

SES

Appendix: C for Embedded Systems

Prof. Dr.-Ing. Bernd-Christian Renner



Contents

1. Intro
2. Modules
3. Bit Manipulation
4. Coding Standards
5. Declarations and Definitions
6. Functions
7. Variable Lifetime and Scope
8. Types and Type Declarations
9. Keyword `volatile`
10. Advice



Intro

The C Programming Language

- C: General-purpose computer programming language developed 1969 - 1973 by B. Kernighan and D. Ritchie at Bell Labs for use with Unix
- C is often used for embedded systems
- Standardization of C
 - C99
 - Inline functions (keyword `inline`)
 - an explicit boolean data type
 - C11
 - Detailed memory model to better support multiple threads
- It is possible to include assembler code in C source code
 - `asm("movl %ecx %eax");`
`/* moves contents of ecx to eax */`
 - `__asm__("movb %bh (%eax)");`
`/* moves the byte from bh to the memory pointed by eax */`

GCC and Cross Compilation

- GNU Compiler Collection (GCC):
 - Toolkit for compiling and assembling programs for a variety of platforms
- GNU Binary Utilities (binutils):
 - Collection of programming tools for the manipulation of object code in various object file formats (assembler, linker, ..)
- GCC requires
 - a compiled copy of binutils for each target platform
 - a portion of target platform's C standard library be available on host platform (at least crt0, a set of execution startup routines)
 - Crt = C runtime



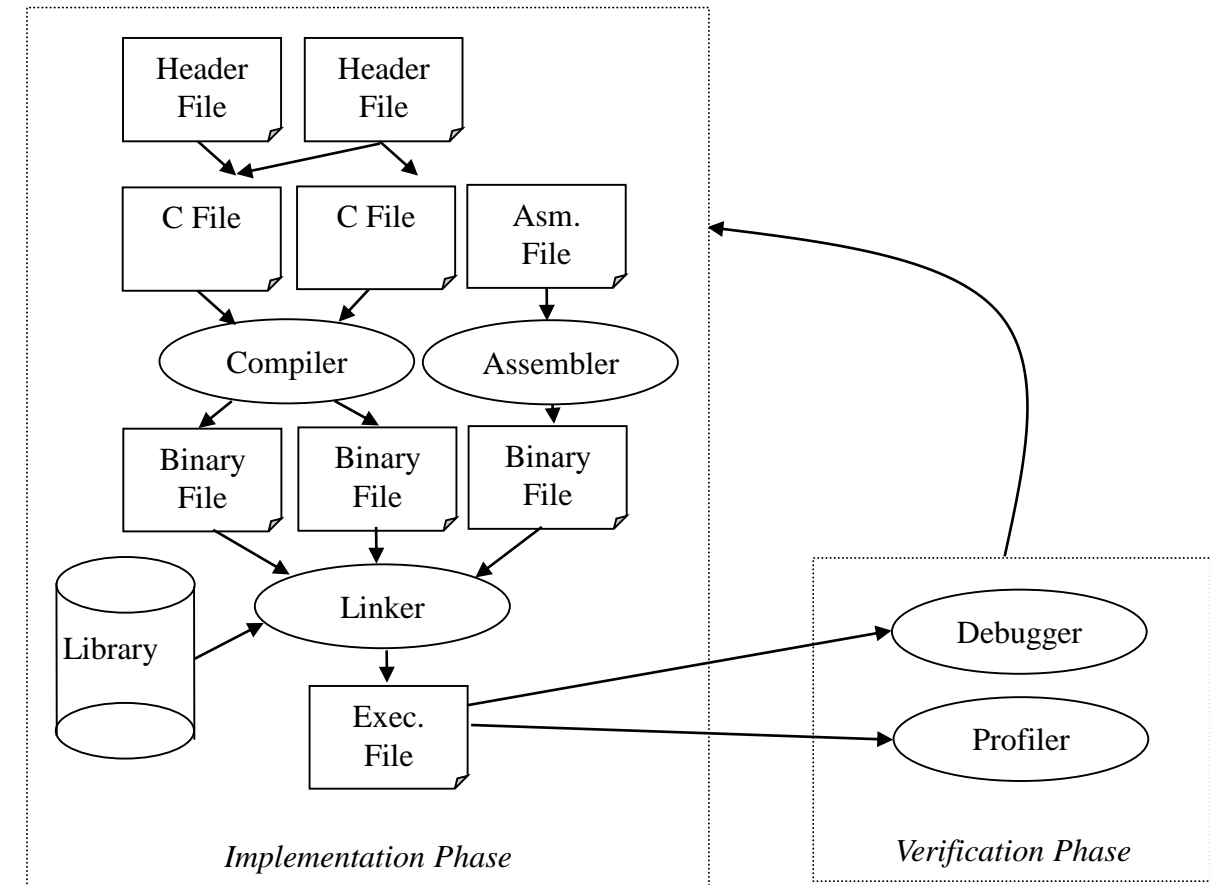
Modules

Modules

- A module should implement just one aspect of a system, e.g. a device driver for an A/D converter
 - It may comprise one or more compilation units (e.g., .c or .asm source code files)
- Conventions for module names
 - must be unique in their first eight characters, suffix .h and .c used for header/source files
 - shall consist entirely of lowercase letters, numbers, and underscores (no spaces)
 - should not share the name of a standard library header file
 - any module containing a `main()` function shall have the word “main” in its filename

Modules II

- Compilers
 - Cross compiler:
Runs on one processor, but generates code for another
- Assemblers
- Linkers
- Debuggers
- Profilers



