Higher-Level Linker Scripts for Embedded Systems

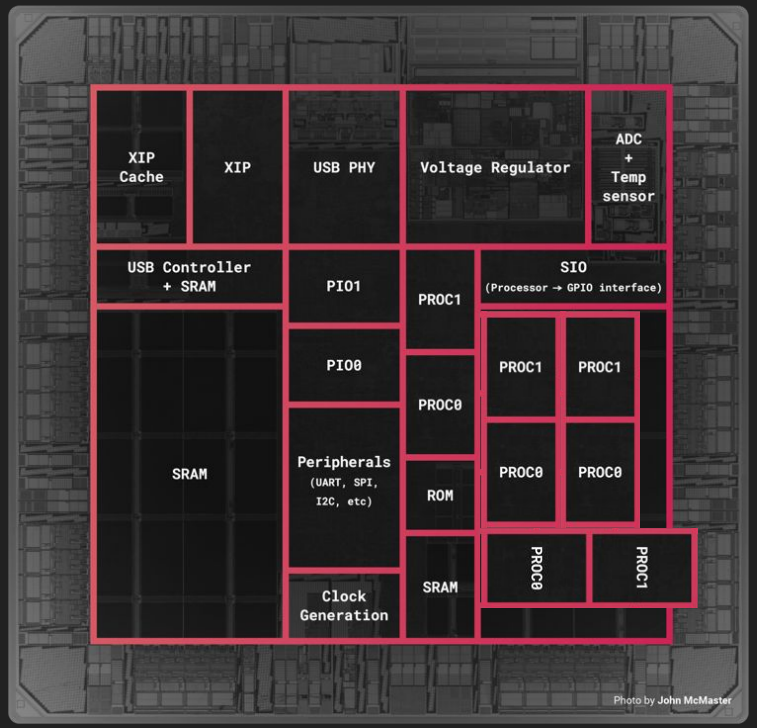
Daniel Thornburgh
Google

Embedded Systems

- Sensitive to...
  - Cost
  - Power
  - Latency
- Special purpose code

SRAM Is Expensive.

264KiB Total!



Heterogeneous Memory

There's no MMU, since an MMU requires an SRAM TLB cache.

Cheaper memory off-die can replace expensive SRAM.

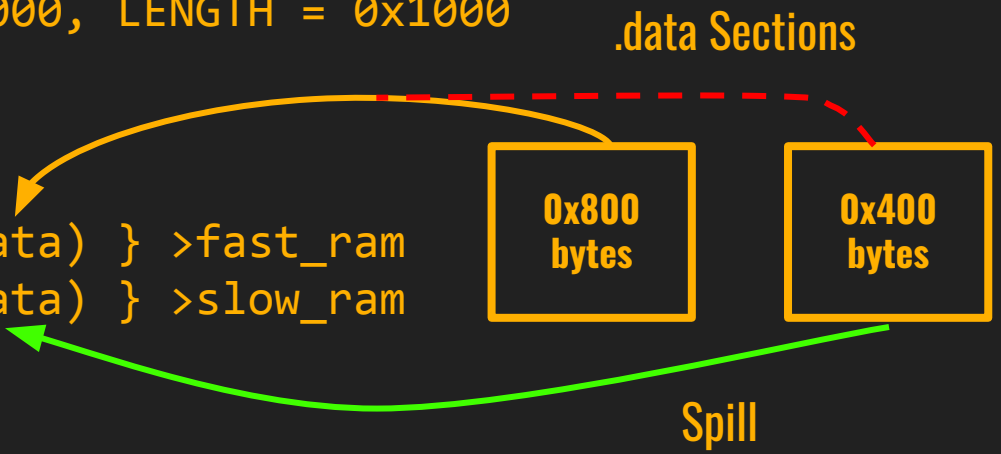
Off-die memory may be read-only, slow, cleared on suspend, unavailable on boot, or have bus contention.

The linker must deal with it!

