HW peripherals as software abstractions

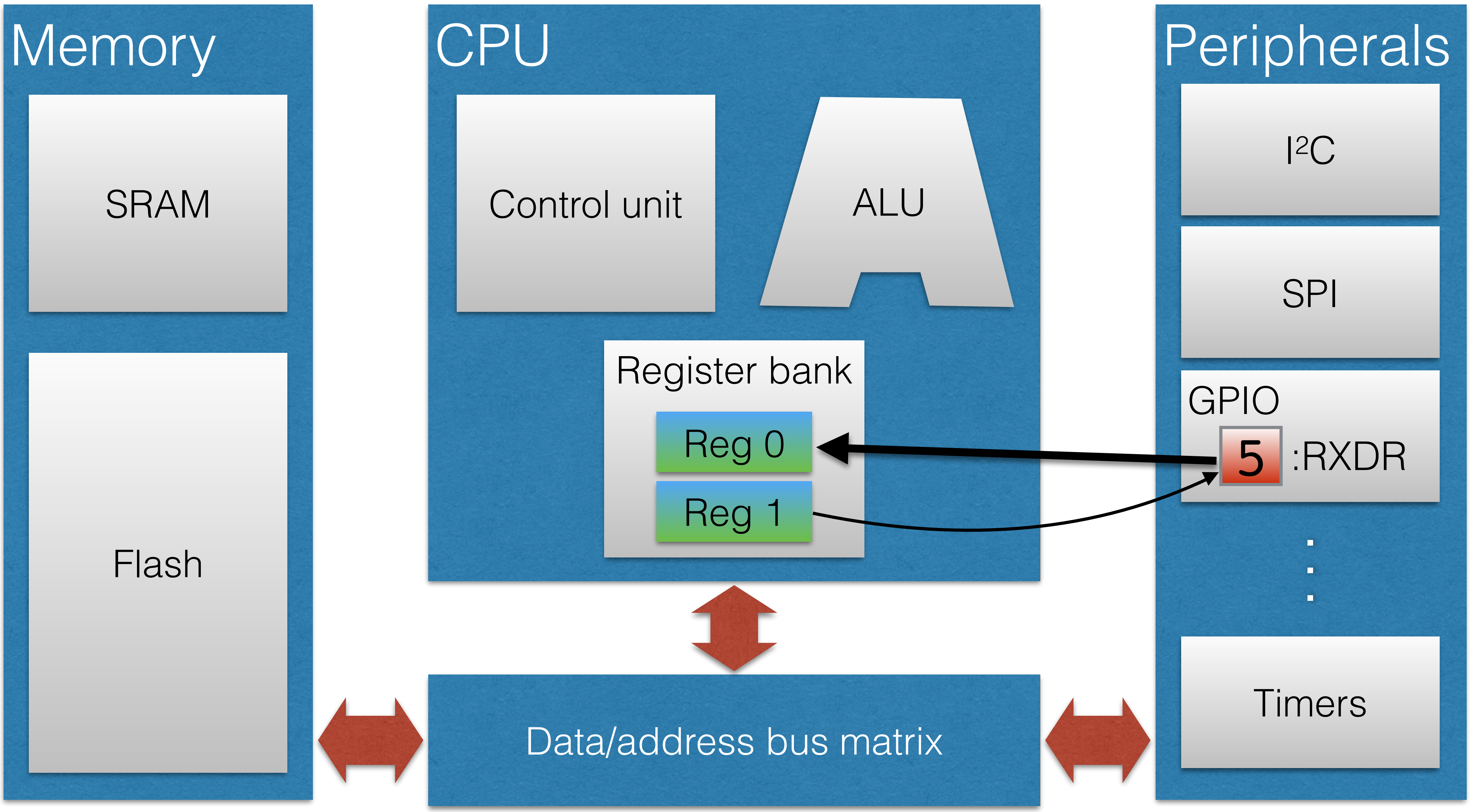
Pekka Nikander

May 2014

Presentation outline

- Recap of memory mapped peripherals
- Basic data type sizes
- Compiler and linker interplay
- Fixed address variables in C / C++
- Meet the optimiser, part I
- Summary

