Stefan Nyman

Bygg och programmera med enchipsdatorn 68HC11

MICRONYM

27 november 2002

INNEHÅLLSFÖRTECKNING

Figurer

Förord

Detta kompendium kan användas som ersättning och komplement till de databöcker som behandlar enchipsdatorn 68HC11.

Efter ett inledande kapitel om grundläggande datorteknik följer en noggrann genomgång (kap 3) av HC11:ans instruktionsrepertoar. Även om man inte tänker skriva sina program i assemblerspråk, är det viktigt att förstå hur instruktionerna fungerar.

I de konstruktioner där enchipsdatorer ingår, har avbrottsrutiner en central roll. Kapitel 4 tar upp det viktigaste om avbrott. Alla programexempel som ges här, är i assemblerspråk.

Kapitel 5 ägnas åt de omfattande periferienheterna. Programexemplen är i assembler och i högnivåspråket C.

Kapitel 6 och 7 handlar om HC11 mer i detalj. Här förklaras de olika typerna av reset och de inbyggda skyddsmekanismerna och de olika arbetssätten.

Kapitel 8 förklarar de grundläggande begreppen i C-programmering. Det är ingen fullständig beskrivning av C, men är en bra början för den som vill programmera enchipsdatorn i C.

Resten av kompendiet vänder sig speciellt till konstruktören och innehåller en mängd tips och idéer för det verkliga konstruktionsbygget.

Observera att detta kompendium inte är en fullständig beskrivning av 68HC11. Många detaljer är medvetet utelämnade. Vill man ha mer detaljerade förklaringar, får man vända sig till den tekniska manualen och referensmanualen som innehåller allt man kan tänkas behöva veta.

Kapitlet om C-programmering är heller ingen fullständig beskrivning av C-språket. En bok som rekommenderas är Bilting/Skansholm: Vägen till C (ISBN 91-44-26732-0).

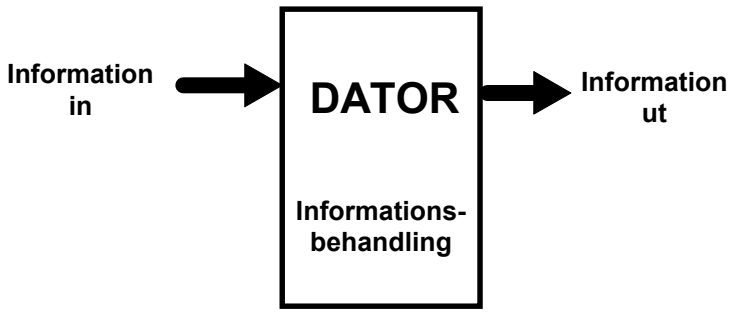
Detta kompendium skrevs ursprungligen som ett läromedel för skolemulatorn EM-11. Erfarenheter från gymnasieskolor och ingenjörsutbildningar har genom åren gett värdefulla tillskott till innehållet.

Lund i januari 2001

Stefan Nyman

1 Hur en dator fungerar.

För att förstå hur en dator fungerar, kan vi titta på en mycket förenklad modell av en dator, se figuren nedan:



Figur 1. Enkel datormodell.

Man kan säga att det är denna modell vi ser dagligen omkring oss. Med andra ord ser vi alltså inte särskilt mycket av själva datorn, utan märker istället att den gör något.

Ta exempelvis en modern bilradio med RDS. Vi knappar in önskad station, och mikrodatorn ser till att högtalarna så bra som möjligt återger det vi önskar.

Vad som finns inuti behöver inte den vanlige radiolyssnaren veta.

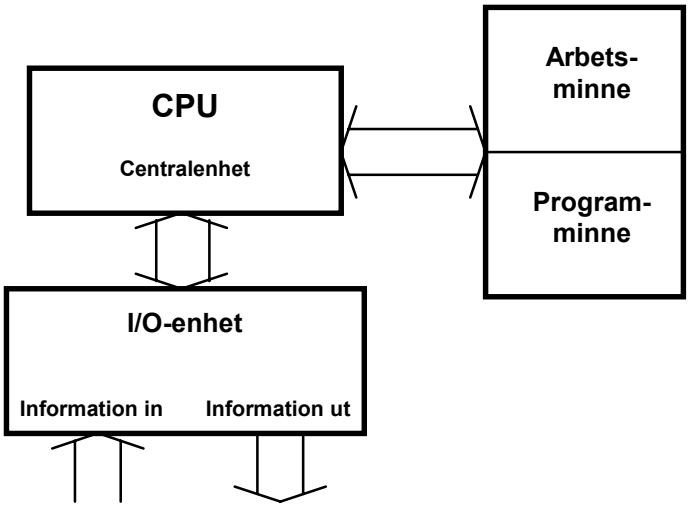
En dator kan beskrivas som en maskin, oftast byggd med elektronik, som kan ta emot information, behandla den och sedan ge ifrån sig annan information. Informationsbehandlingen bör vara någorlunda intelligent, dvs. datorn ska naturligtvis utföra något nyttigt. Kännetecknande för en dator är att informationsbehandlingen styrs av ett **program** och just detta är till stor fördel när maskinen ska konstrueras.

Om vi nu tar en noggrannare titt på innehållet i en dator, figur 2, så kan vi göra en naturlig uppdelning i tre delar: In- och utenhet, centralenhet,CPU, och minne.

- In- och utenheten, som i figuren betecknas I/O-enhet (eng.: In/ Out), är datorns skal mot omvärlden, och det är detta vi kommer i kontakt med när vi använder datorbestyckade konstruktioner.

- CPU:n eller centralenheten innehåller själva 'hjärnan' i datorn, och det är här som det intelligenta arbetet utförs. CPU är förkortning för Central Processing Unit. Vi kommer i detta kapitel att noga ägna oss åt denna del.

- Minnet är uppdelat i ett programminne och ett s.k. arbetsminne. I programminnet finns en arbetsbeskrivning för CPU:n. Arbetsbeskrivningen är uppbyggd av **instruktioner**, som gör att datorn kan behandla den inkommande informationen steg för steg och ge ifrån sig önskade utsignaler. Arbetsminnet är ett minne för både skrivning och läsning och används av programmet bland annat för lagring av insamlade och beräknade värden.



Figur 2. Datormodell.

Lådan i fig.1 skulle ju egentligen kunna innehålla något helt annat än en dator. För mycket länge sedan (40-50 år) byggde man komplicerade styrsystem med reläer och transistorer som ibland kunde utföra nog så intelligent arbete. Men med hjälp av datortekniken kan vi nu istället göra en **arbetsbeskrivning** av vad som ska utföras.

CPU:n och minnet kan vara gemensamma för många vitt skilda konstruktioner. Detta innebär att en och samma dator kan utföra många olika uppgifter.

Det är I/O-enheterna som blir speciellt utformade i varje enskild konstruktion så deras uppbyggnad behöver vi inte ägna oss åt för att förstå datorns funktion.

Vad måste då CPU:n innehålla för att den ska kunna göra datorn till en användbar maskin?

Om man ger den förmågan att göra någon sorts beräkningar och jämförelser och sedan utifrån dessa kan fatta beslut genom att kunna välja bland alternativ, så bör man kunna få något uträttat.

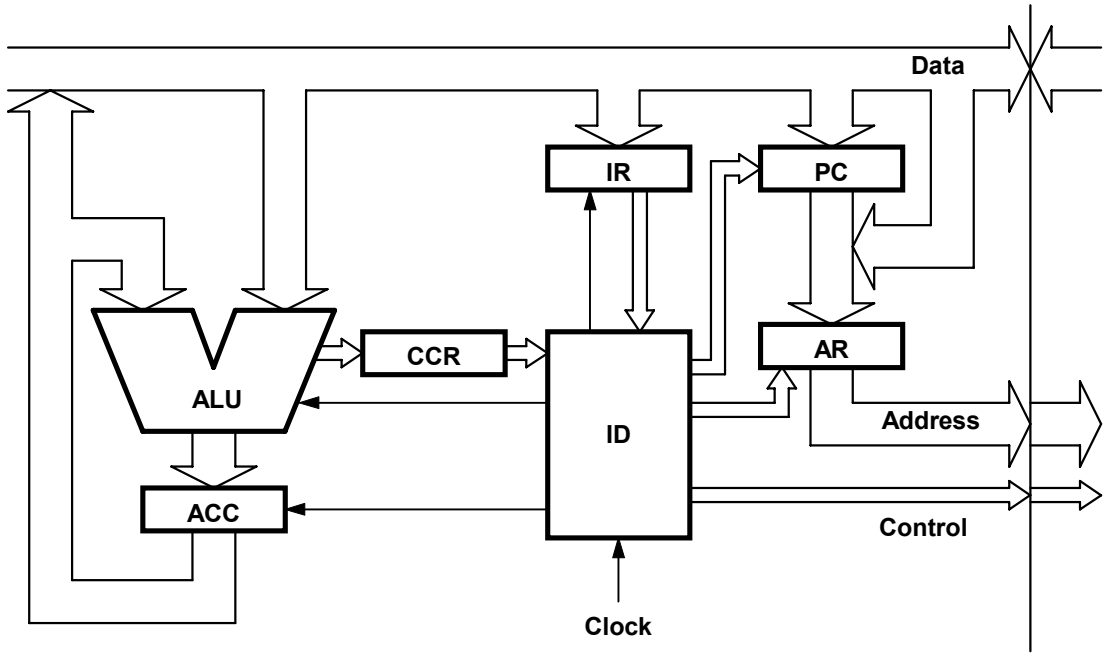
Den måste naturligtvis också kunna läsa de instruktioner man givit den.

De två viktigaste delarna är därför en beräkningsenhet (aritmetisk/logisk enhet) och en instruktionsavkodare.

Vi ska nu se hur dessa delar kan vara uppbyggda.

1.1 CPU.

Datorns arbete sköts av CPU:n som innehåller ett antal urskiljbara delar:



Figur 3. CPU:ns viktigaste delar.

Bilden ovan visar hur en enkel CPU kan vara uppbyggd. Normalt är den mer komplex, men vi kan betrakta de avbildade delarna som den 'obligatoriska' utrustningen.

- **PC**, Program Counter. Programräknare.

Programräknaren används till att peka ut instruktion för instruktion i det program som gås igenom. Detta sker genom att programräknarens innehåll läggs ut på adressbussen, som är kopplade till programminnet.

- **IR**, Instruction Register. Instruktionsregister.

Här hamnar koden som bestämmer vad varje enskild instruktion ska utföra. Till instruktionen hör ibland en adress eller ett värde. I så fall laddas de in i AR eller ALU (se nedan). Varje instruktion initierar en sekvens av händelser i instruktionsavkodaren.

- **ID**, Instruction Decoder. Instruktionsavkodare.

En kod från IR startar en serie händelser som styr vad varje instruktion ska utföra. Denna kod läses in från programminnet via databussen. Dessa händelser skapas genom att instruktionsavkodaren är byggd som en sekvensmaskin med ett antal insignaler och ett antal utsignaler. Koden från IR kan alltså kombineras med exempelvis information från flaggregistret,

CCR, för att kunna utföra ett villkorligt hopp. Vi ser att styrsignaler går från instruktionsavkodaren till andra delar av CPU:n. Klocksignalen ser till att sekvensmaskinen uppdaterar sitt tillstånd med en viss frekvens. När instruktionen är utförd, hämtas automatiskt nästa, varefter denna utförs. En dator arbetar alltså i 2-takt: hämta, utför, hämta, utför osv.

- **ALU**, Arithmetic Logic Unit. Aritmetisk/logisk enhet.

Här sker datorns beräkningsarbete på det sätt som instruktionsavkodaren bestämmer. Det kan tex. vara en subtraktion eller en logisk ellerfunktion. Beräkning sker alltid mellan värdet i ackumulatorn (ACC) och ett värde i minnet.

- **ACC**, Accumulator. Ackumulator.

Resultatregister för ALU:n. Det uträknade värdet hamnar här.

- **CCR**, Condition Code Register. Flaggregister.

Flaggregistret avspeglar resultatet så att tex. likhet kan detekteras. Tillsammans med signaler från instruktionsregistret styr de instruktionsavkodaren.

- **AR**, Address Register. Adressregister.

Detta register innehåller en adress som pekar ut en speciell plats i det utanförliggande minnet. Adressen kommer antingen från PC (om den ska peka ut en instruktion) eller laddas in från databussen (om den ska peka ut en plats där tex. en skrivning ska ske).

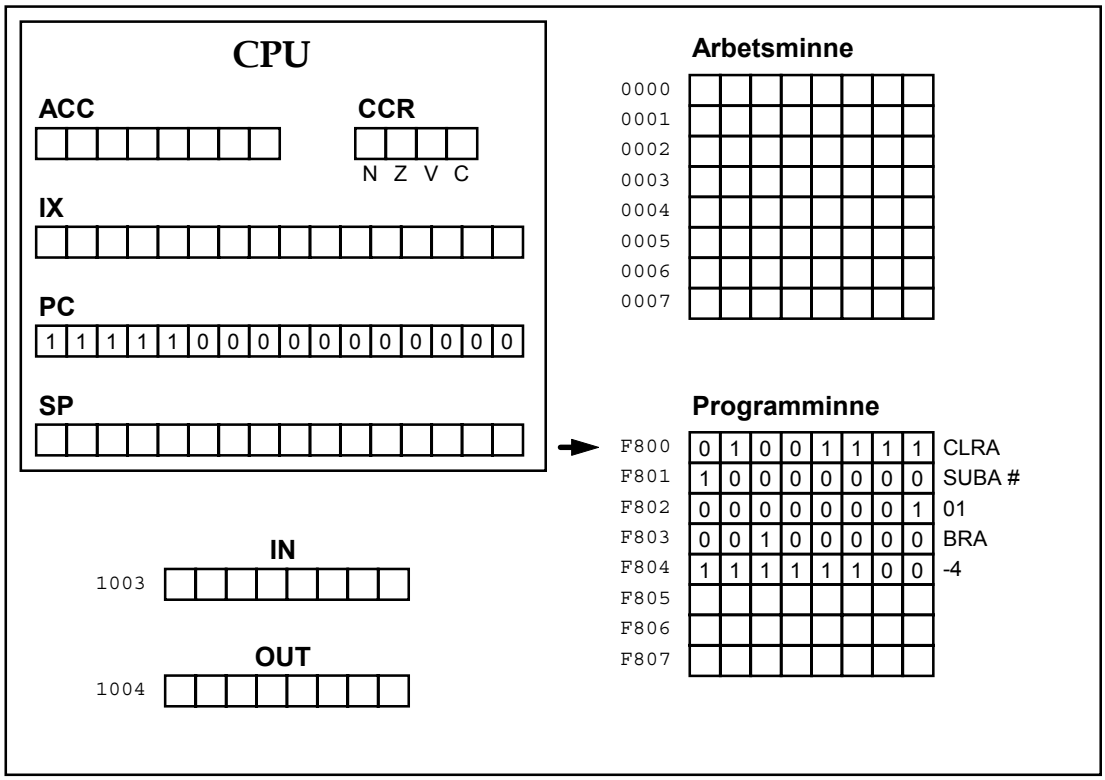
Om vi 'gömmer undan' alla delar utom ackumulatorn, flaggregistret och programräknaren, får vi kvar de delar man brukar kalla programmerarmodellen av CPU:n. Programmerarmodellen är de delar som programmeraren själv kan påverka. Att ALU och instruktionsavkodare finns, är ett faktum som avspeglas i den uppsättning av instruktioner som finns; vi vet tex. att det finns en instruktion som kan subtrahera ett tal från ackumulatorn men behöver inte känna till exakt hur det går till.

1.2 Programmerarmodellen.

För att kunna förstå hur en dator arbetar, ska vi göra en speciell beskrivning av CPU:n, som vi kallar **programmerarmodellen**.

Denna modell upptar de register i CPU:n som programmeraren har viss kontroll över.

Bilden nedan visar en programmerarmodell för en tänkt s.k. minnesorienterad dator. En sådan modell använder arbetsminnet för att utföra många av sina instruktioner och innehåller ganska få register i sin CPU.



Figur 4. Programmerarmodell.

Denna modell kan ses som ett blockschema över innehållet i datorn. Varje ruta är en minnescell, en elektrisk krets som kan minnas en etta eller nolla. Minnescellerna är grupperade 4, 8 eller 16 tillsammans och dessa grupper kallas för register.

De register som är belägna i själva minnet (utanför CPU:n), har adresser som i bilden har angetts till vänster om respektive register.

Innehållet i två av registren är direkt inkopplade till datorns omvärld och kallas i vår modell därför IN och OUT.

Adresserna anges här och i fortsättningen hexadecimalt. Vi ser att varje adress består av 16 bitar och innehållet i en viss adress 8 bitar. Man säger att datorn arbetar med 16 bitars adress och 8 bitars data.

Med minnescell brukar man vanligen mena ett helt 8-bitars register eftersom det är den minsta enhet som datorn kan arbeta med. Vi använder detta språkbruk hädanefter.

Inne i datorns CPU kan beräkningar utföras, och dessa kan ge olika slags resultat.

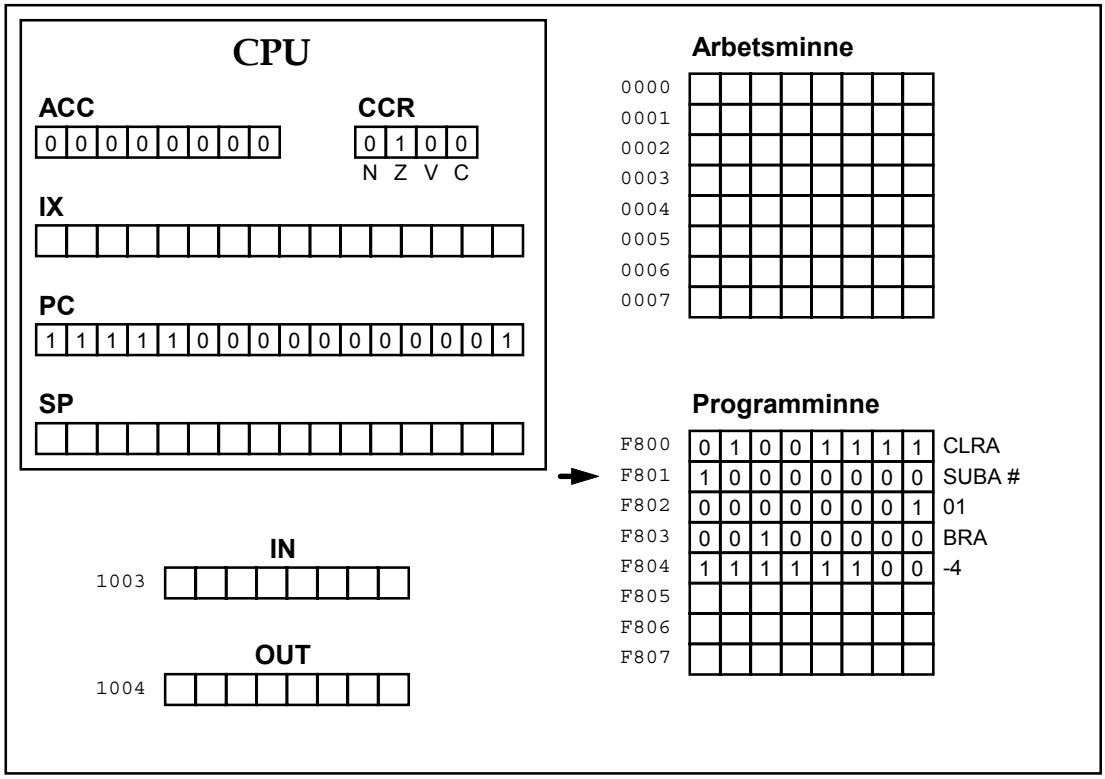
Resultaten hamnar dels i ackumulatorn (ACC) och dels i flaggregistret (CCR).

ACC (ACCumulator) innehåller det 8-bitars resultat som vissa instruktioner ger. Namnet ackumulator antyder att registret 'ackumulerar' information, dvs. varje nytt beräknat värde hamnar här och kan användas för att utföra fortsatta beräkningar.

CCR (Condition Code Register) 'flaggar' för olika sorters resultat (noll-resultat, negativt resultat mm).

Vi ska se på ett exempel:

Vi antar att datorns programräknare **PC (Program Counter)** innehåller ett värde som överensstämmer med första adressen i programminnet. Datorn hämtar nu in den kod som ligger i denna minnescell. Koden hamnar i instruktionsregistret som nu inte finns med i vår modell. För oss är det bara intressant att veta vad som händer.



Figur 5. Verkan av CLRA.

I datorn utförs instruktionen **CLRA** (Clear Accumulator) vars innebörd är att nollställa innehållet i ackumulatorn. Instruktionens kod i ettor och nollor har angetts på första raden i programminnet.

Om vi en stund ändå återvänder till modellen i figur 3.

Datorn har hämtat instruktionen CLRA som påverkar instruktionsavkodaren så att ackumulatorn nollställs.

Vi kan i detalj beskriva vad sekvensnätet ID behöver göra för att denna instruktion ska utföras. Vi förutsätter att F800 ligger i PC:

1. PC överförs till adressregistret AR.
2. En lässignal släpps på (en av signalerna märkta 'Control'). Nu finns instruktionen CLRA, 01001111, på databussen.
3. En laddningssignal skickas till IR.
4. Nu har instruktionsavkodaren fått sin instruktion. Denna kombination styr nu sekvensnätet till att skapa en nollställningssignal till ackumulatorn.
5. PC får en uppräkningssignal och nästa instruktion kan hämtas.

Efter denna instruktion har följande hänt:

Innehållet i ackumulatorn har nollställts och de fyra bitarna i flaggregistret har påverkats. Dessutom har programräknaren ökats med ett för att kunna hämta nästa instruktion.

Flaggbitarna har följande funktion:

- **N** Negative.

Indikerar om ett resultat blivit negativt. Om man använder 2-komplement för att representera negativa tal, visar denna bit om ett värde är negativt.

- **Z** Zero.

Visar att värdet i ackumulatorn blivit noll.

- **V** Overflow.

I likhet med N-biten har denna flagga endast mening om vi arbetar med positiva och negativa tal (representerade i 2-komplement). Biten indikerar att ett resultat hamnat utanför talområdet.

- **C** Carry.

Det 'normala' talområdet (endast positiva tal) har överskridits.

Det som hänt med flaggorna efter instruktionen CLRA är följande:

N = 0: Resultatet är inte negativt.

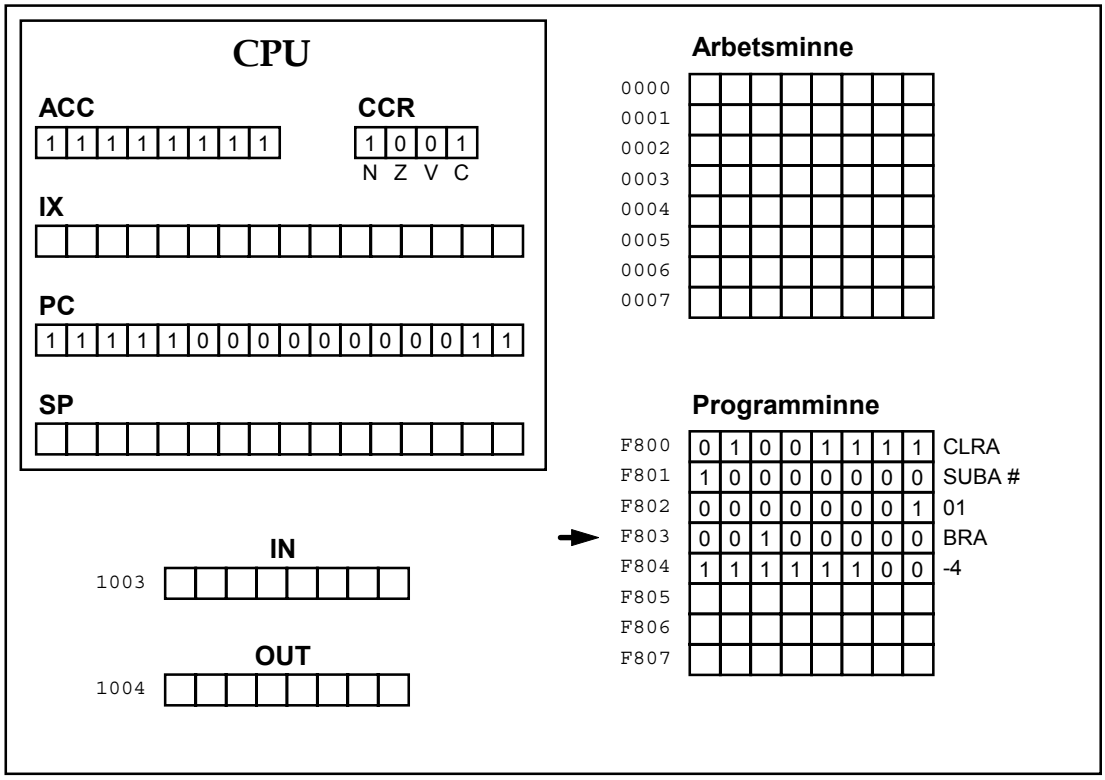
Z = 1: Resultatet är noll.

V = 0: Ingen overflow.

C = 0: Ingen carry.

Vi antar att nästa instruktion är **`SUBA #01`** med innebörden att subtrahera ackumulatorns innehåll med ett. Tecknet '#' innebär att det är **värdet 01** som ska subtraheras.
Instruktionen `SUBA #01` består av två delar: **operationen** (`SUBA`) och **operanden** (`#01`).

Man säger att `SUBA #01` är en tvåbytesinstruktion. En byte avser 8 bitar. Instruktionen `CLRA` var således en enbytesinstruktion.



Figur 6. Verkan av SUBA #01.

I detalj händer följande:

1. PC pekar på F801 och en lässignal öppnar minnet så att `10000000` kommer på databussen.
2. IR laddas och instruktionsavkodaren får besked om att nästa byte (på adress F802) ska dras ifrån ackumulatorn.
3. PC räknas upp, skickas till AR och en ny lässignal kommer. Samtidigt ställs ALU:n in på subtraktion.
4. Resultatet av subtraktionen laddas in i ACC och PC räknas upp.

Vad har nu hänt i datorn?

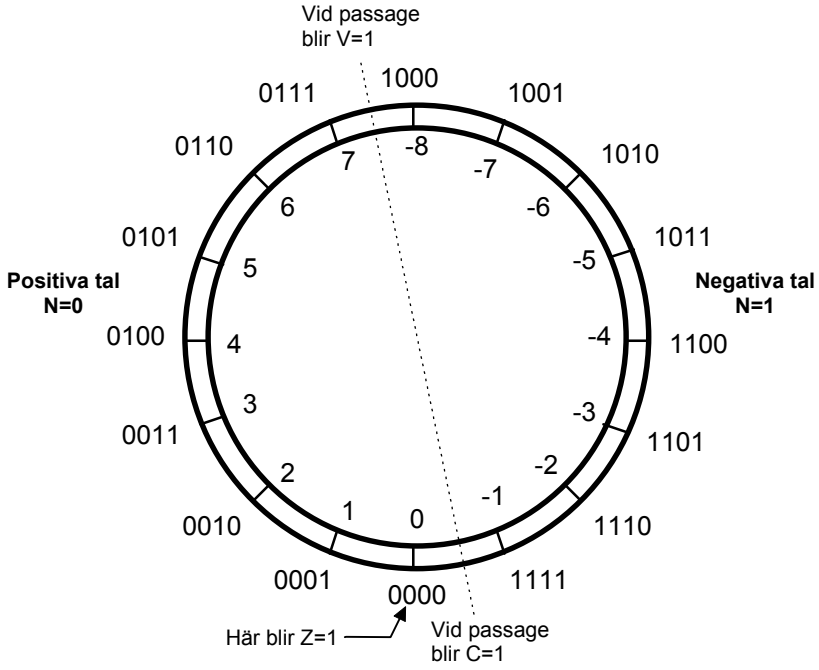
Om man drar ett ifrån noll, underskrids det normala talområdet. Tänker vi oss även negativa tal så bör vi få talet -1. Vi tittar på flaggorna och ser vad dom visar.

N = 1: Resultatet är negativt.

Z = 0: Resultatet är inte noll.

V = 0: Ingen overflow.

C = 1: Carry indikerar att vi har gått utanför det normala talområdet.



Figur 7. De fyra flaggornas funktion.

I figur 7 illustreras de fyra flaggornas funktion med hjälp av fyra bitars talområde.

Vi ser att det stämmer bra med vårt exempel. Det är viktigt att komma ihåg att det är vi själva som tolkar talen som positiva eller negativa. Flaggorna hjälper oss hela tiden, vilken tolkning vi än har.

Förutom resultatet av själva instruktionen har åter registret PC ökats, denna gång med två, för att kunna peka ut nästa instruktion.

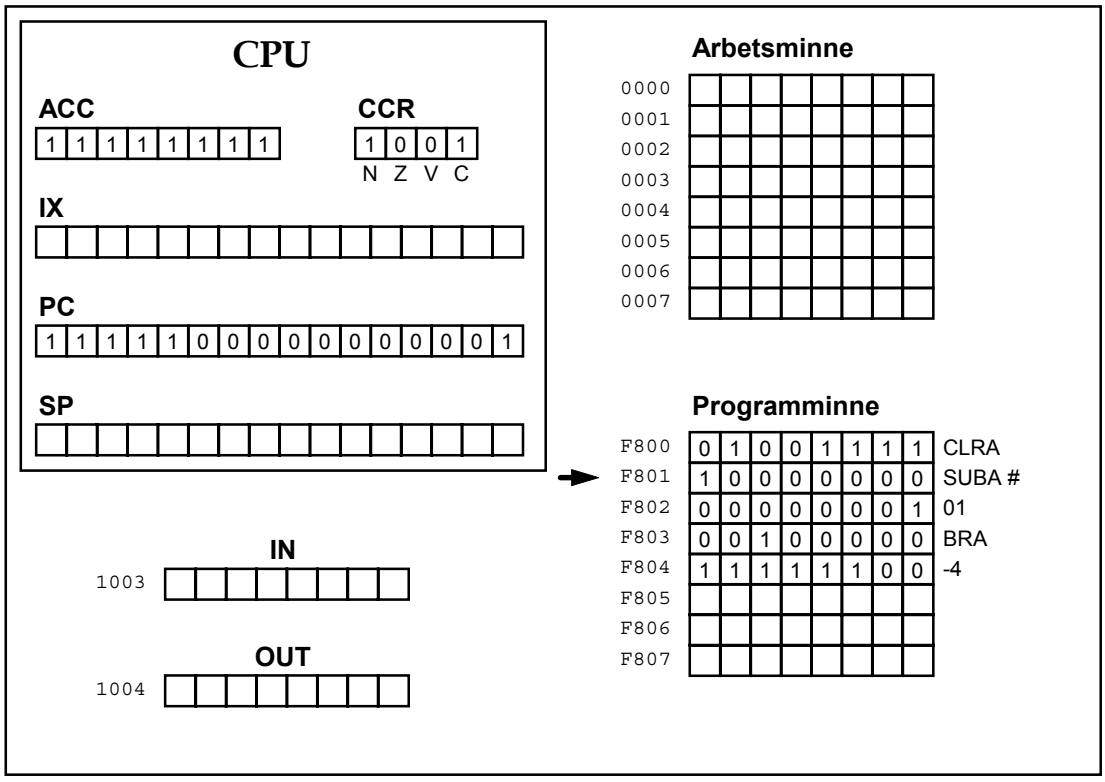
PC pekar nu på en instruktion, som har koden 20 (med hexadecimalt skrivsätt). Denna kod betyder att programräknarens värde ska ökas med det tal som följer.

Talet anges med 2-komplementrepresentation och i vårt exempel motsvarar 11111100 talet -4. Programräknaren som borde ha fått värdet F805 (för att peka på nästa lediga instruktionsplats) får nu istället värdet F801.

Allmänt gäller att varje instruktion ökar programräknaren med det antal byte som instruktionen är lång. I detta fall sker en korrigering med ett negativt tal för att få programmet att fortsätta på rätt ställe.

Denna instruktion är ett relativt hopp och förkortas **BRA** (engelska: BRANCH: sträcka sig).

I figur 8 ser vi att PC nu igen har värdet F801 (som i figur 5). Men nu har CCR det bitmönster som uppstod vid SUBA #01. Själva hoppinstruktionen påverkar inga flaggor.



Figur 8. Verkan av BRA -4.

Detta händer nu:

1. PC pekar på F803 och en lässignal öppnar minnet så att 00100000 (koden för BRANCH) kommer på databussen.
2. IR laddas och instruktionsavkodaren får besked om att nästa byte (på adress F804) ska adderas till PC.
3. PC räknas upp, skickas till AR och en ny lässignal kommer.
4. Innehållet på databussen adderas till PC. Detta kräver uppenbarligen mer utrustning i CPU:n än vad som visas i figur 3. Vi får tänka oss någon form av adderare kopplad till PC, för att kunna utföra ett relativt hopp.

