C. Programming in C

In this chapter we will do:

- C Programming Basics
- Linking C and Assembly
- Graphical notations for algorithms
- Programming Practice

C vs. Assembly

Advantages of C (and other high-level languages):
1. Machine independent
   → programs can be ported to a different computer system
2. Easier to program and debug
   → cheaper software development
   (one C line of code is translated to several assembly lines)

Compiler for C and C++

- Translation of C/C++ source code to machine language
- **Machine-independent**
- Use of variable and function names
- **Cross compiler** if Hardware1 ≠ Hardware2

C.1 C Programming Basics

- Program structure
- Variables
- Assignments and expressions
- Control structures
- Functions
- Arrays
Program Structure

- Comments are enclosed in /* and */
- "Include" required for libraries used
- Each program starts execution with "main"
- Statements follow enclosed by "{" and "}"; separated by semicolons ";"
- Return value or parameters of "main" makes sense for PC, but not in embedded system

```c
/* Demo program */
#include "eyebot.h"
int main()
{ ...
   return 0;
}
```

Variables

- Variables contain data
- Variables must be declared before they can be used
- Variable (and other) names are case sensitive: i ≠ I
- Simple data types are:
  - int (integer)
  - float (floating point) ← Try to minimize in embedded sys.
  - char (character)
- Variables can be
  - local (declaration inside function or main)
  - global (declaration outside) ← Try to avoid!

```c
#include "eyebot.h"

int distance; ← global variable

int main()
{ char direction; ← local variable
distance = 100;
direction = 'S';
printf("Go %c for %d steps\n", direction, distance);
return 0; ↑ function call (system function)
}
```

Notes

How to implement “printf” on an embedded system?
Notes

• Not only correct program **errors**, also correct all **warnings**!
• Frequent warnings are related to data types:
  – Data types can be converted by using “type casts”,
    i.e. placing the desired type name in parenthesis before the expression
    \[
    \text{distance} = \text{(int)} \ \text{direction};
    \]
  – For int ↔ char conversions, the ASCII char code is used:
    \[
    \begin{align*}
    \text{(int)} \ 'A' & \rightarrow 65 \\
    \text{(int)} \ 'B' & \rightarrow 66 \\
    \cdots \ \\
    \text{(char)} \ 70 & \rightarrow 'F'
    \end{align*}
    \]

Control Structures

**Selection (if-then-else):** next step depends on condition
\[
\text{if (distance == 1)} \quad \{ \text{condition satisfied} \}
\]
\[
\text{else direction} = 'N'; \quad \{ \text{other case} \}
\]

\[\uparrow \text{“else-part” is optional}\]

**Comparisons can be:**
\[-, >, <=, >=, ==, !=\]

**Logic expressions** \(a\) and \(b\) can be combined as:
\[
\begin{align*}
\text{a} \ \&\ \text{b} & \rightarrow \text{a and b} \\
\text{a} \ |\ | \text{b} & \rightarrow \text{a or b} \\
\sim\text{a} & \rightarrow \sim\text{a}
\end{align*}
\]

**Bitwise** logic expressions of integer operands returns an **integer**
\[
\begin{align*}
x \ \&\ y & \rightarrow \text{x and y} \\
x \ |\ | y & \rightarrow \text{x or y} \\
x \ ^\wedge y & \rightarrow \text{x xor y} \\
\sim x & \rightarrow \sim x
\end{align*}
\]

**Multiple Selection (switch-case):** instead of multiple if-selections
\[
\text{switch (distance)} \quad \{ \text{case} \}
\]
\[
\text{case 1: direction} = 'S'; \\
\text{distance} = \text{distance} - 10; \\
\text{break;}
\]
\[
\text{case 7: direction} = 'N'; \\
\text{break;}
\]
\[
\text{case 7: direction} = 'W'; \\
\text{break;}
\]
\[
\text{default: direction} = 'W'; \\
\}
\]

\[\uparrow \text{“break” signals end of case}\]
\[\text{optional default case}\]
Control Structures

Iteration (for): execute a statement sequence a fixed number of times

```c
int i;
...
for (i=0; i<4; i++)
{ LCDPrintf("%d\n", i);
  i++;
}
```

Output:
0
1
2
3

Control Structures

Iteration (while): execute a statement sequence a variable number of times

```c
int i;
...
i=0;
while (i<4)
{ LCDPrintf("%d\n", i);
  i++;
}
```

Output:
0
1
2
3

Control Structures

Iteration (do-while): execute a statement sequence a variable number of times

```c
int i;
...
i=0;
do
{ LCDPrintf("%d\n", i);
  i++;
  }
while (i<4)
```

Output:
0
1
2
3

Points

- Pointers are a data structure, as as int, float, char.
- Pointers represent addresses in memory space rather than values.
- Pointers are sometimes difficult to work with and error prone.
- Pointers should be avoided if not required by novice programmers.