Discontinuous Memory Linker Script?

```
MEMORY {
  fast_ram : ORIGIN = 0x1000, LENGTH = 0x1000
  slow_ram : ORIGIN = 0x3000, LENGTH = 0x1000
}

SECTIONS {
  .data_fast_ram : { *(.data) } >fast_ram
  .data_slow_ram : { *(.data) } >slow_ram
}
```

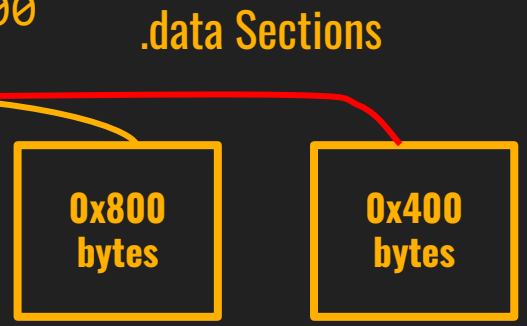


Code needs similar treatment.

Discontinuous Memory Linker Script? Nope!

```
MEMORY {
  fast_ram : ORIGIN = 0x1000, LENGTH = 0x1000
  slow_ram : ORIGIN = 0x3000, LENGTH = 0x1000
}

SECTIONS {
  .data_fast_ram : { *(.data) } >fast_ram
  .data_slow_ram : { *(.data) } >slow_ram
}
```



Sections are **always** assigned to the first match.

ERROR: Overflow

Solution 1: Manual Assignment AKA Toil

```
MEMORY {
  fast_ram : ORIGIN = 0x1000, LENGTH = 0x1000
  slow_ram : ORIGIN = 0x3000, LENGTH = 0x1000
}

SECTIONS {
  .data_fast_ram : {
    file1.o(.data)
    file2.o(.data)
    ...
  } >fast_ram
  .data_slow_ram : {*(.data) } >slow_ram
}
```

Let's play Tetris with our codebase (with blocks that change sizes). Tiny Memory, Difficult Problem.

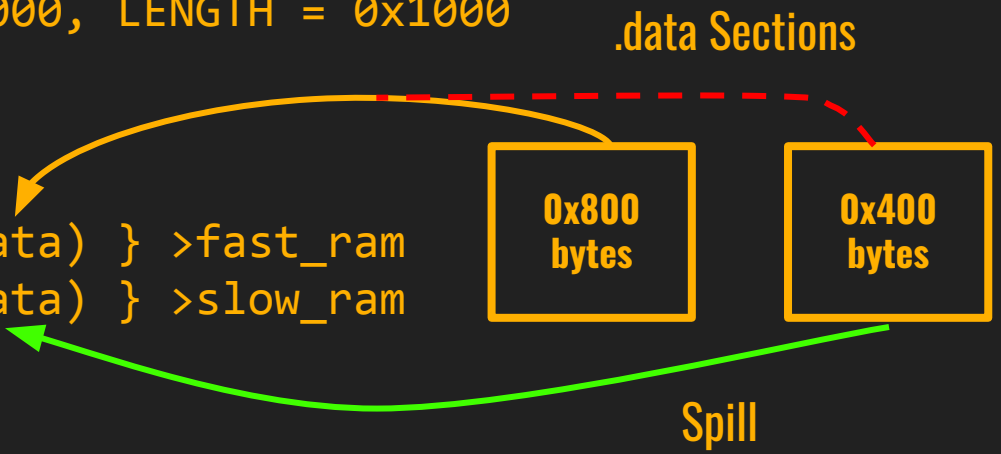
Computers are supposed to *free us* from this. Some people generate linker scripts.

Presently incompatible with Full LTO

Solution 2: `--enable-non-contiguous-regions`

```
MEMORY {
  fast_ram : ORIGIN = 0x1000, LENGTH = 0x1000
  slow_ram : ORIGIN = 0x3000, LENGTH = 0x1000
}

SECTIONS {
  .data_fast_ram : { *(.data) } >fast_ram
  .data_slow_ram : { *(.data) } >slow_ram
}
```



This is just that slide from before.

Solution 2: `--enable-non-contiguous-regions`

Originally from GNU LD, but we recently ported it to LLD.

