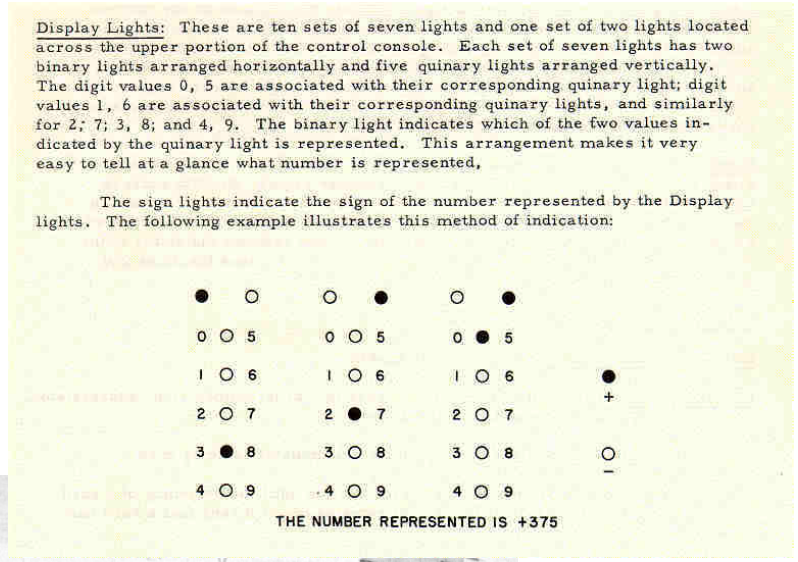
Maskinnära programmering i C

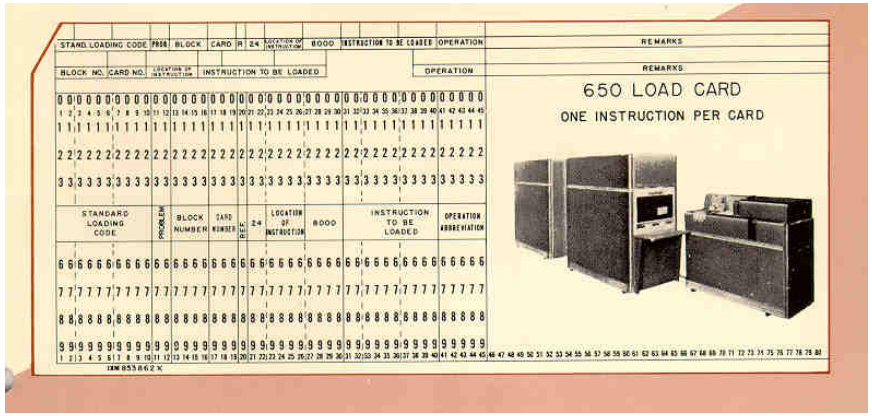
- Snabb historik
- Korsutveckling
- Nödvändiga programbibliotek
- Inbäddad assemblerkod
- Avbrott

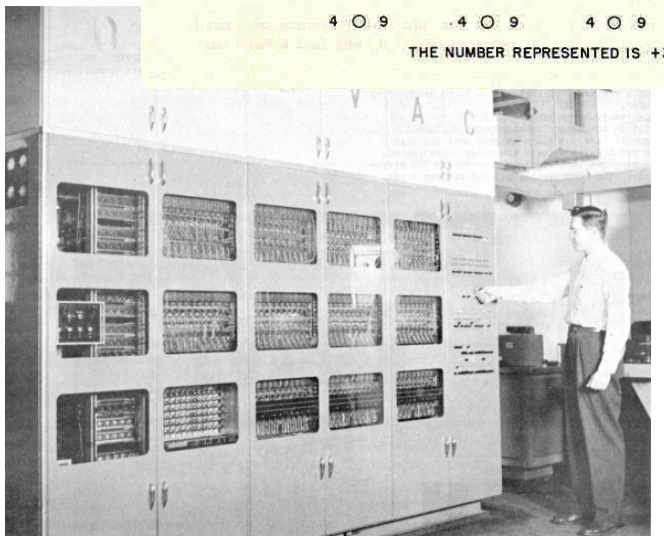


Display Lights: These are ten sets of seven lights and one set of two lights located across the upper portion of the control console. Each set of seven lights has two binary lights arranged horizontally and five quinary lights arranged vertically. The digit values 0, 5 are associated with their corresponding quinary light; digit values 1, 6 are associated with their corresponding quinary lights, and similarly for 2; 7; 3, 8; and 4, 9. The binary light indicates which of the two values indicated by the quinary light is represented. This arrangement makes it very easy to tell at a glance what number is represented,

The sign lights indicate the sign of the number represented by the Display lights. The following example illustrates this method of indication:

THE NUMBER REPRESENTED IS +375





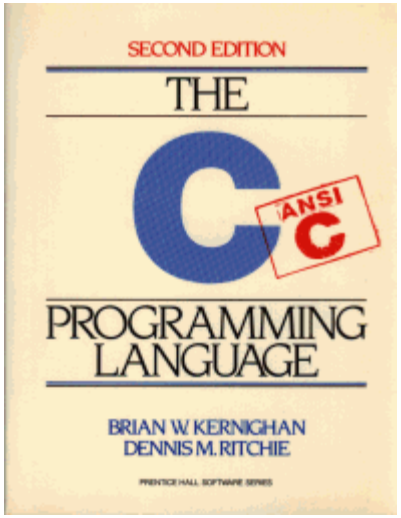
Programspråket ’C’

”The philosophy of BCPL is not one of the tyrant who thinks he knows best and lays down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, ***even when confronted with apparent nonsense****.* ***The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions.”***

