Curs 6

C Language Programming

Why C?

- **The language suitable for embedded systems**

- Microcontrollers accept instructions in machine code
- All software: assembly, C, C++, Java must be translated into machine code in order to be executed by the CPU
- Interpretation of machine code by the engineer is error prone and time consuming
- Efficiency of the programming language: embedded processors have limited processor power and limited available memory

Why C?

- **The language suitable for embedded systems**
- Compromise between safety, maintainability, portability
- Characteristics of the required language:
  - efficient
  - high-level
  - offers low-level access to hardware
  - well defined
  - in common use
  - read from and write to particular memory locations (e.g. by a mechanism such as “pointers”)
  - reuse the code (which has already been tested)

C is a solution

Why C?

- **Features of the C programming language:**
- "mid-level"
  - "low-level" features (access to hardware via pointers)
  - "high-level" features (support for functions and modules)
- In common use
- Different compilers are available for every embedded processor (8-bit to 32-bit or more)

ANSI C – Identifiers

- C Program

Why C?

ANSI C

Identifiers

Numbers, Variables

Expressions

Program flow

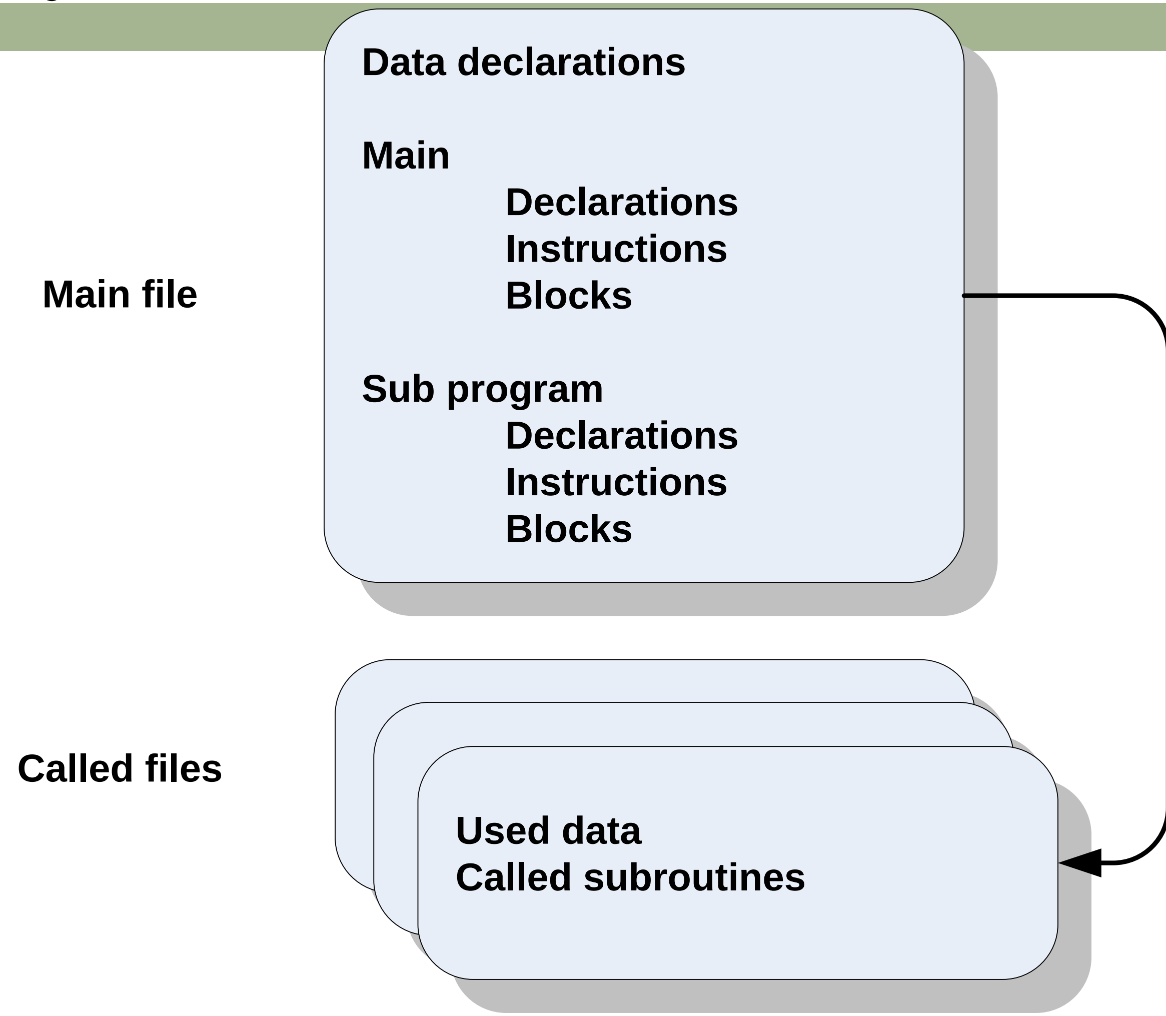
Arrays & Pointers

Functions

Preprocessor

The Embedded Difference

Examples



ANSI C – Identifiers

- **Tokens:**
  - identifiers
  - keywords
  - constants
  - string literals
  - operators
  - other separators
  - blanks, horizontal and vertical tabs, new lines and comments (“white spaces”) - ignored

ANSI C – Identifiers

- **Identifiers:**
  - a sequence of letters and digits
  - functions, structures, unions and enumeration members of structures or unions, enumeration constants, defined types (typedef)
- **Rules:**
  - The first character must be a letter
  - The underscore _ counts as a letter
  - Upper and lower case letter are different
  - Identifiers may have any length and for internal identifiers, at least the first 31 characters are significant

ANSI C – Identifiers

- The compiler allocates memory for all identifiers
- As the compiler reads a program, it records all identifier names in a symbol table.
- The compiler uses the symbol table internally as a reference to keep track of the identifiers: name, type and the location in memory
- When the compiler finishes translating a program into machine language, it will have replaced all the identifier names used in the program with instructions that refer to the memory addresses associated with these identifiers

ANSI C – Identifiers

Why C?
ANSI C
Identifiers
Numbers, Variables
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

- **An identifier can not be a C Keywords:**

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

- Some implementations also reserve the **asm** keyword

ANSI C – Identifiers

- **Variable Data Identifiers:**
  - Identifiers which represent variable data values, called variables, require portions of memory which can be altered during the execution of the program
  - The compiler will allocate a block of its data memory space, usually in RAM, for each variable identifier
    - **Example: variable declaration**

      int currentTemperature; / will cause the compiler to allocate 2 or 4 bytes of RAM */*
  - The keyword *int* in the variable declaration tells the compiler that *currentTemperature* will contain an integer value and will require 2 or 4 bytes of RAM to contain this value

ANSI C – Identifiers

- **Constant Data Identifiers:**
  - allocated from computer program memory space
  - constant data values do not require alterable memory: once the value of a constant has been written in memory it need never change => the compiler will allocate a block of its program memory space, usually in ROM, for each of these identifiers (most of the cases)
    - **Example: constant data value declaration**

      const int maxTemperature = 20;
  - the keyword *const* tells the compiler that the identifier is a constant and that 2 or 4 bytes in ROM should be reserved to contain the value 20
  - when the identifier *maxTemperature* is used in the program it refers to the memory location in ROM which contains the value 20.

ANSI C – Identifiers

- **Function Identifiers:**
  - not altered during program execution
  - Once the value of a function has been written in the computer's memory it need never change
  - When a function is defined, the compiler places the program instructions associated with the function into ROM
  - **What happens to the local variables used in a function's body of statements?**
  - The compiler will write in the data memory, addresses where local variable values will be stored in RAM when the program runs
  - Each program has a *main()* function

ANSI C – Data Types

- Numbers
- Variables
- Memory class modifier
- External variables
- Scope rules
- Static variables
- Register variables
- Initialization

- Data Types
- Data encapsulation with *struct*
- Data encapsulation with *union*
- Bit fields
- Enumerated data types
- Definition of private types
- Type Conversion

ANSI C – Data Types

- **Constants**
  - Text constants
  - Numerical constants

- **Text constants**
  - **char** Ex: ‘P’, ‘R’, ‘7’, ‘#’
  - **string** Ex: “Politehnica”

- For string constants an array of appropriate length is reserved and the character ‘\0’ is automatically added to the end. In this way the string end can be recognized by the program.

