Cred Develop System Version 1.2

User Manual

Innehåll.

Cred Develop System är en debugger för Motorola MC 6809.
Systemet är delat i två delar. Ett program som ska köras i en PC/AT under MS-DOS och ett målprogram som ska implementeras i den målmaskin som programvaran ska utvecklas i.

För att få ett komplet utvecklingssystem krävs:

* PC/XT/AT med 640 kb minne och en serie port.
* Programvara till Värddatorn:
  Norton Commander 2.0 eller högre.
  6809 CrossAssembler.
  CreDBG
* Målmaskin baserad på MC 6809 (minst 4 MHz klocka) bestyckad med:
  minst 3kb ROM.
  minst 2kb RAM (mer krävs i praktiken för programutveckling, rekomenderat RAM minne 32kb.)
  en serieport 6551 (annan serie port kan lätt implementeras)

Med Norton Commander kan man editera och köra assembleringsbatchfiler mm.
Crossasembler kan vara vilken som helst bara den kan generera HEX-filer typ Motorola S1 format.

Instalering

I filen System.src ska vissa parametrar sättas.
I huvudet på den filen hittas en kort beskrivning vilka variabler som ska ändras för att målmaskinen ska fungera.
Man måste definiera var den RAM area som ska användas för målprogram ska ligga. Denna RAM area måste vara kontinuerlig för att måldebuggern ska kunna göra RAM test och bör vara så stor som möjligt.
Finns det fler RAM areor för målprogram anges den största.
Man kan alltid nyttja RAM areor som inte definierats för måldebuggern med den enda skillnad att de inte testas.
Med parametrarna USERAM/USRATO definieras RAMstart / RAMslut.
Man ska också tala om för mål debuggern var det finns RAM till själva måldebuggern. Till det RAM:et anges endast RAMstart = DBRAM.
Efter den adressen och uppåt i minnet måste det finnas minst 1024 byts kontinuerligt RAM.
RAM arean för måldebuggern får absolut inte röras av målkoden.
I denna RAM area spara måldebuggern interna variabler och intern stack,in/ut buffrar mm.
Finns watchdog i målmaskinen ska adressen till den anges på WDOG parametern.
Adresserna till serieporten anges på ACIAD,ACIAS,ACIACO,ACIACT.
Används annan serieport än 6551 kan ny kod implementeras i filen IrqSuprt.src och System.src. Tänk på att både mottagning och sändning går på IRQ.
Det torde vara lätt att ändra till 6850 eller liknande.
Vid Assemblering och länkning av de fyra assemblerfilerna ska också det anges vart måldebuggerns RAM area ligger. Dessutom ska det anges vart själva debuggerkoden ska ligga.
I princip kan måldebuggerns kod ligga vart som helst i minnet med restriktion för en sak. Samtliga avbrottsvektorer (FFF2-FFFF) måste sättas till måldebuggern. Lämpligast är att sätta måldebuggerns kod högst upp i minnet (från F400-FFFF). Finns andra redan debuggade och ROM:ade program kan dessa sättas samman i ett och samma ROM.
Ladda de olika programkoderna till en brännare och tänk på att ladda måldebuggerkoden sist. På så sätt garanteras att avbrottsvektorerna hamnar i måldebuggern.

Efter att ha bränt en EPROM med måldebuggern kan målmaskinen lätt testas med en terminal ansluten till serieporten.
Tryck RESET på målmaskinen och efter max 3 sec ska det komma upp en sträng som börjar med "I" föjlt av några tal i HEX presentation.
ex: "I0C09007FFF7FF2"
Slå då stort i ("I") och RETURN så ska samma sträng dyka upp igen efter max 3 sec.
Testa också att trycka ENTER två gånger och "E11" ska dyka upp.
Fungerar detta är måldebuggern klar att kopplas till värddatorn under CreDBG programmet.

Målmaskinen måste anslutas till COM1 för att kunna kommunicera med CreDBG. Endast GND,TX,RX behövs anslutas då kommunikationen sker med XON/XOFF handskakning. Du behöver bygla på värddatorsidan i kontakten för COM1 RTS-CTS och DTR-DCD.

Hur Arbetar man med CreDBG

CreDBG är ett tvåfönster program. Det ena fönstret är till för att arbeta med själva måldebuggern. Där sätter man brytpunker,watch-punkter,tittar i minnet,laddar filer mm.
Det andra fönstret är en reducerad VT100 terminal. Med terminalen-fönstret står man i direktkontakt men målmaskinens målkod.
Terminalfönstret öppnas varje gång man startar målprogrammet.
Om programmet träffar på en brytpunkt kan du välja på att återgå till debugg fönstret eller fortsätta körningen. Man kan också titta på watchpunkterna i terminalfönstret.
Vid Uppstart av programmet kommer CreDBG be dig om att trycka RESET på målmaskinen. Detta för att säkerställa kommunikationen.
Därefter hamnar man i debugger fönstret.
På högra sidan av föntret finns en meny. Via den menyn sköts samtliga kommandon till måldebuggern.
Aktivera menyn med piltangenterna och ENTER eller via begynnelse-bokstaven i kommandot. Stora eller små bokstäver spelar ingen roll.
Har du öppnat fel fönster (fel kommando), återgå med ESC.