Martin Richards

Dennis Ritchie

Brian Kernighan



’ BCPL’ Basic Combined Programming Language – (Martin Richards) 1966

’B’ - (Johnson/Kernighan) 1973

’C’ – (Kernighan/Ritchie) 1978

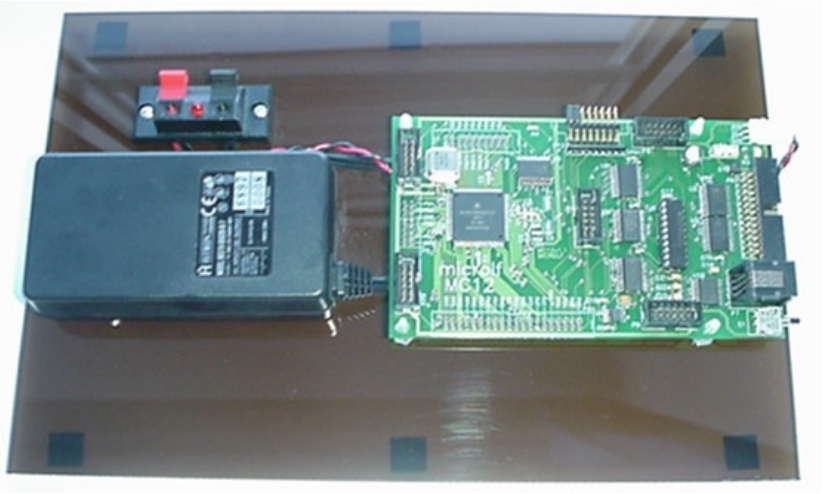
’ANSI C’ – 1983, första standardisering

’C++’ – (Stroustrup) 1986

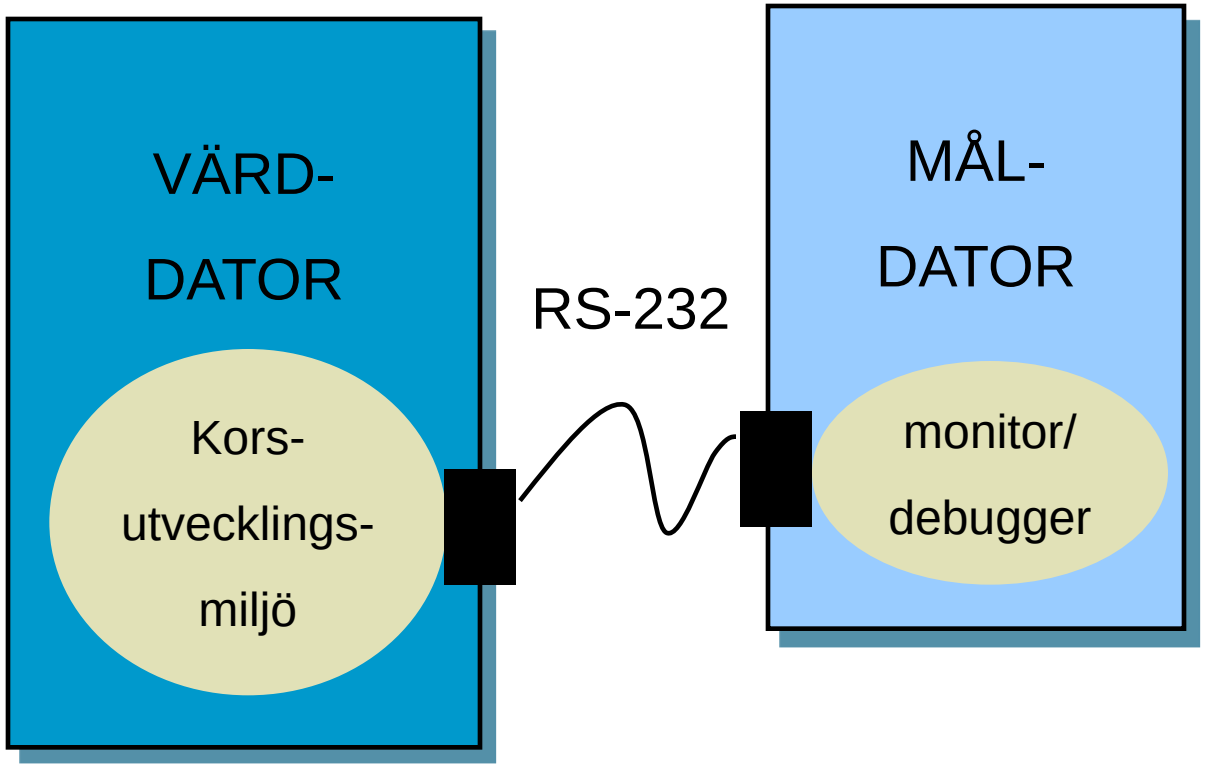
’ISO’ – 1995, 1999, ..., 2011

Korsutveckling

- Utveckling *för* en typ av maskin (*måldator*) med hjälp av *en annan typ* av maskin (*värddator*)
- Korsutvecklingsverktyg:
  - Korsassemblator
  - Korskompilator
  - …



