Enginyeria de Sistemes TIC

Programació a Baix Nivell

Apunts de l'assignatura

Sebastià Vila-Marta

Departament d'Enginyeria Minera
Industrial i TIC

21 de maig de 2020

Índex

1 Tutorial C99

El llenguatge de programació `C` té un paper fonamental en aquesta assignatura. Hi ha molta i molt bona literatura sobre el llenguatge `C`, especialment en la seva variant `C89`.

Com a bibliografia recomanem [KR88], que hauria de ser un dels llibres de capçalera. En segona instància també els llibres lliures [BBD91; BH02] i els apunts [Par03] poden ser d'utilitat.

A mode de xuletari, el tríptic [Sil99], és d'allò més pràctic.

Finalment, molts llibres sobre la disciplina dediquen capítols a introduït el llenguatge `C`. Aquest és el cas de [Rus10]

En aquest apèndix només hi trobareu un resum molt breu i sintètic dels aspectes més interessants en l'àmbit d'aquest document que incorpora l'estàndard `C99`. Les raons i la lògica dels canvis que incorpora `C99` respecte `C89` estan documentades a [Joi03].

1.1 Primer exemple

Un programa codificat en `C` és una col·lecció de funcions una de les quals té un nom privilegiat, `main()`, i actua com a programa principal. La figura 1.1 és un programa en `C` que conté una funció i un programa principal.

Si aquest programa el contingués el fitxer `senzill.c`, el compilaríem i muntaríem amb la comanda:

```
$ gcc -std=c99 -o senzill senzill.c
```

i tot seguit provaríem d'executar-lo fent:

```
$ ./senzill
```

L'executable cal prefixar-lo per nom del directori atès que les shells de UNIX només cerquen executables en els directoris que indica la variable d'entorn `PATH`. Habitualment el directori de treball no forma part d'aquesta col·lecció de directoris. Per tant cal explicitar on és el fitxer executable. En aquest cas usem un camí relatiu.

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

int dobla(int x) {
  int r;

  r = x * 2;
  return r;
}

int main() {
  int v, n;

  n = scanf("%d", &v);
  assert(n == 1);
  printf("%d\n", dobla(v));

  return 0;
}
```

Programa 1.1: Programa senzill en C99

1.2 Variables i tipus escalars

1.2.1 Variables i tipus

Les declaracions de variables tenen aquest aspecte:

```c
int x, y;
```

i poden ocórrer en qualsevol lloc del codi[1]. Com es veu, primer s'escriu el tipus de dades i a continuació una llista d'identificadors de variable.

Els tipus de dades més corrents són els següents:

Tipus	Significat	Exemple
bool	Un booleà. Cal incloure prèviament el header stdbool.h.	`#include <stdbool.h>` `bool b; b = false;`
char	Un caràcter (no una cadena!)	`char c; c = 'a';`
short	Un enter "curt". Habitualment 16 bit.	`short x; x = 67;`
int	Un enter habitualment de 32 bit	`int i; i = -12;`
long	Un enter llarg. Habitualment de 64 bit	`long l; l = 3L;`
float	Un real de precisió simple	`float x;` `x = 23.45; x = 2.0e-13;`

[1] És una característica apareguda a C99

Classificació				Tipus
Integrals	Booleà			bool
	Caràcter			**char**
	Enter	Ordinari	Amb signe	**short**
				int
				long
			Sense signe	**unsigned short**
				unsigned int
				unsigned long
		Mida fixa	Amb signe	int8_t
				int16_t
				int32_t
			Sense signe	uint8_t
				uint16_t
				uint32_t
Reals	Flotant			**float**
				double

Taula 1.1: Classificació dels tipus escalars de C

Les mides dels tipus depenen de la plataforma i el compilador. Per tant no són portables. El header limits.h defineix una sèrie de constants que permeten conèixer els límits de cada tipus de dades.

La taula 1.1 mostra la classificació habitual dels tipus escalars de C99.

Els tipus disposen d'operació de conversió de tipus explícita (*cast*):

```
int i;
short s;

i = 78;
s = (short)i;
```

De la mateixa manera, existeixen un conjunt de regles de conversió que s'apliquen de forma automàtica (*coercion*)[2]. En general un tipus promociona automàticament a un tipus de grandària superior. Cal, però, tenir cura quan es barregen tipus amb i sense signe. És una font important de problemes i cal anar amb peus de plom.

Tots els tipus enters disposen de la versió sense signe. El tipus sense signe associat a un tipus amb signe es construeix prefixant amb **unsigned**. També es poden definir constants sense signe usant el sufix U a tal efecte. Per exemple:

```
unsigned int i;

i = 34U;
```

[2]Noteu la diferència de significat entre *cast* i *coercion*.

Sorprenentment, el tipus **char** es considera amb signe i, per tant, existeix el tipus **unsigned char**.

També és possible usar constants enteres escrites en base 2[3], 8 i 16:

```
unsigned int x;

x = 0b010; // binari
x = 0347; // octal
x = 0x3af; // hexadecimal
```

Per evitar la manca de portabilitat dels tipus de dades pel que fa a la seva longitud, C99 defineix un conjunt de tipus amb la mida ben definida. Per usar-los és imprescindible incloure el header stdint.h. Els tipus en qüestió són uint8_t, uint16_t, uint32_t i els seus corresponents amb signe int8_t, int16_t i int32_t, que com el seu nom indica corresponen a mides de 8 bit, 16 bit i 32 bit respectivament. Cal usar aquests tipus quan ens cal representar quelcom que ha de tenir una longitud fixada, per exemple un port o un registre.

La funció predefinida **sizeof** permet consultar en temps d'execució la mida de qualsevol tipus. El resultat es dona usant com a unitat la mida d'un **char**. Així, per exemple, pot succeir que **sizeof**(**int**) sigui 4, indicant que un **int** ocupa 4 **char**'s. La mida en bits d'un **char** es pot consultar emprant la constant de limits.h anomenada CHAR_BIT. Molt sovint CHAR_BIT==8.

1.2.2 Operacions

A C l'assignació es considera una operació que, per efecte lateral, modifica el contingut d'una variable. Així, per exemple, x=3 és una operació que modifica el valor de la variable x i, a la vegada, val 3. D'aquesta forma, es pot escriure y=(x=3) o, traient els parèntesis y=x=3, que té l'efecte d'assignar el valor 3 tant a x com a y.

Les operacions sobre el tipus bool són les habituals:

Operació	Significat
`||`	Disjunció booleana (OR).
`&&`	Conjunció booleana (AND).
`!`	Negació booleana (NOT).
`==`	Igualtat.
`!=`	Diferència.

Sobre els tipus integrals (enters), les operacions es classifiquen en:

- Operacions aritmètiques:

[3] A partir de C99.

Operació	Significat
+	Suma.
*	Producte.
-	Resta o canvi de signe.
/	Divisió entera quan ambdós operands són enters.
%	Mòdul.

- Operacions booleanes:

Operació	Significat
==	Igualtat.
!=	Diferència.
>, >=, <, <=	Comparació

- Operacions bit a bit:

Operació	Significat
>>	Shift dreta. Amb extensió de signe si el tipus és signed.
<<	Shift esquerra. Amb extensió de signe si el tipus és signed.
~	Complement.
&	AND bit a bit.
|	OR bit a bit.
^	OR exclusiva bit a bit.

Els enters i booleans disposen a més d'operacions de modificació, que combinen l'assignació i una operació específica. Per exemple, les dues operacions següents són equivalents:

```
x *= 2;
x = x * 2;
```

De manera similar, es disposa d'operacions d'auto increment/decrement. Així, i++ significa: "avalua i i, posteriorment, incrementa el seu valor". Això es coneix com post-increment. De manera simètrica, el pre-increment ++i significa "incrementa i i avalua-la". D'acord amb això el següent fragment de programa és tal que en acabar a==128, ra==128, b==128 i rb==127.

```
int a, b, ra, rb;

a = b = 127;
ra = ++a;
rb = b++;
```

També disposem d'operadors de pre i post-decrement amb la sintaxi esperable: j−− i −−j.

En el cas dels reals en coma flotant, les operacions aritmètiques bàsiques són:

Operació	Significat
+	Suma.
*	Producte.
-	Resta o canvi de signe.
/	Divisió real quan un dels operands és real.

A banda de les operacions elementals, si s'inclou el header math.h es pot accedir a una col·lecció d'operacions i funcions més àmplia. Aquesta col·lecció inclou funcions com:

Operació	Significat
sin, cos, tan	Funcions trigonomètriques.
sqrt	Arrel quadrada.
exp	Exponenciació.
log	Logaritme natural.

En qualsevol referència com ara la Viquipèdia i, naturalment l'estàndard [Joi07], trobareu la documentació completa.

1.2.3 Algunes transgressions

De fet a C és corrent, però delicat, tractar indiscriminadament qualsevol tipus enter com si es tractés d'un booleà i confondre deliberadament caràcters i enters. Les conversions entre tipus faciliten aquesta pràctica.

Pel que fa als booleans, qualsevol tipus integral pot ser considerat un booleà sota la premissa de que el valor 0 cal entendre'l com a false i la resta de valors com a true. Així, el següent exemple és plenament vàlid:

```
int i;
bool b;

b = false; i = 33;
if (b || i)
  printf("Aja!");
```

Seguint amb les transgressions, també es comú entendre els **char** com a enters «petits». En el següent exemple, el programador tracta la variable c sense donar-li en cap moment el significat de caràcter:

```
char c;
int i;

c = 34;
i += c
```

En casos com aquest és més interessant usar una variable de tipus int8_t en comptes de **char**.

1.2.4 Prioritat i associativitat dels operadors

Atesa la quantitat d'operadors diferents emprats a C, la interpretació de les expressions pot ser difícil i cal tenir a mà la taula de prioritats i associativitats. La taula 1.2 mostra aquesta informació.

Prioritat	Operador	Associativitat
1	++, −− (post) [], () ., −>	LR
2	++, −− (pre) +, −, ∗, & (unaris) (cast), ~, !	RL
3	∗, /, % (multiplicatives) +, − (additives)	LR
4	>>, <<	
5	>, >=, <, <=	
6	==, !=	
7	&	
8	^	
9	|	
10	&&	
11	||	
12	?: (expr. cond.)	RL
13	=, +=, ∗=, etc.	
14	, (coma)	LR

Taula 1.2: Taula de prioritats i associativitat dels operadors C.

1.3 Implementació de mòduls

1.3.1 Esquema bàsic

Aquesta secció descriu la forma d'implementar el concepte de mòdul usat en el disseny d'una aplicació a l'entorn de `C`. Tot i que hi ha diversos esquemes possibles de treball, aquí se'n presenta només un d'específic que és prou general i ortogonal.

La idea és aprofitar la possibilitat de fer compilació separada per implementar mòduls seguint un esquema d'organització ben establert. Entenem que un mòdul és un contenidor de funcions, constants i tipus de dades que poden ser usats per altres mòduls. Addicionalment un mòdul defineix un espai de noms privat que permet ocultar detalls d'implementació a la resta dels mòduls que composen un projecte. Un mòdul, doncs, té una part pública i una part privada.

L'organització que es proposa implementa un mòdul sobre dos fitxers, un header i un fitxer de codi c. El primer conté la part pública del mòdul mentre que els segon conté la part privada. Vegem-ho en una sèrie d'exemples.

Suposem que el mòdul mod1, per exemple, és un mòdul funcional que implementa dues funcions públiques: inc() i dec(). La implementació d'aquestes funcions és privada. Aleshores la forma d'implementar el mòdul és escriure dos fitxers. El primer, un fitxer de headers l'anomenem mod1.h i té aquesta forma:

```
#ifndef MOD1_H
#define MOD1_H

void inc(int *const i);
void dec(int *const i);

#endif
```

L'estructura d'un header és sempre la mateixa. Primer hi ha un parèntesi format per les sentències de preprocessador **#ifndef** i **#endif** que impossibilita una doble inclusió d'aquest header en un fitxer. Tot i que pot semblar excessiu, és possible que es doni una doble inclusió atès que en un fitxer determinat es pot incloure indirectament el mateix header per diversos camins i ser difícil de detectar. Aquest parèntesi, que cal escriure sempre, evita aquest problema. Noteu que el símbol MOD1_H ha de ser particular de cada mòdul. Així, si el mòdul es digués az34, per exemple, usaríem el símbol AZ34_H.

A l'interior del parèntesi, en aquest cas, el mòdul conté les capçaleres de les funcions públiques. Noteu que les capçaleres són imprescindibles per a que el compilador pugui saber si les crides a aquestes funcions són correctes.

Pel que fa a la part privada del mòdul, la escriuríem en el fitxer mod1.c, que tindria aquest aspecte:

```
#include "mod1.h"

static int suma(int k, int v) {
  return k + v;
}

void inc(int *const i) {
  *i = suma(*i, 1);
}

void dec(int *const i) {
  *i = suma(*i, -1);
}
```

Noteu que, de forma una mica artificial, s'ha usat una funció privada del mòdul per implementar dec() i inc(). Aquesta funció, suma(), s'a declarat com **static** per fer-la privada i, naturalment, el seu prototip no apareix al fitxer mod1.h.

Si des de la implementació d'un segon mòdul, posem que és modu.c, es volen usar les funcions de mod1, cal simplement afegit l'include escaient i cridar-les on convingui:

```
#include <stdio.h>
#include "mod1.h"
```

```
int main() {
  int v = 3;

  inc(&v);
  printf("%d\n", v);

  return 0;
}
```

Per compilar aquest exemplet simplement caldria fer:

```
$ gcc -std=c99 -I. -c mod1.c
$ gcc -std=c99 -I. -c modu.c
$ gcc -std=c99 -o modu modu.o mod1.o
```

o bé, en una sola ordre fent:

```
$ gcc -std=c99 -I. -o modu modu.c mod1.c
```

1.3.2 Singleton

Un *singleton* en la terminologia d'orientació a objectes és un objecte del que només n'existeix una sola instància en una aplicació. Molt sovint s'usen els mòduls per a implementar aquest concepte. Imagineu que en una aplicació feta de diversos mòduls hi ha un únic comptador que s'incrementa de 5 en 5 i es decrementa de 3 en tres. Del comptador la única informació útil és saber si és positiu o negatiu. Aquest comptador es pot implementar com un mòdul que exporta les operacions indicades i que *encapsula* la representació del comptador de manera privada.

La implementació del header seria la següent si assumim que correspon al fitxer `compt.h`:

```
#ifndef COMPT_H
#define COMPT_H

#include <stdbool.h>

void inc(void);
void dec(void);
bool is_positive(void);

#endif
```

Noteu:

- Només conté els prototips de les funcions públiques però cap referència a la implementació del comptador.
- Inclou el header `stdbool` atès que s'usa el tipus `bool`.

La implementació, desada en el fitxer `compt.c` seria la següent:

```
#include "compt.h"

static int comptador = 0;

void inc(void) {
  comptador += 5;
}

void dec(void) {
  comptador -= 3;
}

bool is_positive(void) {
  return comptador >= 0;
}
```

Noteu l'ús de **static** per indicar que la variable comptador és privada i no pot ser usada des de cap altre mòdul.

Usant aquest mòdul assegureu que no pot haver-hi cap altra còpia del comptador en un programa.

1.3.3 Classes

A vegades els mòduls s'usen per implementar estructures similars en certa manera a una classe: un tipus de dades acompanyat d'un conjunt d'operacions. En aquests casos tant el tipus de dades com les seves operacions són públiques. Imaginem que volem implementar un tipus que permeti definir piles d'enters de mida no afitada. Aleshores el header del mòdul, que podríem anomenar `stack.h`, seria similar a:

```
#ifndef STACK_H
#define STACK_H

typedef void *stack;

stack push(stack p, int i);
stack pop(stack p);
int top(stack p);
stack empty(void);

#endif
```

i la seva implementació, sense tenir en compte possibles excepcions, correspondria a:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
```

```
typedef struct sc {
  int v;
  struct sc *next;
} cell_t;

stack empty(void) {
  return NULL;
}

stack push(stack p, int i) {
  cell_t *c = malloc(sizeof(cell_t));
  c->v = i;
  c->next = p;
  return c;
}

stack pop(stack p) {
  cell_t *c = ((cell_t *)p) -> next;
  free(p);
  return c;
}

int top(stack p) {
  return ((cell_t *)p) -> v;
}
```

D'aquesta manera, qualsevol altre mòdul podria definir variables de tipus stack i operar amb elles fent:

```
#include "stack.h"

...
stack s = empty();
s = push(s,30);
s = push(s, 10);
printf("%d\n", top(s));
s = pop(s)
...
```

2 Arduino Uno

2.1 Introducció

Les plaques "Arduino" són una família de plaques de desenvolupament dissenyades al voltant dels microcontroladors de la família AVR d'Atmel. L'Arduino Uno, [Ard11], és una placa amb les següents característiques:

1. Basada en el microcontrolador d'Atmel ATmega328, amb arquitectura AVR, [Atm09].
2. Tensió operativa de 5 V.
3. Voltatge recomanat d'entrada de 7–12 V.
4. Límits del voltatge d'entrada de 6–20 V.
5. 14 pins d'entrada/sortida digital (6 dels quals ofereixen sortida PWM).
6. 6 entrades analògiques,
7. Màxim corrent continu per pin d'entrada/sortida de 40 mA a 5 V i de 50 mA a 3.3 V.
8. Memòria flash de 32 KB, dels quals 0.5 KB els usa el carregador.
9. SRAM de 2 KB.
10. EEPROM d'1 KB.
11. Velocitat de rellotge de 16 MHz

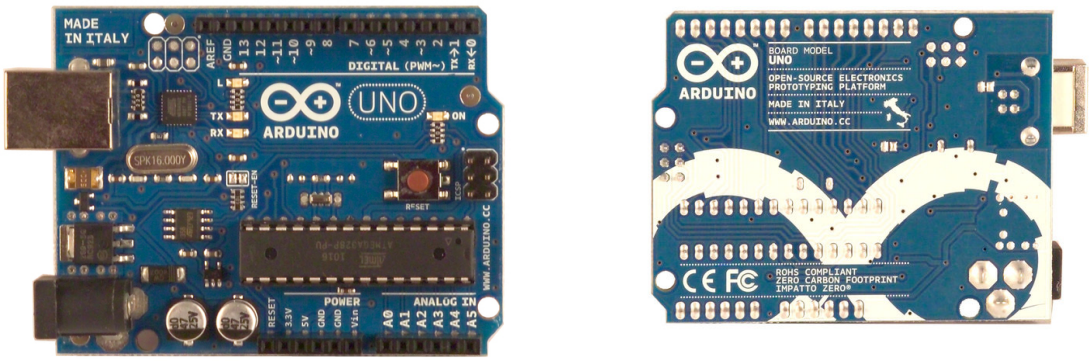


Figura 2.1: Placa d'Arduino Uno. Part superior i inferior.

2.2 Esquemes de la placa

La figura 2.2 mostra un esquema integral de la placa Arduino Uno. La figura 2.3 il·lustra la correspondència entre els pins del microcontrolador ATmega328P i la placa Arduino.

