Introduktion

DBG32 är en monitor/debugger för MC683xx-baserade mikrodatorer. DBG32 ingår i en serie likartade monitor/debugger's för mikrodatorer. För närvarande finns följande varianter:

DBG11	För MC68HC11, speciellt mikrodatorn microlf MC11
DBG12	För MC68HCS12, speciellt mikrodatorn microlf MC12
DBG32	För MC68340, speciellt mikrodatorn microlf MC68

Enhetligheten hos de olika varianterna tjänar flera syften, de viktigaste är:

- Olika mikrodatorvarianter ger samma användargränssnitt, det går därför snabbt att introduceras till en ny mikroprocessor/controller då du en gång lärt dig använda DBG32.
- Källtextdebuggers (som exempelvis ETERM 7 och XCC) har inbyggt stöd för att kommunicera med DBGxx.

I resten av denna beskrivning används beteckningen DBG för att beskriva funktioner som är gemensamma för alla varianter. Processorspecifika avsnitt använder beteckningen DBG32.

Funktion

OBSERVERA
DBG32 kommunicerar med hastigheten 38400 Baud.
Du kan behöva ändra inställningar hos Terminalfunktionen

Då du startar DBG kommer systemet att initieras, dvs. systemet förbereds för att användas som monitor/debugger. Dessa förberedelser innebär bland annat att DBG initierar in- och ut-portar och seriekommunikatio.

DBG identifierar därefter sig genom utskrift av *namn* och *version*.

Följande kommandon accepteras av DBG

mm - modify memory	Ändra minnesinnehåll
dm - display memory	Visa minnesinnehåll
tr - trace instruction	Utför enstaka maskininstruktion
go - run program	Utför användarprogram
dasm - dissassemble memory	Dissassemblera minnesinnehåll
reg - display/modify registers	Visa/ändra processorns registerinnehåll
bp - breakpoints	Hantera brytpunkter i användarprogram
l - load from host	Överför program från värddator
help - display online help	Inbyggt hjälpsystem

Följande kommandon accepteras dessutom speciellt av DBG32:

bootloader	Starta MC68 bootloader (MC68 R1)
fmaint - flash maintenance	Starta användarprogram MC68/ML11

Textkonventioner och notation

Text som skrivs ut av DBG återges med `courier` stil.
Text som skall matas in till DBG återges med understruken courier stil.

Speciella tangenter anges med *små hakar*.
Tangenten "vagnretur", dvs <Enter> på vissa tangentbord, <Return> på andra återges med sekvensen <CR> ("carriage return").

I texten använd *stora hakar* för att ange en parameter som *kan* men inte *behöver* ges.
[*option*] anger att *option* kan utelämnas. Observera, stora hakar ska *inte* ges som kommando.

I texten används *vertical bar* (|) för att ange *alternativ*.
[-b|w|l] anger att *något av* -b, -w eller -l *kan*, men *behöver inte* anges.
b: byte (8 bitar)
w: word (16 bitar)
l:long (32 bitar)
Då detta alternativ utelämnas använder DBG *standardstorlek*.

Speciellt för DBG12:
- Alternativet 'l' används ej.
- *Standardstorlek* är 'b'

Indata kan under vissa omständigheter ges i något av talsystemen *binär*, *oktal*¸ *decimal* eller *hexadecimal* form. Du anger talsystem med ett prefix till det inmatade värdet:
%*värde* anger *binär* form
@*värde* anger *oktal* form

t*värde*
T*värde* anger *decimal* form

$*värde* ***eller***
värde anger *hexadecimal* form

Monitorkommandon

Modify memory
mm [size] address [value]

Med mm-kommandot kan minnesinnehåll visas och ändras. Kommandot har två former, där den första formen används för att modifiera ett enstaka element, den andra formen startar en *interaktiv mod*.

Former:

mm [-b|w|l] *address value*<CR>
Används för att ändra enstaka element på *address* till *value*. *size* kan, men behöver inte anges. Om *size* utelämnas används standardstorleken.

mm [-b|w|l] *address*<CR>
Formen används för att starta *interaktiv mod*. Nu skrivs *address* följt av innehållet ut till bildskärmen.