Valet av utvecklingsmiljö



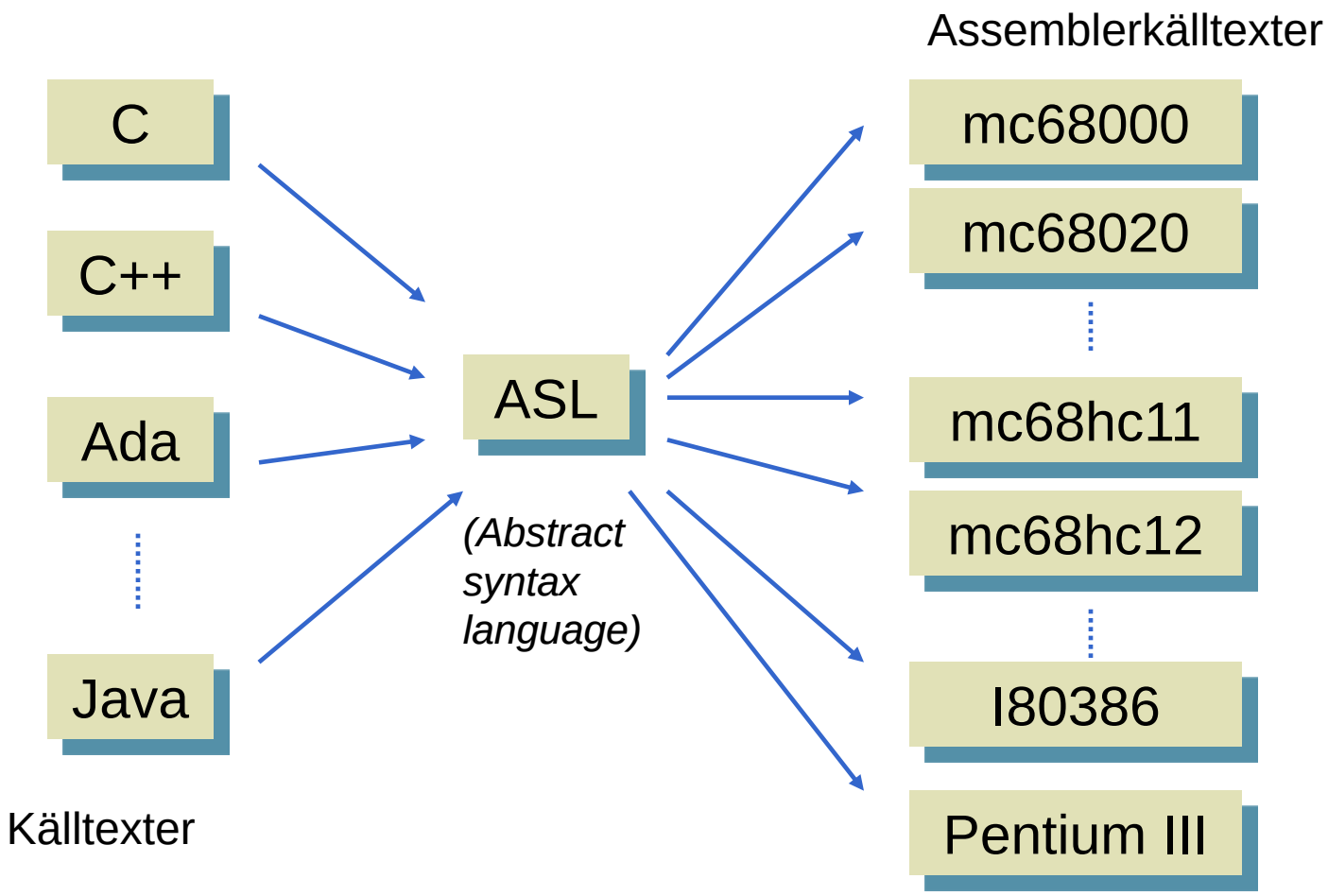
- syfte
- tillgång
- erfarenheter/ kunskap
- ekonomi

Korsutvecklingsmiljöns komponenter

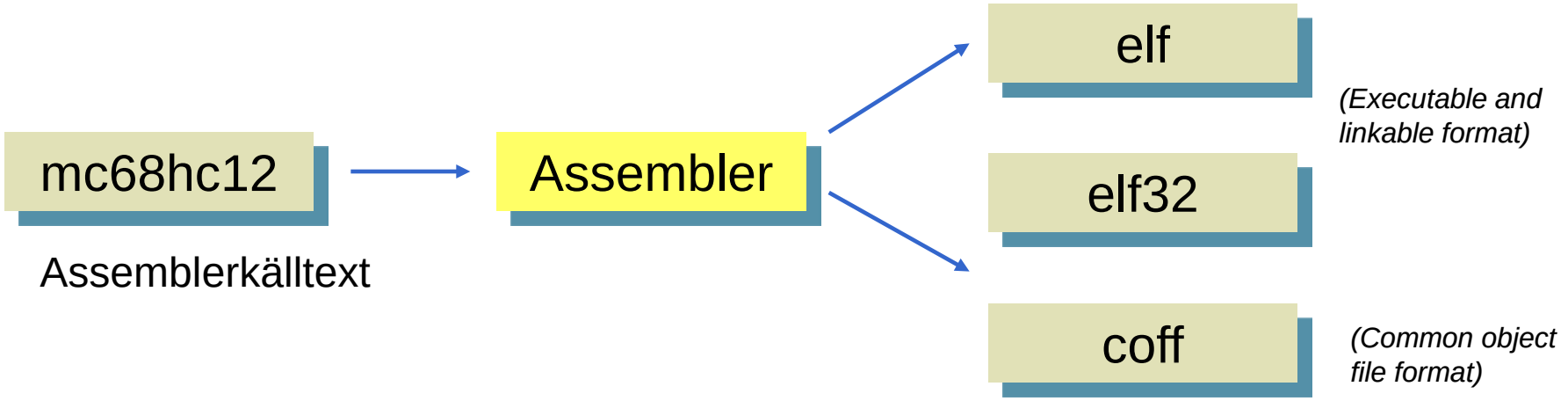
Värdsystemet
- Kompilatorer (C/C++/Java/Ada/Fortran...)
- Assemblatorer
- Länkare
- Terminalemulator
- Simulatorer

Målsystemet
- Processorfamilj
- IO-enheter
- Monitor/debugger
- realtidskärnor
- programbibliotek