Så här långt har vi bara sett vad registren CCR, ACC och PC används till.

I vår modell finns ytterligare några register som vi inte använt än.

I CPU:n finns två 16-bitarsregister, **IX** och **SP**.

IX (IndeX register) används för att indirekt nå olika delar av minnet. Vår modell är ju minnesorienterad och vi behöver ett sätt att läsa och skriva i det externa minnet via en adress som enkelt kan ändras.

SP (Stack Pointer), på svenska: stackpekare, används vid subrutinanrop (mer om det senare) och då man tillfälligt vill spara undan värden som ligger i CPU:n.

Utanför CPU:n finns två 8-bitarsregister som utgör datorns dörrar mot omvärlden.

De kallar vi **IN** och **OUT** och vi har gett dem varsin adress (så vi ska känna igen oss i nästa kapitel.

I denna minnesorienterade modell nås omvärlden alltså via adresser. Man kan tänka sig att IN och OUT är minnesceller i arbetsminnet, och att de dessutom är elektriskt anslutna till omvärlden.

Sådana register kallas för **portar**.

Vi ska nu först se på ett lite större exempel där vi använder register IX och en av de två adresser som är inkopplade till omvärlden. Se figur 9!

Vi utnyttjar också några fler instruktioner:

LDX (Load X) Ladda register X med ett tal.

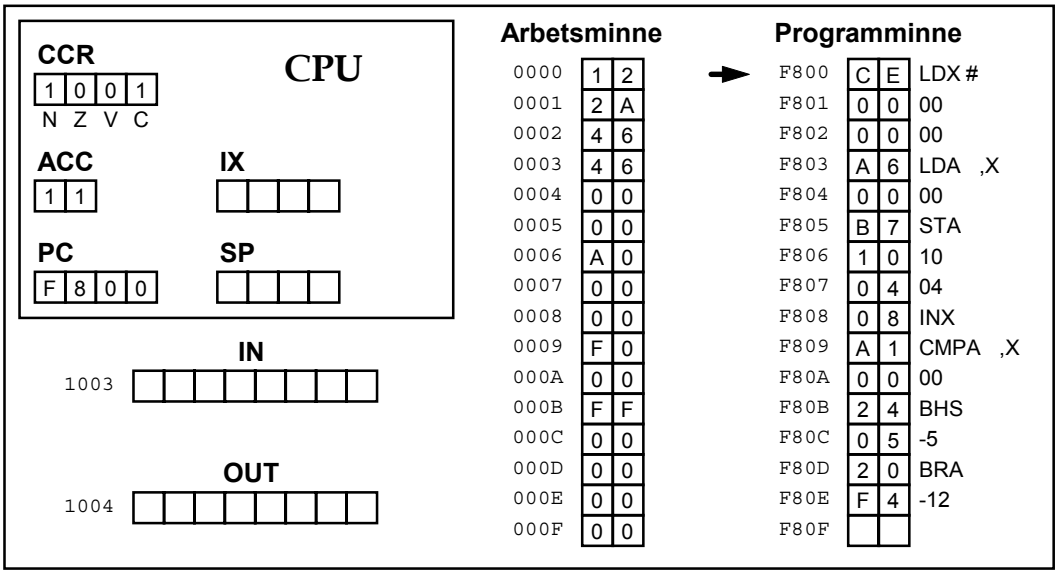
LDA (Load Accumulator) Ladda ackumulatorn med ett tal.

STA (Store Accumulator) Lagra ackumulatorns innehåll på en adress.

INX (Increment X) Öka värdet i X med ett.

CMPA (Compare Accumulator) Jämför ackumulatorn med ett tal.

BHS (Branch Higher or Same) Hoppa om det är högre eller samma.



Figur 9. Program med indirekt adressering och utmatning.

Om vi skulle översätta programmet till vanlig svenska, skulle det låta så här:

1. Ladda X med talet 0000.
2. Ladda A med talet på den adress som X pekar på.

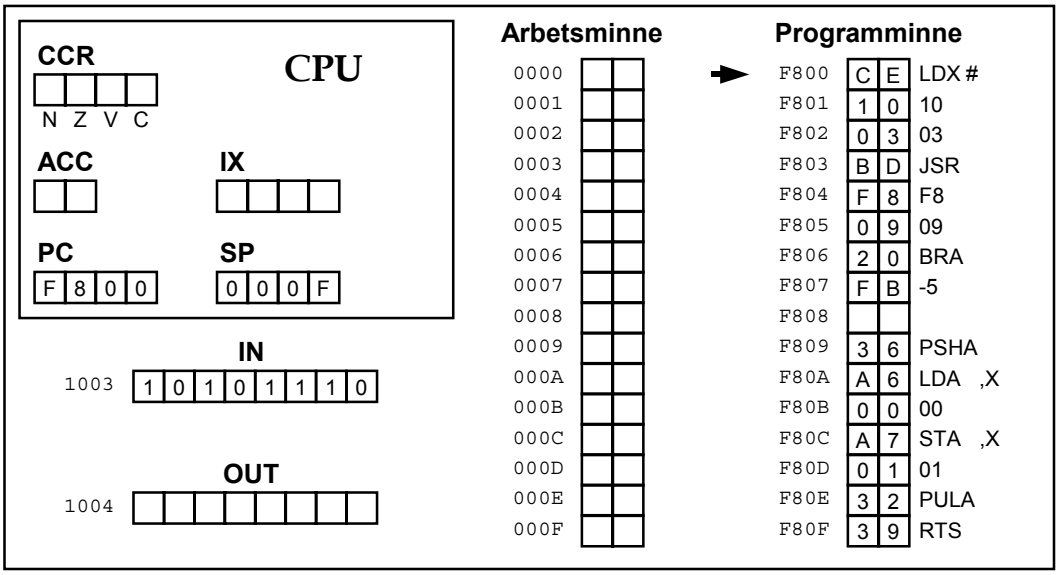
3. Lagra A på adress 1004.
4. Öka X med 1.
5. Jämför A med innehållet i den adress som X pekar på.
6. Om A är högre eller samma, gå till instruktionen INX.
7. Annars: Hoppa tillbaka till LDA.

Vi ser att programmet med hjälp av ett villkorligt hopp kan styras i en av två riktningar.

Just detta är avgörande för att man ska kunna utföra intelligent arbete i en dator: Med hjälp av vettiga beslut kan man få datorn att utföra ett nyttigt arbete.

Lägg märke till att när X används som indirekt adress, så förekommer ett värde (som i vårt fall är 0) som kallas offset. Man kan med den addera ett fast 8-bitars tal till värdet i X så att den utpekade adressen blir en annan. Mer om detta i kapitel 3.

Vi ska också se på ett enkelt exempel som använder stacken.



Figur 10. Exempel där stacken används.

Man kan säga att stacken är ett snabblagringsminne för lagring av sånt man ganska snart kommer att behöva.

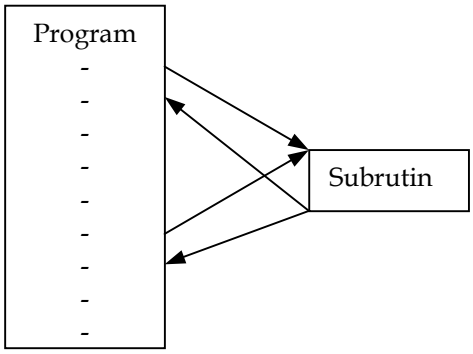
Vi ser att stackpekaren har värdet 0F. Det är alltså arbetsminnet som används som stack. Man måste se till att stackpekaren pekar på en minnescell där det går att skriva och att det finns ett tillräckligt utrymme mot lägre adresser.

En speciellt sätt att använda stacken, är då man gör ett *subrutinanrop* och vill återvända till nästa instruktion efter avklarad subrutin.

Subrutiner (underprogram) är en av de viktigaste uppfinningarna i datortekniken. En subrutin är en programdel som man återvänder från på ett speciellt sätt så att man alltid kommer tillbaka till det ställe man kom ifrån.

Detta går till så här:

- I samband med att man beger sig till subrutinen, sparas adressen till nästa instruktion.
- När subrutinen är klar, hoppar man tillbaka till den sparade adressen.



För att kunna använda subrutiner måste datorn vara utrustad med en mekanism som kan spara återhoppsadressen och sedan använda den vid återhoppet. Vanligtvis har datorn speciella instruktioner för subrutinhopp och återhopp:

- JSR (Jump to SubRoutine) Hoppa till subrutin.

Ett vanligt hopp utförs och dessutom sparas adressen till nästa instruktion på stacken.

- RTS (ReTurn from Subroutine) Återvänd från subrutin.

Den sparade adressen laddas tillbaka till programräknaren.

Ytterligare två nya instruktioner används i detta exempel:

- PSHA (Push A) Lägg ackumulatorns värde på stacken. Stackpekaren (SP) ändras automatiskt så att den pekar på ledig plats.
- PULA (Pull A) Hämtar ett värde på stacken och lägger det i A.

Vi översätter även detta exempel till svenska:

Ladda X med talet 1003.

Gör ett subrutinhopp till adress F809.

Hoppa tillbaka till instruktionen innan.

Subrutinen:

Spar innehållet i A på stacken.

Ladda A med det värd som finns på den plats som X pekar.

Skriv A:s värde på platsen med adress X+1.

Hämta tillbaka det undansparda A.

Återvänd från subrutinen.

1.3 Datorns instruktioner.

Vi ska här inte ge en fullständig förteckning över instruktioner som brukar förekomma, utan istället peka på några viktiga och förklara varför de finns.

Olika datorer har olika uppsättning av instruktioner. Det är användningsområdet som bestämmer om datorn ifråga exempelvis behöver en stor beräkningsförmåga.

1. Flytta data.

När man säger flytta data, menas egentligen att man kopierar ett värde från ett ställe till ett annat. Värdet ligger naturligtvis kvar där man läste av det. För att kunna mata in och ut information behövs uppenbarligen denna möjlighet, liksom då man gör beräkningar och måste spara mellanresultat.

I vår enkla datormodell utför instruktionen LDA en förflyttning (eller kopiering) av data från ett ställe i minnet till ackumulatorn. En kopiering i motsatt riktning kallas STA (av STORE, lagra).

2. Logik.

Eftersom en dator arbetar med logiska grindar, är det speciellt lätt att bestycka den med en s.k. logisk enhet, som kan utföra de vanliga logiska operationerna. Det behövs ju bara ett enkelt kombinatoriskt nät.

De logiska instruktionerna behövs tex. när man ska ettställa eller nollställa bitar för att styra yttre enheter.

Exempel: ANDA och ORA.

3. Aritmetik.

Någon form av beräkningskapacitet, förutom de rent logiska operationerna behövs ofta. Om man samlat in ett antal mätvärden och behöver beräkna medelvärdet, måste man kunna addera. Kan man addera går det även att subtrahera (2-komplement) och att utföra en multiplikation med ett antal instruktioner.

Exempel: ADDA och SUBA.

4. Ändra data.

Under denna rubrik kommer instruktionen INC, öka värdet. Andra exempel är DEC (minska) och CLR (nollställ).

5. Testa.

Dessa instruktioner är en variant av de logiska och aritmetiska, men de ger inget resultat i ackumulatorn utan påverkar bara resultatflaggorna. Exempel: CMPA som är en SUBA utan påverkan av ackumulatorn.

Instruktionerna används när man vill testa ett resultat utan att förstöra det.

6. Programkontroll.

Programkontrollinstruktioner är de som styr programmet olika vägar. Ofta är en test inbegripen för att ge möjlighet att fatta beslut.

Exempel: BNE, BEQ (branch if equal, hoppa vid likhet) och BRA.

Adresseringssätt.

Som vi sett i exemplen tidigare, finns det olika sätt att adressera. Vi har exempel på fem olika sätt:

- Omedelbar adressering.

Talet finns som en del av instruktionen. Används t.ex. vid load (LDX #0000). Parametern är här ingen adress, utan ett värde.

- Absolut adressering.

Man anger adressen med sitt egentliga värde (STA 1004).

- Indexerad (indirekt) adressering.

Adressen till den plats man vill nå, finns i ett register. I vår modell kan man använda register X till att peka ut en plats i minnet (LDA 0,X).

- Underförstådd adressering.

Används för instruktioner som inte behöver någon operand (CLRA, RTS).

- PC-relativ adressering.

Adressen som den anges i instruktionen är ett tal som adderas till programräknaren. Denna adressering används i hoppinstruktioner.

1.4 Övningar.

1. Studera programmet i fig. 9.

   a) I kap. 1.3 är instruktionerna uppdelade grovt i sex grupper. Ange för var och en av instruktionerna i programmet vilken grupp den tillhör.

   b) Ange vilken instruktion som använder omedelbar adressering.

   c) Ange vilka instruktioner som använder indirekt, absolut resp. PC-relativ adressering.

   d) När man skriver ett assemblerprogram, använder man med fördel symboliska lägen i sitt program. Man överlämnar då till översättarprogrammet (assembleraren) att beräkna dessa symbolers värden. Det kan se ut så här:

```
        ORG   $F800
        LDX   #$0000
Load    LDA  $00,X
        STA  $1004
Inc_X   INX
        CMPA  $00,X
        BHS   Inc_X
        BRA   Load
```

   En påtaglig skillnad mot skrivsättet i figur 9, är att det är lättare att följa programflödet. Innan man kör programmet låter man ju assembleraren räkna ut de relativa hoppen och skapa den korrekta koden.

   Den första raden i programmet talar om för översättarprogrammet att det ska läggas från och med adress F800.

   Antag att arbetsminnet ser ut som i figur 9.

   Ange hur du tror att flaggorna har påverkats när programmet första gången har kommit till instruktionen BRA. Tänk efter vilken eller vilka flaggor en instruktion måste testa för att bestämma om ett tal är mindre än ett annat.

   e) Vilket värde du tror ligger i adress 1004 (OUT) när programmet har gått ett tag? Lägg märke till att programmet kommer att gå i alloändlighet, men man kan ändå säga något om det värde som så småningom kommer att finnas i OUT-registret.

   Med andra ord: beskriv med en enkel sats vad programmet uträttar.

2. Se programmet i figur 10.

   Gå igenom det och se efter hur stackpekaren ändras och vad som läggs på stacken. Vi skriver om programmet med lägen och passar också på att ersätta adressen 1003 med 'IN'. Programmet blir genast lite lättare att följa.

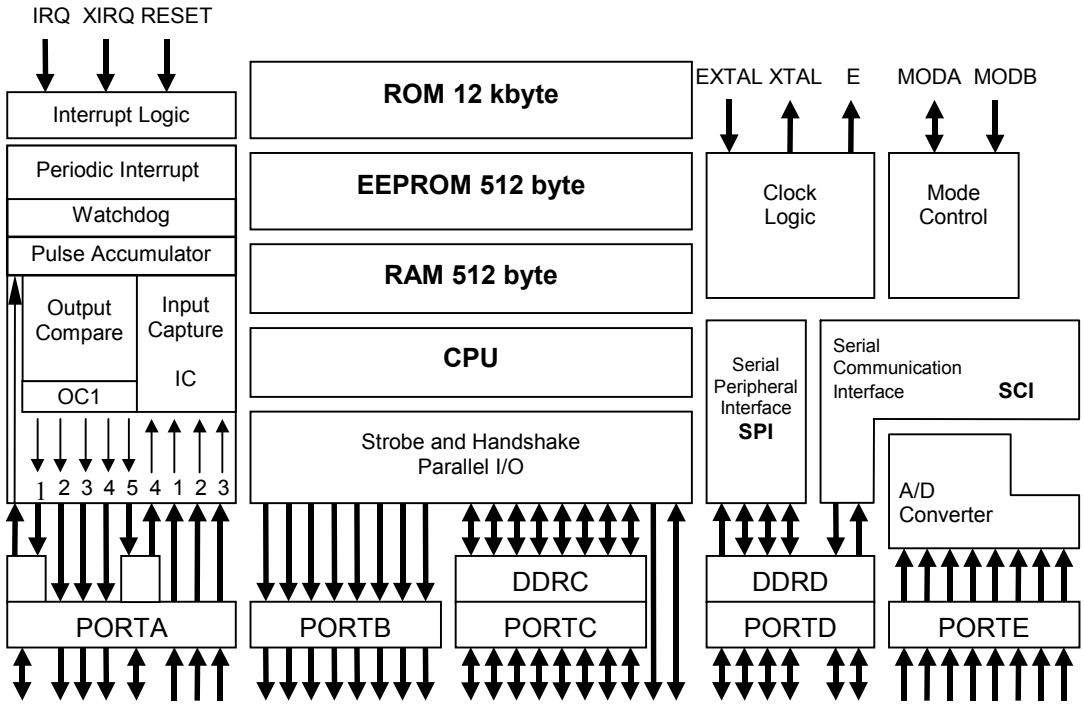
```
Start   LDX    #IN
        JSR    Copy
        BRA    Start

Copy    PSHA
        LDA    0,X
        STA    1,X
        PULA
        RTS
```

3. Data som ligger på adressområdet 00 - 05 ska flyttas till området 06 - 0B så att det som ligger på adress 00 ska kopieras till adress 06 osv. Föreslå ett program som utför detta.

2 En verklig datormodell: 68HC11.

Vi ska nu gå över till att studera en verklig mikrodator: 68HC11.

En dator i en kapsel med inbyggda minnen och periferienheter kallas enchipsdator eller mikrostyrkrets.



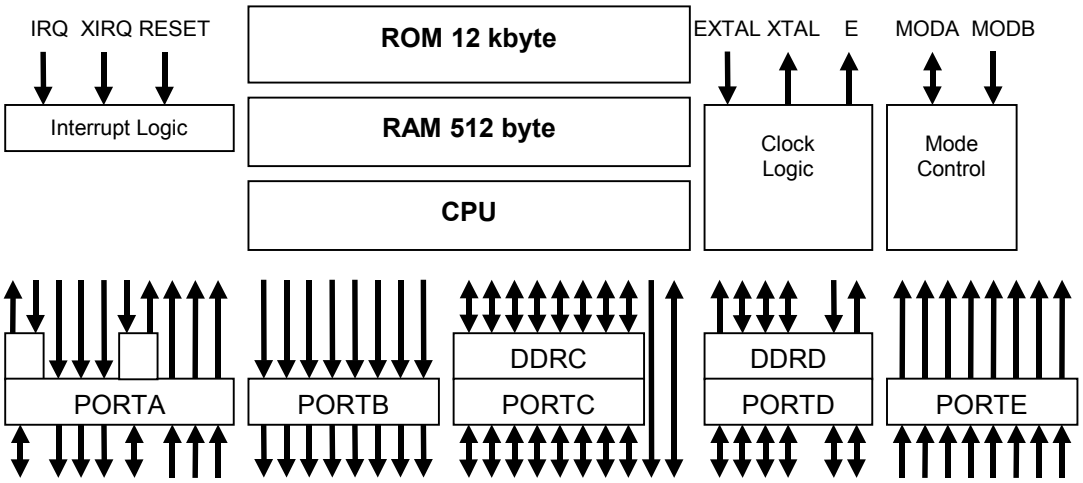
Figur 11. 68HC711E9

Vi ser att det är en mycket komplicerad konstruktion, vars detaljer vi ska gå in på i kapitel 5.

I detta kapitel ska vi studera hur CPU:n använder minnen och portar. De inbyggda kringkretsarna behöver i en enkel konstruktion inte användas, så vår modell av HC11:an kan än så länge vara betydligt enklare.

I den förenklade modellen ingår:

- CPU
- Programminne (ROM) och skrivbart minne (RAM)
- Fem portar med eventuella riktningsregister
- Oscillator och några kontrollsignaler



Figur 12. Förenklad HC11-modell.

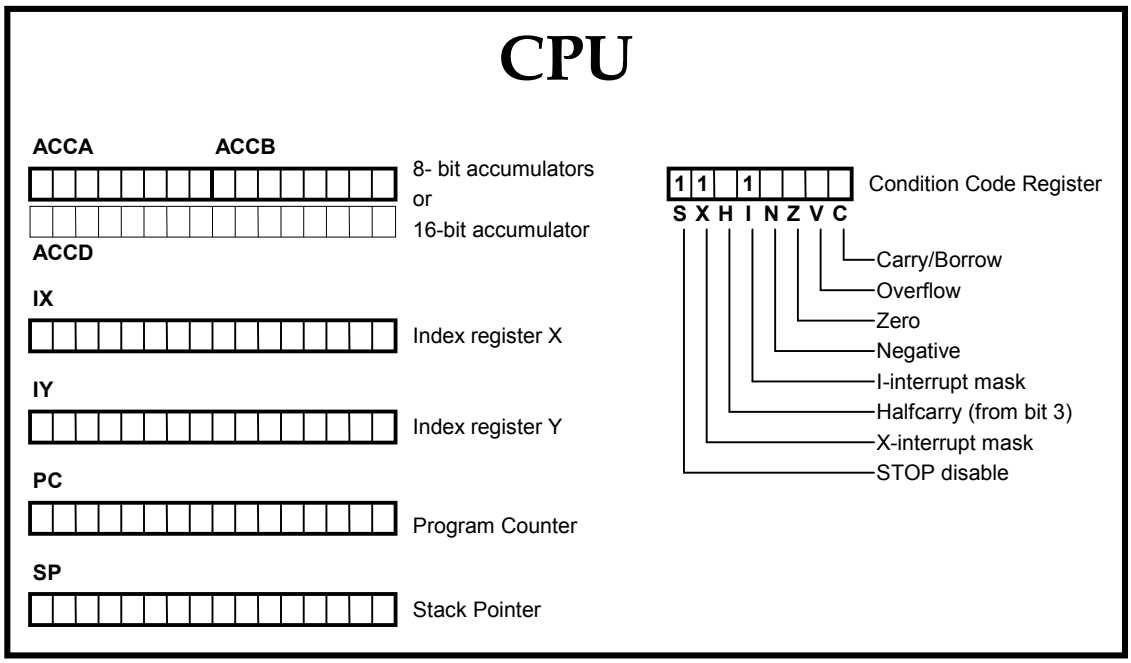
$\overline{\textbf{RESET}}$ (återställning) är en signal som måste ges till mikrodatorn för att den starta i ett bestämt tillstånd. En dator är ju ett stort sekvensnät, och måste alltså vara i ett bestämt tillstånd från början, för att allt ska fungera som tänkt. Denna signal är aktivt låg, dvs. en logisk nolla ger reset till mikrodatorn.

Signalerna **MODA** och **MODB** ger en initieringskod till datorn, så att den vet hur den ska arbeta; man kan tex. få den att via två portar läsa programkod utanför kretsen eller starta i ett 'bootstrap'-program som ligger i datorn.

En svängningskrets (oscillator) finns också inbyggd. Den behöver en yttre återkoppling mellan **XTAL** och **EXTAL**, vanligen en kristall. Kristallfrekvensen delas internt med 4 och utgör arbetsfrekvensen E (systemklockan) hos processorn.

Bilden visar också de två avbrottsingångar $\overline{\textbf{XIRQ}}$ och $\overline{\textbf{IRQ}}$ som är förknippade med två kontrollbitar i CPU:ns flaggregister. Se kap. 2.1.

2.1 CPU



Figur 13. CPU:ns uppbyggnad hos 68HC11.

HC11:an innehåller två 8-bitars ackumulatorer som för vissa instruktioner kan fungera som en 16-bitars dubbelackumulator.

Det finns två indexregister med vilkas hjälp man kan utföra indirekt adressering. X-registret används också vid några instruktioner som använder 16 bitar.

Programräknaren används till att läsa av programkoden.

Stackpekaren är en speciell adresspekare som används vid vissa instruktioner som tex. subrutinanrop.

Flaggregistret innehåller förutom de fyra resultatflaggorna från tidigare exempel, ytterligare en resultatflagga och tre bitar som styr programmet.

Den resultatflagga som tillkommer är:

- **H** Halfcarry. Används endast av instruktionen DAA, Decimal Adjust Accumulator. Efter en addition kan man använda instruktionen för att korrigera uträkningen så att det ser ut som om ALU:n räknar decimalt. Mer om denna instruktion i kap. 3.

De tre kontrollbitarna är:

- **S** STOP disable. Om denna bit är ettställd går inte instruktionen STOP att utföra. STOP-instruktionen stoppar mikrodatorns oscillator så att man kan få en oerhört liten strömförbrukning. All verksamhet stoppas men återupptas om ex.vis RESET-signalen påverkas.
- **X** X-interrupt disable. Biten förhindrar, om den är ettställd, X-avbrottet (XIRQ) från att bryta normal programkörning. Den kan nollställas, men inte ettställas igen. X-avbrottet används till att bryta programmet när något allvarligt inträffar.
- **I** I-interrupt disable. Förhindrar, om den är ettställd varje påverkan från alla andra avbrottskällor, inklusive det externa IRQ-avbrottet.

Dessa tre bitar är alltid ettställda då processorn startar; behöver man använda STOP-instruktionen eller avbrottssystemet får man låta sitt program nollställa respektive bitar.

I vår förenklade modell av 68HC11 väljer vi bort en mängd kringkretsar, men vi låter CPU:n vara fullständig.

Dels behöver vi det fullständiga utseendet för att kunna förstå instruktionerna, och dels är den ändå ganska enkel.

2.2 Minnen.

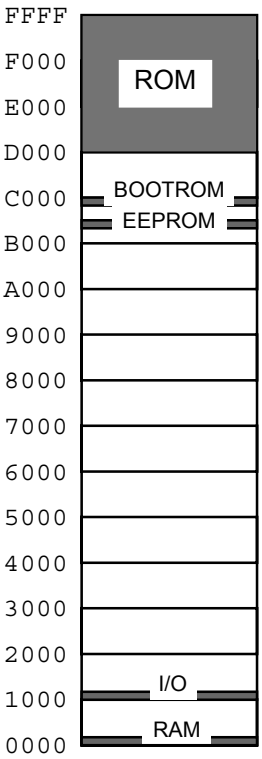
Normalt finns i HC11 fyra olika typer av minne:

- **ROM** Programminnet. Ligger i den del av adressrymden som har de högsta adresserna. Vid reset måste programmets startadress kunna läsas från adresserna FFFE och FFFF (se kap. 6). Olika typer av HC11 har olika stora programminnen.

- **BOOTROM** Program som används vid 'BOOTSTRAP MODE'. Mer om detta i kap. 7.1.2.

- **EEPROM** Alla HC11:or innehåller ett elektriskt raderbart och programmerbart minne, EEPROM, där ett program självt kan ändra innehållet. Se kap. 7.5.

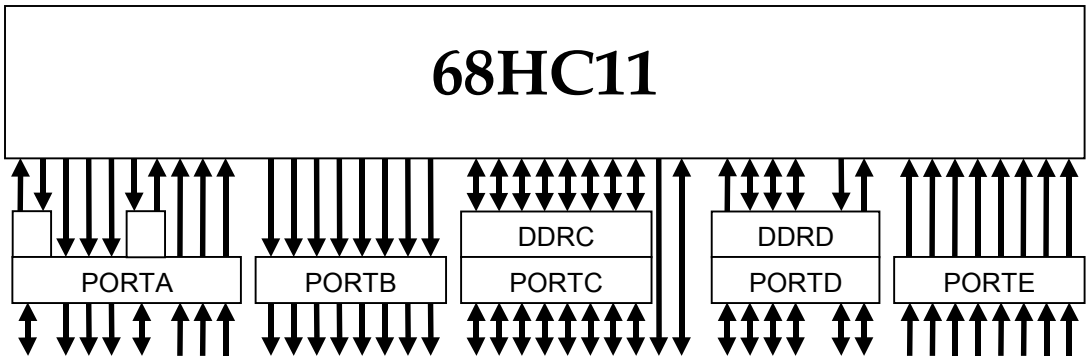
- **RAM** Ett internt skrivminne, minst 256 byte stort. Används till stack och arbetsminne. Ligger normalt på adress 0000 men kan flyttas (se kap. 7.6).



Än så länge behöver vi bara ägna oss åt programminnet och det skrivbara RAM-minnet. EEPROM:et och BOOTROM:et behövs normalt inte för att ett program ska kunna fungera.

Vi kommer att gå noggrannare in på dessa enheter i kapitel 7.

2.3 Portar.



Figur 14 In- och utportar hos 68HC11.

Det finns fem portar inkopplade i HC11:ans minnesområde. Ur programmerarens synvinkel ser de ut som minnesceller med sina adresser.

I figuren ovan finns alla fem portarna utritade. Några har kontrollregister och det är i dessa som riktningen på varje enskild bit kan bestämmas. Om motsvarande riktningsbit är nollställd, fungerar signalen som en ingång. Är riktningsbiten ettställd, blir signalen utgång.

Varje port har en speciell adress, och varje kontrollregister har sin.

I bilden på föregående sida syns ett litet område som är betecknat I/O. Här är portarna placerade.

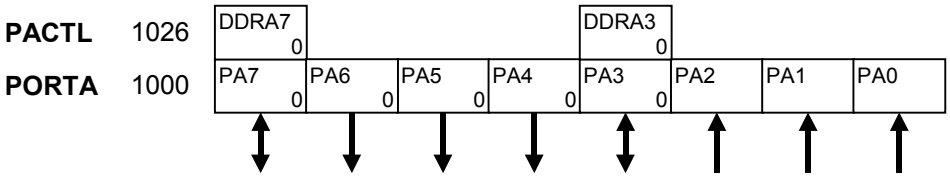
I de fall portsignalen är vändbar, är det viktigt att riktningen alltid är in vid uppstart. Detta sköts automatiskt genom att riktningsregistrena nollställs.

Alla portsignaler är nollställda från början. Detta gäller både de fasta utsignalerna och de vändbara. Alla portsignaler som kan vara utgångar har nämligen ett utregister som är skilt från de signaler som eventuellt läses in.

På de följande sidorna ges en beskrivning av portarna.

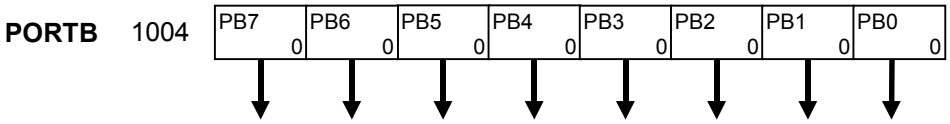
När portsignalerna är vändbara, styrs riktningen av riktningsbitar i speciella register. Dessa bitar anges i bilderna med 0 eller 1 vid de dubbelriktade pilarna.

PORT A.



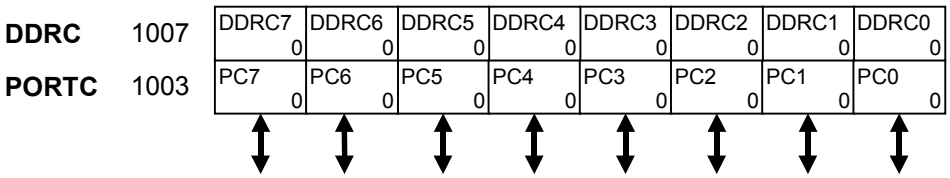
Port A har tre fasta utgångar, tre fasta ingångar och två vändbara, PA3 och PA7. Deras riktning kontrolleras med DDRA3 och DDRA7 i registret PACTL med adress 1026. De är båda ingångar från början, men ställs de om till utgång, dyker genast utregistrets nolla upp. Samma sak gäller andra portars dubbelriktade bitar.

PORT B.



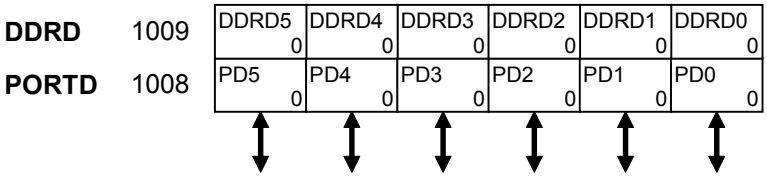
Port B är bara utport och har alltså inget riktningsregister. Alla bitar är nollställda från början.

PORT C.



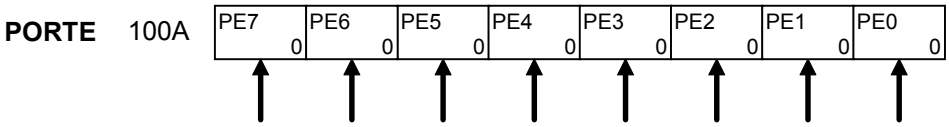
Port C består av 8 vändbara signaler, individuellt programmerbara som ingång eller utgång.

PORT D.



Port D innehåller 6 signaler, individuellt vändbara.

PORT E.



Port E består av 8 ingångar (när HC11 är kapslad i 48-pinnars DIL-kapsel, finns bara fyra inkopplade, E3 - E0).

2.4 Övningar.

1. Vilket bitmönster ska läggas i DDRC om alla bitar ska vara utgångar utom de två mittersta?

2. Ett antal (8) identiska yttre kretsar är så byggda att de aktiveras med en låg signal (nolla). Dessa signaler ska kontrolleras av en av HC11:ans portar.

