



Bringing link-time optimization to the embedded world: (Thin)LTO with Linker Scripts

Tobias Edler von Koch, Sergei Larin,
Shankar Easwaran, Hemant Kulkarni



What is a linker script?

A linker script allows the user to describe how sections in the input files should be mapped into an output file.

The mapping between input and output sections is expressed using patterns

The patterns are matched against input file paths and input section names.

Linker scripts facilitate key features of system-level software

Some examples are

- Tightly Coupled Memories (TCM)
- RAM / ROM placement
- Compression

Agenda

1

Motivating
Example

2

What we need
to address

3

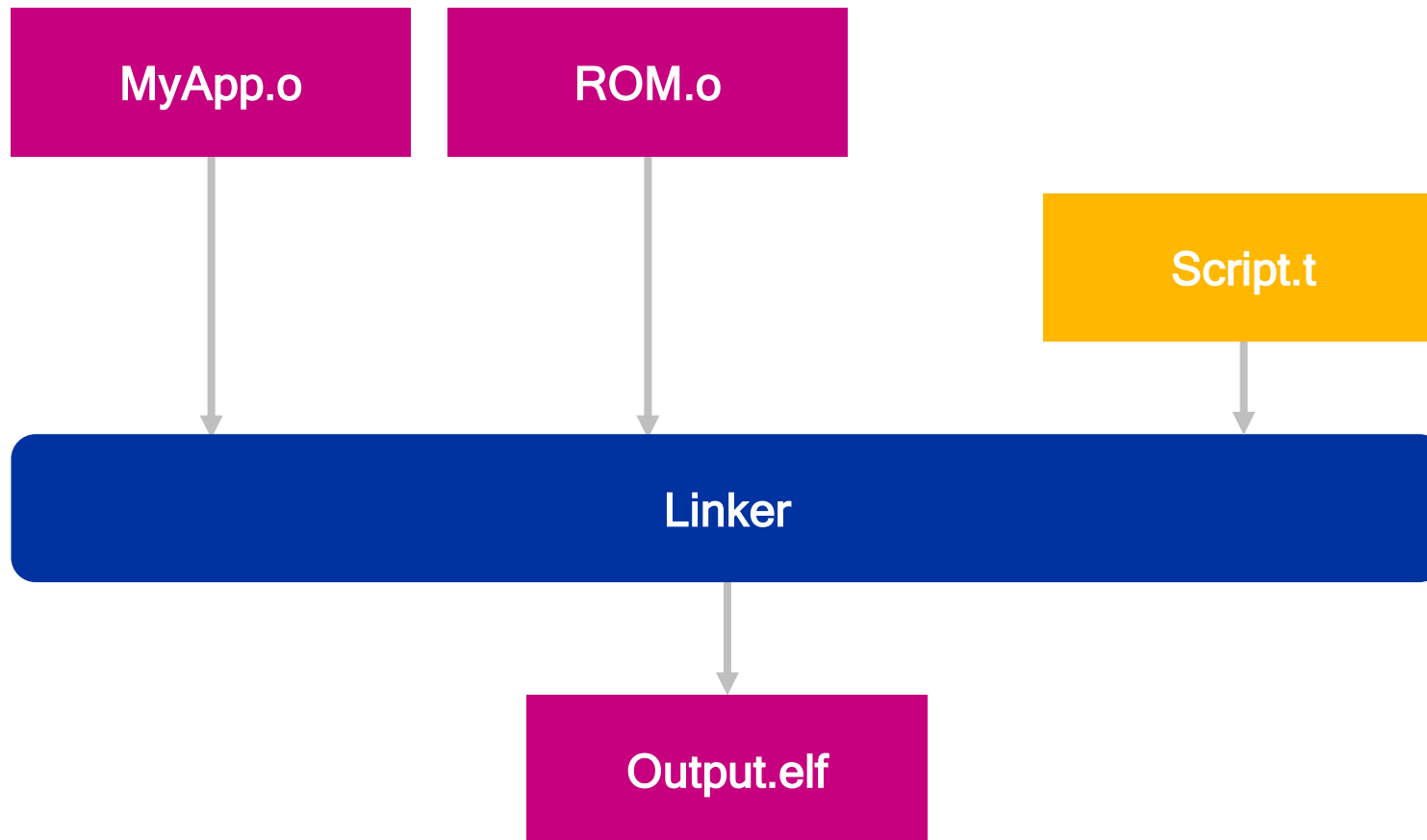
Implementation

4

Summary

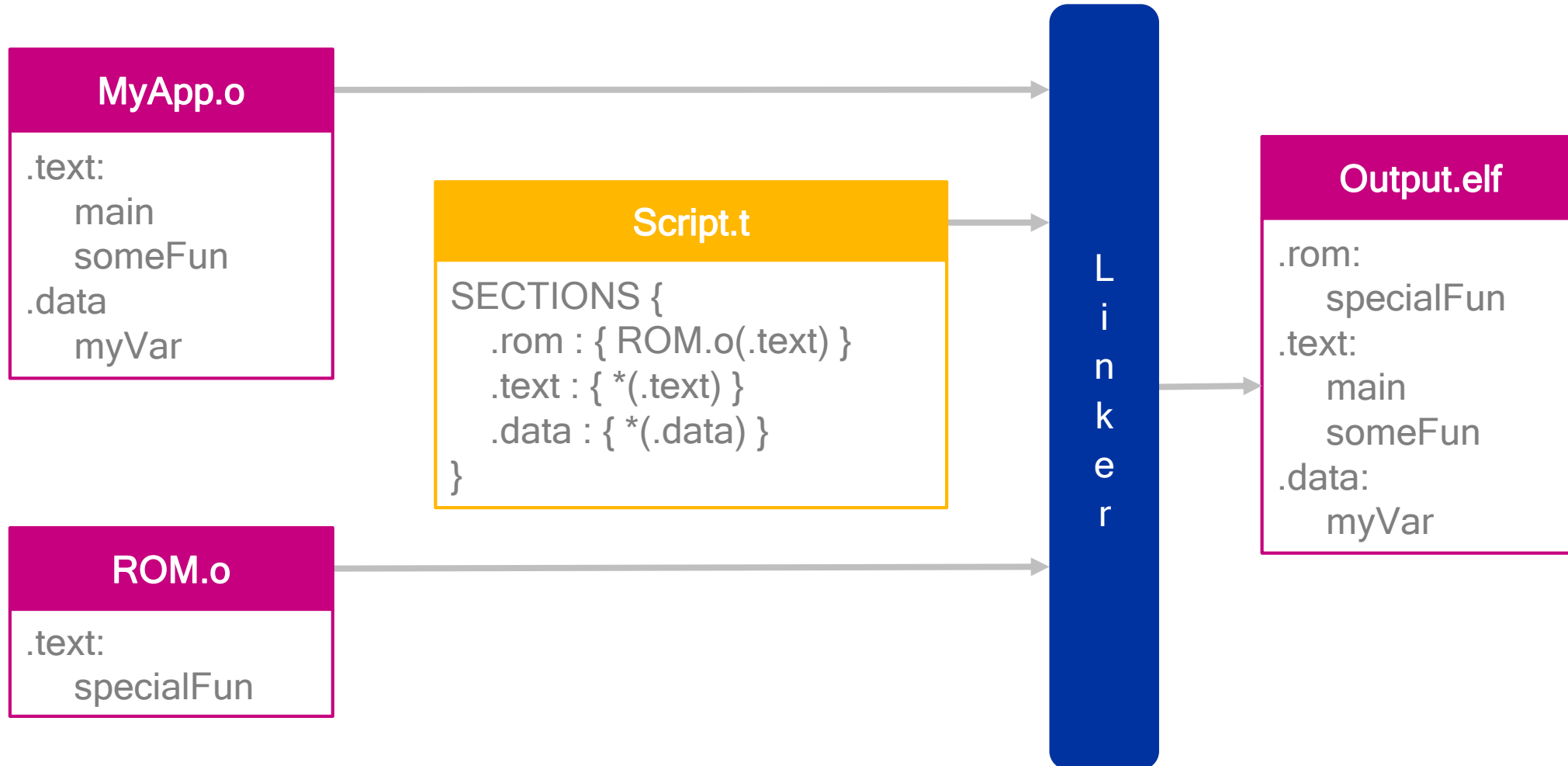
Motivating Example

A typical application with a linker script, compiled without LTO



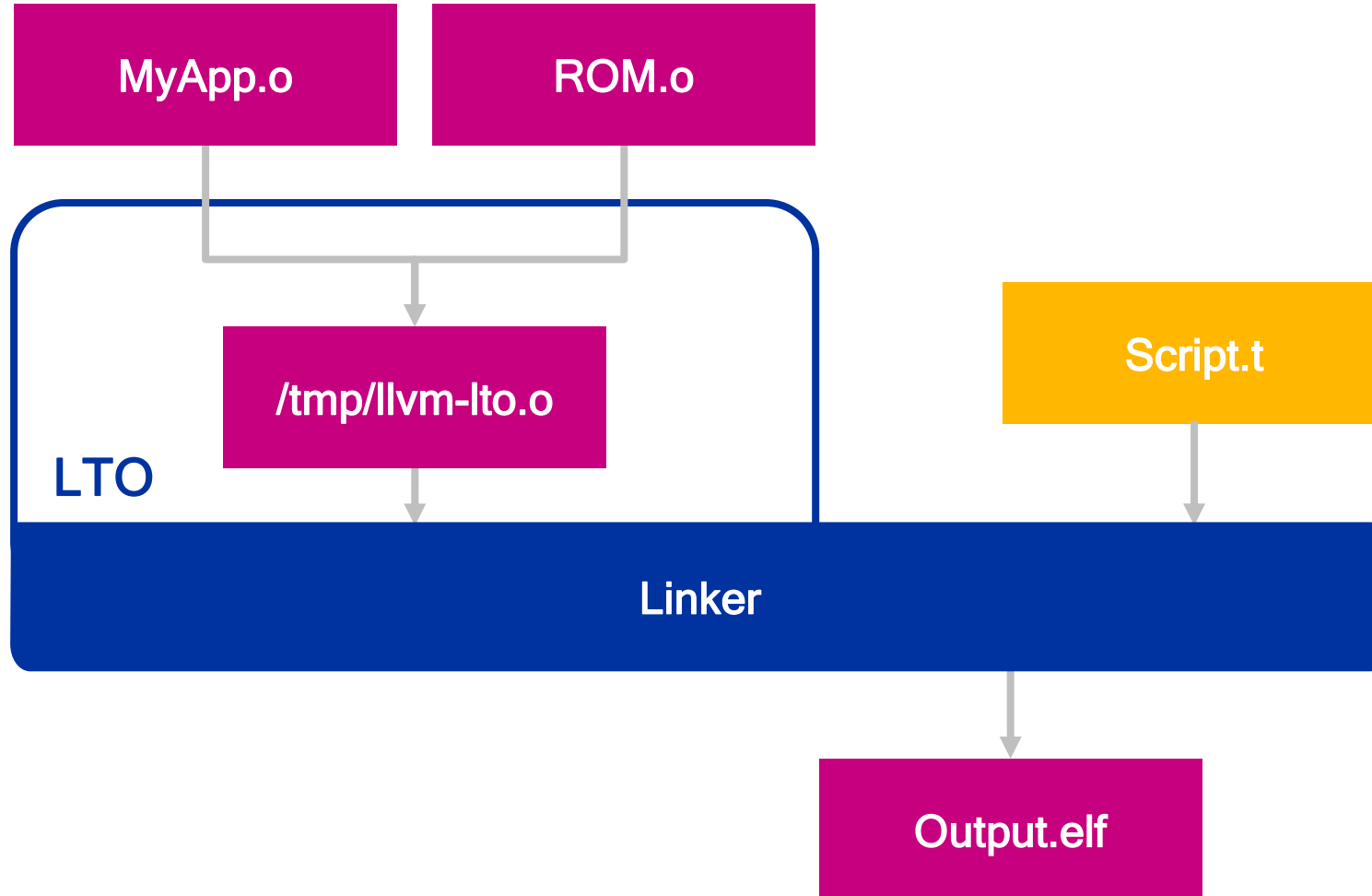
Motivating Example

Section layout by the linker



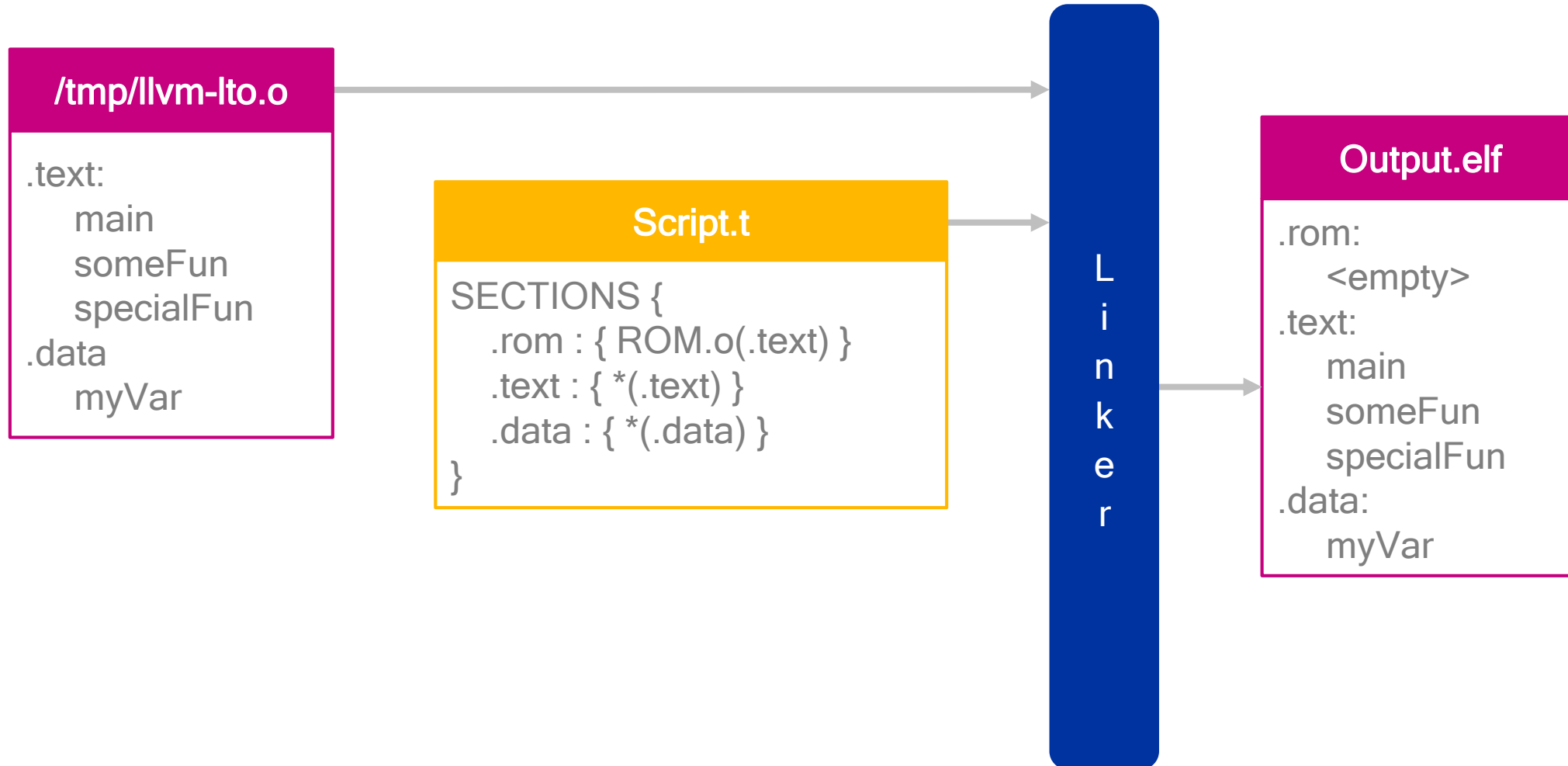
Motivating Example