Header Files (*.h) I

- Header files contain *declarations* and *macro definitions* to be shared between several source files
- **Principle:** Single Point of declaration
- Including a header file produces the same result as copying header file into each source file
- System header files declare interfaces to OS and hardware
- Own header files contain declarations for interfaces between the source files of your program
- Include syntax
 - `#include <file>`: Used for system header files included in a standard list of system directories
 - `#include "file"`: Used for header files of your own program included in directory containing current file and in the same directories used for <file>

Header Files II

- Header files should *not* contain executable lines of code
- Rule: One header file for each source file
- Reasoning:
 - C language standard gives all variables & functions global scope by default
 - Downside: Unnecessary (dangerous) coupling between modules
- To reduce inter-module coupling, hide functions, data types, constants & variables within a module's source file (keyword `static`)
- Each header file should contain a preprocessor guard against multiple inclusion:

```
#ifndef _ADC_H
#define _ADC_H
the entire file
#endif /* _ADC_H */
```

Header Files III

- Header files identify only
 - functions,
 - constants,
 - data types (via prototypes or macros, `#define`, and `struct/union/enum typedefs`), and
 - global variable extern declarations about which it is strictly necessary for other modules to know about
- Header files may expose variables by way of *extern* keyword (defined in some c file), but
 - Proper module encapsulation requires data hiding
 - Goal: All internal state data is stored in private variables inside source code files, these should be declared with keyword `static` to allow linker to hide them

Header Files IV

- Do not expose internal format of any module-specific data structure passed to or returned from the module's interface functions
- Types that need to be passed in and out of a module, should be typedef'ed in header file

- Example:

Instead of

```
struct foo { ... };
```

have

```
typedef struct { ... } foo_t;
```


Preprocessor Directives I

- Preprocessing lets you:
 - Replace tokens in current file with specified replacement tokens
 - Embed files within the current file
 - Conditionally compile sections of current file
 - Generate diagnostic messages
 - Apply machine-specific rules to specified sections of code
- Lines beginning with # communicate with preprocessor
- Preprocessor syntax is independent of C

Preprocessor Directives II

- `#define` Defines a preprocessor macro
 - `# define identifier token-sequence`
 - `# define identifier(identifier-list) token-sequence`
- Object-like macro: Identifier to be replaced by code (Single Point of declaration)

```
#define TABSIZE 1024
int table[TABSIZE]
```
- Function-like macros: Look like function calls

```
#define ABSDIFF(a, b) ((a)>(b) ? (a) - (b) : (b) - (a))
```

(better use inline functions!)
- Standard predefined macros
 - `__FILE__` (name of the current input file)
 - `__LINE__` (current input line number)
 - `__DATE__` (string describing date on which preprocessor was run)
 - `__TIME__` (string describing time on which preprocessor was run)

Preprocessor Directives III

- `#undef` Removes a preprocessor macro definition

- Conditional compilation

`if-line text elif-parts else-partopt #endif`

- `if-line`

- `#if constant-expression`

Conditionally suppresses portions of source code, depending on result of a constant expression

- `#ifdef identifier`

Conditionally includes source text if a macro name is defined

- `#ifndef identifier`

Conditionally includes source text if a macro name is not defined

Preprocessor Directives IV

- `elif-parts`
 - `#elif constant-expression ...`
- `#elif` **allows to abbreviate conditional statements**
- `else-part`
 - `#else text`

- `#line` **Supplies a line number for compiler messages**
- `#pragma` **Specifies implementation-defined instructions to the compiler**
- `#error` **Defines text for a compile-time error message**

Examples

- ```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```
- ```
#if X == 1  
...  
#elif X == 2  
...  
#else /* X != 2 and X != 1*/  
...  
#endif
```
- ```
#ifdef MACRO
definitions for case MACRO is
defined
#endif /* MACRO */
```
- ```
#if defined (__AVR__) || defined (__ARM__)  
definitions for AVR and ARM  
#endif
```

Macros

- Do not use parameterized macros if an inline function can be written to accomplish the same task

- Don't do this:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

if you can do this instead

```
inline int max(int a, int b) { return a>b ? a : b; }
```

- There are risks associated with `#define`

- `#define MAX_VAL 20 + 10`
`printf("Resultat: %d", 10 * MAX_VAL);`

- Extensive use of parentheses (as shown above) is important

- It doesn't eliminate the unintended double increment possibility of a call such as `MAX(i++, j++)`

- Other risks of macro misuse

- Comparison of signed and unsigned data or any test of floating-point data

avr/io.h

```
#ifndef _AVR_IO_H_
#  define _AVR_IO_H_
#  include <avr/sfr_defs.h>                /*Special function registers*/
#  if defined (__AVR_AT94K__)
#    include <avr/ioat94k.h>
#  elif defined (__AVR_ATmega1281__)
#    include <avr/iom1281.h>
.....
#endif
#include <avr/portpins.h>
#include <avr/common.h>
#include <avr/version.h>
/* Include fuse.h after individual IO header files. */
#include <avr/fuse.h>
/* Include lock.h after individual IO header files. */
#include <avr/lock.h>
#endif /* _AVR_IO_H_ */
```

avr/iom1281.h

