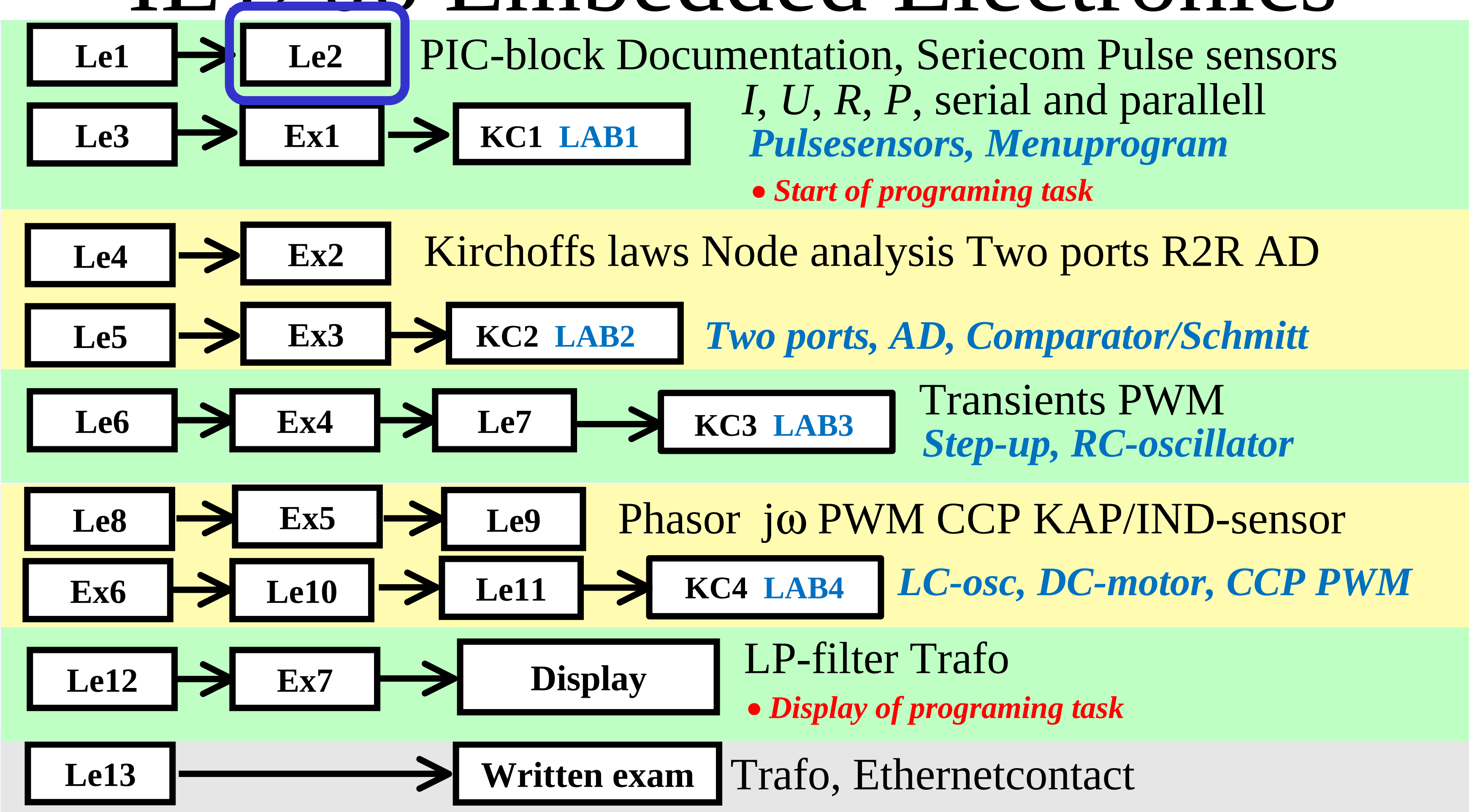
IE1206 Embedded Electronics



William Sandqvist william@kth.se

Handling of text strings and numbers

ABCDEFGHIJKLMNO
PQRSTUVWXYZØabc
defghijklmnopqr
stuvwxyzø&12345
67890($.,!?)

54

String ”Hello world!”

