Lathund till C-språket
B Knudsen's C-kompilator Cc5x för PIC

C-programmets delar, datatyper
- Filnamn
- Kommentarer
- Satser och sammansatta satser
- Kompilatordirektiv, preprocessor

Variabler
- Indexerade variabler
- Globala variabler
- Lokala variabler
- Initiering av variabler

Funktioner

Operatorer
- Aritmetiska operatorer
- Vilkorsoperatorer
- Logiska operatorer
- ”bit för bit” operatorer
- Tilldelning
- Speciella tilldelningsoperatorer
- Increment och decrement -operatorer

Styrning av programflöde
- Programslingor med while satsen
- Programslingor med do satsen
- Programslingor med for satsen
- if satsen
- Tillägg av else
- switch satsen
- brake satsen

Tabeller

Strängkonstanter

Datatyper, ANSI-C respektive B Knutsen CC5x

Cc5x kompilatorn
- Codepages
- Rambanks
- Bitvariabler och bitkonstanter
- Cc5x interna variabler
- Cc5x interna funktioner

Kom igång med C för småprocessorer
C-Dialekt: *B Knudsen's* C-kompilator Cc5x för PIC

C är ett enkelt och litet språk, som kan översättas till maskinkod med enkla och små kompilatorer. Hela vår programmeringsmiljö rymms faktiskt på en enda diskett. Språket C är ett procedurinriktat språk i motsats till de nyare objektorienterade språken som tex C++, Java eller VisualBasic.net. Detta innebär att C:s datatyper och operatorer har sitt ursprung i de register och instruktioner som finns i verklighetens datorkretsar. Den som ”känner” sin processor kan därför lära sig att skriva kompakta och snabba program med C.
Samtidigt är språket tillräckligt abstrakt för att man ska kunna fjärma sig från processorns alla detaljer, och skriva portabla program som kan flyttas mellan olika processorer.

Följande "lathund" till (delar av) C-språket är kortfattad och förutsätter att Du har erfarenhet från något annat programspråk. Har Du redan lärt dig C kan den tjäna som en kort sammanfattande repetition.

C-programmets delar och datatyper

Filnamn
Tag för vana att alltid använda korta filnamn utan de svenska tecknen och utan mellanslag (maximalt 8 tecken och med 3 teckens filnamnstillägg). Detta kommer att bespara dig många problem eftersom en del program i vår programmeringsmiljö är DOS-program.

Kommentarer
En C kommentar börjar med
`/*`
och slutar med
`*/`

Alternativt börjar man en kommentar med
`//`
och då fortsätter kommentaren till *radens slut*.

Använd engelska som språk för dina kommentarer. Än idag finns det många program som hakar upp sig inför våra specialtecken å ä ö. Vid felsökning av program brukar man använda `//` i början av rader som man vill "kommentera bort". Man kan då snabbt växla mellan att prova programmet med eller utan vissa rader.

Satser och sammansatta satser

En sats i C avslutas *alltid* med semikolon "`;`". Sammansatta satser består av ett antal satser som inneslutits mellan klammerparenteser "`{`" "`}`". ("Måsvingar"). Ett sådant block med satser avslutas *inte* med semikolon (inget semikolon efter den avslutande klammerparentesen).
Radbrytningar har ingen betydelse. Man kan arrangera koden på nästan vilket sätt som helst eftersom C hoppar över mellanslag och liknande tecken när koden tolkas. Vana C-programmerare brukar emellertid följa vissa oskrivna regler. Ett typiskt C-program har en sats per rad, och det är praxis att man skriver klammerparenteserna *indragna* så att man lätt kan se vilka som hör ihop.

Kompilatordirektiv, preprocessor

En del rader i ett C-program är kompilatordirektiv. C-kompilatorn har en preprocessor, en "ordbehandlare" som går före genom koden och söker upp "nyckelord" och gör utbyten av fraser eller skjuter in textblock. Kompilatordirektiven börjar alltid med nummertecknet "#", men de ska *inte* avslutas med semikolon eftersom de inte är C-satser.

```
#include "16f84.h"
```

Kompilatordirektivet `include` betyder att filen `16f84.h` ska kopieras in på *denna plats* i programmet och tas med vid kompileringen. Filen `16f84.h` innehåller bland annat variabeldefinitioner för bitvariabler så att de får samma namn som PIC-processorns pinnar. (I allmänhet samma namn som används i processorns datablad.)

```
#define ON 1
```

Överallt i programmet där man skrivit "nyckelordet" `ON` byts detta ut mot siffran `1`.