<table>
<thead>
<tr>
<th>Memory</th>
<th>C Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Data</td>
</tr>
<tr>
<td>$00000007</td>
<td>int x, y; /* integer */</td>
</tr>
<tr>
<td>$00000005</td>
<td>x = 7; y=5;</td>
</tr>
<tr>
<td>$00000006</td>
<td>int <em>a; /</em> pointer */</td>
</tr>
<tr>
<td>$00000004</td>
<td>a = &amp;y; /* address of y */</td>
</tr>
<tr>
<td>$00000003</td>
<td>a = $00000004 (address)</td>
</tr>
<tr>
<td>$00000002</td>
<td>*a = 5 (dereferenced value)</td>
</tr>
</tbody>
</table>
Functions

- Functions are sub-programs
  - take a number of parameters (optional)
  - return a result (optional)
- “main” has the same structure as a function

```c
int main (void) // returns int, no parameters
{ ...
}
```

Parameter values are copied to function

Input parameters are simple

Function declaration:

```c
int sum (int a, int b)
{ return a+b;
}
```

Function call:

```c
int x,y,z;
x = sum(y, z); // can use constants or variables
x = sum(y, 10); // y and z remain unchanged
```

Input/output parameters require pointer

Function declaration:

```c
void increment (int *a)
{ *a = *a + 1;
} // use "*" as reference to access parameter
```

Function call:

```c
int x;
increment(&x); // use "&" as address for variable "x" will get changed
increment(&10); // error !
```

Value and Pointer Parameters

```c
void exchange(int a, int b)
{ int temp;
temp = a;
a = b;
b = temp;
printf("ex: a=%d b=%d\n", a,b);
}
```

```c
int main()
{ int a, b;
a = 5;
b = 7;
printf("main: a=%d b=%d\n", a,b);
exchange(a, b);
printf("end: a=%d b=%d\n", a,b);
getchar();
return(0);
}
```

```c
void exchange (int *a, int *b)
{ int temp;
temp = *a;
*a = *b;
*b = temp;
printf("ex: a=%d b=%d\n", *a,*b);
}
```

```c
int main()
{ int a, b;
a = 5;
b = 7;
printf("main: a=%d b=%d\n", a,b);
exchange(&a, &b);
printf("end: a=%d b=%d\n", a,b);
getchar();
return(0);
}
Arrays

- Data structure with multiple elements of the same type
- Array elements are accessed via an index in square brackets
- Declaration:
  ```
  int field[100]; ← elements field[0] .. field[99]
  char text[20]; ← elements text[0] .. text[19]
  ```
- Use:
  ```
  field[0] = 7;
  text[3] = 'T';
  for (i=0; i<100; i++)
    field[i] = 2*i-1;
  ```
- Note: If arrays are used as parameters, their address is used implicitly - not their contents

Strings are actually character arrays
- Each string must be terminated by a “Null character”: (char) 0
- C provides a library with a number of string manipulation functions
- Declaration:
  ```
  char string[100]; ← 99 chars + Null character
  ```
- Use:
  ```
  string[0] = 'H';
  string[1] = 'i';
  string[2] = (char) 0;
  LCDPrintf("%s\n", string);
  ```
  Output: Hi

C.2 C++ Outlook

- C++ is a super-set of C
- C++ is object-oriented, therefore requires a completely different style to C
- C++ uses class concept
- Not all embedded systems can be programmed in C++
- Embedded systems often have real-time requirements. (This excludes most Java implementations)

Many C compilers are actually C++ compilers, e.g. Visual C++, gnu gcc, MinGW, devcpp, … therefore, some basic C++ extensions can be used in C:
- Comments:
  ```
  // comment until end of line
  ```
- Declare loop variables within for:
  ```
  for (int i=0; i<n; i++) …
  ```
- Call by reference instead of pointers:
  ```
  void swap(int &a, int &b)
  { int m = a; a=b; b=a; }
  ```
C.3 Linking C with Assembly

- Hand-coded assembly code is often faster than compiler-generated assembly code (e.g. from C source)
- Therefore: time-critical passages in embedded systems are often programmed in assembly language, while the main program is written in C
- So: How do I link C and assembly?
  - C program calling assembly function ⇐
  - Assembly program calling C function

Linking C with Assembly

- Call from C like any other C function
- What needs to be done in assembly?
  - Save/restore used registers
  - Get parameters from registers or stack
  - Perform actual computation
  - Place return value in register register(s)

AVR Data Type Lengths (gcc)

- char 8 bits
- int 16 bits
- long 32 bits
- long long 64 bits
- float 32 bits
- double 32 bits (so not really double)
- pointer 16 bits
- function ptr. 32 bits
AVR Function Return Value (gcc)

C conventions for subroutines return values:
- 8-bit values → extended to 16 bits are returned in R24, must clear R25
- 16-bit values are returned in R24:R25
- 32-bit values are returned in R22-R25
- 64-bit values are returned in R18-R25