ANSI C –Data Types

- **Text constants**
- **Example: String Constant**

‘S’	‘y’	‘s’	‘t’	‘e’	‘m’	‘s’	‘\0’
53	79	73	74	65	6D	73	00

A char constant doesn’t end with the terminal ‘\0’

ANSI C –Data Types

- **Numerical constants**
  - Numbers always start with a digit
  - Expressing values in different notations except the decimal:
    - **0x**nnnn hexadecimal number (digits: 0-9, a-f or A-F)
    - **0**nnn octal number (digits: 0-7)
    - **0b**nnnn binary number (digits: 0,1)
      - **Example:**

```
// all octal values begin with 0
int octalInt = 030;
// all hex values begin with 0x
int hexadecimalInt = 0x40;
// all binary values begin with 0b
int binaryInt = 0b00100000;
```

ANSI C –Data Types

- **Numerical constants**
  - Long integer constants are specified with an ending **L**
    - **Example:** long var1 = 11L;
  - Unsigned integer constants are specified with an ending **u**
    - **Example:** unsigned int var2 = 5u;
  - Floating point values should contain a decimal point,
    - **Example:** float var3 = 2.;
  - Floating point values may be written with mantissa and exponent,
    - **Example:** float var4 = 0.4e-5;
- The suffixes may be omitted, but this might sometimes result to an error

- **Symbolic definition of Constants**
  - Symbolic constants are defined with the pre-processor directive **#define**
    - **Example:**

```
#define MAX 350
#define TEXT "Embedded Systems"
```

  - It is recommended to define a symbolic constant for each constant value for improved readability

ANSI C –Data Types

- **Symbolic definition (#define) vs. const**
  - constants declared with **const** are stored in ROM - disadvantage
  - constants declared with **const** are visible in debuggers - advantage
  - symbolic constants do not take up space in ROM, but are not visible in debuggers;
  - Symbolic constants are recommended for embedded systems where ROM memory is critical

ANSI C –Data Types

- **Variables** - characterized by:
  - value
  - address
  - attribute (modifier)
  - type
  - Lifetime

[class] [modifier] [type] <name> [, <name1>][,...]

– When the compiler comes across a variable declaration it checks that the variable has not previously been declared and then allocates an appropriately sized block of RAM

ANSI C –Data Types

- **Variables**
  - The modifier can be one of:
    - **volatile**
    - **Const**
  - **Volatile**
    - volatile <*type*> <*name*>
- directs the compiler not to perform certain optimizations on an object because that object can have its value altered in ways beyond the control of the compiler
- must always be read from its original location and is not kept in a register at any time
- compiler optimizations are not available, therefore it must be used only if necessary
- useful for controlling access to memory-mapped device registers, as well as for providing reliable access to memory locations used by asynchronous processes

ANSI C –Data Types

- **Const**
  - const <*type*> <*name*>
    - allows only read access to the variable
    - may only be initialized once in a program, the initialization being performed in the startup code
    - allows the compiler to perform type checking
    - announces objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization

ANSI C –Data Types

- **Memory class modifiers**

memory_class	local object	global object
`auto`	The object shall be located on the stack. As this is default the `auto` modifier may be omitted.	Meaningless.
`register`	The object shall be located in a register, if possible.	Meaningless.
`extern`	Impossible.	The object is declared and used in the current module but defined in a different one.
`static`	The object shall be located in memory but not on the stack.	The object shall not be public and only accessible in the current module.

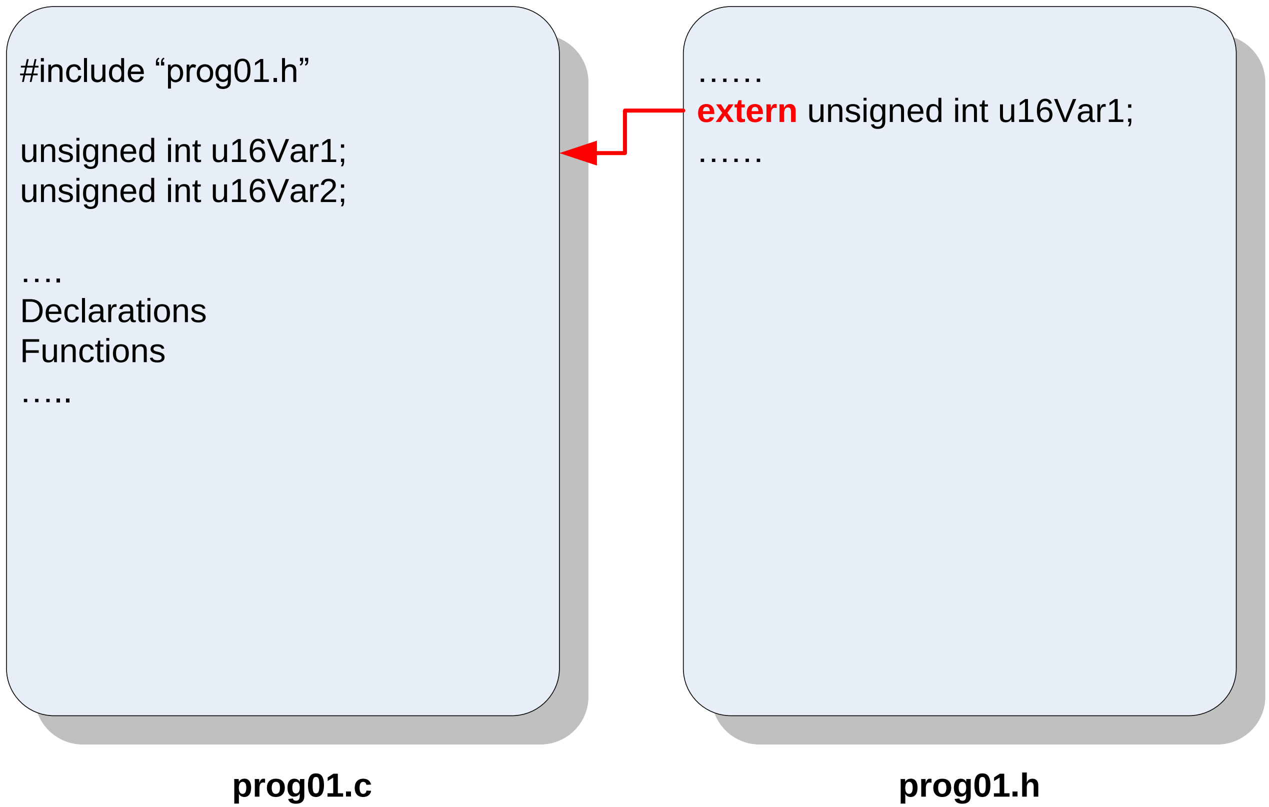
- **auto**
  - default storage class
  - local lifetime
  - is not initialized automatically, but explicitly
  - visible only in the block in which it is declared

ANSI C –Data Types

- **Memory class modifiers**
  - **static**
  - global lifetime
  - visible only within the block in which it is declared
  - **register**
  - mostly obsolete
  - used for optimizations for heavily used variables; the variable is assigned to a high-speed CPU register (rather than an ordinary memory location)
  - you cannot generate pointers to them using the & operator !
  - **extern:**
  - reference to a variable with the same name defined at the external level in any of the source files of the program
  - the identifier declared has its **defining** instance somewhere else
  - used to make the external-level variable definition visible within the block

ANSI C –Data Types

External variables – how to use them:



ANSI C –Data Types

Variable scope

- The visibility of a declared variable is called the variable's scope; if a portion of a program lies outside a variable's scope then the compiler will give an error if you refer to the variable in that portion
  - For automatic variables declared at the beginning of a function, the scope is the function in which the name are declared
  - Local variables of the same name in different functions are unrelated
  - The same is true of the parameters of the function, which are in effect local variables

ANSI C –Data Types

Variable scope

- The scope of an **external** variable or function lasts from the point at which it is declared to the end of the file being compiled; the compiler does not allocate memory when it sees an extern variable declaration

- There must be only one definition of an external variable among all the files that make up the source program; other files may contain "extern" declarations to access it

Variable scope

- What happens when scopes overlap? The most recently declared instance of a variable is used. If you declare a global variable called temp outside all statement blocks and a local variable called *temp* inside your *main()*, function, the compiler gives the local variable precedence inside *main()*.

- While the computer executes statements inside *main()*'s scope (or statement block), temp will have the value and scope assigned to it as a local variable. When execution passes outside *main()*'s scope, temp will have the value and scope assigned to it as a global variable

ANSI C –Data Types

- **Static variables**
  - the scope of the object is limited to the rest of the source file being compiled
  - function names are global, visible to any part of the entire program
  - a **static** function is invisible outside of the file in which it is declared
  - function local variables can also be declared as **static.** These are local to the function, but they remain in existence after the function terminates its execution.
  - **static** variables are allocated in the "global" memory space (heap)

ANSI C –Data Types

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

- **Register variables**
- makes sense for heavily used variables
- *register* variables are to be placed in machine registers, which may result in smaller an faster programs
- obsolete, compilers are able to optimize the code without such hints

ANSI C –Data Types

Variable initialization

- global variables are guaranteed to be initialized to zero
- automatic and register variables have undefined initial values
- for explicitly initialized global variables the initializer must be a constant expression
- for automatic and register variables, the initializer is not restricted to being a constant. It may be any expression involving previously defined values, even function calls

ANSI C –Data Types

- **Example:**

```
...
int able;
void main(void)
{
   long quickstart(void);
   long r;
   ...

   able=17;
   l=quickstart();
  …
}
```

file01.c

```
...
long quickstart(void)
{
   extern int able;
   ...
/* do something with able */
   …
  return result;
}
```

file02.c

ANSI C –Data Types

- **Example**
- When the file 1 is compiled, the variable *able* is marked as external, and memory is allocated for its storage. When the file 2 is compiled, the variable *able* is recognized to be external because of the *extern* keyword, and no memory is allocated for the variable
- All address references to *able* in file 2 will be assigned the address of *able* that was defined in file 1
- The example above in which the declaration

  extern int able;

  allowed access to *able* from the file 2 will not work if *able* had been declared as follows in file 1:

  static int able;

ANSI C –Data Types

Data Types

- act as filters between your program and computer memory
- Type definition is used for:
  - the definition of the range of values
  - the size of memory (the amount of memory the computer must reserve for a value of that type)
  - the operations allowed
  - the scaling of pointers