```
#ifndef _AVR_IOM1281_H_
#define _AVR_IOM1281_H_

#include <avr/iomxx0_1.h>

/* Constants */
#define SPM_PAGESIZE      256          // Flash Page Size
#define RAMSTART          (0x200)
#define RAMEND            0x21FF      // last on-chip RAM address
#define XRAMEND           0xFFFF      // last external RAM address
#define E2END             0xFFF       // last EEPROM address
#define E2PAGE_SIZE       8           // size of the EEPROM page
#define FLASHEND          0x1FFFF     // last byte address in Flash
.....
```


Source Files (*.c) I

- Purpose and internal layout of a source file module must be clear
- Public functions are of most interest and thus appear ahead of private functions they call
- Every function declaration is matched by the compiler against its prototype
- Each source includes only the behaviors appropriate to control one “entity”
 - Examples of entities: Encapsulated data types, active objects, peripheral drivers (e.g., for a UART), and communication protocols or layers (e.g., ARP),

Source Files II

- A source file
 - should include header file of the same name to allow compiler to confirm that each public function and its prototype match
 - must be free of unused include files
- Absolute paths must not be used in include file names (relative to *.c file)
- Source files must not include other source files

Source Files III

- Each source file is comprised of some/all of the following sections, in order listed:
 - file heading
 - revision history
 - include statements
 - data type, constant, macro definitions
 - static data declarations
 - private function prototypes
 - public function bodies
 - private function bodies

Source Files IV

- File heading
 - Comment block that contains company name, copyright notice, file name, author's name, description of module
- Revision history
 - Comment block that describes changes made to module over time
- #defines and macros (three different places)
 - module's .c file
 - if #defines and macros are only applicable to the module
 - module's .h file
 - if #defines and macros are used by multiple .c files

Comments

- Put usage comments in header file
 - Don't put irrelevant implementation comments here
- Put implementation comments (including usage comments for static functions) in source file
- All assumptions shall be spelled out in comments
- Pay attention to fine line between under-commenting and over-commenting
 - Think first about writing self-documenting code
- Do not use comments to disable a block of code
 - Use preprocessor's conditional compilation feature:
 - `#if 0 ... #endif`, or `#ifdef`
 - Even better `#define DEBUG` `#ifdef DEBUG #endif`
- Use capitalized comment markers to highlight important issues: WARNING, NOTE, TODO
- Use a documentation tool (e.g. doxygen)



Bit Manipulation

Hardware Interfacing

Interaction with hardware often via bits in a memory element (e.g. byte or word)

Example: Motor control via register (at `MOTOR_REG`)

Bit Range	Access	Name	Description
0	Write only	Enable bit	0 = disable, 1 = enable
1	Read only	Error status	0 = no error, 1 = error present
2-5	Write only	Motor speed	Range 0000 speed 0 to 1111 top speed
6-7	Write only	LED color	00 = OFF, 01 = GREEN, 10 = YELLOW, 11 = RED

Manipulation with C's bit-operators

Bit Masks

- Use `#define` to create bit masks

- Example

```
#define MOTOR_OFF      0x00
#define MOTOR_ERROR    0x02
#define MOTOR_INIT     0x61
    // 0x61 = 01 1000 01 means
    // GREEN, medium speed, device enabled
```

```
MOTOR_REG &= MOTOR_INIT;
if (MOTOR_REG & MOTOR_ERROR)
    .....
```

Bit Manipulation I

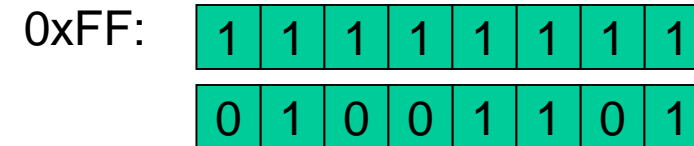
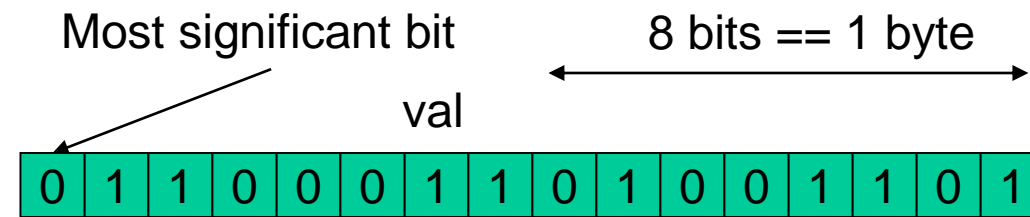
&	and
	inclusive or
^	exclusive or
<<	left shift
>>	right shift
~	one's complement

Some processors offer more operators, e.g. shift with wrap around. They are only usable through assembler

a:	1 0 1 0 1 0 1 0	0xAA
b:	0 0 0 0 1 1 1 1	0x0F
a&b:	0 0 0 0 1 0 1 0	0x0A
a b:	1 0 1 0 1 1 1 1	0xAF
a^b:	1 0 1 0 0 1 0 1	0xA5
a<<1:	0 1 0 1 0 1 0 0	0x54
b>>2:	0 0 0 0 0 0 1 1	0x03
~b:	1 1 1 1 0 0 0 0	0xF0

Bit Manipulation II

- Bitwise operations



- Example:

```
void f(uint16_t val) {           // 2-byte short integer
    uint8_t right = (uint8_t) (val & 0xFF);
        //rightmost (least significant) byte
    uint8_t left = (uint8_t) ((val>>8) & 0xFF);
        //leftmost (most significant) byte
    bool negative = val & 0x8000;    // sign bit (if 2's complement)
    ...
}
```

Bit Manipulation III

■ Exclusive or (xor) ^

- $a \wedge b == (a | b) \ \& \ \sim (a \& b)$ “either a or b but not both”
- Immensely important in graphics and cryptography

a:

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 0xAA

b:

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 0x0F

$a \wedge b$:

1	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

 0xA5

■ Swap two values (better use temporary variable!)

