-1 is fundamentally insecure and they'd be right. However, the purpose to which we're putting the algorithm is not cryptographic in nature.

[2] The hash is a SHA-1 hash of the cartridge.

[3] The `Atari 2600` by right, can only access 4k of cartridge space. However, with some neat electronics, more than 4k can be accessed.

The first step is to create our `main.go` file. We'll also include, the minimum code required to accept an argument from the command-line and walking over all the files in the specified path.

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s <path to ROMs>\n", os.Args[0])
        os.Exit(10)
    }

    // [first block] set up 2600 emulation here

    err = filepath.Walk(os.Args[1],
        func(path string, info os.FileInfo, err error) error {
            if err != nil {
                return err
            }

            if info.IsDir() {
                return nil
            }

            // [second block] we'll put the gopher2600 code here

            return nil
        })

    if err != nil {
        fmt.Println(err)
        os.Exit(10)
    }
}
```

The comments indicate where we going to insert the `Gopher2600` specific code. The *first block* sets up the emulation:

```
// new television with auto-selecting tv protocl
tv, err := television.NewTelevision("AUTO")
```

```
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
defer tv.End()

// new VCS
vcs, err := hardware.NewVCS(tv)
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
```

We first create a new telelvision using the `AUTO` flag. This flag tells the television that we want it to switch from `NTSC` to `PAL` if it looks like the cartridge is intended for that type of television. The detection process for this isn't perfect however so we could also specify `NTSC` or `PAL` to indicate that we definitely want that type of TV and no detection should take place.

After the TV has been created we can set up the VCS emulation. Note that creating an instance of the emulation does not start the emulation. We'll do that below.

The *second block* is to be placed inside the payload function of the `filepath.Walk()` function. We'll break this down as we go:

```
// load and attach cartridge
cartload := cartridgeloader.Loader{Filename: path}
if err := vcs.AttachCartridge(cartload); err != nil {

    // ignore known cartridge errors
    if errors.Is(err, errors.CartridgeError) {
        return nil
    }

    // any other errors are probably serious
    return err
}
```

The first part creates a new `cartridgeloader.Loader` using the `path` variable supplied by the `Walk` function. This is then attached to the `VCS` instance created earlier and errors checked.

We can also see the error system used in `Gopher2600`. The `Is()` function can be used to check for specific errors, in this case we want to filter all errors of type `CartridgeError`. For simplicity in this example, we don't do anything useful with this knowledge, we just return `nil` and carry and allow the `Walk()`

function to continue.

```
// reset VCS after previous iteration
if err = vcs.Reset(); err != nil {
    return err
}
```

We must reset the emulation after every use.

```
// run for 10 frames
err = vcs.Run(func() (bool, error) {
    fr, _ := tv.GetState(television.ReqFramenum)

    if fr > 10 {
        return false, nil
    }

    return true, nil
})
```

The call to the `vcs.Run()`requires as an argument, a function that states whether the emulation should continue. In our auditing example, we want the emulation to run for 10 television frames. This should be enough to get the information we want.

The `Television` type has a function called `GetState()` and we use it here to get the number of elapsed frames. There are a number of other *requests* we cam make to this function and they are listed in the documentation of the `television` package.

```
// collect any errors
errmsg := ""
if err != nil {

    // except any non Gopher2600 specific errors
    if !errors.IsAny(err) {
        return err
    }

    errmsg = err.Error()
}
```

Any error from `vcs.Run()` we will keep and use as part of the audit output.

The last thing to do in this block is to print out the results for the current cartridge.

```
// get short version of cartridge name
name := cartload.ShortName()
if len(name) > 30 {
```

```
    name = name[:30]
}
```

We don't want to print the full path of the cartridge, just the filename. The `ShortName()` function in the `cartridgeloader.Loader` type does the transformation for us.

```
// print table row
fmt.Printf("%30s | %6s | %4s | %v\n", name,
    vcs.Mem.Cart.Format(),
    tv.GetSpec().ID, errmsg)
```

Finally, we print out an audit report for the cartridge. The columns are:

- Short cartridge name
- Detected cartridge format
- Detected TV type
- Errors (from the emulation)

Finally, we should go back and add a table header. We'll place this at the end of the *first block*.

```
// table header
fmt.Printf("%s-+-%s-+-%s-+-%v\n",
    strings.Repeat("-", 30), strings.Repeat("-", 6),
    strings.Repeat("-", 4), strings.Repeat("-", 30))
fmt.Printf("%30s | %6s | %4s | %v\n", "Name", "Format", "TV", "Errors")
fmt.Printf("%s-+-%s-+-%s-+-%v\n",
    strings.Repeat("-", 30), strings.Repeat("-", 6),
    strings.Repeat("-", 4), strings.Repeat("-", 30))
```

Example: Extending With AudioMixer

The audit example works well but we can push it even further. It might be useful to know if a cartridge makes a sound within the first second of operation. To do this we can attach an `AudioMixer` to the emulation and monitor the results.

```
type myAudioMixer struct {
    active bool
}

func (mix *myAudioMixer) SetAudio(data uint8) error {
    mix.active = data > 0
    return nil
}

func (mix *myAudioMixer) EndMixing() error {
    mix.active = false
```

```
    return nil
}
```

`myAudioMixer` implements the `AudioMixer` interface. This is a very simple implementation but that is all we need. We must all *add* an instance of this mixer to the `television`. In our audit program we insert the following just after we initialise the TV.

```
// add audio mixer to television
mix := &myAudioMixer{}
tv.AddAudioMixer(mix)
```

Finally, we should extend the number of frames the emulation runs for before stopping. We can do this by changing the condition in the `Run()` function. A value of sixty frames would be good (60 frames = approximately 1 second)

```
if fr > 60 {
    return false, nil
}
```

And that's it! We should also add columns to our audit table to see the result of the `active` field, and we also need to reset the `active` before emulating a new cartridge (we do this by calling `EndMixing()` at the end of the `Walk()` function) but in principle, that is all there is to it.

Example: Conclusion

Rather than extending the example any further it is sufficient to point out that adding a `PixelRenderer` or an input `Playback` instance follows the same principles shown above. That is:

- Create a Television
- (optional) Add PixelRenderer or AudioMixer
- Create a VCS
- (optional) Attach `Playback` and `EventRecorder` devices
- Run VCS with callback function returning halt condition