Peripheral registers



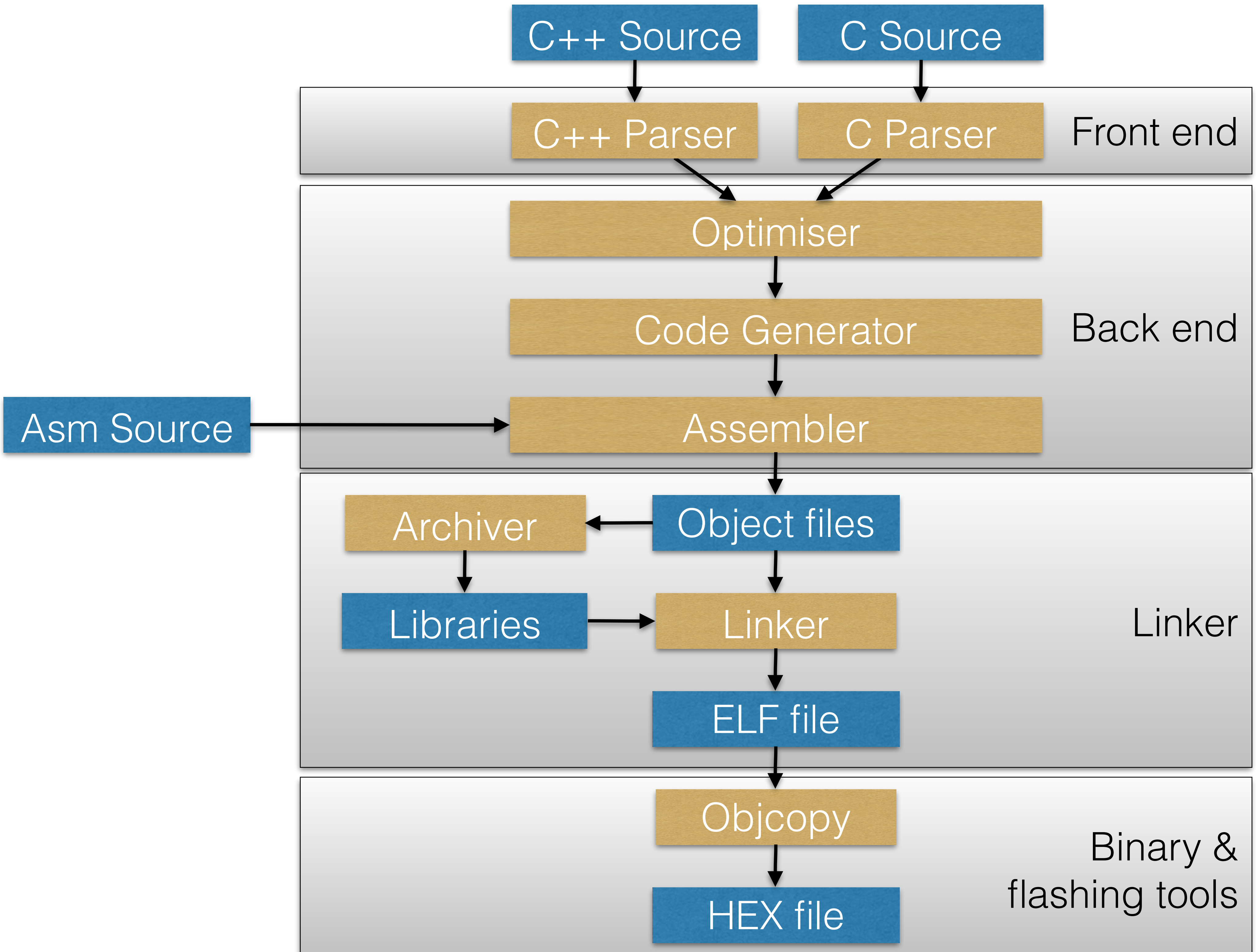
Basic data types

- C was designed to be a *portable* language
  - More modern languages (Java, …) do it much better
- When doing bare metal, better to be explicit; see the table.
- Avoid unnecessary type casts
  - E.g. prefer unnamed unions to **`void`** pointers