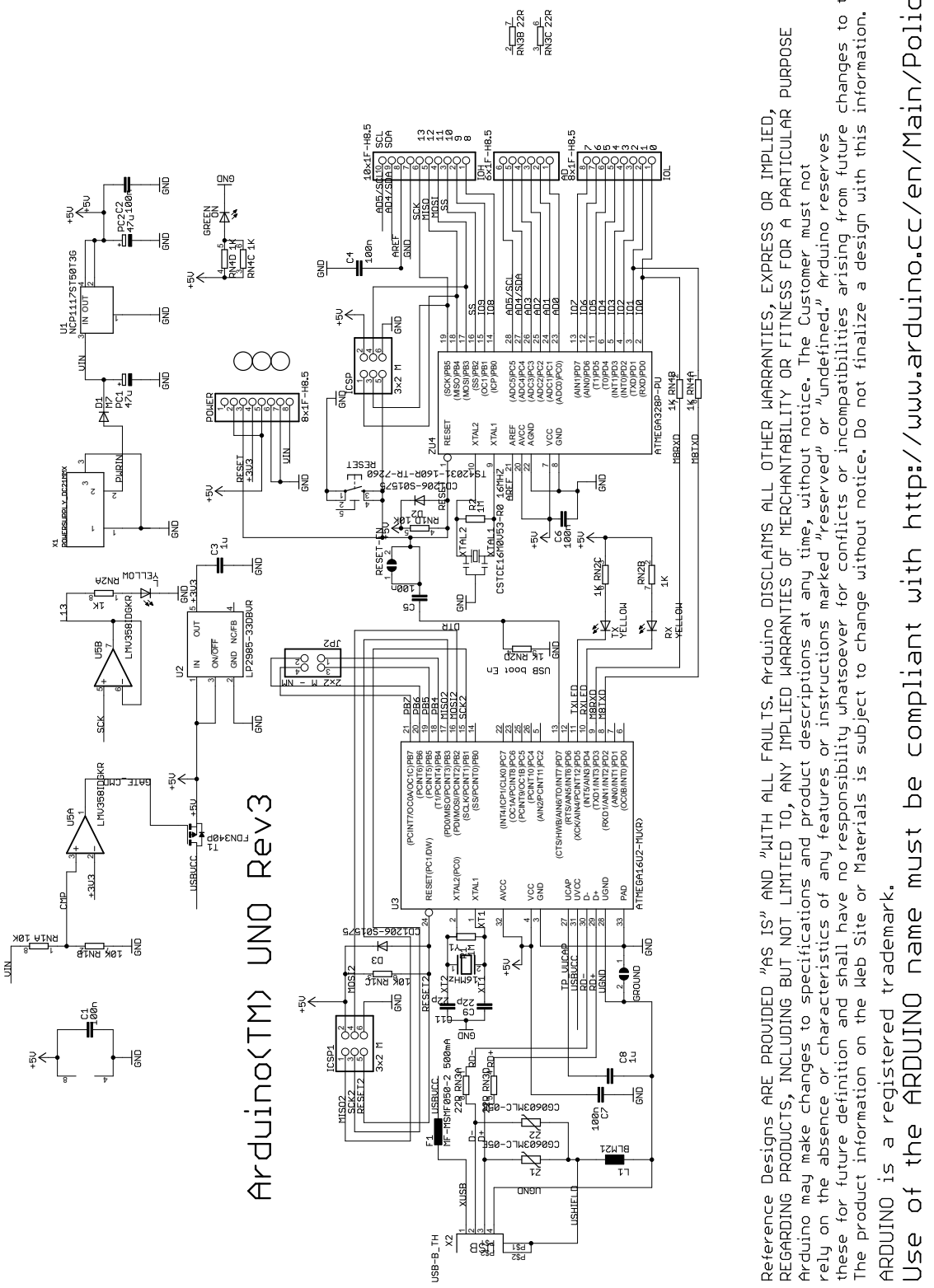


Figura 2.2: Esquema de la placa Arduino Uno

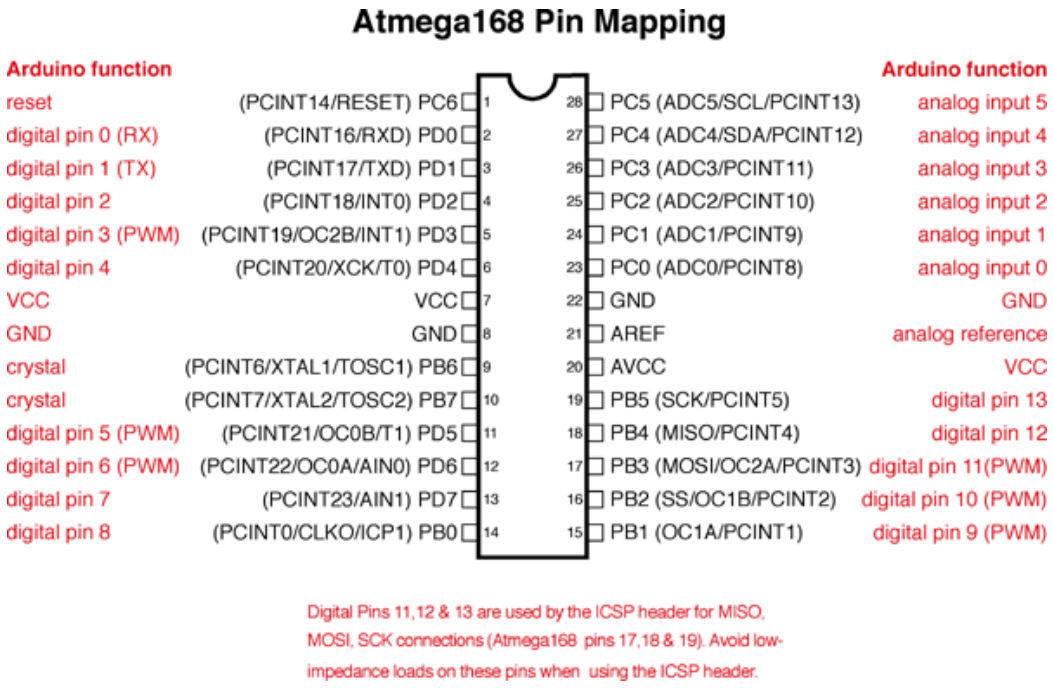


Figura 2.3: Correspondència entre els pins del microcontrolador i la placa.

3 Toolchain de GNU

El llenguatge de programació `C` té un paper fonamental en aquest material. D'Aquí en endavant s'assumirà un bon coneixement d'aquest llenguatge i una certa experiència en el seu ús. En cas que no sigui així, cal tenir en compte que hi ha molta i molt bona literatura sobre el llenguatge `C`, especialment en la seva variant `C89`.

Com a bibliografia recomanem [KR88], que hauria de ser un dels llibres de capçalera. En segona instància també els llibres lliures [BBD91; BH02] i els apunts [Par03] poden ser d'utilitat.

A mode de promptuari, el tríptic [Sil99], és d'allò més pràctic.

Finalment, molts llibres sobre la disciplina dediquen capítols a introduir el llenguatge `C`. Aquest és el cas, per exemple, de [Rus10].

3.1 Compilació. Compilació creuada

Quan es descriuen programes usant un llenguatge d'alt nivell com ara `Python`, `C`, `VHDL` o `Octave`, és necessari acabar-los traduint al llenguatge màquina de la CPU que finalment haurà d'executar el programa. Els programes que fan aquesta feina els anomenem *processadors de llenguatges*.

La tecnologia actual coneix dues grans tècniques per a processar els llenguatges de programació:

Interpretació La interpretació consisteix en llegir el programa escrit en un llenguatge d'alt nivell HLL sentència a sentència i simular allò que hauria de fer la sentència en qüestió. Per aquesta raó els intèrprets sovint es diu que en realitat són *màquines virtuals* que tenen per llenguatge màquina el propi HLL.

Compilació La compilació consisteix en traduir tot el programa escrit en HLL a un nou programa escrit en el llenguatge màquina de la CPU escaient i equivalent al primer. Una vegada traduït, el programa és directament executable.

Una i altra tècniques tenen els seus avantatges i els seus inconvenients i en cap cas pot afirmar-se que una és millor que l'altra. La principal diferència rau en el fet que els intèrprets tradueixen les sentències una a una i en el moment en que s'està executant el programa. Per tant, ho fan cada vegada que el programa s'executa. Com a conseqüència l'execució dels programes és més lenta i el sistema més flexible ja que pot modificar el seu comportament mentre s'està executant. D'altra banda, en la compilació només es tradueixen les sentències una única vegada i es fa en diferit, abans que el programa sigui executat. Generalment els programes que en resulten són més ràpids executant-se. La compilació, però, és una tècnica menys flexible ja que una vegada compilat el programa aquest resta fixat.

Certs processadors de llenguatge combinen una i altra tècnica. En aquests casos és corrent tenir un compilador que tradueix del HLL a un llenguatge intermedi que després és interpretat. Aquest és el cas, per exemple, de `Python` o de `Prolog`. `Octave` es processa amb un intèrpret pur. `Ghdl`, el simulador de `VHDL`, és en realitat un compilador. En el cas de `C` la tècnica usada és la compilació.

Quan cal fer un desenvolupament amb un(s) llenguatge de programació usant un compilador sovint s'usa el terme *toolchain* per referir-se al conjunt d'eines necessàries. Aquest conjunt inclou el compilador i altres eines no menys essencials que l'acompanyen.

Un toolchain de compilació està pensat per a traduir d'un llenguatge d'alt nivell, per exemple `C`, a un llenguatge màquina.

Generalment, si treballem en un cert computador $C1$ i el toolchain que esta instal·lat es T, aquest toolchain es podrà executar en el propi computador $C1$ i els programes compilats amb aquest toolchain també es podran executar en el computador $C1$. Per això aquest toolchain l'escrivim com T_{C1}^{C1}: s'executa a $C1$ i genera programes executables a $C1$. El procés d'un programa P_{HLL} el podem expressar així: $P_{HLL} \Rightarrow_{T_{C1}^{C1}} P_{C1}$.

A vegades, però, ens interessa escriure programes en la nostra estació de treball $C1$ però volem que es puguin executar en un computador diferent $C2$. En aquest cas ens cal un toolchain diferent de l'anterior: T_{C1}^{C2}. Això és el que s'anomena un *toolchain creuat*. Si processem el mateix programa d'abans, P_{HLL}, ara el resultat és: $P_{HLL} \Rightarrow_{T_{C1}^{C2}} P_{C2}$.

3.2 Instal·lació del toolchain per AVR

Per desenvolupar aplicacions sobre Arduino hi ha diverses possibilitats. Dues de les principals són:

1. Usar l'entorn de desenvolupament específic d'Arduino, que permet treballar amb `C++` i un entorn de desenvolupament específic. Aquest entorn és programari lliure i està suportat.

2. Usar el conjunt d'eines estàndards de GNU en versió per a l'arquitectura AVR. Aquestes eines inclouen el compilador (de `C`, `C++` i `Ada`), el muntador, l'assemblador i les eines per gestionar binaris habituals (gestor de llibreries, convertidor de formats, etc.). A més cal l'aplicació per transferir programari al microcontrolador.

En aquest curs optarem per la segona opció ja que ofereix millor control del que s'està fent, és consonant amb les eines de desenvolupament d'aplicacions no encastades i permet una millor comprensió dels elements que intervenen en el procés.

Per install·lar aquestes eines en un sistema Linux distribució GNU/Debian o Ubuntu cal instal·lar els següents paquets:

```
$ apt-get install gcc-avr binutils-avr avr-libc avrdude
```

Aquests paquets instal·len les eines que formen la cadena o la suite de desenvolupament estàndard de GNU en versió específica per als microcontroladors d'arquitectura AVR. Entre altres eines s'instal·len les següents:

avr-gcc És el compilador i muntador de `C`. En realitat `avr-gcc` no és el compilador ni el muntador sinó que és un *front-end* que facilita invocar el compilador i el muntador, que realment és l'eina `avr-ld`.

avr-as L'assemblador de l'arquitectura AVR.

avr-objcopy És una aplicació que permet canviar de format els executables. Habitualment l'usarem per canviar el format d'`elf`, el format estàndart de UNIX, a `hex`, el format necessari per implantar el programa en el microcontrolador.

avrdude L'aplicació usada per transferir un programa al microcontrolador.

Els paquets instal·lats inclouen a més la llibreria `avr-libc`, que ofereix algun serveis bàsics per treballar amb els microcontroladors AVR i de la que es parlarà específicament en capítols posteriors. Noteu que s'acaba d'instal·lar un toolchain creuat, ja que s'executa en la estació de treball però genera programes per ser executats en l'AVR. Per aquesta raó les diferents ordres del toolchain tenen noms prefixats per `avr`, com per exemple `avr-gcc`. D'aquesta manera no es confonen amb les ordres homòlogues del toolchain no creuada com ara `gcc`.

3.3 Ús bàsic del toolchain

En aquest apartat es mostra com usar de manera bàsica la cadena d'eines sobre un exemple particular. Aquest exemple té com a objectiu instal·lar en el Arduino un programa que commuti el led de la placa a una freqüència determinada.

```
#include <avr/io.h>
#include <util/delay.h>

int main (void) {
  uint8_t counter;
  DDRB = 0xFF; /* port B en mode output */

  while (1) {
    PORTB = 0xFF;
    for (counter = 0; counter != 5; counter++)
      _delay_loop_2(30000);

    PORTB = 0x00;
    for (counter = 0; counter != 50; counter++)
      _delay_loop_2(30000);
  }
  return 1;
}
```

Programa 3.1: `led.c` encén i apaga el led de l'Arduino

El programa `led.c` de la figura 3.1 és senzill d'entendre: inicialment posa el port B en mode sortida i, a continuació, itera indefinidament tot i modificant la sortida del port B d'alt a baix iterativament.

El procés d'aquest programa usant la cadena de treball segueix el següent patró, que és pràcticament equivalent al que se seguiria en un desenvolupament convencional:

1. Compilació.

   Primer cal compilar el programa `led.c` de la següent forma:

   ```
   $ avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -c led.c
   ```

   Les opcions de compilació usades tenen el següent significat:

   -mmcu=atmega328p Indica al compilador quin és el microcontrolador per al que estem generant el codi.

   -DF_CPL=16000000UL Defineix una constant anomenada `F_CPU` al valor `16000000UL` (noteu que el sufix `UL` significa *unsigned long*). Aquesta constant defineix la velocitat del rellotge a la que funciona la MCU i s'usa per calibrar adequadament les funcions de retard.

   -c Indica a `gcc`, que recordeu és un front-end, la tasca que cal que faci, en aquest cas cal que compili.

   -Os Indica a `gcc` que optimitzi el resultat de la compilació. Dels diversos nivells possibles d'optimització, es demana que usi el nivell `s`, que busca un compromís entre espai i temps escaient per als microcontroladors amb que treballem.

   El resultat és un fitxer objecte anomenat `led.o`.

2. Muntatge.

   El següent pas és el muntatge, que es fa amb la següent ordre:

   ```
   $ avr-gcc -mmcu=atmega328p -o led led.o
   ```

   Observeu que ja no és necessari definir la constant atès que aquesta únicament té efecte durant la fase de preprocés, que forma part de la compilació.

   El resultat d'aquesta fase és un fitxer executable anomenat `led`. Aquest fitxer cal convertir-lo ara al format escaient per a ser implantat en el microcontrolador.

3. Canvi de format.

   El canvi cap al format `hex` es fa de la següent forma:

   ```
   $ avr-objcopy -Oihex led led.hex
   ```

   En aquesta ordre les opcions tenen el següent significat:

   -Oihex Indica que el format de sortida és *Intel HEX*, el format que es requereix per a la fase d'implantació.

   `led.hex` és un fitxer que conté el programa implantable en el microcontrolador. La següent fase és la implantació.

4. Implantació.

   Aquesta fase és la responsable d'implantar el programa en el microcontrolador. Per dur a terme aquesta part cal primer fer uns passos previs destinats a establir la connexió amb el dispositiu Arduino:

   a) Connecteu l'Arduino al vostre computador a través del cable USB. Això té dues conseqüències: la primera alimentar la placa Arduino, ja que aquesta s'alimenta de la tensió proporcionada pel bus USB; la segona establir una comunicació sèrie a través del bus USB entre l'Arduino i el vostre computador.

   b) Determineu amb quin dispositiu el vostre computador identifica el canal sèrie que s'ha establert amb l'Arduino. Això podeu fer-ho si executeu aquesta ordre just després de connectar l'Arduino:

      ```
      $ dmesg
      ```

      Aquesta ordre UNIX us mostrarà per la terminal el log del kernel del sistema[1] en el log del sistema, just després de connectar l'Arduino hi trobareu un missatge similar a aquest cap al final del log:

      ```
      [147392.268111] usb 4−2: new full speed USB device using uhci_hcd and address 3
      [147392.464423] usb 4−2: New USB device found, idVendor=2341, idProduct=0001
      [147392.464431] usb 4−2: New USB device strings: Mfr=1, Product=2, SerialNumber=220
      [147392.464438] usb 4−2: Product: Arduino Uno
      [147392.464443] usb 4−2: Manufacturer: Arduino (www.arduino.cc)
      [147392.464448] usb 4−2: SerialNumber: 64935343133351817022
      [147392.464653] usb 4−2: configuration #1 chosen from 1 choice
      [147393.723925] cdc_acm 4−2:1.0: ttyACM0: USB ACM device
      [147393.726571] usbcore: registered new interface driver cdc_acm
      [147393.726579] cdc_acm: v0.26:USB Abstract Control Model driver for USB modems and ISDN ada
      ```

      Aquest missatge transcriu el procés de connexió del dispositiu USB, fixeu-vos com el vostre sistema operatiu detecta que es tracta d'un Arduino, el seu número de sèrie, etc. En un punt donat el kernel ens diu que aquest dispositiu es presenta com un mòdem sèrie i que li assigna el dispositiu `/dev/ttyACM0`. Aquest dipositiu és important per comunicar-se amb l'Arduino.

   Una vegada fets els passos previs, només resta fer la implantació. Això és pot fer senzillament usant l'ordre següent:

   ```
   avrdude −c arduino −p atmega328p −P /dev/ttyACM0 −U led.hex
   ```

   Les opcions d'aquesta ordre tenen el següent significat:

   -c arduino Indica en quina placa vol fer-se la implantació. Aquesta opció determina la forma en que cal transferir el programa a la placa.

   -p atmega328p Indica el model exacte de microcontrolador al que es transfereix el programa.

[1] El log del kernel del sistema és un registre en el que s'escriu i conserva tot el que el kernel del sistema va fent.

-P /dev/ttyACM0 Indica el dispositivu sèrie a través del que es transmet el programa a implantar. En cada cas ha de ser el dispositiu escaient que, com heu vist abans, cal consultar-lo mitjançant l'ordre `dmesg`.

-U led.hex Indica el programa que es vol implantar.

COMPTE!! És possible que us sigui impossible accedir al dispositiu sèrie `dev/ttyACM0` com a usuari corrent del vostre sistema operatiu. En aquest cas, per evitar executar l'ordre com a usuari `root`, us heu d'afegir prèviament al group *dialout* fent:

```
sudo adduser usuari_corrent dialout
```

seguit d'un *logout* i un *login*.

Després d'implantar el programa en l'Arduino, el vostre programa hauria de començar a executar-se i, per tant, hauríeu de veure el led de la placa fent pampallugues de forma rítmica amb els temps que heu indicat.

3.4 Dissecció del toolchain

En aquest apartat anem a seccionar amb cert detall cadascun dels passos del procés que condueix des del programa font fins a l'executable funcionant en l'Arduino. Com ja hem vist en l'apartat anterior, aquest procés passa per quatre etapes:

1. Compilació.
2. Muntatge.
3. Canvi de format.
4. Implantació.

L'etapa de compilació, que la invoquem amb una sola ordre, és en realitat formada per tres fases que ocorren seqüencialment sense que l'usuari se n'adoni i que s'anomenen:

1. Preprocés.
2. Compilació (en sentit estricte).
3. Assemblatge.

3.4.1 Preprocés

El preprocés és la primera de les fases que té lloc. La duu a terme una aplicació específica anomenada *preprocessador*. En el nostre cas el preprocessador és `avr-cpp`, [Fre10c]. L'entrada a aquesta fase és el programa font en `C` tal i com s'ha escrit. La sortida és un nou programa escrit també en `C` en el que certes parts del programa s'han substituït.