With LTO enabled



Motivating Example

Section layout with LTO enabled



Motivating Example

Comparison of output

Non-LTO

Output.elf

```
.rom:  
    specialFun  
.text:  
    main  
    someFun  
.data:  
    myvar  
...
```

LTO

Output.elf

```
.text:  
    main  
    someFun  
    specialFun  
.data:  
    myvar  
...
```


Motivating Example

Comparison of output

Non-LTO

```
Output.elf
.rom:
  specialFun
.text:
  main
  someFun
.data:
  myvar
...
```



LTO

```
Output.elf
.text:
  main
  someFun
  specialFun
.data:
  myvar
...
```

Agenda

1

Motivating
Example

2

What we need
to address

3

Implementation

4

Summary

Why do we get this wrong?

Two problems at the heart of this issue:

We do not track the origin of symbols during LTO

The linker can't apply path-based linker script rules without knowing the 'real' origin of a symbol

We apply transformations across output sections

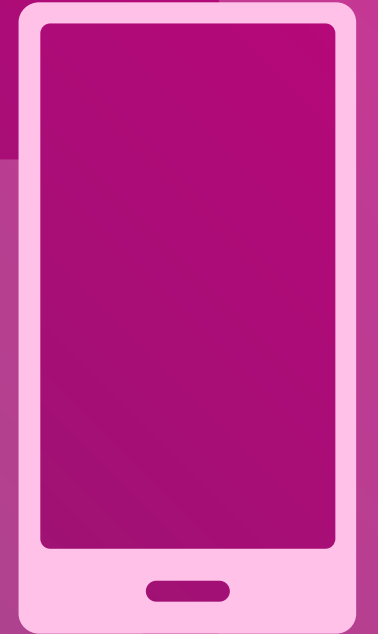
This can lead to correctness and performance problems, e.g.