```
void swapXor(int16_t * a, int16_t * b) {  
    if (a != b) {  
        *a ^= *b;  
        *b ^= *a;  
        *a ^= *b;  
    }  
}
```

Can we leave out
the if statement?

Common Bit Manipulation Techniques I

- Let n be the number of a bit where 0 is number of the least significant bit

- **Set** a bit

$$a \mid= (1 \ll n) ;$$

- **Clear** a bit

$$a \&= \sim (1 \ll n) ;$$

- **Toggle** a bit

$$a \wedge= (1 \ll n) ;$$

- **Test** a bit

$$(a \& (1 \ll n)) \neq 0$$

- **Lowest bit not set :**

$$\sim a \& (a + 1)$$

Which one is better?

$$(a \gg n \& 1) \neq 0$$
$$(a \& (1 \ll n)) \neq 0$$

Common Bit Manipulation Techniques II

- Counting all set bits of an integer:

```
uint16_t countBitsLoop(uint16_t x) {  
    uint16_t result = 0;  
    // strip one set bit per iteration  
    while (x != 0) {  
        x &= x-1;  
        result++;  
    }  
    return result;  
}
```

- Is power of two:

```
bool isPowerOfTwo(uint16_t x) {  
    return ((x & (x - 1)) == 0);  
}
```

Example: Bit Manipulation

- Tiny Encryption Algorithm (TEA)
 - One of the fastest and most efficient cryptographic algorithms in existence
 - Originally by David Wheeler
 - <http://143.53.36.235:8080/tea.htm>
- Intention of this example is to give the flavor of bit manipulation code
 - It takes 8 bytes at a time and encrypts them
 - Key length is 128 bit
- TEA has a few weaknesses!!!
- Code is not meant to be self-explanatory!

TEA I

```
void encipher(const uint32_t * const v, uint32_t * const w,  
              const uint32_t * const key) {  
    uint32_t y = v[0];  
    uint32_t z = v[1];  
    uint32_t sum = 0;  
    uint32_t delta = 0x9E3779B9;  
    uint32_t n = 32;  
  
    while (n-->0) {  
        y += (z << 4 ^ z >> 5) + z ^ sum + key[sum&3];  
        sum += delta;  
        z += (y << 4 ^ y >> 5) + y ^ sum + key[sum>>11 & 3];  
    }  
    w[0]=y;  
    w[1]=z;  
}
```

TEA II

```
void decipher(const uint32_t * const v, uint32_t * const w,  
             const uint32_t * const key) {  
    uint32_t y = v[0];  
    uint32_t z = v[1];  
    uint32_t sum = 0xC6EF3720;  
    uint32_t delta = 0x9E3779B9;  
    uint32_t n = 32;  
  
    // sum = delta<<5; in general, sum = delta * n  
    while(n-->0) {  
        z -= (y << 4 ^ y >> 5) + y ^ sum + key[sum>>11 & 3];  
        sum -= delta;  
        y -= (z << 4 ^ z >> 5) + z ^ sum + key[sum&3];  
    }  
    w[0]=y;  
    w[1]=z;  
}
```



Coding Standards

Coding Standards

- Coding standard: A set of rules for what code should look like
 - Typically specifying naming and indentation rules
 - E.g., Indian Hill Style
- Typically specifying a subset of a language
 - E.g., don't use dynamic memory allocation
- Typically specifying rules for commenting
 - E.g., every function must have a comment explaining what it does
- Often require use of certain libraries
- Misuse
 - Organizations often try to manage complexity through coding standards
 - Often they fail and create more complexity than they manage

Coding standards

- A good coding standard is better than no standard
 - Don't start a major multi-person industrial project without one
- But also: A poor coding standard can be worse than no standard
 - E.g., C++ coding standards that restrict programming to something like the C subset do harm
- All coding standards are disliked by programmers
 - Even the good ones
 - All programmers want to write their code exactly their own way
- A good coding standard
 - is prescriptive as well as restrictive
 - “Here is a good way of doing things” as well as
 - “Never do this”
 - gives rationales for its rules and examples

Coding standards -- Sample rules

- No function shall have more than 200 lines (30 is even better), not counting comments
- Each new statement starts on a new line
 - `int a = 7; x = a+7; f(x,9); // violation!`
- No parameterized macros shall be used
- Identifiers should be given descriptive names
 - May contain common abbreviations and acronyms
 - When used conventionally, x, y, i, j, etc., are descriptive
- Type names and constants start with a capital letter
 - E.g., Device_driver and Buffer_pool
- Identifiers shall not differ only by case
 - E.g., Head and head // violation!

Coding standards -- Sample rules

- Identifiers in an inner scope should not be identical to identifiers in an outer scope
 - `int x = 9; { int x = 7; ++x; }`
// violation: x hides x
- Declarations shall be declared in the smallest possible scope
- Variables shall be initialized
 - `int x; // violation: x is not initialized`
- Code should not depend on precedence rules below the level of arithmetic expressions
 - `x = a*b+c; // ok`
 - `if (a<b || c<=d)`
// violation: parenthesize (a<b) and (c<=d)
- Increment and decrement operations shall not be used as sub expressions
 - `int x = v[++i];`
// violation (increment might be overlooked)

Coding Standards: *-ability

- Common aims
 - Reliability
 - Portability
 - Maintainability
 - Testability
 - Reusability
 - Extensibility
 - Readability



Declarations and Definitions

Declaration vs. Definition

- Definition allocates storage and initializes variable
 - `int c = 20;`
definition – reserves enough memory to hold an int
- Declaration gives meaning to an identifier
 - it defines the type information of an identifier
 - it allows compiler to generate correct object code to access the variable based on its size
 - no memory is allocated!
- A definition is implicitly a declaration
 - DEFINITION = DECLARATION + SPACE RESERVATION
- When compiling a source file, a variable **must** be declared **before** it is used (otherwise compiler error)
- The linker **requires** definitions in order to link references to the corresponding entities

Syntax

- Syntax of declarations and definitions of variables
 - if it has an “=” it’s a definition
 - `int c = 20;`
 - otherwise, if it has “extern” and no “=” it’s a declaration
 - `extern int c;`
 - it must be defined in another file
 - `extern int i=5;`
 - that is a definition, extern is ignored
 - otherwise it’s a tentative-definition that may become a declaration or a actual-definition
 - `int c;`

Examples

- **Example:**