PIC-processors store string constants with the letters embedded in a table of instructions.

You get a letter by first loading the **W**-register with the with the letter sequence number in the table and then you do a jump to the table, eg. **`CALL text1`**.

The tabele **`text1`** starts with the instruction **`ADDWF`** wich forward *redirects* the jump to the sequence number in table (according to the number in **W**-register). There we find the instruction **`RETLW`** wich means return jump, but with the wished letter in the **W**-register.

Assembly code:

```
text1 addwf        ; jump indirect W
      retlw 0x48   ; 'H'
      retlw 0x65   ; 'e'
      retlw 0x6c   ; 'l'
      retlw 0x6c   ; 'l'
      retlw 0x6f   ; 'o'
 - - - - - - - - - -
```

This is called computed goto

Cc5x C-string ”computed goto”

- as function with ”computed goto”.

```
char text1( char W)
{
  skip(W);
  return ’h’; return ’e’; return ’l’; return ’l’;
  return ’o’; return ’w’; return ’o’; return ’r’;
  return ’l’; return ’d’; return ’\r’;
  return ’\n’; return ’\0’;
}
```

Skip-function jumps **W** *C-instruktions* (*special for Cc5x*)

- or the same with a simplified notation (pragma = special).

```
char text1( char W)
{
  skip(W);
  #pragma return[] = "hello world" ’\r’ ’\n’ ’\0’
}
```

To print a string

```
char i, k;
for(i = 0 ; ; i++)
 {
   k = text1(i);
   if( k == '\0') break;   // found end of string
   putchar(k);
 }
```

Fetch a letter

until end of string

print the letter

```
char text1( char W)
{
  skip(W);
  #pragma return[] = "hello world" '\r' '\n' '\0'
}
```

(C pointers * address & and dereference *)

Short on C-language pointer – do you remember?

`int b;` Declaration of integer variable **b**. Place is reserved.

`b = 18;` Definition of variable **b**. Now it contans number 18.

`&b` Address operator. The address to variable **b**. ( )

`int * c;` Declaration of int-pointer variable **c**.

`c = &b;` Now **c** points at **b**.

`*c` Dereference operator. That c points at. ( )

`*c = 19;` Number 19 is stored at the place **c** points to.
Now the content **`b = 19`**. ( )

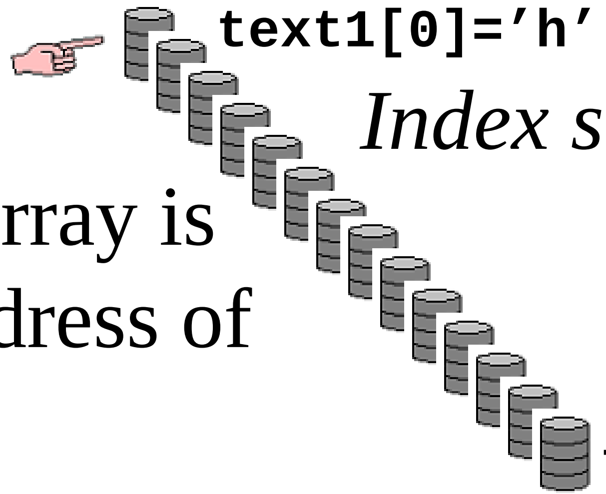
William Sandqvist william@kth.se

C-language ”strings”

`const char text1[]=”hello world\r\n”;`

A string is a char-array wich ends with **`’\0’`**

`&text1[0]=text1`



`text1[0]=’h’`

Index starts with 0

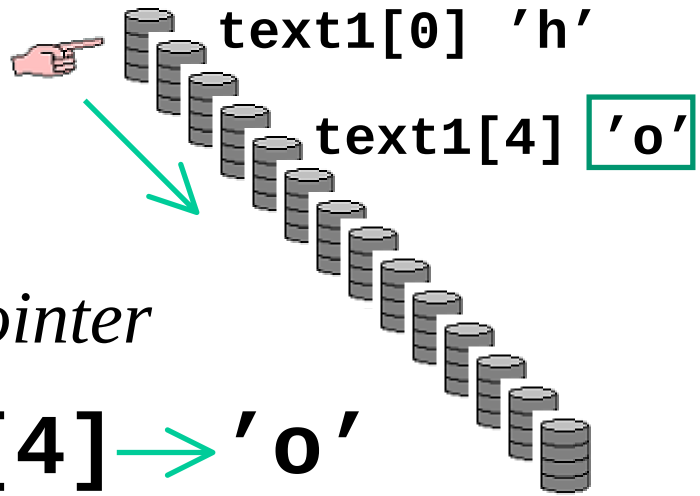
The name of the array is also the starting adress of the string!