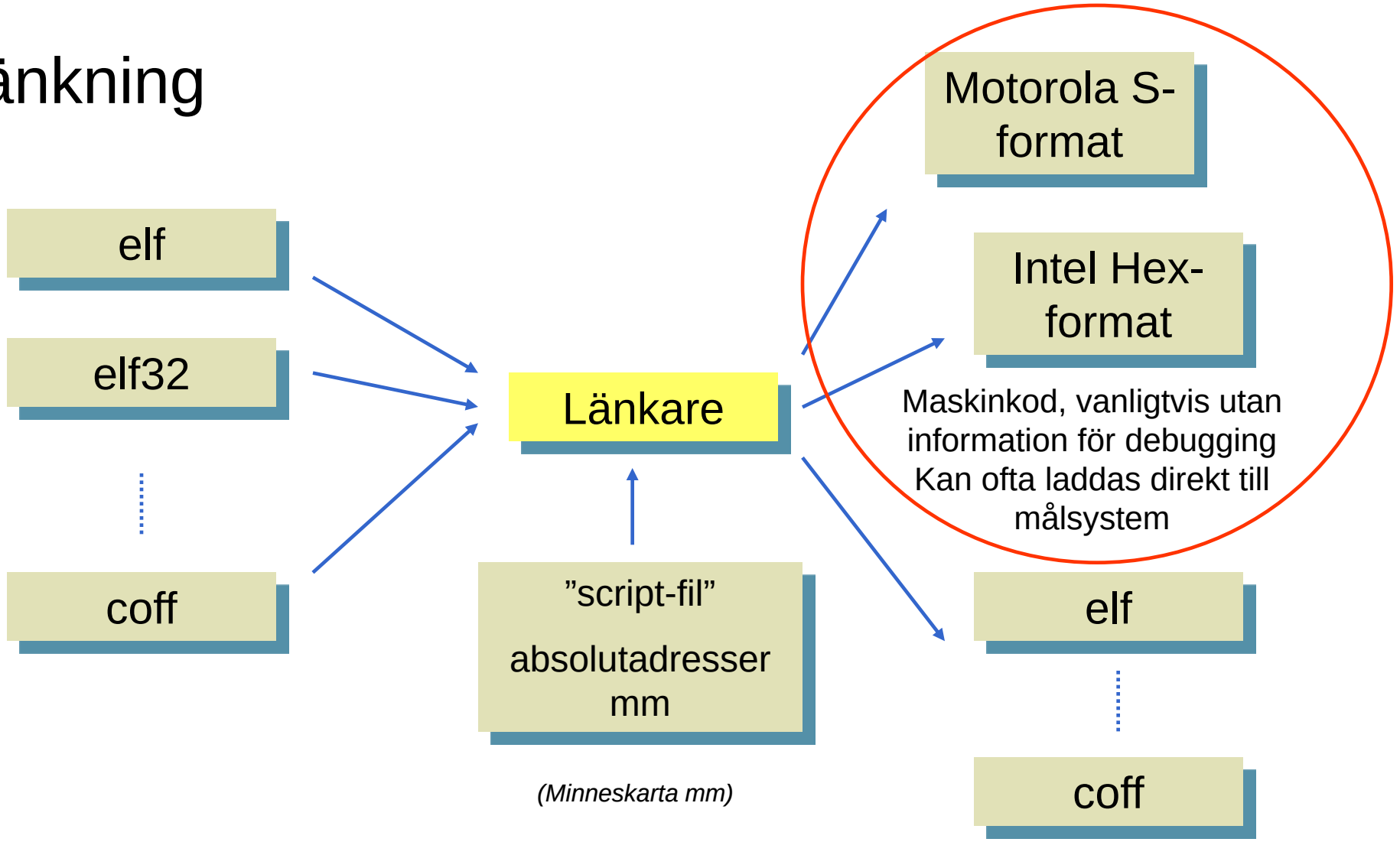
Kompilatorns arbetssätt



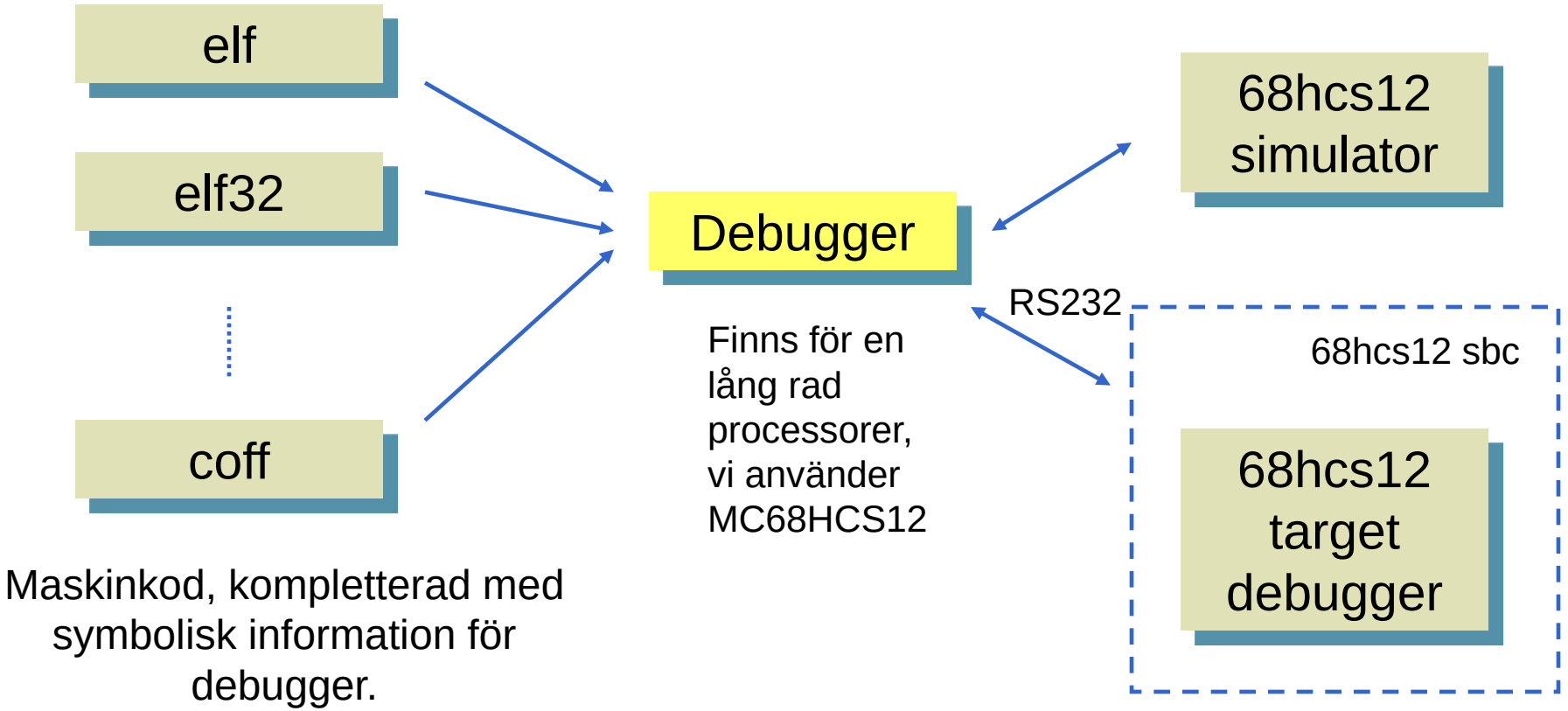
Assemblatorns arbetssätt



Länkning


Motorola S-format
elf
Intel Hex-format
elf32
Länkare
Maskinkod, vanligtvis utan information för debugging
Kan ofta laddas direkt till målsystem
coff
”script-fil”
absolutadresser
mm
elf
(Minneskarta mm)
coff