Específicament, les construccions del programa font que comencen amb el símbol `#` són directives del preprocessador. Algunes de les més freqüents són:

#include FITXER En el moment de ser preprocessat, aquesta directiva se substitueix literalment pel fitxer indicat. Té doncs l'efecte d'incloure un fitxer dins d'un altre.

#define NOM VALOR Indica al preprocessador que cada vegada que es trobi en el text del programa font `NOM` ho substitueixi per `VALOR`. Sovint s'usen per definir valors constants.

Podem demanar al front-end `gcc` que faci explícitament el pas de preprocessat sobre el nostre codi font i veure'n els resultats. Si usem l'exemple de la figura 3.1, podem fer-ho usant l'opció `-E` de l'ordre de compilació:

```
$ avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -E led.c
```

```
int main (void) {
  uint8_t counter;

  (*(volatile uint8_t *)((0x04) + 0x20)) = 0xFF;

  while (1) {
    (*(volatile uint8_t *)((0x05) + 0x20)) = 0xFF;
    for (counter = 0; counter != 5; counter++)
      _delay_loop_2(30000);

    (*(volatile uint8_t *)((0x05) + 0x20)) = 0x00;
    for (counter = 0; counter != 50; counter++)
      _delay_loop_2(30000);
  }
  return 1;
}
```

Programa 3.2: Vista parcial del programa de la figura 3.1 una vegada preprocessat.

Obtenim directament en la terminal el resultat. La figura 3.2 mostra la part corresponent a la funció main(). En el programa preprocessat és fàcil observar com la sentència PORTB = 0xFF; s'ha substituït per:

```
(*(volatile uint8_t *)((0x05) + 0x20)) = 0xFF;
```

Noti's com en aquesta sentència s'està accedint a l'adreça de memòria `0x25`, el contingut de la qual considera de tipus **volatile** uint8_t. El port `PORTB`, d'acord amb el datasheet, té l'adreça `0x05` en l'espai d'adreces de `IN/OUT`. Com es sabut, l'espai d'adreces de `IN/OUT` en l'arquitectura AVR es mapeja automàticament sobre l'espai d'adreces de memòria SRAM sumant `0x20`. Així es pot accedir a alguns ports tant a través d'instruccions d'entrada/sortida com mapejats en memòria. En aquest cas, hi accedim a través de la vista mapejada en memòria.

3.4.2 Compilació s.e.

El pas que segueix al preprocés és la compilació *senso estricto*. L'objectiu de la compilació és traduir el llenguatge d'alt nivell, en aquest cas `C` al llenguatge assemblador de la CPU destí, en aquest cas la CPU de l'ATmega328p. Aquest pas es fa de manera transparent en invocar el front-end `avr-gcc`, [Fre10d]. Com succeïa en el preprocés, però, podem demanar al front-end que de forma explícita generi l'assemblador resultat de la compilació usant el flag `-S`. La següent ordre s'encarregarà de preprocessar i compilar l'exemple de la figura 3.1:

```
.file   "led.c"
__zero_reg__ = 1
.text
.global main
.type   main, @function

ldi  r24,lo8(-1)
out 36-32,r24
ldi  r19,lo8(-1)
ldi  r20,lo8(30000)
ldi  r21,hi8(30000)
out 37-32,r19
movw r24,r20
sbiw r24,1
brne 1b
movw r24,r20
sbiw r24,1
brne 1b
movw r24,r20
sbiw r24,1
brne 1b
movw r24,r20
sbiw r24,1
brne 1b
movw r24,r20
sbiw r24,1
brne 1b
out 37-32,__zero_reg__
ldi  r18,lo8(0)
movw r24,r20
sbiw r24,1
brne 1b
subi r18,lo8(-(1))
cpi  r18,lo8(50)
brne .L2
rjmp .L3
.size   main, .-main
```

Programa 3.3: Resultat de compilar el programa de la figura 3.1.

```
$ avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -S led.c
```

El resultat és un fitxer anomenat `led.s` que conté la traducció a assemblador del programa `C`. Compte, aquesta traducció pot canviar en canviar la versió d'`avr-gcc`. El fitxer `led.s` té una gran quantitat de comentaris que relacionen cada part del programa assemblador amb les línies corresponents de l'original. Una vegada net d'aquests comentaris el resultat és el que es mostra a la figura 3.3. En aquest programa s'observen diversos detalls referents a la traducció feta:

- La primera iteració `for` s'ha traduït a base de repetir cinc cops el cos de la iteració. Aquesta és una decissió presa per l'optimitzador del compilador atès que iterava poques vegades, 5 exactament, i ho feia amb un cos de mida petita.

- Els accessos al `PORTB`, que en el font es feien mapejant a memòria, s'han traduït a instruccions `OUT`.

- La sentència `while` s'ha traduït de forma òptima amb una única instrucció `rjmp`.

- Usa etiquetes locals, com ara `1:`. Aquest tipus d'etiquetes és específic del assemblador de la toolchain de GNU i són essencials per poder fer un ús ric de les macroinstruccions. Per a més informació vegeu la referència [Fre10a, apartat 5.3]

La feina feta pel compilador al traduir el codi font és prou bona.

3.4.3 Assemblat

Una vegada traduït a assemblador el programa font, cal assemblar-lo, és a dir traduir l'assemblador al llenguatge màquina corresponent. Aquesta és la feina de l'assemblador, que en la toolchain de GNU s'anomena `avr-as`, [Fre10a]. Per assemblar `led.s` solament cal executar l'ordre següent:

```
$ avr-as -mmcu=atmega328p -o led.o led.s
```

El resultat és el fitxer objecte `led.o`. Un fitxer objecte és un fitxer que conté codi en llenguatge màquina però que no constitueix un programa complet. Més endavant veurem amb més precisió com són els fitxers objecte.

La fase d'assemblatge habitualment es fa de forma transparent a través del front-end `avr-gcc`. Així, quan executem l'ordre de compilació que hem usat inicialment en l'apartat 3.3:

```
$ avr-gcc -mmcu=atmega328p -Os -DF_CPU=16000000UL -c led.c
```

en realitat estem encadenant el preprocés, la compilació *senso estricto* i l'assemblat.

3.4.4 Muntatge o enllaçat

Una vegada s'ha obtingut el fitxer objecte, en el cas que ens ocupa `led.o`, cal muntar-lo per obtenir finalment un programa executable. Aquesta és la feina del muntador o enllaçador que, en el toolchain de GNU s'anomena `avr-ld`. La forma habitual d'invocar-lo és a través del front-end `avr-gcc` amb una ordre com la següent:

```
$ avr-gcc -mmcu=atmega328p -o led led.o
```

El resultat és el fitxer `led`, un fitxer executable. Podem comprovar-ho executant la següent ordre:

```
$ file led
led: ELF 32-bit LSB executable, Atmel AVR 8-bit, version 1 (SYSV), statically linked, not stripped
```

Com es pot veure, el fitxer `led` és un executable en format ELF de 32 bits per a l'arquitectura Atmel AVR de 8 bits.

3.4.5 Canvi de format

El format de l'executable generat en la fase anterior si bé és apropiat per certes tasques no és susceptible de ser implantat en el microcontrolador, especialment per la seva complexitat. En aquesta fase l'objectiu és canviar de format el programa `led` d'ELF al format Intel Hex, molt més senzill i susceptible de ser implantat en el microcontrolador. Per fer el canvi simplement cal emprar la següent ordre:

```
$ avr-objcopy -Oihex led led.hex
```

El resultat és el fitxer `led.hex` que, ara sí, és implantable en el microcontrolador.

3.4.6 Implantació

De la implantació del programa en el microcontrolador ja se n'ha parlat en l'apartat 3.3. Aquesta etapa del procés és la única en la que intervé el maquinari destí ja que per poder fer la implantació cal la seva col·laboració. Hi ha diverses tècniques possibles per a fer la implantació. En el cas que ens ocupa hem usat un `bootloader`, una petita peça de codi que resideix en la memòria de programa del microcontrolador. El bootloader pren el control quan es detecta que es vol fer una implantació i llegeix el programa a través del port sèrie del microcontrolador tot emmagatzemant-lo en la posició escaient de la memòria de programa del propi microcontrolador (observeu les connexions `M8RXD` i `M8TXD` en l'esquema de la figura 2.2 que connecten la circuiteria USB al port sèrie de l'ATMega328p). Aquesta no és una feina senzilla atès que requereix que, al mateix temps que executem un programa emmagatzemat en la memòria flash, hem d'estar escrivint en el mateix banc de memòria. Aquesta característica s'anomena *Read-While-Write Operation*. Una vegada carregat el programa, se li transfereix el flux d'execució.

3.5 Automatització del procés amb make

Tal i com s'ha vist en els anteriors apartats, el treball amb la toolchain de GNU segueix una seqüència de passos cadascun dels quals processa el resultat de l'anterior. Aquest tipus de processos de treball són molt habituals i els sistemes operatius UNIX disposen d'una eina específica per automatitzar-los. Aquesta eina s'anomena `make`, [Fre10e].

El funcionament bàsic de `make` és molt senzill. El cor de l'eina és una petita base de dades en que s'explica l'estructura del projecte en el que `make` ha de treballar. Aquesta base de dades s'emmagatzema habitualment en un fitxer de text de nom `Makefile` i per això en l'argot es coneix com el makefile. En el makefile es descriu com són els passos que cal fer per dur a terme el procés. Aquesta descripció pren la forma de *regles*. Una regla té la següent sintaxi:

```
target: dependencia1 dependencia2 ...
        ordre
```

En una regla, tant el target com les dependències acostumen a ser fitxers. El significat d'una regla és el següent:

> Si la data de modificació del target és més antiga que la d'alguna de les dependències, aleshores cal reconstruir el target executant l'ordre indicada.

Per exemple, en el nostre cas sabem que el fitxer `led.o` depèn del fitxer `led.c` i que el primer s'obté del segon amb el compilador. Això podem expressar-ho amb la següent regla:

```
led.o: led.c
        avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -c led.c
```

Cada pas del procés es pot expressar amb una regla com aquesta. Si les escriviu totes en un fitxer anomenat `Makefile` obtindreu quelcom similar al que es veu la la figura 3.1.

Tingueu en compte alguns detalls molt importants:

1. El fitxer `Makefile` s'ha d'emmagatzemar en el mateix directori en que teniu el fitxer `led.c`.

```
.PHONY: implanta

led.hex: led
    avr-objcopy -Oihex led led.hex

led.o: led.c
    avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -c led.c

led: led.o
    avr-gcc -mmcu=atmega328p -o led led.o

implanta: led.hex
    avrdude -c arduino -p atmega328p -P /dev/ttyACM0 -U led.hex
```

Figura 3.1: Makefile corresponent al projecte del programa `led.c`.

2. A l'escriure les regles cal tenir en compte que l'ordre se separa del principi de línia per *un tabulador* i en cap cas per espais. Aquest detall és una font clàssica de confusions que cal recordar. En cas que escriviu amb `emacs`, cosa recomanable, veureu que l'editor té un mode específic per a makefiles que dóna suport a aquest format.

3. Finalment, el pas que consisteix en implantar el programa en el microcontrolador té una dependència, el programa a implantar, però no produeix cap resultat en forma de fitxer. Per a casos com aquests es defineix un "fals" target (en anglès *phony*). En el cas que ens ocupa definim `implanta` com un target *phony* i l'usem en la regla per implantar el programa en el microcontrolador.

4. L'ordre de les regles en el makefile no és especialment important.

Una vegada definit el makefile, usar `make` és molt simple. `make` s'usa demanant-li exactament què es vol que construeixi. Per exemplificar-ho, assumim que en el directori del projecte únicament hi ha el makefile i el fitxer `led.c`. Aleshores, si es vol obtenir el fitxer `led.hex`, simplement se li demana a `make` fent:

```
$ make led.hex
avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -c led.c
avr-gcc -mmcu=atmega328p -o led led.o
avr-objcopy -Oihex led led.hex
```

`make`, usant la informació del makefile, anirà encadenant comandes una darrera l'altra fins obtenir el fitxer que se li ha demanat.

Per implantar el programa en el microcontrolador simplement cal demanar-li-ho fent:

```
$ make implanta
```

`make` és una eina interessant. Si li demaneu de nou que construeixi el fitxer `led.hex` us comunicarà que ja el teniu actualitzat. Efectivament, com no heu modificat el programa font `led.c`, no és necessari refer el procés. Només per corroborar-ho:

```
$ make led.hex
make: 'led.hex' is up to date.
```

Encara es pot treure una mica més de profit a `make`. Per exemple, el podem fer jugar al nostre favor per tal de que ens simplifiqui la feina de "netejar" el directori de fitxers intermedis. A tal efecte, definim un nou target *phony*: `clean` i una regla associada a aquest target com la següent:

```
.PHONY: clean
clean:
	rm -f *.o *.hex led *.s
```

Amb aquest afegitó podem usar `make` per netejar el directori de treball simplement fent:

```
$ make clean
```

Per últim només fer notar que si invoquem `make` sense dir què volem obtenir, s'assumeix que estem demanat construir el target de la primera regla que es troba en el makefile, en l'exemple que ens ocupa és `led.hex`. Així doncs, si s'executa:

```
$ make
```

en realitat estarem desencadenant tot el procés per obtenir `led.hex`.

3.6 Exercicis

Exercici 3.1 Per què a l'ordre per muntar l'objecte `led.o` no li cal definir la constant `F_CPU`?

Exercici 3.2 Sovint és interessant treballar amb dues comandes per netejar el directori de treball. Una, `clean`, neteja els fitxers més secundaris però preserva el programa `led.hex`. L'altra, `veryclean`, és més agresiva i només deixa en el directori allò imprescindible per reproduir el projecte, en el exemple que ens ocupa els fitxers `led.c` i `Makefile`. Modifiqueu el makefile del projecte per tal d'implementar aquests dos targets.

Exercici 3.3 Afegiu al makefile de l'exemple una nova regla que descrigui com obtenir el fitxer `led.s`, el fitxer en assemblador resultat de compilar `led.c`.

Exercici 3.4 En quins casos és convenient definir la constant `F_CPU` en temps de compilació? Cal fer-ho sigui quin sigui el programa que estem compilant?

Exercici 3.5 Quina és la funció de les directives del preprocessador `#ifdef` i `#ifndef`?

EXERCICI 3.6 Usant com a exemple el programa `led.c` compila'l i genera l'assemblador usant les següents opcions d'optimització: `-O0`, `-O1`, `-O2`, `-O3` i `-Os`. Analitza els 5 programes en assemblador que es generen i compara'ls. Descriu l'efecte de cada opció d'optimització sobre el resultat.

EXERCICI 3.7 Modifica el programa `led.c` tot i substituint les iteracions `for` per iteracions `while`. Observa quines són les diferències en l'assemblador generat pel compilador.

4 Programació a baix nivell amb C

L'objectiu d'aquest capítol és presentar una perspectiva del llenguatge de programació C poc corrent: la del llenguatge usat per programar aplicacions que interaccionen directament amb el maquinari (ports, uart's, etc.). Això és el que es coneix com programació a baix nivell (*low level programming*). En aquestes circumstàncies els modismes emprats difereixen dels usats habitualment en aplicacions d'alt nivell. El capítol usa com arquitectura de referència la de l'AVR però molta de la informació és aplicable a altres contextos.

4.1 Evolució i estandardització del llenguatge

Qui defineix com ha de ser un llenguatge? Quines construccions del llenguatge ha d'acceptar un compilador per tal que pugui ser anomenat "compilador de C"?

Cada llenguatge té una gènesi i un model d'evolució diferent. Molts llenguatges són de natura acadèmica i la seva definició es a càrrec de grups de recerca. Aquest és el cas de llenguatges com `Oberon`. Altres són definits per una comunitat d'usuaris que fixen la seva definició i marquen la seva evolució. `Python` és un d'aquests llenguatges. Es dona també el cas de llenguatges la definició dels quals està en mans d'organismes d'estandardització. `C`, per exemple, és un llenguatge de programació definit per ISO.

Els llenguatges de programació no són estàtics en el temps. Generalment la seva definició va evolucionant a mida que els seus usos i la tecnologia del moment evoluciona. Generalment l'evolució d'un llenguatge conserva la compatibilitat enrere (*backward compatibility*). Aquesta propietat es podria definir així: la versió V_{k+1} d'un llenguatge de programació és compatible enrere si i solament si qualsevol programa P escrit en la versió V_k del llenguatge és correcte i té exactament la mateixa semàntica entès com un programa escrit en la versió V_{k+1} del llenguatge.

L'evolució d'un llenguatge que manté la compatibilitat enrere és fonamental ja que evita haver de reescriure programes antics per actualitzar-ne el llenguatge de programació.

Quan s'escriuen programes en un cert llenguatge és doncs fonamental tenir present per a quina versió específica del llenguatge s'està escrivint el codi i quines conseqüències pot tenir això en el projecte.

Un altre factor distorsionador important pel que fa als llenguatges de programació es troba en les idiosincràsies de cada toolchain. És freqüent que els compiladors de C, per exemple, estenguin el llenguatge de programació afegint noves construccions que, tot i no formar part de l'estàndard, l'autor del compilador creu que poden ser interessants. Aquestes idiosincràsies *cal evitar-les* de totes totes ja que impossibiliten transportar el programa a entorns on sigui necessari treballar amb altres eines.

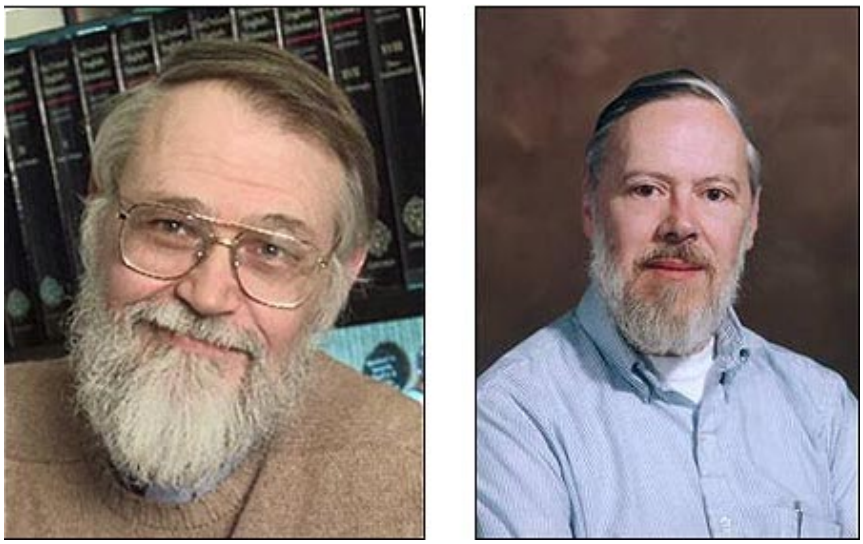


Figura 4.1: Fotografies de B. Kernighan (esquerra) i D. Ritchie (dreta)

Acrònims	Any	Variacions (exemple)
`K&R`	1973	
`ANSI-C`, `C89`, `C90`	1989	prototips funcions, `void*`
`C99`	1999	`complex`, `inline`

Taula 4.1: Evolució dels estàndards del llenguatge `C`.