ANSI C –Data Types

- **Data types:**

data type	size [in bytes]	range of values
`void`	undefined	none
`signed char` `unsigned char`	1 1	-128 .. +127 0 .. 255
`signed short` `unsigned short`	2 2	-32768 .. +32767 0 .. 65535
`signed int` `unsigned int`	min. 2 min. 2	compiler dependent
`signed long` `unsigned long`	4 4	-2 147 483 648 .. +2 147 483 647 0 .. 4 294 967 295
`float`	4	+/-1.176e-38 .. +/-3,40e+38
`double`	8	+/-2,225e-308 .. +/-1,798e+308
pointers	1 .. 4	up to 32 bit addresses

ANSI C –Data Types

- **Data encapsulation - struct**
  - encapsulates several data of different types that belong to the same object
  - support the meaningful grouping of program data
  - Syntax:

```
struct [struct_name]
{
        data_type1 ivar_list1;
        data_type2 ivar_list2;
        ...
} [svar_list];
```

ANSI C –Data Types

- **Data encapsulation - struct**
  - **Example:** a structured type for the number shown by an LED display

```
struct Display_tag
{
    int DisplaySelected;
    int hundreds;
    int tens;
    int ones;
    char AorP;
};
```

- **Data encapsulation - struct**
  - **Example *(contd.)***: the compiler allocates no memory for the structure declaration itself because it is used solely as a template for variable declarations. When you declare a variable for a structure, the compiler will allocate an appropriate block of memory:

    struct Display_tag CurrentTime;
  - You must repeat the keyword *struct* because Display_tag is not a valid data type, it is a structure tag

ANSI C –Data Types

- **Data encapsulation – struct**
  - The access to a member of the structure is performed with the dot operator ”.” and the structure pointer operator, “->”
  - The order of the structure elements bears compiler specific consequences.
  - Usually, data are aligned to even address boundaries.

ANSI C –Data Types

- **Data encapsulation – struct**
  - If there is a structure declaration:

```
struct s1
{
        unsigned char cId;
        unsigned int iLength;
        unsigned char cMsg;
};
```

the compiler may reserve four, six, or even more bytes of memory, depending on the alignment

ANSI C –Data Types

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

Data encapsulation – struct

- alignment of structure members depends on the memory alignment of the system

byte number	byte aligned	word aligned
0	`s1.sid`	`s1.sid`
1	`s1.size` (low byte)	*<hole>*
2	`s1.size` (high byte)	`s1.size` (low byte)
3	`s1.msg`	`s1.size` (high byte)
4		`s1.msg`
5		*<hole>*

ANSI C –Data Types

- **Example:**

```
/* a structure for the number shown by an LCD display */
…
struct Display_tag * Display_Ptr;
struct Display_tag
{
   int DisplaySelected;
   int hundreds;
   int tens;
   int ones;
   char AorP;
}alarmTime;

...
void main()
{
Display_Ptr = &alarmTime; //point Display_Ptr to alarmTime
Display_Ptr->ones = 7; //set alarmTime.ones to 7
Display_ptr->AorP = 'P'; //set alarmTime.AorP to P
Display_Ptr->tens = 9; //set alarmTime.tens to 9
(*Display_Ptr).tens = 9; //set alarmTime.tens to 9
}
```

file01.c

ANSI C –Data Types

- **Data encapsulation – union**
  - A union means an overlay of elements of different data types to the same memory location.
  - Syntax:

```
union [union_name]
{
        data_type1 ivar_1;
        data_type2 ivar_2;
        ...
} [uvar_list];
```

ANSI C –Data Types

- **Data encapsulation – union**
  - The size of the union is determined by the size of its biggest member element
  - **Example:**

```
union
{
   unsigned char c[2];
   long l;
} u1;
```

- The size of u1 is equivalent to the size of long (4 bytes). The lowest byte of u1 may now be accessed by u1.c[0] as well as by u1.l

ANSI C –Data Types

- **Data encapsulation – union**
  - **Example *(contd)*:**

```
union
{
    unsigned char c[2];
    long l;
} u1;
```

byte number	accessed by	accessed by
0	u1.c[0]	u1.l (lowest byte)
1	u1.c[1]	u1.l
2		u1.l
3		u1.l (highest byte)

ANSI C –Data Types

- **Data encapsulation – union**
  - One common use of the union type in embedded systems is to create a scratch pad variable that can hold different types of data. This saves memory by reusing one 16 bit block in every function that requires a temporary variable

Example:

```
struct lohi_tag
{
short lowByte;
short hiByte;
};
union tagName
{
int asInt;
char asChar;
short asShort;
long asLong;
int near * asNPtr;
int far * asFPtr;
struct hilo_tag asWord;
} scratchPad; //scratchPad is the variable name
…
// accesing union elements
union tagName * scratchPad_ptr; //declare pointer type
scratchPadPtr = &scratchPad; //point to scratchPad
someInt = scratchPad_ptr->asInt; //retrieve as integer
scratchPad.asChar = 'b'; //assign b to scratchPad
tempChar = scratchPad.asChar; //retrieve as character
```

file01.c

ANSI C –Data Types

- **Data encapsulation – union**
  - Another common use for union is to facilitate access to data as different types. For example, the Microchip PIC16C74 has a 16 bit timer/counter register called TMR1 made up of two 8 bit registers called TMR1H (high byte) and TMR1L (low byte). It is possible that sometimes you would like to access the register as two 8 bit values or as one 16 bit value. A union will facilitate this type of data access:
  - **Example:**

```
struct asByte
{
   int TMR1H; //high byte
   int TMR1L; //low byte
};
union TIMER1_tag
{
   long TMR1_word; //access as 16 bit register
  struct asByte TMR1_byte;
} TMR1;
```

ANSI C –Data Types

Bit fields

- offer the possibility to access single bits or groups of bits in the not bit addressable memory.
- The order of the bits can be defined with the help of the *struct* keyword:

struct *[bitfield_name]*

{

data_type1 ivar_1: n_bit_1;

data_type2 ivar_2: n_bit_2;

...

} *[bitfield_list];*

ANSI C –Data Types

Bit fields

- **n_bit** is the size of the bit field element **ivar** in bits. Negative values are forbidden, values with more bits than that of the standard word width of the controller might lead to errors. A value of zero means that the current bitgroup fills up the remaining bits to the next word (=int) boundary.
- Bitfields are used especially in connection with control and status registers of the periphery of microcontrollers
- The ordering of the bits is compiler specific. This means that some compilers assign the LSBs to the first bits of the bit field definition, while others use the MSBs

ANSI C –Data Types

Bit fields – Example 1

```
struct TxIC
    {
            unsigned int glvl: 2;
            unsigned int ilvl: 4;
            unsigned int ie: 1;
            unsigned int ir: 1;
            unsigned int : 0;
    } t7ic;
t7ic.ilvl = 12;
```

- The compiler will assign the bits as given below:

bit-no.	15 - 8	7	6	5 - 2	1 - 0
bit name	?	ir	ie	ilvl	glvl
binary value				1100	

Why C?

ANSI C

Introduction

Identifiers

Data Types

Expressions

Program flow

Arrays & Pointers

Functions

Preprocessor

The Embedded Difference

Examples

Bit fields – Example 2

- for the Motorola MC68HC705C8 defines the Timer Control Register (TCR) bits as bit fields in the structure called TCR

```
struct reg_tag
{
  int ICIE : 1; // field ICIE 1 bit long
  int OCIE : 1; // field OCIE 1 bit long
  int notUsed : 3 = 0; //notUsed is 3 bits and set to 0
  int IEDG : 1; // field IEDG 1 bit long
  int OLVL : 1; // field OLVL 1 bit long
} TCR;
…
struct reg_tag * TCRFieldPtr;
TCRFieldPtr = &TCR;
TCR.ICIE = 1; // access using dot operator
TCRFieldPtr->ICIE = 1; // using right arrow operator
```

file01.c

ANSI C –Data Types

- **Enumerated data types**
  - The most straightforward complex data type is the enumerated data type, declared as type *enum*. The *enum* type is used to represent a set of possible values.
    - Syntax:

enum *[enum_name]*

{

value_1 [= ival]

[,value_2 [= ival]]

…

} *[enum_list];*

ival is the value that **value_x** shall be represented with. If not specified value_1 will be assigned 0, value_2 = 1, etc.