Hur ska kretsarna kopplas in, om man kräver att aldrig mer än en av de yttre kretsarna får aktiveras samtidigt?

3. Vilket är det maximala antalet utgångar man kan ha?

Hur många insignaler får man då?

4. Varför är det viktigt att vändbara signaler fungerar som insignaler vid uppstart?

5. I Port C ska alla bitar utom de tre lägsta ställas till utgångar. Föreslå en sekvens av instruktioner som gör detta.

3 Assemblerprogrammering.

Att programmera i assemblerspråk innebär att man skriver program i form av förkortningar (mnemonics) som var och en motsvarar datorns instruktioner. Förkortningarna översätts sedan av en assemblerare (ett program) som skapar maskinkoden som vi sett den i kapitel 1.

I det här kapitlet ska vi gå igenom HC11:ans alla instruktioner och illustrera användningen i några programmeringsexempel.

Avsikten är att kapitlet ska vara ett alternativ till databokens instruktionsförteckning.

Instruktionerna är indelade i tio grupper. Grupperna återspeglar olika instruktionstyper så det blir lättare att se vilka instruktioner som finns. Hur man i detalj skriver ett program i assembler beror på hur den aktuella assembleraren är konstruerad.

Den beskrivning som här följer (liksom den man finner i databöcker) är av generell natur.

Att skriva assemblerprogram är inte heller bara en fråga om att känna till instruktionerna. Man måste också veta hur man bäst använder dem. Genom att använda högnivåspråk som C får man automatiskt ett strukturerat program. Detta kan man läsa om i kapitel 8 Programmering i C.

Före genomgången ger vi en utförlig beskrivning av HC11:s **adresseringssätt**.

Efter instruktionerna följer i avsnitt 3.3 en sammanställning av några vanliga **assemblerdirektiv**. Dessa är kommandon i källkoden som innehåller annan information än rena instruktioner. Bl.a. finns direktiv för att lägga in fasta tabeller och för att styra assembleringen till den adressrymd som finns tillgänglig.

3.1 Adresseringssätt.

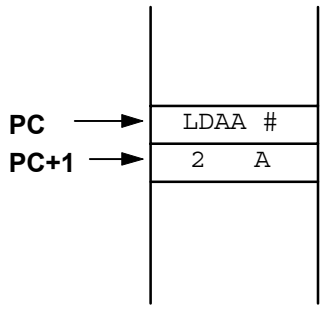
Det är viktigt att känna till de olika sätt som finns att nå delarna utanför CPU:n. Genom att välja lämplig adressering gör man programmeringen enklare och får en överskådlig kod.

Följande tre adresseringssätt används för att nå register med minnesadresser.

- **Omedelbar adressering (Immediate)**

Den adress vars innehåll man vill nå, pekas ut av programräknaren själv. Det innebär att innehållet finns i **omedelbar** anslutning till instruktionskoden. Man kan se själva värdet som en del av instruktionen.

Instruktioner som använder denna adressering kan då naturligtvis inte skriva till minnet.



Exempel: LDAA #2A vilket innebär att det hexadecimala värdet 2A laddas in i ackumulator A. Efter instruktionen ligger alltså operanden i ackumulator A.

I Motorolas assemblersyntax (stavningsregler) betecknar **#** omedelbar adressering och **$** hexadecimalt värde.

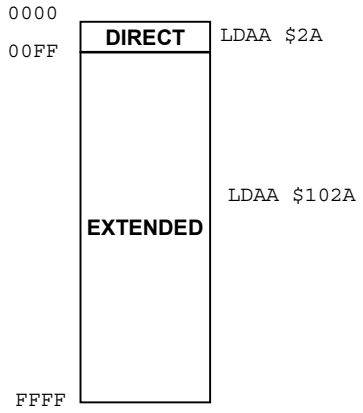
- **Absolut adressering**

Med absolut adressering menas att adressens absoluta värde ingår i instruktionen. I HC11 kan denna adressering ske på två olika sätt:

1. **Direct.** Om adressen har ett värde från 0000 tom 00FF kan en speciell operationskod användas och då anges som operand bara en 8-bitars adress.

Exempel: LDAA $2A: Ladda A med det värde som finns på adress 002A.

2. **Extended** (utvidgad). Är adressen större, behövs hela värdet som operand. Exempel: LDAA $102A. Det värde som finns på adress 102A hamnar i A.



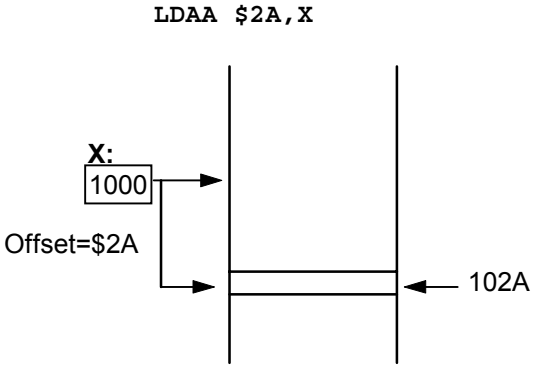
I regel behöver programmeraren inte bry sig om vilket av adresseringssätten som används; assembleringsprogrammet väljer själv direkt adressering om det kan.

- **Indexerad (indirekt) adressering**

Vid indexerad adressering bildas adressen av innehållet i ett 16-bitars register, X eller Y, adderat med en offset som är ett 8-bitars positivt tal.

Exempel: LDAA $2A,X.

I register X finns en adress. Till denna adress adderas talet $2A och det då bildade värdet utgör en adress som pekar ut en plats i minnet. Talet som ligger där kommer att hamna i A. Offseten är alltid positiv.



Ofta använder man inte någon offset, utan nyttjar den indirekta adresseringen till att exempelvis nå innehållet i tabeller. Då kan man använda en instruktion som INX för att öka värdet i X så att X pekar på efterföljande adress.

En variant av indirekt adressering är när stackpekaren används som pekare. Stackpekaren ska peka på skrivbart minne och används av vissa instruktioner för att tillfälligt lagra värden, som senare kommer att behövas.

Instruktioner som använder stackpekaren är så konstruerade att de automatiskt ökar eller minskar dess värde. Se kap. 3.2.

Register inuti CPU:n hanteras med instruktioner utan operand. Istället är det själva instruktionsförkortningen som visar vilket eller vilka register som är inblandade. Man kan säga att operanden är inbyggd i instruktionen.

Detta kallar vi

- **Inbyggd adressering (Inherent)**

Exempel: TBA

betyder Transfer B to A. Flytta (kopiera) från B till A.

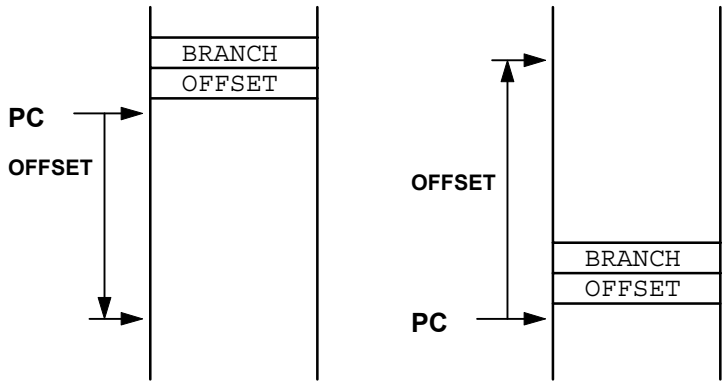
Instruktioner med minnesadressering förekommer ofta i en variant med inbyggd adressering.

Exempel: CLRA

som innebär att A ska nollställas. Instruktionen CLR finns också med absolut och indirekt adressering.

En viss typ av instruktioner, **BRANCH**-instruktionerna, använder adressering i förhållande till programräknarens värde. Adresseringssättet kallas PC-relativ adressering.

- **PC-relativ adressering**



BRANCH-instruktionen har en offset som kan vara både negativ och positiv. Om den är positiv, kan den vara högst 127, dvs. det högsta positiva tal som kan representeras med 8 bitar i 2-komplement. Offseten räknas alltid från adressen efter instruktionen, eftersom programräknaren ökas automatiskt för att peka på nästa instruktion.

3.2 Instruktioner

Instruktionerna är i detta kapitel indelade grupper som avspeglar olika användningsområden.

Följande tre huvudgrupper kan man urskilja, men vissa instruktioner går inte att föra in under någon av dessa grupper.

- Kopiera och ändra data.
- Aritmetik och logik.
- Testning och programstyrning.

I den följande sammanställningen har indelningen gjorts finare och instruktionerna delats in i tio grupper.

I tabellerna förekommer två, tre eller fyra kolumner:

- Den första innehåller instruktionens förkortning, mnemonic på engelska (mnemonic betyder stöd för minnet).
- Den andra förklarar kort vad instruktionen innebär.
- En tredje med rubriken 'Adresser' finns med då olika typer av adresseringar kan förekommer och där finns då angivet vilka som kan användas.

  Följande förkortningar gäller:

#	immediate	omedelbar
D	direct	8-bitars adress
E	extended	16-bitars adress
I	indexed	indexregister + 8-bitars offset
R	relative	8-bitars offset i 2-komplement

- En eventuell sista kolumn visar om och hur instruktionen påverkar flaggorna.

3.2.1 Kopiera data.

A. Från minne till CPU-register (Ladda) :

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
LDAA	M → A	# D E I					↑↓	↑↓	0	
LDAB	M → B	# D E I					↑↓	↑↓	0	
LDD	M:M+1 → D	# D E I					↑↓	↑↓	0	
LDX	M:M+1 → X	# D E I					↑↓	↑↓	0	
LDY	M:M+1 → Y	# D E I					↑↓	↑↓	0	
LDS	M:M+1 → SP	# D E I					↑↓	↑↓	0	

Instruktionerna börjar på **LD** (LOAD).

Beteckningen M:M+1 innebär innehållet i minnescellen M och den efterföljande. 16-bitarsregister måste ju laddas med innehållet från två 8-bitars minnesceller.

Exempel:

```
LDAA  #$23 ladda ackumulator A med talet $23.

LDAA  $23  ladda A med talet som ligger i adress $23.
```

B. Från CPU-register till minne (Lagra) :

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
STAA	A → M	D E I					↑↓	↑↓	0	
STAB	B → M	D E I					↑↓	↑↓	0	
STD	D → M:M+1	D E I					↑↓	↑↓	0	
STX	X → M:M+1	D E I					↑↓	↑↓	0	
STY	Y → M:M+1	D E I					↑↓	↑↓	0	
STS	SP → M:M+1	D E I					↑↓	↑↓	0	

Lagringsinstruktionerna börjar på **ST** (STORE). Naturligtvis finns här inte omedelbar adressering.

Tänk på att LOAD och t.o.m. STORE påverkar flaggorna. Ett inladdat värde behöver inte testas med en speciell instruktion för att man tex. ska kunna se att värdet är = 0.

Exempel:

```
       STD   $23 lagra dubbelackumulatorns innehåll i
*                adresserna $23 och $24. Innehållet i A
*                hamnar i adress $23 och B:s innehåll i
*                adress $24.
```

LOAD och STORE är de normala sätten att hämta från minnen och portar, eller skriva till minnen och portar.

C. Inom CPU:

MNEMONIC	Förklaring	Flaggor							
		S	X	H	I	N	Z	V	C
TAB	A → B					↑↓	↑↓	**0**	
TBA	B → A					↑↓	↑↓	**0**	
TXS	X-1 → SP								
TYS	Y-1 → SP								
TSX	SP+1 → X								
TSY	SP+1 → Y								
XGDX	D ↔ X								
XGDY	D ↔ Y								

T betyder TRANSFER. De instruktioner som börjar på T gör en helt vanlig kopiering, men inom CPU:n.

Lägg märke till att TRANSFER-instruktionerna som använder stackpekaren, gör en automatisk minskning eller ökning av det överförda värdet. Förklaringen är att man tex. med instruktionen TSX får reda på adressen till det värde som ligger överst i stacken, och inte adressen till den lediga platsen ovanpå stacken.

De två sista instruktionerna, som börjar på **X** (EXCHANGE), byter innehållet i dubbelackumulatorn med X resp. Y.

Exempel:

```
XGDX            Denna sekvens av instruktioner
XGDY            byter innehållet i X
XGDX            med innehållet i Y.
```

D. Kopiering med hjälp av stackpekaren:

MNEMONIC	Förklaring
PSHA	A → (SP), SP-1 → SP
PSHB	B → (SP), SP-1 → SP
PSHX	X → (SP), SP-2 → SP
PSHY	Y → (SP), SP-2 → SP
PULA	SP+1 → SP, (SP) → A
PULB	SP+1 → SP, (SP) → B
PULX	SP+2 → SP, (SP) → X
PULY	SP+2 → SP, (SP) → Y

PSH (PUSH) innebär att innehållet i ett av CPU-registren lagras "på stacken", dvs. det ställe i minnet som stackpekaren pekar ut. Efter varje PUSH minskas stackpekaren så att den åter pekar på ledig minnescell. PSHX och PSHY utför lagringen i två steg, så att den låga halvan lagras först.

PUL (PULL) har en direkt motsatt funktion.

Dessa instruktioner påverkar inga flaggor.

Exempel:

En enkel fördröjningsrutin som ej påverkar CPU-register:

```
WAIT    PSHX
        LDX   #$1234
AGAIN   DEX
        BNE   AGAIN
        PULX
        RTS
```

Genom att inleda subrutinen med **PSHX** och avsluta den med **PULX**, kommer innehållet i X att vara detsamma som före subrutinanropet.

3.2.2 Ändra data.

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
DEC	M-1 → M	E I					↑↓	↑↓	↑↓	
DECA	A-1 → A						↑↓	↑↓	↑↓	
DECB	B-1 → B						↑↓	↑↓	↑↓	
DEX	X-1 → X							↑↓		
DEY	Y-1 → Y							↑↓		
DES	SP-1 → SP									
INC	M+1 → M	E I					↑↓	↑↓	↑↓	
INCA	A+1 → A						↑↓	↑↓	↑↓	
INCB	B+1 → B						↑↓	↑↓	↑↓	
INX	X+1 → X							↑↓		
INY	Y+1 → Y							↑↓		
INS	SP+1 → SP									
NEG	0-M → M	E I					↑↓	↑↓	↑↓	↑↓
NEGA	0-A → A						↑↓	↑↓	↑↓	↑↓
NEGB	0-B → B						↑↓	↑↓	↑↓	↑↓
COM	$FF-M → M	E I					↑↓	↑↓	0	↑↓
COMA	$FF -A → A						↑↓	↑↓	0	↑↓
COMB	$FF -B → B						↑↓	↑↓	0	↑↓
CLR	0 → M	E I					0	1	0	0
CLRA	0 → A						0	1	0	0
CLRB	0 → B						0	1	0	0
BCLR	M∧mask' → M	D I					↑↓	↑↓	0	
BSET	M∨mask → M	D I					↑↓	↑↓	0	

Här finns ett stort antal instruktioner för att minska, **DEC** (DECREMENT), eller öka, **INC** (INCREMENT), innehållet i CPU-register och minnesceller.

Om det gäller 16-bitars register heter instruktionerna tex. **DEX**, dvs. utan C.

Exempel:

```
        DEC   Mem        Minska innehållet i minnescellen 'Mem'.
        BEQ   ready      Hoppa till 'ready' om det blivit noll.
```

Vidare finns instruktionen **NEG** (NEGATE), omvandling från positivt till negativt värde och tvärtom.

Exempel:

I minnescellen 'Mem' ligger ett tal, positivt eller negativt. Vi önskar justera det så att minnescellen istället innehåller absolutvärdet.

```
        TST   Mem        Testa innehållet i Mem.
        BPL   ok         Om det är positivt, görs inget.
        NEG   Mem        Annars: negera.
ok      NOP
```

Det finns även invertering, **COM** (COMPLEMENT) och nollställning **CLR** (CLEAR).

Exempel:

Ettställ alla bitar i 'Mem':

```
CLR    Mem
COM    Mem
```

Slutligen två instruktioner för nollställning eller ettställning med ett angivet bitmönster, **BCLR** (BIT CLEAR) och **BSET** (BIT SET).

Dessa två instruktioner har en tredje parameter, som anges efter adressen till den plats där operationen ska utföras.

I **BCLR** utförs en och-funktion mellan minnesinnehåll och maskens invers, dvs. ettställda bitar i masken ger nollställning som resultat.

I **BSET** sker en eller-funktion med maskens ettor.

Exempel:

Nollställ den högsta och den lägsta biten i 'Mem':

```
BCLR   Mem,%10000001
```

Ettställ de två mittersta bitarna i 'Mem':

```
BSET   Mem,%00011000
```

Kom ihåg: Man anger alltid med ettor vilka positioner som ska nollställas resp. ettställas.

3.2.3 Skiftinstruktioner.

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
LSL	C ← b7...b0 ← 0	E I					↑↓	↑↓	↑↓	↑↓
LSLA	C ← b7...b0 ← 0						↑↓	↑↓	↑↓	↑↓
LSLB	C ← b7...b0 ← 0						↑↓	↑↓	↑↓	↑↓
LSLD	C ← b15...b0 ← 0						↑↓	↑↓	↑↓	↑↓
LSR	0 → b7...b0 → C	E I					**0**	↑↓	↑↓	↑↓
LSRA	0 → b7...b0 → C						**0**	↑↓	↑↓	↑↓
LSRB	0 → b7...b0 → C						**0**	↑↓	↑↓	↑↓
LSRD	0 → b15...b0 → C						**0**	↑↓	↑↓	↑↓
ASL	C ← b7...b0 ← 0	E I					↑↓	↑↓	↑↓	↑↓
ASLA	C ← b7...b0 ← 0						↑↓	↑↓	↑↓	↑↓
ASLB	C ← b7...b0 ← 0						↑↓	↑↓	↑↓	↑↓
ASLD	C ← b15...b0 ← 0						↑↓	↑↓	↑↓	↑↓
ASR	b7 → b7, b6...b0 → C	E I					↑↓	↑↓	↑↓	↑↓
ASRA	b7 → b7, b6...b0 → C						↑↓	↑↓	↑↓	↑↓
ASRB	b7 → b7, b6...b0 → C						↑↓	↑↓	↑↓	↑↓
ROL	C ← b7...b0 ← C	E I					↑↓	↑↓	↑↓	↑↓
ROLA	C ← b7...b0 ← C						↑↓	↑↓	↑↓	↑↓
ROLB	C ← b7...b0 ← C						↑↓	↑↓	↑↓	↑↓
ROR	C → b7...b0 → C	E I					↑↓	↑↓	↑↓	↑↓
RORA	C → b7...b0 → C						↑↓	↑↓	↑↓	↑↓
RORB	C → b7...b0 → C						↑↓	↑↓	↑↓	↑↓

Här finns tre huvudgrupper **LS** (logisk skiftning), **AS** (aritmetisk skiftning) och **RO** (rotering).

Den tredje bokstaven i förkortningen betecknar left (**L**) och right (**R**).

En eventuell fjärde bokstav finns med när underförstådd adressering gäller och anger vilket register som gäller.

- **LS**, LOGICAL SHIFT. I den ena änden skiftas en nolla in och ur den andra skiftas värdet in i Carryn.

Exempel:

En subrutin som byter höga och låga halvan i ackumulator A kan se ut så här:

```
swapA   PSHB
        LSRD
        LSRD
        LSRD
        LSRD
        ANDB   #$F0
        ABA
        PULB
        RTS
```

- **AS**, ARITHMETIC SHIFT. För vänsterskift är dessa instruktioner identiska med logisk skift. Vid högerskift bevaras talets tecken genom att bit 7 ej ändras.

Exempel:

Dela ett tal med 4. Det ska fungera även för negativa tal! -4 (hexadecimalt FC) ska bli -1 (hex FF).

```
LDAA  tal
ASRA
ASRA
STAA  tal
```

Om man vill multiplicera med 4 använder man **ASLA** (som är identisk med **LSLA**). Här behövs ingen kopiering av teckenbiten; den bevaras automatiskt så länge man håller sig inom talområdet. Så fort talområdet överskrids sätts V-flaggan.

- **RO**, ROTATE. I den ena änden skiftas Carryn in och ur den andra änden skiftas värdet in i Carryn.

Exempel:

Dela ett 16-bitars tal i minnet med 4. Talet ligger på adress 'value'.

```
ASR   value
ROR   value+1
ASR   value
ROR   value+1
```

3.2.4 Logikinstruktioner.

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
ANDA	$A \wedge M \rightarrow A$	# D E I					↑↓	↑↓	0	
ANDB	$B \wedge M \rightarrow B$	# D E I					↑↓	↑↓	0	
ORAA	$A \vee M \rightarrow A$	# D E I					↑↓	↑↓	0	
ORAB	$B \vee M \rightarrow B$	# D E I					↑↓	↑↓	0	
EORA	$A \oplus M \rightarrow A$	# D E I					↑↓	↑↓	0	
EORB	$B \oplus M \rightarrow B$	# D E I					↑↓	↑↓	0	

Logiska instruktioner kan användas till att nollställa, ettställa eller invertera bitar i ackumulatorn.

AND utför en bitvis och-funktion mellan en ackumulator och innehållet i en minnescell (enligt adresseringssättet).

Exempel:

Nollställ den näst högsta biten i port A.

```
LDAA   PORTA
ANDA   #$BF
STAA   PORTA
```

Motsvarighet i C:

```
PORTA = PORTA & 0xBF
```

ORA (or accumulator) utför en bitvis eller-funktion mellan ackumulator och minnesinnehåll.

Exempel:

Ställ om de fyra översta bitarna i port C till utgångar.

```
LDAA   DDRC
ORAA   #$F0
STAA   DDRC
```

Motsvarighet i C:

```
PORTA = PORTA | 0xF0
```

EOR (exclusive or) gör exklusivt eller mellan ackumulator och minnesinnehåll. Ett behändigt sätt att invertera valda bitar.

Exempel:

Växla bit 4 i port A.

```
LDAA   PORTA
EORA   #$10
STAA   PORTA
```

Motsvarighet i C:

```
PORTA = PORTA ^ 0x10
```

3.2.5 Aritmetik.

Addition och subtraktion

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
ADDA	A + M → A	# D E I			↑↓		↑↓	↑↓	↑↓	↑↓
ADDB	B + M → B	# D E I			↑↓		↑↓	↑↓	↑↓	↑↓
ADCA	A + M + C → A	# D E I			↑↓		↑↓	↑↓	↑↓	↑↓
ADCB	B + M + C → B	# D E I			↑↓		↑↓	↑↓	↑↓	↑↓
ABA	A + B → A				↑↓		↑↓	↑↓	↑↓	↑↓
DAA	Decimal adjust A						↑↓	↑↓	?	↑↓
SUBA	A - M → A	# D E I					↑↓	↑↓	↑↓	↑↓
SUBB	B - M → B	# D E I					↑↓	↑↓	↑↓	↑↓
SBCA	A - M - C → A	# D E I					↑↓	↑↓	↑↓	↑↓
SBCB	B - M - C → B	# D E I					↑↓	↑↓	↑↓	↑↓
SBA	A - B → A						↑↓	↑↓	↑↓	↑↓
ADDD	D + M:M+1 → D	# D E I					↑↓	↑↓	↑↓	↑↓
SUBD	D - M:M+1 → D	# D E I					↑↓	↑↓	↑↓	↑↓
ABX	B + X → X									
ABY	B + Y → Y									

De aritmetiska instruktionerna finns för både 8-bitars och 16-bitars talområde.

DAA (DECIMAL ADJUST A) ska, om den används, utföras direkt efter någon av instruktionerna **ADDA**, **ADCA** eller **ABA**. Den justerar då resultatet till decimalt.

För 8-bitars talområde finns även addition med Carry. Detta gör det möjligt att lätt utvidga talområdet så att minnessiffran påverkar additionen i nästa position.

Exempel:

Ett 16-bitars decimaltal ska ökas med ett. Talet ligger med sin höga halva i adress High och med sin låga halva i adress Low.

```
LDAA  Low        Hämta låga halvan
ADDA  #$01       Addera ett
DAA
STAA  Low        Lägg tillbaka
LDAA  High       Hämta höga halvan
ADCA  #$00       Addera ev. minnessiffra
DAA
STAA  High       Lägg tillbaka
```

Två av instruktionerna, **ABX** och **ABY** sätter inga flaggor; de är avsedda för att manipulera adresser i indexregister.

Exempel:

I variabeln Code ligger ett tal som bestämmer var i tabellen TAB ett värde ska hämtas.

```
LDX   #TAB       X pekar på tabellstart
LDAB  Code       B är platsnummer i tabellen
ABX              X pekar nu rätt
LDAA  0,X        Hämta tabellvärde
```

Om man vill jämföra två tal utan att få ett nytt resultat i ackumulatorn, finns varianter av subtraktionerna under avsnitt 3.2.6, Testning.

Multiplikation och division

MNEMONIC	Förklaring	Flaggor							
		S	X	H	I	N	Z	V	C
MUL	A x B → D								↑↓
IDIV	D ÷ X → X, r → D						↑↓	0	↑↓
FDIV	D ÷ X → X, r → D						↑↓	↑↓	↑↓

Instruktionen **MUL** sätter C = 1 om bit 7 = 1 i resultatet (bit 7 i ackumulator B). På så vis kan man lätt avrunda resultatet i ackumulator A.

Exempel:

```
ADCA  #$00
```

Instruktionen **IDIV** står för "INTEGER DIVIDE". Kvoten fås i register X och resten i ackumulator D.

Instruktionen **FDIV** innebär "FRACTIONAL DIVIDE". Används då nämnaren är större än täljaren. Kvoten blir då i 65536:e-delar. Skulle nämnaren vara mindre än täljaren, sätts V = 1 som en varning.

FDIV kan med fördel användas till att dividera den rest som uppstått efter **IDIV**.

Exempel:

```
LDX   nämnare
IDIV
STX   hela
LDX   nämnare
FDIV
STX   delar
```

Vid division med 0 sätts C = 1.

3.2.6 Testning.

MNEMONIC	Förklaring	Adressering	Flaggor							
			S	X	H	I	N	Z	V	C
CMPA	A - M	# D E I					↑↓	↑↓	↑↓	↑↓
CMPB	B - M	# D E I					↑↓	↑↓	↑↓	↑↓
CBA	A - B						↑↓	↑↓	↑↓	↑↓
CPD	D - M:M+1	# D E I					↑↓	↑↓	↑↓	↑↓
CPX	X - M:M+1	# D E I					↑↓	↑↓	↑↓	↑↓
CPY	Y - M:M+1	# D E I					↑↓	↑↓	↑↓	↑↓
TST	M - 0	E I					↑↓	↑↓	**0**	**0**
TSTA	A - 0						↑↓	↑↓	**0**	**0**
TSTB	B - 0						↑↓	↑↓	**0**	**0**
BITA	A ∧ M	# D E I					↑↓	↑↓	**0**	
BITB	B ∧ M	# D E I					↑↓	↑↓	**0**	

Testinstruktioner påverkar bara flaggor.

COMPARE, **CMP** eller **CP**, utför egentligen en subtraktion men utan att ladda in resultatet i ackumulatorn.

Med hjälp av **TST**, som egentligen är en subtraktion med talet noll, kan man kontrollera om ett värde är noll och om det är negativt eller positivt.

BITA och **BITB** gör en bitvis AND på så sätt att enbart flaggorna påverkas.

Eftersom många andra instruktioner (såsom **LDAA** och **STAA**) påverkar flaggorna, är det inte alltid nödvändigt att använda testinstruktioner.

Exempel:

```
        LDAA  mem     Flaggor sätts.
        BEQ   lika


        LDD   MEM     Flaggor sätts för
*                     dubbelackumulatorns innehåll
        TSTA          Vi vill testa bara A:s innehåll.
        BEQ   lika
```

3.2.7 Programstyrning.

Instruktionerna i detta avsnitt används till att styra vilka vägar programmet ska ta.

Inga av dessa instruktioner påverkar några flaggor.

- **Vanliga villkorliga hopp.**

Dessa instruktioner tillåter oss att fatta beslut baserade på innehållet i flaggregistret.

Enkla flaggtester:

MNEMONIC	Förklaring		Adressering
BMI	Branch if Minus	**N = 1 ?**	R
BPL	Branch if Plus	**N = 0 ?**	R
BEQ	Branch if Equal	**Z = 1 ?**	R
BNE	Branch if Not Equal	**Z = 0 ?**	R
BVS	Branch if Overflow Set	**V = 1 ?**	R
BVC	Branch if Overflow Clear	**V = 0 ?**	R
BCS	Branch if Carry Set	**C = 1 ?**	R
BCC	Branch if Carry Clear	**C = 0 ?**	R

Positivt talområde:

MNEMONIC	Förklaring		Adressering
BHI	Branch if Higher	**acc > M ?**	R
BHS	Branch if Higher or Same	**acc ≥ M ?**	R
BEQ	Branch if Equal	**acc = M ?**	R
BLS	Branch if Lower or Same	**acc ≤ M ?**	R
BLO	Branch if Lower	**acc < M ?**	R

Positivt och negativt talområde (2-komplementrepresentation):

MNEMONIC	Förklaring		Adressering
BGT	Branch if Greater	**acc > M ?**	R
BGE	Branch if Greater or Equal	**acc ≥ M ?**	R
BEQ	Branch if Equal	**acc = M ?**	R
BLE	Branch if Less or Equal	**acc ≤ M ?**	R
BLT	Branch if Less Than	**acc < M ?**	R

Lägg märke till det stora urvalet av testkombinationer.

Observera att man måste testa för rätt talområde!

Ett vanligt programmeringsfel är att skriva **`BGT`** då man menar **`BHI`**. Felet avslöjas inte förrän det testade talet överskrider 7F (127).

- **Hopp efter test av bitmönster.**

MNEMONIC	Förklaring	Adressering
BRSET	Branch if Bits are Set	D I; R
BRCLR	Branch if Bits are Clear	D I; R

Detta är mycket kraftfulla instruktioner. De både testar och utför ett eventuellt hopp beroende på hur bitar i en minnescell är satta.

Observera att de bitar som ska testas alltid anges med ettor!

Tänk också på att alla angivna bitar måste vara ett- respektive nollställda, för att ett hopp ska ske.

Endast indexerad och direkt adressering finns. I exemplet nedan måste alltså mem ligga inom området 00 - FF.

Exempel:

```
BRCLR mem,$03,upp   Hoppa till 'upp' om de båda lägsta
                    bitarna (motsvarande bitmönstret
                    00000011) i minnescellen 'mem' är
                    nollställda.
```

- **Ovillkorliga hopp.**

MNEMONIC	Förklaring	Adressering
BRA	Branch Always	R
BRN	Branch Never	R
JMP	M:M+1→ PC	E I

BRA och **JMP** används som sista instruktion i huvudprogramslingan eller för att styra programmet till en speciell instruktion.

BRN, som innebär att ett hopp aldrig sker till angiven adress, kan tyckas något märklig, men kan vara användbar om man i en färdig maskinkod vill manipulera programvägarna. Instruktionen kan också användas som en s k NO OPERATION (se nästa sida). Om man betraktar operationskoden för varje BRANCH-instruktion, så ser man att den lägsta biten inverterar testvillkoret (**BRA** har koden $20 och **BRN** har koden $21).

JMP kan också adresseras med X eller Y. Detta ger en möjlighet att strukturera programkoden i hopptabeller.

Exempel:

```
        LDX   #jmptab       Första adressen i hopptabellen
        LDAB  uppgft        uppgift innehåller 0, 1 eller 2
        LSLB                Multiplicera med 2
        ABX                 Peka på rätt uppgift
        LDX   $0,X          Ladda X med adressen till uppgiften
        JMP   $0,X          Hoppa till det ställe som X pekar på

jmptab  FDB   Uppg0
        FDB   Uppg1
        FDB   Uppg2
```

- **Subrutininstruktioner.**

MNEMONIC	Förklaring	Adressering
BSR	Branch to Subroutine	R
JSR	Jump to Subroutine	E I
RTS	Return from Subroutine	

Ett subrutinhopp spar automatisk återhoppsadressen på stacken så att man i slutet av rutinen kan hoppa tillbaka till det ställe där man kallade på subrutinen. Stackpekaren minskas automatiskt med 2, så att den åter pekar på ledig plats.

Med subrutiner spar man del plats när samma sak ska utföras återkommande, och dels blir det lättare att dela in programmet i lätthanterliga block. Subrutinhopp kan adresseras ungefär som vanliga hopp; det finns även indirekt adressering.

`RTS` är det naturliga sättet att avluta en subrutin. Instruktionen hämtar det översta 16-bitarsvärdet från stacken till programräknaren och ökar stackpekaren med 2.

- **Diverse.**

MNEMONIC	Förklaring
NOP	No Operation

Instruktionen gör ingenting (utom att naturligtvis öka programräknaren så att nästa instruktion kan hämtas).

Den kan tex. användas till att ge en fördröjning på två maskincykler.

3.2.8 Instruktioner som handskas med flaggregistret.

Detta är instruktioner för att påverka flaggor.