DebuggerMeny

ASM : Startar Disassemblern.
Fönstret till vänster aktiveras. I detta fönster kan man sätta brytpunkter/tabort brytpunkter.

Med 'R' frågar disassemblern vid vilken adress som disassembleringen ska starta. Valfri adress anges och dissasembler kommer att disassemblera och visa målkoden på angiven adress och frammåt så fönstret fylls.
Tänk på att ange rätt startadress! Annars får du en väldigt konstig kod.
Du kan fortsätta att dissasemblerar radvis frammåt med piltangenterna. Eftersom det inte går att disassemblera bakåt i minnet kommer CreDBG ihåg de sista 100 dis-assemblerade raderna.

Med 'B' togglas brytpunkt på den rad cursurn står
Då en brytpunkt sätts läser CreDBG in vilken kod som stod i minnet på aktuell adress och byterut den mot koden för SWI. Då du tar bort brytpunkten skrivs gammla koden tillbaka igen. Därför kan man bara sätt brytpunkter i RAM
Brytpunk satt med 'B' visas genom en asterix framför SWI mnemonicsen. Tidigare kod står innom parantes till höger.

Med 'C' tas samtliga brytpunker bort.

Har du själv via din Crossassembler satt en brytpunkt med SWI i din målkod kommer den att fungera som brytpunkt men den kommer inte kunna tas bort med 'B' eller 'C' kommandot

Med 'E' kan 5 bytes editeras på den rad som cursur står.
Skriv en byte i taget med mellan slag.
Med RETURN sparas dina editerade bytevärden i målmaskinen och disassemblatorn disassemblerar på nytt från adressen högst upp i fönstret för att visa vad du skrivit in.

Med ESC kommer man tillbaka till Debug menyn.

BAR : Öppnar en toppmeny.
Menyval DEBUG ger återhopp till Debugfönstret.
ESC ger också återhopp till Debug fönstret.

Menyval ENVIRONMENT öppnar ett fönster där man kan ange vilken extension som används för HEX filerna.
Denna extension används som mask då biblioteksfönstret öppnas för laddning av s1 filer till målmaskinen.
Under ENVIRONMENT anges också vilket program som ska köras igång med EXIT kommandot i debugger menyn.
Se EXIT.
ESC ger åyerhopp till barmenyn.

Menyvalet QUIT avslutar CreDBG programmet.

CONT : står för continue.
Då målkoden stoppats av brytpunk eller annat break där återstart av målkoden är möjlig görs det av CONT.
Kontrolera alltid att adressen på programräknaren (i fortsättningen PC) och stackpekaren (SP) står riktigt.
Kontrolera också att bit 7 (E flaggan) i condisioncode-registret (CC) är satt.

EXIT : "GateWay" ut tillfälligt från programmet.
CreDBG V1.2 är inte ett integrerat utvecklingsystem då det inte finns någon assembler/Compiler.
Med exit kan du ändå arbeta med CredDBG som om det vore det genom att använda EXIT för att nå din favorit editor och favorit Assembler/Compiler.
Under BAR/ENVIRONMENT kan ett programnamn matas in.
Detta program körs igång med EXIT kommandot.
CreDBG lägger sig i vila och väntar till du kört klart angivet program för att sedan återgå igen.
Alla variabler och parametrar du matat in i CreDBG finns kvar. Hela programmet ligger egentligen kvar i minnet som ett minnesresident program.
Tänk på att CreDBG kräver ca 200kb av minnet.
För att du ska kunna ha plats i minnet för att gå ut via EXIT bör du inte ha andra minnesresidenta program igång då du kör igång CreDBG. Givetvis beror det endast på hur mycket minne som din värddator är bestyckat med.

Ett bra tips är att köra igång Norton Commander eftersom där finns en editor och dessutom möjlighet att köra alla olika program (assemblers/länkare/compilers mm).

FILL : fyller minnet i målmaskinen med angivet data.
Öppnar ett fönster där startadress anges på "FromAdr" och slutadress anges på "ToAdr". Data byte att fylla minnet med anges på "Data". OBS! tänk på att fylla minst 2 bytes I annat fall fylles hela minnet med måldebuggerkrash som följd. "FromAdr" måste vara mindre än "ToAdr".

GO : kör igång målprogram.
Med "RunUserResetVector" körs programmet igång på den adress som laddats i vektorområdet.
Aktuell adress syns till höger (PC) och vilket läge SP har syns också.
Continue kör programmet vidare från brytpunkt.
Med "RunFromAdr" kan du ange startadress och värde till SP. Det sker också en test av att Entireflaggan är satt i CC så att måldebuggern kommer att köra igång allting på rätt sätt.

Efter att du kört igång programmet från GO kommer CreDBG öppnar terminalfönstert. Målprogrammet kan då kommunicera med terminalen om ditt målprogramm suportar det.

HELP : Ett informations fönster öppnas.

INFO : varm RESET:ar målmaskinen och ger information om värd och målprogrammets version
Du får också veta RAM bestyckning och UserVectorerna i målmaskinen.
Använd INFO om målmaskinen inte vill kommunicera juste.