ANSI C –Data Types

- **Enumerated data types**
- Observations
  - Internally, an *enum* variable is treated as a signed integer
  - The compiler does not check the variable against the defined values of the enumeration value list
  - enum variables should only be used in assignment and comparison operations
  - Enumerated values can be used as if they were defined as a macro, what means that they are known at compile time and can thus be used in `switch-case` statements

ANSI C –Data Types

Enumerated data types - Examples:

- creates an enumerated type called WEEK, provides seven possible values, and declares a variable called dayOfWeek of this new enumerated type.

  enum WEEK { Su, Mo, Tu, We, Th, Fr, Sa } dayOfWeek;

- separate this process into two declarations

  enum WEEK { Mo, Tu, We, Th, Fr, Sa, Su };

  enum WEEK dayOfWeek; // WEEK is called a ***tag***

- The tag is useful as it can represent a list of enumerated elements to declare more than one variable of that type

  enum WEEK { Su, Mo, Tu, We, Th, Fr, Sa } dayOfWeek;

  enum WEEK dayOFWeek;

  enum WEEK payDay = Th;

  enum WEEK groceryDay = Sa;

ANSI C –Data Types

Definition of private types

- **struct**, **union**, and **enum** keywords allow both type declaration and data definition in one statement
- in order to separate declaration and definition, the typedef keyword can be used
- **syntax:**

  typedef *basic_type type_name;*

  where:
- **typedef** "C"-keyword for data structure declaration
- **basic_type** may be any type such as char, int, float, struct, union, enum, etc
- **type_name** is any name allowed

ANSI C –Data Types

Definition of private types

- Useful type definitions:
  - typedef unsigned char uint8;
  - typedef signed char int8;
  - typedef unsigned int uint16;
  - typedef signed int int16;
  - typedef unsigned long uint32;
  - typedef signed long int32;
- the above type definitions allow writing platform-independent code

ANSI C –Data Types

- **Type Conversion**
  - Implicit type conversion
  - Explicit type conversion

- **Implicit type conversion**
  - when operands of different types are combined in expressions
  - the conversions are implicitly performed according to standard rules

ANSI C –Data Types

- **Type Conversion**
  - **Implicit cast**
    - all char and short operands are converted to int
    - if one operand is unsigned, the other operand is converted to unsigned as well
    - all float operand are converted to double
    - if the operand of an expression are of different types, calculations always occur with the widest type. (Width of a data type simply means the number of bytes a value occupies.
    - the result of an expression is always adjusted to the type of variable the result has.

ANSI C –Data Types

- **Type Conversion**
  - **Implicit cast – Example:**

```
char v1;
int v2;
double v3;

v2 = v1+v3;    /*expression is double, result is int*/
v1 = v2 - 2*v1;/* expression is int. The result
         variable is of type char. The most
         significant part of result is lost */
```

ANSI C –Data Types

Type Conversion

Explicit cast

- to avoid hidden code which might lead to wrong results, every type conversion must be casted explicitly.
- explicit cast is expressed by writing the desired data type in parenthesis as an operator in front of them
- **Example:**

```
unsigned int i1;
char c1, c2;
c1 = (char)(i1 - (int)c2);
/* there is a value range reduction by casting the expression as char */
```

ANSI C – Assignment, Expression, Operators

- **Variable assignment**
  - Simple variable assignment

    variable = expression;

    variable - defined or declared memory location

    = assignment operator

    expression - constituted from one more operands and operator

    ; end of statement
  - Multiple variable assignment

    var_1 = var_2 = [=...] = expression;

    Assigns the same value to multiple variables

ANSI C – Assignment, Expression, Operators

- **Expressions**
- *[unary operator] operand [binary operator][operand][...];*

unary operator: concern a single operand only

+	positive sign
-	negative sign
++	increment
--	decrement
&	address of
*****	indirection
sizeof(name)	size of name in byte
(type cast)	explicit type casting
!	logical negation
~	bit by bit inversion

ANSI C – Assignment, Expression, Operators

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

- **Binary operators**
- performs a two operand operation

- **arithmetic operators**

+	sum
-	difference
*	multiplication
/	division
%	modulo operation

- **comparison operators**

<	less than
<=	less or equal
>	greater than
>=	greater or equal
==	equivalence
!=	not equal
&&	logical AND
||	logical OR

ANSI C – Assignment, Expression, Operators

- **Binary operators**
- **bit-by-bit operations**

&	AND
|	OR
^	XOR (exclusive OR)
<<	shift left
>>	shift right

- **compound assignment operators**

+= -= *= /= %=
<<= ^= >>= &= |=

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

ANSI C – Assignment, Expression, Operators

- **Binary operators. Bitwise (bit-by-bit) operations**

- **&** is often used to clear one or more bits of an integral variable to 0.

  PTH = PTH & 0xBD // clears bit 6 and bit 1 of PTH to 0

  // PTH is of type char

- **|** is often used to set one or more bits to **1**

  PTB = PTB | 0x40; // sets bit 6 to 1 (PTB is of type char)

- **^** can be used to toggle a bit.

  abc = abc ^ 0xF0; // toggles upper four bits (abc is of type char)

ANSI C – Assignment, Expression, Operators

- **Binary operators. Bitwise (bit-by-bit) operations**

- **>>** can be used to shift the involved operand to the right for the specified number of places. For unsigned numbers, the bit positions that have been vacated by the shift operation are zero-filled. For signed numbers, the sign bit is used to fill the vacated bit positions. In other words, if the number is positive, 0 is used, and if the number is negative, 1 is used

```
xyz = xyz >> 3;          -- shift right 3 places
```

- **<<** can be used to shift the involved operand to the left for the specified number of places. The bit positions that have been vacated by the shift operation are zero-filled.

```
xyz = xyz << 4;          -- shift left 4 places
```

- The assignment operator **=** is often combined with the operator. For example,

```
PTH     &= 0xBD;
PTB     |= 0x40;
xyz     >>= 3;
xyz     <<= 4;
```

ANSI C – Assignment, Expression, Operators

- **Binary operators. Bitwise (bit-by-bit) operations**
  - multiplication and divisions can be avoided for operands of value 2x, what results in a faster code.

- **Example:**

```
1. z = x * 2;    =>    z = x << 1;
2. z = x / 2;    =>    z = x >> 1;
```

- there is no overflow checking mechanism available!

ANSI C – Assignment, Expression, Operators

- **Comparison operators**

```
if (!(ATD0STAT0 & 0x80))
        statement1;     // if bit 7 is 0, then execute statement1
if (i > 0 && i < 10)
        statement2;     // if 0 < i < 10 then execute statement2
if (a1 == a2)

        statement3;     // if a1 == a2 then execute statement3
```

ANSI C – Assignment, Expression, Operators

- **Expressions**

- **ternary operator:** performs a three operand operation

  ? : **conditional**

- **operand:**
  - constant
  - variable
  - pointer
  - return value of a function

ANSI C – Assignment, Expression, Operators

Operator's Hierarchy

Category	Operator	Execution	Description
1.	() [] -> .	left → right	function call or term grouping array subscript indirect structure element selection direct structure element selection
2. unary operators	! ~ + - ++ -- & * sizeof (*type*)	right ← left	logical negation bit-by-bit inversion positive sign negative sign increment decrement address of indirection size in bytes explicit type casting
3. multiply / divide operators	* / %	→	multiplication division modulo operation for integer values
4. additive operators	+ -	→	addition subtraction
5. shift operators	>> <<	→	shift left shift right
6. relational operators	< <= > >=	→	less than less or equal greater than greater or equal

ANSI C – Assignment, Expression, Operators

Operator's Hierarchy

7. equivalence operators	== !=	→	equal not equal
8.	&	→	bitwise AND
9.	^	→	bitwise XOR
10.	|	→	bitwise OR
11.	&&	→	logical AND
12.	||	→	logical OR
13.	? :	←	conditional
14. assignment operators	= *= /= %= += -= &= ^= |= <<= >>=	←	assignment assign product assign quotient assign integer remainder assign sum assign difference assign AND-masked value assign XOR-masked value assign OR-masked value assign left shifted value assign right shifted value
15. comma	,	→	separator

ANSI C – Assignment, Expression, Operators

- **Operators**
- **Examples:**
  - Incrementing *before* the calculated value is used: x=++n;
  - Incrementing *after* the calculated value is used: x=n++;
  - ++n && ++i
  - If n evaluates to 0, ++i is not executed any more.
  - x = ++ (x+y); x = 10++ not allowed

ANSI C – Assignment, Expression, Operators

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

- **Operators**
- **Examples:**
  - `x = x | MASK;` all bits of x which are set in MASK are set
  - `n = ~ n;` negation of bits
  - `res = status & (~1);` clear the bit 0

ANSI C – Assignment, Expression, Operators

- **Operators**
  - Increment and decrement operators can be used in prefix or postfix notation. They must not be used for expressions.
  - An increment/decrement of pointers is type specific
- **Example:**

```
long *i_ptr = 0x100;
i_ptr++; // i_ptr points to 0x104
```

ANSI C – Assignment, Expression, Operators

- **Operators**
  - The order of unary operations is important as this may affect the result.
- **Example:**

```
unsigned char a = 0x7f;
unsigned int b1, b2;
b1 = (unsigned int)~a; // b1 = 0x0080;
b2 = ~(unsigned int)a; // b2 = 0xff80;
```

ANSI C – Assignment, Expression, Operators

- **Operators**
  - The modulo operation works for integer-by-integer divisions only. It returns the integer remainder.
  - Instead of the conditional operator, an if-…construct should be used,
- **Example:**

```
x = (a ? 1 : 0);
```

should be replaced by

```
if (a) x=1;
else x=0;
```

ANSI C – Assignment, Expression, Operators

- **Logical expressions**
  - It is rare for logical values to be stored in variables
  - They are usually generated as required by comparing two numeric values
  - they are evaluated from the left to the right
  - in order to reduce calculation time, put to the first place the most unlike condition in an **AND** connection (&&) and the most probable condition in an **OR** connection (||).