- Constant merging into section not loaded at time of access
- Inlining from "fast" into "slow" memory

”LTO doesn't know about linker scripts and their effects (see other related bug reports).

That means basically a Won't Fix...”

– GCC Bug #65252



Agenda

1

Motivating
Example

2

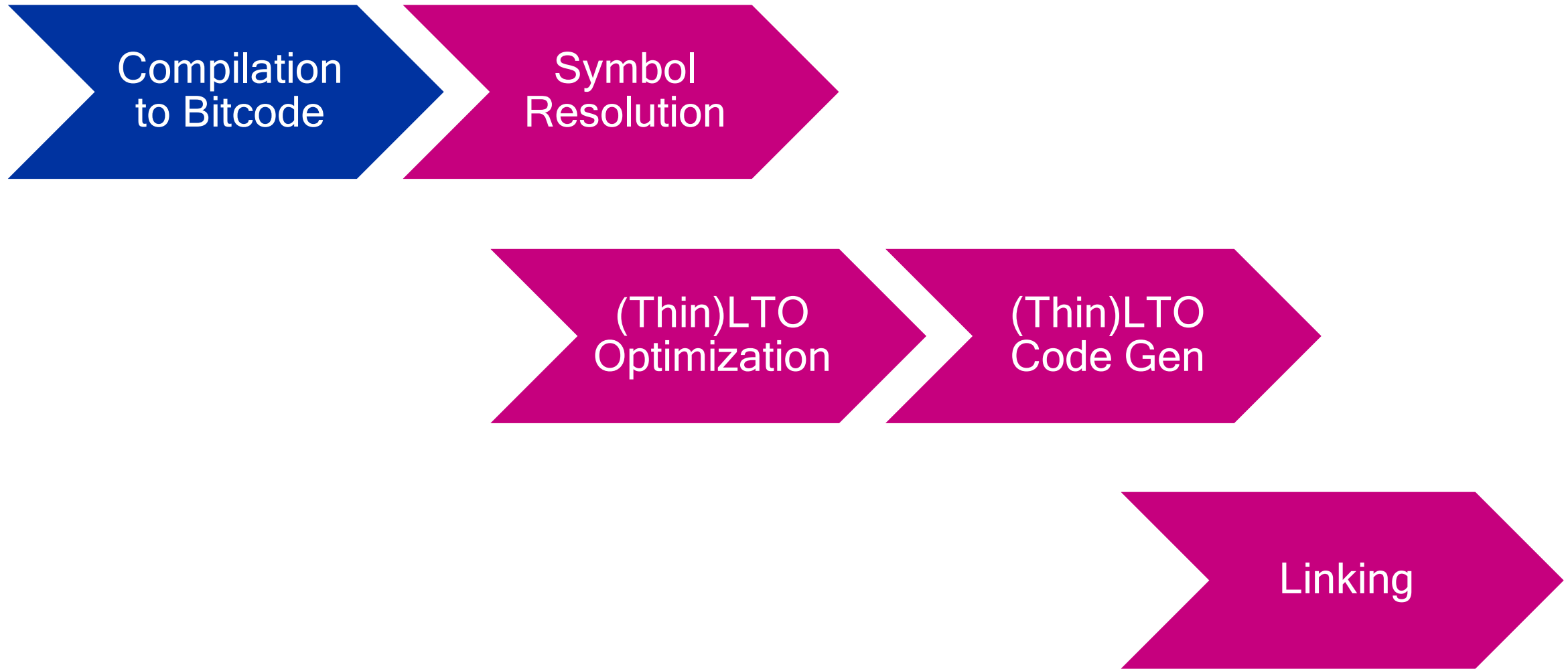
What we need
to address

3

Implementation

4

Summary



Step 1: Compilation of individual files

In addition to producing bitcode,

- Clang invokes backend to obtain a section name for each symbol
- Adds this as “linker_input_section” attribute to each GlobalObject, and
- Stores it as a field in the IR Symbol Table of the bitcode file

```
clang -flto -c -o ROM.o ROM.c
```

```
define void @specialFun() #0 { ... }
```

```
define void @specialFun()  
  “linker_input_section”=“.text” #0 { ... }
```