LOAD : öppnar ett fönster för att ladda S1 filer till målmaskinen. Har ingen tidigare fil laddats kommer ett fönster av Norton typ upp och du väljer fil med piltangenterna och RETURN.
I detta fönster har du också möjlighet att döpa om, radera,copiera,skapa bibliotek.
Filnamnmasken (Extension) du angivit under ENVIRONMENT sorterar ut så att bara dina HEX filer finns tillgängliga i fönstret.
I det fall du redan laddat en S1 fil kommer CreDBG först fråga om du ska ladda ner den filen igen.
Om ja tryck RETURN
Om Nej skriv "*" och RETURN
Då öppnas Norton fönstert istället.

MOVE : flyttar datablock i minnet på målmaskinen.
Ett fönster öppnas där du anger BlockStartAdr på FromAdr.
BlockStoppAdr på ToAdr.
Till Vilken adress som blocket ska flyttas anges på NewAdr
OBS! Tänka på att flytta minst 2 byts.
Tänk på att FromAdr ska vara mindre än ToAdr.
Överlappning av minnesblocken är tillåtet. Move rutinen kontrollerar själv att inget data går förlorat.

REGS : Aktiverar fönstert där målmaskines register kan påverkas
Med "Get" hämtas de register som målmaskinen laddas med vid programstart via GO
Med "Edit" kan registern editeras.
Med "Put" skickas de editerade registrena ned till målmaskinen. "Get" aktiveras automatiskt efter "Put" för att du ska kunna verifiera att rätt värden hammnat i målmaskinen.
"Banks" håller en säkerhets kopia av värdena på registerna efter en bryt punkt. I det fallet då du editerat värdena och tror du gjort fel kan du alltid hämta in de senaste värdena på registerna med "Banks".

TERM : Öppna terminalfönstert för att du ska kunna se vad som stod där sist. Terminalen startar inte och vilken tangent som hellst ger återgång till debug fönstret.

VIEW : Öppnar ett fönster som ger dej möjlighet att undersöka och ändra minnes innehållet.
Med "Read" anger du startadress för undersöknaing.
Från den adressen och 64 byts frammåt i minnet vissas i byte form och ASCII format.
Med Piltangenterna kan du sen scrolla frammåt och bakåt i minnet.
Med "Data" kan du mata in en byte som du sen med "Write" kan skicka till målminnet.
I det fallet "Write" adressen finns i aktuellt fönster kommer fönstert att uppdateras.

WATCH : öppnar ett fönster där av dej angivena variabler visas.
Använd piltangenterna för att peka ut aktuell variabel.
Med RETURN kan du editera variabler.
Under "Label" kan ett namn på 6 tecken anges
Ges inget namn kommer Watchfönsteret ange variabeln adress.
Under "Adress" anger du variabelns adress.
Anges ingen adress raderas variabeln ur watchfönstret
Under "Size" väljer du med piltangenterna 8 eller 16 bitars storlek.
Under "Type" väljer du HEX,DEC,ASCII som presentations form i watchfönstert. ASCII kan bara vara presentations form för 8 bitars variabler.
Är inte variablelvärdet på en ASCII variabel i intervallet för standard ASCII kommer det att presenteras i HEX format.

Med 'C' rensas samtliga variabler.

Med 'R' läses variablerna in på nytt.

Brytpunkter

Oavsett vilket avbrott som inträder (Userbreak/hardware/software) kommer programmet att kunna återstartas med CONT.
Ett undantag finns. Har du själv ändrat eller skrivit in i målkoden ett SWI utan att använda funktionen under ASM att sätta brytpunkter kan värd datorn inte ta bort SWI för att byta ut det mot rätt kod. I det fallet då du i målkoden använder SWI2/SWI3 kommer inte värd-försöka att ändra målkoden vid återstart med CONT utan bara fortsätta programkörningen på nästa instruktion såvida du inte ändrat på PC.
Egna småprogram som man vill provköra avslutas lämpligast med SWI efter som då kan du inte av misstag starta programmen för fortsatt körning.
Fel adress på PC vid återstart eller vanlig start resulterar oftas i att måldebuggern spårar ur. OBS! i de fallen då du tror att debuggern spårat ur (Då inte ^C har någon verkan) tryck ALLTID på RESET FÖRST och vänta tills en sträng som börjar med 'I' ('I0C09007FFF7FF2') kommer upp på skärmen INNAN du ger något kommando med menyerna i värdprogrammet. Detta för att om måldebuggern kommit ur synk kan den troligtvis inte kommunicera via serielänken och värdprogrammet KAN spåra ur.

Efter ett avbrott/brytpunkt får du en meny med valen DEBUG/WATCH/CONT
Dessutom öppnas ett watchfönstern där de variabler du eventuelt tidigare angivit du vill titta på finns.
Med menyvalet WATCH kan du gå in och tabort eller lägga till nya variabler.
Med CONT fortsätter du att köra programkoden.

Med DEBUG kommer du tillbaka till debug fönstret.
Vid återhopp till debug fönstret kommer Disassemblatorn att aktiveras och visa var i målkoden brytpunkten eller avbrottet skedde.
Dessutom uppdateras automatiskt registervärdena i TargetRegister fönstret.