```
int main(void) {  
    c = 10;      /* error: c hasn't been declared */  
    return 0;  
}  
uint16_t c = 20; /* definition - allocates 2 bytes */
```

- **An object declaration does not reserve memory**

```
extern uint16 c;  
/* declaration: no memory reserved but defines sizeof(c) */  
int main(void) {  
    c = 10; /* okay to use c */  
    return 0;  
}  
uint16 c = 20;  
/* definition - memory reserved & initialized */
```

Declarations and definitions in C

- If no declaration is encountered before definition, then definition acts as an implicit declaration
- In a compiled source file there may be only **one definition** for an identifier, but there may be multiple declarations (as long as they agree)
- With a tentative definition:
 - If an actual definition is found later in source file, then tentative definition just acts as a declaration
 - If end of source file is reached and no actual definition is found, then tentative definition acts as an actual definition (and implicit declaration) with an initialization of 0

Tentative definition

■ Example:

```
int c;  
/* tentative definition becomes declaration */  
  
int td;  
/* tentative definition becomes actual definition initialized to 0 */  
  
int main(void) {  
    ...  
    return 0;  
}  
int c = 20; /* actual definition */
```



Functions

Function Declaration and Definition

- A function definition includes the function's body
- A function's declaration (called its *prototype*) makes compiler aware there is a valid function with this identifier

- Example:

```
void f(int p); /* declaration */

int main(void) {
    f(10); /* okay to call f as declared */
    return 0;
}

void f(int p) /* definition */ {
    // ...
}
```

- A function can be declared several times in a program, provided that all the declarations agree, but it can have only one definition

Calling a Function

- On call of function `f` in `main`, the declaration enables compiler to construct correct call frame based on:
 - validity of identifier
 - storage required to pass any parameters (by stack or register)
 - storage required for any return information

- This code does not compile

```
int main() {  
    f(20); /* call f with no declaration */  
    return 0;  
}  
  
void f(int i) {  
    /* definition and implicit-declaration */  
    // ...  
}
```

- Why?

Calling a Function II

- Called a function that hasn't been declared
- An identifier must be declared before being used, but this only applies to variables and not to functions!
- With functions, if no declaration is found before its first call, compiler creates an **implicit declaration** with return type `int`
 - If `f`'s return type is changed to `int`, then code compiles
- Worse: A common mistake is to assume that an empty parameter list means the same as `void` in parameter list
 - In some cases it does and in others it doesn't

Function Declarations and Prototypes

- For declarations this isn't the case:

```
void f();           /* declaration */  
void f(void);      /* prototype-declaration - not the same  
as above */
```
- If a declaration has a parameter list (including void) then it becomes a prototype-declaration
 - Standard states that number and types of arguments are **not** compared with those of parameters in a function definition that does **not** include a function prototype-declaration
 - If there is an empty parameter list the compiler assumes that arguments to the call are correct

Examples

- Legal C code:

```
void f(); /* declaration */
int main(void) {
    f(20); /* okay to call f as declared */
    return 0;
}
void f(int i) /* definition */ { /* ... */ }
```

- Code with undefined behaviour:

```
void f(); /* declaration */
int main(void) {
    f(20); /* okay to call f as declared */
    return 0;
}
void f(void) /* definition */ { /* ... */ }
```

Guideline:
For all functions
always supply a
function prototype-
declaration!



Variable Lifetime and Scope

Lifetime of Variables I

- Lifetime of an object: Time in which memory is reserved while program is executing
 - static, automatic, dynamic

- Example:

```
int global_a; /* tentative defn; becomes actual defn init to 0 */
int global_b = 20; /* defn and implicit decl */
```

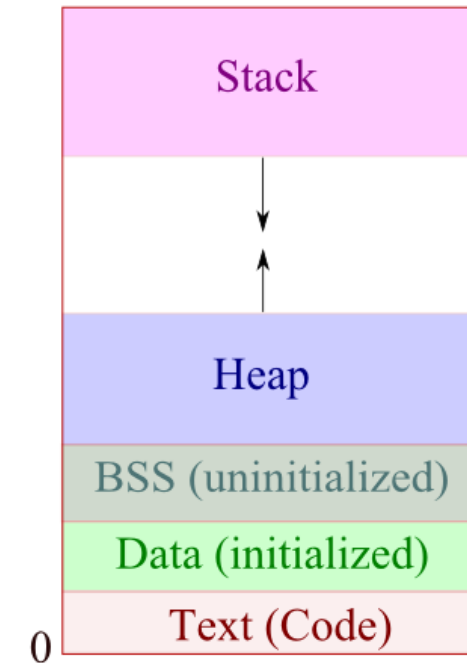
```
int f(int * param_c) {
    int local_d = 10;
    ...
    return local_d;
}
```