I *interaktiv mod* kan följande kommandon ges:

nytt värde<CR>	modifiera minnesinnehåll
-	ingen ändring, visa föregående
+	ingen ändring, visa nästa
.	('punkt') ingen ändring, avsluta

Display Memory
dm [size] address [count]

dm används för att visa minnesinnehåll. En *startadress*, dvs. adressen till den första minnespositionen, anges. Minnesinnehållet kan visas med formen *byte* (8 bitar), *word* (16 bitar) eller *long* (32 bitar). Minnesinnehållet visas *alltid* på *hexadecimal* form *och* med ASCII-representation.

Former:

dm [-b|w|l] *address* [*count*]<CR>

size-fältet *kan* men *behöver ej* anges. Om *size* utelämnas används standardstorleken.

address-fältet måste *alltid* anges. Adressen anges på *hexadecimal* form. Udda adress kan endast anges om *size*-fältet samtidigt är -b.

count-fältet *kan* men *behöver ej* anges. *count* ges på *decimal* form. Om *count*-fältet utelämnas kommer DBG att bestämma antalet visade ord beroende på *size*-fältet. Om *size* är -b , kommer 256 bytes att visas, om *size* är -w kommer 128 words att visas, om *size* är -l, kommer 64 long words att visas.

Trace instruction(s)
`tr address [count]`

`tr`-kommandot används för att utföra *enstaka* instruktioner. Detta kan jämföras med att sätta en brytpunkt *på varje* instruktion i programmet. Efter utfört tr-kommando skriver DBG *användarregistrens* innehåll till bildskärmen.

Former:

`tr` *address count*<CR>
Utför *count* (tolkas som decimal form) antal instruktioner, den första på *address* (tolkas som hexadecimal form).

`tr` *address* <CR>
Utför *en* instruktion med start på *address*.

`tr`<CR>
Utför instruktionen på den adress som anges av *användarregister PC*.

`tr +`<CR>
Aktiverar kortkommando för `tr`, (“HOT trace”). Då kortkommandot är aktivt räcker det med att trycka ‘.’ (punkt) från DBG's prompter för att utföra kommandot `tr`<CR>.

`tr -`<CR>
Stänger av kortkommandot för `tr`.

Execute program
`go [-n] [address]`

go-kommandot används för att starta ett tillämpningsprogram. Då DBG startar tillämpningsprogrammet kommer innehållet i *användarregistren* (se *Display/Modify Registers*) först att laddas till processorns register.

Former:

`go` *address*<CR>
Formen används för att starta ett tillämpningsprogram som börjar på *address*.

`go`<CR>
Samma som föregående men programmet startas från adress som anges av *användarregistret PC*.

`go -n`<CR>
Utför program *till nästa instruktion*. Speciellt används formen då man vill utföra en hel subrutin.

Dissassembly
dasm [address] [count]

Med dasm-kommandot kan minnesinnehåll *disassembleras*. Dvs. DBG läser innehållet på någon adress, *tolkar* detta som en *maskininstruktion* och skriver ut den *mnemonic* och de operander som anges av instruktionen.

Former:

dasm *address*<CR>
Disassemblera 10 instruktioner med start på *address*. Information om den sist disassemblerarade instruktionen sparas internt av DBG.

dasm *address instructions*<CR>
Med denna form ges också det antal instruktioner (*count*) som skall disassembleras. *address* tolkas av DBG som hexadecimalt, medan *count* tolkas som decimalt.

dasm<CR>
Den enklaste formen används för att fortsätta disassembleringen där den sist avslutades. 10 instruktioner disassembleras.

Display/Modify registers
reg [regname] [value]

reg-kommandot används för att *visa* eller *ändra* de registervärden som används vid utförande av tillämpningsprogram (*användarregister*). Dessa värden laddas i registren av go- respektive tr-kommandona. Vid brytpunkt sparas de aktuella registervärdena i den interna tabellen för *användarregister*.