Ett tjuvknep för att kunna manipulera minnet i målmaskinen är att trycka RESET under programmets gång. (innan brytpunktmenyn har öppnats). Då kommer reset koden från måldebuggern ('I0C09007FFF7FF2') och då står du i direkt terminalkommunikation. I det läget kan du ge alla lågnivå kommandon till debugger.
Lågnivå kommandona finns beskrivena under TARGET DOC.

Terminalfönstret

När målprogrammet startas via GO eller CONT öppnas terminalfönstret och en reducerad VT100 terminal emuleras mot målmaskinen serieport. Om ditt målprogram vill komunicera till terminalen kan flera i måldebuggern implementerade rutiner användas. Rutinen PRNTXT sänder textsträngar till terminalen och GETON hämtar enstacka tecken ur den buffert som "lyssnar" på terminalen.
Många gånger kan det vara bättre att låta skicka lite ledtexter om vad målprogrammet håller på med istället för att sätta in en massa brytpunker som måste omstartas med CONT.
Sådan information kan sändas som enstaka texter som avslutas med CR.
Ett annat sätt kan vara att i loppar skriva ut loopens varvnr.
För att då slippa en massa onödig och långsam scroll på skärmen med fyllda buffrar och handskakning som följd kan man formater utskriften med ESC koder. Ditt målprogram har tillgång till följande ESC koder som förljer standart VT100 emulering.

Tillgängliga VT100 terminalemuleringskoder:

```
ESC[xA        = Cursur Upp x steg
ESC[xB        = Cursur Ned x steg
ESC[xC        = Cursur Fram x steg
ESC[xD        = Cursur Bak x steg
ESC[y;xH      = Cursur till position (x,y)
ESC[25l       = Stäng av cursur
ESC[25h       = Sätt på cursur

ESC[2J        = Rensa Skärmen
ESC[0K        = Rensa Till radslut från Cursur position
ESC[2K        = Rensa hela raden  Radnr = cursur position

ESC[xL        = Sätt in x antal rader  Radnr = Cursur position
ESC[xM        = Tabort x antal rader   Radnr = Cursur position

ESC[0m        = Sätt Text Attribut = normal
ESC[1m        =          ""        = högintensiv
ESC[7m        =          ""        = inverterad

* Tillåtet format på x och y  = '0','01','12'.
* ESC = ASCII 27.
```

Vill du mata in information till målprogrammet är det bara för målprogrammet att använda GETON rutinen för att hämta tecken som kommer till inbufferten varje gång du trycker på tangentbordet.
Alla vanliga tangenter sänds som ASCII.
Dessutom finns några övriga tangenter som skicka speciella ESC koder.
Tillgängliga funktionstangenter med respektive ESC kod är:

```
PilUpp  = ESC[A
PilNed  = ESC[B
PilFram = ESC[C
PilBak  = ESC[D
PilEnd  = ESC[K
PilHome = ESC[H
DEL     = ASCII 127
```

Terminalspecifica tangenter

Några tangenter är vikta speciellt för själva terminalen.

Med ALT_C rensas terminalskärmen.

Med ALT_I ominitieras värddatorns serieport.
Detta för att kunna få tillbaka en urspårad måldebugger.

Med F1 hämtas ett hjälpfönster in.

Innehåll.

Kommandon och svar

Debuggern i målmaskinen kommuniserar via en ACIA på målkortet. De "samtal" som förs är av lågnivåtyp vilket innebär att debuggern i målmaskinen endast skickar koder till värddatorn och vise versa. Dessa koder delas in i två kategorier. Kategori 1 är sådana koder som är direkta svar på frågor från värddatorn. Dessa koder börjar alltid med en versal vilken talar om vilket svar som kommer. Ex frågar värddatorn vad som finns på adress $FF00 med frågan "RFF00" svara målmaskinen med "RFF0045" där 45 är datat på adress FF00 (alltid hex). Kategori 2 är sådana koder som målmaskinen vill göra värddatorn uppmärksam på. Ex error medelanden mm. Dessa koder består också av en versal samt ett hextal men föregås alltid av ett ^C för att värddatorn ska uppmärksamma att det hänt något i målmaskinen som den vill göra värddatorn uppmärksam på. Båda kategorierna av svar från målmaskinen avslutas med CRLF och målmaskinen väntar på att värddatorn minst ska skicka CR efter kommando.

Kommando och svarskoder i kategori 1

Kommandon från värddatorn:

"C":
Kommando för att återstarta program i målmaskinen efter brytpunkt.

"F30004000FF":
Kommando för att fylla minnet från adress $3000 till $4000 med datat $FF.

"G":
Kommando för att hämta samtliga register ur målmaskinen efter en brytpunkt.

Svar från målmaskinen:

"C":
Svar på att målmaskinen startat från brytpunkt.

"F30004000FF":
Svar på att minnet fyllts med $FF från adress $3000 till $4000.

"G0102030400050006000700080009"
Acca = $01
Accb = $02
CC = $03
DP = $04
PC = $0005
SP = $0006
US = $0007
X = $0008
Y = $0009

Kommandon från värddatorn:

"I":
Kommandot initierar om hela debbuger-målprogrammet. Testar det av debuggern känt RAM i systemet mm.