MNEMONIC	Förklaring	Flaggor							
		S	X	H	I	N	Z	V	C
CLI	Clear Interrupt Mask Bit				**0**				
SEI	Set Interrupt Mask Bit				**1**				
CLV	Clear Overflow Bit							**0**	
SEV	Set Overflow Bit							**1**	
CLC	Clear Carry Bit								**0**
SEC	Set Carry Bit								**1**
TPA	Transfer CCR to A								
TAP	Transfer A to CCR	↑↓	↓	↑↓	↑↓	↑↓	↑↓	↑↓	↑↓

Endast tre av flaggorna har speciella instruktioner för nollställning och ettställning. Anledningen är att de övriga flaggorna antingen inte ska kunna påverkas så lätt (S och X), inte behöver påverkas (H påverkar instruktionen **DAA** efter en addition) eller lätt kan ändras av andra instruktioner. Instruktionerna **TPA** och **TAP** kan annars användas för att påverka vilka flaggor som helst. Notera att **TAP** ej kan ettställa X; detta kan bara ske av själva XIRQ-avbrottet.

Exempel:

Sätt på XIRQ-avbrottet.

```
TPA
ANDA  #%10111111
TAP
```

3.2.9 Avbrottshantering.

MNEMONIC	Förklaring	Flaggor							
		S	X	H	I	N	Z	V	C
RTI	Return from Interrupt	↑↓	↓	↑↓	↑↓	↑↓	↑↓	↑↓	↑↓
SWI	Software Interrupt				1				
WAI	Wait for Interrupt								

RTI är den instruktion som ska avsluta en avbrottsrutin. Den återställer innehållet i CPU-registren.

SWI är ett avbrott som begärs av programmet istället för övriga avbrott som begärs av händelser utanför programmet.

WAI utför det som alla avbrott har gemensamt, dvs. lagring av CPU-register och väntar sedan på att något avbrott ska ske. På så vis kan man få avbrottshanteringen att ske med minimal fördröjning. En bieffekt är att processorn drar extra lite ström medan den väntar.

I kapitel 4 förklaras vad avbrottshantering är.

3.2.10 Processorkontroll.

MNEMONIC	Förklaring
STOP	Stop clocks
TEST	

STOP är en speciell instruktion som helt enkelt stoppar processorns klocka så att den dramatiskt minskar sin strömförbrukning. Instruktionen kan bara utföras om S i flaggregistret = 0. Om S = 1, fungerar instruktionen som en **NOP**. Endast yttre avbrott eller reset kan lossa CPU:n från **STOP**.

TEST kan inte användas i vanliga program. Den används vid Motorolas tillverkningstest och får adressbussen att bete sig som en 16-bitars räknare.

3.3 Assemblerdirektiv.

För att styra assembleringen till rätt minnesutrymme och för att skapa kod som inte är instruktioner, används ett antal assemblerdirektiv. Här ges exempel på några av de viktigaste. Observera att olika assemblerare kan ha olika uppsättningar direktiv. Direktiven är alltså inte bundna till en viss processor och finns alltså inte i processorns manual.

Här förklaras några av de viktigaste.

ORG ORGINATE, placera koden.

Detta direktiv sätter ett värde på placeringspekaren, dvs. den pekare som används under assembleringspassset för att bestämma plats för den kod som skapas. Med **ORG** styr man t.ex. var programkoden och skrivminnet ska läggas. Man får se till att den processor man använder verkligen har sitt minne placerat där.

Exempel:

```
        ORG    $F800
        LDS    #$FF
```

EQU EQUATE, definiera symbol.

Med EQU ger man ett värde till en symbol. Det kan vara mycket värdefullt att i sin kod t.ex. slippa de absoluta adresserna till portar. Istället anger man portens namn och ökar därmed läsbarheten avsevärt.

Exempel:

```
IOBASE EQU    $1000
PORTC  EQU    IOBASE+3
```

FCB Form Constant Byte, skapa ett 8-bitars värde.

FCC Form Constant Character, skapa ett 8-bitars värde.

Detta är två ekvivalenta skrivsätt för att placera ett (eller flera) 8-bitars värden i minnesarean.

Exempel:

```
        FCB    $41
        FCC    'A
```

FDB Form Double Byte, skapa ett 16-bitars värde.

Direktivet används ofta när adressen till t.ex. en avbrottsrutin ska placeras på en speciell plats.

Exempel:

```
ORG   $FFFE
FDB   RESET
```

RMB Reserve Memory Bytes, reservera minnesutrymme.

Om man vill reservera ett visst utrymme i minnet, används detta direktiv. Placeringspekaren (som sätts med ORG), ökas med det antal byte som operanden anger.

Exempel:

```
Buffer RMB 10
```

END slut på koden.

Kod efter END-direktivet nonchaleras av assembleraren.

3.4 Programexempel.

Kapitel 3.2 kan tjäna som en uppslagsbok över instruktionerna, utifrån deras användning. Men ofta behövs bara ett fåtal för att lösa de vanligaste uppgifterna.

I kapitel 1 gick vi igenom några programexempel för en tänkt datormodell. Den har faktiskt stora likheter med den enchipsdator som denna bok fortsättningsvis handlar om, och vi ska därför skriva om dessa program för en 68HC11.

Nedanstående program motsvarar det i figur 9!

```
        ORG   $F800     Bestäm var koden ska ligga

        LDX   #$0000    Ladda X med värdet $0000.
Load    LDAA  $00,X     Ladda A-ackumulatorn med innehållet i
*                       den minnescell som X pekar på.
        STAA  $1004     Skicka ut det till port B.
Inc_X   INX             Öka X med ett.
        CMPA  $00,X     Jämför A med det X nu pekar på.
        BHS   Inc_X     Om det är högre eller samma: leta vidare.
        BRA   Load      Annars: gå till Load och läs in ett nytt.
```

Programmet letar efter det största 8-bitarstalet med start på adress 0. Utan indirekt adressering hade det varit praktiskt omöjligt att göra denna sökning.

Då hade det kunnat bli så här:

```
        ORG   $F800     Bestäm var koden ska ligga

        LDAA  $0000     Ladda A-ackumulatorn med innehållet i adress 0.
        STAA  $1004     Skicka ut det till port B.
        CMPA  $0001     Jämför A med innehållet i adress 1.
        BHS   Up1       Om det är högre eller samma: gå till Up1
        LDAA  $0001     Annars: läs in ett nytt tal från adress 1
        STAA  $1004     Skicka ut det till port B.

Up1     CMPA  $0002     Jämför A med innehållet i adress 2.
        BHS   Up2       Om det är högre eller samma: gå till Up2
        LDAA  $0002     Annars: läs in ett nytt tal från adress 2
        STAA  $1004     Skicka ut det till port B.

Up2     CMPA  $0003     Jämför A med innehållet i adress 3.
        BHS   Up3       Om det är högre eller samma: gå till Up3
        LDAA  $0003     Annars: läs in ett nytt tal från adress 3
        STAA  $1004     Skicka ut det till port B.

Up3     etc.
```

Alltid då man ska behandla data som ligger efter varandra, ska man välja indirekt adressering. Vi ser att redan när man ska undersöka 3 - 4 värden, blir annars programmet stort och klumpigt.

Det lilla programmet med indirekt adressering kommer att genomlöpa alla 65536 minnesadresser. Detta är nu inte särskilt användbart, utan man vill hellre att det ska avsluta sökningen vid en viss adress.

Då kan det se ut såhär:

```
       LDX   #$0000   Ladda X med värdet $0000.
Load   LDAA  $00,X    Ladda A-ackumulatorn med innehållet i
*                     den minnescell som X pekar på.
       STAA  $1004    Skicka ut det till port B.
Inc_X  INX            Öka X med ett.

       CPX   #$0080   Har X blivit = 80 ?
       BEQ   Stop     Stanna i så fall.

       CMPA  $00,X    Jämför A med det som X nu pekar på.
       BHS   Inc_X    Om det är högre eller samma: leta vidare.
       BRA   Load     Annars: gå till Load och läs in ett nytt.

Stop   BRA   Stop     Tvärnit.
```

Programmet kommer att söka efter det största talet i minnesarean 0 - 7F. Direkt efter instruktionen **INX** testar vi om X nått utanför området som ska genomsökas. **BEQ** kunde ha ersatts av **BHS** (Higher or Same) och till och med instruktionen **BGE** (Greater or Equal) eftersom värdet av X aldrig går utanför det positiva talområdet (den högsta biten i X är alltid = 0).

Man bör emellertid vara mycket uppmärksam på vilket talområde man handskas med.

Som det nu är skrivet söker programmet rätt på det största talet i betydelsen att FF är det största. Om vi byter ut **BHS Inc_X** mot **BGE Inc_X** beter sig programmet på ett annat sätt. Instruktioner som **BGE** tolkar ju tal mellan 80 och FF som negativa och kommer alltså inte att tycka att de är särskilt stora.

Såhär ser det ut då:

```
       LDX   #$0000   Ladda X med värdet $0.
Load   LDAA  $00,X    Ladda A-ackumulatorn med innehållet i
*                     den minnescell som X pekar på.
       STAA  $1004    Skicka ut det till port B.
Inc_X  INX            Öka X med ett.

       CPX   #$0080   Har X blivit = 80 ?
       BEQ   Stop     Stanna i så fall.

       CMPA  $00,X    Jämför A med det X nu pekar på.
       BGE   Inc_X    Om det är större eller lika: leta vidare.
```

```
        BRA   Load     Annars: gå till Load och läs in ett nytt.

Stop    BRA   Stop     Tvärnit.
```

Programmet har den nackdelen att det är fastlagt i vilket område som sökningen ska ske. I allmänhet kan man tänka sig att de grupper av värden där man vill leta upp det största, kan ligga på olika ställen och vara olika stora. Ibland vill man kanske söka bland 10 värden och ibland 50 värden på en annan plats i minnet.

Denna uppgift löser vi med att konstruera om sökprogrammet som en subrutin (en funktion eller procedur i högnivåspråk).

Vi ska kunna kalla på subrutinen med två olika parametrar:

1. Startadress för de värden som ska genomsökas.
2. Antal värden som ska genomsökas.

Om vi väljer att överföra dessa parametrar via X-registret och B-ackumulatorn, kan subrutinen se ut så här:

```
Search LDAA  $00,X    Ladda A-ackumulatorn med innehållet i
*                     den minnescell som X pekar på.
       STAA  $1004    Skicka ut det till port B.
Inc_X  INX            Öka X med ett.

       DECB           Räkna ner antalet.
       BEQ   Ready    Om det är noll, avsluta rutinen.

       CMPA  $00,X    Jämför A med det X nu pekar på.
       BHS   Inc_X    Om det är högre eller samma: leta vidare.
       BRA   Search   Annars: Läs in ett nytt.
Ready  RTS
```

När vi återvänder från subrutinen, finns det eftersökta värdet i ackumulator A och i port B.

Ett program som använder subrutinen kan se ut såhär:

```
        LDX   #$0000   Ladda X med värdet $0.
        LDAB  #$80     Antalet värden som ska testas.
        JSR   Search

Stop    BRA   Stop
```

3.5 Övningar.

a) Ändra i subrutinen Search (programexempel i föregående avsnitt) så att den söker rätt på det minsta talet i stället.
Vilken ändring behövs om alla värden tolkas som positiva ?
Vilken blir ändringen om vi antar att värdena har 2-komplementrepresentation?

b) Skriv ett program som nollställer skrivminnet (RAM) i en HC11.
Skrivminnet börjar på adress 0 och slutar på adress FF.

c) Skriv om uppgift 2 som en subrutin som nollställer ett visst antal minnesplatser (antalet ska finnas i B) från och med den adress som finns i X.
Använd idéerna från subrutinen Search!

d) Skriv ett program som kopierar ett block om 256 byte från adress E000 till adress 0.

e) Gör om föregående uppgift till en generell subrutin.
Bestäm själv vilka parametrar som kan vara lämpliga att förse subrutinen med.

Normalt vinner man mycket på att generalisera uppgifter. En subrutin som den i uppgift 5 kan ju användas oberoende av hur många tal som ska undersökas och oberoende av var de ligger. Det är bara att förse rutinen med de rätta parametrarna.

Jämför med hur funktioner och procedurer fungerar i högnivåspråk.

4 Avbrottshantering.

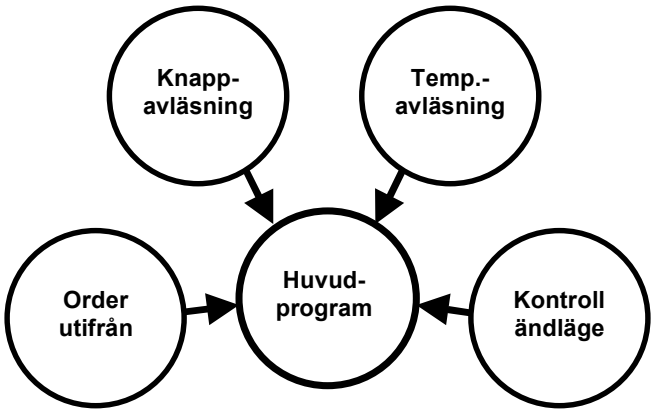
Avbrott är en inbyggd mekanism som gör det möjligt att bygga upp programdelar som fungerar oberoende av det övriga programmet.

Med avbrott kan programkörningen styras bort från sin 'normala' verksamhet till en bestämd rutin när en speciell händelse inträffar.

En händelse som alltså absolut inte får missas kan tas om hand direkt. Miljön kring en enchipsdator innefattar ofta sådana händelser.

Några exempel:

- En temperatur ska övervakas och måste kännas av tillräckligt ofta så att en reglering ska kunna ske.
- En ändlägesgivare ger en signal som omedelbart måste resultera i att en motor stängs av.
- En order från en annan dator måste tas emot och verkställas omedelbart.
- Ett antal tryckknappar ska avläsas tillräckligt ofta för att ingen ska missas.



Figur 15. Avbrottssystem.

En stor fördel med avbrottssystem är att man kan låta en och samma dator utföra flera jobb som kan fungera som fristående uppgifter.

Knappavläsning och mottagning av order utifrån kan kräva en hel del program för att fungera. De kan då byggas upp som fristående s.k. **komponentprogram** (se kapitel 9.6), och av huvudprogrammet betraktas som yttre kretsar som utför sitt arbete oberoende av andra program.

Händelser som kräver avbrottshantering kan vara av två olika slag:

1. **Händelsen genererar själv sitt avbrott.**
   Fördel: Snabb reaktion på händelsen.
   Nackdel: Signalen behöver snyggas upp; knappstudsar måste exempelvis tas bort.
2. **Händelsen övervakas med jämna mellanrum av ett periodiskt avbrott.**
   Fördel: Signaler som avläses blir automatiskt filtrerade.
   Nackdelar: Tar onödig tid; ofta händer kanske inget. Man får ingen omedelbar reaktion på händelsen.

I vårt exempel kan order utifrån och ändlägesdetektering skötas av den första typen av avbrott, medan knappavläsning och temperaturavläsning med fördel sköts i periodiskt avbrott.

Fler händelser än man tror kan tas om hand med övervakning i periodiskt avbrott.

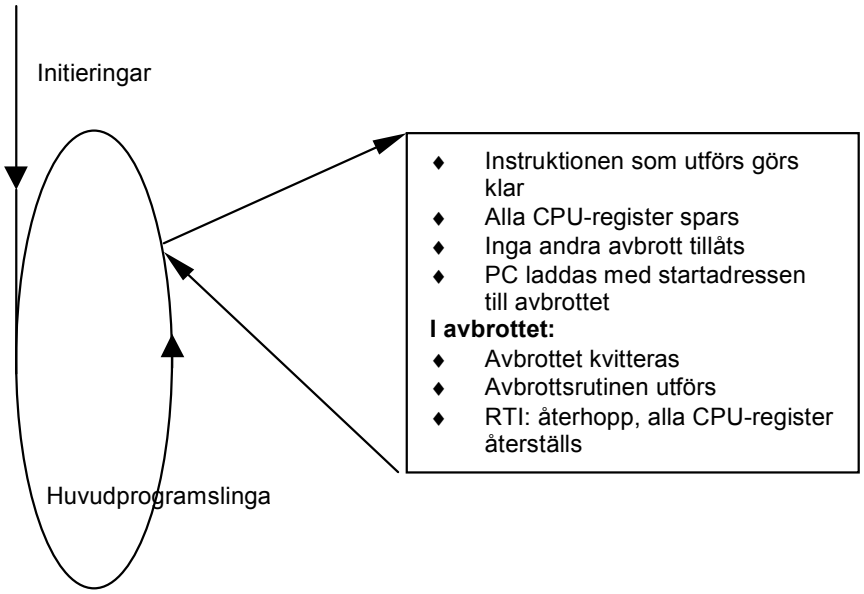
4.1 Hur avbrott fungerar i HC11.

Avbrottsmekanismen kräver att processorn ska kunna lämna sina normala program och utföra det som avbrottsprogrammet kräver. När sedan avbrottsprogrammet är slut ska processorn återvända till normal verksamhet; det enda som ska märkas är att det tagit lite tid.

Jämför med avbrottshantering i det verkliga livet:

Du sitter och jobbar med något och någon kommer in: "Hjälp mej med det här". Du lägger undan det du håller på med, lägger det i en trave på skrivbordet och får instruktioner om vad du ska göra. När du är klar, plockar du fram det du jobbade med och fortsätter där du slutade.

Detta liknar den mekanism som används i många datorer för att sköta avbrottshanteringen: När avbrott begärs (och godkänns), spar man på en stack innehållet i CPU:ns register.



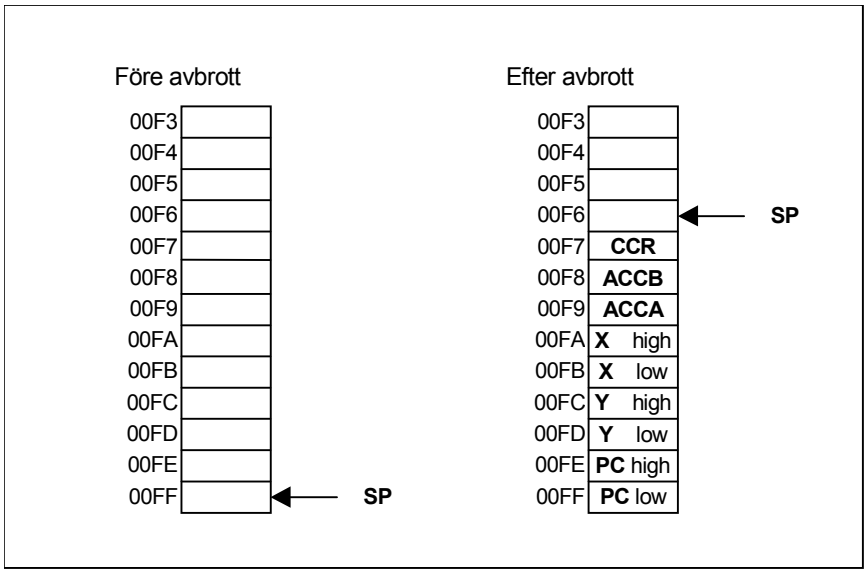
Figur 16. Hur avbrott hanteras.

När avbrottsprogrammet är färdigt, återställs alla CPU-register från stacken och man återvänder till det avbrutna programmet.

I HC11 sker lagringen av PC och alla CPU-register helt automatiskt.

I slutet av ett avbrottsprogram använder man instruktionen RTI som sköter återladdningen av alla register.

Lägg märke till att stackpekarens värde (naturligtvis) inte spars på stacken. Stackpekaren ska ju förhoppningsvis inte få ändra sitt värde okontrollerat; den är ju en garant för att programmet återvänder till rätt ställe med rätt innehåll i CPU.



Figur 17. Stackens användning vid avbrott.

För att ett avbrott ska ske, krävs tre saker:

1. **Det enskilda avbrottet är påslaget.** Nästan alla avbrottskällor har en separat påslagnings- och avstängningsbit.
2. **I-biten i CCR = 0.** Detta innebär att avbrottssystemet över huvud taget är inkopplat.
3. **Det enskilda avbrottets flagga har satts.** Avbrottets flagga hör samman med den händelse som kan utlösa avbrott. Den sätts vid händelsen även om avbrottet inte är påslaget, och måste kvitteras (nollställas) av avbrottsrutinen om man vill ha avbrott först vid nästa händelse.

Instruktionerna som körs på grund av en avbrottsbegäran liknar ett subrutinanrop. Skillnaden är, att instruktionerna utförs helt automatiskt direkt efter att avbrott begärts och ytterligare avbrott hindras att avbryta genom att I-biten i CCR sätts = 1. CPU:s register lagras på stacken varefter PC laddas med den adress som hör till detta avbrottet.

Tydligen krävs också att ett visst utrymme av skrivminne finns tillgängligt. Varje avbrott behöver 9 byte tillfälligt lagringsutrymme.

Metoden att låta varje avbrottskälla styra programmet direkt till den programkod som hör till detta avbrott, kallas vektoriserat avbrott. Nästan alla avbrottskällor i HC11 har sin egen vektor. Undantaget finns beskrivet i kapitel 5.6. Utan vektoriserat avbrott, måste avbrottsrutinen inledningsvis undersöka vilket avbrott som skett och därpå köra det program som hör till detta avbrott. Vektoriserat avbrott gör naturligtvis avbrottshanteringen snabbare, vilket är viktigt i många sammanhang.

68HC11 är ju ett komplett datorsystem med en mängd inbyggda enheter. De flesta av dem kan begära avbrott.

När ett avbrott begärs (beroende på att något händer i periferienheten), och det aktuella avbrottet tillåts, avslutas den instruktion som håller på att utföras, CPU-innehållet spars och programräknaren laddas med det värde som finns på en förutbestämd plats i minnet. Dessa platser är bestämda av HC11-tillverkaren och ligger från adress FFD6 och uppåt.

Prioritet	Avbrott	Vektoradress
1	Interrupt Request (IRQ)	FFF2
2	Real Time Interrupt (RTI)	FFF0
3	Input Capture 1 (IC1)	FFEE
4	Input Capture 2 (IC2)	FFEC
5	Input Capture 3 (IC3)	FFEA
6	Output Compare 1 (OC1)	FFE8
7	Output Compare 2 (OC2)	FFE6
8	Output Compare 3 (OC3)	FFE4
9	Output Compare 4 (OC4)	FFE2
10	Output Compare 5 (OC5)	FFE0
11	Timer Overflow (TO)	FFDE
12	Pulse Accumulator Overflow (PAOV)	FFDC
13	Pulse Accumulator Input (PAI)	FFDA
14	Serial Periferal Interface (SPI)	FFD8
15	Serial Communication Interface (SCI)	FFD6
16	Software Interrupt (SWI)	FFF6

Tabellen ovan upptar alla normala avbrottskällor i den prioritetsordning som finns förutbestämd. Det kan ju hända att två eller flera avbrott begärs samtidigt och då sker inte tilldelningen av exekveringstid slumpmässigt utan i en viss ordning. Observera att prioritetsordningen inte tillåter ett avbrott att utan vidare avbryta ett annat; detta är något man i regel undviker eftersom det lätt gör programmen "ömtåliga".

Ibland kan man ändå vara tvungen att tillåta att ett avbrott avbryts av ett annat. Då används instruktionen CLI. Å andra sidan kan instruktionen SEI användas i anslutning till avsnitt i huvudprogramslingan som absolut inte får bli avbrutna.

I kapitel 6 finns ytterligare avbrottskällor beskrivna. Dessa är av mer drastisk natur och deras användning används för att säkerställa funktionen hos datorsystemet.

Alla avbrottskällor utom IRQ och SWI hör samman med de inbyggda periferienheterna, och beskrivs under resp. avsnitt i kapitel 5.

IRQ och SWI beskrivs i slutet av detta kapitel, avsnitten 4.5 och 4.6.

Vill man av någon anledning ändra på den inbyggda prioritetsordningen (varför skulle man just vilja ha den just såhär), så kan lyfta upp ett av avbrotten till högsta prioritet. Detta sker i ett speciellt register, HPRIO, se databoken.

4.2 Instruktioner för att hantera avbrott.

Avbrottsrutiner kan uppfattas som subrutiner som man inte kallar på i programmet. Istället är det en händelse utanför programexekveringen som tvingar programmet att köra avbrottsrutinen.

Det är alltså ingen i förväg planerad verksamhet: "när jag har gjort de här sakerna klart, så ska jag göra detta". Som namnet visar blir man istället avbruten i sitt normala arbete för att utföra den sekvens av instruktioner som finns i avbrottet.

(Det finns emellertid ett undantag: instruktionen SWI som förklaras i avsnitt 4.6.)

Däremot finns det instruktioner för att tillåta eller förbjuda avbrott. Det är viktigt, eftersom en avbrottskälla (som vi senare kommer att se) ofta måste ställas in på ett visst sätt för att fungera som vi vill ha den. Innan detta är gjort, vill vi ännu inte ha avbrott. Det kan också finnas tillfällen då man måste se upp med att inte bli avbruten av ett annat program.

- **CLI** Clear Interrupt Mask. Används för att tillåta avbrott. Ofta används instruktionen bara en gång i ett program: när initieringarna är utförda. Eftersom I-biten i CCR från uppstart alltid är ettställd, kan alla inställningar göras i lugn och ro innan avbrott tillåts. Instruktionens verkan är fördröjd med en maskincykel så att den efterföljande instruktionen alltid kommer att utföras även om avbrott redan väntar. Denna mekanism har betydelse i samband med instruktionen WAI (se nästa sida) så att denna instruktion kan utföras direkt efter CLI. Instruktionen behöver inte användas i slutet av en avbrottsrutin eftersom RTI återställer I som den var före avbrottet. CLI används ibland för att tillåta andra avbrott i en avbrottsrutin.
- **SEI** Set Interrupt Mask. Används för att stänga av avbrott. Används tillsammans med CLI kring instruktionssekvenser som absolut inte får störas av ett avbrott. Instruktionen behöver inte användas för att förhindra andra avbrott i en avbrottsrutin; det sker automatiskt.

Instruktionen **TAP** kan alternativt användas istället för CLI och SEI. Exempelvis utför sekvensen

```
TPA
ANDA  #%11101111
TAP
```

samma sak som instruktionen **CLI** .

Avbrottsrutiner startas ju normalt av en inbyggd mekanism som styr programmet till rätt ställe. När rutinen är slut, vill man ha det omvända förfarandet, dvs. återladdning av alla register och återhopp till det ställe man kom ifrån. Egentligen skulle man kunna utföra detta med en serie instruktioner men man får en större snabbhet om man använder

- **RTI** Return from Interrupt. Återställer alla CPU-register med innehåll från stacken. Används för att avsluta en avbrottsrutin.

När ett avbrott begärs, tar det alltid en viss tid innan själva avbrottsrutinen kan exekveras. I bästa fall utförs inget annat avbrott utan programmet befinner sig bland instruktioner som när som helst får bli avbrutna. Ofta händer det att programmet inte har något alls att göra utan bara står berett att utföra uppgifter som avbrottsrutiner kräver. Om det i ett sådant fall är oerhört viktigt att avbrottsrutinens arbete kommer igång omedelbart, kan datorn stå och vänta med en speciell instruktion, WAI.

Eftersom den instruktionen ska kunna invänta vilket avbrott som helst, kan man låta den utföra det som är gemensamt för alla avbrott: undanlagring av CPU-register på stacken.

- **WAI** Wait for interrupt. Lagrar alla CPU-register på stacken och väntar på avbrott. Används för att minimera tiden för avbrottshantering.

En speciell egenskap hos instruktionen är att processorns strömförbrukning minskas till hälften mot normalt.

4.3 Initiering av avbrott

Som vi tidigare sett, är det tre saker som ska vara uppfyllda för att ett avbrott ska ske:

1. **Det enskilda avbrottet är påsatt.**
2. **I-biten i CCR = 0.**
3. **Händelsen som förorsakar avbrottet har skett.**

Punkt 1 och 2 kräver åtgärder från programmet; dvs. vi måste se till att vissa bitar är inställda på rätt sätt. Ofta (som i exemplet nedan) vill man att avbrottskällan ska fungera på ett speciellt sätt.

Subrutin som ställer in ett enskilt avbrott:

```
Init_RTI  LDAA  PACTL
          ORAA  #%00000011   Periodiskt avbrott var 30 ms
          STAA  PACTL
          LDAA  TMSK2
          ORAA  #%01000000   Sätt på RTI-avbrottet
          STAA  TMSK2
          RTS
```

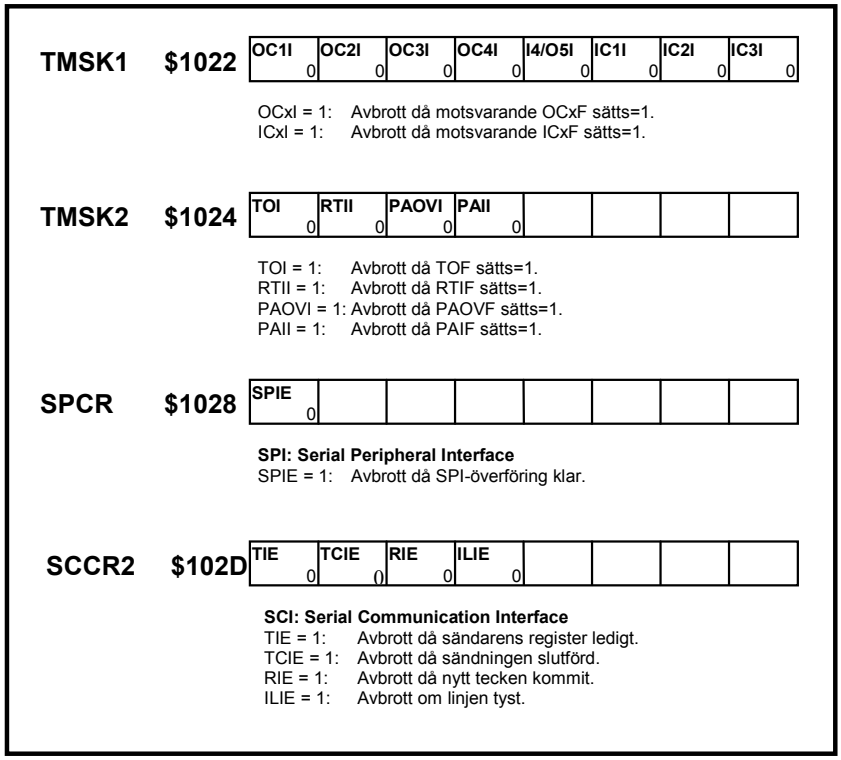
Nollställ I-biten:

```
          CLI
```

Exemplet visar hur det periodiska avbrottet ska initieras för att komma igång med en periodtid på ungefär 30 ms (8MHz kristall).

I språket C blir motsvarande:

```
void Init_RTI (void) {
    PACTL |= 0x03;
    TMSK |= 0x40;
}

enable_interrupt();
```



Figur 18. Register för påslagning av avbrott.

Ovanstående figur är en sammanställning över alla de enskilda avbrottens påslagningsbitar. De två översta registren, TMSK1 och TMSK2, har med timersystemet att göra; de två understa, SPCR och SCCR2, styr avbrotten från de båda inbyggda seriekanalerna.

Kapitel 5 beskriver utförligt för alla periferienheter hur inställningarna ska vara, för att man ska kunna använda avbrotten.

4.4 Avbrottsrutinens uppbyggnad.

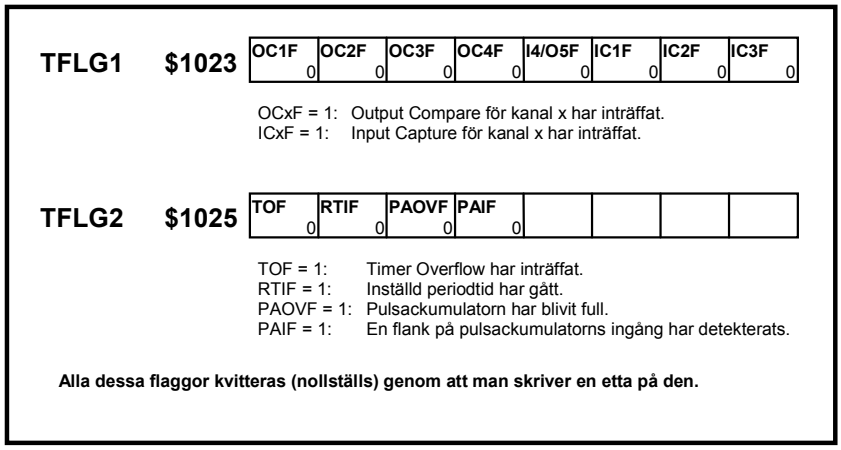
Varje händelse som kan ge avbrott, förorsakar att en speciell bit (flagga) sätts. Även om avbrottet inte är påslaget, sätts denna flagga. Meningen med detta är att man ska kunna övervaka händelsen utan att använda avbrott.