IR Symbol Table

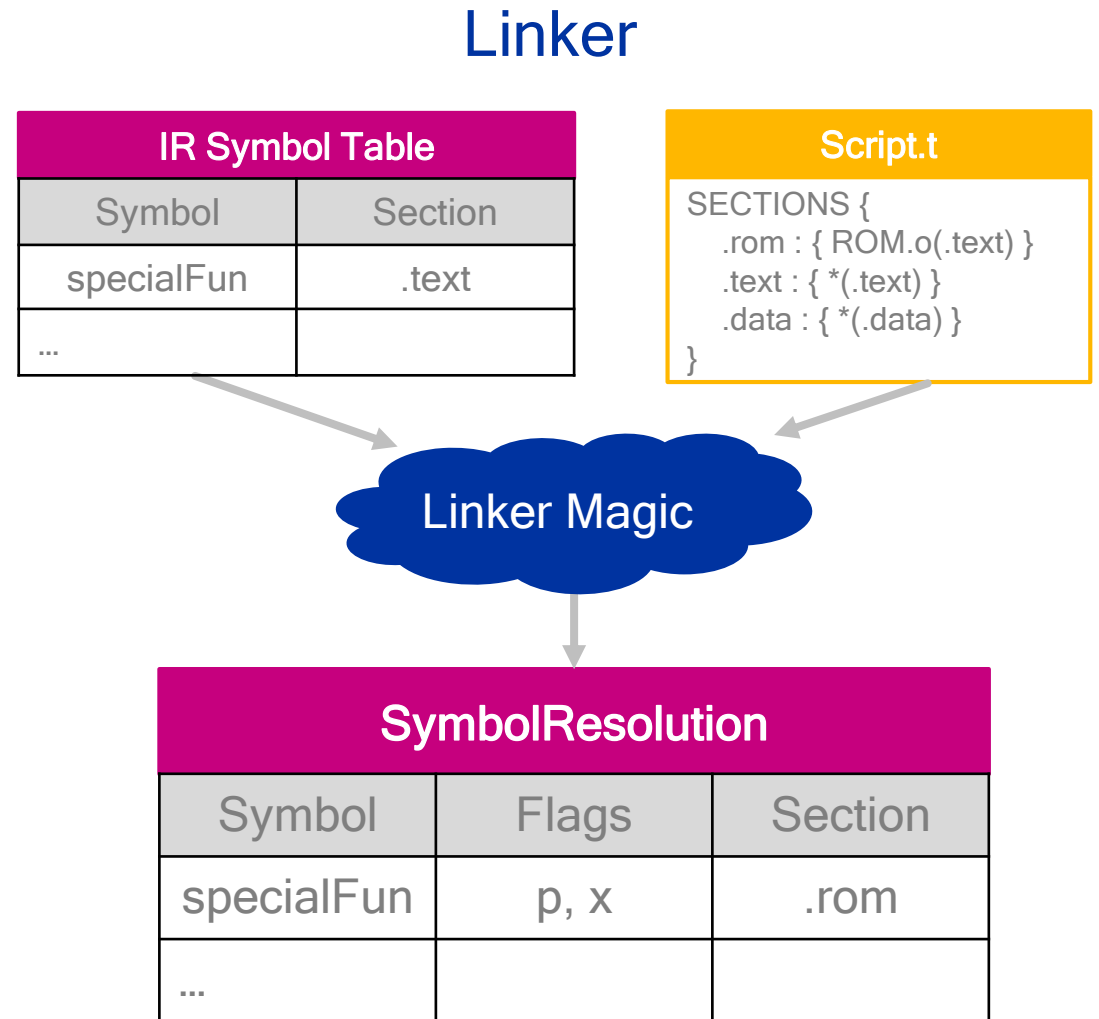
Symbol	Section	...
specialFun	.text	
...		



Step 2: Symbol resolution in the linker

Linker iterates over IR symbol table and matches symbols to linker script rules

- With input section names in the symbol table, this is just like reading an object file
- The linker determines an output section for each symbol - including locals - and communicates it to LTO along with the existing SymbolResolution information
- We set the **Module Id** to a string known to the linker e.g. “library.a(filename.o)”





Step 3: (Thin)LTO Optimization

Set additional attributes based on information provided by linker, then apply optimizations as usual

- For Regular LTO, set attributes before merging
- For ThinLTO, set attributes before & after import
- Some optimizations are modified to become aware of attributes as needed (constant merging, inlining, function merging, outlining...)