Sections spill to later matches if they won't fit.

Unblocks Full LTO, since scripts can be written without filenames.

Downside: Globally changes linker script semantics. This can break existing scripts.

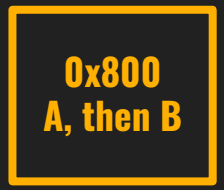
Solution 3: Section Classes

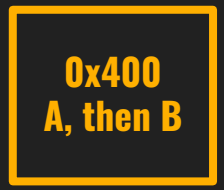
```
MEMORY {
  fast_ram : ORIGIN = 0x1000, LENGTH = 0x1000
  slow_ram : ORIGIN = 0x3000, LENGTH = 0x1000
}

SECTIONS {
  CLASS(data) : { *(.data) }
  .data_fast_ram : { CLASS(data) *(.other) } >fast_ram
  .data_slow_ram : { CLASS(data) } >slow_ram
}
```

This is almost, but not quite, that slide from before.

Spilling Logic

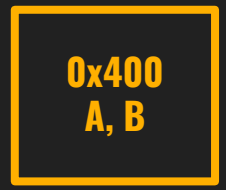




Memory A (0x1000):

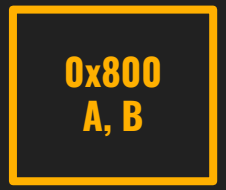
Memory B (0x1000):

Spilling Logic



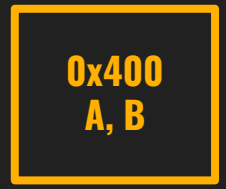
Overflow

Memory A (0x1000):



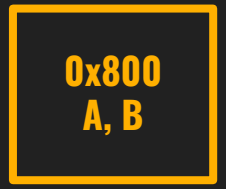
Memory B (0x1000):

Spilling Logic



Memory A (0x1000):

Memory B (0x1000):



Spill at least 0x1200 - 0x1000 = 0x200 bytes

Spilling Logic

Memory A (0x1000):

0x400 A	0x400 A, B

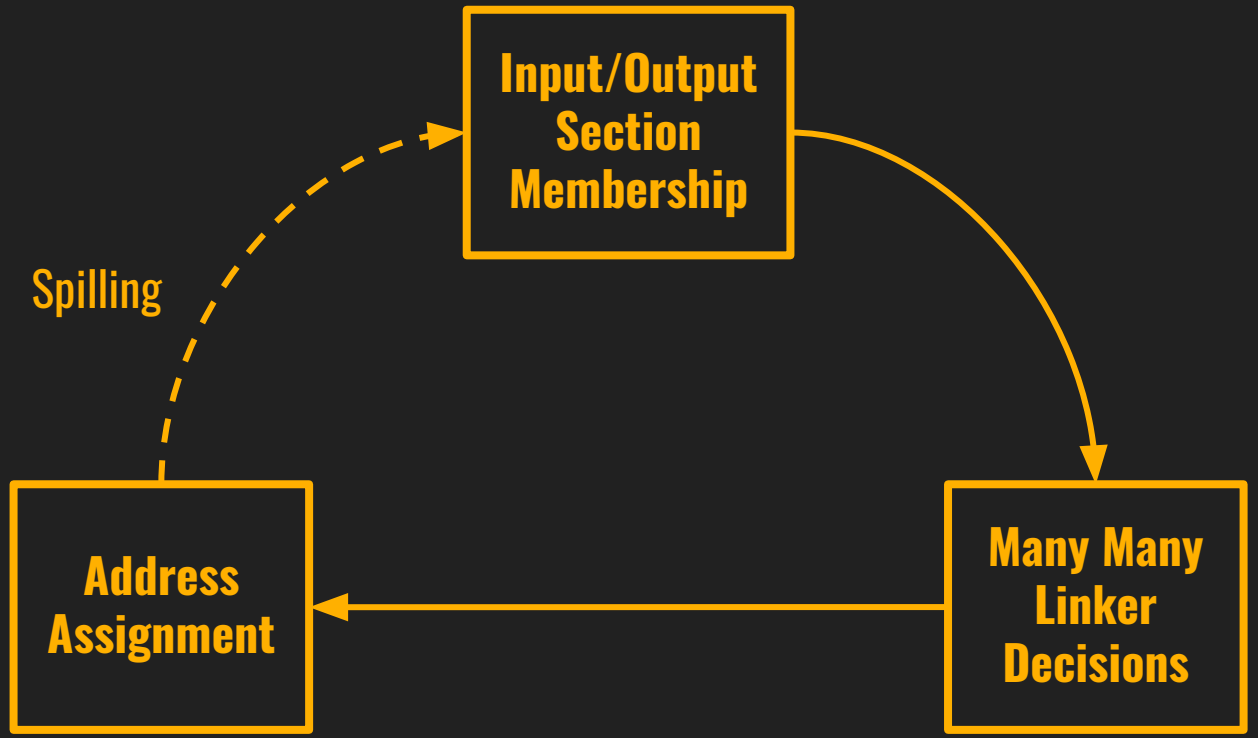
Memory B (0x1000):