Speciellt för DBG12
Som registernamn "regname" används:

Former:

reg<CR>
Kommandot används för att *visa* registerinnehåll. DBG genererar en utskrift till bildskärmen.

reg *register value*<CR>
Kommandot används för att *ändra* de registerinnehåll som används då tillämpningsprogram utförs. *register*-fältet måste ange ett register i processorn, små eller stora bokstäver kan anges och följande namn accepteras:
d0,d1,d2,d3,d4,d5,d6,d7 för DATA-register
a0,a1,a2,a3,a4,a5,a6,a7 för ADRESS-register, i stället för a7 kan sp användas.

sr	för processorns statusregister
ccr	för de 8 minst signifikanta bitarna i processorns statusregister
pc	för processorns programräknare
usp	för “user stack pointer”
sfc	Source Function Code Register
dfc	Destination Function Code Register
vbr	Vector Base Register

value-fältet anger det värde som ska placeras i registret. Det kan anges på *binär* form (med prefix %), på *oktal* form (med prefix @), på *decimal* form (med prefix t) eller på *hexadecimal* form (utan prefix).

Breakpoints
bp [number] [action] [address]

bp-kommandot används för att *visa*, *sätta ut* och *ta bort* brytpunkter i tillämpningsprogrammet. En brytpunkt är antingen *aktiv* eller *inaktiv*. Om brytpunkten är aktiv, kommer DBG att avbryta utförandet av tillämpningsprogrammet då processorn *skall utföra* den instruktion som ersatts med brytpunkt. Brytpunkter sätts lämpligen i *början* av subrutiner som ska testas. Genom att därefter utföra programmet *instruktionsvis* (*se* "*Trace*RefTrace") kan programflödet enkelt följas och kontrolleras.

Brytpunkterna placeras (internt) av DBG i en *brytpunktstabell*, Maximalt 10 brytpunkter åt gången, kan hanteras av DBG32. Brytpunkterna numreras ("namnges") 0 t.o.m.9.

Former:

bp<CR>
Hela brytpunktstabellen skrivs till skärmen. För varje brytpunkt skrivs, om den används för tillfället, om den är aktiv och dess adress.

bp *number* set *address*<CR>
Används för att sätta ut en ny brytpunkt. *number* skall vara brytpunktens nummer och tolkas av DBG som ett decimalt tal. *address* är brytpunktens adress. Om en annan brytpunkt tidigare definierats med samma nummer modifieras denna brytpunkt. Brytpunkten är aktiv.

bp *number* dis<CR>
Inaktiverar brytpunkt *number* om denna är aktiv. Brytpunkten behålls i tabellen men orsakar inget programavbrott.

bp *number* en<CR>
Aktiverar brytpunkt *number* om denna tidigare var inaktiv.

bp *number* rem<CR>
Tar bort brytpunkt *number* ur brytpunktstabellen.

bp clear<CR>
Tar bort **samtliga** brytpunkter ur brytpunktstabellen.

Load from host
l

l-kommandot används då man vill överföra kod/data till måldatorns primärminne. DBG accepterar endast Motorola S1,S2 och S3 format.

Anmärkning
Om Du använder ETERM eller XCC behöver du inte skriva l-kommandot till DBG, detta utförs då du ger ladd-kommando till ETERM (XCC).

Display Help-menu
help

help-kommandot används i två olika former:

help<CR>
ger en översikt av *tillgängliga* kommandon.

help *command*<CR>
där *command* är ett av de angivna tillgängliga kommandona.
Ger utförligare hjälp om detta kommando

Start Bootloader
bootloader

Kommandot är bara användbart tillsammans med Revision 1 av MC68. I denna tidiga version har man ingen möjlighet att bestämma bootloadern's uppstartsförfarande med hjälp av byglar på kortet. Detta innebär att om en fungerande monitor en gång laddats finns det inget enkelt sätt att starta bootloadern för underhåll, uppgradering etc. Därför finns detta kommando som gör att DBG32 stoppas och bootloadern startas. Därefter kan MC68 revision 1 underhållas, uppgraderas på samma sätt som MC68 revision 2.