Variabler

Alla variabler måste deklareras. Detta gäller namn och typ. I namnen görs det skillnad på stora och små bokstäver. `Hej` och `heJ` och `hEj` är således tre olika variabler.
Använd engelska namn på variablerna (inga å ä ö eller andra "konstiga" tecken).
En del namn är upptagna, reserverade för C-språket. Därför kan man till exempel inte kalla en variabel som används av en strömställare för `switch` eftersom C-språket har en "switch"-sats. Se avsnittet om Switch-satsen.
Namn på de vanligaste registren tex. `PORTB`, `TRISB` är kända för kompilatorn och ska *inte* deklareras.

PIC-processorerna är små datorkretsar med 8-bitarsregister. En teckenlös 8-bitarsvariabel (0...255) med namnet tal deklareras som:

```
char tal;
```

Och en 8-bitarsvariabel med tecken (-128...0...127) som:

```
int tal;
```

Behöver man arbeta med längre tal kan man för 16-bitars teckenlösa tal (0...65536) deklarera:
```
uns long tal;
```
och för 16-bitarstal med tecken (-32768...0...32767) deklarera:
```
long tal;
```
Eftersom PIC-processorerna har ett antal bitoperationer är det lämpligt att använda bitvariabler (detta finns normalt inte i C). En bitvariabel (0/1) deklareras som:
`bit a;` Om variabeln har en fast adress, tex om det gäller en portpinne eller en registerflagga skriver man:
```
#pragma bit a @ PORTB.3
```
Detta betyder att bit nummer 3 i `char PORTB` har namnet `a`.
I includefilerna för de olika processorerna har "pinnarnas" namn deklarerats på detta sätt.
Observera att kompilatorn Cc5x tillåter bekväm adressering av enskilda bitar i variabler genom att man skriver: `variabelnamn.bitnummer`
Man kan tex "flytta" (kopiera) bitar genom att skriva:
```
a.4 = a.3;
```
Man kan byta värde på en bit (! betyder inte, tvärtom):
```
a.4 = !a.4;
```

Indexerade variabler

Man kan använda indexerade variabler, men eftersom PIC-processorernas minne är litet bör de vara endimensionella.
```
char vektor[3];
```
Här dimensionerar man 3 variabler. Index börjar med 0. Variablernas namn blir således `vektor[0]`, `vektor[1]` och `vektor[2]`.
(Observera! Kompilatorn Cc5x tillåter *inte* **indexerade konstanter**. Man kan således *inte* skriva:
```
char month[12]= { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
/* does not work */
```
Inte heller kan man definiera texter, **strängkonstanter**:
```
char text[]="Hello world!"; /* does not work */
```
PIC-processorerna har en effektiv mekanism för att "slå upp" i tabeller och det är i stället den man ska använda. (Se avsnittet om tabeller).

Globala variabler

Om en variabel deklareras i början av programmet, *utanför* huvudprogrammet `main()`, blir den global. Detta innebär att variabeln blir direkt åtkomlig från alla andra delar av programmet och från alla funktioner. Programmets funktioner kan därför kommunicera med varandra med hjälp av globala variabler.
Man brukar varna för att använda globala variabler, men för de små program som rymms i PIC-processorerna är det trots allt små risker med globala variabler.

Stora program som är uppdelade över många filer bör däremot inte använda globala variabler. Om det visar sig att en global variabel innehåller "fel" värde, så är det svårt att hitta var någonstans i programmet felet har uppstått, det kan ju vara var som helst, och i vilken fil som helst.

Lokala variabler

Om en variabel deklareras i början av en funktion, tex i funktionen `main()`, så blir den lokal. Variabeln är då "okänd" i andra funktioner. Om man deklarerar en variabel med samma namn i någon annan funktion, så blir det inte samma variabel utan en ny. Kompilatorn Cc5x tillåter även att man deklarerar variabler i början av programslingor (tex. while, for). Sådana variabler blir då lokala för slingan.
Lokala variabler innebär att man utnyttjar minnet effektivt eftersom det är samma minnesceller som används av alla funktioner, men till olika ändamål.