0x800 A, B

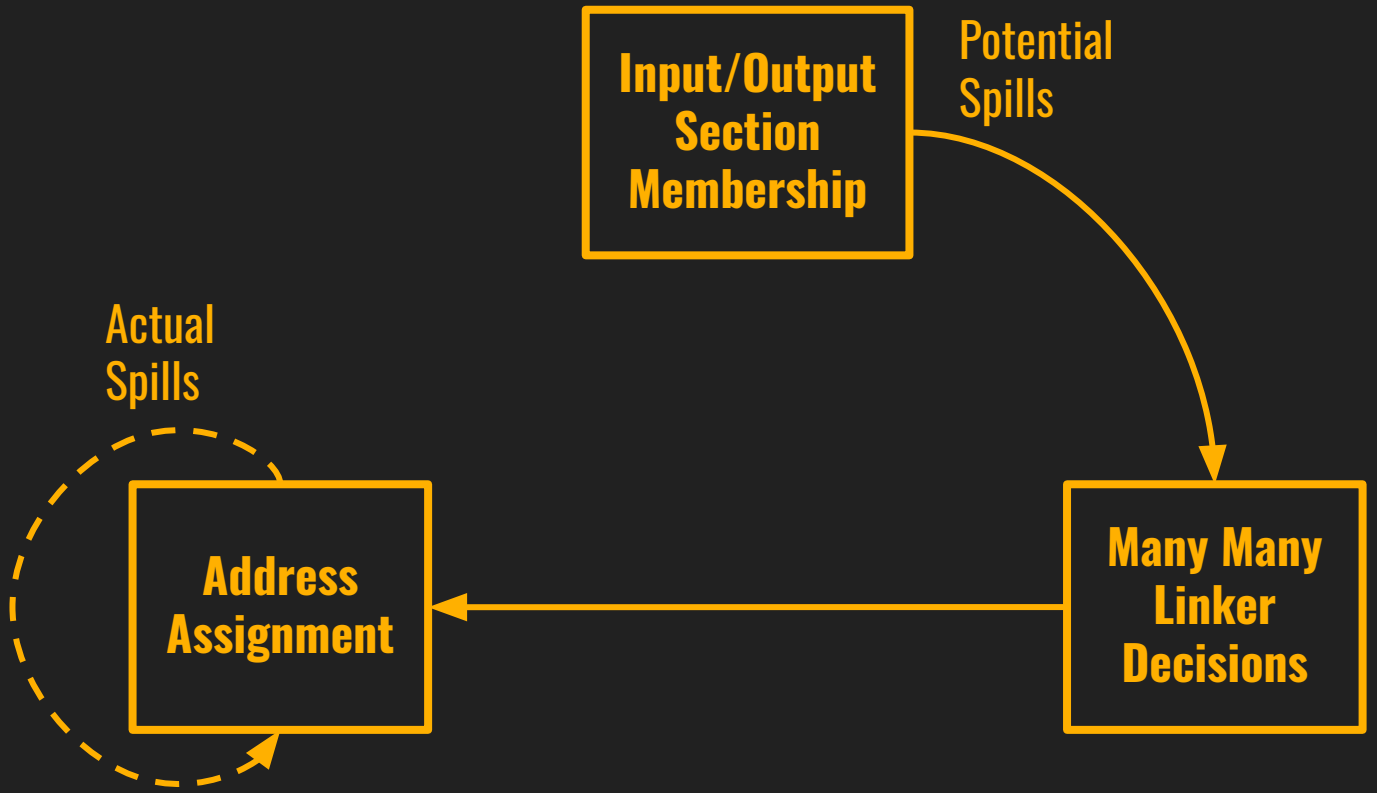
Feature Interactions

- `/DISCARD/`
- `SHF_MERGE` Section Merging (e.g. strings)
- `ONLY_IF_RO` / `ONLY_IF_RW`
- Output Section Alignment
- Identical Code Folding (ICF)
- `SHF_LINK_ORDER`
- `INSERT AFTER` / `INSERT BEFORE`
- `OVERWRITE_SECTIONS`