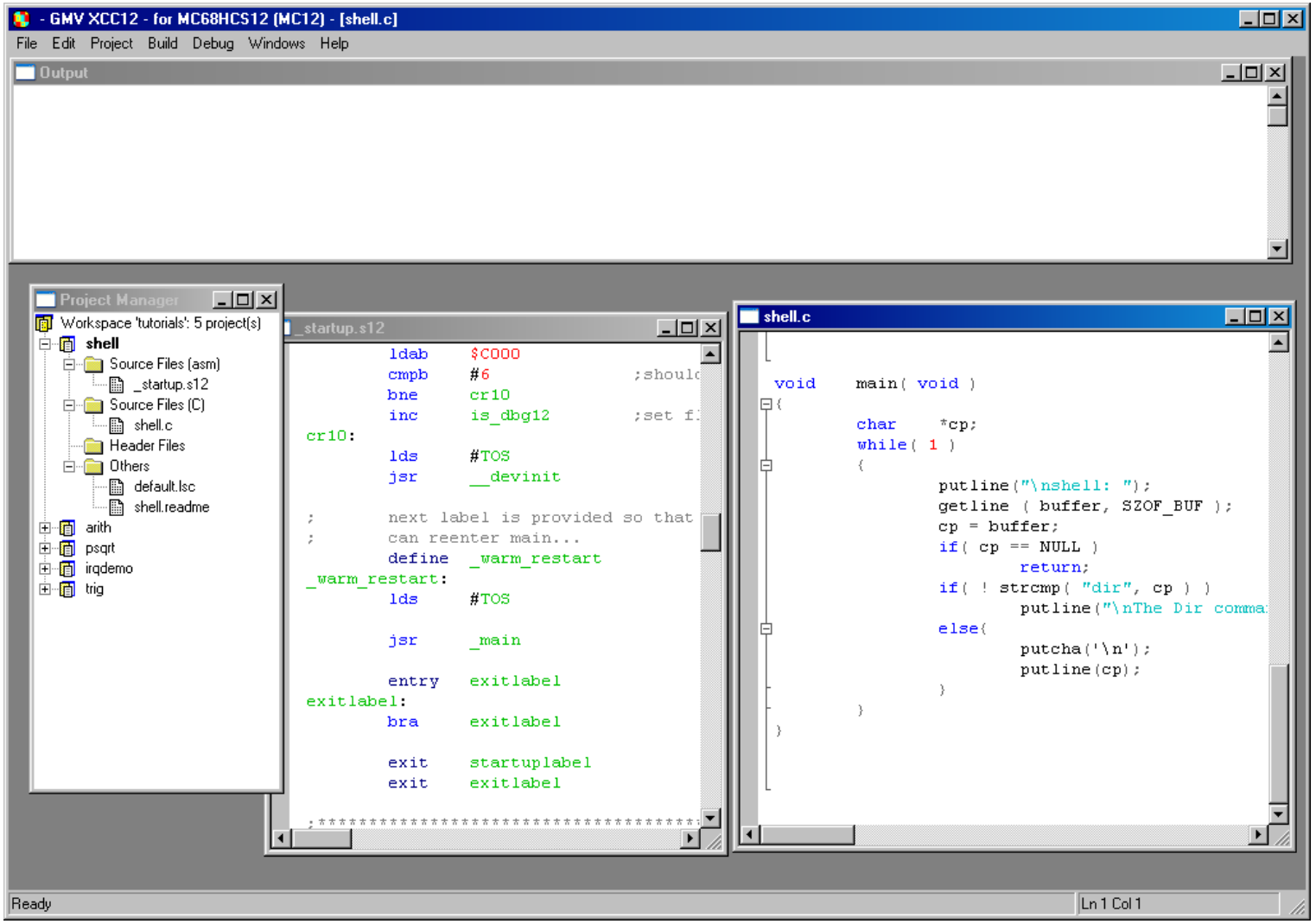
Källtextdebugger



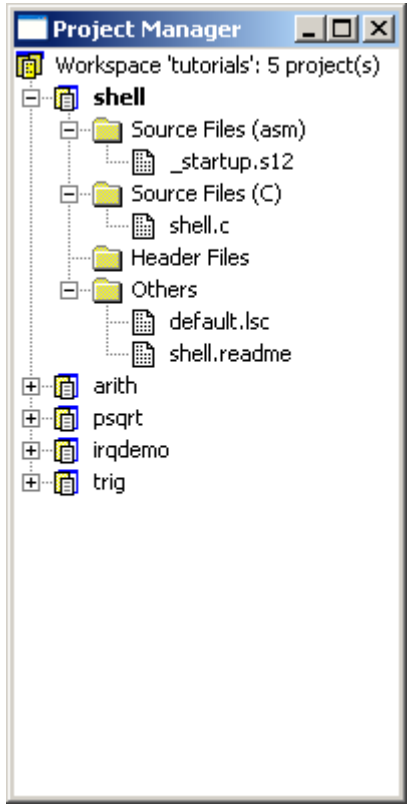
Maskinkod, kompletterad med symbolisk information för debugger.

Finns för en lång rad processorer, vi använder MC68HCS12

Programutveckling med XCC12



XCC12 ’Project Manager’



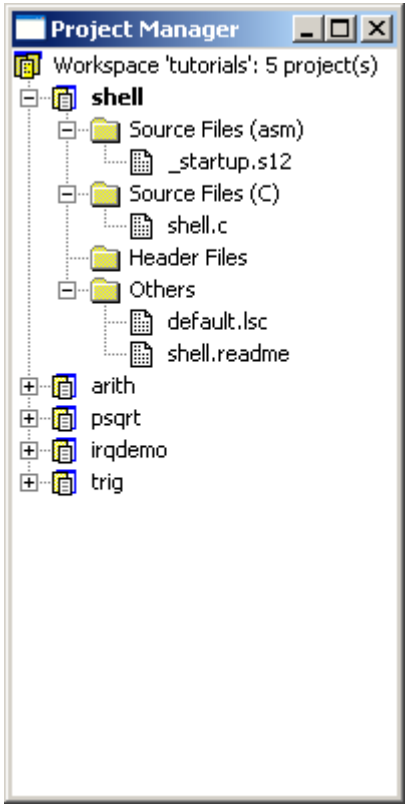
’Project’

Alla källtexter som berör en enskild tillämpning.