Initiering av variabler

När programmet startar har alla variabler ett okänt innehåll. Man kan initiera lokala variabler direkt när man deklarerar dem, men Cc5x tillåter *inte* att *globala* variabler initieras vid deklarationen.

```
char tal = 3;
```

Cc5x har en inbyggd funktion för 0-ställning av *allt* RAM-minne, det vill säga det minne som variabler ligger i.

```
clearRAM();
```

C saknar logiska variabler, Boolska variabler. **Falskt** representeras av talet 0, medan alla andra tal står för **sant**. Den som vill kan åtgärda detta genom att definiera:

```
#define TRUE 1
#define FALSE 0
```

Funktioner

C-språket är funktionsbaserat. Alla C-program har åtminstone en funktion med namnet **main** som markerar var programmet börjar och slutar. Ett program med bara main är det det enklast tänkbara C-programmet:

```
main()
  {
  }
```

Klammerparenteserna visar var programmet börjar och var det slutar.

Här följer ett program, som visserligen inte utför något vettigt, men som är ett mer komplett exempel på C:s användning av funktioner.
(begrepp: deklaration, anrop, retur, definition)

```
char summa( char, char);  /* declare function summa        */
void neg( void );         /* declare function neg          */
char a;                   /* declare global variable a     */
a = 1;                    /* initialize global variable a  */

void main( void )         /* main function - the "program" */
  {
     char b=3; /* declare and initialize local variable b  */
     char c=2; /* declare and initialize local variable c  */
     char d;   /* declare local variable d                 */
     char e;
     e=summa( b, c);         /* function call             */
     neg( );
     d=summa( b, a );        /* function call             */
  }                          /* end of program            */

char summa( char x, char y)  /* function header           */
  {                          /* start of function body    */
     char f;
     f = x + y;
     return f;
  }                          /* end of function body      */

void neg( void )
  {
     a = -a;
  }
```

Programexemplet börjar med att programmets funktioner *deklareras*. Här anges funktionernas namn, vilken typ av variabler som ska skickas med, och vilken typ av variabel som man kan få tillbaka när funktionerna avslutas.

Funktionen **summa** ska *anropas* med två tal av typen char, och den kommer att *returnera* ett tal av typen char. Denna information behöver kompilatorn.

Själva funktionen **summa** *definieras* på annat ställe, i detta exempel efter **main**. Funktionens huvud ger samma information som den tidigare deklarationen
```
char summa( char x, char y)
```
men denna gång står även variabelnamnen `x` och `y` med. De tal som skickas med när funktionen anropas kopieras och läggs i dessa variabler. I funktionen används sedan `x` och `y`.
```
f = x + y;
```

De flesta funktioner både mottar och returnerar värden. Det enda sättet att returnera ett värde från en funktion är med en **return** sats. Den kan utföra två saker. Den avslutar funktionen och återför kontrollen till den sats som anropat funktionen, men den kan också överföra ett värde.
```
return f;
```

Här avslutas funktionen och värdet i variabeln `f` returneras.
I huvudprogrammet anropas funktionen:
```
e = summa( b, c);
```
Här anges att det är värdena i variblerna `b` och `c` som ska skickas med till funktionen vid anropet, och att resultatet ska lagras i variabeln `e` när funktionen avslutats.

Funktionen **neg** deklareras och definieras som:
```
void neg( void )
```
void står för att det *inte* är något värde som skickas med när funktionen anropas och att den *inte* kommer att returnera något värde när den avslutas.
Funktionen kan ändå "läsa" variabeln `a` och direkt förändra den eftersom det är en global variabel (den har deklarerats i början av programmet, utanför funktionerna).
```
a = -a;
```

Funktionen saknar **return** sats och därför avslutas den när klammerparentesen "`}`" passeras.

Funktionerna definieras alltid utanför **main**. En funktion *kan inte* definieras inuti en annan funktion, men en funktion kan anropa en annan funktion, som i sin tur kan anropa en annan funktion ...