What	Type
8	`uint8_t`
16	`uint16_t`
32	`uint32_t`
Integer for arithmetic*	`int32_t`
Generic pointer (to be avoided)	~~`void *`~~

* ARM and thumb generate more efficient code with 32 bit integers

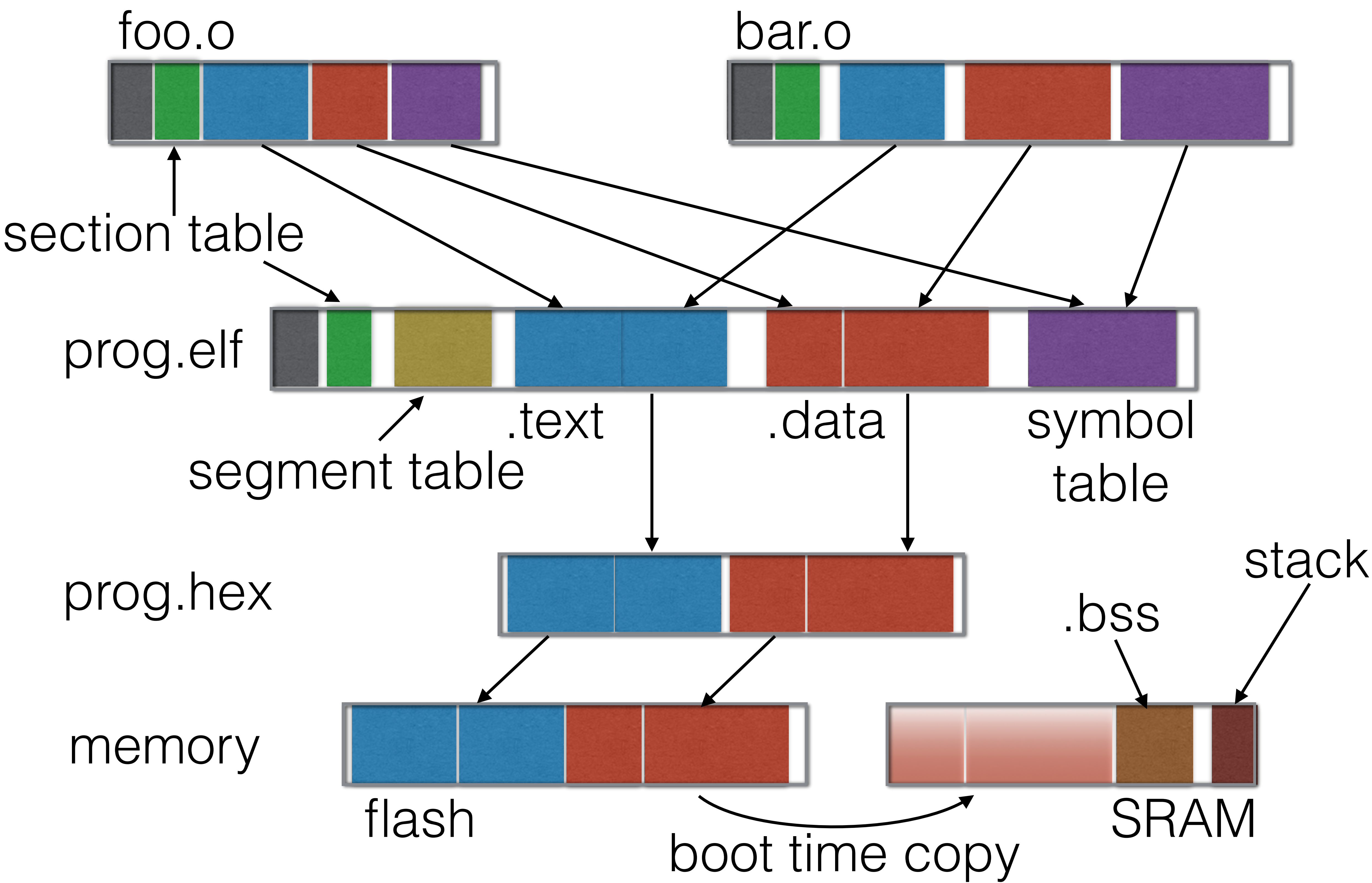
Compiler tool chain



Main components

- Front end: Converts source code to intermediate
- Back end: Analyses, optimises, generates assembler code
- Assembler: Generates binary object files
- Linker: Combines object files into an executable
    - Resolves symbols into addresses

