Zephyr® Project
Developer Summit 2022

Linkage in the Third Millenium

Ideas, Innovations, and Experiments
for Flexible RTOS Builds on Modern Devices

Andy Ross, Intel

andy@plausible.org
github.com/andyross

What does a Linker Do, Anyway?

Historically obscure, but not actually complicated

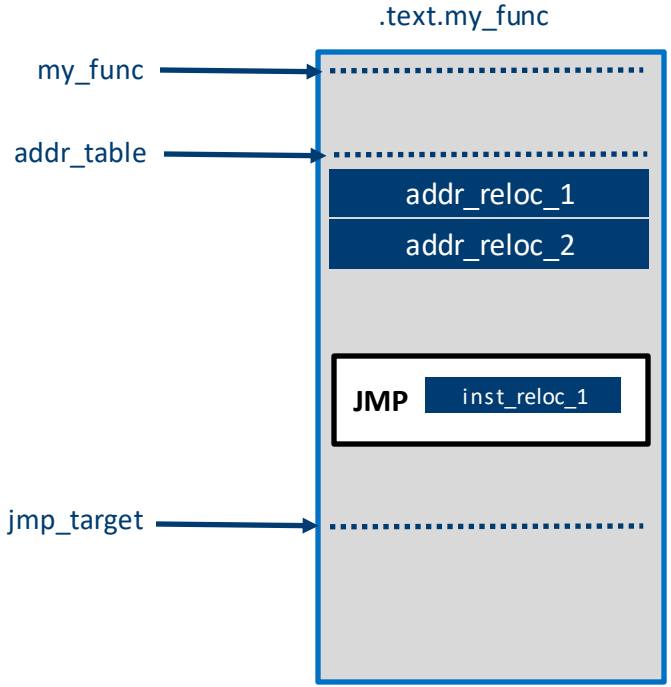
- Code references other code
- We don't know the addresses yet when generating binary output
- Linkers assign addresses in already-compiled binaries

And that's all linkers do!

- Well, all they're supposed to do...
- Zephyr plays tricks

Linking happens in "chunks"

- Cannot be subdivided
- Has a name ("section")
  - Not unique!
- Associated with an object file
- Has multiple (!) "symbols"
  - Just named byte offsets
  - Can be global or "local" (file scope)
- Has "relocations" inside
  - Place where symbol addresses go
  - Arch-specific formats!
    - Obscure, largely undocumented
- Other metadata
  - Runtime usage/permissions (r/w/x)
  - Alignment
  - Present at runtime (SHF_ALLOC)



Linking Algorithm

- Trace dependencies between input chunks
  - Requires "bootstrap" symbols
    - Interrupt tables, boot entry vector...
  - Symbol visibility rules are ad-hoc, historically weird
    - "Local" variables have "object file" scope (not source file)
    - Files in static libraries presented to linker only if referenced
    - COMDAT rules (".bss data with same name gets unified")
    - "weak" symbols, .gnu_linkonce, …
    - Maybe put some of these rules into the syntax and not the assumptions?
- Assign remaining chunks to addresses
- Compute relocations and modify chunks accordingly

"Link Editor" as language

1970's tool, System III Unix

- Declarative idiom
  - Sorta, address assignment is sequential
- Minimal error reporting
  - Multi-assign output addresses
  - Leave 512MB gap to be padded
  - Clobber a chunk-defined symbol
  - Forget an output section
  - All OK with the linker!
- First just linked one output blob
  - Later grew multiple named regions
    - `"… } >REGION"`
  - …then ELF program headers
    - `"… } >REGION :PHDR"`
  - …and separate phys/virt addresses
    - `"… } >REGION :PHDR AT>OTHER"`

Odd chunk selection syntax

```
KEEP(*(SORT_BY_NAME(.my_secname
                    .my_secname.*)));
```

FILE(SECTION_WILDCARDS)

- Why are files even there?
- "ld" used to do what "make" did later

Note double selection idiom

- gcc –f{function,data}-sections convention
- But not used on all output
- Can't be expressed as a wildcard!

"Sort" operation only works on name

- Linker has no other usable chunk state

What does Zephyr ask a Linker to do?

(Outrageous Linker Tricks for Fun and Profit!)