Operatorer

C-språket har fler operatorer än andra programmeringsspråk (fler än 30 st). Ett antal ovanliga operatorer har bidragit till att C-kod verkar svår (för den oinvigde).
Man behöver inte använda de udda operatorerna, men om man hittar kodexempel tex på Internet, är det vanligt att programmeraren använt alla C-språkets "finesser" fullt ut. Därför måste man känna till att de finns för att vid behov kunna slå upp dem i en handbok.

Aritmetiska operatorer	
Operator	**Beskrivning**
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo, rest

Modulo operatorn ("%") är förmodligen mindre känd än de övriga operatorerna. Den gör en heltalsdivision, och resultatet blir divisionens rest.
Heltalsdivision: 22/3 = 7
Modulo, resten: 22%3 = 1

Vilkorsoperatorer	
Operator	**Beskrivning**
<	Mindre än
>	Större än
< =	Mindre än eller lika med
> =	Större än eller lika med
= =	Lika med
! =	Inte lika med

Logiska operatorer	
Operator	**Beskrivning**
&&	Logiskt AND
||	Logiskt OR
!	Logiskt NOT

De logiska operatorerna används för att kombinera olika vilkorsuttryck till sammansatta vilkor, tillexempel för användning i if-satser. Att x ska vara större än 10 eller y mindre eller lika med 5 uttrycks så här:

```
( x > 10 ) || ( y < = 5 )
```

"bit för bit" operatorer	
Operator	**Beskrivning**
&	AND
|	OR
^	EXOR
~	Complement (NOT)
< <	Vänsterskift
> >	Högerskift

"bit för bit" operatorerna används för att "vaska ut" information om enskilda bitar i en variabel. Eftersom PIC-processorerna har bit-operationer kan man enkelt nå enskilda bitar direkt. Man har då mindre användning för "bit för bit" operationerna.

Tilldelning

Tilldelningsoperatorn ("=") låter en variabel bli lika med en annan.

```
x = y;
```

I C-språket kan man kombinera tilldelningsoperatorn med de aritmetiska operatorerna (eller "bit för bit" operatorerna) till bekväma förkortningar, för att erhålla speciella tilldelningsoperatorer.

Speciella tilldelningsoperatorer		
Uttryck	**Ekvivalent uttryck**	**Operation**
x + = y	x = x + y	Addition
x - = y	x = x - y	Subtraktion
x * = y	x = x * y	Multiplikation
x / = y	x = x / y	Division
x % = y	x = x % y	Modulo
x & = y	x = x & y	AND
x | = y	x = x | y	OR
x ^ = y	x = x ^ y	EXOR

Increment och Decrement operatorer

En vanlig uppgift i program är att öka eller minska en variabel med ett. Det går naturligtvis att göra med addition eller subtraktion, men de flesta processorer har även speciella, optimerade, instruktioner för detta. C-språket har därför Increment (öka med ett) och Decrement (minska med ett) -operatorer:

```
x++ ; /* add 1 to x */
x-- ; /* subtract 1 from x */
```

Operatorerna kan även skrivas `++x` och `--x`. Då utförs ökningen/minskningen av `x` *innan* variabeln används (annars *efter*). Med dessa fyra varianter kan programmeraren skräddarsy uttryck så att de blir så korta och effektiva som möjligt.

Styrning av programflöde

Programslingor med while satsen

```
while ( x < 5)
  {
    x = x-1;
  }
```

Den grundläggande programslingan är **while** satsen. De satser som står mellan klammerparenteserna utförs gång på gång tills vilkoret `x < 5` i parentesen blir uppfyllt, även vilkoret i parentesen testas således varje gång. Eftersom variabeln x i exemplet ovan minskas för varje varv i slingan, kommer den till sist att bli mindre än 5 och då avbryts slingan och programexekveringen fortsätter med nästa uttryck i programmet.
De flesta program ska gå för evigt, ett bankomatprogram till exempel börjar ju om för varje ny kund. Till en sådan evighetsslinga behövs en **while** sats med ett vilkor som *alltid* är uppfyllt. Eftersom tex talet 1 är "sant" enligt C, så brukar man skriva så här:

```
void main ( void )
{
   while ( 1 )  /* program runs forever */
     {
     }
}
```

(Den som inte fått detta förklarat för sig har nog svårt att förstå …)

Programslingor med do satsen

En **do** slinga är helt enkelt en uppochnervänd **while** slinga.

```
do
  {
    x = x-1;
  } while ( x < 5 );
```

Innehållet i en **do** slinga utförs *innan* testet görs. Det innebär att **do** slingan alltid utförs åtminstone en gång.