Program Flow Statements

- **Sequences**
  - Single Branching: The “if“-Statement
  - Double Branching: The “if-else“-Statement
  - Multiple Branching: The “switch - case“-Statement
  - Loop with Testing in the Beginning: The “while“-Statement
  - Indexed Loop with Testing in the Beginning: The “for“-Statement
  - Loop with Testing in the End: The “do - while“-Statement
  - Other Program Flow Statements: continue, break, goto

Program Flow Statements

- **Single Branching: The “if“-Statement**

  if (*expression*)

  sequence

  - Only if the boolean **expression** in parenthesis evaluates to logical TRUE, the **sequence** will be executed
  - To compare an *unsigned int* variable with unlike 0 the following syntax should be preferred :

    if (var != 0) or *if (var)* instead of if (var>0)

Program Flow Statements

- **Double Branching: The "if-else"-Statement**

  if (*expression*)

  sequence_1

  else

  sequence_2

  - the expression should be formulated so that the most probable result is TRUE. Then, most often the branch to **sequence_2** may not be taken, which results in a shorter program execution time.
  - encapsulation of several "if - else" statements can be avoided by using boolean algebra:

  if (expression_1 || (expression_2 && expression_3))

  sequence_1

Program Flow Statements

- **Multiple Branching: The "switch - case"-Statement**

```
switch ( expression )
{
    case const_expression_1: statement_1; break;
            ...
    case const_expression_n: statement_n; break;
    default: statement_n+1; break;
}
```

- **expression** will be evaluated. The result is of type **int** or **unsigned int**

Program Flow Statements

- **Multiple Branching: The "switch - case"-Statement**
  - **const_expression** must be a constant value which is known at compile time. If the **expression** in parenthesis matches the **const_expression**, the subsequent statements are evaluated up to the next break statement respectively the end of the block.
  - The order of the constant expressions should be chosen according to the plausibility of their occurrence for runtime reasons.

Program Flow Statements

- **Multiple Branching: The “switch - case“-Statement**
  - **break** “C“-keyword that causes leaving the actual block. Every **case** should have a **break** statement.
  - **default** is a predefined label for all cases that do not match any of the other constant expressions in the block. In ANSI-“C“ it may be missing, which however means a bad programming style.
  - put the cases in the order according to their probability because the compiler sometimes generates an assembly code which rather reflects an if … elseif … else structure, especially if only few cases are given

Program Flow Statements

- **The “while“-Statement**

  while (*expression*)

  sequence

- The condition given by **expression** is evaluated first. Only if the result is TRUE the sequence is executed. It will be repeated as long as the **expression** is TRUE.
- Potentially endless loops must be equipped with a software watchdog:

```
while (*int_ptr <= TopValue && special_exit == 0);
```

Program Flow Statements

- **The “for“-Statement**

  for (*[init_list] ; [expression]; [continue_list]*)

  sequence

  - **init_list** is a list of statements separated by commas which will be executed unconditionally in advance of the loop.
  - **expression** results in a boolean value TRUE or FALSE. As long as it is TRUE, the **sequence** is executed, followed by the statements of the **continue_list.**
  - **continue_list** is a list of statements separated by commas which are evaluated as long as the **expression** is TRUE.

Program Flow Statements

The "for"-Statement

- The loop control variable must only be changed in the **continue_list** but nowhere else.
- Likewise, the loop exit condition must only be checked in the **expression.**
- To compare a decreasing local counter with unlike 0 the following syntax should be used:

`for (i = 10; i > 0; i--)` instead of `for (i = 10; i != 0; i--)`

Program Flow Statements

- **The "do - while"-Statement**

```
do
    sequence
while ( expression );
```

- The **sequence** is executed at least once. After that the **expression** is evaluated, and the **sequence** is repeated as long as it's result is TRUE.

Program Flow Statements

- **Other Program Flow Statements**
  - goto
  - continue
  - break

  - *goto* **must** and the other two instructions **should** be avoided as they lead to an ill structured program flow
  - **goto** causes the program to continue at the label which is defined at any other place in the program by a subsequent colon.

Program Flow Statements

Other Program Flow Statements

- **continue** is used in loop constructs and means a shortcut to continue with testing the next loop condition. Therefore, the statements following the **continue** statement won't be executed.
- **break** statement causes the program to continue at the next label outside the current block. It should be used in connection with **switch - case** constructs but not otherwise.

Arrays, Pointers

- Arrays
- Pointers
  - Pointers scaling
  - Pointers and Function Arguments
  - Pointers and Arrays
  - Pointers vs.. Multi-dimensional Arrays
- Functions
  - Function Header
  - Function Body
  - Calling a function

Arrays, Pointers

- **Arrays**
  - Encapsulate multiple elements of a single data type with one variable name.
  - No checking mechanism for the boundaries
  - Index must be of an unsigned type
  - Data are contiguously allocated

Array Declaration:

<*type*> <*name*> [<*size*>];

Arrays, Pointers

- **Arrays**
  - **type** could be any type;
  - **size** is mandatory: it's the maximal number of elements in the array
  - The declaration is going to allocate enough bytes for the whole array
  - Array could be initialized when declaring :
    - <*type*> <*name*> [<*optional size*>] = { <*value list*> }
  - in this case, the **size** is optional (if the size is not given, the size of the array is set to the number of element in the declaration list)

Arrays, Pointers

- **Arrays**
- **Examples:**

```
int t [4] ;
int j [] = {1, 2, 3 } ;          /* size 3 */
int k [5] = {1,2};  /* other elements are
                initialized to 0 */
float t[2][3] = {{2.3, 4.6, 7.2}, {4.9, 5.1, 9.3}};
int l [2] = {1, 2, 3 } ;         /* Incorrect */
char vocals[5] = {'a', 'e', 'i', 'o', 'u' };
char string[] = "This is a string";
```

Arrays, Pointers

Arrays

- The access to an element of the array is done with the index between brackets

  Example: r = a [i] + b [j];

- The first element of the array is 0 (zero-based index)

		memory address
`array[0]`		x + (0 * size of one array element in bytes)
`array[1]`		x + (1 * size of one array element in bytes)
`array[2]`		x + (2 * size of one array element in bytes)
`array[3]`		x + (3 * size of one array element in bytes)

Arrays, Pointers

- **Pointers**
  - Are variables that contain the address of an object (variable or function).
  - Enable indirect access to an object.
  - Are defined with the dereference operator (*)
- **Definition:**

  *type * pointer_name;*

- **type** is the type of the object the pointer points to.
  - **pointer_name** is a free chosable name of the pointer.

Arrays, Pointers

- **Pointers**

unsigned u16Val;

unsigned *p = &u16Val;

….

p is 0x0186 (address of u16Val)

*p is 0x55AA; (value located at address 0x0186, the value of u16Val)

Arrays, Pointers

- **Pointers**
  - A pointer can take the value of another pointer, an address of an object with "address-of" operator **&**, or a pointer expression

```
char *p ; int *q;
char a; int b;
p = &a; /*correct*/
p = &b; /* incorrect */
q = &a; /* incorrect */
q= &b ; /* correct */
p = &q; /* address of the pointer */
```

Arrays, Pointers

- **Pointers**
  - Const keyword can be used in pointer declarations.
- **Example:**

```
int a;
int *p;
int *const ptr = &a;      // Constant pointer
*ptr = 1;                 // Legal
ptr =p;                   // Error
```

- **Pointers**
  - A pointer to a variable declared as const can only be assigned to a pointer that is also declared as const.
- **Example:**

```
int a; int *p; const int *q;
const int *ptr = &a;      // Pointer to constant data
*ptr = 1;                 // Error
ptr =p;                   // Error
ptr =q;                   // Legal
```

- **Pointers scaling**
  - The type of the object a pointer points to serves the compiler as a basis for relative address calculations using pointer addition, subtraction, increment, and decrement.

Arrays, Pointers

- **Pointers scaling**
- **Example:**

```
        long *sl_ptr = 0x4000;
        char *sc_ptr;
        sc_ptr++ = (char *)(sl_ptr + 2);
will result in
        tmp = sl_ptr   + 2*sizeof(long)
              = 0x4000 + 2*4
              = 0x4008                    => sc_ptr
        sc_ptr++ = 0x4008 + 1*sizeof(char)
              = 0x4008 + 1*1
              = 0x4009                    => sc_ptr
```

- **Pointers and Function Arguments**
  - Pointer arguments enable a function to access and change objects in the function that called it. For instance, a sorting routine might exchange two out-of-order elements with a function called swap:

    swap(&a, &b);

  - Since the operator & produces the address of a variable, &a is a pointer to a.
  - the parameters are declared to be pointers, and the operands are accessed indirectly through them.

Arrays, Pointers

- **Pointers and Function Arguments**

```
void swap(int *px, int *py)   /* interchange *px and *py */
{
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
```

Arrays, Pointers

- **Pointers and Arrays**
  - Any operation which can be achieved by array subscripting can also be done with pointers.
  - The pointer version will in general be faster
- The declaration

  int a[10]

  defines an array a of size 10, that is a block of 10 consecutive objects named a[0], a[1], ..., a[9].

Arrays, Pointers

Pointers and Arrays

- The notation **a[i]** refers to the **i**-th element of the array If pa is a pointer to an integer, declared as

  int *pa

  then the assignment

  pa = &a[0];

  sets pa to point to element zero of **a**:

  pa contains the address of **a[0]**.

  Now the assignment **x = *pa** will copy the contents of **a[0]** into **x**.

Arrays, Pointers

- **Pointers and Arrays**
  - If **pa** points to a particular element of an array, then by definition:
    - **pa+1** points to the next element, **pa-i** points **i** elements before **pa**, and
    - **pa+i** points **i** elements after.

Thus, if **pa** points to **a[0]**, **then** ***(pa+1)**

refers to the contents of **a[1]**, **pa+1** is the address of **a[i]**, and ***(pa+i)** is the contents of **a[i]**.