Kommandot har ingen betydelse för MC68 Revision 2 och kommer då bara att generera en informativ utskrift.

För ytterligare information om MC68 bootloader, se GMV applikationsnot 402 "Uppgradering av debugger DBG32 för microlf MC68".

Flash Maintenance
fmaint

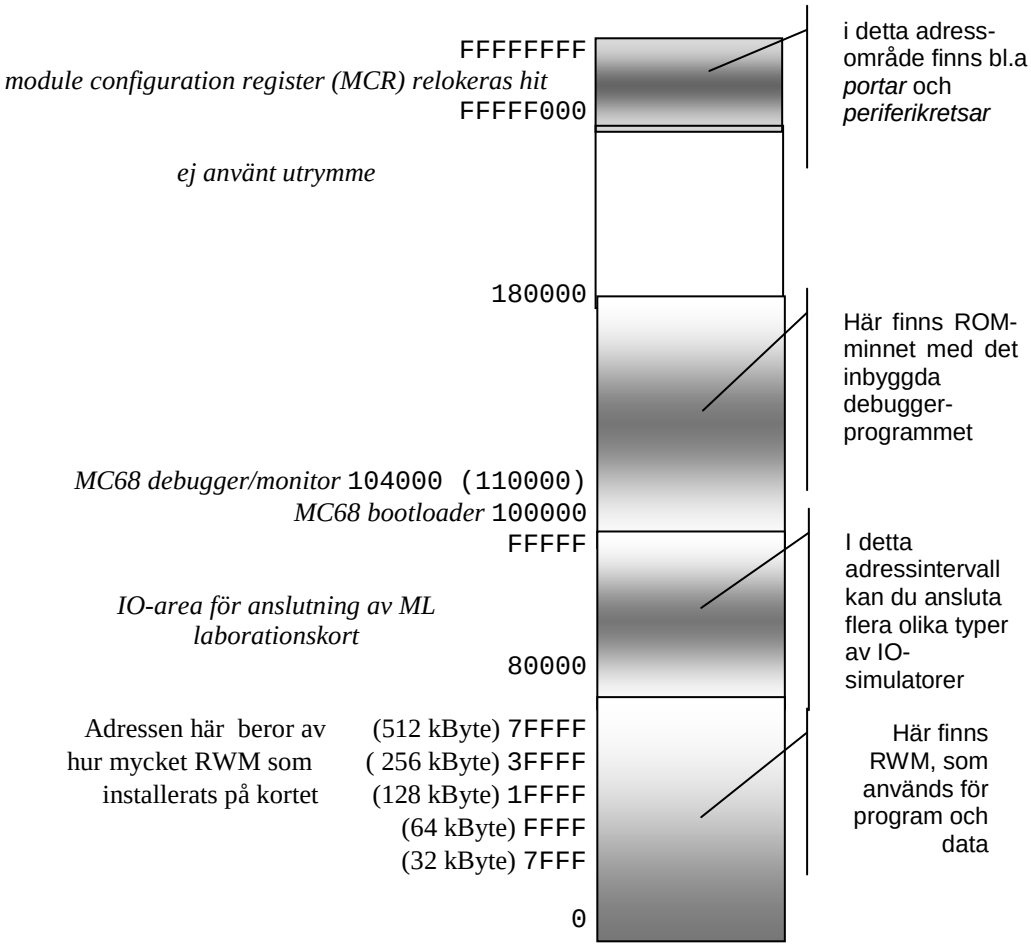
'fmaint' är ett användarprogram utformat för MC68/ML11. Med tillbehörskortet ML11 kan olika typer av FLASH-minnen programmeras. Speciellt används kortet för att initiera ett nytt minne med en 'bootloader'.

Då 'fmaint' startats från DBG32 kan flera olika kommandon ges. Dessa beskrivs kortfattat efter kommandot 'help'.

Användning av 'fmaint' beskrivs speciellt i olika applikationsnoter från GMV.