"L" [s1 fil]:
Kommando för att väcka måldebuggern på att det kommer en s1 fil som ska laddas ned i RAM.

"MC000D000C400":
Kommando för att flytta datablock i minnet.
Blockstart = $C000
Blockstopp = $D000
Ny blockstartadress = $C400

"P0102030400050006000700080009":
Kommando för att från värddatorn lägga ned nya värden till registren. Registren laddas innan målprogrammet körs igång.
Acca = $01
Accb = $02
CC = $03
DP = $04
PC = $0005
SP = $0006
US = $0007
X = $0008
Y = $0009

Svar från målmaskinen:

"I0C8000DFFFDFF2"
Svarar med att tala om att versionen är $0C = 12 = 1.2
Start av RAM för målprogram = $8000.
målprogram RAM topp = $DFFF
Adress till användarens avbrottsvektorer startar på adressen $DFF2.

"LW":
Svar på att måldebuggern är redo för att ta emot en s1-fil.
"LO":
Talar om att mottagnigen har börjat.
"LR96FF":
Talar om att mottagningen är klar och att filslutet låg på adress $96FF

"MC000D000C400":
Svarar att blocket $C000-$D000 har flyttats till $C400 och uppåt.

"P":
Svar på att registern fått nya värden.

Kommandon från värddatorn:	Svar från målmaskinen:
"R8000": Kommando för att läsa data på adress $8000	"R8000FF": Svar att data på adress $8000 är $FF
"SFF03": Kommando för att starta mål-programmet med laddade register från adress $FF03	"SFF03": Svar på att programmet startat på adress $FF03
"VC000D000": Kommando för att läsa ett block data ur minnet med startadress $C000 och slutadress $D000.	"VC000xxyyzz....??": Svar på att data som följer börjar på adress $C000. xx = datat på adress $C000 yy = datat på adress $C001 zz = datat på adress $C002 .. = följande data ?? = datat på adress $D000
"W8900FF": Kommando för att skriva datat $FF till adress $8900.	"W8900FF": Svar på att datat på adress $8900 läst efter det att det skrivits är $FF Ingenting hindrar från att skriva på en ROM adress men då blir inte svarets data lika med det som kommandot angivit.

Error koder (svars kategori 2)

Dessa koder är beteknade som error koder då de i måldebuggern behandlas lika. Det vill säga att då målprogrammet stöter på en brytpunkt ser debuggern detta som ett avbrott av felkaraktär och stoppar målprogrammet för att sedan köra igång debuggern.
Vid "rena" error då debuggern ligger i debugger "mode" startas också debuggern upp på nytt med en error kod till värddatorn som följd.
Det är alltså upp till värddatorn att bedömma vad som är error och vad som inte är error i måldebuggern.
En error kod förgås alltid av ett ^C och där efter "Exx" där xx är ett nummer som hänvisar till ett specifikt error.

Error koder:	Betydelse:
^C"E00"	RAM fel vid systemstart eller varmreset.
^C"E01"	Brytpunkt hittad i målprogram.
^C"E02"	Ett oinitierat SWI3 uppstod.
^C"E03"	Ett oinitierat SWI2 uppstod.
^C"E04"	Ett oinitierat FIRQ uppstod.
^C"E05"	Ett oinitierat IRQ uppstod.
^C"E06"	Ett oinitierat NMI uppstod.
^C"E07"	Målprogrammets psevdoresetvektor oinitierad.
^C"E10"	För många tecken i kommandosträng.
^C"E11"	Icke känt kommando.
^C"E12"	Checksumma fel vid laddning av S1 fil.
^C"E13"	För många tecken i laddad S1 sträng.
^C"E21"	ACIA framing error.
^C"E22"	ACIA overrun error.
^C"E23"	ACIA paritet error.
^C"E24"	ACIA spök IRQ.
^C"E25"	^C mottagits.

Error koder mer i detalj.

E00 RAM error vid systemstart eller varmreset via kommandot "I"
Då debuggern initieras kommer det RAM som definierats i system filen att testas. Två RAM areor definieras.
USERAM - USRATO är det område som är till för målprogram.
DBRAM - DBRAM+1024 är det område som är specielt för debuggern.
Om någon adress inte svarar med samma data som skrivits till adressen kommer debuggern att skicka ^C"E00" till värddatorn och stoppa all program körning.
E00 är ett allvarligt error. Kontrollera att målmaskinen är riktigt bestyckad men RAM. I annat fall får man byta ut RAM kretsen (kretsarna).

E01 Brytpunkt hittad i målprogramet.
Detta tycker debuggern är ett error eftersom program-körningen avbröts. Vad som hänt är att målprogrammet stött på ett SWI i målkoden och gjort inhopp till debuggern via SWI avbrottsvektor. Om ett "wedge" prog-ram implementerats via SWI vektorn har det också gjorts. Dessutom har alla processorn register lagts upp på psevdo variabler för att kunna manipuleras av värddatorn via kommanodot "P".
Efter ett avbrott och om inte PC påverkats av värddatorn kan programmet i målkoden fortsättas med kommandot "C". SP får heller inte ändras om programmet ska kunna fortsätta.