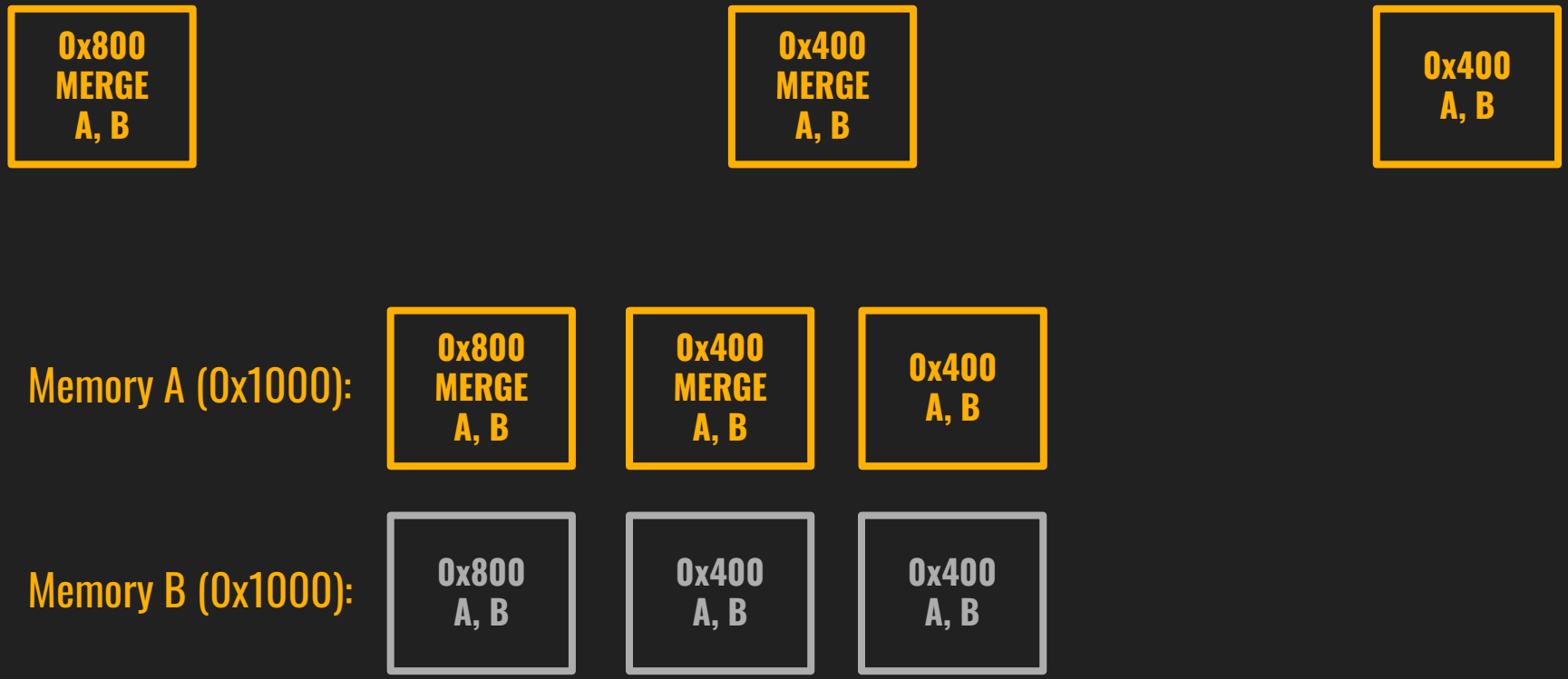
Horrible Circular Dependency



Circular Dependency Resolution



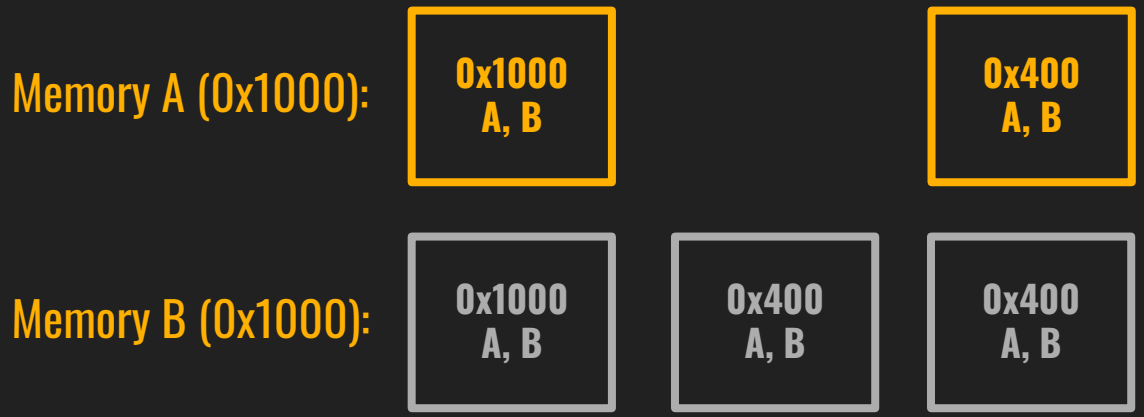
Potential Spills



Linker Decisions (e.g. SHF_MERGE)