```
int main(void) {
    int * ptr = malloc(sizeof(int)*100);
    ...
    global_a = f(ptr);
    ...
    free(ptr);
}
```

- global_a and global_b are static
- Memory allocated by call to malloc is dynamic.
- All others are automatic (including ptr, param_c, local_d and return value from function f).


Lifetime of Variables II

- Placement of the definition affects lifetime
- Static objects
 - All variables defined outside functions
 - Memory is allocated at compile/link time
 - Static region is subdivided into two parts
 - one for initialized-definitions `int global_b = 20;`
 - one for uninitialized-definitions `int global_a;`
- Automatic objects
 - Almost all variables defined within functions
 - including parameters, temporary-returned-object from a function
 - Memory is allocated from the stack and thus is only available while function is executing
 - Automatics are not explicitly initialized



Lifetime of Variables III

Two different meanings
of `static`!



- Dynamic objects
 - Memory must be allocated by programmer
 - Memory is allocated from the heap (not the stack)
 - Lifetime is
 - under control of programmer rather than C run-time system
 - from allocation until call to either `free` or `realloc`
- Static local variables
 - Keyword `static` can be applied to local variables
 - This changes object's lifetime from automatic to static
 - Static local variables
 - retain their value from function call to function call
 - are initialized only in the first call of the function

Lifetime of Variables IV

- Example:

```
void f(void) {
    static int slocal = 10; /* static local */
    int alocal = 10;        /* automatic local */
    printf("In f: slocal = %d, alocal = %d\n",
           slocal, alocal);
    slocal++; alocal++;
}

int main(void) {
    f(); // In f: slocal = 10, alocal = 10
    slocal = 11, alocal = 10
    f(); // In f: slocal = 12, alocal
    = 10
}
```

Dynamic Memory

- Allocation/de-allocation is performed with library functions `malloc()` and `free()` from the heap
- Size and location of heap is specific to a particular toolchain
- Dynamic memory allocation is typically non-deterministic and heap space may be insufficient
 - Fragmentation of heap can lead to unpredictable failures
- Alternative to `malloc()` offered in real time operating system: Allocation of memory in fixed size blocks
 - one or more pools of memory blocks are defined
 - pools have a specific, fixed block size and a defined number of blocks
 - multiple pools facilitate a choice of block size

Scope I

- Scope of a variable: Part of program where variable can be accessed (i.e. where it is *visible*)
 - File scope
 - Variables declared with file scope can be accessed by any function defined after declaration
 - Block scope
 - Block scope is defined by the pairing of curly braces { and }
 - Avoid overlapping scopes!
- Good programming practices limit scope as much as possible
- General case
 - Static objects: file scope
 - Automatic objects: block scope (i.e. function scope)

Scope II

- Variables with file scope can be accessed by any function in whole program
- If a variable is defined with file scope in one file, but is required in another, then it can be brought into scope using a declaration with `extern` storage-class specifier
- **Example:** file `a.c`

```
int global_a = 10;    /* definition of global_a */

int f(int * param_c) {
    int local_d = param_c;
    static int local_s = 10;
    local_s = global_a;
    return local_d;
}
```

Scope III

- Example (cont.)

```
/* file main.c */

extern int global_a;
/* declaration of global_a, now visible */

int f(int*);

int main(void) {
    int *ptr = malloc(sizeof(int)*100);
    ...
    global_a = f(ptr);
    /* global_a visible so can be accessed */
    ...
    free(ptr);
}
```


External and Internal Linkage

- By default all variables declared outside of a function have external linkage
 - Objects with external linkage are considered to be in outermost program level
 - All instances of a particular name with external linkage refer to the **same** object in program
- Case:
 - A variable with static lifetime is needed but it should not be globally accessible (i.e. limit its use to functions in current file)
 - It cannot be defined as a local static as it is needed in multiple local functions
- This can be achieved with keyword `static`
 - but this time to affect scope rather than lifetime
- Use of `static` in this way decreases coupling and furthers encapsulation
- File scoped variable tagged as `static` have internal linkage

Two different meanings
of `static`!



External and internal linkage

- Example:

```
int global_a = 10;          /* ext. linkage: global scope */
static int internal_b;      /* int. linkage: this-file scope */

int f(int * param_c) {
    int local_d = param_c;  /* function scope, auto */
    static int local_s = 10; /* function scope, static */

    local_s = global_a;
    return local_d;
}
```

- If another file declares `internal_b` as `extern`, then this results in a link-time error
- Internal linkage can also be applied to functions
 - Functions have external linkage by default
 - It's good practice to declare a function as `static` if it's only being used in current file



Types and Type Declarations

Types

- ISO C standard allows implementation-defined widths for char, short, int, long, and long long types, leading to portability problems
- This was changed in the ISO C99 standard
- Header <stdint.h> declares integer types having specified widths
- Important type names:

```
typedef signed char int8_t;  
typedef unsigned char uint8_t;  
typedef short int16_t;  
typedef unsigned short uint16_t;  
typedef long int32_t;  
typedef unsigned long uint32_t;  
typedef long long int64_t;  
typedef unsigned long long uint64_t;
```

int8_t	signed 8-bit
uint8_t	unsigned 8-bit
int16_t	signed 16-bit
uint16_t	unsigned 16-bit
int32_t	signed 32-bit
uint32_t	unsigned 32-bit
int64_t	signed 64-bit
uint64_t	unsigned 64-bit

Bit Fields

- Purpose
 - to compactly store a value as a short series of bits
 - used to represent integral types of known, fixed bit-width
- Bit fields
 - typically fit within one machine word (save storage in RAM)
 - ordering is compiler/processor specific
 - Padding may occur (maybe mapping to 4 bytes)
 - Example