El llenguatge de programació `C` fou definit per primera vegada per Brian Kernighan i Denis Ritchie als laboratoris Bell, vegeu-los fotografiats a la figura 4.1. Aquesta versió inicial del llenguatge, que sovint es coneix amb l'acrònim de `K&R`, està documentada en el llibre seminal *The C Programming Language*. L'èxit del llenguatge i la seva àmplia acceptació en la indústria va portar a `ANSI` a estandarditzar el llenguatge. Durant el procés d'estandardització incorporaren alguns canvis notables que modificaven detalls obscurs del llenguatge inicial i milloraven la seva semàntica tot i que es preservava la compatibilitat enrere. Alguns elements incorporats de nous són els prototips de funcions o els apuntadors a `void`, per exemple. La nova versió de llenguatge, anomenada `ANSI-C` o també `C89`, fou reflectida en una segona edició del llibre citat anteriorment, [KR88]. L'estàndard `ANSI` amb algunes correccions molt menors fou promogut a estàndard `ISO` el 1990, una any després. Per aquesta raó, a vegades també es coneix la versió del llenguatge com a `C90`. `ISO` va promoure una nova revisió de l'estàndard que finalitzà el 1999 amb la versió de `C` coneguda com a `C99`. La nova versió incorpora elements com les funcions *inline* o el tipus de dades `complex`, per exemple. Aquesta és la darrera versió estàndard del llenguatge fins al moment que, tot i que majoritàriament preserva la compatibilitat enrere, introdueix alguns canvis que poden deixar obsolets alguns programes escrits per a les versions anteriors. La taula 4.1 resumeix aquesta evolució.

4.2 L'ús d'avr-libc

Com s'ha vist al capítol anterior per treballar amb la placa Arduino cal usar un entorn de compilació creuada. Això no només afecta al compilador i per a quina CPU genera codi sinó que també afecta a la llibreria estàndard sobre la que es treballa.

Tot i que més endavant es tractarà extensivament el concepte de llibreria i els mecanismes que hi estan relacionats, en aquest punt és convenient esbossar-ne cinc cèntims.

Una llibreria és un contenidor de trossos de programa (mòduls objecte) compilats prèviament. Cada un d'aquests trossos de programa implementa una funcionalitat concreta. Per exemple, un d'aquests trossos pot implementar una funció per calcular el sinus d'un angle expressat en radians.

Quan escriviu codi `C`, moltes de les crides a funcions que useu ho són a funcions que estan precompilades i emmagatzemades a la llibreria. En alguns casos, fins i tot algunes operacions predefinides del llenguatge, com pot ser suma d'enters, esdevenen crides a llibreries. Efectivament el compilador pot traduir algunes operacions predefinides com a crides a funcions que es troben a les llibreries en comptes d'inserir en el seu lloc el llenguatge màquina corresponent. Això és molt habitual en arquitectures de CPU senzilles que no disposen d'unitats aritmètiques sofisticades.

Aquest és el cas de l'AVR: com no disposa de divisió d'enters, per exemple, quan escriviu un codi com aquest:

```
int a,b;

a = b / 23;
```

la divisió es tradueix per una crida a una funció que resideix a la llibreria. Més endavant, durant la fase de muntatge, el muntador afegirà a l'executable tots els «trossos» de codi de la llibreria necessaris per completar el programa executable.

Aquesta llibreria que acompanya al compilador de `C` es coneix com *llibreria estàndard de C*. Les facilitats que ofereix a l'usuari aquesta llibreria estan definides en l'estàndard de `C`. Al final de [KR88], per exemple, podeu trobar l'especificació de les funcions de la llibreria estàndard de `C`.

En el cas dels computadors molt petits, sovint sense sistema operatiu, és complex implementar tota la funcionalitat que requereix la llibreria estàndard. Aquest és exactament el cas de l'AVR. En aquests casos normalment s'opta per implementar només una part de la llibreria estàndard. Quan treballem amb l'entorn creuat d'`avr-gcc` no disposem de tota la potència de la llibreria estàndard sinó només de la part que implementa la llibreria específica `avr-libc`, la documentació de la qual podeu trobar a [Boe+11]. Noteu com, per exemple, només teniu una versió molt reduïda de les operacions d'entrada/sortida i una versió de poca precisió de les operacions matemàtiques de coma flotant.

A banda de parts de la llibreria estàndard, `avr-libc` també ofereix funcionalitats específiques de la plataforma AVR que faciliten l'accés a funcions específiques d'aquest hardware. A la documentació els headers d'aquests mòduls els trobareu sota el directori `avr`, tal és el cas, per exemple, d'avr/delay.h. Una d'aquestes funcionalitats essencial és la que aporta el mòdul `io`.

Aquest mòdul fa visibles al programa una col·lecció d'objectes (ports, registres, bits específics, etc.) sota el mateix nom exacte amb que es referencien en el datasheet del microcontrolador. Això permet traslladar exactament la informació que indica el datasheet de l'AVR al corresponent codi `C` com s'explicarà en els següents apartats.

4.3 Accés als registres dels perifèrics

L'accés als perifèrics de l'AVR es fa a través de diversos registres que es troben a l'espai d'adreçament de memòria RAM, i.e. són mapejats a memòria. Per usar aquests perifèrics és necessari poder accedir a aquests registres.

L'estratègia per accedir a registres mapejats en memòria des d'un programa `C` passa per usar de forma escaient apuntadors. Seguidament s'exemplifica aquesta estratègia.

Suposem que el registre REG1 al que es vol accedir ocupa l'adreça 0x32 de memòria i és d'un byte. Aleshores, es pot assumir que el tipus del registre és uint8_t i, per tant, la seva adreça és de tipus uint8_t $*$, és a dir un apuntador a un byte. Sent així podem treure profit de la sentència de preprocessador **#define** i escriure:

```
#define REG1 ( * ((uint8_t *) 0x32) )
```

Noti's que estem indicant que REG1 és «la cosa apuntada per un apuntador a byte que val 0x32». Amb aquesta definició és possible escriure en el registre simplement fent:

```
REG1 = 0xff;
```

Afortunadament, el compilador d'AVR amb combinació amb la llibreria `avr-libc` predefineixen tots els registres dels microcontroladors de la família AVR i faciliten enormement accedir-hi. Com s'ha dit abans, ho fan *donant-els-hi el mateix identificador amb que es coneixen a la documentació del xip*. Això comporta que, per accedir al port B, per exemple, es pugui escriure directament:

```
#include <avr/io.h>

PORTB = 0xab;
```

Per a que aquestes definicions siguin les adequades és imprescindible compilar indicant el model de microcontrolador exacte usant l'opció del compilador −mmcu.

4.4 Registres i camps de bits

La majoria de vegades els registres de control dels perifèrics s'han d'interpretar com a vectors de bits en que cada bit o grup de bits tenen un significat particular. EL llenguatge `C` no disposa d'operacions per accedir a bits o grups de bits d'una variable i per tant aquest tipus d'operacions no es poden expressar directament.

Hi ha varies alternatives per accedir als bits individuals d'un registre. La més habitual passa per l'ús de màscares i operacions booleanes bit a bit. Per exemple, el PORTB té 8 pins que poden

configurar-se com a pins d'entrada o de sortida. La configuració de la direcció es fa a través del registre de control DDRB (*Data Direction Register B*). Suposeu que volem configurar la direcció del pin 2 com sortida i després escriure-hi un pols d'un segon de durada. En essència caldria fer:

```
#include <avr/io.h>
#include <avr/delay.h>

DDRB = DDRB | 0b00000100;
PORTB = PORTB | 0b00000100;
_delay_ms(1000);
PORTB = PORTB & ~0b00000100;
```

Noti's com s'usen les màscares i les operacions bit a bit. Noti's també que les operacions *actualitzen* els registres en comptes d'assignar-els-hi un nou valor atès que només es vol modificar un bit en particular.

Una forma alternativa d'escriure el mateix fóra usar les operacions de desplaçament per crear les màscares. Això és:

```
#include <avr/io.h>
#include <avr/delay.h>

DDRB |= 1 << 2;
PORTB |= 1 << 2;
_delay_ms(1000);
PORTB &= ~(1 << 2);
```

Com succeïa en els cas dels noms dels registres, el compilador i la llibreria predefineixen els noms dels bits concrets de cada registre de forma concordant amb el que es fa en el datasheet del microcontrolador. En aquest manual de referència, per exemple, es parla dels bits PORTB2 i DDRB2 [Atm09, pàg. 92]. Aquests símbols estan predefinits de forma que equivalen al número de bit corresponent. Es convenient doncs usar-los per augmentar la legibilitat del codi. El mateix exemple anterior ara es pot reescriure com:

```
#include <avr/io.h>
#include <avr/delay.h>

DDRB |= 1 << DDRB2;
PORTB |= 1 << PORTB2;
_delay_ms(1000);
PORTB &= ~(1 << DDRB2);
```

La construcció de màscares a partir de posicions de bits és tant habitual que existeix una macro predefinida per fer-ho:

```
#include <avr/io.h>
#include <avr/delay.h>

DDRB |= _BV(DDRB2);
PORTB |= _BV(PORTB2);
```

```
_delay_ms(1000);
PORTB &= ~_BV(DDRB2);
```

Naturalment es pot treballar amb màscares que afecten a més d'un bit. Si, per exemple, volem que la direcció dels pins 2,4 i 6 del PORTB sigui de sortida podem configurar-ho escrivint:

```
DDRB |= _BV(DDRB2) | _BV(DDRB4) | _BV(DDRB6);
```

5 Procés de compilació

5.1 La traducció de sentències bàsiques

L'objectiu d'aquest apartat és conèixer una mica millor com es comporta el compilador al traduir les sentències més corrents. la forma de fer-ho serà compilar certs programes simples i estudiar el codi assemblador generat en cada cas.

5.1.1 Primers experiments

El primer exemple que estudiem és el corresponent a la figura 5.1. Compilem aquest programa, anomenat `basiques.c`, i obtenim el seu equivalent en assemblador fent:

```
$ avr-gcc -Os -mmcu=atmega328 -S basiques.c
```

El resultat, contingut en el fitxer `basiques.s` és pot veure a part dreta de la mateixa figura.

Observeu que el compilador s'ha comportat d'una forma sumament intel·ligent, si es pot dir així. Se n'ha adonat que els valors de les variables eren, en realitat, constants i que sumats valien exactament 6. Per tant ha reduït la traducció del main exactament al programa equivalent **return** 6;. Aquesta és la feina que fa precissament l'optimitzador, una de les etapes fonamentals del compilador que hem activat amb la flag `-Os`. Si eviteu usar aquesta flag i repetiu l'experiment veureu com el resultat és molt més matusser.

Fixeu-vos també com la funció retorna el valor 3 en els registres *r25:r24*, dos registres concatenats per que la capacitat del tipus **int** que es retorna és de 2 bytes. Aquest conveni de retorn de valor d'una funció concorda amb el que s'explica en les FAQ del manual de la llibreria `avr-libc`, [Boe+11].

Per tal d'anar coneixent l'optimitzador, mirem ara de compilar un exemple equivalent a l'enterior però més recargolat. Es tracta del programa 5.2. Aquest programa cal compilar-lo usant la comanda:

```
avr-gcc -Os -std=c99 -mmcu=atmega328 -S basiques2.c
```

Noteu que la comanda té una nova flag, la flag `-std=c99`. Aquesta flag indica al compilador que el programa cal interpretar-lo des del punt de vista de la versió `C99` del llenguatge. En aquest cas això és imprescindible atès que definim una variable dins mateix de la construcció **for**() i això només s'admet a partir de la versió `C99`.

Sorprenentment, el resultat en aquest cas és exactament el mateix programa 5.1. L'optimitzador és doncs notablement potent.

```c
#include <inttypes.h>

int main(void) {
  uint8_t a,b,c;

  a = b = c = 0;
  a = a + 4;
  b += 1;
  c++;
  return a+b+c;
}
```

```
1   .file "basiques.c"
2  __SREG__ = 0x3f
3  __SP_H__ = 0x3e
4  __SP_L__ = 0x3d
5  __CCP__ = 0x34
6  __tmp_reg__ = 0
7  __zero_reg__ = 1
8       .text
9  .global main
10      .type main, @function
11 main:
12 /* prologue: function */
13 /* frame size = 0 */
14      ldi r24,lo8(6)
15      ldi r25,hi8(6)
16 /* epilogue start */
17      ret
18      .size main, .-main
```

Programa 5.1: `basiques.c`. Font (esquerra) i assemblador (dreta).

```c
#include <inttypes.h>

int main(void) {
  uint8_t a;

  for (uint8_t i = 0; i < 6; i++)
    a++;
  return a;
}
```

Programa 5.2: `basiques2.c`, programa semànticament equivalent al programa 5.1

5.1.2 El significat de «volatile»

Una hipòtesi fonamental que fa l'optimitzador consisteix en assumir que les dades que emmagatzemen les variables no es modifiquen si no és a través del propi programa. Així, en la línia 7 del programa 5.1 s'asumeix que la variable a val 0 per que a la línia anterior aquest és el valor que se li ha assignat. Si aquesta variable pogués ser modificada de forma *assíncrona* podria passar que durant el temps que va de l'execució de la línia 6 a la 7, la variable a hagués canviat de valor i no es pogues assumir que, en executar la línia 7, a==0.

Unes variables que tenen aquest comportament són aquelles que estàn associades a un port d'entrada del microcontrolador. Un agent extern pot canviar els senyals del port independentment del flux d'execució del programa del microcontrolador i de forma assíncrona.

Quan es dona aquesta circumstància cal advertir a l'optimitzador per que assumeixi que la va-

```
#include <inttypes.h>

int main(void) {
  volatile uint8_t a;
  uint8_t b,c;

  a = b = c = 0;
  a = a + 4;
  b += 1;
  c++;
  return a+b+c;
}
```

```
1  main:
2      push r29
3      push r28
4      push __tmp_reg__
5      in r28,__SP_L__
6      in r29,__SP_H__
7  /* prologue: function */
8  /* frame size = 1 */
9      std Y+1,__zero_reg__
10     ldd r24,Y+1
11     subi r24,lo8(-(4))
12     std Y+1,r24
13     ldd r18,Y+1
14     ldi r19,lo8(0)
15     subi r18,lo8(-(2))
16     sbci r19,hi8(-(2))
17     movw r24,r18
18 /* epilogue start */
19     pop __tmp_reg__
20     pop r28
21     pop r29
22     ret
```

Programa 5.3: `volat.c`. Font (esquerra) i assemblador (dreta).

riable pot canviar el valor al marge de l'execució del programa. Això es fa usant el qualificador **volatile** sobre la variable en qüestió. Observeu el programa 5.3 i el corresponent assemblador.

El programa ha canviat substancialment pel simple fer de declarar **volatile** la variable a. Anem a analitzar el codi i a observar l'efecte d'aquesta declaració.

linies 2–6 En aquestes línies es fixa el bloc d'activació de la funció main(). En essència es preserva a la pila la parella de registres *r29:r28*, que corresponen al registre Y, ja que s'usen en la funció. A la línia 4 es reserva 1 byte a la pila per emmagatzemar la variable a. Les dues línies següents inicialitzen el registre Y amb el valor de l'apuntador de pila.

línia 9 Aquesta instrucció inicialitza la variable a a 0. Noti's que s'accedeix a la variable a, que està emmagatzemada a la pila, usant l'adreçament indirecte més desplaçament via el registre Y i que el registre __zero_reg__ és un registre que sempre conté el valor zero.

línies 10–12 En aquest bloc s'incrementa a amb 4 unitats. Noti's l'efecte del **volatile**: no s'assumeix que a valia 0 sinó que es torna a recuperar el seu valor ja que circumstancialment podria haver canviat.

línies 13–16 En aquest bloc es calcula l'expressió a+b+c. En aquesta expressió, b+c s'ha considerat constant i igual a 2. El valor d'a no s'ha considerat constant per la declaració de volatilitat. El que es calcula doncs és a+2. Com la suma és de 2 bytes, són necessàries dues

```c
#include <inttypes.h>
#include <avr/io.h>

int main (void) {
  uint8_t a;

  /* port B en mode output */
  DDRB = 0xFF;
  /* port C en mode input */
  DDRC = 0x00;
  if (PINC == 0)
    a = 0;
  else
    a = 1;
  PORTB = a;
  return 1;
}
```

```
1  main:
2  /* prologue: function */
3  /* frame size = 0 */
4      ldi r24,lo8(-1)
5      out 36-32,r24
6      out 39-32,__zero_reg__
7      in r24,38-32
8      cpse r24,__zero_reg__
9      ldi r24,lo8(1)
10 .L2:
11     out 37-32,r24
12     ldi r24,lo8(1)
13     ldi r25,hi8(1)
14 /* epilogue start */
15     ret
```

Programa 5.4: `condicional.c`. Codi font (esquerra) i vista parcial del codi assemblador (dreta).

instruccions de byte, la segona d'elles tenint en compte el carry. Noti's que el resultat es deixa en el parell de registres *r19:r18* ja que es vol com a resultat un word.

línia 17 Aquesta instrucció còpia el word resultant del càlcul anterior als registres *r25:r24*, que recordem són els usats per tornar un valor de tipus **int**.

línies 19–22 La resta de línies recuperen l'estat de la pila i retornen del subprograma.

5.1.3 Traducció de sentències condicionals

Observeu el codi corresponent al programa 5.4 i la seva traducció.

Fixeu-vos que la traducció del condicional ha estat, de nou, prou concisa. El gruix es troba entre les línies 7 i 11. En la línia 7 s'obté el valor del port `C` i es desa en el registre *r24*. En la línia 8, es comprova si *r24* val zero i, en cas afirmatiu, s'ignora la següent instrucció. Per tant, en arribar a la línia 11, el registre *r24* val 0 si del port `C` s'ha llegit un 0 i 1 en cas contrari.

També és interessant observar com s'accedeix als diversos registres associats als ports. Així, l'accés al registre `DDRB` es tradueix per `out 36-32, r24`. L'operació $36 - 32$ cal entendre-la en el context de la particular correspondència entre l'espai d'adreces dels ports i l'espai d'adreces dels mateixos ports mapejats en memòria que es dona en l'arquitectura AVR. La correspondència respòn a la fórmula següent:

$$\text{Adreça espai I/O} = \text{Adreça espai data memory} - 32$$

La linia 5 del programa 5.4 s'ha d'entendre doncs com l'accés per la via de l'espai d'adreces d'I/O al port que en l'espai d'adreces de la memòria de dades ocupa l'adreça 36.

```
#include <inttypes.h>
#include <avr/io.h>

int main (void) {
  uint8_t a;

  /* port B en mode output */
  DDRB = 0xFF;
  /* port A en mode input */
  DDRC = 0x00;
  if (PINC == 0)
    a = 0;
  else if (PINC == 12)
    a = 1;
  else
    a = 3;
  PORTB = a;
  return 1;
}
```

```
 1  main:
 2  /* prologue: function */
 3  /* frame size = 0 */
 4      ldi r24,lo8(-1)
 5      out 36-32,r24
 6      out 39-32,__zero_reg__
 7      in r24,38-32
 8      tst r24
 9      breq .L3
10      in r24,38-32
11      cpi r24,lo8(12)
12      breq .L4
13      ldi r24,lo8(3)
14      rjmp .L3
15  .L4:
16      ldi r24,lo8(1)
17  .L3:
18      out 37-32,r24
19      ldi r24,lo8(1)
20      ldi r25,hi8(1)
21  /* epilogue start */
22      ret
```

Programa 5.5: `condicional2.c`. Codi font (esq.) i vista parcial del codi assemblador (dreta).

Compliquem una mica més l'estructura condicional fins a obtenir el programa 5.5 i el corresponent assemblador.