Om vi däremot använder avbrott, måste vi vara noga med att kvittera denna flagga. Om den inte kvitteras, kommer händelsen på nytt att orsaka avbrott direkt efter det att vi avslutat avbrottsrutinen.

Normalt vill man inte använda avbrottet på det sättet, utan väljer att omedelbart vid inträdet till avbrottsrutinen göra kvittensen.

I 68HC11 kvitteras (nollställs) en händelseflagga genom att skriva en etta på den. (Seriekanalerna utgör undantag och kvitteras helt enkelt genom att använda dem.)

Det är alltså timersystemets flaggor som fungerar så här, och de ligger samlade i två register, TFLG1 och TFLG2.



Figur 19. Register för kvittering av händelser.

Förutom kvitteringen, är en speciell instruktion obligatorisk i en avbrottsrutin, nämligen RTI. Den sköter ju återladdningen av alla CPU-register så att avbrottet ej ska lämna några onödiga spår efter sig.

En typisk avbrottsrutin för en av timersystemets avbrottskällor kan alltså se ut såhär:

```
RTI_interrupt  LDAA  #%01000000   Kvittera RTI-avbrottet
               STAA  TFLG2
               ............
               ............
               ............       Övrig kod
               ............
               ............
               RTI
```

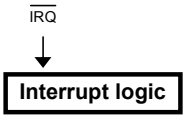
Motsvarande i C:

```
interrupt void RTI_interrupt( void) {
    TFLG2 = 0x40;
    ............
    ............
    ............       Övrig kod
    ............
    ............
}
```

Om vi sammanfattar den kod som behövs generellt för ett avbrott, kan vi särskilja följande tre avsnitt:

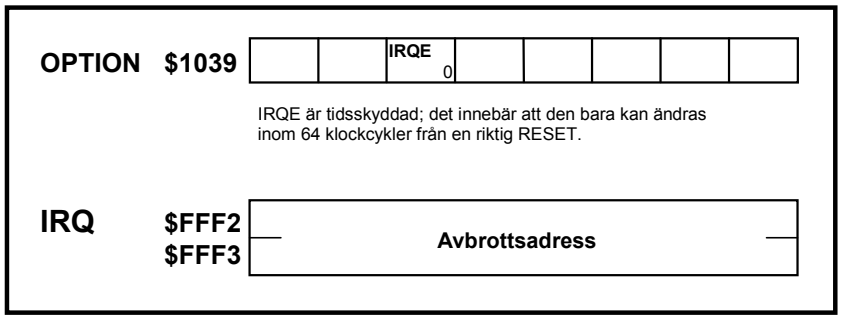
1. **Initiering av avbrott.** Varje avbrott har en lokal påslagningsbit som sätts =1. Dessutom behövs ibland vissa inställningar göras så att avbrottet beter sig som önskat.
   När alla programmets initieringar är gjorda, utförs **CLI** som tillåter avbrott över huvud taget.
2. **Avbrottsrutinen.** Alla avbrott måste kvitteras på något sätt. Man återvänder med instruktionen **RTI**.
3. **Avbrottsvektorn.** Varje avbrott måste ha sin startadress utplacerad på ett speciellt ställe.

4.5 IRQ, Interrupt Request



Förutom avbrott från de inbyggda periferienheterna, finns det möjlighet att använda externa avbrottskällor. Dessa kopplas då vanligen till ingången IRQ.

Egentigen finns det ytterligare signaler som kan användas som avbrottsingångar; dessa är timersystemets IC-ingångar och pulsackumulatoringång samt STRA som är strobsignal för parallell kommunikation. Se vidare kapitel 5.3, 5.5 och 5.9.



Figur 20. Användning av IRQ.

Observera att IRQ-avbrottet inte har någon speciell påslagningsbit. Vill man inte använda det, men utnyttjar något annat internt avbrott, måste insignalen IRQ läggas till en logisk etta (direkt eller via motstånd). Detta är ganska naturligt eftersom de periferienheter som använder IRQ, normalt har sin egen påslagning i ett kontrollregister.

Man kan använda IRQ både som nivåkänslig och flankkänslig avbrottsingång. Detta går att bestämma med en bit IRQE (E = edge) i ett kontrollregister som heter OPTION. Vid uppstart är denna bit = 0, dvs. IRQ är en nivåkänslig ingång som genererar avbrott så länge signalen är = noll. Vill man istället att en negativ flank ska generera avbrott, måste IRQE ettställas inom de första 64 klockcyklerna efter reset. (Se kap 7).

Vanligen gör man IRQ nivåkänslig, eftersom man då kan koppla ihop många avbrottskällor till denna ingång. I avbrottsrutinen får man då testa vilken enhet som begärt avbrott, och utföra vad som krävs för att detta avbrott ska kvitteras. Finns sedan avbrottsbegäran från annat håll, genereras nytt avbrott direkt efter slutförd avbrottsrutin. Detta avbrott betjänas då, och inte förrän alla avbrottskällor blivit inaktiva, kan processorn återgå till annan verksamhet. Metoden att koppla ihop flera avbrottssignaler till en enda ingång, fungerar bara om avbrottssignalerna är av typ öppen kollektor; varje enhet genererar en aktiv nolla, men ingen etta. I inaktivt läge lyfts spänningen upp till etta med ett motstånd.

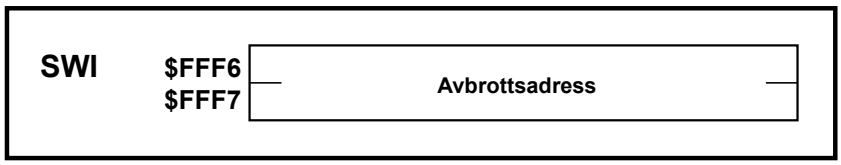
IRQ delar funktion med avbrottsbegäran från det parallella gränssnittet med PORTC och STRA. En speciell flagga sätts, så att rätt avbrott kan identifieras.

Om denna avbrottskälla ska användas, finns naturligtvis en påslagning för den. Se kap 5.9.

4.6 SWI, Software interrupt

SWI är en instruktion som fungerar som ett avbrott.

Normalt begärs ju avbrott av händelser som inträffar oberoende av programmet. Med instruktionen SWI har man möjlighet att avbryta programmet på samma sätt som sker med avbrott, dvs. alla register lagras automatiskt och inga andra avbrott tillåts. Instruktionen används ofta av felsökningsprogram (debuggers) på så sätt att brytpunkter sätts genom att ersätta en viss instruktion med SWI. I avbrottsrutinen som hör till SWI kan då programmets status undersökas genom att alla register finns sparade på stacken.



Figur 21. SWI.

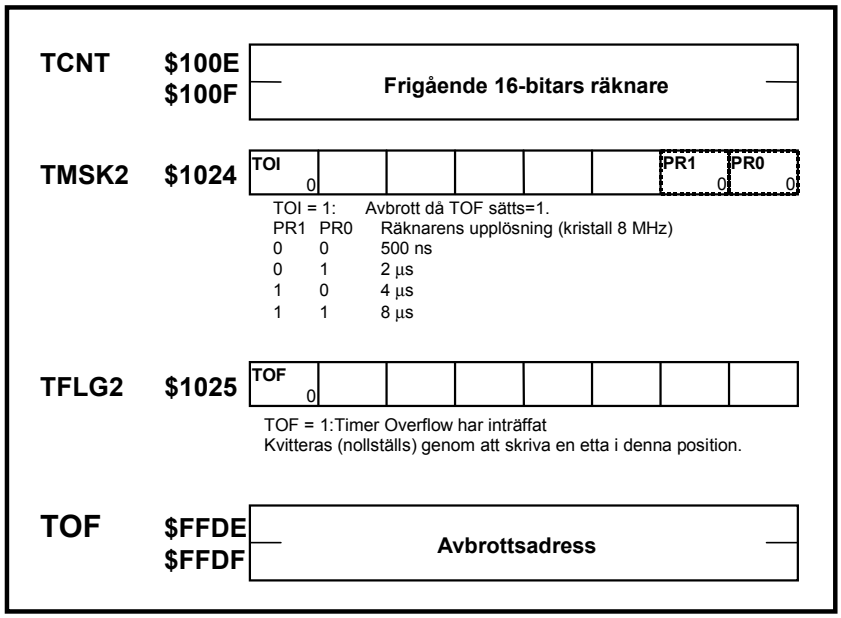
5 Inbyggda periferienheter.

5.1 Timer/Counter

HC11:an har en frigående 16-bitars räknare som utgör tidbas för alla enheter som ska kunna mäta eller kontrollera exakta tider.

Den går inte att stoppa, men kan programmeras att arbeta i en av fyra olika hastigheter (i förhållande till kristallfrekvensen).

En overflowfunktion gör att räknaren lätt kan 'skarvas' till ex.vis 24 bitar.



Figur 22. Användning av TCNT.

Räknaren TCNT kan läsas av när som helst i programmet med någon av instruktionerna LDD, LDX eller LDY. Dessa instruktioner läser ju ur två på varandra följande adresser och naturligtvis kan räknaren råka ändras emellan två sådana läsningar. Om räknaren ändrar sig från tex. $30FF till $3100, skulle man ha läst av det felaktiga $3000. Nu har man byggt in en mekanism som förhindrar detta: när man läser den höga halvan av TCNT låses den låga halvan i ett speciellt register under den busscykel den avläses av instruktionen.

TCNT sätts till $0000 vi reset. Normalt kan man annars inte påverka räknarens värde. Men då HC11:an befinner sig i TEST-mod, kan man sätta räknaren till $FFF8 med en skrivning (oavsett vad man försöker skriva). TEST-moden finns beskriven i kapitel 7, och används dels för testning hos

tillverkaren (Motorola) och dels hos den som konstruerar med HC11:an, för att kunna testa en färdig konstruktion.

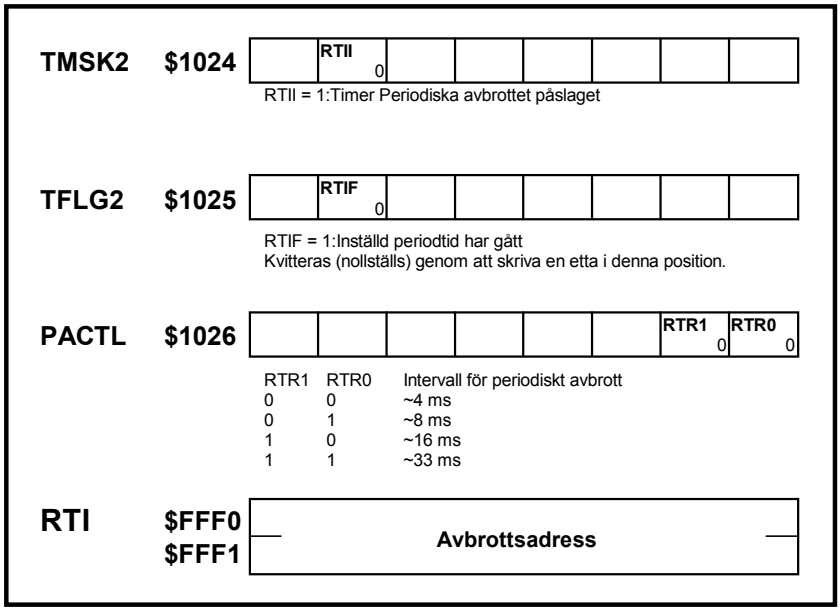
Med hjälp av avbrott vid overflow kan räknaren "skarvas". Overflowflaggan sätts varje gång räknaren går från $FFFF till $0000. Om TOI är satt, tillåts avbrott när detta händer. Timer overflow-avbrott kan också användas till att ge periodiska avbrott med ganska långa intervall (med 8 MHz kristall upp till ungefär 0,5 sekunder).

Räknarens upplösning bestäms av bitarna PR1 och PR0. Med dessa kan man välja fyra olika frekvenser för timern. Detta påverkar alltså i någon mån strömförbrukningen men viktigast är dock att se till att de pulslängder som hanteras (med ex.vis output compare) håller sig inom räknarens periodtid. Låter man räknaren gå onödigt fort, måste man ta hänsyn till overflow, vilket ställer extra krav på programmet. Se vidare kapitel 5.3 och 5.4.

PR1 och PR0 är tidsskyddade, vilket innebär att de bara kan påverkas under de första 64 klockcyklerna från reset. De nollställs vid reset, och får alltså opåverkade samma räknehastighet som systemklockan.

5.2 Periodiskt avbrott.

Det periodiska avbrottet (eller RTI, som det heter i databoken) kan användas för att ge avbrott med bestämda intervall. Avbrottet styrs av en frekvensdelare som är kopplad till HC11:ans systemklocka.



Figur 23. Periodiska avbrottet, RTI.

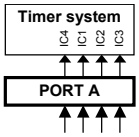
Intervallet kan programmeras till fyra olika tider genom att ställa in två bitar i registret PACTL.

Efter reset kommer det första periodiska avbrottet efter den förinställda tiden.

Biten RTIF sätts periodiskt med det inställda intervallet. De olika intervallen varierar ungefär från 4 till 33 millisekunder. Avsikten med detta avbrott är inte att man ska ha en mycket exakt periodtid, utan snarare att man med jämna, lagom stora intervall ska kunna övervaka utifrån kommande signaler.

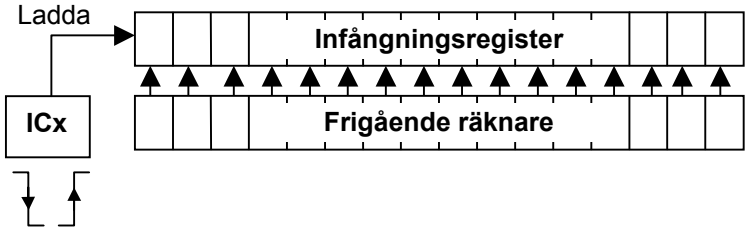
Ett typiskt användningsområde är avsökning av tryckknappar. Detta kräver ju inga exakta intervall, men bör göras tillräckligt ofta (för att inte missa en nedtryckning) och tillräckligt sällan (för att eliminera verkan av tangentstuds).

5.3 Input Capture



Input Capture är en metod för exakt mätning av tiden mellan två händelser. Den är uppbyggd av en flankdetektor och ett 16-bitars infångnings (capture)-register. När en ingång ändrar nivå händer följande:

- Den frigående räknarens värde fångas upp av infångningsregistret.
- En flagga sätts.
- ***Eventuellt**: En avbrottsrutin startar.*



Figur 24. IC-funktionen.

HC11 har fyra ingångar som oberoende av varandra kan detektera en nivåförändring på det här sättet. I registret TCTL2 kan man ställa in vilka nivåförändringar som ska avkännas.

Nedanstående programexempel illustrerar hur Input Capture kan användas för att mäta en pulslängd (som är kortare än 65536 maskincykler).

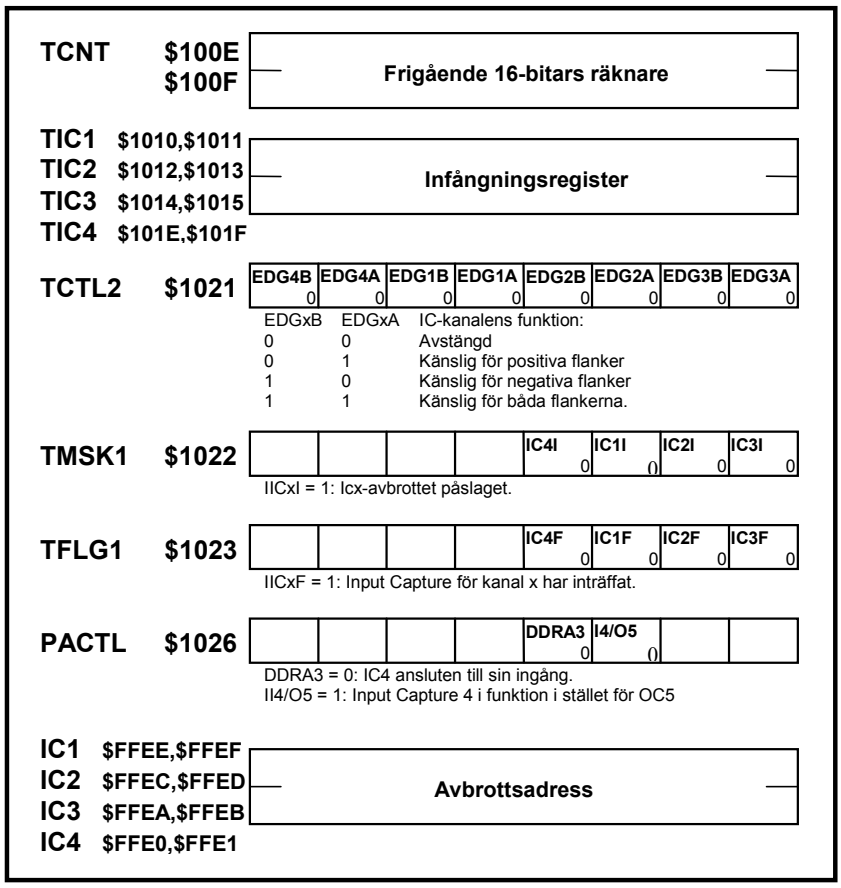
```
*  Subrutin StartM, Start measurement. Startar kontinuerlig mätning
*  av positiv puls på IC3.
StartM  LDAA  TCTL2
        ANDA  #%11111101
        ORAA  #%00000001
        STAA  TCTL2       IC3 känslig för positiv flank
        LDAA  TMSK1
        ORAA  #%00000001
        STAA  TMSK1       Sätt på avbrottet
        RTS

*  Avbrottsrutin IC3, mätning av pulslängd.
*  Pulslängden finns i 'Width'.
IC3     LDAA  #%00000001
        STAA  TFLG1       Kvitterar flaggan.
        LDX   TIC3        Läs tid för pos. flank.
        LDAA  TCTL2
        EORA  #%00000011  Ställ om för avkänning
        STAA  TCTL2       av den andra flanken.
        BITA  #%00000001
        BNE   Ner         Testar vilken flank.
        STX   Upp         Lagra tiden för upp.
```

```
        RTI

Ner     XGDX
        SUBD  Upp         Beräkna
        STD   Width       pulslängden.
        RTI
```

Med en något annorlunda avbrottsrutin kan periodtiden automatiskt mätas:

```
*  Avbrottsrutin IC3, mätning av periodtid.
IC3     LDAA  #%00000001
        STAA  TFLG1       Kvitterar flaggan.
        LDX   TIC3        Läs tid för pos. flank.
        XGDX
        SUBD  Previous    Beräkna skillnaden.
        STD   Period      Lagra den.
        STX   Previous    spara den nya tiden.
        RTI
```

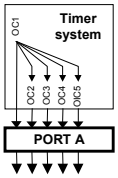
Mycket korta pulser blir svårt att mäta med ovanstående program; avbrottsrutinens längd sätter en gräns för vad som blir mätbart. Men med hjälp av två ingångar, varav den ena detekterar uppflanken och den andra nedflanken, går det att mäta pulser lika korta som räknarens upplösning. Upplösningen kan programmeras på fyra olika sätt. Med den grövsta fås noggrannheten 8 µs. Detta ger maximal mätbar pulslängd = 0,52 s. Bitarna som ställer upplösningen är tidsskyddade och kan inte när som helst ändras. Se kap 7.



Figur 25. Input Capture.

För mätning av långa tider krävs att man tar hänsyn till overflow och "skarvar" timern med ytterligare 8 eller 16 bitar. Man måste ta speciell hänsyn till i vilken ordning overflow och Input Capture kommit. Mer om detta problem finns diskuterat i M68HC11 Reference Manual.

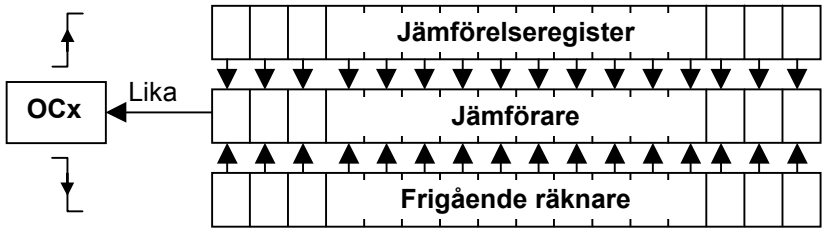
5.4 Output Compare



Detta avsnitt beskriver ***inte*** Output Compare 1. Den har en något annorlunda funktion och beskrivs i nästa avsnitt.

Output Compare har den omvända funktionen jämfört med Input Capture: Ett förinställt 16-bitarsvärde jämförs med den frigående räknaren; när de stämmer överens händer följande:

- En flagga sätts.
- ***Eventuellt:*** *En utgång kan fås att ettställas, nollställas eller växla.*
- ***Eventuellt****: En avbrottsrutin startar.*



Figur 26. OC-funktionen.

HC11 innehåller fyra OC-funktioner som fungerar på ovanstående sätt: OC2, OC3, OC4 och OC5 (OC1 har ett något annorlunda beteende, se nästa avsnitt). Bitarna i register TCTL2 bestämmer vad som ska hända vid överensstämmelse.

Ett exempel på när man kan använda enbart avbrottsrutin (utan utmatning), är då man vill ha ett periodiskt återkommande programavsnitt (för att hålla reda på tiden). Här visas hur man använder OC3 för att få avbrott var 10:e ms.

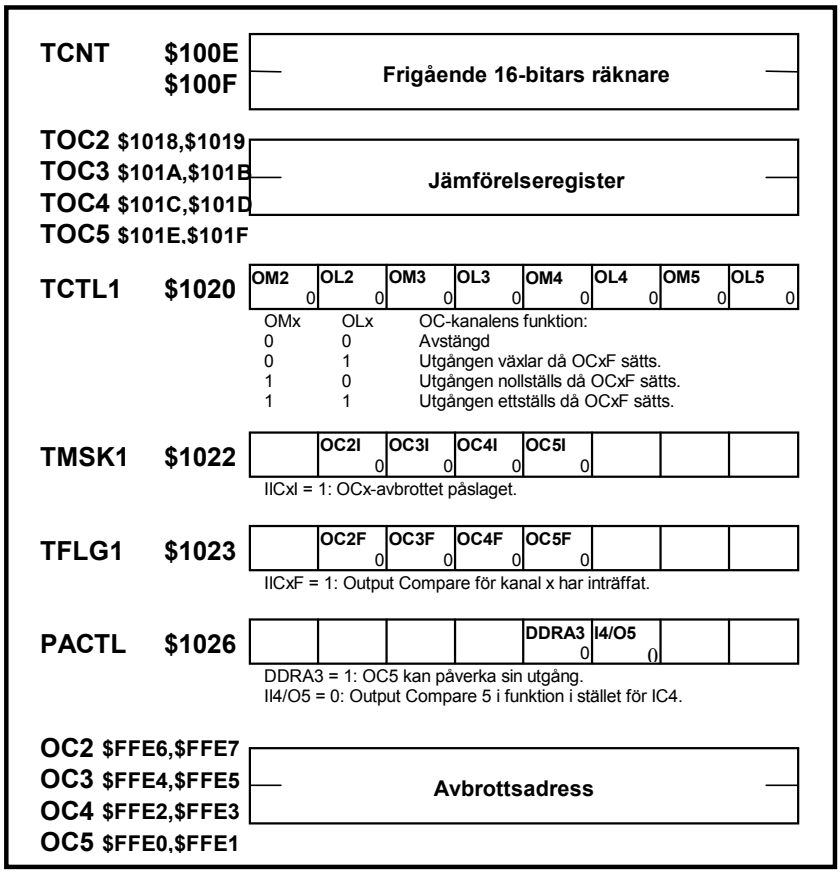
```
* Subrutin InitOC3.

InitOC3 LDAA  TMSK1
        ORAA  #%00100000         Sätt på avbrottet.
        STAA  TMSK1
        RTS

*  Avbrottsrutin OC3.
OC3     LDAA  #%00100000
        STAA  TFLG1              Kvitterar flaggan.
        LDD   TOC3               Läs tidpunkt som utlöste
*                                detta avbrott.
        ADDD  #20000             Addera värde
*                                som motsvarar 10 ms
        STD   TOC3               Skriv tillbaka så att nästa
*                                avbrott sker vid den tidpunkten
```

```
        RTI
```

Man kan lätt modifiera ovanstående så man exempelvis får en programmeringspuls till EEPROMet på 10 ms. Se avsnitt 7.5.

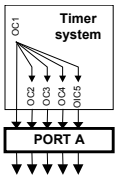
Nedanstående avbrottsrutin genererar en fyrkantsvåg på PA6 med frekvensen 1 kHz. PA6 kontrolleras av OC2. 1 kHz motsvarar 1 ms, och utgången ska alltså växla polaritet var 500:e µs.

```
*  Subrutin InitOC2.
InitOC2 LDAA  TMSK1
        ORAA  #%01000000          Sätt på avbrottet.
        STAA  TMSK1
        LDAA  TCTL1
        ORAA  #%01000000          Ställ in för växling
        ANDA  #%01111111
        STAA  TCTL1
        RTS

*  Avbrottsrutin OC2.
OC2     LDAA  #%01000000
        STAA  TFLG1               Kvitterar flaggan.
        LDD   TOC2                Läs tidpunkt som utlöste
*                                 detta avbrott.
        ADDD  #1000               Addera till räknevärde
*                                 som motsvarar 0.5 ms
        STD   TOC2                Skriv tillbaka så att nästa
*                                 avbrott sker vid den tidpunkten
        RTI
```



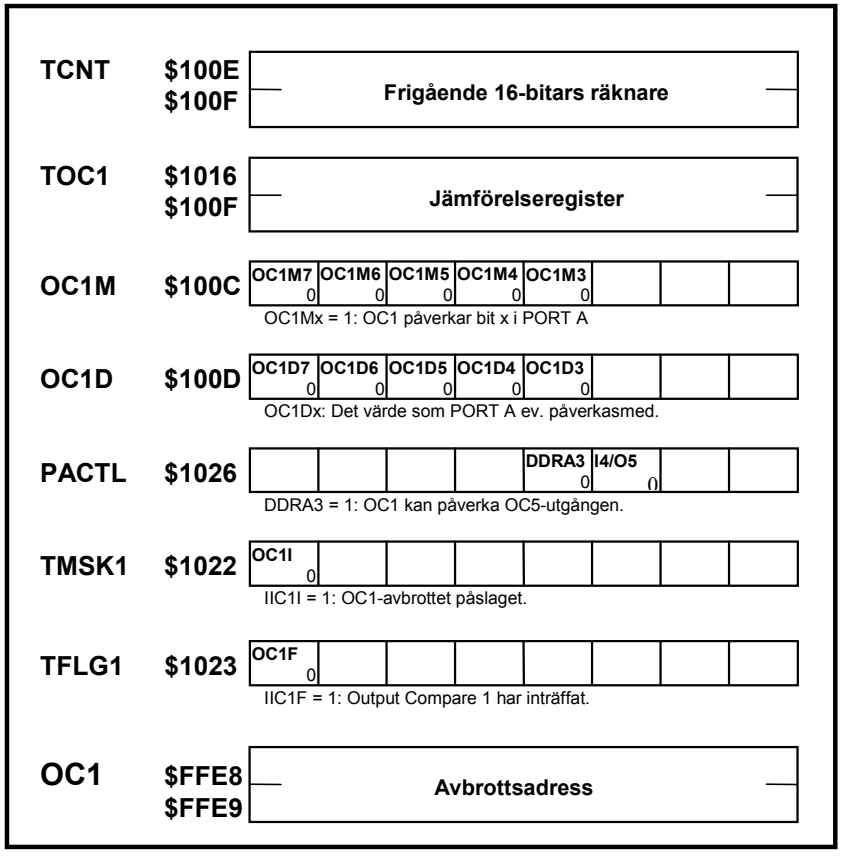
Figur 27. Output Compare.

5.4.1 Output Compare 1



I allt väsentligt fungerar OC1 på samma sätt som OC2 - OC5. Skillnaden är att den har andra möjligheter att styra utgångarna på A-porten. Dels kan man i registret OC1M bestämma vilken eller vilka av de fem OC-utgångarna som ska påverkas. Dels sker påverkan på ett lite annorlunda sätt genom att sätta databitar i registret OC1D.

Denna samverkan med de övriga OC-funktionerna gör att man exempelvis lätt kan skapa 4 st. oberoende pulsbreddsmodulatorer.

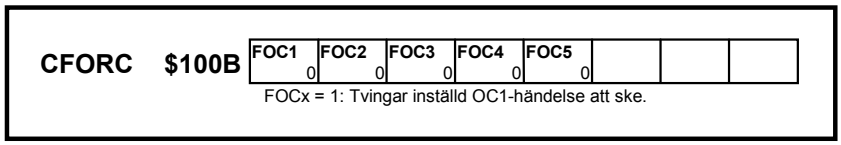


Figur 28. Output Compare 1.

I nedanstående programexempel samverkar OC1 med OC2 för att generera en pulsbreddsmodulerad signal med periodtiden 2 ms. Pulsbredden styrs av variabeln Width. OC1 ska initieras till att ge avbrott och ettställa PA6 (som är den utgång som ska styras både av OC1 och OC2) och OC2 ska nollställa PA6. Detta kan göras genom att programmera TMSK1, OC1M, OC1D och TCTL1 på följande sätt:

```
*  Initiering
InitPWM LDAA  TMSK1
        ORAA  #%10000000
        STAA  TMSK1         OC1 ger avbrott

        LDAA  OC1M
        ORAA  #%01000000    och
        STAA  OC1M          ska påverka PA6
        LDAA  OC1D
        ORAA  #%01000000
        STAA  OC1D          med en etta
        LDAA  TCTL1
        ORAA  #%10000000
        ANDA  #%10111111
        STAA  TCTL1         OC2 nollställer
        RTS

*  Avbrottsrutin OC1.
OC1     LDAA  #%10000000
        STAA  TFLG1         Kvittera
        LDD   TOC1
        ADDD  Width
        STD   TOC2          Uppdatera pulsbredden
        LDD   TOC1
        ADDD  #4000
        STD   TOC1          Uppdatera periodtiden
        RTI
```

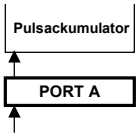
5.4.2 Forced Output Compare



Figur 29. Forced Output Compare.

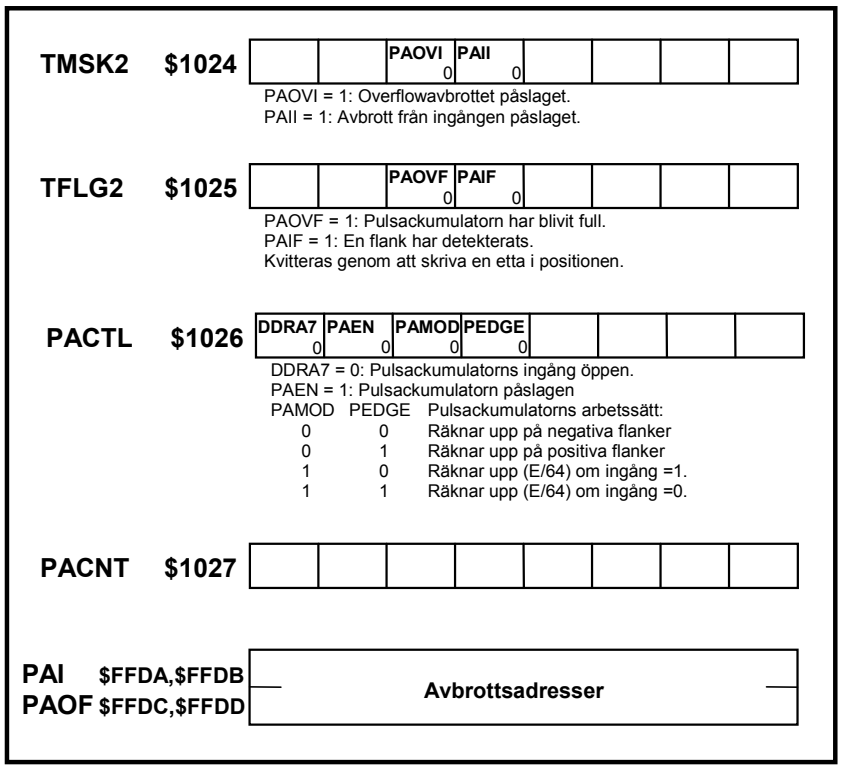
Denna mekanism tvingar den inprogrammerade händelsen för Output Compare att ske omedelbart då skrivningen sker utan att invänta den tid vid vilken utsignalen egentligen skulle ha påverkats. Flaggan för motsvarande OC-avbrott sätts inte, utan endast utsignalen påverkas.