Disposition av adressrum

DBG32 disponerar adressrummet hos MC68 på följande sätt:



DBG32 använder internt RWM upp till 3FFF. Applikationsprogram kan använda intervallet 4000 och uppåt, så länge det finns fysiskt minne (beror på hur mycket minne som installerats).

Undantag för detta är minnesarean 1000-1FFF som kan användas av applikationsprogram. Speciellt bör detta minne endast användas för avbrottsvektorer.

Exception vektorer för BUS-/ADDRESS- ERROR, TRACE, TRAP 14 och TRAP 15 används internt av DBG32. Aplikationsprogram bör därför undvika att ersätta dessa för att inte interferera med debugger-funktioner.

Debugger-funktioner för in- och utmatning via serieporten använder avbrottsnivå 4. För att debuggern'n ska fungera korrekt krävs att applikationsprogrammet inte höjer processornn's avbrottsmask över denna nivå.

Inbyggda hjälprutiner

DBG32 tillhandahåller inbyggda hjälprutiner som kan vara praktiska i olika sammanhang. Rutinerna kan endast användas från assemblerprogram och utnyttjar en enkel tabell placerad efter DBG32's startpunkt.

tstchar, adress 104028 (128 kB FLASH)
tstchar, adress 110028 (512 kB FLASH)
(Test if character) kontrollera om tecken finns i DUART.
Rutinen använder samma DUART som DBG32 (DUART A). Om ett tecken finns tillgängligt returneras detta i D0-registret, annars returneras 0.
Endast D0-registret påverkas

EXEMPEL:
Följande rutin returnerar nästa tecken från DUART

```
...
inchar:
      JSR         $104028     använd DBG32/128 kb FLASH
      CMPI.B      #0,D0       finns tecken ?
      BEQ         inchar      om inte, försök igen
      RTS                     returnera med tecken i ack B
```

outchar, adress 10402B (128 kB FLASH)
outchar, adress 11002B (512 kB FLASH)
(Output character) skriv tecken till DUART
Inget register påverkas.

EXEMPEL:
Följande sekvens skriver ASCII tecknet 'A' till DUART

```
...
      MOVE.B      #'A',D0
      JSR         $10402B     använd DBG32/128 kb FLASH
...
```

restart, adress 104030 (128 kB FLASH)
restart, adress 110030 (512 kB FLASH)
Då ett applikationsprogram avslutas kan ‘restart’ anropas för att ge återstart vid DBG32’s prompter.

SYSCALLS
En alternativ metod att använda DBG32's inbyggda hjälprutiner är med användning av TRAP #14 (syscalls-metod).

Fördelen med denna metod är att applikationsprogrammet blir oberoende av absoluta adresser, dvs det spelar ingen roll om programmet utförs under DDBG32/128 k FLASH eller 512 k FLASH.

Följande funktioner är definierade:

restart (0)
Återstartar DBG32

EXEMPEL
```
        TRAP        #14
        DC.W        0
```

outchar (1)
Skriv tecken till DUART. Inget register påverkas.

EXEMPEL
```
...
        MOVE.B      #'A',D0
        TRAP        #14
        DC.W        1
...
```

tstchar (2)
Kontrollera om tecken finns i DUART. Rutinen använder samma DUART som DBG32 (DUART A). Om ett tecken finns tillgängligt returneras detta i D0-registret, annars returneras 0.
Endast D0-registret påverkas

EXEMPEL:
Följande rutin returnerar nästa tecken från DUART

EXEMPEL
```
...
inchar:
        TRAP        #14
        DC.W        2
        CMPI.B      #0,D0       finns tecken ?
        BEQ         inchar      om inte, försök igen
        RTS                     returnera med tecken i ack B
```

devinit (3)
Ger en ny initiering av DUART A. Denna funktion måste användas med viss eftertanke eftersom DBG32 utnyttjar DUART A. Det finns alltid en risk att debuggern störs då funktionen anropas från applikationsprogram.

EXEMPEL
```
        TRAP        #14
        DC.W        3
```