Arrays, Pointers

Pointers and Arrays

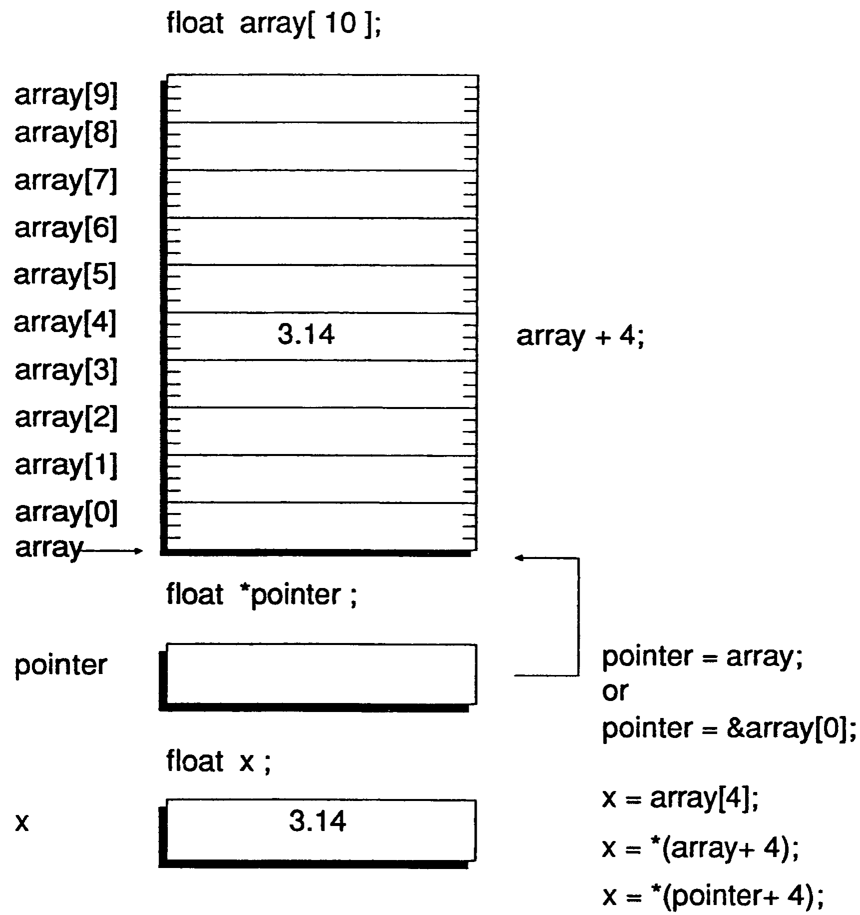
- By definition, the value of a variable or expression of type array is *the address of element zero of the array*. Thus after the assignment

  pa = &a[0]

  pa and **a** have identical values
- The assignment **pa=&a[0]** can also be written as **pa = a;**

Arrays, Pointers

Pointers and Arrays



Arrays, Pointers

Arrays of Pointers

- The most common use for an array of pointers is to use an array of pointers to type char which are pointed to strings. This technique can be used to send messages to a screen.

- the array is declared in main but the array is passed to a function where the values of the pointers are assigned

(about functions – a little bit later ☺)

```
void func1(char *p)
{
  p[0]="Press 1 to start";
  p[1]="Press 2 to continue";
  p[2]="Press 3 to RESET";
  p[3]="Press 4 to quit";
}
void main(void)
{
  int val;
  char *message[10];
  if (val==TRUE)
  {
    func1(message);
  }
 else
    message[0]="Status is OK";
}
```

file01.c

Arrays, Pointers

- **Pointers vs. Multi-dimensional Arrays**
  - Given the declarations

    int a[10] [20];

    int *b[10];
  - Array **a** is a true two-dimensional: 200 int-sized locations have been set aside, and the calculation **20xrow+col** is used to find the element **a[row,col]**
  - For **b**, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either syntactically or with code.

Arrays, Pointers

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

- **Pointers vs. Multi-dimensional Arrays**

```
char array [3][12] = {"The Course\n",
                      "is\n",
                      "boring\n"};
```

array[2]	b	o	r	i	n	g	\n	\0				
array[1]	i	s	\n	\0								
array[0]	T	h	e		C	o	u	r	s	e	\n	\0

Arrays, Pointers

- **Pointers vs. Multi-dimensional Arrays**

  char *array[] = {"The Course\n",
  "is\n",
  "boring\n"};

array[2]	b	o	r	i	n	g	\n	\0				
array[1]	i	s	\n	\0								
array[0]	T	h	e		C	o	u	r	s	e	\n	\0

Advantage of the pointer array is that the rows of the array may be of different lengths. That is, each element of **b** need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all

Arrays, Pointers

Pointer to Strings

- C does not know string, only arrays of characters.
- for the processing of strings, (searching, comparing, copying), standard routines exist.
- to be able to use subprograms, pointers to strings are used when passing parameters. The whole processing of strings relies on the use of pointers

Arrays, Pointers

- **Pointer to Strings**
  - the incrementing of a pointer on characters shifts the pointer to the next valid element of the string.

```
char *text, character;
text = “C - tutorial”;
character = *(text+4);
```

Arrays, Pointers

- **Pointers**
  - only a small selection of operations are defined for pointers:

Operation	Sign	Examples:
		`int *iptr, *jptr;`
		`unsigned int offset, tmp;`

Assignment	=	`iptr = 0x4000;` `jptr = iptr;`
Increment Decrement	++ --	`iptr++;` `--jptr;`
Comparison	== != <= >= < >	`if (iptr >= 0x2000);` `tmp = (jptr != iptr);`
Addition	+	`jptr = iptr + offset;`
Subtraction	-	`jptr = iptr - 0xf800u;`
Pointer Distance	-	`tmp = iptr - jptr;`

Functions

- **Functions**
- A function is defined by
  - a header (name, type, formal parameters)
  - a body
- A function has
  - a scalar type (char, int, long or pointer) : actual function
  - void type : procedure

- non void functions have a result
- void functions do not return any result

Functions

- **Functions**
- Function Header

Syntax:

<visibility<type><name>(<parameters list>)

Type

- void for procedure
- scalar type (default int)

Visibility

- *static* modifier used to hide the function

Functions

Functions - Function Header

Parameter types:

- *input* parameters (or value parameters): when calling the function, the caller copies a value in each input parameter. The const keyword could be used to declare those parameters as constant (optional)

- *input-output* parameters (or address parameters): when calling the function, the caller gives the address of a variable to the formal parameter. Those parameters are marked with the dereference, *, operator

Functions

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

Functions - Function Header

Examples of headers :

- a function f which has an input parameter x and return an integer value

```
int f (const int x);
int f (int x);
```

- a function f which has an input parameter x and an input-output parameter y and return an integer value.

```
int f (int x, int *y);
```

Functions

- **Functions - Function Body**
  - Delimited by **{** and **}**
  - Allows local declarations
    - Automatic data – located on the stack
    - Static data – located in global memory
  - If type is not void, the function includes a return statement which affects the result
  - the ***return*** exits the function
  - Input parameters are used as they have been declared
  - Output parameters are used using the * operator

Functions

- **Functions - Calling a function**
  - The call of a function is done using the name of the function, and with a list of actual parameters
  - parenthesis are mandatory, even if no parameter is required
  - **input** parameters accept expression of same type
  - **input-output** parameters require the address of a variable. The address of a variable is given by the operator &

Functions

- **Functions – Example:** Function with input parameter :

```
int f (int x)
{
    return x+1 ;
}
```

Calling:

```
int a;
const b = 14 ;
int r;

a=3;
r = f (a) ;          /* r is set to 4 */
r = f(b+r-1)         /* r is set to 18 */
```

Functions

- **Functions – Example:** Function with input-output parameter:

```
void g (int *y, int x)
{
   *y = x+1 ;
}
```

Calling:

```
int a;  const b = 14 ;  int r;


a=3;
g(&r,a) ;            /* r is set to 4 */
g(&r,b+r-1)          /* r is set to 18*/
g(r,a) ;             /* incorrect, protection fault */
```

- **Functions – Example:** Function bad example:

Function with input parameter as an output parameter

```
void g (int y, const int x)
{
   y = x+1 ;
}
```

Calling:

```
int a;  const b = 14 ;  int r;
r=0;
a=3;
g(r,a) ;       /* r is not affected */
g(&r,a) ;      /* compilation error */
```

Pointer to Functions

- The address of a function can be assigned to a pointer variable
- a function can also be passed as a parameter
- the call happens with the aid of the dereferencing operator "*"

Functions

Pointer to Functions

```
type (*name)();

void sort(char *v[], int n, int (*comp)())
{
        …
        int comp_res;
        comp_res = (*comp)();
        ...
}
```

```
int main()
{
    extern int strcmp(), numcmp();
    …
    if(numerical)
            sort(lineptr, nlines, numcmp);
    else
            sort(lineptr, nlines, strcmp);
}
```

Functions

Creating Header File

- To reuse functions that we have created, we can collect the set of functions that are of the same nature into one file and also put the **prototype declarations** of these functions into one header file.

- When reusing these functions, we need to add the C file that contains them into the project and also add the header file into the file that invoke these functions.