(Thin)LTO Optimization

```
define void @specialFun()  
  "linker_input_section"=".text" #0 { ... }
```



```
define void @specialFun()  
  "linker_input_section"=".text"  
  "linker_output_section"=".rom"  
  "module_id"="(ROM.o)" #0 { ... }
```

```
graph LR; A[Compilation to Bitcode] --> B[Symbol Resolution]; B --> C["(Thin)LTO Optimization"]; C --> D["(Thin)LTO Code Gen"]; D --> E[Linking];
```

Compilation to Bitcode

Symbol Resolution

(Thin)LTO Optimization

(Thin)LTO Code Gen

Linking

Step 4: (Thin)LTO Code Generation

Emit symbols with linker script attributes to 'augmented' section names understood by the linker

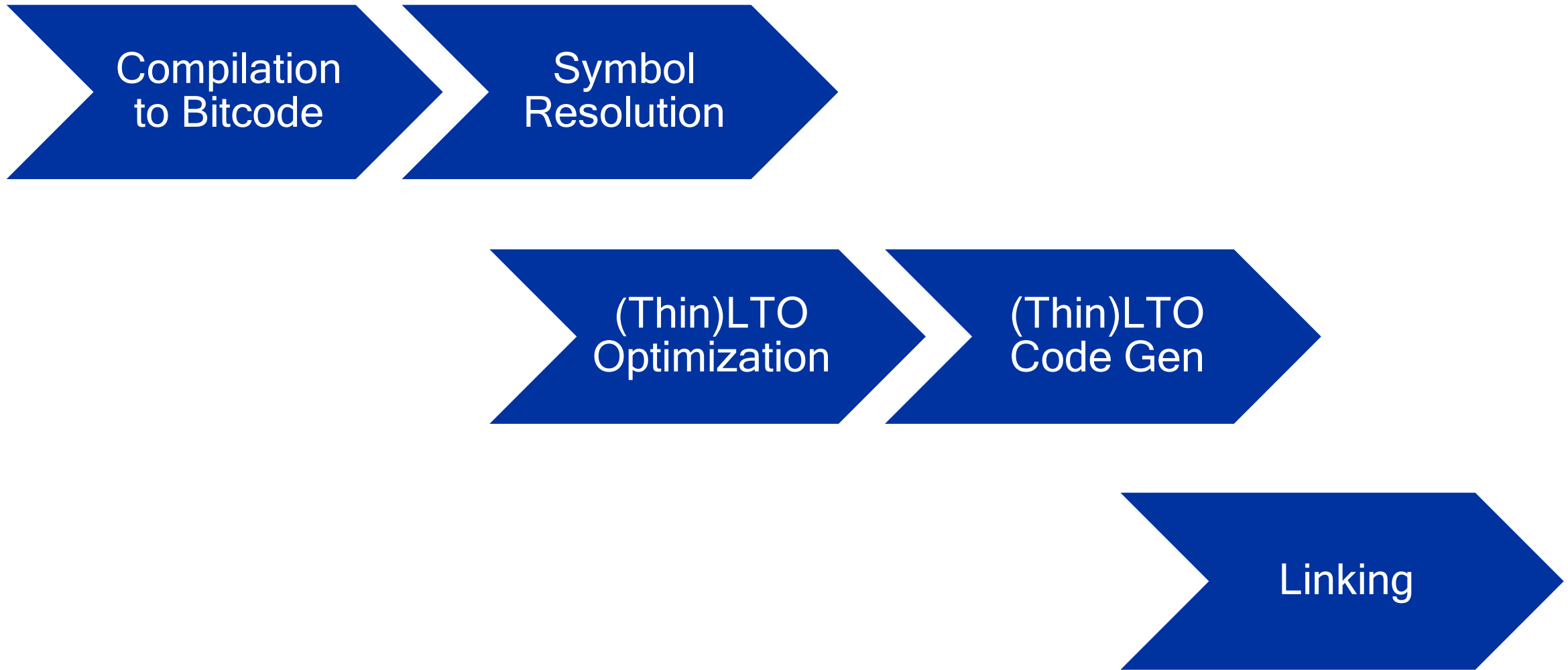
The 'augmented' section name encodes

- the *linker_input_section*, and
- the *module_id*

CodeGen

```
define void @specialFun()  
  "linker_input_section"=".text"  
  "linker_output_section"=".rom"  
  "module_id"="(ROM.o)" #0 { ... }
```

```
.section ".text^(ROM.o)", "ax", @progbits  
.globl specialFun  
...
```