Resulterar i ett exekverbart program

’Workspace’

Ett praktiskt sätt att organisera flera relaterade tillämpningar

XCC12 ’Applikation’



Applikationen, dvs det speciella programmet kräver sällan speciell ’startup’- procedur. Därför finns en ”standard” startup...

”startup” procedur behandlas i ARB1, sid 72-73.

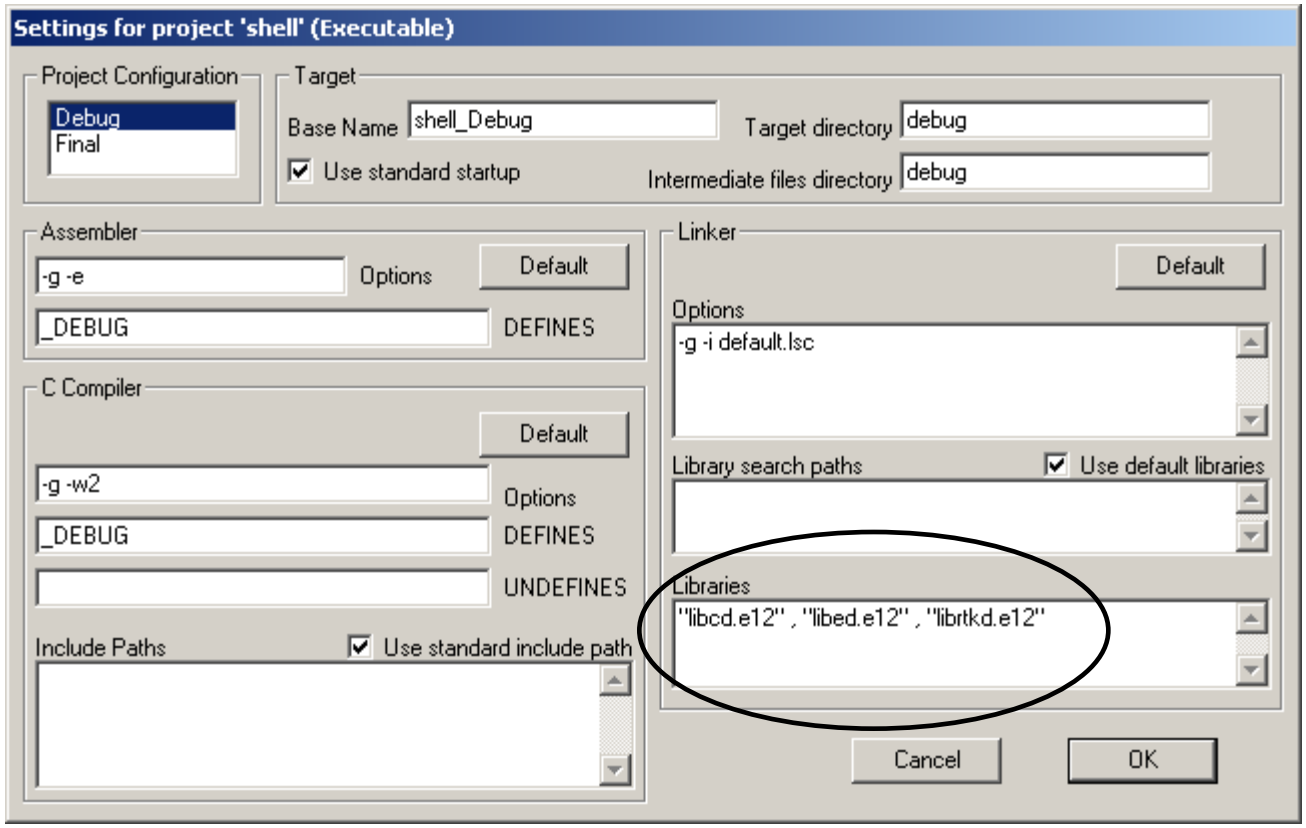
XCC12 ’Kompilatorbibliotek’