```
struct status {  
    uint8_t running : 1; // range 0,1  
    uint8_t blink : 1;  
    uint8_t battery : 1;  
    uint8_t foo : 4; // range 0, 1, ..., 15  
};
```

```
struct status s; // Just 1 byte  
s.battery = 1;  
s.blink = 0;  
if (s.running == 1) { ...
```

Enumerated Types

- Enumerated type definition defines a type along with associated constant values of that type
 - Example:
 - `enum color {RED, GREEN, BLUE};`
 - By default, RED has value 0, GREEN has value 1, and BLUE has value 2
 - Alternative:
`enum color {RED = 1, GREEN = 2, BLUE = 4};`
- Unlike macros, enumeration constants obey scope rules
- Enumeration constants must have integer values
- Use `const` definition if you want other types
- `const` objects may incur a performance penalty, which enumeration constants avoid

Pointers I

■ Pointer types

- Example: `long * circle;`
 - To declare a variable as a pointer, precede its name with an asterisk
 - Standard also allows to attach the asterisk to type name such as `long* circle;` don't do this

- What about: `long* first, second;`

- Generic pointer: `void * square;`

- The pointed type can take all of usual keywords:

```
long int * rectangle;
unsigned short int * rhombus; //Better to use C99!!
const char * kite;
// kite is a (non-const) pointer to a const char
char * const pentagon = &some_char;
// pentagon is a constant pointer, pointing at a char
```

Pointers II

- Pointers to pointers

- `char ** heptagon;`
- Usage examples
 - Returning pointers from functions, via pointer arguments rather than as formal return value
 - Dynamically allocated, simulated multidimensional arrays

- Arrays

- `int cat[10];` // Valid in C
- `int[10] cat;` // This is not valid in C

- Arrays of arrays

- `double entries[5][12];`

- Arrays of pointers

- `char * locks[10];`

- Pointers to arrays

- `double (* heights)[20];`

- When array syntax is used as type of a parameter of a function it is identical to declaring that argument as a pointer. The following definitions are all identical:

```
void foo(int *x);      // pointer syntax
void foo(int x[]);     // array syntax
void foo(int x[10]);   // array syntax
```

Issues with C Types

- C's declaration syntax is trivial for a compiler to process, but hard for the average programmer
- Problem: Declarations cannot be read from left to right
- Examples:
 - `Char * const * (* next) ();`
 - next is a pointer to a function returning a pointer to a const pointer-to-char
 - `char * (* c[10]) (int ** p);`
 - c is an array[0..9] of pointer to a function returning a pointer-to-char

Precedence Rule for C Type Declarations I

1. Declarations are read by starting with the name (first identifier from left) and then reading in precedence order
2. Precedence, from high to low, is:
 - a) parentheses grouping together parts of a declaration
 - b) postfix operators:
parentheses () indicating a function, and square brackets [] indicating an array
 - c) prefix operator: the asterisk denoting “pointer to”
3. If a `const` resp. `volatile` keyword is next to a type specifier (e.g. `int`, `long`) it applies to the type specifier. Otherwise the `const` resp. `volatile` keyword applies to the pointer asterisk on its immediate left

Precedence Rule for C Type Declarations II

- **Example:** `char * const * (* next) ();`
 1. Go to variable name, `next`, note that it is directly enclosed by parentheses
 2. Group it with what else is in parentheses, to get “next is a pointer to ...”
 3. Go outside the parentheses, and have a choice of a prefix asterisk, or a postfix pair of parentheses
 4. This rule tells us the highest precedence thing is the function parentheses at right, hence “next is a pointer to a function returning ...”
 5. Then process the prefix `*` to get “pointer to”
 6. Finally, take `char * const`, as a constant pointer to a `char`
- **Result:** `next` is a pointer to a function returning a pointer to a const pointer-to-char

Keyword `const`

- `const` shall be used whenever possible:
 - To declare variables that should not be changed after initialization
 - To define call-by-reference function parameters that should not be modified (example: `const char * p_data`)
 - To define fields in structs and unions that cannot be modified (such as in a struct overlay for memory-mapped I/O peripheral registers)
 - As a strongly typed alternative to `#define` for numerical constants
- Always use symbols, rather than raw numbers, to represent arbitrary constant values!
- Note the difference:

```
const char * pt          // pointer to const char
char * const pt          // const pointer to char
const char * const pt    // const pointer to const char
```


Keyword `const` – Examples

- `uint16_t const max_temp = 1000;`
 - Creates a 16-bit unsigned integer value of 1000 with a scoped name of `max_temp`
 - Variable will exist in memory at run time, but will typically be located, by linker, in a non-volatile memory area e.g. ROM
 - Any reference will result in a read from that location
 - Attempts to write to a `const` variable directly result in a compile-time error
- `uint8_t const * p_latch_reg = 0x10000000;`
 - Attempts to write to that physical memory address via pointer (e.g., `*p_latch_reg = 0xFF;`) results in a compile-time error

Function Parameters

A. Inputs (Read-only)

- caller-supplied objects manipulated within function only

B. Outputs (Write-only)

- objects generated by function for use by caller

C. Input-Outputs (Read-Write)

- caller objects that can be manipulated by function

■ C uses call-by-value paradigm

- When a C function is called the compiler sets up a call frame that holds copies of function parameters
- Changes to parameter in function have no effect on original value

Function Parameters

- Input parameters should be passed as a `const` parameter (in particular for `structs`)
 - Compiler has opportunity to perform a lazy evaluation, i.e. pass address of parameter instead of making a copy
 - To really avoid a copy pass a pointer to a `const` object
 - By passing a 'raw' pointer the function can manipulate the caller's object, this is avoided by the `const`
 - To also avoid that the function changes the pointer, make it a `const-pointer-to-const-object`
- The only real mechanism for output parameters is function return values
 - Note: Return value is copied, thus for big data structures input-output parameters are to be preferred



Keyword volatile

Keyword `volatile` I

- `volatile` shall be used whenever appropriate:
 - To declare a global variable accessible
 - by any interrupt service routine
 - by two or more tasks
 - To declare a pointer to a memory-mapped I/O peripheral register set
Example: `volatile timer_t * const p_timer`
Address of timer register is fixed while contents of register may change at any time
- To declare a variable volatile, include keyword `volatile` before or after data type in variable definition
- Proper use of volatile eliminates difficult to detect bugs by preventing compiler from making optimizations that would eliminate requested reads or writes to variables or registers that may be changed at any time by a parallel-running entity
- Improper usage of volatile causes runtime penalty!

Keyword `volatile` II

- Example: Serial port interrupt tests each received character to see if it is an ETX character (signifying the end of a message). If character is ETX, ISR sets a global flag
- Incorrect implementation:

```
uint8_t etx_rcvd = FALSE;
void main(void) {
    ...
    while (!etx_rcvd) {
        // Wait
    }
    ...
}

interrupt void rx_isr(void) {
    ...
    if (ETX == rx_char) {
        etx_rcvd = TRUE;
    }
    ...
}
```

- Code works if compiler optimization is turned off
- Compiler has no idea that `etx_rcvd` can be changed within ISR, hence compiler believes that expression `!etx_rcvd` is always true, and you can never exit the while loop
- Consequence, all code after while loop may simply be removed by the optimizer

Keyword `volatile` III

- Quoting the standard:
 - An object that has `volatile`-qualified type may be modified in ways **unknown** to the implementation or have other unknown side effects
 - Expressions referring to such an object shall be evaluated strictly according to the rules of the abstract machine
 - Actions on objects declared `volatile` shall not be optimized *out* by an implementation or reordered

volatile and const I

- volatile ("ever-changing") and const ("read-only") are orthogonal
- In some scenarios they are used together
 - Constant addresses of hardware registers
 - `uint8_t volatile * const p_led_reg = (uint8_t *) 0x80000;`
(`p_led_reg` is a constant pointer to a volatile 8-bit unsigned integer)
 - Advantage:
 - This error is detected: `p_led_reg = LED1_ON;`
 - `*p_led_reg = LED1_ON;` (real intention)

volatile and const II

- Read-only hardware register
 - In addition to enforcing compile-time checking so that software doesn't try to overwrite memory location, you also need to ensure that all requested reads actually occur
 - ```
uint8_t const volatile * const p_latch_reg =
 (uint8_t *) 0x10000000;
```
- Read-only shared-memory buffer
  - Two processes communicate via a shared memory area
  - Code for side that only reads from a shared memory buffer

```
int const volatile comm_flag;
uint8_t const volatile comm_buffer[BUFFER_SIZE];
```

# Keywords to avoid

- The `auto` keyword is an unnecessary historical feature of the language
- The `register` keyword presumes the programmer is smarter than the compiler
- Keywords `goto`, `continue`, and `break` often lead to spaghetti code
  - Avoid `goto`
  - Use `continue` and `break` with care!

# Predictability

- C/C++ operations execute in constant, measurable time
  - E.g., you can simply measure the time for an add operation or a virtual function call and that'll be cost of every such add operation and every virtual function call
    - Pipelining, caching, implicit concurrency makes this trickier on modern processors
- With the exception of:
  - Free store allocation (`new`)
  - Exception thrown
- So `throw` and `new` are typically avoided in hard real-time applications
- Not just in C/C++ programs
  - Similar operations in other languages are similarly avoided



10

Advice



# Advice I

- One source of errors is complicated problems
  - Inherent complexity
- Another source of errors is poorly-written code
  - Incidental complexity
- Reasons for unnecessarily complicated code
  - Overly clever programmers
    - Who use features they don't understand
  - Undereducated programmers
    - Who don't use the most appropriate features
  - Large variations in programming style

# Advice II

- Keep the highest level of abstraction
  - Don't write glorified assembler code
  - Represent your ideas directly in code
- Try to write clear, clean, maintainable code
- Don't optimize until you have to
  - People far too often optimize prematurely
- John Bentley's rules for optimization
  - First law: Don't do it
  - Second law (for experts only): Don't do it yet

# Advice III

- How to reduce code size
  - Compile with full size optimization
  - Use local variables whenever possible
  - Use smallest applicable data type., i.e. use unsigned if applicable
  - If a non-local variable is only referenced within one function, it should be declared static
  - Collect non-local data in structures whenever natural. This increases possibility of indirect addressing without pointer reload
  - Use descending loop counters and pre-decrement if applicable
  - Use `for(;;) { }` for eternal loops (gen. single jump statement)
  - Use `do { } while(expression)` if applicable
  - Use inline functions (or macros) for tasks that generates less than 2-3 lines assembly code
  - Reduce size of Interrupt Vector segment (INTVEC) to what is actually needed by application

# Advice IV

- How to reduce RAM requirements
  - All constants and literals should be placed in Flash by using the `Flash` keyword.
  - Avoid using global variables if variables are local in nature. This also saves code space. Local variables are allocated from stack dynamically and are removed when function goes out of scope
  - If using large functions with variables with a limited lifetime within the function, the use of sub-scopes can be beneficial
  - Check applicability of linker options



# SES

## Appendix: C for Embedded Systems

Prof. Dr.-Ing. Bernd-Christian Renner