Object code anatomy



GNU Linker script

- Defines how the ELF file is constructed
  - Which sections go to which segment
  - Where the segments start in the address space
- Is able to define values (i.e. addresses) for symbols
  - E.g. locate a given variable at a specific address
- Be aware of which linker script is being used

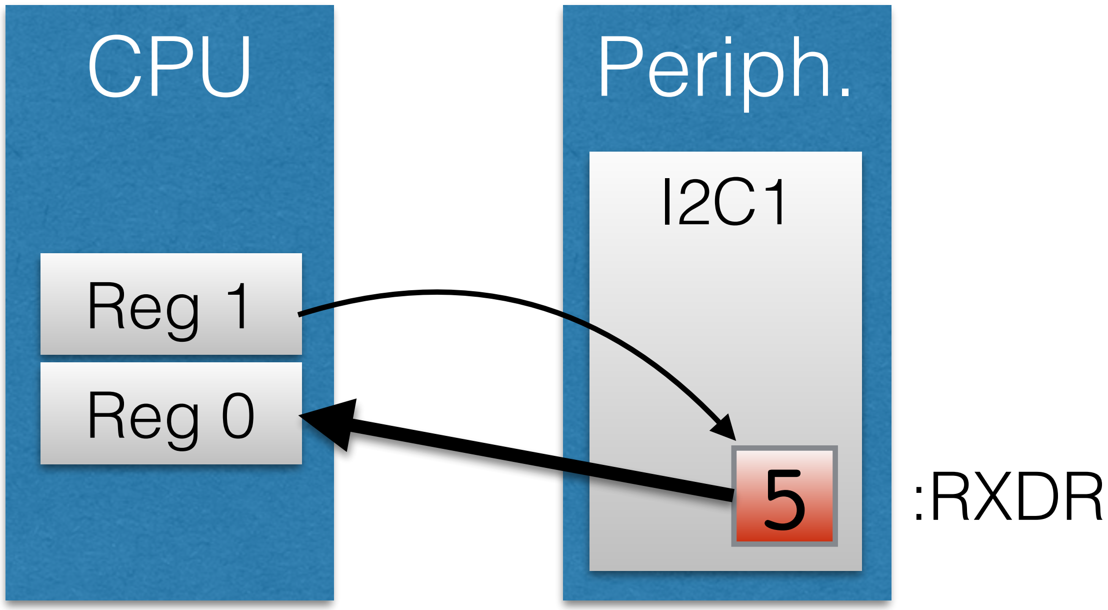
Fixed address variables

- A simple example: a 32-bit integer
  - Address in source code vs. linker script
- A fixed address struct
  - Alignment and padding
- Peripherals in C and C++
  - *POD* in C++ and *standard layout* in C++11

The simple example

- Load I^2C read data (RXDR) peripheral register address to the CPU register R1
- Load “memory” from the address in R1 to R0
    - Reads the I^2C peripheral register RXDR through the data bus matrix to the CPU register R0
- Return the value in R0

```
{
   volatile int *p;
   p = 0x40005424;
   return *p;
}
```



What’s wrong?

- Context completely lost
  - What does `0x40005424` mean?
- Address defined in the C source code
  - A header file would be better
  - The linker script *might* be even better
- (Besides, generates inefficient ARM thumb code)

#1: Use a header file

header.h:

```
#define I2C1_RXDR (volatile int *)0x40005424
```

source.c:

```
{
     *I2C1_RXDR = 5;
}
```

#2: Use a struct & define

header.h:

```
struct I2C {
        …
        volatile int RXDR;
};

#define I2C1 (struct I2C *const)0x40005400
```

source.c:

```
{
        I2C1->RXDR = 5;
}
```

#3: Use a struct & linker

header.h:

```
extern struct I2C {
      …
      volatile int RXDR;
} I2C1[]; // note the brackets for defining an “array”
```

linker.ld:

```
      I2C1 = 0x40005400;
```

source.c:

```
{
      I2C1->RXDR = 5;
}
```

Structs & padding

- Be aware of alignment, padding, and endianness
  - Especially with bare metal & communication
- Again, better to be explicit
  - Avoid implicit padding, add explicit fields instead
- Note! Some ARM instructions fail on unaligned data
- Learn to use `__attribute__((...))`