`text1[13]=’\0’`

Addresses are stored in pointer variables

```
char * a;
```

```
a = &text1[0];
```



step by index or step with the pointer

Dereference with Index: a[4] → ’o’

Dereference with Pointer: *(a+4) → ’o’

William Sandqvist william@kth.se

Addresses as function parameters

If a function is to print different strings one needs to be able to send the string starting address.

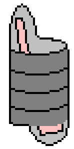
Function declaration:

```
void string_out( const char * );
```

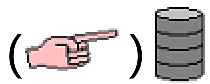
Place for string start address

Addresses as function parameters

Function definition:



```
void string_out( const char * s )
{
  char i,k;
  for(i = 0 ; ; i++)
   {
     k = s[i];
     if( k == '\0') return;
     putchar(k);
   }
}
```



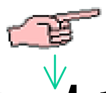
Now dereference with index

William Sandqvist william@kth.se

Addresses as function parameters

Function call:

send the start address



```
string_out( &text1[0] );
```

or like this

```
string_out( ”hello world\r\n” );
```

With this way of writing the string constant is first stored in memory (as a computed goto-table) but then it's just the "start address" that is passed as a parameter to the function.

Not much to point at ?

```
const char text1[]=”hello world\r\n”;
```

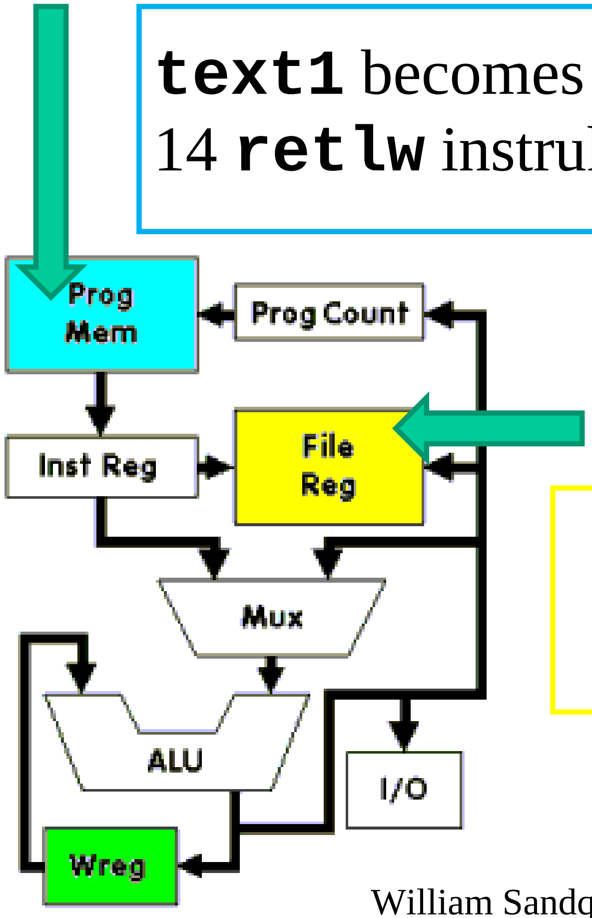
There is not much to point at in a midrange PIC processor. The compiler Cc5x permits, despite this, the use of pointers and some C syntax for pointers.

This is just "for show". Constant strings are stored as "computed goto" in the program memory, and string variables as arrays in register area.