5.5 Pulsackumulatorn



Pulsackumulatorn är en 8-bitars räknare som man både kan skriva i och läsa av. Den kan användas på två olika sätt:

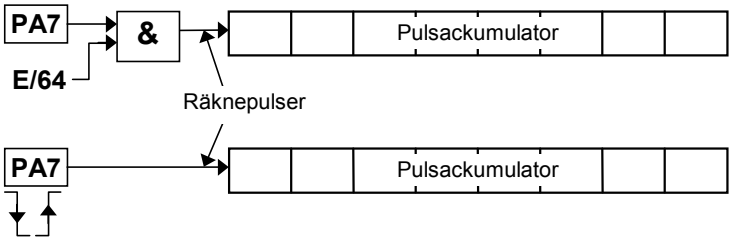
- Räkna antalet inkommande pulser. Räknaren räknas upp för varje inkommande puls. Uppräkningen kan styras så den sker på positiv eller negativ flank (eller båda).
- Bestämma längden på inkommande pulser. Räknaren fås att räkna upp så länge ingången är aktiv.



Figur 30. Pulsackumulatorn.

Räkningen styrs av signalerna på bit 7 i port A.

Pulsackumulatorn används till att mäta mycket höga frekvenser och längden på mycket korta pulser.



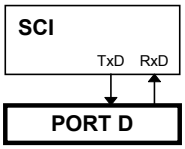
Figur 31. Förenklad bild av pulsackumulatorns två arbetssätt.

Nedan visas ett exempel där pulsackumulatorn används till att räkna negativa flanker på port A:s bit 7.

I minnescellen PAhigh, som är en 8-bitars variabel, är pulsackumulatorn "skarvad"; dvs. om pulsackumulatorn blir full (overflow), så fås en uppräkning i denna minnescell. Resultatet blir alltså en 16-bitars pulsackumulator bestående av PAhigh och PACNT.

```
;  Initiera pulsackumulatorns arbetssätt.
InitPA  LDAA  TMSK1
        ORAA  #%00100000
        STAA  TMSK1              Aktivera overflow-avbrottet.
        LDAA  PACTL
        ORAA  #%01000000         Sätt på pulsackumulatorn.
        ANDA  #%11001111         Känn av negativa flanker.
        STAA  PACTL
        RTS

;  Avbrottsrutin för hantering av overflow.
PAOVF   LDAA  #%00100000
        STAA  TFLG2              Kvitterar flaggan.
        INC   PAhigh             Räkna upp höga byten.
        RTI
```

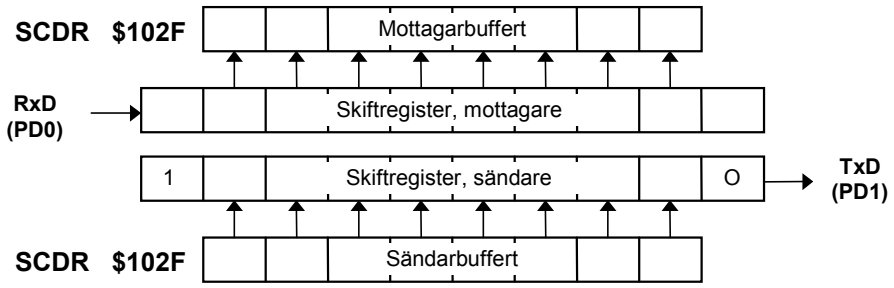
5.6 Asynkron seriekommunikation, SCI



68HC11 innehåller en enkel och flexibel asynkron seriekanal, Serial Communication Interface **SCI**.

Asynkron kommunikation används då data ska överföras långa sträckor med en rimlig hastighet.

En enchipsdator som HC11, använder ofta denna seriekommunikation då den är en del av ett större system där en överordnad dator sköter kommunikationen.



Figur 32. Förenklad modell av SCI.

Vid överföringen används startbit, vanligen 8 databitar och en stoppbit så att mottagaren bara ”ungefär” behöver känna till i vilken fart bitarna överförs. Startbiten talar om att ett nytt 8-bitars meddelande kommer, så att mottagaren kan läsa av databitarna med lämpligt intervall. Stoppbiten är avslutning på bitströmmen och fungerar också som ett minsta mellanrum mellan meddelandena. Ett 8-bitars meddelande kallas ofta för ”tecken”, eftersom asynkron överföring länge varit förknippat med teckensändning till skrivare mm.

I 68HC11 kan överföringen ske som snabbast med ungefär 10.000 tecken/sek, men vanligare är nog att man använder en tiondel av den hastigheten. Överföringshastigheten blir då 9.600 baud, vilket innebär att alla bitar (inkl. start- och stoppbit) skickas med en fart av 9.600/sek.

Den asynkrona seriekanalen i 68HC11 är utrustad med en del mekanismer för att detektera fel i det överförda tecknet. Det finns också inbyggda mekanismer som låter processorn avlyssna en dataledning och reagera för att en speciell bit är ettställd i det mottagna tecknet. Se vidare i databoken.

En enkel form av asynkron kommunikation kan exempelvis bestå av:

Manuell sändning av tecken:

Innan ett tecken sänds, kontrolleras biten TDRE, Transmission Data Register Empty. Denna bit visar om själva sändningskanalen är ledig.

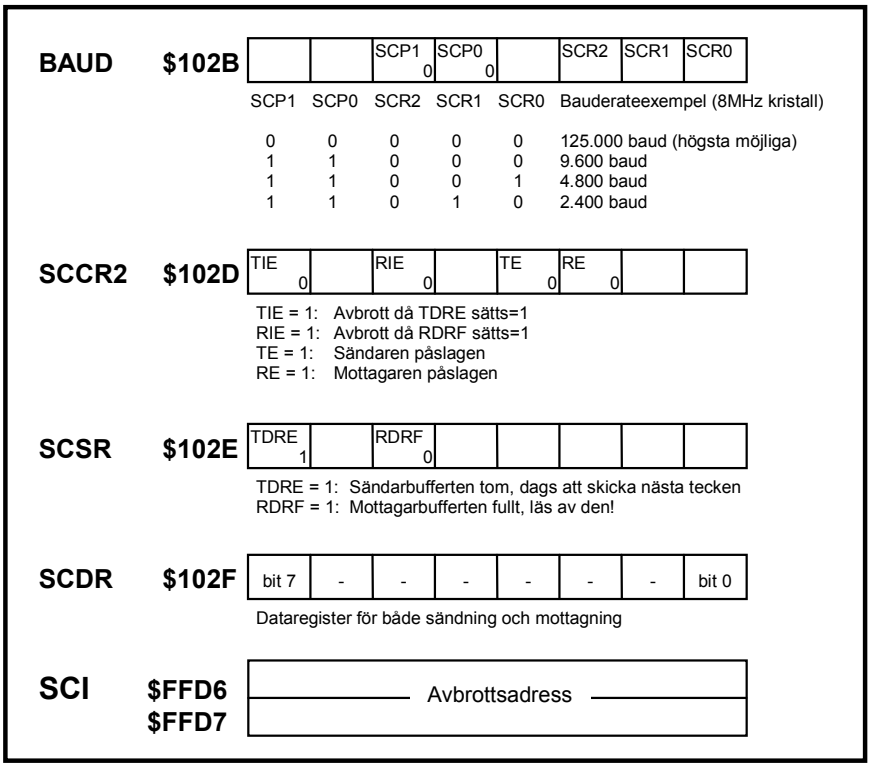
Avbrottsstyrd mottagning av tecken:

Mottagna tecken läses av i en avbrottsrutin och läggs i en ringbuffert.

Ibland måste man komplettera med handskakningssignaler så att den sändande parten alltid vet att mottagaren är beredd att ta emot. Man får från fall till fall bygga sitt kommunikationsprotokoll så att det blir säkert.

Vid sk mjukvaruhandskakning sänder mottagaren ett speciellt tecken, XOFF (transmit off, $13), om den inte kan ta emot tecken just då. När den är redo igen sänds XON (transmit on, $11).

Nedan visas de viktigaste registerbitarna för att enkelt kunna styra kommunikationen.

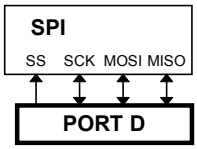


Figur 33. Asynkron seriekommunikation.

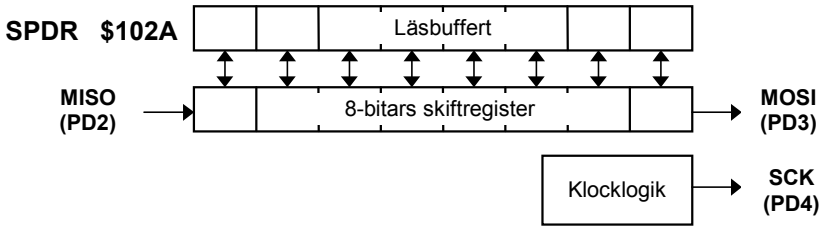
Exempel på programrutiner för den asynkrona seriekanalen:

```
* Transmit
* Sänder tecken i ackumulator A.
* Om  tecknet XOFF finns i SCDR, väntar rutinen.
Transmit   PSHA
Txwait1    LDAA  SCDR    Läs senast inkomna tecken
           CMPA  #XOFF   Var det XOFF?
           BEQ   Txwait  Vänta i så fall
Txwait2    LDAA  SCSR
           ANDA  #%10000000    Redo att sända?
           BEQ   Txwait2
           PULA          Ja
           STAA  SCDR
           RTS


* Avbrottsrutin SCI_int
* Mottagna tecken läggs i buffert
SCI_int    LDD   SCSR    Hämta tecken och kvittera
           CMPB  #XON    Strunta i XON
           BEQ   sciend  och
           CMPB  #XOFF   XOFF
           BEQ   sciend
           JSR   Putbuff Lägg tecken i buffert
           JSR   Diff    Antal tecken i buffert?
           CMPA  #15
           BLO   sciend  under 15: OK
           BEQ   xon     =15: Starta mottagning
           LDAA  #XOFF   annars: stoppa mottagning
           BRA   sciout
xon        LDAA  #XON
sciout     JSR   Transmit
sciend     RTI
```

5.7 Synkron seriekommunikation, SPI

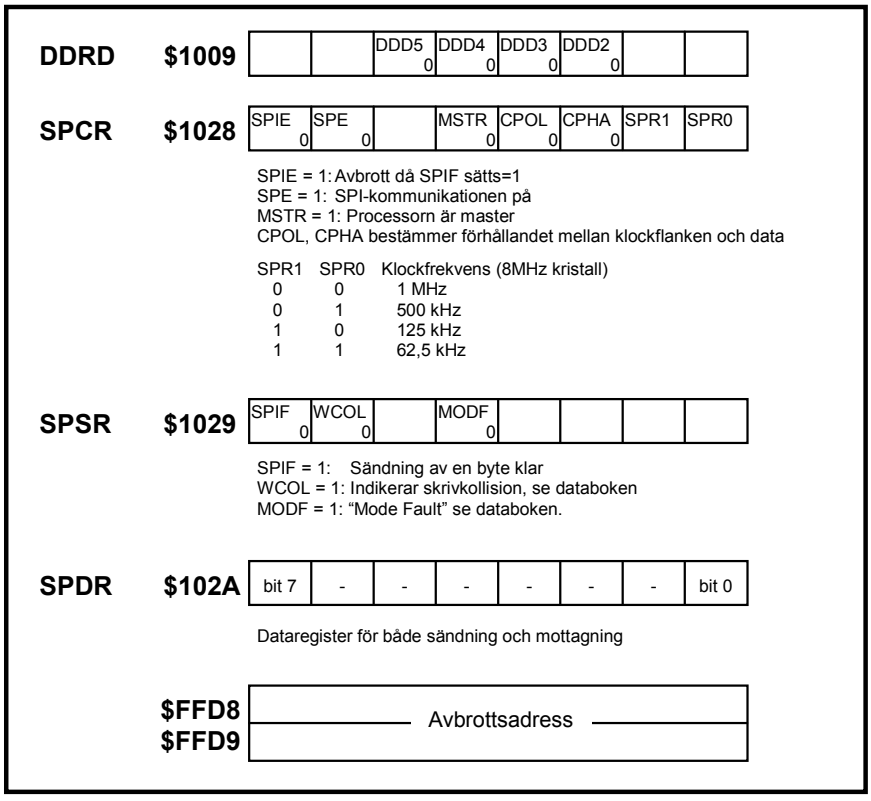


Förutom den asynkrona seriekanalen innehåller HC11 en synkron seriekanal, Serial Peripheral Interface, SPI. Den använder mycket högre överföringshastighet men kräver en synkroniserande klocksignal.



Figur 34. Förenklad modell av SPI.

SPI används främst för kommunikation med periferienheter som finns i närheten av processorn. Kommunikation mellan flera processorer går också att sköta med SPI. Många andra tillverkare av enchipsdatorer har motsvarigheter till SPI (Micro Wire, I²C), men SPI är det snabbaste och det enda som är kompatibelt med ett helt vanligt skiftregister.



Figur 35. SPI.

SPI används normalt med separata dataledningar för sändning och mottagning men kan också användas för dubbelriktad entrådskommunikation.

När HC11 är master, genereras klocksignalen på SCK, men som slave är denna signal en klockingång.

Tack vare den synkroniserande klocksignalen kan överföringen vara så snabb som 1 Mbit/sek.

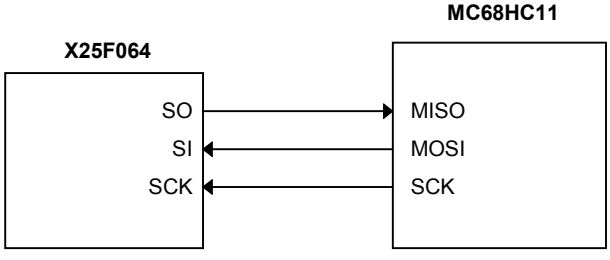
Det finns en mängd kretsar som direkt kan kommunicera med SPI: EEPROM, D/A-omvandlare, A/D-omvandlare, LCD mm.

En överföring av en byte sker genom att skriva till dataregistret, SPDR. Då genereras klocksignaler och data skiftas ut på MOSI. När man vill läsa från exempelvis ett inkopplat skiftregister, skriver man också till SPDR, men bara för att generera klocksignaler. Det inskiftade 8-bitarsordet finns i SPDR när en speciell flagga, SPIF har satts.

Som exempel visas en rutin från EEPROM-tillverkaren XICOR's hemsida. XICOR har en mängd applikationstips för de som använder deras kretsar.

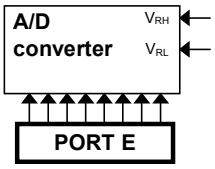
Rutinen gör en läsning från det seriella minnet

```
*************************************************************
* Namn:         ReadByte
* Beskrivning:  Läser innehållet i en speciell minnescell
*               i ett seriellt minne
* Datum:        29/9 1997
* Invärden:     D = Adress i minnet, X pekar på IO-arean
* Utvärden:     A = Värdet på den adressen
* Ändrar:       Inget
*************************************************************
ReadByte psha
         ldaa  #READ_CMD
         staa  spdr,X          Sänd läskommando
         brclr spsr,X,#$80,*   Vänta tills sändningen klar
         pula                  Hämta höga halvan av adressen
         staa  spdr,X          Sänd den till minnet
         brclr spsr,X,#$80,*   Vänta tills sändningen klar
         tba                   Hämta låga halvan av adressen
         staa  spdr,X          Sänd den till minnet
         brclr spsr,X,#$80,*   Vänta tills sändningen klar
         staa  spdr,X          Skifta in data
         brclr spsr,X,#$80,*   Vänta tills överföringen klar
         ldaa  spdr,X          Läs av data från minnet
         rts
```



Figur 36. Anslutning av seriellt minne.

5.8 A/D-omvandlare



A/D-omvandlarblocket består av en A/D-omvandlare som arbetar med successiv approximation, en väljare som kan välja en av åtta ingångar och kontrollogik för att styra omvandlingen på några olika sätt. Normalt används systemklockan för att sköta konverteringen men man kan också använda en inbyggd RC-oscillator.

En fullständig konvertering (av fyra kanaler) tar 128 E-cykler, eller 64 µs, om en 8 MHz-kristall används.

A/D-omvandlaren har ingen egen avbrottsmekanism.

ADCTL $1030

CCF		SCAN	MULT	CD	CC	CB	CA
7	0						

CCF = 1: A/D-omvandling klar. Kvitteras genom att skriva till ADCTL
SCAN = 0: A/D-omvandling aktiveras en gång
SCAN = 1: AID-omvandling återstartar automatiskt
MULT = 0: A/D-omvandling av en enda ingång
MULT = 1: A/D-omvandling av fyra ingångar

CD	CC	CB	CA	Ingång	Resultat i
0	0	0	0	E0	ADR1
0	0	0	1	E1	ADR2
0	0	1	0	E2	ADR3
0	0	1	1	E3	ADR4
0	1	0	0	E4	ADR1
0	1	0	1	E5	ADR2
0	1	1	0	E6	ADR3
0	1	1	1	E7	ADR4

CD = 1 reserverat för testning

ADR1 $1031
ADR2 $1032
ADR3 $1033
ADR4 $1034

bit 7	-	-	-	-	-	-	bit 0

Resultatregister för A/D-omvandlaren

OPTION $1039

ADPU	CSEL						
7	0						

ADPU = 1: A/D-omvandlaren spänningssatt
CSEL = 0: Spänningsgeneratorn drivs av kristalloscillatorn
CSEL = 1: Spänningsgeneratorn drivs av intern RC-oscillator

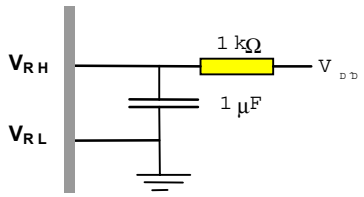
Figur 37. A/D-omvandlaren.

A/D-omvandlaren startas genom att ettställa biten ADPU (A/D Power Up). Därefter får man vänta c:a 100 µs på att omvandlarens matningsspänning stabiliserats (7-8 volt). Sedan startar man omvandlaren på ett av fyra olika sätt genom att skriva till ADCTL.

Här ges ett exempel på en typisk funktion som kan användas för att starta omvandlingen för kontinuerlig omvandling av PE4-PE7:

```
void InitAD(void)
{
   char a;
   OPTION |= 0x80;
   a = 11;
   while (a) a--; // Ett varv tar 9 mikrosek
   ADCTL = 0x34;

}
```

Till A/D-omvandlaren hör två referensspänningar som används för att få ett noggrant resultat. De bör gärna anslutas till spänningskällor isolerade från processorns egen matning men nedanstående enkla koppling kan också användas. Mindre omvandlingsområden kan fås genom att justera ner skillnaden mellan VRH och VRL. Dock bör man inte minska den under 2.5 volt.

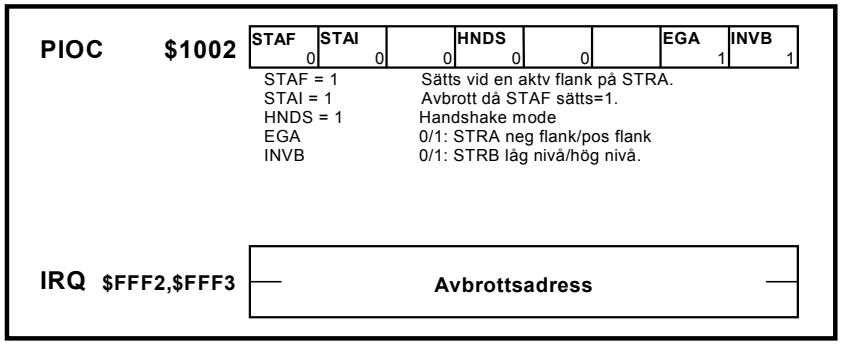


Figur 38. Förenklad inkoppling av referensspänning.

5.9 Parallell kommunikation

Portarna B och C kan användas för effektiv parallell kommunikation:

- Simple Strobed I/O
- Full Handshake I/O



Figur 39. Register för inställningar vid parallell kommunikation.

5.9.1 Simple Strobed I/O

PORTC används som ingång med STRA som en klockingång. Med biten EGA i registret PIOC bestäms vilken flank på STRA som ska vara aktiv.

PORTB är utgång med STRB som strobsignal. INVB bestämmer strobsignalens polaritet.

5.9.2 Full Handshake I/O

PORTC används tillsammans med STRA och STRB som en dubbelriktad databuss. Handskakningsmekanismerna i HC11 är rätt så avancerade. För en fullständig beskrivning hänvisas till databoken.

6 Reset och inbyggda skyddsmekanismer.

Reset används för att starta processorn på ett ordnat sätt. Vid reset blir alla periferienheter och avbrottskällor avstängda. Det ankommer sen på programmeraren att ställa in alla enheter på ett önskat sätt.

Prioritet	Reset / Avbrott	Vektoradress
1	Power on / External	FFFE
2	Clock Monitor	FFFC
3	Watchdog (COP)	FFFA
4	Nonmaskable Interrupt Request (XIRQ)	FFF4
5	Illegal Opcode	FFF8

Figur 40. Reset- och avbrottsvektorer

Den mest uppenbara formen av reset är den som behövs när kretsen spänningsätts. Med yttre elektronik kan man också tvinga processorn till detta läge.

Två andra reset-metoder finns också inbyggda i 68HC11:

Clock Monitor: Om klockfrekvensen understiger c:a 10 kHz, kommer en reset att ske och programexekveringen återupptas när oscillatorn kommit igång igen.

Watchdog: Om programmet på något sätt lämnar de normala banorna, kan man få en reset att utlösas.

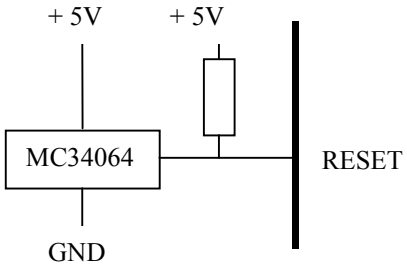
6.1 Reset vid spänningspåslag.

Denna reset skall initiera alla enheter vid kallstart.

Resetpinnen hålls låg (0 volt) under 4096 klockcykler så att även anslutna periferienheter blir initierade.

6.2 Yttre resetsignal.

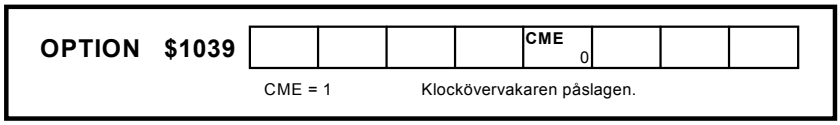
Normal har man alltid en yttre resetfunktion. Man använder ofta speciella kretsar som övervakar matningsspänningen och ger reset då den understiger en viss nivå.



Figur 41. Inkoppling av yttre resetsignal.

6.3 Klocköervakare

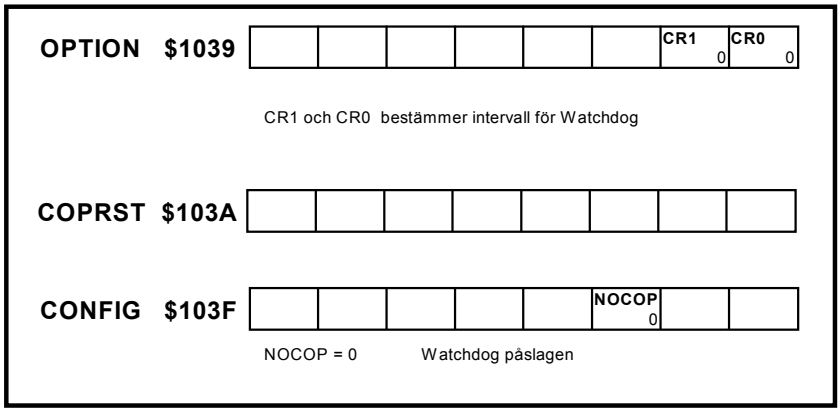
En speciell resetfunktion kan får med hjälp an en intern RC-oscillator. Om E-klockfrekvensen understiger 10 kHz görs en reset och programmet återstartar på den adress som finns på vektoradressen FFFC.



Figur 42. Kontroll av klockmonitor.

6.4 Vakthund (COP)

Ett speciellt skydd mot programfel. Det består av en timer som hindras från att gå ner till 0 med en speciell instruktionssekvens. Sekvensen består i skrivningar till ett speciellt register, COPRST: $55 följt av $AA.



Figur 43. Kontroll av watchdog.

Om man använder en kristall på 8 MHz och alltså har en intern klockfrekvens på 2 MHz, får man följande tidsinställningar på watchdogtimern:

CR1	CR0	Timeout
0	0	16 ms
0	1	66 ms
1	0	260 ms
1	1	1,05 s

Mer om watchdog i kap 9.12 Metoder för att säkerställa programkörningen.

6.5 Illegal Opcode

All kod i programminnet motsvarar inte riktiga instruktioner. Därför finns denna mekanism för att fånga upp de fall då CPU:n läser en felaktig kod. Ett vanligt sätt att använda sig av Illegal Opcode är att låta avbrottsrutinen för en Illegal Opcode vara en oändlig loop, så att Watchdog-funktionen kan användas.

För övrigt så används Illegal Opcode för att implementera brytpunkter.

Programexempel som ger reset när en illegal opcode påträffas:

```
Ill_op      bra   Ill_op
Watchd      equ   RESET

            org   $FFF8
            fdb   Ill_op
            fdb   Watchd
```

6.6 XIRQ, Non Maskable Interrupt

Detta avbrott kan aldrig stängas av om det väl blivit påslaget. Man kan använda det som en nödströmbrytare och det används av debugprogrammen för att stoppa ett program som körts igång i realtid.

7 68HC11:s arbetssätt och inbyggda minnen

Detta avsnitt förklarar de inställningar man kan göra, både hårdvarumässigt och i program, för att få HC11:an att arbeta på en mängd olika sätt.

De inbyggda minnena beskrivs också.

Inställningarna för arbetssätt kan beskrivas i tre nivåer:

- Hårdvaruinställning.
- EEPROM-baserad konfigurationsinställning; CONFIG-registret.
- Tidsskyddade registerbitar.

7.1 Hårdvaruinställning

Den mest grundläggande inställningsnivån är den som görs med de två ingångarna **MODA** och **MODB**.

Ingångar		Beskrivning	Kontrollbitat i registret HPRIO (Laddas in vid reset)			
MODB	**MODA**		**RBOOT**	**SMOD**	**MDA**	**IRV**
1	0	**Single Chip**	0	0	0	0
1	1	**Expanded**	0	0	1	0
0	0	**Special Bootstrap**	1	1	0	1
0	1	**Special Test**	0	1	1	1

Figur 44. Hårdvaruinställningar.

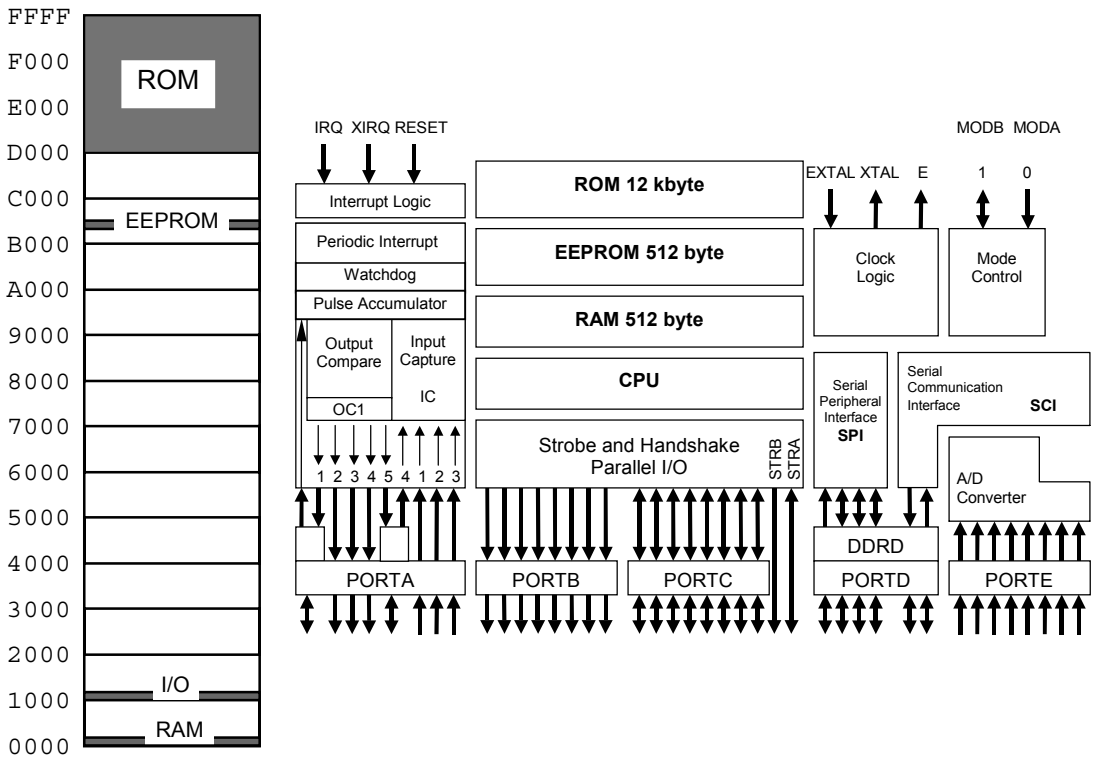
De logiska nivåerna på MODA och MODB laddas in i registret HPRIO vid reset-signalens positiva flank. Dessa nivåer avspeglas sen i bitarna SMOD och MDA som bestämmer hur HC11:ns grundläggande arbetssätt ska vara.

Beroende på uppstartsmod initieras även bitarna RBOOT och IRV.

7.1.1 Single Chip Mode

I denna mod fungerar 68HC11 som en enchipsdator utan tillgång till extern databuss och adressbuss. Det interna programminnet är inkopplat och alla avbrotts- och reset-vektorer är belägna på adresserna $FFD6 - $ FFFF.

Detta är den normala konfigurationen.



Figur 45. Konfiguration vid Single Chip Mode.

7.1.2 Special Bootstrap Mode

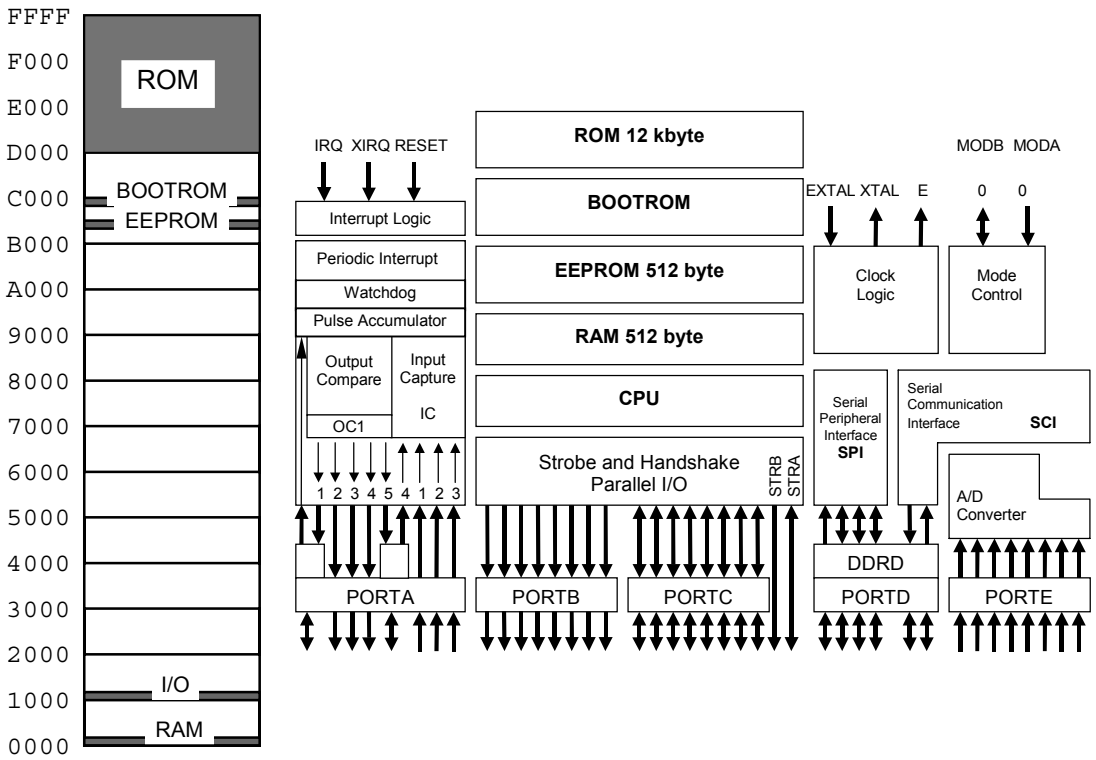
När HC11 startar i Special Bootstrap Mode kopplas ett speciellt inbyggt BOOTROM ($BF40 - $BFFF) in och startar exekvering. Alla avbrotts- och resetvektorer ligger nu på adresserna $BFD6 - $BFFF. Avbrottsvektorerna pekar nu på platser i RAM (pseudovektorer).

ROM och EEPROM kan finnas närvarande, men behövs inte.

Bootstrap-programmet ser olika ut beroende på HC11-typ, men innehåller alltid en möjlighet att via seriekanalen ladda ner programkod i RAM.

Detta ger en utmärkt möjlighet att testa en färdig konstruktion med hjälp av nerladdade programsekvenser.