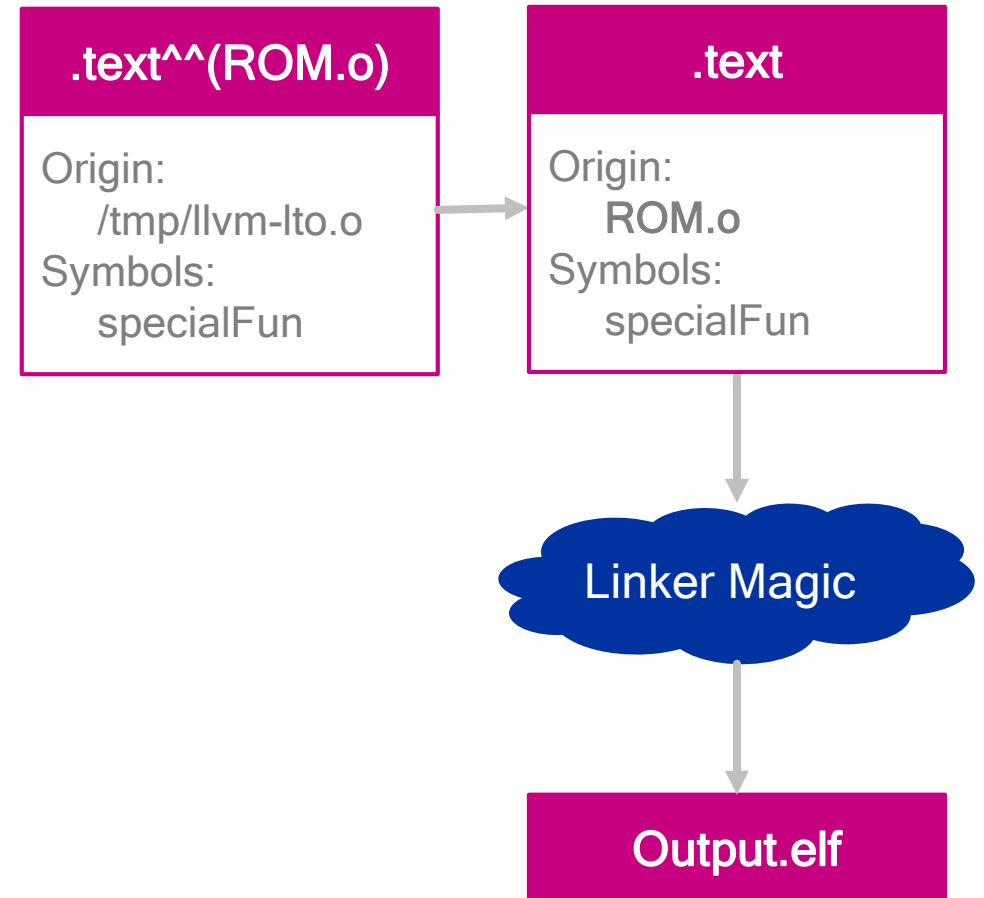


Step 5: Linking

The linker parses the augmented section names and lays out the output according to the linker script

- The Module Id encoded in the section name allows the linker to override the origin of sections coming out of LTO
- It can then apply linker script rules as if these sections came from a regular object file

Section Layout



Agenda

1

Motivating
Example

2

What we need
to address

3

Implementation

4

Summary

Summary

We have shown how (Thin)LTO can be enhanced to support path-based rules in linker scripts

The proposed approach does not require fundamental changes to the architecture of LTO

We add a small number of attributes to GlobalObjects, and augment the LTO APIs for the linker during symbol resolution

This extends the benefits of (Thin)LTO to a vast field of embedded applications.

The proposed approach is already in production use, enabling LTO for applications with 10,000+ linker script rules

Thank you

Follow us on:   

For more information, visit us at:

www.qualcomm.com & www.qualcomm.com/blog



Nothing in these materials is an offer to sell any of the components or devices referenced herein.

©2017 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to “Qualcomm” may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes Qualcomm’s licensing business, QTL, and the vast majority of its patent portfolio. Qualcomm Technologies, Inc., a wholly-owned subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of Qualcomm’s engineering, research and development functions, and substantially all of its product and services businesses, including its semiconductor business, QCT.