Programslingor med for satsen

C-språkets **for** slinga är mer finessrik än övriga programspråks.

```
for ( i = 1 ; i < 4 ; i++)
  {
    char sum;
    sum += i^2;
  }
```

Parentesen i slingans huvud har tre positioner åtskilda med semikolon.
Den första positionen är till för ett initieringsuttryck. Här placerar man något som man vill få utfört innan slingan startar. I exemplet initieras räknevariabeln `i` till talet 1.
Positionen i mitten är för slingans termineringsvilkor (det vilkor som avbryter slingan), i exemplet avbryts slingan när `i` är 3 (`i < 4`). Satsen i den sista positionen utförs varje varv, precis som satserna mellan klammerparenteserna. Här brukar man räkna upp eller räkna ner räknevariabeln, men andra satser är också möjliga.
Man behöver *inte* utnyttja alla positionerna. **for** slingan här nedan saknar termineringsvilkor så detta blir ett evighetsprogram för det som står mellan klammerparenteserna.

```
for ( ; ; )  /* program runs forever */
  {
  }
```

if satsen

if else och **switch** är "beslutsfattande" satser. Dessa satser överför kontrollen till andra delar av programmet beroende på utfallet av ett logiskt test.

```
if ( (contact == 1) && (a < 5) )
  {
     lightdiode = 1;
  }
```

Om vilkoret `(contact == 1) && (a < 5)` är uppfyllt så utförs satsen `lightdiode = 1;`, annars inte.

Tillägg av else

Nyckelordet **else** används alltid i samband med en **if** sats. Med **else** bildar man ett "antingen eller" uttryck som utför en sats när vilkoret är sant och en annan när det är falskt.

```
if ( contact == 1 )
  lightdiode = 1;
else
  lightdiode = 0;
```

switch satsen

En nackdel med **if** och **if - else** satser är att de endast tillåter ett val för varje nyckelord. Antingen utför man den sats som följer efter **if** (eller inte) eller så utför man den som följer efter **else**. För att utföra mer komplexa test måste man stapla fler **if** eller **if - else** satser efter varandra.

switch satsen är en elegant lösning på detta problem. En variabel som kan ha många värden testas, och utifrån variabelns värde utförs olika satser. En vanlig tillämpning är avkodning av tangentbord, där de olika knapparna har olika funktioner.

```
switch( a )
  {
    case 1:
      b = 5;
      break;
    case 2:
      b = 7;
      break;
    case 3:
      b = 19;
      break;
    default:
      b = 0;
      break;
  }
```

Som för de andra "beslutsfattande" satserna innehåller parentesen det uttryck som ska utvärderas. Här är det variabeln `a`. De olika alternativen följer mellan klammerparenteserna. `case 1:` och `case 2:` och `case 3:` är etiketter (labels) som programmet kan hoppa till om `a` är 1 2 eller 3. Det som utförs är att en annan variabel `b` tilldelas värdet 5, 7 eller 19 (således en omkodning, $1 \rightarrow 5$, $2 \rightarrow 7$, $3 \rightarrow 19$). Nyckelordet **break** behövs för att programmet ska "hoppa ur" **switch** satsen när det rätta alternativet utförts - annars fortsätter man med alla efterföljande alternativen i listan.

Vad händer om variabeln `a` har ett värde som *inte* finns med i listan? Under etiketten `default:` hamnar alla värden som inte finns med i den övriga listan. Varibeln `b` tilldelas där 0.
I Switch-satsen ska case etiketterna betecknas med heltalsvärden. Med C:s preprocessor kan man ordna så att olika namn på etiketterna ersätts med heltalsvärden.

break satsen

I samband med **switch** satsen har vi sett att man kan "bryta sig ur" en slinga med **break**. Detta är även användbart i andra situationer, man kan tillexempel lämna en "evighetsslinga"!

```
while ( 1 )  /* loop forever  */
  {
    if ( contact == 1 ) break;  /* no, it was not forever */
  }
```