Assemble collections of static data

- Put each item in a named section
- Find (and KEEP( )!) all instances
  - Still doesn't find library data
  - Unless file otherwise used
  - Used to "hide" data (not kconfig?!)
- Link sequentially
- Put a symbol at start/end
- Alignment bugs
  - Linker doesn't know C alignment
  - Array-of-chunk not array-of-struct
- Often handled by ITERABLE api

```
/* file1.c */
static struct foo
  __attribute__((__section__(".foos")))
  my_foo = {...};
```

```
/* file2.c */
static struct foo
  __attribute__((__section__(".foos")))
  my_foo = {...};
```

```
/* foo.ld */
.foos {
    _foo_start = .;
    KEEP(*(".foos"));
    _foo_end = .;
} >ROM
```

```
/* foo-runtime.c */
extern struct foo _foo_start;
extern struct foo _foo_end;
for (struct foo *f = &_foo_start; f < &_foo_end; f++)
{
    initialize_foo(f);
}
```

Linker Tricks: Execute in Place (XIP)

- XIP indicates full image is linked into addressable ROM
- **Really** means .data must be moved to RAM
- Link in .rodata, emit boundaries for copy to RAM
- Use ld "AT>" syntax
  - But alignment can get skewed between the two!
  - ("." variable only affects main, not "AT" section)
  - My first Really Hard Zephyr bug
- Would be nicer to have a way to link where it belongs
  - Then just "add more chunks" to .rodata mid-link
  - Linker can't do that (declarative)

Linker Tricks: Multi-pass Linkage

- Generate intermediate data via "linkage"
- Lists of / IDs for referenced objects
  - ISR tables
  - Kernel object hash table
    - (based on DWARF debug type info)
  - Thread index generation
  - DTS device handle numbering
- All together, as many as three (!!!) passes
- But in all cases, really nothing more than garbage collection
  - See what's included, adjust/add/generate, link again
  - Would be nice to "pause" the link at this spot...

Linker Tricks: Static Libraries

Some subsystems are .a archives

- Why? Only linking once (thrice…)
- No need for fast symbol lookup

Actually a trick for dependency mgmt!

- Normally linker sees all input chunks
- Libraries only include chunks referenced
  - c.f. -Wl,--whole-archive
- So earlier tricks won't "see" data in .a
- Prevents generation of unused hardware

Not really the Right Thing

- Wasted work (a **fourth** linker pass!)
- Confusing
- Often needless
  - CCSD: Cargo cult software dev
- Supposed to be kconfig's job

What if the linker could manage that?

- "Link only if CONFIG_X"
- "Link only if $symbol is referenced"
- "$sym1 should be required by $sym2"
  - (Reverse dependency)

Linker Tricks: Xtensa Cache Trickery

Xtensa has incoherent L1 cache

- Memory mapped twice: cached/uncached
- Use linker to put symbols in correct region
  - .text, .rodata cached
  - .data, .bss uncached
  - Thread stacks cached (always "local")
- Assign to "." to bounce back and forth
  - Error prone (esp. when #include <x.ld>)
  - Must pad to cache line!
  - Leaves big holes in output
    - (512MB if you get the wrong PHDR!)
  - Requires complicated objcopy reassembly
    - ANOTHER "linker" pass!

Would be nice to stitch up mid-link...

```
SECTIONS {
  . = . + 0x20000000;
  . = ALIGN(64);
  .text { *(.text .text.*) }>CACHED :CACHED_PHDR

  . = . - 0x20000000;
  . = ALIGN(64);
  .data { *(.data .data.*) }>UNCACHED :UNCACHED_PHDR

  . = . + 0x20000000;
  . = ALIGN(64);
  .rodata { *(.rodata .rodata.*) }>CACHED :CACHED_PHDR
}
```

Linker Tricks: Endless!

Devicetree Handles

- Clever "smaller than pointers" representation of device graph

Dictionary Logging

- Format strings not present at runtime
- Stores string ID + arguments instead
- Strings go in separate linker section, w/ ID generation

System call marshalling generator

Arch-specific stuff:

- Auxilliary memory spaces (bootstrap environments, EFI...)
- Metadata for signing tools (rimage)
- Boundaries & offsets for boot-time initialization/copies
- Generation of output formats
  - Some simple (Cortex M), some complex (intel_adsp/rimage)
  - Mix of objcopy, scripts, signing tools
  - All has to start from zephyr.elf, can't use data known to linker

A Better Way?

Write our own linker? Are you serious?

- It's complicated! (Same job was done on a PDP-11)
- It's slow! (We already link 3+ times!)
- It's the toolchain's job!
  - Fair point. Arch layer is very poorly specified
  - Still, even complicated archs (x86, xtensa) aren't so bad

What might it look like?

- Python, standard Zephyr extension language
- "Linker scripts" as modules
  - Can be defined per-subsystem, per-arch, etc...
- Split "dependency" step from "relocate" (eliminates multi-pass?)
- Can infer much more from context/convention
  - No more preprocessor boilerplate madness

A Better Way: Pseudoexamples

Some examples of the way these tasks become much simpler in a robust, imperative environment.

