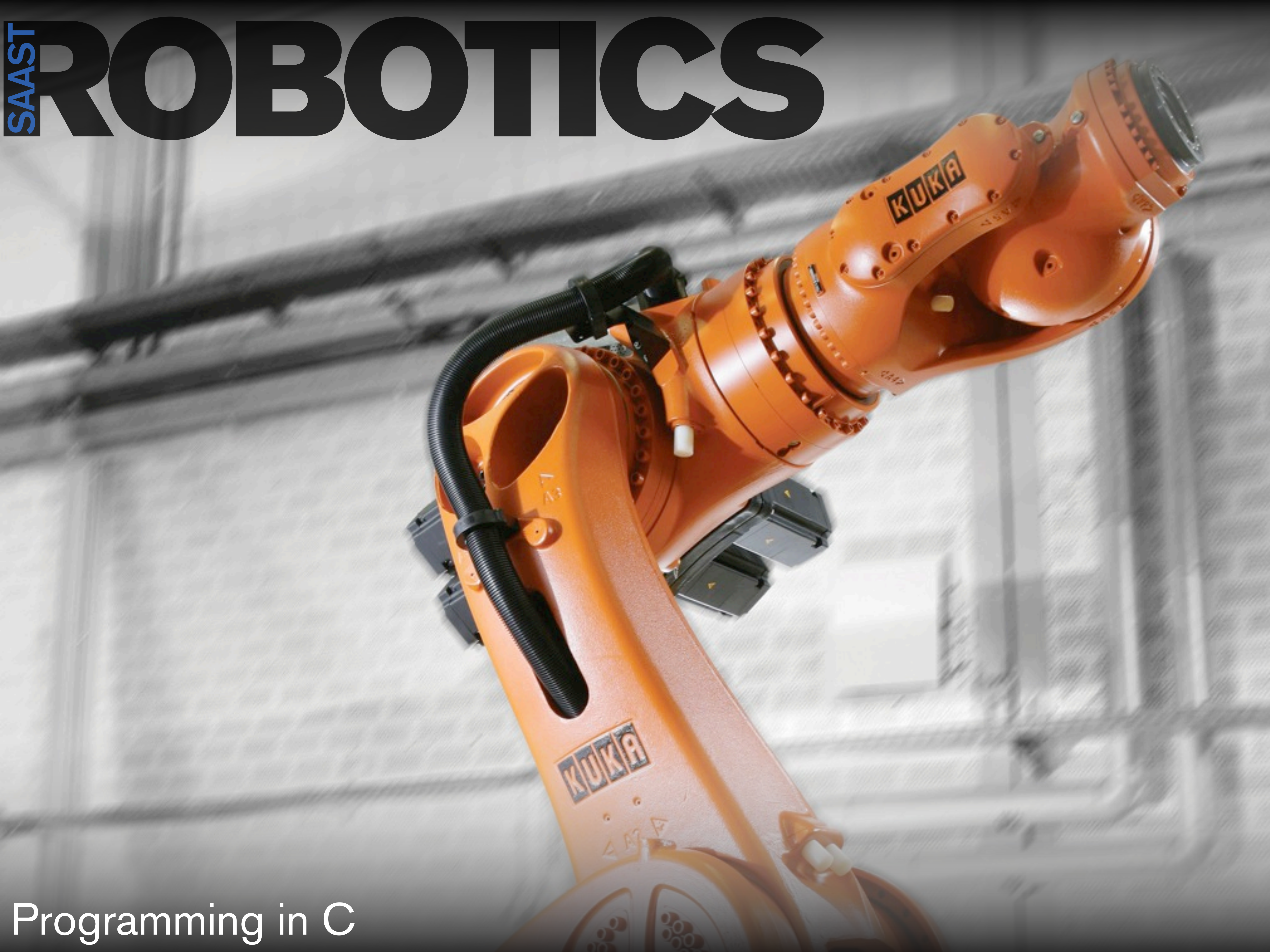
SAAST

ROBOTICS



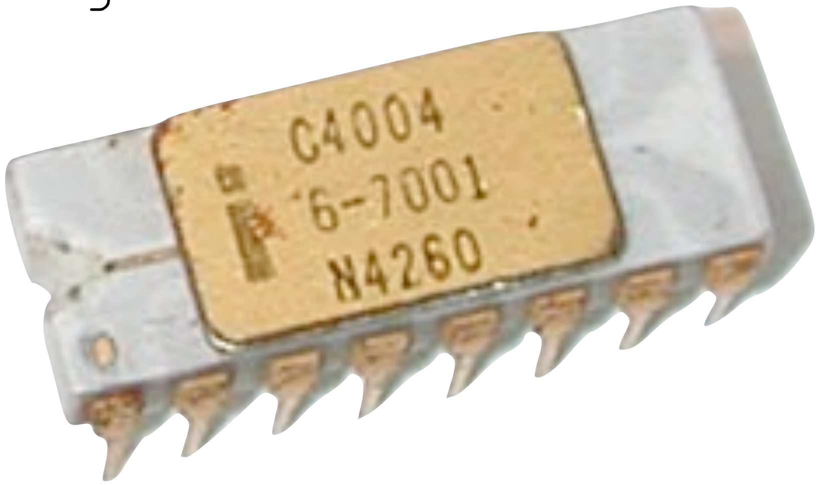
Programming in C

C

An introduction to programming

In the beginning...
things were written in **machine code**
and directly executed by the CPU

```
mem[0]=0x23; // load register a with following
mem[1]=0x00;
mem[2]=0xa8; // output a to r0
mem[3]=0x17; // increment a
mem[4]=0xa9; // output a to r1
mem[5]=0x17; // increment a
mem[6]=0xaa; // output a to r2
mem[7]=0x17; // increment a
mem[8]=0xab; // output a to r3
mem[9]=0x17; // increment a
mem[10]=0xac; // output a to r4
mem[11]=0x17; // increment a
mem[12]=0xad; // output a to r5
mem[13]=0x17; // increment a
mem[14]=0xae; // output a to r6
mem[15]=0x17; // increment a
mem[16]=0xaf; // output a to r7
mem[17]=0x17; // increment a
mem[18]=0x04; // jump to first page with following
mem[19]=0x02;
```



Then along came abstraction, with an **assembly language** to provide symbolic references for the numeric machine codes

```
Main:
   clrf   PORTB                ; initialize PORTB
   bsf    STATUS , RP0         ; Move to bank 1
   movlw  PORTB_DIR            ; value for TRISB
   movwf  TRISB                ; set by defined variable
   bcf    STATUS , RP0         ; Move to bank 1
   movlw  MAX_BITS
   movwf  BIT_COUNT            ; sets the bit count to seven
   clrf   INPUT_BYTE

SSTestFall:
   btfsc  PORTB , SS_BIT       ; check slave bit, if clear, skip next
   goto   SSTestFall           ; loop to check again
   goto   ClockTestFall        ; move on
```

and now we have...

C

preprocessor
directives

```
#include “saast.h”          // custom macros
```

constants

```
#define MAX 7
```

variables

```
void main(void){
  int i;
```

```
  m_init();                 // initialize the system
```

primary loop

```
  while(1){
    for(i=0; i<MAX; i++){
      toggle(PORTE,i);      // toggle Port E pins
    }
  }
}
```

C is case-sensitive!
white space does not matter

don’t forget the semicolon;
don’t forget the { }

#define constants
declare variables before use
no magic numbers!

compile and test as you go

comment your code!
please, comment your code...

translating common compiler errors

```
main.c: In function 'main':
main.c:17: error: 'i' undeclared (first use in this function)
main.c:17: error: (Each undeclared identifier is reported only once
main.c:17: error: for each function it appears in.)
make: *** [main.o] Error 1
```

(undeclared variable)

```
main.c: In function 'main':
main.c:43: error: expected declaration or statement at end of input
make: *** [main.o] Error 1
```

(missing “}”)

```
main.c: In function 'main':
main.c:19: error: expected ';' before '}' token