Not much to point at ?

const char text1[]="hello world\r\n";

text1 becomes a "computed goto" table with 14 **retlw** instruktions in program memory



char buffer[21];

buffer becomes 21 gpr-registers in the registerfile

hello.c

```
void main(void)
{
  char i;
  initserial(); /* Initialize serial port  */
  string_out(”hello world\r\n”);
  while(1) nop();
}

void string_out( const char * s )
{
  char i,k;
  for(i = 0 ; ; i++)
   {
     k = s[i];
     if( k == ’\0’) return;
     putchar(k);
   }
}
```

Enter a string

```
char s[11]; /* room for 10 charachters and ’\0’ */
```

Call: `string_in( &s[0] );`

`string_in()` enters characters into the buffer **`s`** until **”enter”** (or max number of characters, **10**).

```
void string_in( char * str )
{
  char n, c;
  for(n = 0; ; n++ )
   {
     c = getchar( ); /* input 1 character       */
     str[n] = c;     /* store the character     */
     putchar( c );   /* echo the character      */
     if( (n == 10)||(c=='\r' )) /* end of input */
       {
         str[n] = '\0'; /* add "end of string"  */
         return;
       }
   }
}
```

comp_str()

```
bit comp_str( char * in_str, const char * ref_str )
{
  char i, c, d;
  for(i=0; ; i++)
   {
     c = in_str[i];
     d = ref_str[i];
     if(d != c ) return 0;     /* no match    */
     if( d == '\0' ) return 1; /* exact match */
    }
}
```

comp_str() jcompares contet of two strings, if they are equal a bitvariable returns = 1

Password Control!

Formated print `printf()`

the time is 12 : 34

- *ANSI-C has function* **`printf()`**

Format string, text with variable positions — Variable list

```
printf("the time is %d : %d\n", hours, minutes);
```

print of integer variables — New line

`%d` print a integer

`%x` print a integer in hexadecimal

`%f` print a float

`\n` newline **`\t`** tabulator

to use `printf()`

```
#include <stdio.h>
```

`printf()` -lookalike

Standard ANSI-C has input and output functions `printf()` and `scanf()` in standard library `stdio.h`. This This is far too broad and complicated functions of a PIC processor. For those who are familiar with C a function thats at least similar to `printf()`**would be good to have**a. Especially when debugging program code.

```
void printf(const char *str, char var);
```

```
printf("Hello World!\n\r", 0);
printf("Number 234 as unsigned: %u\n\r", 234);
printf("Number 234 as signed: %d\n\r", 234);
printf("Number 234 as binary: %b\n\r", 234);
printf("Print a char: %c\n\r", 'W');
```

What would printout be?

`printf()` -lookalike

This is the printout!

```
Hello World
Number 234 as unsigned: 234
Number 234 as signed: -022
Number 234 as binary: 11101010
Print a char: W
```

From binary to ASCII string

Eg. numbert 123

```
s[3]='\0';

s[2] = 123%10 +'0' = '3'
123/10 = 12

s[1] = 12%10 +'0'  = '2'
12/10 = 1

S[0] = 1%10 +'0'   = '1'
```

A conversion algorithm.

After three steps **s[]** contains the ASCII-characters to print.

s[]={'1','2','3','\0'}

```
void printf(const char *str, char var)
{
  char i, k, m, a, b;
  for(i = 0 ; ; i++)
   {
     k = str[i];
     if( k == '\0') break; // at end of string
     if( k == '%') // insert variable in string
      {
        i++;
        k = str[i];
        switch(k)
         {
           case 'd': // %d signed 8bit
             if( variable.7 ==1) putchar('-');
             else putchar(' ');
             if( variable > 127) variable = -variable; // no break!
           case 'u': // %u unsigned 8bit
             a = variable/100;
             putchar('0'+a); // print 100's
             b = variable%100;
             a = b/10;
             putchar('0'+a); // print 10's
             a = b%10;
             putchar('0'+a); // print 1's
             break;
             ...
```

Conversion into decimal is demanding.

```
 ...
 case 'b': // %b BINARY 8bit
    for( m = 0 ; m < 8 ; m++ )
     {
       if (variable.7 == 1) putchar('1');
       else putchar('0');
       variable = rl(variable);
     }
    break;
 case 'c': // %c 'char'
    putchar(variable);
    break;
 case '%':
    putchar('%');
    break;
 default: // not implemented
    putchar('!');
}
   }
  else putchar(k);
 }
}
```

Print binary variables do not take that much resources

In practice, only the conversions you really need is included - everything takes place and takes time execution time.

William Sandqvist william@kth.se