För mer information, se databoken eller någon av applikationsanvisningarna AN1060 eller EB422/D. Man kan också själv ta en titt på programkoden som ligger på adress $BF40 och se hur den fungerar på din speciella HC11.



Figur 46. Konfiguration vid Special Bootstrap Mode.

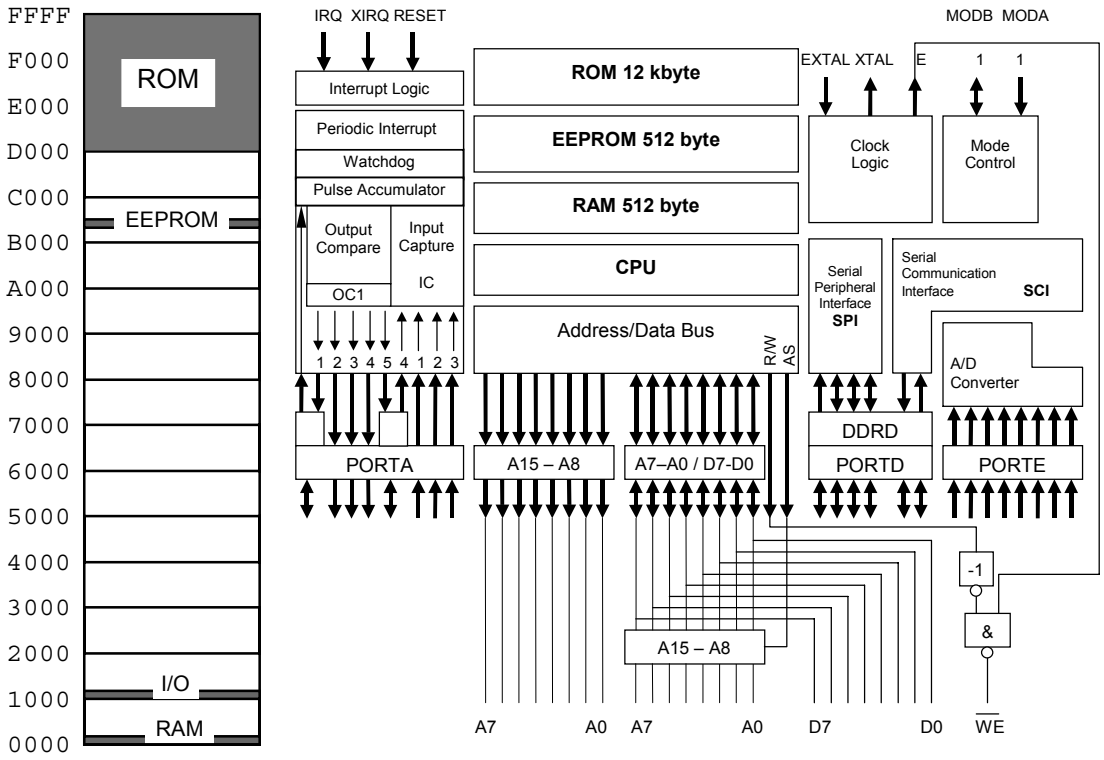
7.1.3 Expanded Mode

I expanderad mod fungerar portarna B och C som adress- och databuss.

STRA får en annan funktion och heter AS (adresstrob).

STRB används som statisk skriv- eller lässignal.

Hela adressrymden på 64 kbyte är tillgänglig men de interna minnena kan fortfarande användas. ROM och EEPROM kan stängas av med bitar i CONFIG-registret (se kap 7.2)



Figur 47. Konfiguration vid Expanded Mode.

7.1.4 Special Test Mode

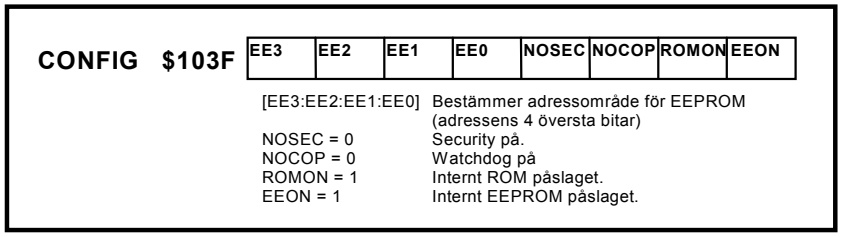
Detta är en variant av expanderad mod som används i första hand för att testa kretsarna vid tillverkningen.

Den är även användbar om man vill programmera om CONFIG-registret och i emuleringssammanhang ifall man vill ha säker tillgång till alla inställningsmöjligheter. Liksom i BOOTSTRAP mode är alla skyddsmekanismer urkopplade.

EM-11 använder denna mod vid uppstart.

7.2 EEPROM-baserad konfigurationsinställning

Nästa nivå av inställningsmöjligheter består av ett register, CONFIG, som implementerats i EEPROM och RAM!



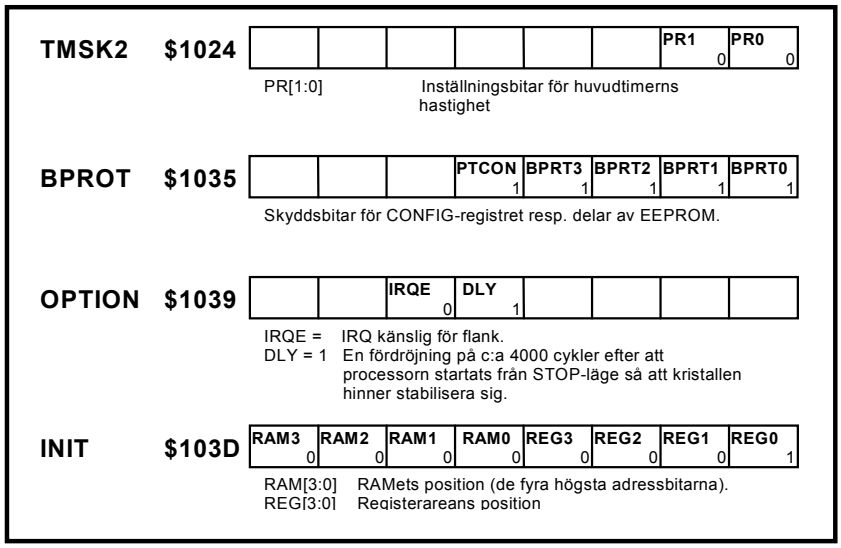
Figur 48. CONFIG-registret.

Vid reset kopieras innehållet i EEPROM-cellen in i RAM-kopian. I HC811E2 är denna RAM-kopia skrivbar om man startar i Special Test eller Special Bootstrap mode.

Det är också i denna typ som de fyra bitarna EE[3:0] finns.

7.3 Tidsskyddade registerbitar

Den tredje nivån av systeminställning utgörs av några register som innehåller ”tidsskyddade” bitar. Dessa är påverkbara inom 64 klockcykler från reset. Om man startar i någon av Special-moderna, går de alltid att ändra.



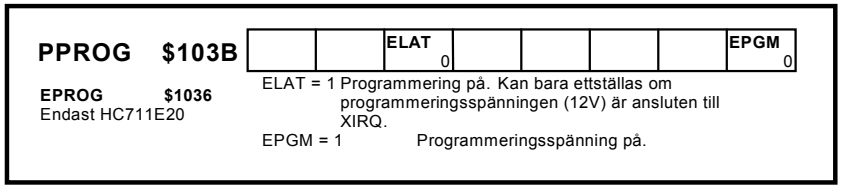
Figur 49. Tidsskyddade registerbitar.

7.4 EPROM

Typerna HC711E9 och HC711E20 har inbyggt EPROM, 12 resp. 20 kbyte stora.

Programmeringsspänning ansluts till ingången XIRQ och programmering kan ske antingen med egna rutiner eller de som finns färdiga i BOOT-ROMet.

HC711E20 har ett separat register för EPROM.



Figur 50. Kontrollregister för EPROM.

Nedanstående lilla program kan tjäna som exempel på hur programmeringen görs.

```
* Vid anrop, A = data som ska programmeras
*            X = en adress i E-minnet
EPROG    PSHB
         LDAB  #$20        Programmera
burn     STAB  PPROG       ELAT ettställs.
         STAA  0,X
         INCB
         STAB  PPROG       EPGM ettställs.
         BSR   delay       2-4 ms
         CLR   PPROG
progexit PULB
         RTS
```

Både HC711E9 och HC711E20 finns i OTP-varianter (One Time Programmable). De innehåller exakt samma chip som de modeller som har fönster, men blir mycket billigare i sin plastkapsel.

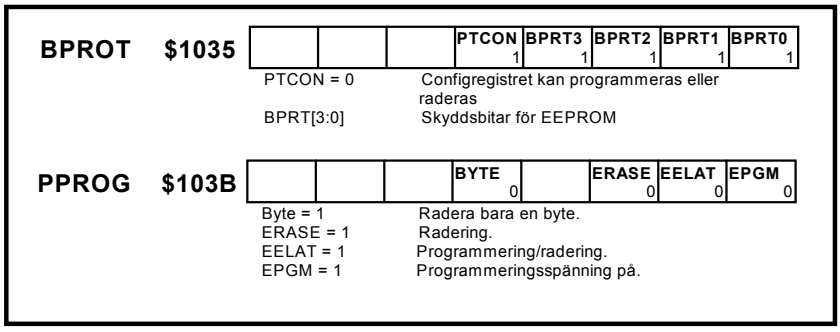
7.5 EEPROM

Alla varianter av 68HC11 har inbyggt EEPROM. Det är mycket lämpat för lagring av värden som inte så ofta ändras.

Varje byte i detta minne kan raderas och programmeras separat. Dessutom kan man radera hela minnet på en gång.

Vid radering blir värdet $FF dvs. alla bitar blir ettställda. Vid programmering kan bitarna bara nollställes.

Detta innebär att om man vill programmera värdet $05 på en plats där det sår $55, behövs ingen radering. Om däremot $55 ska skrivas där det innan stod $05, måste programmeringen föregås av en radering.



Figur 51. Kontrollregister för EEPROM.

För att programmering ska vara möjlig måste motsvarande skyddsbitar i registret BPROT vara nollställda. Registret är tidsskyddat; detta står mer om i databoken, se även 7.3 Tidsskyddade registerbitar.

Programmeringen görs med hjälp av en internt skapad spänning på ungefär 19 volt. Rekommenderad programmeringstid är 10 ms.

Följande programexempel i assemblerspråk illustrerar hur programmering och radering går till:

```
* Vid anrop, A = data som ska programmeras
*            X = en adress i EE-minnet
progbyte PSHA
         PSHB
         CMPA  0,X       Finns detta värde redan i EEPROM?
         BEQ   progexit
         TAB
         ANDB  0,X       Jämför det nya och det gamla.
         CBA             Behövs radering?
         BEQ   noerase
         LDAB  #$16      Radera!
         JSR   burn
noerase  LDAB  #$02      Programmera
         JSR   burn
progexit PULB
         PULA
         RTS
```

```
burn     STAB  PPROG       ELAT ettställs.
         STAA  0,X
         INCB
         STAB  PPROG       EPGM ettställs.
         BSR   delay
         CLR   PPROG
         BSR   delay       Här behövs en kort fördröjning
         RTS

* Ungefär 10 ms:

delay    PSHX
         LDX   #3300
         DEX
         BNE   *-1
         PULX
         RTS
```

Om låg kristallfrekvens används på konstruktionen (under 4 MHz), klarar inte HC11:an att skapa tillräckligt hög programmeringsspänning. Genom att ettställa biten CSEL i registret OPTION, får man en alternativ klockgenerator till den laddningspump som används för att skapa tillräckliga spänningar för EEPROM och A/D-omvandlare.

7.6 RAM

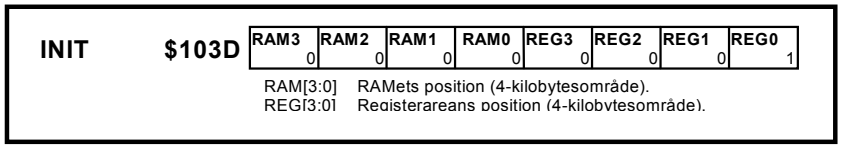
Skrivminne för beräkningar och stack finns naturligtvis i alla typen av HC11. Storleken varierar dock:

HC811E2: 256 byte

HC711E9: 512 byte

HC711E20: 768 byte

Skrivminnet kan inte användas för omfattande datalagring, men det är heller inte meningen. Med det tidsskyddade registret INIT kan man placera skrivminnet på vilket 4-kbytesområde som helst.



Figur 52. Register för placering av skrivminne och register.

8 Programmering i C

Detta kapitel innehåller ett urval nyttiga tips för den som programmerar enchipsdatorn 68HC11 i C.

8.1 Datatyper

När man skriver program för HC11 bör man hålla sig till 8 eller 16-bitars variabler. CPU:n kan ju direkt handskas med dessa typer eftersom den innehåller 8 eller 16-bitars ackumulator.

Några datatyper	Antal byte	Talområde
unsigned char	1	0 — 255
signed char	1	-128 — 127
unsigned short, unsigned int	2	0 — 65535
signed short, signed int	2	-32768 — 32767
unsigned long	4	0 — 4294967295
signed long	4	-2147483648 — 2147483647
float, double, long double	4	±1,18 E-38 — 3,39 E+38

Deklaration av variabel:

```
char Tal, Max, Min;
unsigned int Adress;
```

Data kan deklareras som konstanter med **const**

Ex:

```
const char Tabell [][3] = { { 23, 30, 64 } ,
                            { 12, 31, 16 } ,
                            { 42, 54, 86 } ,
                            { 29, 32, 64 } };

const char String [] = "abc";   kan skrivas i stället för:
const char String [] = {0x41, 'B', 0x43, 0};
```

Variabler som kan ändras av hårdvaran, deklareras **volatile** och behandlas speciellt varsamt av kompilatorn.

Ex:

```
volatile char A;
A = 5;
A = 5;
```

Vid optimering översätts koden till två tilldelningar. Annars hade den ena betraktats som onödig.

Variabler som används i andra filer än där de är deklarerade, måste ha en extern-deklaration:

```
extern volatile char A;
```

Processorns egna register fungerar som variabler som ligger på speciella adresser. De brukar man ha fördefinierade i en särskild fil. Exempel på en sådan finns i bilaga 1.

Variabler som deklareras inuti någon funktion (se nedan) finns bara där. Variabler som ska nås av flera funktioner måste deklareras utanför (före) funktionen.

Tänk på att C tillåter att en lokal variabel har samma namn som en global, men de har inget med varandra att göra. Detta kan ibland skapa förvirring.

I en del fall är det bra att kunna behandla data i en variabel på ett alternativt sätt. Om man t.ex. vill lagra ett flyttal i det interna EE-prommet så måste själva inprogrammeringen ske byte för byte.

Exempel: Vi deklarerar union-variabeln u som kan vara antingen ett flyttal eller ett teckenfält.

```
union {
    float flyttal;
    char byte[4];
} u;
```

Vid beräkningar används det som ett vanligt flyttal:

```
  u.flyttal = 1.38;
```

Om vi vill använda de fyra bytes som flyttalet upptar som enstaka 8-bitersvärden, kan vi skriva såhär:

```
   c = u.byte [0];
   c = u.byte [1];
   c = u.byte [2];
   c = u.byte [3];
```

Ett alternativt (men inte lika tydligt) sätt är med typkonvertering:

```
float flyt;
   c = *((char*)&flyt);
   c = *((char*)&flyt+1);
   c = *((char*)&flyt+2);
   c = *((char*)&flyt+3);
```

8.2 Aritmetiska operationer

+ Addition
- Subtraktion
***** Multiplikation
/ Division
% Modulodivision: `13%5` ger värdet 3

Ex:

Om man vill ha tag i tiotalssiffran och entalssiffran i ett 8-bitarsvärde (< 100) kan man göra såhär:

```
Tiotal = Tal / 10;
Ental = Tal % 10;
```

Observera att man får bättre kod om man kan använda skiftoperationer (se nedan) i stället för multiplikation och division. Förklaringen ligger i hur divisionen definieras i ANSI-C.

Jämför:

```
A = B / 2;      // 36 bytes

A = B /(char)2; // 5 bytes

A = B>>1;       // 5 bytes
```

Ett aritmetiskt uttryck kan vara hur långt som helst. Man måste då vara noga med den ordning som operationerna utförs. Ett råd är att använda gott om parenteser i uttrycket.

8.3 Stränghantering

I C finns det speciella funktioner för att omvandla ett tal till en teckensträng (ex.vis printf och sprintf). Nackdelen är att de slukar oerhört mycket minne.

Det är enkelt att själv skriva sådana funktioner själv.

Omvandla en char, a, till 3 st ASCII-tecken:

```
String [0] = 0x30 + (a / 100);
String [1] = 0x30 + (a % 100)/10;
String [2] = 0x30 + (a % 10);
```

8.4 Bitoperationer

~	Bitvis negation
<<	vänsterskift
>>	högerskift
&	Bitvist OCH
^	Bitvist exklusivt ELLER
|	Bitvist ELLER

Exempel:

Ettställ högsta och lägsta bitarna i variabeln V: `V = V | 0x81;`

Nollställ bit 4 i variabeln V: `V = V & 0xEF;`

Invertera alla bitar i variabeln Var: `Var = ~Var;`

Skifta värdet ett steg till vänster: `Var = Var<<1;`

8.5 Tilldelningssatser

Alla satser avslutas med **;** (semikolon)

Flera satser efter varandra innesluts med **{ }**

Tilldelningar sker med **=**

Den kan kombineras med valfri aritmetisk operation eller bitoperation (utom **~**).

Ex:

```
PORTB |= 0x80;
```

Tilldelningar av typen **|=** och **&=** översätts av kompilatorn till HC11:s bithanteringsinstruktioner, BSET och BCLR.

En speciell typ av tilldelning är **++** (ökning med ett) och **--** (minskning med ett)

Denna operation kan bara göras direkt på variabler.

Om antal = 5: `A = antal++` ger A= 5 `A = ++antal` ger A = 6

Tilldelning från ett fält (sträng eller tabell) görs såhär:

```
PORTB = Tabell [2][0];
```

Tilldelning:	**=**	
Ökning	**++**	`antal = antal+1: antal++`
Minskning	**--**	`antal = antal-1: antal--`

8.6 Jämförelser och logiska operatorer

För att få allehanda villkorsuttryck till programstyrningen, finns följande jämförelseoperatorer och logiska operatorer.

Jämförelseoperatorer	**<**	Mindre än
	>	Större än
	<=	Mindre än eller lika med
	>=	Större än eller lika med
	==	Lika med
	!=	Inte lika med

Logiska operatorer	**&&**	OCH
	||	ELLER
	!	ICKE

Med hjälp av parenteser kan man bygga upp ganska komplicerade uttryck:

```
while (((Input&Night)!=0)&&((NewInput&Car)==0)&&((NewInput&Walk)==0));
```

Precis som aritmetiska uttryck och bitoperationer, bör villkorsuttryck ha gott om parenteser så att man lätt kan förstå innebörden.

Ett annat skäl till att använda parenteser är att undvika fällor som här:

```
if (PORTC & 0x02 != 0) key = 1;
```

Här evalueras först `(0x02 !=0)` som är sant och tilldelas värdet 1.

Därefter görs `(PORTC & 1)` som avläser bit 0 i PORT C. Programmeraren tänkte sig nog bit 1 istället.

Skriv istället:

```
if ((PORTC & 0x02) != 0) key = 1;
```

8.7 Programkontroll: if, while, for, switch/case

För att ett program ska kunna utföra intelligent arbete, måste beslut fattas. Det görs med olika konstruktioner av satser som testar uttryck. Den enklaste är

if-satsen

```
if (villkorsuttryck)
{
    satser;
}
else if (uttryck)
{
    satser;
} else
{
    satser;
}
```

Ibland kan if-satsen vara en klumpig lösning (men man skulle kunna klara sig med bara den). Därför har man konstruerat några andra testande satser som gör programmet mer lätthanterligt.

En är

while-satsen

```
while (uttryck)
{
    satser;
}
```

While betyder ”så länge” och ger ofta ett mycket tydligare skrivsätt.

Om man vill utföra testet i slutet av alla satser, använder man

do-while-satsen

```
do
{
    satser;
}
while (uttryck);
```

En generalisering av while är

for-satsen

```
for (startvärde;villkor;sats(som påverkar villkoret))
{
    satser;
}
```

Följande två konstruktioner är ekvivalenta:

```
a = 5;
while (a != 0) a--;
```

```
for (a = 5; a != 0; a--);
```

Om man t.ex. har en variabel med en mängd olika alternativa värden så är det bekvämt att använda

switch-satsen

Väljer bland olika värden hos variabeln month
Uppgiften för varje case avslutas med break.

```
switch (month)
{
    case 1:
        ....;
        ....;
        break;
    case 2:
        ....;
        ....;
        break;
 ............
    default:
        ....;
        ....;
        break;
}
```

Man kan döpa de olika alternativen till egna namn:

```
enum month { januari = 1, februari, mars, ......};
```

och får då tydligare program:

```
case januari:
     ....;
     ....;
     break;
case februari:
     ....;
     ....;
     break;
```

8.8 Funktioner, funktionsvärden och parametrar

Alla delprogram i C är funktioner, t.o.m. huvudprogramslingan main. Många funktioner behöver inga parametrar och lämnar inga funktionsvärden. De har då beteckningen **void**.

Ex.

```
void main(void)
{
    satser;
}
```

En funktion som ska ha inparametrar definieras såhär

```
char Count (char Knappar)
{
    char Antal;
    // Satser som räknar ut hur många ettor det finns i Knappar.
    return Antal;
}
```

Knappar och Antal är lokala variabler i funktionen Count.

Denna funktion lämnar också ifrån sig ett värde så att den kan finnas med i en tilldelningssats:

```
PORTB = Count(PORTC & 7);
```

8.9 Bitfält

När man vill komma åt enskilda bitar i en variabel, kan man deklarera ett *bitfält*. Det deklareras i en post (**struct**):

```
struct LCD_control {
    char Enable:1;      // bit 0
    char R_W:1;         // bit 1 etc.
    char RS:1;
    char :5;
    };
```

Vi använder struct-definitionen till PORTB:

```
struct LCD_control *LCD = ((struct LCD_control *) 0x1004);
```

Om biten Enable ska ettställas skriver man:

```
LCD->Enable = 1;
```

8.10 Avbrottsfunktioner

En speciell typ av funktion är avbrottsfunktionen.

Den deklareras:

```
interrupt void OC1_interrupt(void)
{
 ....;
 ....;
}
```

Nyckelordet ”interrupt” gör att funktionen görs till en avbrottsrutin, dvs. den avslutas med assemblerinstruktionen **RTI** (return from interrupt).

I filen som innehåller avbrottsfunktionen måste man inkludera en fil med makron som definierar alla avbrottsvektorer:

```
#include "Vectors.h"   // Ser till att avbrottsvektorer initieras
```

Denna fil innehåller alla avbrottsvektorer. (Se bilaga 2)

I huvudprogrammet, **main**, gör man de inställningar som behövs för att avbrottet ska fungera som önskat. När inställningarna är gjorda måste hela avbrottssystemet slås på. Detta görs med instruktionen ”**CLI**” som man kan utföra med intrinsic-funktionen **enable_interrupt()**. Alla sådana finns definierade i en speciell fil:

```
#include "intr6811.h"   // Deklaration av alla intrinsicfunktioner
```

8.11 Monitorfunktioner

Nyckelordet monitor gör att avbrott stängs av när funktionen exekveras. Detta är viktigt när en funktion innehåller tidskritiska delar.

Exempel:

```
// Funktion som skriver en etta till temperaturgivaren DS1820.
// Om den skulle bli avbruten mer än 50 µs blir det problem.

monitor void Wr_One(void)
{
   char I;
   PORTC &= ~0x01;          /* Nollställer PORTC:0
   DDRC |= 0x01;            /* Ställer PORTC:0 till utgång */
   DDRC &= ~0x01;           /* Pullup drar hög efter 4,5 microsek */
   for (I=1; I<=10; I++);   /* Väntar 90 microsek */
}
```

8.12 Pekare och adresser till variabler

I C är det lätt att handskas med adresser till variabler.

Följande kod illustrerar detta. To och From är 16-bitarsvariabler som innehåller första adressen till varsin dataarea. Programmet kopierar från den ena till den andra.

```
int To,From;

void write_byte (char *addr, char data) // Deklaration av
{                                       // pekarvariabel
    *addr = data;                       // Det som addr pekar på
}                                       // får värdet 'data'

char read_byte (char *addr)
{
    return (*addr);                     // Returnera det som
}                                       // addr pekar på

void main(void)
{
    To = 0x0020;
    From = 0x0010;
    while (From != 0x0020)
    {
        write_byte((char*)To, read_byte((char*)From));
        To++;
        From++;
    }
}
```

`*addr` betyder 'det värde som addr pekar på'.

På motsatt sätt kan man skriva:

`&PORTB` (adressen till PORTB) om man vill åt den fysiska adressen till en variabel.

Om man refererar till början av en tabell, kan man antingen skriva:

```
Write_String (&String1[0]);
```

eller

```
Write_String (String1);
```

I båda fallen består parametern av adressen till det första värdet.

8.13 Makron

Ett makro är (slarvigt uttryckt) något som ska föreställa något annat.

Ex:

```
#define PORTB   (*(char *)(0x1004))
```

Vid kompilering byts `PORTB` ut mot `(*(char *)(0x1004))`

så att PORTB blir deklarerad som en pekare till en char som är placerad på adressen 1004H.

```
#define DDRA3   0x08
```

definierar ett namn på en speciell bit i PACTL.

8.14 Filinkludering

Ofta är det behändigt att lägga exempelvis definitioner och annan information i speciella filer. Dessa kan man få med som textmassa vid kompileringen om man infogar **include**.

Ex:

```
#include ”io6811.h”
#include ”traffic.h”
```

8.15 ANSI-C

En stor fördel med att skriva program i C är att man lätt kan byta processor, t.ex. från Motorola 68HC11 till Intel 80C196. Det är då viktigt att C-språket håller sig till vissa givna regler. Den C-standard som många, bl.a. IAR, följer heter ANSI-C. Ibland kan den standarden verka egendomlig. Exempelvis definierar ANSI både **`int`** och **`short int`** som "minst 16 bitar". Använder man en kompilator som inte följer ANSI-C, måste man vara väl medveten om inskränkningarna.

8.16 Assemblerprogrammering

Det finns fall då man behöver skriva sitt program i assemblerspråk.

Ett typiskt exempel är uppstarten av ett program där stackpekaren måste sättas till ett lämpligt värde. Det finns inget annat sätt att göra detta än i assemblerspråk.

Ett annat exempel är då man behöver ha fullständig kontroll över exekveringstider.

Bara i undantagsfall behöver man använda assemblerspråk för att få ner kodstorlek och exekveringstid. Kompilatorns kodoptimerare gör detta ofta på ett bättre och framför allt säkrare sätt, men ett program skrivet i assemblerspråk kan ibland bättre utnyttja CPU-registren.

Detta exempel visar en funktion skriven i C som räknar antalet ettor i en byte:

C-kod	Skapad assemblerkod

char Antal (**char** Value)
{
  char Antal = 0;
 	
  while (Value != 0)
 	
  {
  Antal = Antal + (Value & 1);
 	
  Value = Value >>1;
 	
  }
  return Antal;
}
	Ant: PSHB ***Ingen optimering***
	DES
	TSX
	CLR 0,X
	?0020: LDAA 1,X
	TSTA
	BEQ ?0019
	?0021: LDAA #1
	ANDA 1,X
	ADDA 0,X
	STAA 0,X
	LDAA 1,X
	LSRA
	STAA 1,X
	BRA ?0020
	?0019: LDAB 0,X
	PULX
	RTS ***29 bytes***

char Antal (**char** Value)
{
  char Antal = 0;
 	
  while (Value != 0)
  {
  Antal = Antal + (Value & 1);
 	
  Value = Value >>1;
  }
  return Antal;
}
	Ant: PSHB ***Optimering***
	DES
	TSX
	CLR 0,X
	?0020: LDAA 1,X
	BEQ ?0019
	ANDA #1
	ADDA 0,X
	STAA 0,X
	LSR 1,X
	BRA ?0020
	?0019: LDAB 0,X
	PULX
	RTS ***23 bytes***

Här är ett exempel på hur samma program kan skrivas direkt i assembler. Det krävs goda kunskaper i hur CPU:n och varje instruktion fungerar. Man måste också veta hur parametrar skickas till och från funktionen. I gengäld kan man utnyttja CPU-register mer effektivt.

Det är mycket lätt att göra svårupptäckta fel.

```
                        Skrivet för hand
        RSEG CODE
        PUBLIC Antal

Antal   TBA
        CLRB
        CLC
Next    LSRA
        BCS     Upp
        BEQ     Klar
        BRA     Next
Upp     INCB
        BRA     Next
Klar    RTS
        END                     14 bytes
```

När man använder assemblerfunktioner tillsammans med C-kod måste man veta hur kompilatorn hanterar parametrarna.

Funktioner med 8-bitarsparameter använder ackumulator B!

Funktioner med 16-bitarsparameter använder dubbelackumulatorn.

Funktioner med två 8-bitarsparametrar använder ackumulatorerna A och B.

För ytterligare parametrar används stacken.

Om man inte vet hur parametrar lämnas till och från funktioner, kan man alltid använda globala variabler. Funktionen blir då av typen void utan några funktionsvariabler. Vi antar att värdet som ska undersökas ligger i en global variabel **`Value.`** Resultatet lägger vi i en annan global variabel, **`Count`**.