Programbibliotek för att utföra standardoperationer som inte enkelt kan utföras av hårdvaran (CPU12). Exempelvis addition av 32-bitars tal...

```
* long int la,lb,lc;
*   lc = la + lb;
  ldd   2+_lb
  ldx   _lb
  pshd
  pshx
  ldd   2+_la
  ldx   _la
  pshd
  pshx
  jsr   add32
  leas  8,sp
  std   2+_lc
  stx   _lc
```

’add32’ är en färdig funktion som ingår i kompilatorbiblioteket

XCC12 ’Standardbibliotek’



Olika programbibliotek innehåller färdiga program för funktioner som används ofta .

CC12 ’segment’

Beroende på källtext placeras assemblerkoden i olika *segment*. CC12 använder fem olika segment:

`init` – här placeras startsegmentet (prefix) (programkod).

`text` – här reserveras plats för maskininstruktioner (programkod).

`rodata` – här reserveras plats för data som från början har definierade värden (initierade variabler) men som *inte* kan komma att ändras under programexekvering.

`data` – här reserveras plats för data som från början har definierade värden (initierade variabler) men som kan komma att ändras under programexekvering.

`bss` – här reserveras plats för data som från början har odefinierade värden

CC12 'segment' - Exempel

```
int   var;
```

```
segment  bss
export   _var
_var:    rmb 2
```

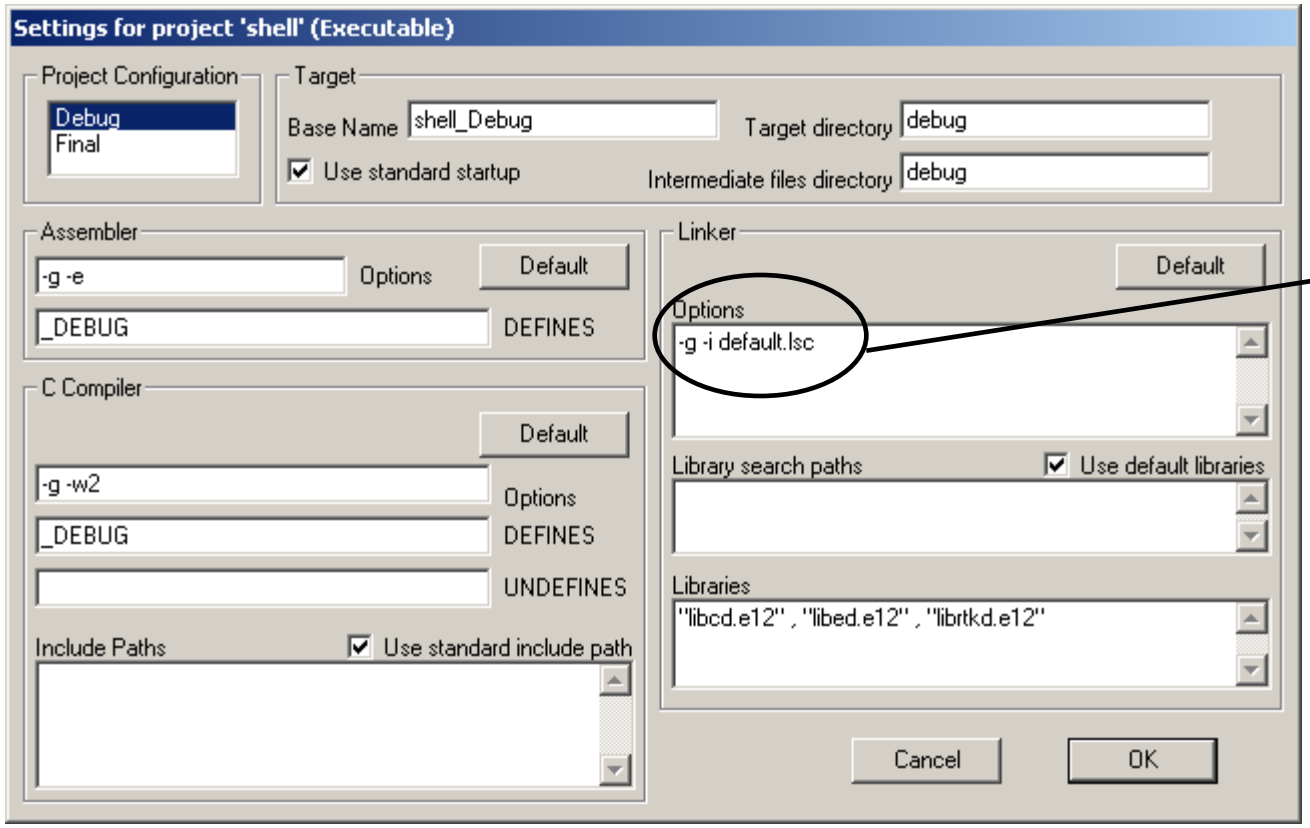
```
const int novar = 1;
```

```
segment  rodata
export   _novar
_novar:  fdb $1
```

```
int   init_var = 2;
```

```
segment     data
export      _init_var
_init_var:  fdb $2
```