En aquest cas, el gruix del càlcul es produeix entre les línies 7 i 18 del codi assemblador. Si seguim el fil d'execució trobarem que la instrucció de la línia 7 assigna al registre *r24* el valor del port `C`, que en la següent instrucció es comprova si és zero i a la línia 9 se salta a la 18, on s'escriu al port `B`, en cas afirmatiu. Si el valor llegit no fos zero, a la línia 10 es llegeix de nou (compte! això és conseqüència d'assumir que un port és volàtil). Aquesta vegada es compara el valor llegit amb 12 i en cas afirmatiu se salta a la línia 16, cosa que acaba comportant l'escriptura del valor 1 al port `B`. Finalment, si el valor llegit no era 12, es procedeix a partir de la línia 13 i s'acaba escrivint 3 al port `B`.

5.1.4 Traducció de sentències iteratives

Pel que fa a les sentències iteratives, comencem primer per analitzar la traducció d'una sentència que itera indefinidament com la del programa 5.6.

La traducció de la sentència interativa se centra en el rang de línies 6–13. La instrucció de la línia 6 correspon a la inicialització de la variable `a`, que s'emmagatzema sobre el registre *r24*. La línia 8 és un desplaçament esquerra de la variable i a continuació, en cas que el resultat del desplaçament esdevingui 0, a la línia 10 se li assigna de nou el valor 1 a la variable `a`. La

```
#include <inttypes.h>
#include <avr/io.h>

int main (void) {
  uint8_t a;

  /* port B en mode output */
  DDRB = 0xFF;
  a = 1;
  for(;;) {
    a <<= 1;
    if (!a) a = 1;
    PORTB = a;
  }

  return 1;
}
```

```
1  main:
2  /* prologue: function */
3  /* frame size = 0 */
4      ldi r24,lo8(-1)
5      out 36-32,r24
6      ldi r24,lo8(1)
7  .L3:
8      lsl r24
9      brne .L2
10     ldi r24,lo8(1)
11 .L2:
12     out 37-32,r24
13     rjmp .L3
14     .size main, .-main
```

Programa 5.6: `itera1.c`. Font (esquerra) i part del codi assemblador corresponent (dreta).

línia 12 s'ocupa d'escriure el valor d'`a` al port `B`. Finalment la instrucció de la línia 13 tanca la iteració.

El seguent exemple, el programa 5.7 , mostra una iteració finita construida amb una sentència **for**(). Fixeu-vos que la variable local de la iteració, i, es mapeja sobre el registre *r25*. A la línia 7, s'inicialitza aquesta variable a 0. Les línies de la 9 a la 13 són pròpiament la traducció del cos de la iteració. En el bloc, s'escriu al port el byte i es fa el desplaçament de la variable amb el reset si és necessari. Finalment, les línies 14 a 16 implementen la iteració propiament: la línia 14 incrementa la variable i, la línia 15 compara amb el valor 10 i finalment la 16 transfereix el flux al principi del bloc, `L3`, si s'escau. A la figura 5.1 podeu veure el diagrama de flux corresponent a aquest codi assemblador.

5.2 Ús dels registres

5.2.1 Limitacions en l'ús de registresen l'AVR

Per entendre amb claredat com funcionen aquests mecanismes és necessari tenir al cap l'estructura i limitacions del conjunt de registres de l'arquitectura AVR, que no és totalment ortogonal.

Els microcontroladors de la família AVR tenen 32 registres d'un byte amb adreces de la 0 a la 31. El banc de resgistres es pot considerar també amb amplada de 2 bytes. aleshores els registres d'adreça parell constitueixen el LSB i els d'adreça senar el MSB. D'aquesta forma, el primer registre de mida word és *r1:r0*. Els registres de mida word són imprescindibles per encabir apuntadors als diferents bancs de memòria.

```
#include <inttypes.h>
#include <avr/io.h>

#define N 10

int main (void) {
  uint8_t a;

  /* port B en mode output */
  DDRB = 0xFF;
  a = 1;
  for(uint8_t i=0; i<N; i++) {
    PORTB = a;
    a <<= 1;
    if (!a) a = 1;
  }

  return 1;
}
```

```
1  main:
2  /* prologue: function */
3  /* frame size = 0 */
4      ldi r24,lo8(-1)
5      out 36-32,r24
6      ldi r24,lo8(1)
7      ldi r25,lo8(0)
8  .L3:
9      out 37-32,r24
10     lsl r24
11     brne .L2
12     ldi r24,lo8(1)
13 .L2:
14     subi r25,lo8(-(1))
15     cpi r25,lo8(10)
16     brne .L3
```

Programa 5.7: `itera2.c`. Font (esquerra) i part del codi assemblador corresponent (dreta).

Els registres amb adreces de la 16 a la 31 es poden usar amb modes d'adreçament immediat, mentre que la resta no. Així, podem escriure instruccions com **ldi** *r18*, 0 usant aquests registres però no amb els registres que tenen adreces entre la 0 i la 15.

Els tres darres registres de mida word tenen un ús diferenciat atès que poden usar-se en el modes d'adreçament indirecte per accedir a la memòria de dades. Sovint aquests registres es rebatejen com a X (*r27:r26*), Y (*r29:r28)*) i Z (*r31:r30*). Són registres especialment interessants per emmagatzemar apuntadors. D'entre els anteriors, únicament el registre Z pot ser usat com a apuntador a la memòria de programa.

Addicionalment el registre de mida word *r25:r24* pot ser usat en la instrucció d'incrementa word en mode immediat, la qual cosa el fa un bon candidat a emmagatzemar comptadors de 16 bits.

La taula 5.1 resumeix les no ortogonalitats dels registres en la família AVR.

5.2.2 Política d'ús de registres

Els compiladors, quan tradueixen a assemblador, acostumen a establir una política en l'ús dels registres de la CPU. Naturalment aquesta política ha de ser coherent amb les imposicions que el hardware imposa a causa de la manca d'ortogonalitat en l'ús dels registres com s'ha vist a l'apartat anterior. El coneixement d'aquesta política és important quan es fa un ús a baix nivell del llenguatge. En aquest apartat s'explica la política aplicada per `avr-gcc`. És important fer notar que aquesta política pot variar, to i que no és freqüent, en versions diferents del toolchain i, per tant, cal tenir aquesta possibilitat en compte.

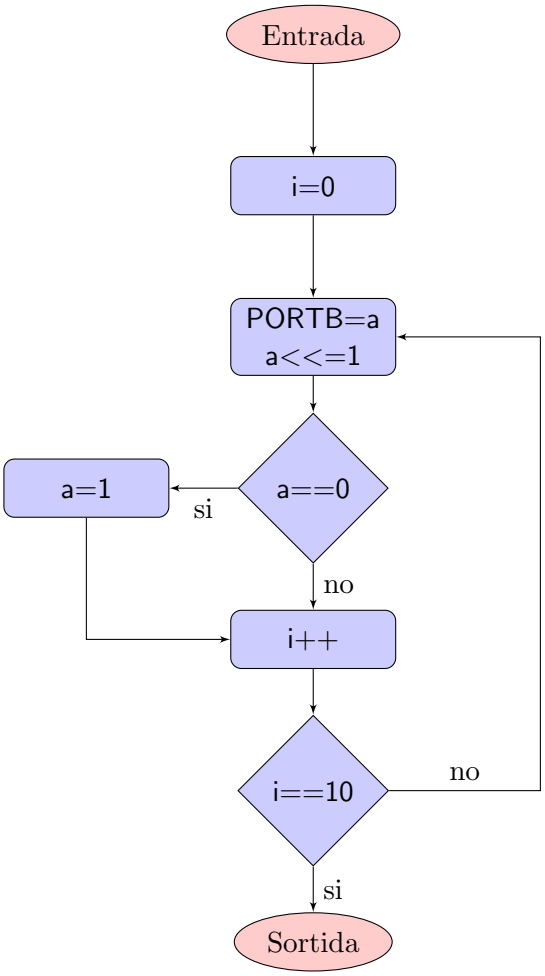


Figura 5.1: Diagrama de flux corresponent a l'assemblador del programa 5.7

El compilador `avr-gcc` fa servir un esquema fixat d'ús dels registres al traduïr el codi `C`. Aquí es reprodueixen les infomacions publicades en la FAQ 13, titulada *What registers are used by the C compiler?*, del manual de la llibreria `avr-libc`, [Boe+11].

Pel que fa a l'ús i la responsabilitat de preservar-ne el contingut durant les crides, els registres segueixen el següent esquema:

r0 Registre temporal a curt termini. Pot usar-se sense haver de conservar-ne el valor anterior (excepte en rutines d'interrupció). En el codi assemblador s'anomena __tmp_reg__.

r1 Sempre s'assumeix que conté el valo 0. Es pot usar com a temporal a curt termini sempre que es restauri a zero posteriorment. Les rutines d'interrupció han de conservar-ne el valor en cas qeu no sigui zero. En el codi assemblador s'anomena __zero_reg__.

r2–r17 Poden usar-se per emmagatzemar variables locals d'un subprograma. Les crides els mantenen inalterats i, per tant, és obligació dels subprogrames cridats preservar-ne el valor fins i tot si són usats per a transferir arguments.

Mode d'adreçament	Espai d'adreçament	Registres
Immediat	—	r16 – r31
Directe	Dades	r0 – r31
Indirecte	Dades	X, Y, Z
Ind. + desplaçament	Dades	Y, Z
Ind. + predecrement	Dades	X, Y, X
Ind. + postincrement	Dades	X, Y, X
Programa indirecte	Programa	Z
Programa ind. + postinc.	Programa	Z

Taula 5.1: Limitacions dels registres en l'arquitectura AVR

r18–r25, r26–r27 (X) S'usen per emmagatzemar variables locals d'un subprograma de manera temporal. Les crides no els preserven i, per tant, si cal preservar-los cal fer-ho abans de la instrucció de crida.

r28–r29 (Y) Poden usar-se per emmagatzemar variables locals d'un subprograma. Les crides els mantenen inalterats i, per tant, és obligació dels subprogrames cridats preservar-ne el valor fins i tot si són usats per a transferir arguments. Si el subprograma té dades locals a la pila, s'usa com a *frame pointer*.

r30–r31 (Z) Es consideren locals i no preservats per les crides. Si és important preservar-los cal fer-ho en el subprograma orígen de la crida.

5.3 Traducció de subprogrames

Els compilador també estableixen al traduïr un protocol determinat per implementar el mecanisme de crida i retorn a un subprograma incloent el pas dels paràmetres. Els elements bàsics que cal tenir en compte són:

1. La implementació del mecanisme de pas de paràmetres.
2. La implementació del concepte de variable local.
3. La implementació del valor de retorn.
4. Les optimitzacions associades a la traducció de subprogrames.

Pel que fa al pas de paràmetres `avr-gcc` els passa per registre i no a través de la pila com és habitual. D'acord amb [Boe+11] els arguments es passen en els registres de mida word *r25:r24* fins *r9:r8* i en aquest ordre. Els arguments que tenen mida senar (per exemple un argument de tipus `uint8_t`) deixen registres per ocupar. En cas que amb aquests registres no sigui suficient o que la funció tingui un nombre variable de paràmetres es recorre a la pila.

Pel que fa a les variables locals aquestes s'implementen sobre registres seguint l'esquema indicat a l'apartat anterior mentre és factible. Quan això no és possible, llavors s'emmagatzemen a la pila com és habitual i s'hi accedeix a través del *frame pointer* `Y`.

Els valors de retorn d'una funció es passen també via registres. En particular, si són d'una byte a *r24*, de dos bytes a *r25:r24* i així successivament.

```c
#include <inttypes.h>
#include <avr/io.h>

uint8_t calc_pattern(uint8_t t) {
  uint8_t r = UINT8_C(0b01010101);
  if (t)
    return r << 1;
  else
    return r;
}


int main () {
  /* port B en mode output */
  DDRB = 0xFF;

  /* itera sobre la taula */
  for(uint8_t i=0;;i ^= 1)
    PORTB = calc_pattern(i);

  return 1;
}
```

```
1  calc_pattern:
2  /* prologue: function */
3  /* frame size = 0 */
4      tst r24
5      brne .L2
6      ldi r24,lo8(85)
7      ret
8  .L2:
9      ldi r24,lo8(-86)
10     ret
11     .size calc_pattern, .-calc_pattern
12 .global main
13     .type main, @function
14 main:
15     push r16
16     push r17
17 /* prologue: function */
18 /* frame size = 0 */
19     ldi r24,lo8(-1)
20     out 36-32,r24
21     ldi r17,lo8(0)
22     ldi r16,lo8(1)
23 .L6:
24     mov r24,r17
25     call calc_pattern
26     out 37-32,r24
27     eor r17,r16
28     rjmp .L6
29     .size main, .-main
```

Programa 5.8: `funcio1.c`. Font (esquerra) i part del codi assemblador corresponent (dreta).

Comprovem exeprimentalment aquesta política a través de l'exemple que podeu veure en el programa 5.8. Aquest programa `C` s'ha compilat amb la comanda següent:

```
$ avr-gcc -Os -std=c99 -mmcu=atmega328 -fno-inline -S funcio1.c
```

i el resultat és el que es pot veure (parcialment) en el codi assemblador de la mateixa figura. En aquest resultat s'aprecien clarament les dues funcions: calc_pattern, que va de les línies 1 a 10 i main que s'estén de les línies 14 a 28.

Pel que fa a calc_pattern, la traducció és força immediata i simplement val la pena fer notar alguns detalls. Primerament cal notar que la variable r ha desaparegut ja que l'optimitzador l'ha transformat en una constant. També és interessant veure que l'operació de desplaçament s'ha calculat en temps de compilació i ja no apareix a l'assemblador. Finalment notar que el registre *r24* s'usa tant pel paràmetre d'entrada com pel valor de retorn, la qual cosa és coherent amb la política vista anteriorment.

```
1  main:
2  /* prologue: function */
3  /* frame size = 0 */
4      ldi r24,lo8(-1)
5      out 36-32,r24
6      ldi r24,lo8(85)
7      ldi r25,lo8(0)
8      ldi r18,lo8(1)
9      rjmp .L9
10 .L7:
11     ldi r24,lo8(-86)
12 .L9:
13     out 37-32,r24
14     eor r25,r18
15     brne .L7
16     ldi r24,lo8(85)
17     rjmp .L9
```

Programa 5.9: `funcio1.s`. Part del codi corresponent a una compilació en que l'optimitzador ha aplicat *inlining*.

Pel que fa a la traducció del programa principal, cal fixar-se en la implementació de la crida a les línies 24 i 25 on és clar el pas del paràmetre a través del registre *r24*. Similarment, a la línia 26 s'observa com es recull el valor de retorn també del registre *r24*.

L'optimitzador es capaç de fer certes feines en el cas dels subprogrames. Una de les més clàssiques és la substitució *inline*. Si el programa 5.8 el compilem amb la comanda habitual (en comptes de la usada prèviament) fent:

```
$ avr-gcc -Os -std=c99 -mmcu=atmega328 -S funcio1.c
```

el resultat és un codi assemblador com el del programa 5.9 . Es pot observar com la crida a la funció ha desaparegut del main i s'ha substituit directament pel càlcul que fa la funció. Això es coneix com a substitució *inline*. Aquest tipus d'optimització afavoreix principalment la velocitat tot i que, en certs casos, pot fer més llarg el programa. La flag del compilador `-fno-inline` evita aquest tipus d'optimització.

5.4 Traducció de l'accés a taules

la traducció de l'accés a taules comporta sovint modes d'adreçament indirectes i apuntadors, atesa la relació entre taules i apuntadors en el llenguatge `C`. En l'arquitectura AVR els apuntadors han de ser 2 bytes de longitud per poder codificar l'espai d'adreces tant de dades com de programa. Així doncs en la traducció a codi assemblador de càlculs sobre taules trobarem amb molta freqüència càlculs sobre words.

El programa 5.10 presenta un exemple amb una petita taula en que primer s'escriu i després llegeix. En aquest exemple hi ha molts elements per comentar:

```c
#include <inttypes.h>
#include <avr/io.h>

static uint8_t t[8];

int main () {
  /* omple la taula */
  uint8_t i=0, v=1;
  do {
    t[i++] = v;
    v <<= 1;
  } while (v);

  /* port B en mode output */
  DDRB = 0xFF;

  /* itera sobre la taula */
  for(uint8_t j=0; j<200; j++)
    PORTB = t[j % 8];

  return 1;
}
```

```
    .file "taules.c"
__SREG__ = 0x3f
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__CCP__ = 0x34
__tmp_reg__ = 0
__zero_reg__ = 1
    .text
.global main
    .type main, @function
main:
/* prologue: function */
/* frame size = 0 */
    ldi r24,lo8(0)
    ldi r25,lo8(1)
.L2:
    mov r30,r24
    ldi r31,lo8(0)
    subi r30,lo8(-(t))
    sbci r31,hi8(-(t))
    st Z,r25
    subi r24,lo8(-(1))
    lsl r25
    cpi r24,lo8(8)
    brne .L2
    ldi r24,lo8(-1)
    out 36-32,r24
    ldi r18,lo8(0)
    ldi r19,hi8(0)
.L3:
    movw r30,r18
    andi r30,lo8(7)
    andi r31,hi8(7)
    subi r30,lo8(-(t))
    sbci r31,hi8(-(t))
    ld r24,Z
    out 37-32,r24
    subi r18,lo8(-(1))
    sbci r19,hi8(-(1))
    cpi r18,200
    cpc r19,__zero_reg__
    brne .L3
    ldi r24,lo8(1)
    ldi r25,hi8(1)
/* epilogue start */
    ret
    .size main, .-main
    .lcomm t,8
.global __do_clear_bss
```

52 Programa 5.10: `taules.c`. Font (esquerra) i codi assemblador corresponent (dreta).

- La reserva de l'espai per a la taula t, que es fa a la línia 48. Noti's també la línia següent en la que es defineix el símbol _do_clear_bss_. Aquesta declaració provocarà que, abans de començar a executar el main s'inicialitzin a zero les cel·les de la taula. Aquest pas es fa ja que l'estàndart de C indica que les variables **static** s'inicialitzen automaticament a zero.

- La primera part del programa abasta de les línies 14 a 25. La variable i es mapeja sobre el registre *r24* i la v sobre *r25*. El registre word *r31:r30*, és a dir Z, s'usarà com apuntador per accedir a la taula. Les línies 17 i 18 l'inicialitzen al valor d'i i a les 19 i 20 se li suma l'adreça base de la taula. Això permet accedir a l'element corresponent a la línia 21. Noti's que les línies 17 – 21 formen un bloc totalment local l'objetiu del qual és accedir a la posició i de la taula.

- Una altra observació interessant és el canvi que ha introduit l'optimitzador respecte a la variable que controla la iteració. En el codi C és la variable v qui controla la iteració. En canvi, l'optimitzador ha triat la variable i per fer-ho!. En efecte, si s'observa la línia 24, on s'implementa el test de la iteració, es veu que és el registre *r24*, que correspon a la i qui intervé. Aquest es compara amb 8 que és el nombre màxim de desplaçaments que poden fer-se sobre v.

- La part que correspon a la segona iteració va de la línia 28 a la 42. La variable de control de la iteració, i, es representa sobre la parella de registres *r19:r18* tot i estar declarada com a uint8_t. La raó és que s'usarà realment com si es tractés d'un apuntador. La operació j%8 s'ha transformat en una operació and, com es pot veure a les línies 32–33.