What’s wrong here?

```
#define ETHER_ADDR_LEN 6
typedef uint32_t in_addr_t;

struct arp {
    uint16_t  hardware_address_space;
    uint16_t  protocol_address_space;
    uint8_t   hardware_address_length;
    uint8_t   protocol_address_length;
    uint16_t  opcode;
    uint8_t   src_eth_addr[ETHER_ADDR_LEN];
    in_addr_t src_ip_addr;
    uint8_t   dst_eth_addr[ETHER_ADDR_LEN];
    in_addr_t dst_ip_addr;
};
```

Fixed

```
#define ETHER_ADDR_LEN 6
typedef uint32_t in_addr_t;

struct arp {
    uint16_t  hardware_address_space;
    uint16_t  protocol_address_space;
    uint8_t   hardware_address_length;
    uint8_t   protocol_address_length;
    uint16_t  opcode;
    uint8_t   src_eth_addr[ETHER_ADDR_LEN];
    in_addr_t src_ip_addr;
    uint8_t   dst_eth_addr[ETHER_ADDR_LEN];
    in_addr_t dst_ip_addr;
} __attribute__((packed));
```

C++ PODs

- C++ structs vs. classes
- POD = Plain Old Data
- Making POD structs and classes

C++ structs vs classes

- The *only* difference:
  - **`struct`** members are **`public`** by default
  - **`class`** members are **`private`** by default
- As a matter of *style*:
  - use the **`struct`** for something that be a struct in C
  - use the **`class`** if it uses C++-specific features
- http://stackoverflow.com/questions/7762085/difference-between-a-struct-and-a-class

Plain Old Data (POD)

- Type (incl. classes) where the compiler guarantees that there is no “magic”, for example
  - no hidden pointers to vtables
  - no offsets that get applied to the address cast
  - no constructors or destructors
- Roughly speaking, a type is a POD when the only things in it are built-in types and their combinations

Why to consider PODs?

- Define exact binary-level memory layout
- Allow structs (or unions) to be coated with syntactic sugar
  - Define member functions
  - Define static data
- Allows going to C++ and still “think in C”

C++11 changes

- Goal: a POD in C++11 gives you the same memory layout as a struct compiled in C
    - A POD is both trivial and standard layout
- Standard layout is often enough, but be careful if you decide to use non-trivial features
    - Possible to use (limited) inheritance, constructors, protected or private, etc
- http://stackoverflow.com/questions/4178175/what-are-aggregates-and-pods-and-how-why-are-they-special

Rules of thumb

- Definitely OK to define a `struct` or `union` as in C
- The following are not OK
    - Virtual functions
    - Multiple inheritance
    - Constructors that are not `constexpr`

A C++11 example

header.h:

```
class I2C {
   volatile int RXDR;
public:
   size_t read(uint_t *buf, size_t bufsize);
} I2C1[];  // note the brackets for defining an “array”

inline size_t I2C::read(uint_t *b, size_t *b) {
   while (…) {
      …
      b[c] = RXDR;
```

linker.ld:

```
      I2C1 = 0x40005400;
```

Meet the optimiser

Part I

A few common tricks & traps

- Inlining
- Constant propagation
- LLVM and undefined behaviour
- Do volatilise

Inlining

- Expand function at where it is called
  - Avoids function call overhead (not so important)
  - Allows other optimisations to be applied
    - Especially constant propagation
- If a function is called only once, trivially useful
- Otherwise you need to estimate cost vs benefit

Constant propagation

- Evaluate constant expressions at compile time
  - C++11 has even the **`constexpr`** keyword
- Propagate the newly computed constants, i.e. evaluate more values with their constant values
- Sometimes lead to significant code size reduction
- Some parts of Ell-i runtime are carefully designed to become optimised due to constant propagation