make: *** [main.o] Error 1
```

(missing “;”)

preprocessor directives

directives processed before compilation

include other files (generally “header” files with other # defines, function prototypes, etc.)

```
#include <filename.h>        // file in the include path
#include “filename.h”        // file in the current directory
```

define constants (essentially a find & replace - no semicolon!)

```
#define CONSTANT value
```

```
#define ENC_LINES 1024
#define TRUE 1
```

variables

variables must be declared before they are used!

```
type variable=initial, variable=initial;
```

```
int x;
short y, z;
long foo = 456;
unsigned int a=5, b=6;
char c = ‘b’;
```

type	bits	min	max
`char`	8	-128	127
`unsigned char`	8	0	255
`int`	16	-32768	32767
`unsigned int`	16	0	65535
`long`	32	-2147483648	2147483647
`unsigned long`	32	0	4294967295
`float / double`	IEEE32	1.175494E-38	3.402823E+38

ultimately, everything is binary to the CPU

basic operators

Arithmetic	
+	add
-	subtract
*	multiply
/	divide
%	modulus (remainder)

Conditional	
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Unary	
++	increment
--	decrement
!	not

<table>
<tr><th colspan="2">Logical</th></tr>
<tr><td>&&</td><td>and</td></tr>
<tr><td>||</td><td>or</td></tr>
<tr><td>!</td><td>not</td></tr>
</table>

all arithmetic and bitwise operators can be used in assignments

operator precedence

higher operators will be applied first

parenthesis	() []
structure access	. ->
unary	! ~ ++ -- - * &
multiply, divide, modulus	* / %
add, subtract	+ -
bit shifts	>> <<
inequality	< <= >= >
equal, not equal	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise OR	|
logical AND	&&
logical OR	||
ternary conditional	? :
assignment	= *= /= %= += -= <<= >>= &= |= ^=
comma	,

(when in doubt, add parentheses!)

iteration

WHILE: as long as the expression equals any non-zero value, the directives will be executed repeatedly

```
while(expression){
  directives;
}
```

```
while(!flag){
  directives;
}
```

```
int i=0;
while(i<10){
  directives;
  i++;
}
```

FOR: as long as the initialized variable is less than the termination value, the directives will be executed repeatedly

```
for(initialization; continuation; increment){
  directives;
}
```

```
int i;
for(i=0; i<10; i++){
  directives;
}
```

conditionals

IF: if the expression equals any non-zero value, directives will be executed

```
if(expression){
  directives;
} else {
  other directives;
}
```

expressions can be formed using:

Conditional	
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Logical	
&&	and
||	or
!	not

examples

```
if(a==b)

if(a!=b)

if(a<b)

if(a&&b)

if((a==5)&&(b!=4))

if(!c)
```