5.5 Inserció d'assemblador en línia

En algunes ocasions el codi generat pel compilador de C pot no ser el que es requeria i aleshores cal recórrer a l'escriptura de codi assemblador directament. Alguns casos paradigmàtics en que cal fer això són:

- La reescriptura de parts molt crítiques del codi per tal d'optimitzar-les manualment.

- L'escriptura de parts del codi que estan subjectes a restriccions temporals o d'ordre molt específiques.

- L'accés a instruccions que no tenen una correspondència evident amb les construccions de C, com pot ser la instrucció `cli` o `nop` en el cas dels microcontroladors AVR.

Hi ha diverses possibilitats per a fer això. Una d'elles consisteix a encastar directament codi assemblador en mig de codi C. Aquesta no és una funcionalitat estàndart del llenguatge tot i està admesa com una extensió habitual, [Joi07]. Tot i això molts compiladors inclouen aquesta facilitat.

La forma més simple d'usar aquesta facilitat és quan vol inserir-se una instrucció única i sense operands. Per exemple, si es vol posar el microcontrolador en mode SLEEP, existeix una instrucció assemblador específica. Aquesta instrucció es pot afegir amb la següent sentència C:

```
__asm__("sleep");
```

Tot i la simplicitat cal anar amb compte amb aquest tipus de construccions ja que l'optimitzador es pot sentir temptat a eliminar o canviar l'ordre. Les instruccions d'aquests tipus més corrents ja es tenen en compte en la llibreria `avr-libc`, que s'abordarà en capítols posteriors, en forma de macros de preprocessador. Aquest és el cas de sleep_cpu() que està definit en el header `avr/sleep.h` o bé de cli() que està definit en el header `avr/interrupt.h`.

De la mateixa forma que s'insereix una única instrucció, es pot inserir una seqüència. Per exemple, podem inserir 3 `nop` senzillament fent:

```
__asm__("nop"
        "nop"
        "nop"
        );
```

Quan s'insereix assemblador aquest es copia literalment sobre el codi assemblador resultant. Si es necessar que el codi resultant sigui legible cal tabular correctament. L'exemple anterior millorat en aquest aspecte resultaria:

```
__asm__(" nop \n"
        " nop \n"
        " nop \n"
        );
```

El següent nivell de dificultat es dona quan cal inserir un petit càlcul escrit en assemblador. Per exemple un retard de 10 instruccions `nop` executades en una iteració. En aquest cas, i d'acord amb la política d'ús de registres, podem usar el registre temporal __tmp_reg__ com a comptador i inserir el següent codi:

```
__asm__(" ldi __tmp_reg__, lo8(10) \n"
        "1: nop \n"
        " dec __tmp_reg__ \n"
        " brne 1b \n"
        );
```

És interessant notar l'ús d'*etiquetes relatives*, una funcionalitat específica de l'assemblador del toolchain de GNU que facilita en gran manera escriure insercions d'assemblador. En aquest cas, la instrucció `brne 1b` cal interpretar-la com a salta si s'escau a la primera etiqueta `1` que es troba anant enrera. La referència última sobre aquesta funcionalitat es troba a [Fre10a, apartat 5.3].

La dificultat més gran en inserir assemblador es troba quan el codi que s'insereix cal que complementi el codi escrit en `C`. En aquest cas molt sovint cal compartir variables o accedir a paràmetres i valors de retorn. Això implica poder saber quina correspondència ha establert el compilador entre les variables i els registres o posicions de memòria. Aquí s'il·lustrarà a través d'un cas senzill. Si cal més informació s'ha de consultar [Boe+11, *Inline assembler cookbook*] i, en última instància, [Fre10d, apartat 5.37].

Com a exemple prendrem la primera iteració del programa 5.10, que té com a objectiu la inicialització de la taula `t`. Assumirem que es vol la màxima eficiència possible en aquest càlcul i per aquesta raó s'inserirà una implementació directament en assemblador com en el programa 5.11.

```c
#include <inttypes.h>
#include <avr/io.h>

static uint8_t t[8];

int main () {
  /* omple la taula */
  uint8_t v=1;

  __asm__("\n"
      " ldi r30, lo8(%0) \n"
      " ldi r31, hi8(%0) \n"
      "1: \n"
      " st Z+, %1 \n"
      " lsl %1 \n"
      " brne 1b \n"
      :
      : "i" (t), "r" (v)
      : "r30", "r31"
          );

  /* port B en mode output */
  DDRB = 0xFF;

  /* itera sobre la taula */
  for(uint8_t j=0; j<200; j++)
    PORTB = t[j % 8];

  return 1;
}
```

```
1  main:
2  /* prologue: function */
3  /* frame size = 0 */
4      ldi r24,lo8(1)
5  /* #APP */
6   ; 10 "taules2.c" 1
7
8      ldi r30, lo8(t)
9      ldi r31, hi8(t)
10 1:
11     st Z+, r24
12     lsl r24
13     brne 1b
14
15  ; 0 "" 2
16 /* #NOAPP */
```

Programa 5.11: `taules2.c`. Font (esquerra) i part del codi assemblador corresponent (dreta).

En aquest programa l'assemblador conté elements sintàctics com `%0` que seràn substituïts automàticament pel compilador pels símbols corresponents.

5.6 Make: segon atac

En la definició d'un `Makefile` associat a un projecte és habitual usar variables. Les variables de `make` sempre s'avalúen a cadenes de caracters i típicament tenen identificadors formats per lletres majúscules. Usant amb intel·ligència les variables podem generalitzar més còmodament un `Makefile`.

Considereu el següent `Makefile`:

```
GCC=avr-gcc
```

```
OPTIONS=-Os -std=c99 -mmcu=atmega328 -S

basiques.s: basiques.c
    $GCC $OPTIONS basiques.c
```

Noteu com per obtenir el valor d'una variable s'usa el seu identificador precedit de dòlar com a `$GCC`. A vegades, també és convenient envolar el nom de la variable de parèntesis com, per exemple, a `$(GCC)`.

Les variables es poden composar a base d'altres variables. El següent exemple il·lustra aquesta praxis:

```
GCC=avr-gcc -mmcu=atmega328
OPTIONS=-Os -std=c99
COMPILE=$GCC -S $OPTIONS

basiques.s: basiques.c
    $(COMPILE) basiques.c
```

`make` també admet un tipus de variables especials que s'anomenen *automàtiques*. Aquestes variables tenen la particularitat de que el seu valor s'assigna de forma automàtica segons certes condicions.

Per exemple, la variable `$<`, té un valor que correspon amb la primera dependència de la regla on es troba. Per exemple, l'exemple anterior es podria haver escrit com:

```
GCC=avr-gcc -mmcu=atmega328
OPTIONS=-Os -std=c99
COMPILE=$GCC -S $OPTIONS

basiques.s: basiques.c
    $(COMPILE) $<
```

En aquest cas, la variable automàtica valdria `basiques.c`. Si incorporem la variable a la definició de `COMPILE`, podem reescriure el `Makefile` de forma més general, fins i tot si incorporem més fitxers en el projecte, fent:

```
GCC=avr-gcc -mmcu=atmega328
OPTIONS=-Os -std=c99
COMPILE=$GCC -S $OPTIONS $<

basiques.s: basiques.c
    $(COMPILE)

codi1.s: codi1.c
    $(COMPILE)

codi2.s: codi2.c
    $(COMPILE)
```

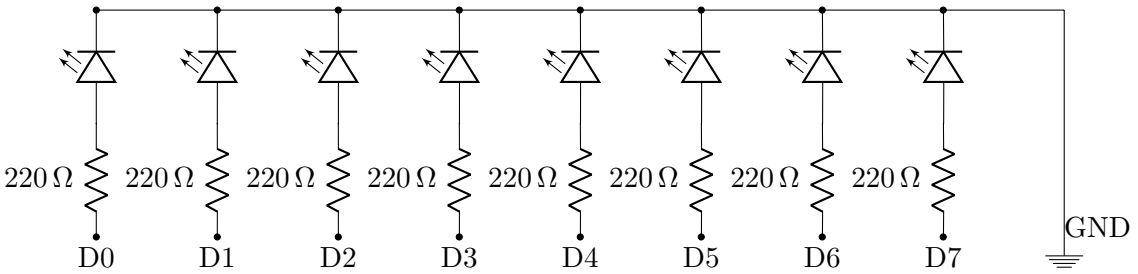
5.7 Exercicis

Exercici 5.1 Dibuixeu el diagrama de flux corresponent al programa 5.9.

Exercici 5.2 Quan el compilador ha aplicat *inlining* al programa 5.8 s'ha produït un efecte curiós. La crida des de main a calc_pattern s'ha substituït pel cos de la funció directament. Això no obstant, la traducció de la funció continua estant en el codi assemblador (comproveu-ho!). D'altra banda, si en el codi C precediu la capçalera de la funció calc_pattern de la paraula clau **static** i recompileu de nou es fa *inlining* i la funció desapareix del codi assemblador (comproveu-ho!). Per quina raó passa això?

Exercici 5.3 Si volguéssim que la taula t del programa 5.10 no s'inicialitzés a zero abans d'arrencar el programa, quina podria ser la solució? Pista: Repasseu les FAQ's de [Boe+11].

Exercici 5.4 Implanteu el programa 5.6. A tal efecte useu 8 leds connectats al port B de la placa Arduino segons l'esquema que trobareu a continuació:



Quan l'executeu, què observeu? Com podríeu millorar el programa per veure una rotació més clara?

Exercici 5.5 Modifiqueu pe programa 5.7 per tal que la variable que fa de comptador decreixi en comptes de créixer. Compileu-lo i estudieu com a canviat l'assemblador respecte l'original.

Exercici 5.6 Estudieu com tradueix el compilador el càlcul de les següents expressions. Assumiu que les variables que hi surten són totes de tipus uint8_t.

1. a + a + 3 ∗ b
2. (a + 3) ∗ (a + 3)
3. a + b / 2 − a
4. a % 2 == 0

Exercici 5.7 Seguint la mateixa estratègia que hem usat per a les sentències bàsiques, exploreu les característiques de la traducció de la sentència **switch**(). Comproveu què succeeix amb pocs casos, amb molts casos, quan els valors dels casos són consecutius, etc.

Exercici 5.8 Els dos programes següents semblen equivalents però no ho són. Escriviu-los, compileu-los, compareu el codi assemblador que heu obtingut i, finalment, determineu on rau la diferència exactament. En quin cas un i altre podrien donar resultats diferents?

```
#include <inttypes.h>
#include <avr/io.h>

int main (void) {
  uint8_t a;

  /* port B en mode output */
  DDRB = 0xFF;
  /* port A en mode input */
  DDRC = 0x00;
  if (PINC == 0)
    a = 0;
  else if (PINC == 12)
    a = 1;
  else
    a = 3;
  PORTB = a;
  return 1;
}
```

```
#include <inttypes.h>
#include <avr/io.h>

int main (void) {
  uint8_t a,b;

  /* port B en mode output */
  DDRB = 0xFF;
  /* port A en mode input */
  DDRC = 0x00;
  b = PINC;
  if (b == 0)
    a = 0;
  else if (b == 12)
    a = 1;
  else
    a = 3;
  PORTB = a;
  return 1;
}
```

Exercici 5.9 Castigueu una mica més l'optimitzador. A tal efecte construïu dos nous programes equivalents al programa 5.1. El primer ha d'usar una iteració **while**() en comptes d'una iteració **for**(). El segon ha d'utilitzar una iteració **for**() imbricada dins d'una altra. Genereu l'assemblador corresponent als dos casos i determineu com s'ha comportat l'optimitzador.

Exercici 5.10 A la vista de la implementació del pas de paràmetres que fa el compilador `avr-gcc` creus que és possible fer crides recursives usant aquest compilador? Per què?

Exercici 5.11 La versió 2.6 de Python és compatible enrere amb la 2.5. Cerca quines són les diferències entre una i altra versió del llenguatge. Tria'n una i explica per què aquesta diferència és compatible enrere.

EXERCICI 5.12 La versió 3.1 de Python no és compatible enrere amb la 2.5. Cerca quines són les diferències entre una i altra versió del llenguatge. Tria'n una i explica per què aquesta diferència no és compatible enrere.

EXERCICI 5.13 Escriu un petit programa en `C` que mostri la incompatibilitat enrere de `C99`. A tal efecte cal que el programa sigui correcte en `C90` però no ho sigui en `C99`.

6 La fase de muntatge o d'enllaçat

L'objectiu d'aquest capítol és aprofundir en la fase d'enllaçat. L'enllaçat és una fase de gran importància quan estem programant a baix nivell. El coneixement dels aspectes tècnics més rellevants obre la porta a nombrosos recursos importants per aquest àmbit de treball.

Com s'ha vista al capítol 3, el resultat de compilar un mòdul de `C` és un fitxer objecte. De fet, sigui quin sigui el llenguatge de programació en què s'escriuen els fonts, els compiladors generen un fitxer objecte, fins i tot quan el llenguatge font és assemblador. Un fitxer objecte, però, no és un "producte final" que pugui ser executat. Per que pugui ser considerat un programa *executable*, cal aplicar-hi prèviament un conjunt de transformacions. Per entendre aquesta tecnologia és convenient començar per comprendre l'estructura d'un fitxer objecte i quina és la relació amb la compilació. A continuació és interessant entendre què fa un muntador i com transforma el fitxer objecte. Finalment és interessant entendre com podem treure profit d'aquest procés per exemple treballant amb més d'un mòdul, integrant mòduls escrits en llenguatges diferents, alterant el mòdul d'inicialització o fent ús de llibreries.

6.1 Primera aproximació a la compilació i enllaçat

Considereu el programa 6.1 i la corresponent traducció a assemblador.

Si s'executa l'assemblador i s'assembla el fitxer `modul1.s` que ha generat el compilador s'obté el corresponent fitxer objecte `modul1.o`. Els objectes són fitxers binaris amb un format específic anomenat ELF, [Lab95; Gro10]. Usant la comanda `avr-objdump` es pot extreure el contingut d'un fitxer objecte. Per exemple, la següent comanda extreu el codi màquina que conté el fitxer objecte corresponent a `modul1.o` i genera el llistat 6.1:

```
$ avr-objdump -d modul1.o
```

En aquest llistat s'observen diverses coses que permeten anar perfilant algunes de les funcions que ha de fer la fase de montatge:

- A la línia 15, la instrucció de salt té per opcode `00 c0`. Com es pot apreciar el desplaçament no s'ha instanciat. El mateix succeeix amb el la crida a setup_led de la línia 25. Efectivament, aquests valors no completats depenen de símbols definits en l'assemblador. Per exemple, la línia 15 depèn del valor del símbol commuta_led i la línia 25 depèn del valor del símbol setup_led. En els mòduls objecte els símbols no es *resolen* sinó que es deixen per completar. Una de les tasques del muntador és *resoldre els símbols* pendents de resoldre.
- El programa en codi màquina, vegeu la línia 8, comença a l'adreça 0. Això no obstant, quan el programa s'ubica en la memòria del computador, no ho fa necessàriament a partir de l'adreça 0. El programa resultant, doncs, ha de ser *reublicable* per tal qe pugui grabar-se a partir de la posició de memòria més escaient.

```c
#include <inttypes.h>
#include <avr/io.h>

#define LED UINT8_C(1<<5)

/* eines gestio del led */
void setup_led(void) {
  DDRB = 0xFF;
}

void commuta_led(void) {
  uint8_t pb = PORTB;

  PORTB = (pb & LED) ?
          (pb | LED) :
          (pb & ~LED);
}



int main () {
  setup_led();

  for(;;) {
    commuta_led();
  }

  return 1;
}
```

```
1  .global setup_led
2      .type setup_led, @function
3  setup_led:
4  /* prologue: function */
5  /* frame size = 0 */
6      ldi r24,lo8(-1)
7      out 36-32,r24
8  /* epilogue start */
9      ret
10     .size setup_led, .-setup_led
11 .global commuta_led
12     .type commuta_led, @function
13 commuta_led:
14 /* prologue: function */
15 /* frame size = 0 */
16     in r24,37-32
17     sbrs r24,5
18     rjmp .L4
19     ori r24,lo8(32)
20     rjmp .L5
21 .L4:
22     andi r24,lo8(-33)
23 .L5:
24     out 37-32,r24
25 /* epilogue start */
26     ret
27     .size commuta_led, .-commuta_led
28 .global main
29     .type main, @function
30 main:
31 /* prologue: function */
32 /* frame size = 0 */
33     ldi r24,lo8(-1)
34     out 36-32,r24
35 .L8:
36     call commuta_led
37     rjmp .L8
38     .size main, .-main
```

Programa 6.1: `modul1.c`. Codi font (esquerra) i vista parcial del codi assemblador (dreta).

```
1
2  modul1.o:     file format elf32-avr
3
4
5  Disassembly of section .text:
6
7  00000000 <setup_led>:
8     0:   8f ef           ldi     r24, 0xFF       ; 255
9     2:   84 b9           out     0x04, r24       ; 4
10    4:   08 95           ret
11
12 00000006 <commuta_led>:
13    6:   85 b1           in      r24, 0x05       ; 5
14    8:   85 ff           sbrs    r24, 5
15    a:   00 c0           rjmp    .+0
   ; 0xc <commuta_led+0x6>
16    c:   80 62           ori     r24, 0x20       ; 32
17    e:   00 c0           rjmp    .+0
   ; 0x10 <commuta_led+0xa>
18   10:   8f 7d           andi    r24, 0xDF       ; 223
19   12:   85 b9           out     0x05, r24       ; 5
20   14:   08 95           ret
21
22 00000016 <main>:
23   16:   8f ef           ldi     r24, 0xFF       ; 255
24   18:   84 b9           out     0x04, r24       ; 4
25   1a:   0e 94 00 00     call    0       ; 0x0 <setup_led>
26   1e:   00 c0           rjmp    .+0             ; 0x20 <main+0xa>
```

Llistat 6.1: Resultat de la comanda `avr-objdump -d modul1.o`

- El codi màquina del llistat no inicialitza alguns elements del maquinari imprescindibles per poder executar un programa. Per exemple, el registre de pila o la taula d'interrupcions. Això implica que no és pot considerar un programa complet a punt de ser executat. No és un executable. Una altra de les missions del muntador serà completar aquest codi amb les parts que manquen per acabar essent un programa executable.

Així doncs, el resultat de compilació+assemblat és un mòdul o fitxer *objecte*[1]. Un objecte[2] conté el programa traduït a llenguatge màquina però amb certes "mancances". Una pregunta natural és per què el resultat de la compilació+assemblat no és directament un programa executable. La resposta és senzilla: és més interessant que no ho sigui. A mida que avancem capítols aquesta raó serà més evident.

Si muntem l'objecte `modul1.o` i obtenim l'executable `modul1`, podem desassemblar el seu contingut usant la mateixa comanda que hem usat abans:

```
$ avr-objdump -d modul1
```

El resultat, que podeu veure en el llistat 6.2 és ara notablement diferent:

- Ha aparegut nou codi que no existia en l'objecte. Per exemple, entre les línies 3 i 16 hi ha el codi que inicialitza la taula d'interrupcions. O al final del codi, a la línia 56, s'hi ha afegit una iteració infinita que s'executa una vegada acabat el programa.

- Els símbols no resolts s'han resolt i ara tenen valors específics. Per exemple, la instrucció `call` de la línia 49 ara té com a argument l'adreça `0x32`, que correspon a la subrutina commuta_led.

El muntador és doncs el programa encarregat de:

- Resoldre els símbols no resolts que conté el mòdul objecte.

- Completar el mòdul objecte amb el codi precompilat que sigui convenient.

- Finalment obtenir un programa executable que sigui auto-contingut.