Tabeller

PIC-processorernas instruktioner är 12-14 bitar långa. Konstanter, sk "literals", lagras i programminnet inbakade i instruktioner. Det finns flera instruktioner av denna typ. Tabellvärden lagras i allmänhet med assemblerinstruktionen `RETLW k` (RETurn with Litteral in `W`), där `k` är den 8 bitars konstant som ska bakas in i instruktionen. Om processorn "hoppar" till en sådan instruktion, skickas den tillbaks med talet `k` i arbetsregistret, w-registret. Vid assemblerprogrammering används detta tillsammans med indexerad adressering så att man med hjälp av ett index kan styra vilken av tabellens konstanter man ska komma tillbaks med. Detta kallas för Computed GOTO. Exempel assemblerprogram. `k1` `k2` `k3` är tabellens tre konstanter.

```
          CALL TABLE ; W contains table offset
          ...
TABLE     ADDWFPC    ; W = offset, go to table + offset
entry
          RETLW k1   ; Begin table,  return with k1 in w
          RETLW k2   ;               return with k2 in w
          RETLW k3   ; End of table, return with k3 in w
```

Det finns inget "självklart" sätt att utrycka detta med C-språket. B Knutsen har infört en (inbyggd) funktion `skip(i)` som ska lösa detta. Denna innebär att man hoppar över *det antal* C-instruktioner som anges med "i".

Exempel C-funktion (enligt CC5X). `k1 k2 k3` är tabellens tre konstanter.

```
char table(char i)
{
  skip(i);   /* internal function in CC5X  */
  return k1;
  return k2;
  return k3;
}
```

Med detta skrivsätt får man samma kompakta kod som vid assemblerprogrammering med Computed GOTO, med en C-funktion som är "nästan" korrekt (undantaget är skip).

Strängkonstanter

Cc5x tillåter inte indexerade strängkonstanter. Man kan således *inte* skriva:

```
char text[]="Hello world!"; /* does not work */
```

I stället för en indexerad strängkonstant definierar man en funktion som returnerar strängens tecken. Variabeln `i` pekar ut vilket tecken man vill åt.

```
char text(char i)
{
  skip(i);   /* internal function in CC5X  */
  return 'H';
  return 'e';
  return 'l';
  return 'l';
  return 'o';
  return ' ';
  return 'w';
  return 'o';
  return 'r';
  return 'l';
  return 'd';
  return  0 ;
}
```

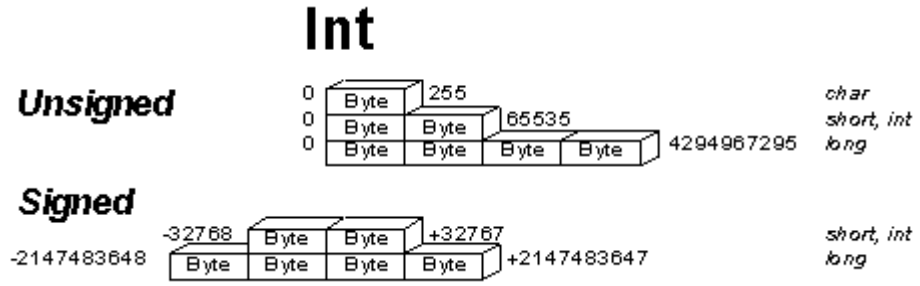
Det finns ett *kortare skrivsätt* för detta (med kompilatordirektivet `#pragma return`):

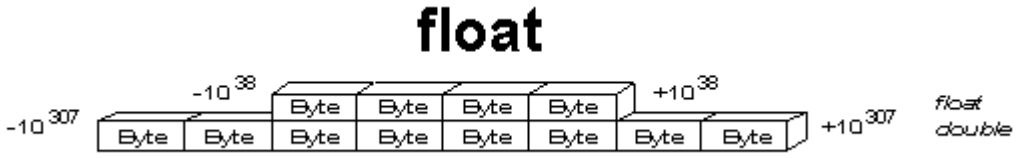
```
char text(char i)
{
  skip(i);   /* internal function in CC5X  */
  #pragma return "Hello world" 0
}
```