E02 Ett oinitierat SWI3 uppträdde.
Målprogrammet stötte på ett SWI3 i målkoden och gjorde hopp via avbrottsvektorn i ROM till psevdoavbrottsvektorn i RAM som inte var initierad för hopp till målkoden utan gjorde inhopp till debuggern igen. Detta händer alltså då man inte initierat psvedovektornerna från målprogrammet.
Precis som vid en brytpunkt kommer processorns samtliga register att sparas undan och debuggern startar upp och väntar på kommandon från värddatorn. Målprogrammet kan precis som vid brytpunkter återstartas om inte PC registert ändrats av värddatorn. Övriga register kan ändras förutom SP som inte heller här får ändras.

E03 Ett oinitierat SWI2 uppträdde.
Samma betingelser gäller här som vid E02 fast nu skedde hopp via psevdoavbrottsvektor SWI2.

E04 Ett oinitierat FIRQ uppträdde.
Samma betingelser gäller här som vid E02 fast nu skedde hopp via psevdoavbrottsvektor FIRQ.
En viktigt skillnad finns dock! Vid FIRQ avbrott sparas bara CC och PC på stacken. Detta ordnar debuggern så att samtliga register kan nås via Get/Put kommandona.
Vid återstart laddas också samtliga variabler in så continue kommandot fungerar som vanligt.
En liten skilland som man ska lägga märke till för att detta ska fungera är att debuggern själv sätter Entire-flaggan i CC. (Den är normalt inte satt vid FIRQ)

E05 Ett oinitierat IRQ uppträdde.
Samma betingelser gäller här som vid E02 fast nu skedde hopp via psevdoavbrottsvektor IRQ.
Eftersom att debuggerprogrammet själv nyttjar IRQ avbrott kommer debbuggern först att kontrollera om avbrottet kom från ACIA:n som är kommunikationsport med värddatorn.
Detta test tar ca 20 E cykler och bör tas hänsyn till i det läge då målprogrammet verkligen nyttjar IRQ avbrott.
Det kan ju finnas applikationer som inte kan vänta så länge, Ex snabb kommunikation. Vid utveckling av sådana applikationer föreslås istället att man nyttjar NMI avbrott under själva testningen av avbrotts rutinen.
I den färdiga applikationen kopplar man sedan om till IRQ avbrott då debuggern inte finns med.

E06 Ett oinitierat NMI uppträdde.
Samma betingelser gäller här som vid E02 fast nu skedde hopp via psevdoavbrottsvektor NMI.

E07 Ett försök att starta målprogrammet via psevdoramvektorn RESET då denna vektor ej initierats.
Denna vektor ska laddas då målkoden laddas ned i debuggern i form av FCB/FDB assembler direktiv av målkoden.
Det gäller förövrigt de övriga psevdoramvektorerna som ska användas också.

E10 För många tecken i kommandosträngen.
Ett kommando till debuggern ska alltid avslutas med ett ^M = #13 = "Carrige Return" (Vagn retur).
Debuggerns kommandobuffert kan maximalt ta 40 tecken.
Inget av kommandona till debugger är eller får vara längre än så. Troligt fel kan vara att värddatorn inte har skickat "CR" som skiljetecken eller att värddatorn inte supportar XON/XOFF.

E11 Icke känt kommando.
Varje kommando börjar med en versal som leder debuggern till rätt kommando rutin. Då ett icke känt tecken inleder ett kommando skickas detta error. Troligt fel är att värddatorn "sjablat" med kommandot eller att någon störning på kommunikationskabeln orsakat upphov till ett tecken som debuggern tagit in i inbufferten.

E12 Checksumfel vid laddning av S1-Fil.
Detta error uppstår endast vid laddning av S1 filer. Mottagningen av S1 filen upphör och debuggern lägger sig i väntläge på kommando.

E13 S1 sträng för lång.
En S1 sträng i en S1 fil får inte innehålla fler än 80 tkn. Mottagningen av S1 fil upphör och debuggern lägger sig i väntläge för kommando.

E21 ACIA framing error.
Spökpulser (störningar) har inkommit på kommunikations ledningen mellan värddatorn och debuggern.
Debuggern måste troligtvis resetas.

E22 ACIA overrun error.
Eventuell hårdvaruhandskakning (CTS,RTS,DTR) ingorerats av värddatorn. En reset av debuggern måste ske.

E23 ACIA paritet error.
Fel kommunikationstyp från värddatorn.

E24 ACIA spök IRQ.
Störning på kommunikationsledningen gav upphov till ett IRQ i debuggern som härstammade från ACIA:n men ACIA hade inte tagit emot något tkn.

E25 ^C mottogs.
Debbuggern avbryter eventuell programkörning och lägger sig att vänta på kommando.
Ett avbrott med ^C får samma effekt som en brytpunkt och går på så sätt att återstartas med Continue.

