Introduction

This page documents some less well-known (perhaps advanced) tricks for the `gc` toolchain (and the Go tool).

C code without `cgo`

Use `syso` file to embed arbitrary self-contained C code

Basically, you write your assembly language in GNU as(1) format, but make sure all the interface functions are using Go's ABI (everything on stack, etc., please read Go 1.2 Assembler Introduction for more details).

The most important step is compiling that file to file.syso (`gcc -c -O3 -o file.syso file.S`), and put the resulting syso in the package source directory. And then, suppose your assembly function is named Func, you need one stub cmd/asm assembly file to call it:

```
TEXT ·Func(SB),$0-8 // please set the correct parameter size (8) here
    JMP Func(SB)
```

then you just declare Func in your package and use it, go build will be able to pick up the syso and link it into the package.

Notes:

- The binary produced won't use cgo, and the overhead is just an unconditional JMP that could be perfectly branch predicted. But, please be aware that because it doesn't use cgo, your assembly function is running on Go stack, and it **shouldn't use too much stack** (a safe value is less than ~100 bytes) or terrible things will happen. For compute kernels, this requirement isn't too restricting.
- Please make sure you've included all library dependencies in your C code. `libc` is not available, and most notably, `libgcc` is also not available (esp. when you're using gcc `__builtin_funcs` , please use `nm(1)` to double-check that your file doesn't contain any undefined symbols).
- It's also possible to call back Go functions from C code, but this is left as an exercise for the reader.
- this trick is supported on all Go 1.x releases.
- the Go linker is pretty capable in that you just need to prepare .syso file for each architecture, not for each OS/Arch combination (assuming you don't use OS-specific constructs, obviously), and the Go linker is perfectly capable to link, for example, Mach-O object files into ELF binaries. So be sure to name your syso file with names like `file_amd64.syso` , `file_386.syso` .

Bundle data into Go binary

There are a lot of ways to bundle data in Go binary, for example:

- `zip` the data files, and append the zip file to end of Go binary, then use `zip -A prog` to adjust the bundled zip header. You can use `archive/zip` to open the program as a zip file, and access its contents easily. There are existing packages that helps with this, for example, https://pkg.go.dev/bitbucket.org/tebeka/nrsc; This requires post-processing the program binary, which is not suitable for non-main packages that require static data. Also, you must collect all data files into one zip file, which means that it's impossible to use multiple packages that utilize this method.
- Embed the binary file as a `string` or `[]byte` in Go program. This method is not recommended, not only because the generated Go source file is much larger than the binary files themselves, also because

static large `[]byte` slows down the compilation of the package and the `gc` compiler uses a lot of memory to compile it (this is a known bug of `gc`). For example, see the tools/godoc/static package.

- use similar `syso` technique to bundle the data. Precompile the data file as syso file using GNU `as(1)`'s `.incbin` pseudo-instruction.

The key trick for the 3rd alternative is that the linker for the `gc` toolchain has the ability to link COFF object files of a different architecture into the binary without problem, so you don't need to provide syso files for all supported architectures. As long as the syso file doesn't contain instructions, you can just use one to embed the data.

The assembly template to generate the COFF .syso file:

```
/* data.S, as -o data.syso */
.section .rdata,"dr" /* put in COFF section .rdata */
.globl _bindataA /* no longer need to prepend package name here */
.globl _ebindataA
_bindataA:
.incbin "dataA"
_ebindataA:

.globl _bindataB /* no longer need to prepend package name here */
.globl _ebindataB
_bindataB:
.incbin "dataB"
_ebindataB:
```

And two other files, first a Plan 9 C source file that assembles the slice for Go:

```
/* slice.c */
#include "runtime.h"
extern byte _bindataA[], _bindataB[], _ebindataA, _ebindataB;

void ·getDataSlices(Slice a, Slice b) {
  a.array = _bindataA;
  a.len = a.cap = &_ebindataA - _bindataA;
  b.array = _bindataB;
  b.len = b.cap = &_ebindataB - _bindataB;
  FLUSH(&a);
  FLUSH(&b);
}
```

And finally, the Go file that uses the embedded slide:

```
/* data.go */
package bindata

func getDataSlices() ([]byte, []byte) // defined in slice.c

var A, B = getDataSlices()
```

Note: you will need an `as(1)` capable of generating the COFF syso file, you can build one easily on Unix:

```
wget http://ftp.gnu.org/gnu/binutils/binutils-2.22.tar.bz2   # any newer version also
works
tar xf binutils-2.22.tar.bz2
cd binutils-2.22
mkdir build; cd build
../configure --target=i386-foo-pe --enable-ld=no --enable-gold=no
make
# use gas/as-new to assemble your data.S
# all the other file could be discarded.
```

Drawback of this issue is that it seems incompatible to cgo, so only use it when you don't use cgo, at least for now. I (minux) is working on figuring out why they're incompatible.

Including build information in the executable

The gc toolchain linker, cmd/link, provides a `-X` option that may be used to record arbitrary information in a Go string variable at link time. The format is `-X importpath.name=val`. Here `importpath` is the name used in an import statement for the package (or `main` for the main package), `name` is the name of the string variable defined in the package, and `val` is the string you want to set that variable to. When using the go tool, use its `-ldflags` option to pass the `-X` option to the linker.

Let's suppose this file is part of the package `company/buildinfo`:

```
package buildinfo

var BuildTime string
```

You can build the program using this package using `go build -ldflags="-X 'company/buildinfo.BuildTime=$(date)'"` to record the build time in the string. (The use of `$(date)` assumes you are using a Unix-style shell.)

The string variable must exist, it must be a variable, not a constant, and its value must not be initialized by a function call. There is no warning for using the wrong name in the `-X` option. You can often find the name to use by running `go tool nm` on the program, but that will fail if the package name has any non-ASCII characters, or a `"` or `%` character.