Data typer i C

ANSI-C respektive *B Knutsen* CC5x C-kompilator

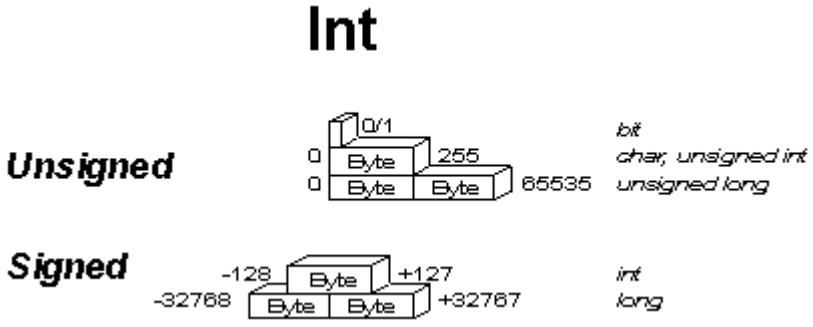
C:s data typer för en "typisk" processor (ANSI-C)





C:s data typer för PIC-processorn (CC5x C-kompilator)

Omfånget för C:s datatyper är inte fastställt, utan beror på den använda processorns uppbyggnad och prestanda. int är den grundläggande datatypen, och den blir för PIC-processorn 8-bitar eftersom processorns instruktioner arbetar med bytes om 8-bitar. PIC-processorn har även bit-instruktioner och därför har Cc5x utökat C med den mindre datatypen bit 1/0. (C saknar normalt denna datatyp).
De alternativa namnen i tabellen nedan kan användas i stället för C:s typbeteckningar. De anger på ett tydligare sätt vilket omfång olika variabler har.



Data type	Meaning	Size	Min	Max
bit	bit in byte	1/8	0	1
unsigned	8 bit unsigned	1	0	255
char	8 bit unsigned	1	0	255
unsigned long	16 bit unsigned	2	0	65535
int	8 bit signed	1	-128	+127
signed char	8 bit signed	1	-128	+127
long	16 bit signed	2	-32768	+32767

Alternativa namn som kan användas för datatyperna: (CC5x)

```
  TYPE   SIZE            MIN          MAX
  ----   ----            ---          ---
  int8      1           -128          127
 int16      2         -32768        32767
(int24      3       -8388608      8388607)
(int32      4    -2147483648   2147483647)

  uns8      1              0          255
 uns16      2              0        65535
(uns24      3              0     16777215)
(uns32      4              0   4294967295)
```

(Datatyperna int24, int32, uns24, uns32 är *inte* tillgängliga för den fria utvärderingsversionen av CC5x, utan är reserverade för den betalade versionen)

OBSERVERA! När man gör multiplikationer eller divisioner med 16-bitars tal ska man ta med biblioteksfilen math16.h.

Cc5x - kompilatorn

Codepages

PIC-processorerna har programminnet uppdelat i "sidor" (0, 1, 2, 3), sk. codepages, om 512 instruktioner. Cc5x börjar att lägga ut kod på `page 0` och ger felmeddelande om denna sida inte räcker till . Skulle detta inträffa så skriver man dit instruktionen `#pragma codepage 1`, så att alla instruktioner därefter hamnar på nästa sida (och så vidare med `codepage 2` vid behov).

För att få kompakt kod behövs en noggrann "sidplanering", men detta är något som man knappast bryr sig om vid prototyputveckling.

Rambanks

PIC-processorernas registerarea (RAM-minne) är uppdelat i "rambankar" (0, 1, 2, 3). Cc5x börjar att fylla `rambank 0`. Man kan byta rambank med instruktionen `#pragma rambank 1` och då placeras alla variabler som deklareras därefter ut i nästa rambank (`rambank 1`).

En del minnesceller återfinns på samma plats i alla rambankar, så kallad *mapped RAM*. Man kan välja att placera variabler i "mappad ram" (så länge det finns plats) med instruktionen `#pragma rambank - .`

Bästa användningen av RAM-bankar kräver mycket planering, men detta är något som man knappast behöver bry sig om vid prototyputveckling.