6.2 El mòdul objecte

L'objectiu d'aquest apartat és aprendre quina és l'estructura d'un mòdul objecte. Per tal d'entendre-ho, cal aclarir abans alguns conceptes que apareixen en el codi assemblador i també, de forma anàloga, en els llenguatges d'alt nivell.

[1] Quan es parla de mòdul objecte o de mòdul executable no s'està fent referència al mateix concepte que quan es parla de mòdul `Python`.

[2] Quan parlem d'objecte en aquest context fem referència a un fitxer o mòdul objecte, no al concepte d'objecte referit a la programació orientada a objectes.

```
1  Disassembly of section .text:
2
3  00000000 <__vectors>:
4     0:   0c c0           rjmp    .+24
   ; 0x1a <__ctors_end>
5     2:   13 c0           rjmp    .+38
   ; 0x2a <__bad_interrupt>
6     4:   12 c0           rjmp    .+36
   ; 0x2a <__bad_interrupt>
7     6:   11 c0           rjmp    .+34
   ; 0x2a <__bad_interrupt>
8     8:   10 c0           rjmp    .+32
   ; 0x2a <__bad_interrupt>
9     a:   0f c0           rjmp    .+30
   ; 0x2a <__bad_interrupt>
10    c:   0e c0           rjmp    .+28
   ; 0x2a <__bad_interrupt>
11    e:   0d c0           rjmp    .+26
   ; 0x2a <__bad_interrupt>
12   10:   0c c0           rjmp    .+24
   ; 0x2a <__bad_interrupt>
13   12:   0b c0           rjmp    .+22
   ; 0x2a <__bad_interrupt>
14   14:   0a c0           rjmp    .+20
   ; 0x2a <__bad_interrupt>
15   16:   09 c0           rjmp    .+18
   ; 0x2a <__bad_interrupt>
16   18:   08 c0           rjmp    .+16
   ; 0x2a <__bad_interrupt>
17
18 0000001a <__ctors_end>:
19   1a:   11 24           eor     r1, r1
20   1c:   1f be           out     0x3f, r1        ; 63
21   1e:   cf e5           ldi     r28, 0x5F       ; 95
22   20:   d2 e0           ldi     r29, 0x02       ; 2
23   22:   de bf           out     0x3e, r29       ; 62
24   24:   cd bf           out     0x3d, r28       ; 61
25   26:   0d d0           rcall   .+26            ; 0x42 <main>
26   28:   11 c0           rjmp    .+34            ; 0x4c <_exit>
27
28 0000002a <__bad_interrupt>:
29   2a:   ea cf           rjmp    .-44            ; 0x0 <__vectors>
30
31 0000002c <setup_led>:
32   2c:   8f ef           ldi     r24, 0xFF       ; 255
33   2e:   84 b9           out     0x04, r24       ; 4
34   30:   08 95           ret
35
36 00000032 <commuta_led>:
37   32:   85 b1           in      r24, 0x05       ; 5
38   34:   85 ff           sbrs    r24, 5
39   36:   02 c0           rjmp    .+4
   ; 0x3c <__CCP__+0x8>
40   38:   80 62           ori     r24, 0x20       ; 32
41   3a:   01 c0           rjmp    .+2             ; 0x3e <__SP_H__>
42   3c:   8f 7d           andi    r24, 0xDF       ; 223
43   3e:   85 b9           out     0x05, r24       ; 5
```

6.2.1 Símbols

Els programes escrits en assemblador és molt habitual que continguin *símbols*. Els símbols poden jugar diverses funcions:

- *Etiquetes* que denoten una posició de memòria (l'adreça d'un subprograma, d'una variable, etc.).
- *Constants* que denoten un valor numèric.

El el programa 6.1, per exemple, els símbols commuta_led a la línia 13 o .L4 a la línia 21 són etiquetes i, com a tals, denoten una adreça en la memòria del programa. D'altra banda, en el programa 5.10 símbols com __CPP__ són constants i, com a tals, denoten valors numèrics. En aquest cas __CPP__ denota l'enter 0x34.

Tots els símbols es caracteritzen per:

- Tenir un *identificador*.
- Poder tenir un *valor* associat, ja sigui una adreça de memòria o un valor numèric. En alguns casos els símbols no tenen un valor definit. Llavors es diu que són *símbols indefinits*.
- Tenir un conjunt d'*atributs* associats. Cada atribut indica alguna propietat específica del símbol. Per exemple, un atribut pot indicar si el símbol és local o bé, en cas de ser una etiqueta, a quin tipus d'entitat es refereix.

6.2.2 Seccions

Una secció és un bloc d'informació, instruccions o dades, que ocupa un rang continu d'adreces. Tota la informació en una secció es tractada de manera uniforme durant el muntatge. Per exemple, una secció pot considerar-se de "només lectura", o bé de dades constants, etc. Quan el muntador fusiona codi màquina de diverses fonts fins obtenir un programa executable sempre ho fa movent seccions. En cap cas desfà el contingut d'una secció. Les seccions doncs són les unitats atòmiques amb que treballa un muntador.

Les seccions:

- Tenen un *identificador*. L'identificador no és arbitrari i cada toolchain/arquitectura defineix els identificadors admesos.
- Tenen un conjunt d'atributs que indiquen propietats de la secció. Per exemple si és una secció escrivible o només de lectura.
- Sovint es defineixen de forma automàtica símbols que referencien les seccions i en codifiquen algunes característiques.

Les seccions estàndard que troben en la majoria d'arquitectures són les següents:

.bss Conté dades no inicialitzades que el sistema inicialitzarà a zero. Habitualment s'usa per contenir variables globals. L'espai dedicat a aquesta secció no acabarà formant part de la imatge executable atès que no conté cap informació significativa més enllà de l'espai ocupat. El nom és l'acrònim de *Block Started by Symbol*.

.data Conté dades inicialitzades. Habitualment variables amb un valor predefinit o dades constants. Acabarà formant part de la imatge executable.

.text Codi del programa. Habitualment és de només lectura i acabarà formant part del programa.

Noteu que, en el cas de l'AVR, en tractar-se d'una arquitectura Harvard, els diferents segments no aniran a parar al mateix banc de memòria: el segment `.text`, per exemple, anirà a parar a la memòria flash mentre que el segment `.bss`, per exemple, anirà a la RAM.

A banda de les seccions estàndard poden existir altre seccions. En el cas de l'arquitectura AVR, per exemple, la secció `.eeprom` s'usa per marcar les dades que es volen emmagatzemar en la EPROM del microcontrolador. Més endavant aniran sortint seccions específiques de l'arquitectura AVR i el seu ús.

6.2.3 L'estructura dels objectes

Un mòdul objecte no és altra cosa que un fitxer binari que conté el resultat de l'assemblador. Aquest resultat està estructurat en tres parts:

taula de símbols La taula de símbols recull els símbols que contenen les diferents seccions que conté l'objecte així com les seves propietats i, en particular, el seu valor.

taula de reubicacions La taula de reubicacions recull la llista d'adreces de les seccions de codi en que hi ha instruccions que cal reubicar així com el tipus de reubicació que cal practicar.

seccions Finalment, el mòdul objecte conté una col·lecció de seccions que constitueixen l'objecte en qüestió.

La manera precisa en que es codifiquen i emmagatzemes cadascuna de les parts està determinada pel format del fitxer. En el cas de la toolchain de GNU que s'usa en aquest document el format és *ELF* i, específicament, *elf32-avr*. És a dir la versió d'ELF per a 32 bits i arquitectures AVR. La especificació genèrica del format ELF la podeu llegir a [Lab95]. També podeu consultar la pàgina de `man` corresponent executant l'ordre:

```
$ man 5 elf
```

En els següents apartats analitzarem amb més detall aquesta estructura i la seva raó de ser.

6.2.4 La taula de símbols

Considereu el mòdul objecte `modul1.o`. Podem conèixer la seva taula de símbols usant la següent ordre:

```
$ avr-objdump -t modul1.o
```

El resultat és una taula com la del llistat 6.3. Aquesta taula té cinc columnes:

1. El valor del símbol. Sovint és una adreça.
2. Els atributs del valor, que s'indiquen amb un conjunt de lletres. Algunes de les més rellevants són les següents:

```
1
2 modul1.o:     file format elf32-avr
3
4 SYMBOL TABLE:
5 00000000 l    df *ABS*  00000000 modul1.c
6 00000000 l    d  .text  00000000 .text
7 00000000 l    d  .data  00000000 .data
8 00000000 l    d  .bss   00000000 .bss
9 0000003f l       *ABS*  00000000 __SREG__
10 0000003e l       *ABS*  00000000 __SP_H__
11 0000003d l       *ABS*  00000000 __SP_L__
12 00000034 l       *ABS*  00000000 __CCP__
13 00000000 l       *ABS*  00000000 __tmp_reg__
14 00000001 l       *ABS*  00000000 __zero_reg__
15 00000000 g     F .text  00000006 setup_led
16 00000006 g     F .text  00000010 commuta_led
17 00000016 g     F .text  0000000a main
```

Llistat 6.3: Resultat de l'ordre `avr-objdump -t modul1.o`

l,g Diuen si el símbol és local o global.

F Indica que el símbol és una funció.

La codificació completa dels atributs es pot consultar a la pàgina man d'`avr-objdump`.

3. La secció en que es defineix el símbol. En aquest cas, la secció `*ABS*` indica que el símbol és una constant i no una etiqueta. També podem trobar-nos amb la secció `*UND*`, que indica que el símbol es referencia en aquest mòdul però no es defineix aquí.
4. Usada per a diversos motius, generalment conté una mida associada al símbol. Per exemple, en el cas que el símbol es refereixi a una funció, conté la llargada en bytes de la funció. Recordi's com l'assemblador generat per gcc insereix al final de les funcions una part de codi que calcula la llargada de la funció l'assigna a l'atribut corresponent de l'etiqueta de la funció. Això és el que es fa a la línia 27 del programa 6.1, per exemple.
5. L'identificador del símbol.

Si ens fixem en la taula de símbols del llistat 6.3, veurem per exemple que el símbol __SP_H__ és una constant i val 0x3e o que el símbol commuta_led correspon a una funció global que ocupa 0x10 bytes i es refereix a l'adreça 0x6. La taula també conté alguns símbols que no pertanyen pròpiament al les seccions, com per exemple el símbol modul1.c, que s'usa per emmagatzemar el nom del programa font que ha general el objecte.

És interessant adonar-se de que algunes etiquetes presents en el programa assemblador no han esdevingut símbols. Aquest és el cas de les etiquetes .L4 o .L5 del programa 6.1. Aquestes etiquetes són considerades per l'assemblador com a *símbols amb identificadors locals* i no es transfereixen a l'objecte resultant. Habitualment corresponen a adreces internes de les funcions i s'usen per implementar les sentencies de control de flux del llenguatge de programació.

El concepte d'etiqueta local és confús atès que s'aplica amb significats subtilment diferents:

```
1
2 modul1.o:     file format elf32-avr
3
4 RELOCATION RECORDS FOR [.text]:
5 OFFSET   TYPE              VALUE
6 0000000a R_AVR_13_PCREL    .text+0x00000010
7 0000000e R_AVR_13_PCREL    .text+0x00000012
8 0000001a R_AVR_CALL        .text+0x00000006
9 0000001e R_AVR_13_PCREL    .text+0x0000001a
```

Llistat 6.4: Taula de reubicacions de `modul1.o`. Resultat de l'ordre `avr-objdump -r modul1.o`

1. En el context d'un programa assemblador que una etiqueta sigui un símbol amb identificador local significa que no es transferirà a l'objecte. En el cas de l'assemblador de l'AVR, això correspon de forma naturals als identificadors de símbols que comencen amb el prefix `.L`. Aquests identificadors no s'enregistren a la taula de símbols de l'objecte corresponent.

2. En el context d'un mòdul objecte es parla d'etiquetes locals per referir-se a etiquetes l'ús de les quals el muntador només pot circumscriure al propi mòdul que les conté. Aquesta "altra" propietat de localitat de les etiquetes es declara explícitament en el codi assemblador: si no s'indica res, l'etiqueta és local; si es qualifica amb la directiva .global és global. Aquestes etiquetes sempre surten a la taula de símbols i es qualifiquen amb `l` o bé `g` segons tinguin l'atribut de local o global.

En el programa assemblador 6.1, l'etiqueta .L4 té un identificador local i no es transfereix a l'objecte. L'etiqueta commuta_led és global ja que així es diu a la línia 11. Per tant es transfereix a l'objecte amb atribut de global.

6.2.5 La taula de reubicacions

Considereu de nou el mòdul objecte `modul1.o`. Podem conèixer la seva taula de reubicacions usant la següent ordre:

```
$ avr-objdump -r modul1.o
```

El resultat és una taula com la del llistat 6.4. Aquesta taula indica quines dades del mòdul han de ser reubicades si l'adreça base del codi canvia. Observeu que la línia 4 indica clarament que la taula s'aplica al codi del segment `.text`. La taula té tres columnes. La primera diu en quina posició del codi hi ha la dada que cal reubicar; la segona indica el tipus de reubicació que cal fer i la tercera explica quin valor ha de tenir la dada després de ser reubicada.

La línia 8 de la taula, per exemple, diu que en l'adreça 0x1a del segment `.text` del mòdul, cal aplicar una reubicació de tipus R_AVR_CALL de forma que el resultat acabi essent .text + 0x6. El listat 6.1 mostra el segment .text del mòdul. L'adreça 0x1a correspon a la línia 25. Com es veu, es tracta de la instrucció `CALL`. En aquest cas el que cal reubicar és precissament l'adreça de la subrutina que es crida ja que l'adreça concreta dependrà la regió de memòria on se situi finalment el mòdul. Una vegada coneguda la posició on s'ubicarà el mòdul, és a dir l'adreça de començament del segment de text, la taula de reubicacions indica que aquesta adreça ha de ser exactament

```
1
2 modul1.o:     file format elf32-avr
3
4 Sections:
5 Idx Name          Size      VMA       LMA       File off  Algn
6   0 .text         00000020  00000000  00000000  00000034  2**0
7                   CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
8   1 .data         00000000  00000000  00000000  00000054  2**0
9                   CONTENTS, ALLOC, LOAD, DATA
10  2 .bss          00000000  00000000  00000000  00000054  2**0
11                  ALLOC
```

Llistat 6.5: Taula de les seccions del mòdul `modul1.o`.

.text + 0x6. Si, per exemple, el mòdul cal ubicar-lo en la posició de memòria 0x2c, l'argument de CALL hauría de ser 0x2c + 0x6 = 0x32. Això és exactament el que ha succeït quan el muntador ha reubicat el mòdul per obtenir l'executable que teniu en el llistat 6.2. Com l'executable té pel davant el vector d'interrupcions i altre codi afegit per gestionar l'entorn, el mòdul `modul1.o` s'ha de reubicar a partir de l'adreça 0x2c. El muntador, doncs, ha reubicat el mòdul i ha decidit que l'operand de CALL és 0x32 com es pot comprovar a la línia 49.

La taula de reubicacions també conté altres tipus de reubicacions. La línia 7, per exemple, fa referència a una reubicació de tipus R_AVR_13_PCREL. En aquest cas es tracta d'un salt relatiu RJMP. En principi un salt relatiu no requereix reubicació ja que el desplaçament és independent de la ubicació del segment a la memòria. Això no obstant, l'assemblador deixa aquesta feina al muntador.

6.2.6 Les seccions de l'objecte

La darrera de les parts que conté un objecte són les diferents seccions de codi i dades. Aquesta és la part més significativa atès que la funció específica d'un mòdul objecte és precissament encabir aquestes dades. Si sobre el mòdul `modul1.o` s'executa l'ordre:

```
$ avr-objdump -h modul1.o
```

s'obté la taula de les seccions que conté el mòdul. El llistat 6.5 mostra la taula corresponent a `modul1.o`.

En aquesta taula s'observa que la secció .text d'aquest mòdul ocupa 32 B (0x20 en hexadecimal), està ubicada a partir de l'adreça 0, conté codi i és de només-lectura. També es veu com la resta de seccions no tenen contingut: són buides.

6.3 Exercicis

EXERCICI 6.1 En el programa 6.1 l'optimitzador ha substituït la crida a setup_led() per la seva implementació aplicant la tècnica d'*inlining*. Això no obstant, en el codi resultant continua existint la funció setup_led(). Per què?

EXERCICI 6.2 En el text d'aquest capítol es diu que un programa executable és auto-contingut. Què significa exactament això?

EXERCICI 6.3 Un programa (o mòdul) escrit en C pot definir una multitud d'identificadors de diversa natura, per exemple noms de variables, noms de funcions, etc. Alguns d'aquests identificadors acabaran generant símbols que sortiran a la taula de símbols del mòdul objecte corresponent. A més, aquests identificadors C poden ser precedits per qualificadors com ara **static** o **extern**. La presencia o no d'aquests qualificadors també pot afectar a la manera com aquests identificadors esdevindran símbols.

En aquest exercici es demana que investigueu de forma experimental la relació entre els tipus d'identificadors que poden trobar-se a C i els símbols que generen si és el cas. A tal efecte cal que considereu tots els casos que defineix la següent taula:

Tipus d'identificador	Qualificador		
	static	**extern**	(cap)
Variable global del mòdul	?	?	?
Nom de funció	?	?	?
Variable local en una funció	?	?	?

A base d'escriure petits programes de prova i explotar la taula de símbols del corresponent mòdul objecte determineu en quins casos l'identificador C acaba constituint un símbol de la taula i amb quins atributs. Raoneu la relació que hi ha entre les regles de visibilitat de C i el que heu observat a la taula de símbols.

EXERCICI 6.4 Què succeirà si el compilador genera una instrucció de salt relatiu RJMP i la distància entre el salt i el destí és superior al límit que imposa la instrucció? En quin pas del procés es detectarà aquest problema? Dissenya un exemple en què es doni aquest problema i analitza els resultats.

7 Llibreries

L'objectiu d'aquest capítol és introduir les llibreries estàtiques: el concepte, els seus usos i la forma de crear-les i mantenir-les.

Les llibreries són una eina d'organització que facilita en gran manera la construcció i el manteniment dels projectes. Entren en joc en la fase de muntatge i estan, per tant, íntimament relacionades amb el procés de muntatge.

En aquest capítol es presenten els principals conceptes, la forma com s'usen i gestionen i altres aspectes relacionats amb les llibreries estàtiques. Deliberadament es deixen fora les *llibreries dinàmiques*, que si bé superficialment tenen una forma similar, necessiten de la concurrència d'un sistema operatiu. En el nostre context aquest requeriment no es dóna.

7.1 Concepte

Una llibreria d'objectes —habitualment en direm llibreria— és en essència un fitxer que conté una col·lecció de mòduls objecte. Les llibreries entren en joc durant la fase de muntatge. En el capítol 6 hem vist com el muntador construïa els executables a base d'enllaçar diversos objectes. A més d'enllaçar objectes, els muntadors són capaços d'emprar mòduls objecte continguts dins de llibreries per a completar els executables com veurem més endavant. La fig. 7.1, mostra un esquema conceptual d'una llibreria. Com es pot apreciar, una llibreria conté un conjunt de mòduls objecte i, a més, algunes metadades que faciliten al muntador el seu ús. Per exemple, un índex dels objectes que conté o informació sobre els símbols públics que aquests mòduls exporten —una mena de taula de símbols de la llibreria—.

Molt resumidament, quan es munta un executable se li pot indicar al muntador que usi una o més llibreries i, en aquest cas, en la fase de resolució de referències, el muntador va afegint a l'executable mòduls objecte extrets de les llibreries si aquests mòduls permeten tancar la fase de resolució de referències. És a dir, si aquests mòduls de les llibreries aporten símbols que el muntador necessita resoldre.