Såhär kan subrutinen Antal se ut då:

```
Antal   LDAA    Value
        CLRB
        CLC
Next    LSRA
        BCS     Upp
        BEQ     Klar
        BRA     Next
Upp     INCB
        BRA     Next
Klar    STAB    Count
        RTS
```

8.17 Exempel på headerfil som innehåller registerdefinitioner

```
/* FILENAME: IO-reg.h
 *
 * Register and bit macro definitions for
 * all HC11 types in A and E series.
 *
 */

#define REG_BASE 0x1000

#define PORTA   (*(unsigned char *)(REG_BASE + 0x00))
#define PIOC    (*(unsigned char *)(REG_BASE + 0x02))
#define PORTC   (*(unsigned char *)(REG_BASE + 0x03))
#define PORTB   (*(unsigned char *)(REG_BASE + 0x04))
#define PORTCL  (*(volatile unsigned char *)(REG_BASE + 0x05))
#define DDRC    (*(unsigned char *)(REG_BASE + 0x07))
#define PORTD   (*(unsigned char *)(REG_BASE + 0x08))
#define DDRD    (*(unsigned char *)(REG_BASE + 0x09))
#define PORTE   (*(unsigned char *)(REG_BASE + 0x0A))

#define CFORC   (*(unsigned char *)(REG_BASE + 0x0B))
#define OC1M    (*(unsigned char *)(REG_BASE + 0x0C))
#define OC1D    (*(unsigned char *)(REG_BASE + 0x0D))
#define TCNT    (*(unsigned int *)(REG_BASE + 0x0E))
#define TIC1    (*(unsigned int *)(REG_BASE + 0x10))
#define TIC2    (*(unsigned int *)(REG_BASE + 0x12))
#define TIC3    (*(unsigned int *)(REG_BASE + 0x14))
#define TIC4    (*(unsigned int *)(REG_BASE + 0x1E))
#define TOC1    (*(unsigned int *)(REG_BASE + 0x16))
#define TOC2    (*(unsigned int *)(REG_BASE + 0x18))
#define TOC3    (*(unsigned int *)(REG_BASE + 0x1A))
#define TOC4    (*(unsigned int *)(REG_BASE + 0x1C))
#define TOC5    (*(unsigned int *)(REG_BASE + 0x1E))
#define TI4/O4  (*(unsigned int *)(REG_BASE + 0x1E))
#define TCTL1   (*(unsigned char *)(REG_BASE + 0x20))
#define TCTL2   (*(unsigned char *)(REG_BASE + 0x21))
#define TMSK1   (*(unsigned char *)(REG_BASE + 0x22))
#define TFLG1   (*(unsigned char *)(REG_BASE + 0x23))
#define TMSK2   (*(unsigned char *)(REG_BASE + 0x24))
#define TFLG2   (*(unsigned char *)(REG_BASE + 0x25))

#define PACTL   (*(unsigned char *)(REG_BASE + 0x26))
#define PACNT   (*(unsigned char *)(REG_BASE + 0x27))

#define SPCR    (*(unsigned char *)(REG_BASE + 0x28))
#define SPSR    (*(volatile unsigned char *)(REG_BASE + 0x29))
#define SPDR    (*(unsigned char *)(REG_BASE + 0x2A))

#define BAUD    (*(unsigned char *)(REG_BASE + 0x2B))
#define SCCR1   (*(unsigned char *)(REG_BASE + 0x2C))
#define SCCR2   (*(unsigned char *)(REG_BASE + 0x2D))
#define SCSR    (*(volatile unsigned char *)(REG_BASE + 0x2E))
#define SCDR    (*(unsigned char *)(REG_BASE + 0x2F))

#define ADCTL   (*(unsigned char *)(REG_BASE + 0x30))
#define ADR1    (*(unsigned char *)(REG_BASE + 0x31))
#define ADR2    (*(unsigned char *)(REG_BASE + 0x32))
#define ADR3    (*(unsigned char *)(REG_BASE + 0x33))
#define ADR4    (*(unsigned char *)(REG_BASE + 0x34))

#define BPROT   (*(unsigned char *)(REG_BASE + 0x35))

#define EPROG   (*(unsigned char *)(REG_BASE + 0x36))

#define OPTION  (*(unsigned char *)(REG_BASE + 0x39))

#define COPRST  (*(unsigned char *)(REG_BASE + 0x3A))

#define PPROG   (*(unsigned char *)(REG_BASE + 0x3B))

#define HPRIO   (*(unsigned char *)(REG_BASE + 0x3C))

#define INIT    (*(unsigned char *)(REG_BASE + 0x3D))

#define TEST1   (*(unsigned char *)(REG_BASE + 0x3E))

#define CONFIG  (*(unsigned char *)(REG_BASE + 0x3F))
```

8.18 Avbrottsvektorer

```
/*                      - Vectors.h -
   This file defines the interrupt vector addresses of the 68HC11
   and appropriate function names that can be used with the interrupts.
   It is assumed that the segment INTVEC is located at address 0xFFD6.
   Watchdog and Clock Monitor are defined i cstartup.s07
*/

#pragma language=extended

#define INTVEC_START     0              /* Default for 68HC11 (must be matched
                                           to the value used at link-time)  */

                /*=======================*/
                /* Interrupt Definitions */
                /*=======================*/

                /* SCI Serial Communication Interface */
interrupt [INTVEC_START +  0] void SCI_interrupt(void);

                /* SPI Serial Transfer Complete */
interrupt [INTVEC_START +  2] void SPI_interrupt(void);

                /* Pulse Accumulator Input Edge */
interrupt [INTVEC_START +  4] void PAIE_interrupt(void);

                /* Pulse Accumulator Overflow */
interrupt [INTVEC_START +  6] void PAO_interrupt(void);

                /* Timer Overflow */
interrupt [INTVEC_START +  8] void TO_interrupt(void);

                /* Timer Output Compare 5 */
interrupt [INTVEC_START + 10] void OC5_interrupt(void);

                /* Timer Output Compare 4 */
interrupt [INTVEC_START + 12] void OC4_interrupt(void);

                /* Timer Output Compare 3 */
interrupt [INTVEC_START + 14] void OC3_interrupt(void);

                /* Timer Output Compare 2 */
interrupt [INTVEC_START + 16] void OC2_interrupt(void);

                /* Timer Output Compare 1 */
interrupt [INTVEC_START + 18] void OC1_interrupt(void);

                /* Timer Input Compare 4 */
interrupt [INTVEC_START + 10] void IC4_interrupt(void);

                /* Timer Input Compare 3 */
interrupt [INTVEC_START + 20] void IC3_interrupt(void);

                /* Timer Input Compare 2 */
interrupt [INTVEC_START + 22] void IC2_interrupt(void);

                /* Timer Input Compare 1 */
interrupt [INTVEC_START + 24] void IC1_interrupt(void);

                /* Real Time Interrupt */
interrupt [INTVEC_START + 26] void RTI_interrupt(void);

                /* Interrupt ReQuest */
interrupt [INTVEC_START + 28] void IRQ_interrupt(void);

                /* eXtended Interrupt ReQuest */
interrupt [INTVEC_START + 30] void XIRQ_interrupt(void);

                /* SoftWare Interrupt */
interrupt [INTVEC_START + 32] void Software_interrupt(void);

                /* Illegal Opcode Trap */
interrupt [INTVEC_START + 34] void Illegal_Opcode(void);
```

9 Goda råd till konstruktören

Här hittar du en del nyttiga tips för både hårdvarukonstruktionen och programkonstruktionen.

9.1 Kravspecifikationen.

En kravspecifikation ska vara en enkel men ganska noggrann beskrivning på vad din konstruktion ska uträtta. Tänk på att val av kretsar ofta inte hör till kraven.

Exempel på krav:

- Mätnoggrannhet.
- Lagringskapacitet.
- Styrning av utrustningen (knappar, seriell kommunikation).
- Presentation av data (display, seriell kommunikation).
- Batteridrift.

När konstruktionen ska realiseras kommer däremot en del begränsningar in; det blir kanske för dyrt med den önskade noggrannheten, utvecklingssystem för en viss processor råkar finnas och bör då användas. Notera att en HC11 ofta är onödigt avancerad för många projekt. Hade man fritt fått välja så kanske en PIC eller HC05 varit bäst lämpad.

Valet av HC11 hör ***inte*** till kravspecifikationen.

Kravspecifikationen kan ses som projektets mål. Det är därför viktigt att vara bestämd i sina krav. Man kan gärna sätta kraven högt; det är kanske inte förrän i ett senare skede man ser att vissa krav inte går att realisera.

9.2 Val av komponenter.

Detta är egentligen en mycket omfattande uppgift. Man kan behöva göra en rejäl marknadsundersökning för att hitta den mest passande komponenten. I vissa fall kanske man inte har någon aning om hur ett problem ska lösas. Men vi ska komma ihåg att den prototyp som ska tillverkas hellre kan ses som ett konstruktionsförslag där man testar rimligheten i genomförandet. Senare kan man göra rätt stora förändringar i komponentvalet.

Egentligen stöter man här på en mängd begränsningar. Företaget har kanske inte tillgång till vilka utvecklingssystem eller komponenter som helst. Tillsammans med erfaret folk kan man ändå göra ett bra komponentval.

9.3 Schemaförslag.

En typisk HC11-konstruktion utnyttjar ofta en hel del av de inbyggda periferienheterna och ibland också externa kretsar som behövs för konstruktionen.

Om du använder inbyggda enheter, måste du tänka på att de är anslutna till speciella pinnar som du då inte kan använda till något annat:

- Port A används av timersystemet om det behöver kopplas till omvärlden. Lägg märke till att många timerfunktioner kan användas utan att det är inkopplat till en pinne. Exempelvis används med fördel en Output Compare-funktion som periodiskt avbrott utan att påverka sin utgång. Alla utgångar och ingångar som inte används av timersystemet är lediga till vad som helst. Tänk på vilka pinnar som är utgångar resp. ingångar. Observera också att alla ingångar i port A kan användas som generella avbrottsingångar.
- Port B kan bara användas som utgångar. Här kopplar du in t.ex. lysdioder och kontrollsignaler till externa kretsar. Du kanske också kan utnyttja att varje skrivning till port B ger en puls på pinnen STRB.
- Port C är dubbelriktad och bör användas som databuss till alla kretsar som har någon form av parallell kommunikation. Ofta kan du ha flera kretsar på samma buss eftersom de brukar ha någon form av CHIP SELECT.
- Port D kan anslutas till SCI, den inbyggda UARTen, som använder bit 0 och 1. Använder du inte SCI, kan du ha dessa bitar till ingångar eller utgångar. SPI (synkrona seriekanalen) använder minst 3 av resterande bitar på port D. Bit 5 kan du använda som utgång samtidigt som SPI används. Om inte SPI behövs, kan bitarna 2, 3, 4 och 5 användas till ingångar eller utgångar.
- Port E innehåller 4 eller 8 ingångar som kan kopplas in på den inbyggda A/D-omvandlaren. Används A/D-omvandling kan man fortfarande läsa av portens digitala ingångar. Det är faktiskt bara SCI:n i port D som helt lägger beslag på sina två pinnar.

  Om A/D- omvandlaren används, måste V_{RH} och V_{RL} kopplas in på rätt sätt.

Ingångar som inte används, bör alltid kopplas till någon fast spänningsnivå för att undvika störningar. Vändbara signaler kan programmeras om till utgångar.

Inkoppling av några typiska externa enheter:

LCD Det vanligaste är att man ansluter dataledningarna till port C och de tre styrsignalerna till port B. Lägg märke till att många displayer kan kommunicera med endast 4 dataledningar. Ett annat sätt är att ansluta port B till LCD:ns data. Då förlorar man möjligheten att läsa från LCD. Tänk på att andra enheter kan dela på dataledningarna.

Tryckknappar Har man ett fåtal knappar, ansluter man var och en till en ingång. Har man många, kan man använda en tangentbordsavkodare eller avkoda knapparna i program. Man organiserar då knapparna i rader och kolumner och skickar ut bitmönster till exempelvis en kolumn och kan sen läsa av knapparna genom att ta in radernas signaler. Det är olämpligt att låta knappnedtryckningar orsaka avbrott. Istället bör man med jämna mellanrum (2 - 10 ms) avsöka knapparna. Man slipper då många problem som förorsakas av studsar mm.

Klockkrets Ibland är det bra att ansluta en kalenderkrets till sin konstruktion, men det kan ofta vara lika bra att bygga dessa funktioner i program. Fördelen med en klockkrets är att den fungerar utan processor. Den kan t.ex. gå på batteri och med jämna mellanrum ”väcka” processorn för att utföra arbete.

Gör ett fullständigt schema över din konstruktion. ***Alla*** anslutningar som ska finnas i verkligheten, måste ritas ut.

Glöm inte dessa signaler:

- MODA och MODB (ställ in för single chip mode).
- IRQ och XIRQ bör om de inte används kopplas till 5V via motstånd. Man kan koppla ihop dem med RESET.

Det är viktigare att signalernas funktion framgår än att du har med pinnumreringen. Ett schema över en konstruktion med 68HC11 blir något av ett blockschema. Schemat ska fylla två funktioner:

- Det ska vara en översiktlig bild av konstruktionen så att man lätt kan förstå hur de olika delarna samverkar. Speciellt vid felsökning är det bra att veta vilken funktion varje signal har.
- Det ska vara ett underlag för prototypkonstruktionen. Det är alltså inte dumt om pinnumreringen framgår också. Numreringen är naturligtvis också bra att ha vid felsökning.

Du kan komma att modifiera ditt bygge under konstruktionsarbetets gång. **Var då noga med att införa alla förändringar på schemat**.

När du har ett färdigt schema kan du börja bygga din prototyp, nedan kallas den också för **målsystem**.

9.4 Prototypbygge.

Placera ut komponenterna (socklarna) på ett prototypkort. Tänk på att du vänder sockeln till HC11:n så att det lätt går att ansluta emuleringskontakten från emulatorn och så att inte knappar och display hamnar under emuleringskabeln.

Löd fast socklarna i ett par hörn så att de sitter stadigt.

Diskreta komponenter kan du gärna löda fast, speciellt kristallen med tillbehör som ska placeras så nära processorpinnarna som möjligt. Lämna kvar högst c:a 5 mm av anslutningsbenet så det inte spretar och förorsakar kortslutning.

Löd fast matningsspänningen till kretsar som drar 10 mA eller mer.

Placera analoga kretsar för sig, så att de inte störs av processorns klocksignaler. Vissa delkonstruktioner (ofta de analoga) kan behöva testas ut på en experimentplatta. När de fungerar bra kan du föra in dem på kopplingsschemat.

Ibland kan det vara praktiskt att montera vissa motstånd o.dyl. på en adapter som kan sättas i sockel. Tänk bara på att varje extra kontaktövergång är en extra felkälla.

Använd avstörningskondensator till varje krets. Lämpligt värde: 10-100 nF.

Alla övriga ledningar virar du med lämpliga färger: svart används till signaler anslutna till GND och rött till +5V. Övriga signaler får andra färger.

Undvik alltför kompakt ledningsdragning. Ett virat prototypkort ska se ut som ett fågelbo på undersidan. Detta minskar risken för överhörning och underlättar felsökning och modifieringar. Ledningarna ska naturligtvis inte heller vara överdrivet långa; utgå från en spänd ledning och lägg till ett par tre centimeter.

9.5 Testning av målsystemet.

Innan du ansluter matningsspänning, bör du kontrollera att alla ingångar på kretsarna är kopplade till någon signal eller matningsspänning. Oanvända ingångar på CMOS-kretsar kan förorsaka självsvängningar och skada kretsen.

Anslut spänningen utan kretsar i socklarna. Kontrollera med logikprob att spänningarna är rätt.

Observera att spänningen alltid ska vara bortkopplad när du sätter i och tar ur kretsar.

Koppla in emuleringskontakten försiktigt. **Var noga med att orientera den rätt**. Du behöver inte ta ur den förrän mycket senare när du ska testa konstruktionen helt fristående.

Om emulatorn går igång med din konstruktion inkopplad, så har du troligen inte gjort några allvarligare kopplingsfel.

Här kan du passa på att testa den klockoscillator du byggt. Flytta byglingen "X-SELECT" ett snäpp från emulatorns kristall. Emulatorn drivs nu av din egen svängningskrets och kan du fortfarande ladda ner kod t.ex. så fungerar den bra.

Fungerar inte emulatorn så kan du ha gjort något fel bland de signaler inte är rena in- eller utgångar.

Försök sedan att testa de externa kretsarna på **ett så enkelt sätt som möjligt**. I registerfönstret kan man enkelt ställa in så det går att följa och ändra värdet i portarna.

Exempel:

Välj: **Options: Settings: New**

och skriv in

Name **PORTB**

Size: **1**

Address: **1004**

Du kan nu andra portens värde och kontrollera din hårdvara stämmer.

Nu kan du testa enkla insignaler och enkla utsignaler.

Försök aldrig att testa ditt bygge med omfattande program. Då blir det omöjligt att avgöra om felet finns i programmet eller hårdvaran.

Mer komplicerade periferienheter (displayer, klockkretsar, telefonkretsar) kräver att du skriver enkla drivrutiner för att få kontakt med kretsarna.

Tips:

- Testrutin för display kan bestå i att du skickar ut de kommandon som behövs för att få markören synlig och blinkande. Får du detta att fungera, så är displayen med all säkerhet rätt inkopplad.
- Testrutin för kalenderkretsen 68HC68T1 kan vara att skicka ett kommando som får CLOCKOUT att blinka med 1 Hz. Försöker du redan från början att ställa klockan och läsa in värden från den så är det ju bra om det fungerar, men annars vet du ju inte var felet är; i skrivningen eller läsningen.
- Testrutin för ett seriellt minne kan bestå av att du börjar med att försöka läsa i ett statusregister (brukar finnas) där det inte kan stå vadsomhelst. Då kan du verifiera att läsningen fungerar. Sen fortsätter du med att försöka skriva och se om du kan läsa tillbaka samma sak som du skrivit.

Några påpekanden:

- Om du använder SPI-kanalen, ska du tänka på att den har en automatisk avstängning som träder i funktion om PD5 (slave select) = 0 och SPI programmeras som master. Initiera därför alltid PD5 som utgång innan SPI startas.
- Om du har minnesfönstret öppet när du testar dina rutiner, kan det ske en oavsiktlig kvittering av vissa flaggor, t.ex. SPIF, TDRE, RDRF. Dessa kvitteras ju genom läsningar, och det är ju det som debuggern utför när den presenterar värden.

9.6 Drivrutiner och komponentprogram.

Drivrutiner kallas de programdelar som ligger som ett skal närmast hårdvaran. De ska vara så utformade att det räcker att ändra i dem om man förändrar hårdvaran (man byter kanske till en display av ett annat fabrikat). En drivrutin är alltså hårt bunden till en viss komponent.

Komponentprogram är program som utför något som typiskt kunde ha gjorts med en speciell komponent. Ett komponentprogram kan utgöras av en drivrutin eller vara en självständig enhet.

Exempel:

- Drivrutiner till en LCD kan vara funktionerna Write_char och Write_line som skriver tecken resp. teckensträng enligt ett visst format. När man skriver själva applikationsprogrammet behöver man bara veta vad dessa funktioner gör; inte hur de gör det.
- Ett typiskt komponentprogram är ett som avsöker tangenter. Det kan göras så det exakt uppför sig som en verklig komponent (74HC922).

```
Actual_keys = PORTC;
New_keys = ~Old_keys & Actual_keys;
Old_keys = Actual_keys;
```

  ettställer New_keys vid nya knappnedtryckningar.

- En monostabil vippa kan också med fördel realiseras i programkod.

Drivrutinerna är de programdelar man först bygger. Det gäller att göra dem så bekväma som möjligt. När drivrutinerna är testade ordentligt, kan man ägna sig åt att implementera det slutliga programmets funktioner. Man slipper då bekymra sig över hur portar och periferienheter används.

Kalla inte på drivrutinerna från både huvudprogram och avbrottsrutin. Det kan ställa till mycket oreda. Exempelvis låter man utmatning till display (Write_char) skötas antingen från huvudprogrammet eller en avbrottsrutin.

9.7 Avbrott.

Normalt använder man en eller flera avbrottsrutiner i sitt program.

1. Avbrott från periferienheter, interna eller externa.

   Exempel: SCI-avbrott som ger avbrott när ett tecken kommit från en PC eller liknande utrustning. Programmet i HC11:an vet inte när tecken kommer, men kan ta om hand varje tecken i ett avbrott, lägga det i en buffert och sen återgå till den normala verksamheten. När sista tecknet kommit, sätter avbrottsrutinen en flagga som huvudprogrammet testar och då tar hand om hela datasträngen.

2. Exakta periodiska avbrott för att hålla en noggrann tid.

   En klocka med sekunder, minuter, timmar etc. går lätt att implementera i program om man låter ett OC-avbrott komma ex.vis. var 10:e ms.

3. Periodiska avbrott för sampling med lagom intervall.

   För att söka av knappar mm är det lämpligt att ha ett avbrott som körs ungefär var 5:e - 10:e ms. Man slipper då alla problem med tangentstuds.

Ett gott råd är att alltid beakta möjligheten att sampla insignaler med jämna mellanrum innan man ger sig på att konstruera någon enhet som ger avbrott.

Avbrottsmekanismerna är avstängda vid uppstart; varje avbrott har sin speciella inkopplingsbit. Avbrottsrutinen måste ha precis det namn som finns i int6811.h (du kan ändra de namnen om du vill). Denna fil placerar avbrottsvektorn (adressen till avbrottsrutinen) på sin särskilda plats om man skriver sin avbrottsrutin såhär:

```
interrupt SCI_interrupt()
{    // koden skrivs här
}
```

En kvittering av avbrottet måste alltid finnas i koden. Den kvitteringen ser olika ut för olika avbrott. För den asynkrona seriekanalen gäller att avbrottsflaggan kvitteras om statusregistret läses följt av en läsning eller skrivning i dataregistret. Timersystemets avbrottsflaggor fungerar lite egendomligt: man nollställer en flagga genom att skriva en etta till den!

Man behöver aldrig stänga av avbrottssystemet när man fått avbrott eller sätta på det igen. Detta sköts automatiskt i HC11:n. Normalt ska man aldrig låta ett avbrott avbryta ett annat. Gör man det, ska man ha goda skäl och vara extra uppmärksam på vad som kan hända. Se alltid till att alla avbrottsrutiner är korta. Om någon mer omfattande verksamhet måste sättas igång på grund av en händelse i ett avbrott, så får det skötas genom att en flagga sätts så att uppgiften kan tas omhand av huvudprogrammet.

För att sätta igång hela avbrottssystemet anropar man

```
enable_interrupt();
```

som bara innehåller assemblyinstruktionen **CLI**.

Den fil som innehåller detta anrop måste inkludera filen intr6811.h.

```
#include "intr6811.h"
```

Man får se upp när man använder avbrottsingången IRQ och vill ha den flanktriggad. Den inställningen sker med en bit, **IRQE**, i en registervariabel **OPTION**. Biten är tidsskyddad, dvs. går bara att påverka inom 64 klockcykler från RESET, men kan sättas för hand på detta sätt:

- Tryck på emulatorns reset-knapp innan du startar C-SPY. HC11an kör nu i *Special Mode* som tillåter att man ändrar de tidsskyddade bitarna.
- Öppna minnesfönstret med  **(Memory Window)**. Dubbelklicka på adress 1039 och ändra bit 5 till en etta (om det står 10 så skriv 30).
- Starta programmet (stegning eller full fart). *Special Mode* lämnas och HC11an kör nu i *Normal Mode*.

Senare när du kör processorn med fast kod, ska denna initiering läggas i cstartup.s07. Du kan lägga den där redan nu, men den "tar" alltså inte förrän du startar HC11an riktigt från RESET.

Kombinationen  (Reset),  (Reset),  (Go) ger en hårdvarureset: resetvektorn hämtas till PC och HC11:an utför en riktig reset! Om du inte har någon C-kod eller kör i realtid, så kan du då bara bryta programmet med emulatorns reset-knapp.

Externa avbrott kopplas med fördel in på lediga ingångar i port A. Dessa kan programmeras att vara känsliga för positiv eller negativ flank eller bådadera. Du slipper då ovan beskrivna problem med **IRQE**.

9.8 Dags att testköra program.

För att köra i verklig fart, ska du bocka för **Real Time** under menyn **Control** eller bocka bort markeringen framför **Calls**. Har man inte någon av dessa inställningar, så går inte programmet i full fart. Vid **Real Time** kan du inte stega programmet men det går naturligtvis fortfarande att sätta brytpunkter.

Om du har C-koden framme, märker du att det inte går att sätta brytpunkter varsomhelst. Sätter du markören på `if` så syns brytpunktsknappen  . Om markören sätts på else, så gråas den. Detta beror på att if-satsen motsvarar instruktionen där testet utförs, medan else inte motsvaras av någon instruktion. Prova med att växla till assembler!

Ett enkelt och smidigt sätt att få reda på variabelvärden är att i källkodsfönstret försiktigt föra markören I över variabelnamnet. Observera att lokala variabler bara är definierade i sin funktion och att de kan ha vilka värden som helst innan du tilldelat dem värden. De kan inte initieras vid uppstart eftersom de finns i det lediga stackutrymme som funktionen skaffar sig.

9.9 Internt EEPROM.

Det interna EEPROMet är mycket användbart när man vill spara inställningar så att de ligger kvar vid strömavbrott, eller om man lätt vill ge varje konstruktion en personlig inställning utan att ha olika program i varje.

De HC11-typer som tillhör E-serien har alla en inbyggd säkerhetsmekanism. Denna består av fem tidsskyddade bitar i registret BPROT, som bara kan nollställas inom 64 klockcykler från reset. Sättet att upphäva skrivskyddet liknar det som beskrivits ovan för att påverka biten **IRQE**.

Du kan också utföra en hårdvarureset med sekvensen

 (Reset),  (Reset),  (Go).

Om du använder EMX-11 eller EM-11 med monitorprogram W21e eller senare, är tidskyddet av EEPROMet bortkopplat vid uppstart. Detta för att du enkelt ska kunna testköra dina funktioner i C-SPY. Tänk på att du ändå har med motsvarande instruktioner i cstartup.

Programmering och radering av enskilda minnesceller görs med hjälp av rutiner som finns exemplifierade i databoken.

När du gör rutiner för skrivning i EEPROM, tänk på detta:

- Före varje programmering bör du kontrollera att data ska ändras; det är ingen mening att programmera in samma sak igen, det nöter bara på minnet.
- Testa om en radering krävs; ska ettor ersätta nollor, kan man inte bara programmera in ett värde utan måste radera cellen först. Endast nollor kan alltså programmeras in.

Om du använder kristallfrekvens på under 4 MHz, bör du ställa om biten **CSEL** i **OPTION**-registret.

Variabler som ska ligga i EEPROM deklareras t.ex.:

```
no_init char eemem;
```

De placeras då i området **B600-B7FF**.

9.10 När du ska programmera in programkoden.

För den slutliga inprogrammeringen behöver du en emulator med programmeringsmöjlighet. Du använder egentligen en likadan emulator men med ett annat monitorprogram, **Bug-11**. I EMX-11 byter du till detta program med en switch. Detta program innehåller en mängd enkla kommandon som kan användas för programtestning och för inprogrammering av koden.

Den kod som ska laddas ner, ska vara av typ Motorola S-kod. Detta är ett standardformat som används av emulatorer och programmeringsutrustningar. Namnet kommer av att varje rad i textfilen inleds med ett S.

Använder du IAR:s EW6811, så är det bäst om man gör vissa inställningar i

Target: Release

så behöver man inte ändra i debugversionen.

Här ställer du in under **Project: Options: Xlink: Output:** ***other, motorola***.

Du kan också ändra namnet under **Output File** så att det slutar på .txt. Då blir det enklare att ladda ner med programmet Hyperterminal.

En annan intressant inställning är under **Processing** där man kan begära att få all oanvänd programarea fylld med något speciellt. Man väljer ofta en ”Illegal Opcode” så att watchdogfunktionen kan användas om programräknaren skulle komma vilse.

Den kod som skapas för nerladdning ligger i

Release\Exe

I terminalprogrammet ser du till att få kontakt med emulatorn. Använd sedan kommandot **LOAD<ENTER>**. Under **Transfer** väljer du **Send Text File** och bläddrar fram koden som ska laddas ner. Meddelandet **Done!** kommer när det är klart. För en kod som tar upp nästan 12 kbyte tar nerladdningen lite mer än en halv minut.

Nu ligger programmet i emulatorns emuleringsminne (precis som vanligt). Koden programmeras in i EPROM (eller EEPROM) med kommandot **PROG**. CONFIG-registret ändras automatiskt vid inprogrammeringen så att programminnet sätts på, men kan ställas om manuellt med kommandot **CONF**.

9.11 Sluttestning.

När koden blivit inlagd i processorns programminne, kan man flytta ut kretsen till den sockel i konstruktionen där emuleringskontakten har suttit.

Men innan man lämnar emulatorns trygga värld kan man provköra programmet från emulatorn:

Avlägsna byglingen "EMUL MODE". I EMX-11 ställer man om en switch. Om du nu trycker på reset-knappen eller startar med spänningspåslag, så ska ditt eget program gå igång.

Nästa steg är att prova helt utan emulator.

Med processorn isatt i sin sockel ska nu programmet fungera lika bra som innan. Om det inte gör det, kontrollera följande:

- **MODA** ska vara inkopplad till GND så att HC11:an startar i "single chip mode".
- **RESET** måste vara rätt inkopplad på målsystemet. Normalt använder man en speciell krets (MC34064 eller DS1233) som övervakar matningsspänningen.
- **XIRQ** är i emulatorn kopplad till 5V via ett motstånd. Se till att motsvarande koppling finns på målsystemet.
- **IRQ** måste också anslutas till 5V med eller utan motstånd.
- Kontrollera matningsspänningen.
- Kontrollera att oscillatorn verkligen svänger (mät på E).

9.12 Metoder för att säkerställa programkörningen.

En yttre störning kan, om den är kraftig, få precis vad som helst att hända i processorn. Sådana störningar kan man gardera sig mot genom att ta hänsyn till några olika sätt att förbättra konstruktionen:

- Kretskortslayout.

  Ett dålig layout är den vanligaste orsaken till störproblem. De kritiska områdena är:

  Dålig spänningsförsörjning. Använd alltid en avstörningskondensator vid varje krets. Använd helst speciella jordplan och spänningsplan.
  Störningar från oscillator. Placera kristallen så nära **EXTAL** och **XTAL** som möjligt. Använd exemplet i databoken. En bra konstruerad oscillator ger också mindre störningar till omgivningen.
  Störningar på ingångar. Inga ingångar får hänga i luften. Anslut dem till 0V eller +5V. Man kan också programmera om vändbara ingångar till utgångar. Speciellt flanktriggade ingångar bör skyddas med en buffertkrets.

- Watchdog och andra inbyggda mekanismer.

  HC11:an har två inbyggda skyddsmekanismer:

  Watchdog. En mekanism som ser till att programmet återstartas med en riktig **RESET** om man inte kvitterar den genom att utföra ett par speciella skrivningar till ett speciellt register. Den kommer att utlösas om man t.ex. hamnar i en oändlig loop där denna kvittering saknas. Det kan vara svårt att bestämma den bästa platsen för en sådan kvittering; helst ska den bara finna på ett ställe i hela programmet men det kan vara svårt att lösa. Helst ska kvitteringen placeras på strategiska ställen i huvudprogramslingan.
  Clock Monitor. Om signalen på kristallingången upphör under en viss tid, sker en riktig **RESET** när oscillatorn kommit igång igen.

- Defensiv programmering.

  Genom att använda lämpliga programmeringsmetoder, kan man uppväga brister i den omgivande hårdvaran.

Här är några exempel:

Pollning av tangenter för att gardera sig mot tangentstuds.
För att filtrera högfrekventa störningar kan man t.ex. gör tre avläsningar direkt efter varandra för att godkänna en nivå.

Kontinuerlig uppdatering av ingångar och utgångar.
Eftersom in- och utgångar vanligen ligger nära chipets kanter, är tillhörande logik ofta utsatt för störningar. Genom att regelbundet uppdatera utsignaler från kopior i RAM och regelbundet avläsa insignaler, kan man få en avsevärd förbättring av säkerheten. Även portarnas riktningsregister kan man passa på att ställa in regelbundet.

Infångning av vilsen programräknare.
Stora delar av den teoretiskt tillgängliga minnesarean innehåller ju inte program. Genom att fylla allt outnyttjat område med en otillåten instruktion t.ex. 0x55, kan man tvinga programräknaren till en viss adress om den hamnat fel. Detta kan man kombinera med watchdog-funktionen ovan om man på denna adress lägger on oändlig loop.

Periferischema

Fyll i vilka in- och utgångar som används av de olika yttre enheterna

		In	Ut					
PORTA	7	X	X					
	6		X					
	5		X					
	4		X					
	3	X	X					
	2	X						
	1	X						
	0	X						
PORTB	7		X					
	6		X					
	5		X					
	4		X					
	3		X					
	2		X					
	1		X					
	0		X					
PORTC	7	X	X					
	6	X	X					
	5	X	X					
	4	X	X					
	3	X	X					
	2	X	X					
	1	X	X					
	0	X	X					
PORTD	5	X	X					
	4	X	X					
	3	X	X					
	2	X	X					
	1	X	X					
	0	X	X					
PORTE	7	X						
	6	X						
	5	X						
	4	X						
	3	X						
	2	X						
	1	X						
	0	X						

10 Lösningar till övningsexempel

Lösningar kapitel 1

1.

a) LDX: Flytta data.
LDA: Flytta data
STA: Flytta data
INX: Ändra data
CMPA: Testa
BHS: Programkontroll
BRA: Programkontroll

b) LDX #0000

c) Indirekt adressering: LDA 0,X; CMPA 0,X
Absolut: STA 1004
PC-relativ: BHS och BRA

d) N=1, Z=0, V=0, C=1. Det är C=1 som gör att programmet kommer till just denna rad.

f) $FF. Programmet letar upp större och större värden

2.

På stacken ligger: ??, f8, 06

?? är det värde som ackumulator A råkar ha.

3.

```
          LDX   #0
igen      LDA   0,X
          STA   6,x
          INX
          CPX   #$0006
          BNE   igen
stopp     BRA   stopp
```

Lösningar kapitel 2

1. DDRC = 11100111.

2. Om man kopplar in kretsarna på en port som bara kan vara utport, blir alla kretsarna aktiverade från början efter reset, innan man hunnit ställa om portsignalerna.
   Om man däremot använder en vändbar port (tex. PORTC) så kan man med pullupmotstånd säkerställa att alla kretsar ar avaktiverade vid uppstart. Därefter ettställer man alla bitar i porten och gör den sedan till utport.

3. Maximala antalet utgångar är 27. Man får då 11 ingångar.
   Maximala antalet ingångar blir också 27 med 11 utgångar

4. Om de var utgångar, skulle de eventuellt bli ihopkopplade med andra utgångar. Däremot är det ofarligt att koppla ihop en ingång med en annan.

5. Om vi förutsätter att vi inte vet de aktuella riktningarna på signalerna, så kan det se ut såhär:

```
LDAA  #$F8
STAA  DDRC
```

Lösningar kapitel 3

1. Byt ut BHS mot BLS (Branch Lower or Same).
   För 2-komplement används i stället BLE (Branch Less or Equal).
2. Ett program som nollställer skrivminnet (RAM) i en HC11:

```
        LDX   #0
nollst  CLR   0,X
        INX
        CPX   #100
        BNE   nollst
```

3. Subrutin som nollställer ett visst antal minnesceller från en viss adress.

```
Zero    TSTB
        BEQ   klar
        CLR   0,X
        INX
        DECB
        BRA   Zero
klar    RTS
```

4. Program som kopierar E000-E0FF till0000-00FF.

```
        LDX   #$E000
        LDY   #0
next    LDAA  0,X
        STAA  0,Y
        INX
        CPX   #$E100
        BNE   next
```

5. Subrutin som kopierar ett visst antal minnesceller från ett område till ett annat.
   Antal i ackumulator B. Frånadress i X. Tilladress i Y.

```
Copy    TSTB
        BEQ   klar
        LDAA  0,X
        STAA  0,Y
        INX
        INY
        DECB
        BRA   Copy
klar    RTS
```

Register