AVBROTT

Alla typer av avbrott gör inhopp till avbrottsvektorerna i debugger ROM:et ($FFF2-$FFFE). Debuggern nyttjar två av avbrotten för egen del. Det ena avbrottet (SWI) är endast menat till debuggern. SWI uppträder då målprogrammet träffat på en brytpunkt. Hur debuggern behandlar brytpunkter finns omskrivet under rubriken HOST DOC. "Brytpunkt".
Det andra avbrottet som kan vara menat till debuggern är IRQ. Då ett IRQ uppträder kontrollerar debuggern om IRQ avbrottet härstammar från den ACIA som är ansluten till värddatorn. Om det är så har det kommit ett tecken från värddatorn och detta sparas på inbufferten. IRQ avbrott sker också då man sändet till värddatorn. Om IRQ avbrottet inte kom från ACIA:n kommer debuggern att ge program kontrollen till målprogrammet om målprogrammet initierat den IRQ vektor som finns i RAM. Alla avbrottsvektorer finns representerade i RAM vektorer Dessa vektorer initieras av debuggern vid reset eller varmreset ("I" kommando från värddatorn) till att peka in i debugger ROM:et. Då målprogrammet inte initierat en avbrottsvektor och avbrott till den vektorn uppträder kommer debuggern att svara med ett error medelande som talar om att ett icke initierat avbrott uppträtt. Ett sådant avbrott får samma funktion som en brytpunkt och går att starta med "Continue" kommandot.

En speciel USER vektor är SWI vektorn. Då man vill sätta brytpunkt i en IRQ/FIRQ/NMI rutin måste en mycket viktig sak göras.
För att det överhuvutaget ska gå att sätta brytpunkt i en sådan rutin måste avbrottet gå via en maskbar port.
En speciell brytpunks rutin måste skrivas som när målkoden startar greppar SWI vektorn och ser till att den pekar på detta brytpunkts-program. När det blir SWI kommer måldebuggern att hoppa in till måldebuggerkoden i ROM för att sedan hoppa via SWI vektorn till detta brytpunktsprogram du själv skrivit.
Ditt brytpunktsprogram måste maska bort så att inga fler avbrott kan uppstå. Därefter ska ditt brytpunksprogram hoppa till den adress som tidigare stod på SWI vektorn för att måldebuggern ska kunna behandla ditt avbrott. Man brukar kalla detta SwaptVektors eller wedge rutin. (wedge = kil).
Ditt brytpunkts program får heller inte flytta SP värdet.
Efter behandlad brytpunkt för att köra vidare med CONT ska PC ställas till ett program som maskar av och där efter kör vidare i koden.
I det fallet kommer du att bli av med något register. Att detta är viktigt fårstår man lätt genom att om inte avbrottet maskas bort kommer debuggern hela tiden att hitta denna brytpunkt vid repetitiva avbrott.

Vart Psevdo avbrotssvektorerna i RAM ligger definieras av debugger programmet i målmaskinen. De ligger samlade från en basadress och har samma ordning som de fasta avbrottsvektorerna i ROM:et ($FFF2-). Med Kommandot "I" som initierar måldebuggern får man information om var RAM avbrottsvektorerna ligger.

RAM avbrottsvektorer

RamVektorStart	SWI3 UserRamVektor
RamVektorStart + 2	SWI2 UserRamVektor
RamVektorStart + 4	FIRQ UserRamVektor
RamVektorStart + 6	IRQ UserRamVektor
RamVektorStart + 8	SWI vedge vektor
RamVektorStart + 10	NMI UserRamVektor
RamVektorStart + 12	START UserPGM

ProgrameringTips

När man kör ett målprogram och har en värddator som ligger i terminal mode kan det vara bra att kunna kommunicera med den. Därför finns det implementerat en hopptabell till måldebuggerns I/O rutiner. Via hopptabellen kan ett målprogram komma åt de I/O rutiner som måldebuggern själv använder för att kommunicera med värddatorn. Via hopptabellen kan målkoden också komma åt andra rutiner som kan underlätta kommunikationen. Oavsett vilken version av måldebugger man har kommer alltid de tidigare versionerna ha samma adress i hopptabellen.
Vill man använda dessa rutiner i ett färdigt program senare får man naturligtvis skriva in dessa rutiner i sin egen målkod.

Hopptabell

FFD8	JMP PRNTXT	Sänder en sträng via serieporten
FFDB	JMP GETON	Hämta ett tkn ur inbufferten.
FFDE	JMP INIO	Returnerar antal fria byts i inbufferten
FFE1	JMP CHR16B	Omvandlar 4 tkn (HexTal) till 16 bitarsvärde
FFE4	JMP BYTBCD	Omvandlar en byte till BCD tkn
FFE7	JMP KONHEX	Omvandlar tkn-byte/byte-tkn (0-F)
FFEA	JMP BYTCHR	Omvandlar 8bitsbyte till 2 tkn HEX
FFED	JMP CHRBYT	Omvandlar 2 tkn HEX till 8bitarsbyte

CHRBYT

OMVANDLAR TVÅ HEXADECIMALA TKN TILL EN 8 BITS BYTE. LADDA ACC A MED MSB TKN. LADDA ACC B MED LSB TKN. ANROPA RUTINEN OCH SVARET RETURNERAS I ACC A. OM ETT ICKE HEXADECIMALT TKN ANGIVITS RETURNERAS ERROR VIA CARRY