Så här kan den "rapport" som kompilatorn skriver ut om användningen av program-minne och RAM-minne se ut:

```
   RAM: 00h : -------- -------- -------- --------
   RAM: 20h : ==.***** ******** ******** ********
   RAM: 40h : ******** ******** ******** ********
   RAM: 60h : ******** ******** ******** ********
   RAM: 80h : -------- -------- -------- --------
   RAM: A0h : ******** ******** ******** ********
   RAM: C0h : ******** ******** ******** ********
   RAM: E0h : ******** ******** ******** ********
   Codepage 0 has   68 word(s) :   3 %
   Codepage 1 has    0 word(s) :   0 %

Symbols:
    *  :  free location
    -  :  predefined or pragma variable
    =  :  local variable(s)
    .  :  global variable
```

Bitvariabler och Binära konstanter

Cc5x har bitvariabler som tillägg till ANSI-C. Så här deklarerar man en bitvariabel:

```
bit IOvalue;
#pragma bit IOpin @ PORTB.3
```

I det första fallet får kompilatorn placera bitvariabeln `IOvalue` i någon ledig minnescell (8 bitvariabler får rum i en minnescell). I det andra fallet tilldelas bitvariabeln `IOpin` en fast plats i registerarean, denna gång portpinne B3.

Vill man hänvisa till bitposition "4" i en variabel `TALET` kan man skriva `TALET.4`.

ANSI-C ger möjligheten att skriva konstanter på hexadecimal form, tex `0xF8`. Cc5x ger dessutom möjlighet att skriva konstanter på binär form `0b11111000`. Det är också tillåtet att gruppera bitarna genom att inför punkter `0b111.1.1000`. Punkterna har ingen betydelse men kan underlätta för "ögat" att hitta i bitföljden. (Jämför med hur ekonomer brukar inskjuta punkter mellan siffergrupper om tre när de anger mycket stora belopp).

Bitvariabler kan även vara returvärde till funktioner, det vill säga en funktion kan vara av typen `bit`.

Cc5x interna variabler och funktioner

Cc5x interna variabler

Inbygt i kompilatorn finns följande namn på register och flaggor (= bitar i register):

```
char W;
char INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB;
char OPTION, TRISA, TRISB;
/* STATUS : */   bit Carry, DC, Zero_, PD, TO, PA0, PA1, PA2;
/* FSR : */      bit FSR_5, FSR_6;
char PORTC, TRISC;
char PCLATH, INTCON;
/* OPTION : */   bit PS0, PS1, PS2, PSA, T0SE, T0CS, INTEDG, RBPU_;
/* STATUS : */   bit Carry, DC, Zero_, PD, TO, RP0, RP1, IRP;
/* INTCON : */   bit RBIF, INTF, T0IF, RBIE, INTE, T0IE, GIE;
```

Man ska således *inte* deklarera dessa i sitt program.
För varje processortyp finns det dessutom en **headerfil** där ytterligare register och deras bitar (flaggor) deklareras.

Om man alltid tar med en hänvisning till den aktuella processorns headerfil i sina program, så förstår Cc5x vad som menas med register och flaggor enligt de beteckningar som förekommer i Microchips manualer.

```
#include "16F84.h"
/* innehållet i filen 16F84.h i arbetsbiblioteket tas med */
#include <16F84.h>
/* innehållet i filen 16F84.h i programbiblioteket tas med */
```

Cc5x interna funktioner

Cc5x har många interna funktioner, som direkt utnyttjar användbara maskininstruktioner. Några av dem presenteras nedan.

`clearRAM(); /* void clearRAM( void ) */`
0-ställer processorns *alla* RAM-register. Denna funktion kan man använda i början av sitt program för att initiera sina variabler eftersom Cc5x inte alltid tillåter direkt initiering,.

`void nop(void);`
genererar direkt maskinkodsinstruktionen `NOP` (**N**o **O**peration) som ger en clockcykels fördröjning.

`void nop2(void);`
innebär hopp till nästföljande instruktion, och det blir en fördröjning på 2 klockcykler.

`i = rl(i); /* char rl(char); */` "roterar" talet `i` ett steg åt vänster.
`i = rr(i); /* char rr(char); */` "roterar" talet `i` ett steg åt höger.

`skip(i); /* void skip(char); */` hoppar över `i` stycken C-instruktioner. Funktionen har tillkommit för att göra det möjligt att använda "computed goto", PIC-processorernas teknik för indexerade tabeller .