Tot i la obvietat, cal destacar que llibreries i *headers* no són el mateix. Per alguna raó misteriosa hi ha una confusió terminològica molt habitual, fins i tot en textos escrits, que consisteix a

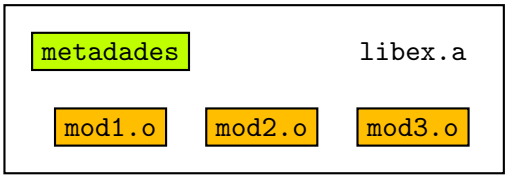


Figura 7.1: Concepte de llibreria

denominar els fitxers *header* —per exemple `math.h`— «llibreries». És un hàbit que crea una confusió poc edificant. És convenient dir les coses pel seu nom. Dels fitxers com `math.h` o `avr/io.h` cal dir-ne «capçaleres» o bé *headers*. El terme «llibreria» cal reservar-lo per als contenidors de mòduls objecte que aquest capítol tracta.

7.2 Creació i manteniment de llibreries

Les llibreries es creen i mantenen amb una eina específica: el gestor de llibreries (*library manager*). En el cas del toolchain de l'AVR, el gestor és l'aplicació `avr-ar`, [Fre10b]. El seu ús és molt senzill. A continuació es comenten les tasques més habituals de gestió. Considerarem a tall d'exemple que `obj1.o`, `obj2.o`, etc. són els que constitueixen la llibreria.

Creació La creació d'una llibreria es fa afegint-hi els objectes que en formen part. Naturalment abans cal haver compilat de forma escaient els fonts per tenir els objectes corresponents. Per crear la llibreria `libex.a` i afegir els mòduls `obj1.o` i `obj2.o`, podem fer:

```
$ avr-ar cr libex.a obj1.o obj2.o
```

L'opció `c` indica que es crea per primera vegada la llibreria. No és una opció d'ús obligat però ajuda a indicar la creació de la llibreria.

Afegir i modificar objectes Si el que es vol es afegir objectes a una llibreria existent, l'ordre adequada és la següent:

```
$ avr-ar r libex.a obj3.o obj4.o
```

Si algun dels objectes ja existia a la llibreria, es pot substituint simplement afegint-lo de nou. Suposeu, per exemple, que hem descobert un error en el mòdul `obj3.o`, l'hem corregit, hem re-compilat `obj3.c` i hem obtingut una nova versió de `obj3.o`. Per actualitzar la llibreria —recordeu que la llibreria conté una còpia del mòdul `obj3.o` que no està actualitzada!— cal que feu:

```
$ avr-ar r libex.a obj3.o
```

Amb això, inserireu el mòdul `obj3.o` del vostre directori de treball a la llibreria tot i canviant la versió antiga del mòdul que contenia.

Consulta dels mòduls Podem conèixer quins mòduls conté una llibreria amb la següent ordre, que llista els objectes d'una llibreria:

```
$ avr-ar t libex.a
```

Altres operacions A més de les operacions vistes anteriorment, que són les més rellevants, el gestor de llibreries permet altres possibilitats menys freqüents. Per exemple, es poden esborrar mòduls objecte continguts en la llibreria fent:

```
$ avr−ar d libex.a obj3.o
```

O també és possible extreure una còpia d'un objecte contingut en una llibreria fent:

```
$ avr−ar x libex.a obj3.o
```

7.3 Ús de llibreries

Les llibreries entren en joc en la fase de muntatge de l'executable. La lògica fonamental és que el muntador enllaça tots els objectes (i també el *runtime*) per crear un executable auto-contingut. Si en aquest procés resten referències per resoldre generalment es considera un error de muntatge. Si en l'ordre de muntatge s'hi afegeixen una o més llibreries, el muntador buscarà mòduls objecte que resolguin les referències pendents i, en cas de trobar-los, els incorporarà a l'executable i resoldrà les referències. En certa forma, el muntador tria de forma intel·ligent quins objectes de la llibreria ha d'incorporar a l'executable per crear l'executable.

Les llibreries s'incorporen a l'ordre de muntatge en la llista d'objectes a muntar. Per exemple:

```
$ avr−gcc −mmcu=atmega328p −o main main.o modA.o modB.o libex.a modC.o
```

El comportament d'aquesta ordre de muntatge, per exemple, provocarà que les referencies no resoltes que hi hagi després de processar `main.o`, `modA.o` i `modB.o` forcin la incorporació de mòduls de la llibreria `libex.a` que les puguin resoldre. És important notar que la posició de les llibreries és significativa: els símbols no resolts que incorpori el mòdul `modC.o` no es buscarien en la llibreria atès que aquest mòdul es processa una vegada ja s'ha processat la llibreria. Per aquesta raó, molt sovint trobem que les llibreries es citen al final de la ordre de muntatge, de manera que recullin totes les referències no resoltes que han resultat dels objectes.

A més de citar explícitament una llibreria en l'ordre de muntatge, també es pot usar un mecanisme de cerca automàtica de les llibreries similar al que usa el pre-processador per als *headers*. En aquest cas les llibreries es citen eludint el sufix i el prefix `lib` del seu nom[1] i referenciant-les amb l'opció `-l`. Una ordre equivalen a l'anterior fóra:

```
$ avr−gcc −mmcu=atmega328p −o main main.o modA.o modB.o −lex modC.o
```

La llibreria `libex.a` es busca en els directoris específics del sistema de fitxers que contenen llibreries, per exemple `/usr/lib`. En cas que tinguem interès en que busqui també en altres directoris, es pot indicar amb l'opció `-L`. Així es fa en el següent exemple:

```
$ avr−gcc −mmcu=atmega328p −L.. −L/home/projecte/libs −o main main.o modA.o modB.o −lex modC.o
```

[1] El conveni per anomenar les llibreries és `libxxx.a`. Aquest conveni és d'obligat compliment quan volem usar el mecanisme de cerca automàtica de les llibreries.

Llibreria estàndard Quan escrivim una ordre de muntatge, encara que explícitament no la citem, sempre es munta amb la llibreria estàndard. En el cas de l'AVR, la llibreria `libc.a`, que en la seva versió per l'AVR es coneix com `avr-libc`, [Boe+11], és la llibreria estàndard. De fet hi ha diverses versions de la llibreria estàndard de l'AVR segons el microcontrolador concret per al que muntem. El muntador tria l'apropiada segons el microcontrolador que indiquem en l'ordre de muntatge.

La llibreria estàndard conté un bon grapat d'objectes, molts dels quals contenen funcions que són ben usuals. Per exemple, atoi(), printf(), scanf() o strlen(). Noteu que aquestes funcions les usem sense implementar-les: d'on surt doncs la seva implementació? Dels mòduls objecte corresponents de la llibreria estàndard.

Llibreria matemàtica La llibreria matemàtica, anomenada `libm.a`, conté les implementacions de les funcions matemàtiques habituals: sin(), cos(), sqrt(), etc. El muntador no usa aquesta llibreria llevat que s'indiqui explícitament. Per això, si un cert programa usa funcions matemàtiques, l'ordre de muntatge ha d'incloure la llibreria matemàtica. Per exemple fent:

```
$ avr-gcc -mmcu=atmega328p -o main main.o mod.o -lm
```

7.4 Aspectes del disseny de llibreries

Tot i que tècnicament és possible, generalment una llibreria no és una col·lecció dispersa de mòduls objecte sense cap relació enter uns i altres. Normalment les llibreries juguen un paper estructural en els projectes i els sistemes: acostumen a agrupar objectes que constitueixen una unitat organitzativa coherent. Vegem-ne alguns exemples que coneixeu:

- La llibreria matemàtica, que s'ha comentat a l'apartat anterior, agrupa els objectes que implementen el conjunt de funcions matemàtiques predefinides.
- La llibreria de *unit testing* `cunits`, que va usar-se en les primeres pràctiques, agrupa tots els objectes per implementar el framework de testing.

Hi ha certes consideracions estratègiques qua cal considerar quan es decideix crear una llibreria. A continuació se n'esbossen algunes.

- La manera com s'organitza la pròpia llibreria a nivell d'objectes depèn sensiblement de l'objectiu que persegueix. La llibreria matemàtica, per exemple, acostuma a contenir multitud de petits objectes, cadascun dels quals implementa una sola funció —per exemple sqrt(). Aquesta disposició s'usa per minimitzar el codi que s'afegirà finalment a l'executable durant el muntatge. A tall d'exemple considereu que un cert executable usa la funció sqrt() i per aquest motiu es munta amb els objectes propis i, a més, la llibreria matemàtica. Durant el muntatge, la referència no resolta de la funció sqrt() arrossegarà el mòdul objecte de la llibreria matemàtica que la contingui. Si aquest mòdul és gran —per que conté altres funcions a banda de sqrt()— l'executable augmentarà significativament de mida. La decisió de com organitzar la llibreria, finalment, és de caire estratègic i cal prendre-la conscientment.

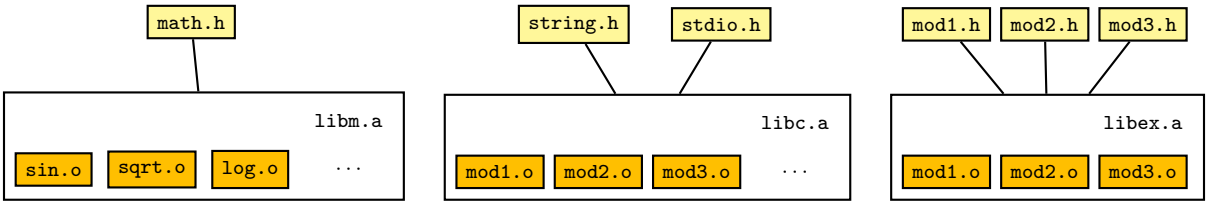


Figura 7.2: Diferents relacions entre una llibreria i els seus *headers*.

- Un segon aspecte important és la interfície —l'API si es vol dir així— que ofereix a l'usuari. Com succeeix quan parlem de mòduls, el conjunt de funcions exportades, les que veu l'usuari, constitueixen la interfície. El bon disseny d'aquesta interfície té una responsabilitat fonamental en la qualitat i usabilitat de la llibreria.

- Un tercer aspecte a considerar és la relació entre llibreries i fitxers de capçalera. Com les llibreries contenen funcions que s'usen dels dels mòduls dels projectes, cal que puguin disposar dels prototips. Així doncs, com passa en un mòdul, una llibreria acostuma a tenir associats un o més fitxers de `headers`.

  Hi ha diverses possibilitats per organitzar aquests *headers*. En un extrem, simplement podem usar els *headers* dels mòduls que conté la llibreria. A l'altre extrem existeix la possibilitat d'agrupar tota la informació en un sol *header*.

  La llibreria matemàtica, per exemple, agrupa tota la informació de capçaleres en un sol fitxer de *headers*: `math.h`.

  La llibreria estàndard de C, `libc`, aplica una política diferent. En aquest cas, les capçaleres de les funcions de la llibreria s'agrupen en diversos *headers* seguint un criteri semàntic. Cada *header* conté un conjunt de prototips que contenen funcions del mateix àmbit. Per exemple, `string.h` conté els prototips de les funcions de tractament de cadenes de caràcters o `stdio.h` conté els prototips de les funcions d'entrada/sortida. Observeu que no hi ha necessàriament relació entre els *headers* i els mòduls que conté la llibreria.

  Finalment, com és el conjunt de *headers* que acompanya una llibreria és una decisió de disseny i no ve forçada per cap limitació tècnica. La figura 7.2 il·lustra diversos tipus de relació entre una llibreria i els *headers* associats.

7.5 Manteniment de llibreries amb `make`

Quan en un projecte s'usen llibreries internes —que formen part del mateix projecte— el manteniment i construcció del projecte es complica: quan un objecte que forma part d'una llibreria s'actualitza, cal actualitzar també la llibreria. Aquest patró pot ser força enrevessat. Per exemple, pot succeir que la modificació d'un font desencadeni la re-compilació del seu conjunt de dependències i entre aquestes n'hi hagi una que forma part d'una llibreria.

`Make` té un conjunt d'eines pensades específicament pel treball amb llibreries que en simplifiquen molt el manteniment i l'ús. Aquestes eines tenen dues vessants: les dependències de llibreries i el manteniment de llibreries.

Dependències de llibreries Si un executable es munta en base a uns objectes i unes llibreries, en el sentit de `make` depèn dels objectes i també de les llibreries. Com es pot expressar la dependència d'una llibreria?

Hi ha dos camins per fer-ho que depenen de com es cita la llibreria en l'ordre de muntatge.

Si la llibreria es cita explícitament —sense usar el mecanisme de cerca—, simplement es tracta com una dependència ordinària.

Si la llibreria es cita usant el mecanisme de cerca —o sia amb la opció de muntatge `-lxx`— aleshores `make` ofereix un mecanisme especial per expressar la dependència. En una regla de muntatge podem escriure:

```
main: main.o mod1.o mod2.o -lex
```

La dependència `-lex` té una consideració especial: es reconeix com una referència a la llibreria `libex.a`, que cal buscar en un conjunt de directoris. `make` la cercarà en els directoris estàndard de llibreries i, a més, en els indicats per la variable `VPATH` i, una vegada trobada, considerarà el fitxer trobat.

El següent `makefile` esquemàtic il·lustra el seu ús:

```
VPATH=.:~/projecte/libs

main: main.o mod1.o mod2.o -lex
main.o: mod1.h mod2.h ex.h
mod1.o: mod1.h
mod2.o: mod2.h ex.h
```

Manteniment de llibreries `Make` permet citar els objectes que conté una llibreria usant una notació específica. Així, `libA(mod1.o)` significa el mòdul objecte `mod1.o` que conté la llibreria `libA`. Aquesta notació es pot extendre per parlar de diversos mòduls: `libA(mod1.o mod2.o)`.

Poder identificar un mòdul dins d'una llibreria permet escriure regles com la següent:

```
libex.a(mod1.o) : mod1.o
    avr-ar r libex.a mod1.o
```

Fixeu-vos que s'expressa la dependència entre un mòdul dins de la llibreria i el mòdul fora de la llibreria. Vegem una situació concreta per entendre aquesta situació. Suposeu que en un directori hi tenim els següents fitxers:

```
$ ls
libex.a
mod1.o
mod1.c
mod1.h
```

En aquest directori hi ha realment dues còpies del mòdul `mod1.o`, la que apareix en el llistat del directori i la continguda dins la llibreria `libex.a`. Si ara modifiquem `mod1.c` i recompilem es genera un nou `mod1.o`, però la còpia que conté la llibreria no ha canviat! Per canviar-la cal usar `avr-ar` i actualitzar la llibreria. Això exactament és el que expressa la regla de `make` anterior.

Les regles predefinides també apliquen als membres de les llibreries: existeix una regla predefinida que relaciona un objecte membre d'una llibreria amb el corresponent objecte fora de la llibreria. El seu dispar produeix l'actualització del mòdul en la llibreria. Aquesta regla es parametritza amb la variable `AR`, que conté el nom del gestor de llibreries que cal aplicar.

A efectes de mostrar el seu ús, suposeu que voleu mantenir la llibreria `libex.a`, que conté els mòduls `mod1.o`, `mod2.o` i `mod3.o`. Suposeu que llibreria, fonts i objectes viuen en un mateix directori. El següent `makefile` faria la feina:

```
AR = avr-ar
CC = avr-gcc
CFLAGS = -Wall -Os -mmcu=atmega328p -fshort-enums
CPPFLAGS = -DF_CPU=16000000UL

libex.a: libex.a(mod1.o mod2.o mod3.o)

mod1.o: mod1.h
mod2.o: mod2.h
mod3.o: mod3.h
```

Bibliografia

[Ard11] Arduino. *Arduino Uno*. Ang. 2011. URL: http://arduino.cc/en/Main/ArduinoBoardUno (cons. 08-02-2017).

[Atm09] Atmel. *ATMega328P datasheet*. Ang. Vers. 8161D–AVR–10/09. 2009. URL: http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf (cons. 08-02-2017).

[BBD91] Mike Banahan, Declan Brady i Mark Doran. *The C Book*. Ang. 2a ed. Addison-Wesley, Pearson Education, 1991. URL: http://publications.gbdirect.co.uk/c_book (cons. 08-02-2017).

[BH02] Mark Burgess i Ron Hale-Evans. *The GNU C Programming Tutorial*. Ang. 4.1. 2002. 290 pàg. URL: http://www.crasseux.com/books/ctut.pdf (cons. 08-02-2017).

[Boe+11] Werner Boellmann et al. *AVR Libc User Manual*. Ang. 2011. URL: http://www.nongnu.org/avr-libc/user-manual (cons. 08-02-2017).

[Fre10a] Free Software Foundation. *GNU Assembler User Manual*. Ang. Vers. 2.21. 2010. URL: http://sourceware.org/binutils/docs-2.21/as (cons. 08-02-2017).

[Fre10b] Free Software Foundation – GNU Project. *GNU Binutils User Manual*. Ang. Vers. 3.24. 2010. URL: https://sourceware.org/binutils/docs-2.34/binutils/index.html.

[Fre10c] Free Software Foundation – GNU Project. *GNU C Preprocessor User Manual*. Ang. Vers. 4.6.1. 2010. URL: http://gcc.gnu.org/onlinedocs/gcc-4.6.1/cpp (cons. 08-02-2017).

[Fre10d] Free Software Foundation – GNU Project. *GNU GCC User Manual*. Ang. Vers. 4.6.1. 2010. URL: http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc (cons. 08-02-2017).

[Fre10e] Free Software Foundation – GNU Project. *GNU Make User Manual*. Ang. Vers. 3.82. 2010. URL: http://www.gnu.org/software/make/manual (cons. 08-02-2017).

[Gro10] The SCO Group. *System V Application Binary Interface —DRAFT— 19 October 2010*. Ang. 2010. URL: http://www.sco.com/developers/gabi/latest/contents.html (cons. 08-02-2017).

[Joi03] Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Working Group 14. *Rationale for International Standard – Programming Languages – C*. C99. Ang. Rationale. Vers. Revision 5.10. ISO/IEC, 2003. 224 pàg. URL: http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf (cons. 08-02-2018).

[Joi07] Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Working Group 14. *Draft of the C99 standard with corrigenda TC1, TC2, and TC3 included*. Ang. Draft standart ISO/IEC 9899:TC3. ISO/IEC, 2007. 519 pàg. URL: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf (cons. 08-02-2017).

[KR88] Brian Kernighan i Denis Ritchie. *The C Programming Language*. Ang. 2a ed. Addison-Wesley, Pearson Education, 1988. 274 pàg. ISBN: 9780131103627.

[Lab95] Unix System Laboratories. *Executable and Linkable Format (ELF)*. Ang. 1995. URL: `http://refspecs.linuxfoundation.org/elf/elf.pdf` (cons. 08-02-2017).

[Par03] Nick Parlante. *Essential C*. Ang. 2003. 45 pàg. URL: `http://cslibrary.stanford.edu/101/EssentialC.pdf` (cons. 08-02-2017).

[Rus10] David Russell. *Introduction to Embedded Systems. Using ANSI C and the Arduino Development Environment*. Ang. Synthesis Lectures on Digital Circuits and Systems 30. Morgan & Claypool Publishers, 2010. 255 pàg. ISBN: 9781608454983. DOI: `10.2200/S00291ED1V01Y201007DCS030`.

[Sil99] Joseph H. Silverman. *C Reference Card (ANSI)*. Ang. 1999. 2 pàg. URL: `http://www.math.brown.edu/~jhs/ReferenceCards/CRefCard.v2.2.pdf` (cons. 08-02-2017).