Primer, Refresher, Reference to the

C Programming Language

ENGN8537

Embedded Systems

C

C is the most widely used programming language in Embedded Systems. The reason for this is simple: It's simple. It does only and exactly what you ask it to do, though this isn't always what you *think* you've asked it to do.

There are other languages out there that are arguably more suited to ES development; particularly Ada but also niche languages such as Esterel, Pearl, Euclid…

C provides no significant runtime environment, it stands completely on its own, but most C distributions ship with a standard set of libraries.

Looping Constructs

for loop

when the number of iterations is known a-priori

while loop

when the number of iterations is data- or event-driven, possibly zero

do loop

as with the while loop, but when the enclosed code must be executed at least once

executed once on loop entry

condition for loop exit

executed at the end of every iteration

```
for (i = 0; i < n; i++) {
    <code>
}
```

condition for loop exit

```
while (!finished) {
    <code>
}
```

```
do {
    <code>
} while (!finished);
```

condition for loop exit

Looping Constructs

for loop

when the number of iterations is known a-priori

while loop

when the number of iterations is data- or event-driven, possibly zero

do loop

as with the while loop, but when the enclosed code must be executed at least once

```
for (d = list_head;
    d->next;
    d = d->next ) {
    <code>
}
```

executed once on loop entry

condition for loop exit

executed at the end of every iteration

```
while (true) {
    <code>
}
```

condition for loop exit

```
do {
    <code>
} while (0);
```

condition for loop exit

Branching Constructs

if statement

Complex conditions and/or few options

case statement

Many ‘equals’ conditions on a single variable

```
if (fu) {
    confrobnicate();
} else if (bar) {
    crubnuckle();
} else {
    fail();
}

case (var) {
0:
    is_zero();
    break;
1:
2:
    is_one_or_two();
    break;
default:
    otherwise();
}
```

Arithmetic Operators

```
a = b     Basic Assignment
a + b     Addition
a – b     Subtraction
-a        Unary negation (additive inverse)
a * b     Multiplication
a / b     Division
a % b     Modulus
++a       Increment before evaluation
a++       Increment after evaluation
--a       Decrement before evaluation
a--       Decrement after evaluation
```

Binary and Boolean Operators

```
~a        Binary negation
!a        Boolean negation
a & b     Binary AND operation
a && b    Boolean AND operation
a | b     Binary OR operation
a || b    Boolean OR operation
a ^ b     Binary XOR operation
a == b    Boolean equals (test)
a != b    Boolean not-equals
a < b     Less-than
a > b     Greater-than
a <= b    Less-than or equal to
a >= b    Greater-than or equal to
a >> n    Bit-wise right shift by n places
a << n    Bit-wise left shift by n places
```

Other Operators

a[n]	Array element extraction
*a	Pointer dereference (value-at)
&a	Reference (address-of)
a.b	Structure reference (member b of structure a)
a->b	Structure dereference (member b of the structure pointed to by a)
a, b	'Comma', sequence point. Evaluates each expression in turn e.g. a=1, b=3
a?b:c	Turnery operator. If 'a' then evaluates to b, else evaluates to c
sizeof(a)	Evaluates to the number of bytes allocated for a
typeof(a)	Evaluates to the data type of a
offsetof(a,b)	Evaluates to the offset of member b of structure a (in bytes)

Data Types

{unsigned, signed} char	8-bits. If neither signed nor unsigned is specified, the signedness is implementation-dependent.
[unsigned] short	16-bits.
[unsigned] int	16-bits on most 8-bit microcontrollers, native bit width otherwise.
[unsigned] long	At least 32-bits, same as int on most 32- and 64-bit platforms.
[unsigned] long long	At least 64-bits.
float	IEEE single-precision floating point (usually)
double	IEEE double-precision floating point (usually)
long double	IEEE double- or quadruple-precision floating point

1	Number 1 as an int
1L	1 as a long
1UL	1 as an unsigned long
1.0	1 as a float
(int)(1.5)	The number 1.5 cast to the integer ‘1’, not rounded up as might be expected!
(int*)0x900000000	An integer address cast to a pointer

Data Types

```
enum a {
 FU,
 BAR,
 END
};
```

An int-like type designed to take any of three values FU, BAR, END (which the compiler will map to integer values).

```
struct a {
 int nub;
 char *nozzle;
};
```

A single data structure (record) containing two elements, an int called ‘nub’ and a pointer to a char where the pointer is called ‘nozzle’

Pointers

`int a, *b;`	Declare an int called ‘a’ and a variable ‘b’ that will hold the address of an integer; i.e. is a **pointer** to an integer
`b = &a;`	Set it so the pointer ‘b’ contains the address of the variable ‘a’
`a = 5;`	Set ‘a’ to 5 directly
`*b = 6;`	Set ‘a’ to 6 indirectly, by **dereferencing** the pointer

Register Access

```
volatile int *reg = (int*)0x90008000;
```
Create a pointer ‘reg’ whose value is the address of a register in memory. Tell the compiler not to optimize accesses to this variable.

```
*reg = 0x80;
```
Write the value ‘0x80’ to the register

```
*reg |= (1 << 4);
```
Set the 4th bit of the register to ‘1’

```
*reg &= ~(1 << 7);
```
Clear the 7th bit of the register