XCC12 instruktioner för länkaren




```
//
//   default.lsc
//   script for QLD
//   for XCC12 applications in RWM
//
...
```

XCC12 ’default.lsc’

```
//          OPTIONS SECTION
            -M          // generate listfile <basename>.map

//          define program entry for debugger
            entry( __start )

            group  ( c , const_group )
            {
                        abs
            }
            group( r , test_group)
            {
                        init,
                        text,
                        cdata,
                        data,
                        bss
            }
            group( r, interrupt_vectors )
            {
                        vectors
            }
            layout
            {
                        0x1000,0x3C80 <= test_group,
                        0x3F80,0x3FFF <= interrupt_vectors
            }
```

XCC12 Inbäddad assemblerkod

För operationer som inte kan utföras i ’C’ kan man använda ”inbäddad assemblerkod.

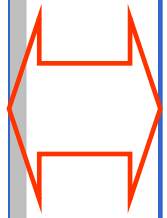
```
/* EXEMPEL */
void    main( void )
{
        __asm( " andcc #$F0"); /* nollställ flaggor i CC */
}
```

OBSERVERA: Detta är INTE en del av ’C’ och användningen är alltså kompilatorberoende...

XCC12 Inbäddad assemblerkod

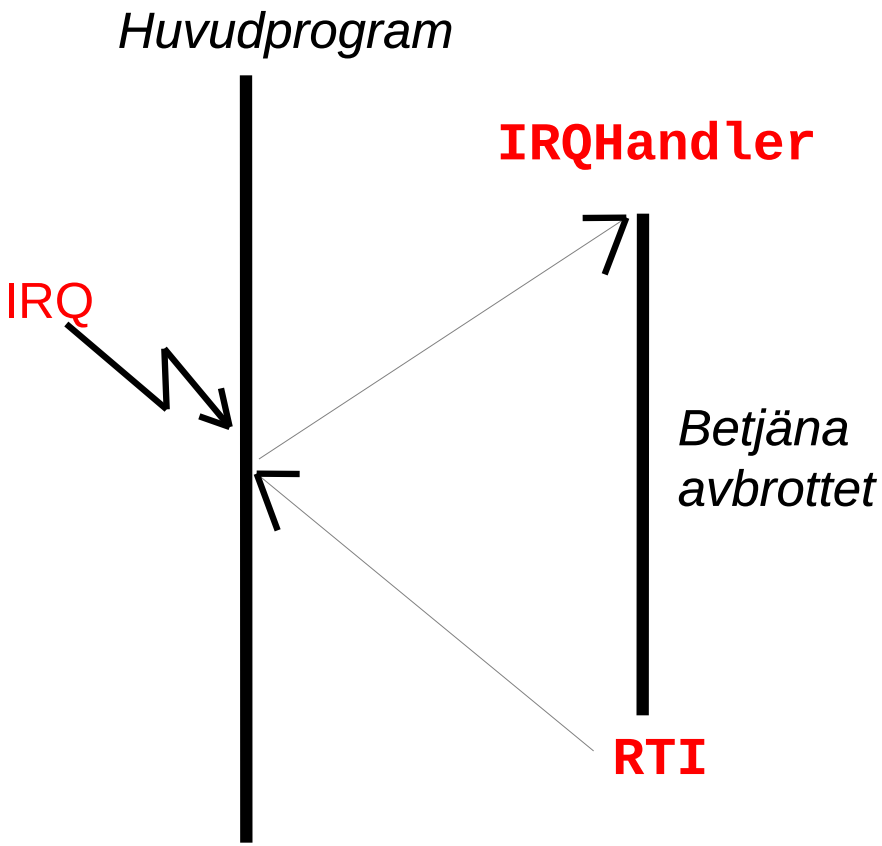
CC12 tillåter också att variabler och parametrar refereras på ett enkelt sätt...

```
void callfunc( int aa , int ab )
{
   aa = 1;
   ab = 2;
}
```



```
void callfunc( int aa , int ab )
{
        __asm(" movw #1,%a", aa);
        __asm(" movw #2,%a", ab);
}
```

DBG12 och avbrott



ROM

Adress (hex)	Funktion
FFFE	RESET, Startvektor
FFFC	Clock Monitor Fail, JMP [3FFC]
FFFA	COP Watchdog Timeout, JMP [3FFA]
FFF8	Illegal Op Code, JMP [3FF8]
FFF6	SWI, JMP [3FF6]
FFF4	XIRQ, JMP [3FF4]
FFF2	IRQ, JMP [3FF2]
FFF0 FF8C	Enhetsspecifika vektorer JMP [3Fxx]

RWM

Adress (hex)	Funktion
3FFE	Används ej
3FFC	ClockFailHandler
3FFA	COPFailHandler
3FF8	IllOpHandler
3FF6	SWIHandler
3FF4	XIRQHandler
3FF2	IRQHandler
3FF0 3F8C	Enhetsspecifika vektorer

Specifikation av avbrottsrutin

```
__interrupt void rutinens_namn( void );
```

Keyword ”__interrupt” måste inleda specifikationen

Inga returvärden eller parametrar är tillåtna, ange `void` för dessa.

Avbrottsvektor med XCC12

```
__interrupt void IRQHandler( void )
{
        /* avbrottsrutin */
}

void main(void)
{
        *(unsigned short *) 0x3FF2 = (unsigned short) IRQHandler;
        ...
}
```

alternativt...

```
#define  SET_IRQ_VECTOR(x,y) *(unsigned short *) y = (unsigned short) x

void main(void)
{
        SET_IRQ_VECTOR( IRQHandler , 0x3FF2 ) ;
        ...
}
```