ADRESS	$FFED
KOMMUNIKATIONS REG	A,B
PÅVERKADE REGISTER	-
STACK ÅTGÅNG	2 (1)
ANROPS NAMN	CHRBYT
ERROR	CARRY
FÖRBEREDANDE RUTINER	-
ANNAN ANVÄND SUBRUT	KONHEX

BYTCHR

OMVANDLAR EN BYTE TILL TVÅ HEXADECIMALA TKN. LADDA ACC A MED BYTE. ANROPA RUTINEN BYTCHR OCH ACC A RETURNERAS MED MSB TKN OCH ACC B RETURNERAR LSB TKN

ADRESS	$FFEA
KOMMUNIKATIONS REG	A,B
PÅVERKADE REGISTER	-
STACK ÅTGÅNG	2 (1)
ANROPS NAMN	BYTCHR
ERROR	-
FÖRBEREDANDE RUTINER	-
ANNAN ANVÄND SUBRUT	KONHEX

KONHEX

SUBRUTIN FÖR CONVERTERING AV HEX-TECKEN TILL BINÄR-TAL ELLER TVÄRTOM.
CARRY=0 LADDA ACCA MED TECKEN OCH ANROPA
SVARET FINNS I ACCA
CARRY=1 LADDA ACCA MED 0-15 OCH ANROPA
SVARET FINNS I ACCA

ADRESS	$FFE7
KOMMUNIKATIOSREG.	A
PÅVERKADE REG.	B
STACKÅTGÅNG	1
ANROPNINGSNAMN	KONHEX
ERROR	CARRY
FÖRBEREDANDE RUTIN	0
ANNAN ANVÄND SUBRUT	0

BYTBCD

DENNA RUTIN OMVANDLAR EN HEX BYTE TILL EN BCD BYTE.
ERROR = CARRY OM STÖRRE BYTE ÄN $63 ELLER D99
LADDA ACC A MED HEX BYT OCH HÄMTA SVARET I ACC A

ADRESS	$FFE4
KOMUNIKATIONS REG	A
PÅVERKADE REG	A,B
STACKÅTGÅNG	1
ANROPS NAMN	BYTBCD
ERROR	CARRY
FÖRBEREDANDE RUTINEN	0
ANNAN ANVÄND SUBRUT	0

CHR16B

Omvandlar 4 HEX character till en 16 bits hextal.
Anropa med Y med en adress pekande på första tkn.
talet returneras i ACCD (A=MSB,B=LSB)

ADRESS	$FFE1
KOMMUNIKATIONS REG	Y,A,B
PÅVERKADE REGISTER	A,B
STACKÅTGÅNG	6
ERROR	CARRY
FÖRBEREDANDE RUTINER	-
ANDRA ANVÄNDA RUTINER	CHRBYT[KONHEX]

INIO

DENNA RUTIN KONTROLERA HUR MÅNGA FRIA BYTS DET FINNS I IN BUFFERTEN. ANROPA RUTINEN JSR INIO. ACC A RETURNERAR ANTAL FRIA BYTS I UTBUF

ADRESS	$FFDE
KOMUNIKATIONS REG	A
PÅVERKADE REGISTER	-
STACKÅTGÅNG	1
ERROR	-
FÖRBEREDANDE RUTINER	-
ANNAN ANVÄND SUBRUT	-

GETON

SUBRUTIN FÖR LÄSNING AV TKN UR INBUFFERTEN I DEN TURORDNING DE INKOMMIT. ANROPA RUTINEN MED JSR GETIN OCH TKN RETURNERAS I ACC A OM BUFFERTEN ÄR TOM RETURNERAS 0 I ACC B ANNARS ÄR ACC B=1

ADRESS	$FFDB
KOMUNIKATIONS REG	A,B
PÅVERKADE REGISTER	-
STACKÅTGÅNG	1
ERROR	B (B=0=TOM INBUFFERT)
FÖRBEREDANDE RUTINER	-
ANNAN ANVÄND SUBRUT	-

PRNTXT

DENNA RUTIN SKICKAR EN TECKEN STRÄNG TILL UTBUFFERTEN OCH STARTAR SÄNDIRQ.
ANROPA MED JSR PRNTXT OCH Y LADDAT MED STARTADRESS TILL STRÄNGEN.
STRÄNGFORMATET ÄR:
BYTE 1 = ANTAL FAKTISKT SÄNDA TKN
BYTT 2-N = STRÄNG
BYTE N+1 = SLUT TECKEN
MAX ANTAL FAKTISK SÄNDA TKN = 254
TEST GÖR OM STRÄNGEN RYMMS I UTBUFFERTEN. OM INTE RETURNERAS ACCB = 0. OM SÄNDNINGEN TILL BUFFERTEN GICK HEM RETURNERAS ACCB = 1.

SPECIELLA TKN FÖR STRÄNGTEXT PRESENTERAS NEDAN.

@ = AVSLUTAR ALL TXT
~ = ESC = $1B
| = CR OCH LF
_ = CR

ADRESS	$FFD8
KOMUNIKATIONSREG	Y,B
PÅVERKADE REGISTER	A,B,X,Y
STACKÅTGÅNG	3
ERROR	B = 0 = PLATSBRIST
FÖRBEREDANDE RUTINER	-
ANDRA ANVÄNDA SUBRUT	-