Potential spills manipulated as if they were regular sections... for the most part.

Ensures that spill locations are always okay to move sections to.

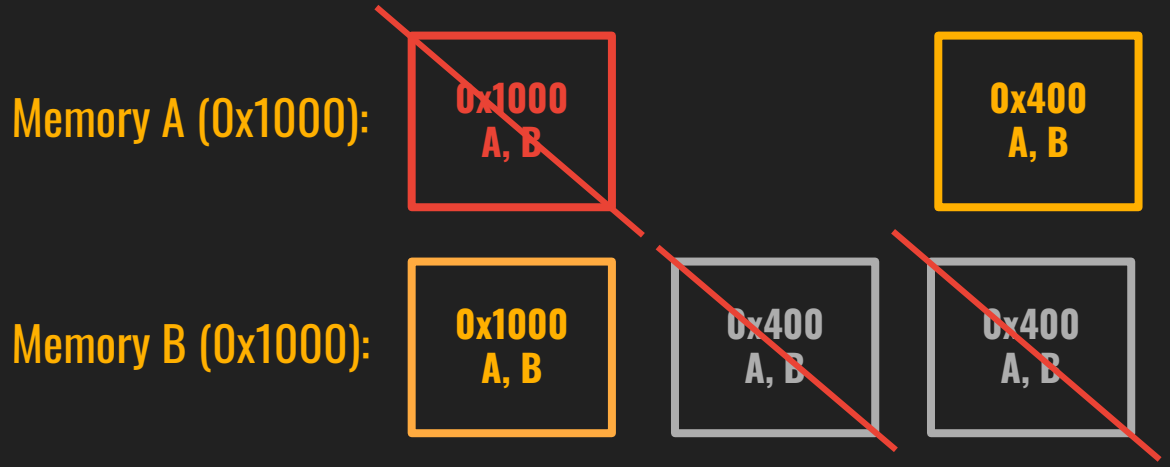


Actual Spills

Replace a potential spill with the original section.