Preprocessor directives

- Macro definition
- File inclusion
- Conditional compilation
- Memory model

Preprocessor directives

- preprocessing is scheduled before compilation process
- It mainly consist in a text analyzing and processing tool. The produced file is used as input for compilation process
- the goal: flexible software for different application parameters and different software development tools
- preprocessor directives always start with # character in the first column line

Preprocessor directives

#define

introduces the definition of a preprocessor macro

#undef

clears a preprocessor macro definition

#include

includes the source text of another file

#if

evaluates an expression for conditional compilation.

#ifdef

checks for conditional compilation whether a macro is defined

Preprocessor directives

#ifndef: checks for conditional compilation whether a macro is not defined

#elif: introduces an alternative #if branch following a not compiled #if, #ifdef, #ifndef or #elif branch

#else: introduces an alternative branch following a not compiled #if, #ifdef, #ifndef or #elif branch

#endif: completes a conditional compilation branch

#line: indicates the line number and, optionally, a file name which is used in error logging files to identify the error position.

#error: eports an error which is determined by the user

Preprocessor directives

#pragma

Inserts a compiler control command.

Options for the compilation can be given just as in the command line.

A **#pragma** directive not known to the compiler is ignored and leads to a portable code.

Preprocessor directives

Macro definitions

#define *Macro_Name [[(parameters)] replace_text]*

#define: preprocessor directive

Macro_Name: is the name of the macro. Will be substituted by replace_text.

parameters: used literally in the replace_text and replaced when the macro is expanded

replace_text: the given text will replace the macro call literally

Preprocessor directives

It is forbidden to define macro with side effect (eg. increment) - disturbs the error checks consistency.

String operator: #

- is used in the replace_text to induce that the subsequent string is interpreted as the name of parameter. This name is replaced at the time the macro is expanded.

Token-pasting operator: ##

- precedes or follows a formal parameter. When it is expanded, the parameter is concatenated with the other text of the token.

Preprocessor directives

Files inclusion

#include <*file_name*>

#include ”*file_name*”

Additional information can be used in a source file (data types, function prototypes, ...) => header files.

“…” - source file path is the first path were the include file is searched. If it can not be found there, the project specific include path will be inspected.

<...> - omits the current source path and start the search of the include search path of the project.

Preprocessor directives

Files inclusion

- Some problem might arise if a header file defines some symbols and is included in multiple modules. The content of a header file must be delimited by a preprocessor switch
- Solution:

```
#ifndef __HEADER_FILE_H__
#define __HEADER_FILE_H__
……..
header-file content
……..
#endif
```

Miscellaneous Items

Input and Output — Examples

- Not part of the C language itself.
- Four I/O functions will be discussed.

1. int *getchar* ( ).
   -- returns a character when called

   Example:
   ```
   char        xch;
   xch = getchar ();
   ```

2. int *putchar* (int).
   -- outputs a character on a standard output device

   Example:
   ```
   putchar('a');
   ```

3. int *puts* (const char *s).
   -- outputs the string pointed by **s** on a standard output device

   Example:
   ```
   puts ("Hello everyone :) ! \n");
   ```

4. int *printf* (*formatting string, arg1, arg2, …*).
   -- converts, formats, and prints its arguments on the standard output device.

Formatting String for Printf

- The arguments of ***printf*** can be written as constants, single variable or array names, or more complex expressions.
- The formatting string is composed of individual groups of characters, with one character group associated with each output data item.
- The character group starts with %.
- The simplest form of a character group consists of the percent sign followed by a *conversion character* indicating the type of the corresponding data item.
- Multiple character groups can be contiguous, or they can be separated by other characters, including whitespace characters. These "*other characters*" are simply sent to the output device for display.

- **Examples:**

```
printf ("this is a challenging course ! \n");
printf(%d %d %d", x1, x2, x3); // outputs x1, x2, and x3 using minimal number
                               // of digits with one space separating each value
printf("Today's temperature is %4.1d\n", temp);
```

Miscellaneous Items

Using the C Compilers

- The special edition C compiler from **CodeWarrior** has a size limit of 32 kB.
- The demo version of **ImageCraft C** compiler has a size limit of 8 kB.
- The **GNU C compiler** does not have size limit but the **EGNU IDE** cannot program flash memory which limits it to download program into the on-chip SRAM only.

Accessing Peripheral Registers

- Depending on the size of a peripheral register, it can be declared in one of the following methods:

```
#define  reg_name  *(volatile unsigned char *) reg_addr  // 8-bit wide register
#define  reg_name  *(volatile unsigned int *) reg_addr   // 16-bit wide register
```

- The HCS12 allows all of the peripheral registers to be relocated as a block. For this reason, the ImageCraft uses the following method to define peripheral registers:

```
#define  reg_base    0x0000                                   // base address
#define  reg_name1   *(volatile unsigned char *) (reg_base + offset1)
#define  reg_name2   *(volatile unsigned int *) (reg_base + offset2)
```

Miscellaneous Items

- The header file **hcs12.h** uses the following format:

```
#define        IOREGS_BASE   0x0000
#define    _IO8(off)    *(unsigned char  volatile *)(IOREGS_BASE + off)
#define    _IO16(off)  *(unsigned short volatile *)(IOREGS_BASE + off)
#define    PORTA        _IO8(0x00)     // port A data register
#define          PTA    _IO8(0x00)     // alternate name for PORTA
#define    ATD0DR0   _IO16(0x90)     // ADC result 0 register (a 16-bit register)
```

Peripheral Register Bit Definitions

- A value is assigned to each bit in the peripheral register that represents its weight.
- For example, the ADPU bit of the ATD0CTL2 register is defined to be 0x80 that is the positional weight (bit 7) of this bit. It is defined as follows:

```
ADPU   equ     $80
```

This bit can be set using the following statement:

```
ATD0CTL2    |= ADPU;
```

Miscellaneous Items

C Programming Style

- Programming style refers to **a set of rules and guidelines** used when writing a program.
- The essence of good programming style is **communication.**
- The objective of a good programming style is to make the program **easy to understand** and **debug**.
- A programmer uses three primary tools to communicate their intentions: comment, **naming of** variables, constants, and functions, and **program appearance** (spacing, alignment, and indentation).

General Guidelines to Comments

- The programmer should explain what every function does, what every variable does, and every tricky expression or section of code.
- The comment should be added before the programmer defines a variable or writing a function or section of code.
- Avoid obscure programming language constructs and reliance on language-specific precedence rules.

Miscellaneous Items

Program Documentation

All programs should include, at the beginning of the program, a comment block that

Includes at least:

- The programmer's name
- The date of creation
- The operating system and IDE for which the program was written
- Hardware environment (circuit connection) to run the program
- Program structure (organization)
- Algorithm and data structures of the program
- How to run the program

An Example is in the next page:

Miscellaneous Items

Why C?
ANSI C
Introduction
Identifiers
Data Types
Expressions
Program flow
Arrays & Pointers
Functions
Preprocessor
The Embedded Difference
Examples

```
// ------------------------------------------------------------------------------------------------
// Program: RtiMultiplex7segs
// Author: Some Name
// Build Environment: CodeWarrior IDE under Windows XP
// Date: 07/09/2008
// Hardware connection: short description
// Operation: This program shifts seven-segment patterns of 4 consecutive BCD digits
//     by using time-multiplexing technique with each pattern lasting for 0.5 s. The
//     time-multiplexing operation is controlled by the real-time interrupt. The patterns are
//           1 2 3 4
//           2 3 4 5
//           3 4 5 6
//           4 5 6 7
//           5 6 7 8
//           6 7 8 9
//           7 8 9 0
//           8 9 0 1
//           9 0 1 2
//           0 9 1 2
// ------------------------------------------------------------------------------------------------
```

Miscellaneous Items

Function Documentation

- Every function should be preceded by a comment describing the purpose and use of that Function. An example is as follows:

//---
// **Function:** FindSquareRoot
// **Purpose:** Computes and returns the square root of a 32-bit unsigned integer
// **Parameter:** A 32-bit unsigned integer of which the square root is to be found
// **Called by:** PrimeTest()
// **Returned value:** the square root of a 32-bit unsigned integer
// **Side effects:** none
//---

Code Appearance

Program readability can be improved by

- Proper spacing
- Proper indentation
- Vertical alignment

Miscellaneous Items

Proper Spacing

for (i=0;i<15;i++) & for (i = 0; i < 15; i++)

easier to read

Proper Indentation

- The following code is hard to read without proper indentation:

```
while(TRUE) {
for (i = 0; i < 10; i++) { // pattern array start index
for (j = 0; j < 100; j++) { // repeat loop for each pattern sequence
for (k = 0; k < 6; k++) {  // select the display # to be lighted
PTB        = SegPat[i+k];    // output segment pattern
PTP        = digit[k];              // output digit select value
delayby1ms(1);       // display one digit for 1 ms
}
}
}
}
```

Miscellaneous Items

Proper Spacing, Proper indentation

With proper indentation, the code section becomes easier to read:

```
while(TRUE) {
    for (i = 0; i < 10; i++) {              // pattern array start index
        for (j = 0; j < 100; j++) {         // repeat loop for each pattern sequence
            for (k = 0; k < 6; k++) {       // select the display # to be lighted
                    PTB    = SegPat[i+k];         // output segment pattern
                    PTP    = digit[k];            // output digit select value
                delayby1ms(1);              // display one digit for 1 ms
            }
        }
    }
}
```