These are not real APIs yet! Just examples of the kind of linkage interface I'd like to see.

```
# Kconfig integration to replace the
# static library trick: mark symbols by
# section such that they require a given
# kconfig, and eject from the link in an
# early pass over input data
for c in all_chunks():
    m = re.match(r"CONFIG_(.*)$", c.section())
    if m:
        if not kconfig_defined(m.group(0)):
            remove_chunk(c)
```

```
# XIP handling: mid-link, relocate and
# copy .data into new blob for .rodata.
# Note that get_bytes() cat lazy-evaluate
# relocation and then warn if any symbols
# so-relocated ever get changed.
blob = find_sect(.data).get_bytes()
c = find_sect(".rodata").add_blob(blob)
new_sym("xip_start", blob.start())
new_sym("xip_end", blob.end())
```

Generate tables of ISRs using struct.pack()!

Enumerate over kobject types before the link!

All the objcopy/binutils usage is now Python IO!

Mix architecture object files without hacking headers!

```
# Automatically generate an interable list of "XXX"
# from all objects in sections named "XXX.iterable"
for s in find_sects("*.iterable"):
    blob = new_blob()
    new_sym(f"{s.name}_start", blob.start())
    for c in sect_chunks(s):
        blob.add_chunk(c)
        # Can also validate all have same size/alignment!
    new_sym(f"{s.name}_start", blob.end())
```

```
# Xtensa cache linkage part 1: Find all
# "incoherent" chunks and make sure they
# are cache-line aligned and padded.
# Can also warn if any globally-visible
# symbols from non-aligned addresses
# exist within the chunk.
for c in all_chunks():
    m = re.match(r"\.incoherent(\..*)?$", c.section())
    if m:
        c.set_align(max(c.get_align(), 64))
        c.pad_size_to(64)

# Xtensa cache linkage part 2: After
# packing but before relocation, adjust
# symbol addresses in all incoherent
# chunks
for c in all_chunks():
    for s in c.syms():
        s.set_addr(s.addr() + 0x20000000)
```

Advantages

Much more robust error checking

- Duplicate/undefined symbols, sure, also...
- Multiple weak definitions
- .gnu_linkonce definitions that don't match exactly (common C++ goof)
- COMDAT can become a warning condition (common mixup to forget initializer/static)
- Orphan symbols (enabled driver/subsys but missing DTS? Typo in section?)
- Warn on all-zero static initializer (currently an "optimization" to .bss that causes bugs)

Futureproof validation

- Check that there are no accidental relocations from "boot" code into uninitialized mem (uncleared .bss, uncopied XIP .data)

Faster, probably

- Yes, python is slower
- But linking at our scale is mostly an I/O bound single pass
- And we do it three times!

Compatibility

Migration: can't just drop binutils ld instantly

- Many (most) platforms have and maintain their own linker tricks
- Some projects may require specific tooling
- Some toolchains may have proprietary linker features we can't duplicate
  - (None do currently)

Fallback solution #1: maintain both schemes

- Expensive long term, likely to evolve badly as new features don't port cleanly
- Requires adherence to same e.g. section naming conventions
- Do this to start, obviously

Solution #2: drop binutils ld

- Make Andy happy.
- Guaranteed this is a showstopper for someone

Compatibility...

Solution #3: Zephyr as "ld preprocessor"

- We know the dependency graph
- We already did all the intermediate/derived symbol generation
- Spit out ELF output (maybe just one file) with exactly the chunks we need
- Emit matching toolchain linker script with minimal content
- Add platform layer stuff via #include

Not as clean as doing it all ourselves, but not bad.

Also works as an intermediate step to...

Link Time Optimization

New hotness in the world of open source toolchains!

- GCC and clang/LLVM competing robustly
- Still bleeding edge: a few binary-delivering projects, no full Linux distros yet
- Does a lot of what Zephyr's heavily preprocessed trickery does, for everything

Basic idea is simple: "chunks" contain syntax trees, not binary code

- Linker does first pass with only dependency analysis
- Linker presents remaining chunks back to compiler
  - GCC owns this process, LLVM provides a "toolkit" API
- Compiler looks at all code again, does global optimizations, emits final chunk
- Linker does final relocations on the compiler output chunk

Wait! Isn't that exactly the scheme we're talking about?

Link Time Optimization...

Our scheme works great here:

- We do dependency analysis, which we have to do anyway
  - Also means that classic linkage is a **very poor** fit for LTO!
  - Zephyr needs to stop and re-link, but LTO needs to start over there too
- We stop before doing relocation
  - Again, this is a place we need to run code, so easy
- We present the "only needed chunks" output back to the compiler
  - This is the same single ELF blob we need for compatibility linking!
- The compiler gives us back an optimized single chunk we relocate

Still lots of decisions to make (not a v1 feature for sure):

- We have multiple .text spaces. Is it OK to cross-space optimize? Sometimes?
  - Can cross-inline, but can't in general reference the same data values
  - One side might see a constant, the other a variable?