```c
int example() { ... }
```

AVR Function Parameter Passing (gcc)

C conventions for calling subroutines:
- Parameters are passed from left to right
- Parameters are passed in registers R25/24, R23/22, ... R9/R8
- All parameters start in odd register number (high byte)
  - so all parameters occupy at least 16 bits
  - lower byte is in even register (e.g. R24), so odd register =0 for 8-bit param.
- When all registers (R25..8) are used, additional parameters are passed on the stack
  - only happens for functions with many parameters
  - or for functions with variable parameter number (e.g. printf)

```c
void example(int a, char b) { ... }
```

AVR Register Usage (gcc)

- R0 temporary register, only saved for interrupts
- R1 must always contain 0x00
- All other registers that are used inside an assembly subroutine should be saved and restored on the stack.

AVR Example: C + Assembly 8-bit

```c
file c.c
main()
{ char a,b,c;
  a=7; b=5;
  c = add(a,b);
  ...
}
```

```asm
file asm.asm
add: ; a is already in R24
ADD R24,R22 ; add b
CLR R25 ; 16 bit res.
RET
```

Note: Result will overwrite rightmost parameter!
AVR Example: C + Assembly 16-bit

file c.c
main()
{ int a,b,c;
a = 7; b = 5;
c = add(a,b);
...
}

file asm.asm
add: ; a is already in R25:R24
ADD R24, R22 ; add low-b
ADC R25, R23 ; add hi-b
RET

equivalent function in C:
int add(int x, int y)
{
    return x + y;
}

file c.c
main()
{ char a,b,c;
...
    c = add(&a, &b);
}

file asm.asm
add: MOVW X, R24   ; move addr(a)
LD R24, X   ; load val(a)
MOVW X, R22   ; move addr(b)
LD R25, X   ; load val(b)
ADD R24, R25 ; add values
CLR R25     ; 16bit ret.
RET

equivalent function in C:
char add(char *x, char *y)
{
    return (*x) + (*y);
}

R23:R22
R25:R24
R22:R24

AVR Example: Pointer Types

file c.c
main()
{ char a, b, c;
...
}

file asm.asm
add: MOVW X, R24   ; move addr(a)
LD R24, X   ; load val(a)
MOVW X, R22   ; move addr(b)
LD R25, X   ; load val(b)
ADD R24, R25 ; add values
CLR R25     ; 16bit ret.
RET

equivalent function in C:
char add(char *x, char *y)
{
    return (*x) + (*y);
}

R25:R24
R25:R24
R23:R24
R22:R24

Pointer types in C are addresses in assembly language

C.4 Graphical Program Notations

Flow Chart
- Graphical representation of program flow
- Special symbols for
  - Start
  - Stop
  - Computing
  - I/O
  - Subroutine
  - Branching
- Link components by arrows

Flow Chart for Fibonacci

START

STOP

print f_i
print f_i

Init registers with f_0, f_1

Computation

Subroutine

Input/ Output

Question?

f_i := f_i+1 + f_i+2

f_i > 100

Yes

STOP

No

f_i := f_i+1
f_i+2 := f_i

Definition:
Fibonacci numbers
f_0 = 0
f_1 = 1
f_i = f_i-1 + f_i-2
→ 0, 1, 1, 2, 3, 5, 8, ...
Structogram

- Graphical representation of program flow
- Higher level concepts, e.g. for C, Java, Modula
- Special symbols for
  - Computing
  - Branching
  - Loop
- Link components by nesting

Structogram for Fibonacci

- Structogram enforces design rules
- Loops are easier recognized than in flow diagram

```
f_0 := 0; f_1 := 1
print f_0, f_1

f_i := f_{i-1} + f_{i-2}

if f_i > 100 then
  yes
  print f_i
  f_{i-2} := f_{i-1}
  f_{i-1} := f_i
else
  no
  loop until f_i > 100
```

C.5 Programming Projects

- Fibonacci numbers
- String conversion
- ...

Project: Fibonacci Numbers

**Definition:**
Fibonacci numbers
\[ f_0 = 0 \]
\[ f_1 = 1 \]
\[ f_i = f_{i-1} + f_{i-2} \]
\[ \rightarrow 0, 1, 1, 2, 3, 5, 8, \ldots \]

**Algorithm:**
Loop
1. Add: \[ f_i = f_{i-1} + f_{i-2} \]
2. Move: \[ f_{i-2} = f_{i-1}; f_{i-1} = f_i \]

**Project:**
Compute all Fibonacci numbers \( \leq 100 \)
Fibonacci Numbers in C

```c
int main()
{
    int current, last, last2;
    last = 1; last2 = 0;
    printf("0
1
"); /* print first 2 num.*/
    do {
        current = last + last2;
        printf("%d\n", current);
        last2 = last; last = current;
    } while (current <= 100);
    return 0;
}
```

Output:
0
1
1
2
3
5
8
...