Miscellaneous Items

Vertical Alignment

- It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious.
- Compare the next two sections of code. It is obvious that the second one is easier to Identify errors:

```
// values to generate 1 cycle of sine wave
unsigned rom char upper[60] = {0x38,0x38,0x39,0x3A,0x3B,0x3C,
   0x3C,0x3D,0x3D,0x3E,0x3E,0x3F,0x3F,0x3F,0x3F,
0x3F,0x3F,0x3F,0x3F,0x3F,0x3E,0x3E,0x3D,0x3D,
0x3C,0x3C,0x3B,0x3A,0x39,0x38,0x38,0x37,0x36,0x35,0x34,0x34,
   0x33,0x32,0x32,0x31,0x38,0x37,0x36,0x35,0x34,0x34,
0x30,0x30,0x30,0x30,0x30,0x31,0x31,0x32,0x32,0x33,
      0x34,0x34,0x35,0x36,0x37};
```

(a) non-vertically aligned array

```
// values to generate 1 cycle of sine wave
unsigned rom char upper[60] = {
        0x38,0x38,0x39,0x3A,0x3B,0x3C,0x3C,0x3D,0x3D,0x3E,
        0x3E,0x3F,0x3F,0x3F,0x3F,0x3F,0x3F,0x3F,0x3F,0x3F,
        0x3E,0x3E,0x3D,0x3D,0x3C,0x3C,0x3B,0x3A,0x39,0x38,
        0x38,0x37,0x36,0x35,0x34,0x34,0x33,0x32,0x32,0x31,
        0x31,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x30,
        0x31,0x31,0x32,0x32,0x33,0x34,0x34,0x35,0x36,0x37};
```

(b) vectically aligned array

Figure 5.39 Influence of vertical alignment of array on readability

Miscellaneous Items

Naming of Variables, Constants, and Functions

- The names of variables, constants, and functions should spell out their meaning or purpose.
- A variable name may have one word or multiple words. Use lowercase when the name contains only one word. For example, **sum**, **limit**, and **average**.
- A multiple-word variable name should be in mixed case starting with lowercase. For example, **inBuf**, **outBuf**, **squareRoot**, and **arrayMax**.
- Function names should follow the same principle. A few examples follow: **sevenSegShift()**, **putcSPI()**, **putsSPI()**, **openLCD()**, and **putsLCD()**
- Named constants should be all uppercase using underscore (_) to separate words. **SONG_TOTAL_NOTES**, **HI_DELAY_COUNT**, **LO_DELAY_COUNT**

The Embedded Difference

- Most embedded systems programs include a header file which describes the target processor
- The header files contain descriptions of reset vectors, ROM and RAM size and location, register names and locations, port names and locations, register bit definitions and macro definitions
- Most compiler companies will provide header files for devices supported by their compilers

- **Example: for MC9S12DJ256**

  #include "mc9s12dj256.h"

The Embedded Difference

- Another important aspect of device knowledge is the limits of the device for which the program is written.
- A certain device may have very limited memory resources and great care must be taken in developing programs which use memory frugally
- Along with issues of size comes issues of speed. Different devices run at different speeds and use different techniques to synchronize with peripherals
- It is essential that you understand device timing for any embedded systems application

The Embedded Difference

- Embedded systems developers require direct access to registers such as the accumulator
- The regular use of the infinite loop while(1). Embedded developers often use program control statements which are avoided by other programmers
- **Example:**

```
while (1)
{
   if (PortA.1 == PUSHED)
   {
          PortA.0 = ON;
          wait(10);  // wait ten seconds
          PortA.0 = OFF;
   }
}
```

The Embedded Difference

- Techniques used in an embedded system program are often based upon knowledge of specific device or peripheral operation
- **Example:**
  - When a button is pressed it "bounces" which means that it is read as several pushes instead of just one
  - It is necessary to include debouncer support in order to ensure that a real push has occurred and not a bounce
  - **Solution: a** wait() function which creates a delay before the button is checked again

The Embedded Difference

- Many developers prefer to write some code segments in assembly language for reasons of code efficiency or while converting a program from assembly language to C
- The following two definitions of the wait() function show the function written in C and the equivalent function in Motorola 68HC705C8 assembly language
- **#asm vs. #endasm**

The Embedded Difference

Why C?

ANSI C

The Embedded Difference

Examples

- **Example:**

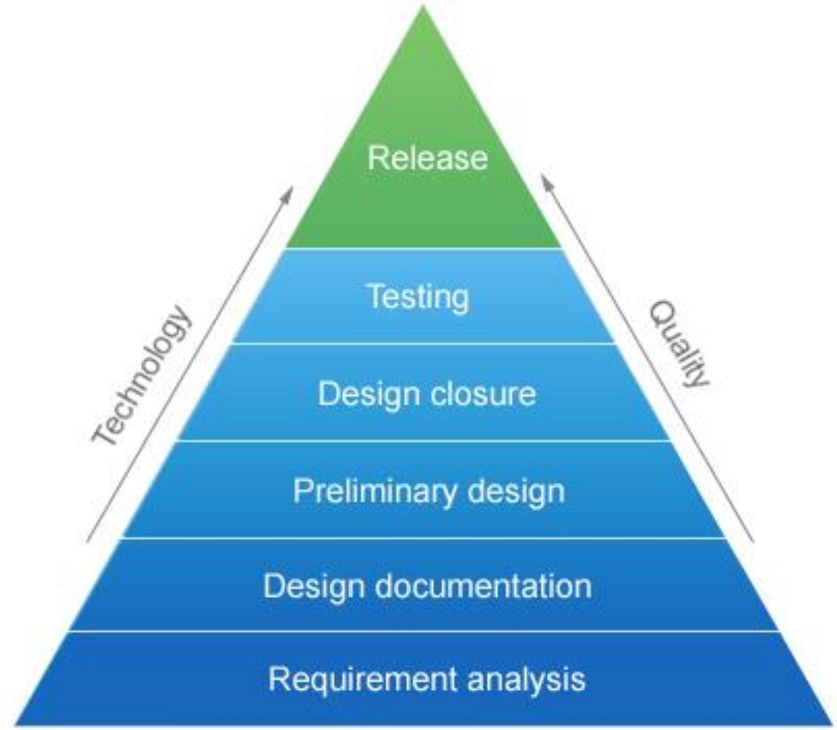
MC68HC705C8

```
void wait(registera delay)
{
        while (--delay);
}

//function with inline assembly
void wait(registera)
{
        char temp, time;
        // ocap_low and Ocap_hi are the output compare register
        //this register is compared with the counter and the ocf
        //bit is set (bit 6 of tim_stat)
        #asm
        STA time ;store A to time
        LDA #$A0 ;load A with A0
        ADD ocap_low ;add ocap_low and A
        STA temp ;store A to temp
        LDA #$25 ;load A with 25
        ADC ocap_hi ;carry + ocap_hi + accumulator
        STA ocap_hi ;store A to ocap_hi
        LDA temp ;load temp to accumulator
        STA ocap_low ;store a to ocap_lo
        LOOP BRCLR 6,tim_stat,LOOP ;branch if OCF is clear
        LDA ocap_low ;load ocap_lo to A
        DEC time ;subtact 1 from time
        BNE LOOP ;branch if Z is clear
        #endasm
}
```

The Embedded Difference

- **Embedded SW Development**
  - 1. Problem specification
  - 2. Tool/chip selection
  - 3. Software plan
  - 4. Device plan
  - 5. Code/debug
  - 6. Test
  - 7. Integrate



- Biggest mistake: “temptation of the keyboard”
- Use of development processes

The Embedded Difference

- **Embedded SW Development**
  - 1. Problem specification
- The problem specification is a statement of the problem that your program will solve without considering any possible solutions. The main consideration is explaining in detail what the program will do.
- Once the specification of the problem is complete you must examine the system as a whole. At this point you will consider specific needs such as those of interrupt driven or timing-critical systems.
- **Example:** If the problem is to design a home thermostat the problem specification should examine the functions needed for the thermostat. These function may include reading the temperature, displaying the temperature, turning on the heater, turning on the air conditioner, reading the time, and displaying the time. Based on these functions it is apparent that the thermostat will require hardware to sense temperature, a keypad, and a display.

The Embedded Difference

- **Embedded SW Development**
  - 2. Tool/chip selection
- Needs based on memory size, speed and special feature availability will determine which device will be most appropriate. Issues such as cost and availability should also be investigated.
- It is essential to determine if the development decisions you have made are possible with the device you are considering.
- **Example:** using C => compiler for C
- It is also useful to investigate the availability of emulators, simulators and debuggers
  - 3. Software plan
- Select an algorithm which solves the problem specified in your problem specification
- The overall problem should be broken down into smaller problems
- **Example:** The home thermostat project quite naturally breaks down into modules for each device and then each function of that device

The Embedded Difference

Embedded SW Development

- 4. Device plan – hardware specific routines
  - **1) Set up the reset vector**
  - **2) Set up the interrupt vectors**
  - **3) Watch the stack (hardware or software)**
  - **4) Interact with peripherals such as timers, serial ports, and A/D converters**
  - **5) Work with I/O ports**
- 5. Code/debug
  - Syntactic correctness of the code
  - Timing of the program
- 6, 7. Test, Integrate
  - Test each component and integrate in a functional product (project)

References

- C Programming Language (2nd Edition)
  - Brian W. Kernighan and Dennis M. Ritchie; Prentice Hall
- First Steps with Embedded Systems
  - Byte Craft Limited
- C Programming for Embedded Systems
  - Kirk Zurell; R&D Books