The context was prepared for this by the potential spill.

Delete unused potential spills.



Future Work: Priority Ordering

Sections are placed first-fit but the order of sections and the order of regions is arbitrary.

Order memory regions best-first and sections by decreasing importance.

First-fit greedily optimizes performance.

Propeller? Use profiles to group code/data into sections with priorities?

How is priority information communicated to the linker?

How do priorities interact with e.g. ordering for thunk minimization?

Thanks for Listening! Questions?

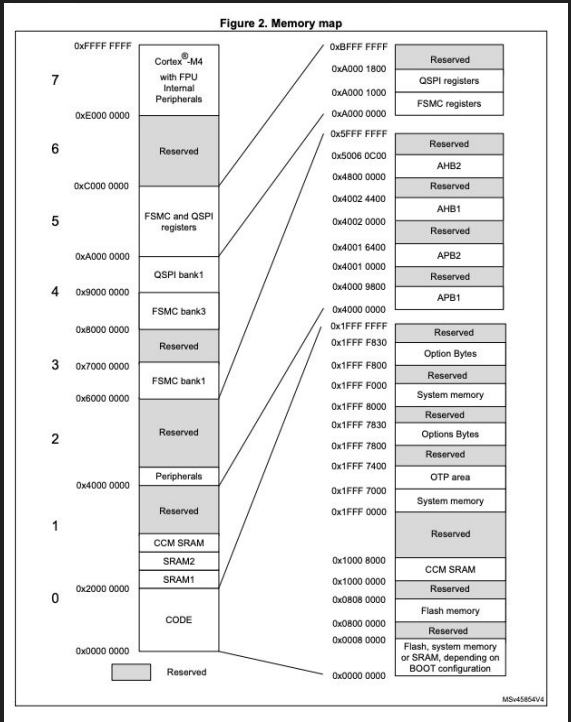
LLD Section Class Documentation

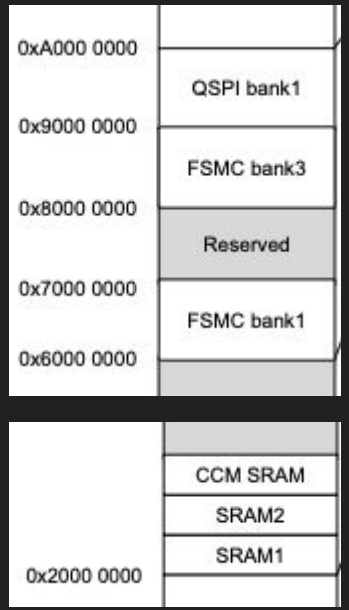
LLD `--enable-non-contiguous-regions` Documentation

LLD Linker Section Packing RFC

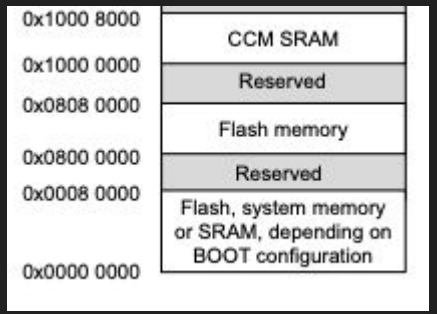
LLD `--enable-non-contiguous-regions` Implementation RFC

STM32G4 Datasheet (ARM Cortex M0+ MCU)





More RAM?



No MMU, huh? Not even a tiny one?

Microcontrollers typically have as much MMU as they can afford.

A Memory Protection Unit (MPU) provides a small number (e.g. 8) of segments.

These have protection but not remapping.

Remapping is just an add, so why's it missing?

Eight manual segments isn't very good. Even a Motorola 68451 MMU had 32.

Very many memory operations would fault and require the OS to update the segment table.