Example: inlining & constant propagation

```
static inline
void  digitalWrite(pin_t pin, uint32_t val) {
  if (val)
    GPIOPIN[pin].gpio_port->BSRR = GPIOPIN[pin].gpio_mask;
  else
    GPIOPIN[pin].gpio_port->BRR  = GPIOPIN[pin].gpio_mask;
}

…
digitalWrite(13, 1);
…
```

Step #1: Inlining

```
static inline
void  digitalWrite(pin_t pin, uint32_t val) {
  if (val)
    GPIOPIN[pin].gpio_port->BSRR = GPIOPIN[pin].gpio_mask;
  else
    GPIOPIN[pin].gpio_port->BRR  = GPIOPIN[pin].gpio_mask;
}

…
digitalWrite(13, 1);
if (HIGH)
  GPIOPIN[13].gpio_port->BSRR = GPIOPIN[13].gpio_mask;
else
  GPIOPIN[13].gpio_port->BRR  = GPIOPIN[13].gpio_mask;
…
```

Step #2: `if` optimisation

```
…
digitalWrite(13, 1);
if (1)
  GPIOPIN[13].gpio_port->BSRR = GPIOPIN[13].gpio_mask;
else
  GPIOPIN[13].gpio_port->BRR  = GPIOPIN[13].gpio_mask;
…
```

Step #3: Constant propagation

```
static const struct {} GPIOPIN[] = {
  …
  { .gpio_port = GPIOA, .gpio_mask = 0x02000 },
  …
};
```

```
…
digitalWrite(13, 1);
if (1)
  GPIOPIN[13].gpio_port->BSRR = GPIOPIN[13].gpio_mask;
else
  GPIOPIN[13].gpio_port->BRR  = GPIOPIN[13].gpio_mask;
…
```

Step #3: Constant propagation

```
static const struct {} GPIOPIN[] = {
  …
  { .gpio_port = GPIOA, .gpio_mask = 0x2000 }, // pin 13
  …
};
```

…

~~digitalWrite(13, 1);~~

~~if (1)~~

  ~~GPIOPIN[13].gpio~~GPIOA->BSRR = 0x2000~~N[13].gpio_mask~~;

~~else~~

~~ GPIOPIN[13].gpio_port->BRR = GPIOPIN[13].gpio_mask;~~

…

Example: End result

…

```
GPIOA->BSRR = 0x2000;
```

…

Undefined behaviour

- Designers of C wanted it to be an extremely efficient
- Therefore some seemingly reasonable things in C have *undefined* behaviour
  - And it really is *completely* undefined
- LLVM could utilises undefined less than it could …
  - … but it still produces surprises in many cases
- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

An LLVM example

- Consider the following code
    - It checks for the `null` pointer, doesn't it?

```
void contains_null_check(int *P) {
  int dead = *P;
  if (P == 0)
    return;
  *P = 4;
}
```

An LLVM example

- But what if the compiler happens to run
  - "Redundant Null Check Elimination" first and
  - "Dead Code Elimination" afterwards

```
void contains_null_check(int *P) {
  int dead = *P;
  if (P == 0)
    return;
  *P = 4;
}
```

An LLVM example

- **`dead = *P`** dereferences **`P`**
  - Remember, dereferencing **`NULL`** is *undefined*
- Hence, **`P`** cannot be NULL

```
void contains_null_check(int *P) {
  int dead = *P;
  if (false)  // P cannot be NULL
    return;
  *P = 4;
}
```

An LLVM example

- Next eliminate dead code

```
void contains_null_check(int *P) {
  int dead = *P;
  if (false)
    return;
  *P = 4;
}
```

Once again `volatile`

- LLVM (and GCC) *will* optimise your writes away

```
for (int i = 0; i < 100000; i++)
    ;
```

- Be sure to declare `volatile` if you don't want that
- Remember
  - LLVM and GCC both try to be helpful
  - But sometimes they aren't, as they try to generate efficient code, and can't read your intentions

Summary

- Feel free to declare peripherals as structs or PODs
  - Consider using the linker to define the address…
- Heavy inlining, with constant propagation, allows you to write high-level code that optimises well
- Beware of *undefined* behaviour, lest thou regret