commodore **SuperPET** computer

Waterloo 6809 Assembler

commodore COMPUTER

Dieses Handbuch wurde gescannt, bearbeitet und ins PDF-Format konvertiert von

Rüdiger Schuldes

schuldes@itsm.uni-stuttgart.de

Waterloo 6809 Assembler

Tutorial and Reference Manual

D. D. Cowan

M. J. Shaw

Disclaimer

Waterloo Computing Systems Limited makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Waterloo Computing Systems Limited, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

PREFACE

The Waterloo 6809 Assembler is a full software development system to be used on the Commodore SuperPET Microcomputer. This system contains an assembler, a linker to connect separately assembled modules, and a monitor which includes a loader. The assembler accepts standard assembly language, macro definitions, and conditional assembly statements as well as good primitives for Structured Programming.

This manual is presented in two parts. The first part is a collection of annotated examples intended to introduce the reader to many of the features of the Waterloo 6809 Assembler. In this way, a novice is provided with a staged introduction to the language constructs in a concise manner. The second part is a comprehensive reference manual for the Waterloo 6809 Assembler.

Acknowledgement

Many people have made significant contributions to the design of the Waterloo 6809 Assembler and so it is difficult to acknowledge anyone individually. The design is based upon ideas evolved and proven over the past decade in other software projects in which these people have been involved. The major portion of the implementation was performed by Eric Mackie, John Bossom, Fred Crigger and Jack Schueler. Tammy Tilson was very helpful in the production of the manual.

D. D. Cowan,
M. J. Shaw,

June, 1981.

Table of Contents

Waterloo 6809 Assembler

Tutorial Examples

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 1

TUTORIAL

The extensive software package for the Commodore Business Machines (CBM) SuperPET includes an assembly language development system. A set of examples has been prepared which illustrates many of the features of the 6809 assembly language as well as of the development system. Working through the examples should provide a good understanding of all the basics. The sections following the examples present the details of the software development system.

When the SuperPET is first turned on, the operating system displays the following menu on the screen:

```
Waterloo microSystems

Select :

  setup
  monitor
  apl
  basic
  edit
  fortran
  pascal
  development
```

In order to enter the development system, type 'd' and press 'RETURN' on the keyboard.

After a short pause, the following menu will then be displayed on the screen:

```
Waterloo microSystems

Select:

a<sm>
e<dit>
l<inker>
m<onitor>
q<uit>
```

a<sm> is for the assembler; e<dit> is for the editor, l<inker> is for the linker; m<onitor> is for the monitor (which includes the loader); and q<uit> is to allow a return to the previous menu.

EXAMPLE 1

There are several steps involved in writing and running an assembly language program: creating the program, assembling the program, creating the linker file, linking the program, loading the program and executing the program. The program in Example 1 displays the letter 'a' in the upper left-hand corner of the screen and is used primarily to illustrate the steps in preparing and running an assembly language program. In this first example, all these steps will be illustrated; in subsequent examples, only changes which occur will be described.

Creating The Program

ex1.asm

```
;example 1
        lda #'a             ; display 'a'
        sta $8000           ; on the screen
        swi
        end
```

Notes:

1. First we enter the program using the editor (see Waterloo microEDITOR manual).

   - The first line of the above program is a comment line as specified by the semi-colon.

   - # specifies an immediate value. The character 'a' is loaded into accumulator A.

   - $ specifies a hexadecimal value. 'sta $8000' stores the contents of accumulator A into hexadecimal memory location $8000.

   - Memory location $8000 is equivalent to the upper left-hand corner of the display screen.

- 'swi' stands for software interrupt. The swi instruction stops the program, passes control to the monitor, and displays the contents of the computer's registers on the screen.

- All 6809 assembler programs must be concluded by 'end', which is an assembler directive.

2. After the program has been entered, save it in a file using the editor command p (for put). For example, 'p ex1.asm'.

   - The filename for each 6809 assembler program must be suffixed by '.asm' (for assembler).

Assembling The Program

Notes:

1. In order to assemble the program, enter 'a' (for asm) when selecting from the Waterloo microSystems menu.

   - When asked to 'Enter filename:', enter the name of the assembler file without including the '.asm' suffix. The suffix is assumed and, if it is included accidentally, a file with a '.asm.asm' suffix will be sought.

   - The assembler file will then be assembled. The assembly is complete when the 'Assembler completed' message is displayed. Press 'RETURN' to return to the menu.

2. The assembler creates two new files with names identical to the assembler filename except for the suffixes which are '.lst' and '.b09'. These files can be examined using the editor. The '.lst' file contains a listing of the original assembly language program along with its machine language translation (see 'ex1.lst' below). The '.b09' file contains the object code created by assembling the assembly language source code. This file is used by the linker described later in this example.

ex1.lst

```
0000                    ;example 1
0000 86  61                     lda #'a
0002 B7  80  00                 sta $8000
0005 3F                         swi
0006                            end
0006
```

Creating The Linker File

ex1.cmd

```
"ex1"
org $1000
"ex1.b09"
```

Notes:

1. The linker requires a '.cmd' (command) file which the user must create using the editor. The file must be saved with a name such as ex1.cmd.

   - The first line of the '.cmd' file is the name (in quotes) of the load module to be created.

   - The second line indicates the hexadecimal address at which the module is to be loaded. (org stands for origin.)

   - The third line gives the name (in quotes) of the object code file which is to be used in the desired load module.

Linking The Program

Notes:

1. To start the linking process, enter 'l' (for linker) when selecting from the Waterloo microSystems menu.

   - When asked to 'Enter filename:', enter the name of the command file without including the '.cmd' suffix which is assumed.

   - The linking step will then take place with the 'Linker completed' message indicating its completion. Press 'RETURN' to return to the menu.

2. The linker creates two new files whose names, except for the suffixes, correspond to the name given in the first line of the '.cmd' file. Their suffixes are '.mod' and '.map'. These files can be examined using the editor.

   - The '.mod' file ('ex1.mod') is the executable load module which is to be run in the next step.

   - The '.map' file contains information which shows how 'ex1.b09' is mapped into 'ex1.mod'.

Loading and Running The Program

Notes:

1. In order to run the program, invoke the monitor by entering 'm' when selecting from the Waterloo microSystems menu.

   - When prompted by '>', say 'l ex1.mod'. This will load the module created by the linker into memory at the address specified by 'org' in the '.cmd' file.

   - When next prompted by '>', say 'g 1000'. This will start execution at hexadecimal address 1000 which is the address into which the first instruction of the program was just loaded.

- The program will then run. When finished, the contents of the computer's registers will be displayed on the screen by the monitor because of the aforementioned 'swi'.

2. When prompted by '>', entering 'q' (for quit) will cause a return to the menu.

3. The monitor has many commands which are useful for debugging programs and these are described in another section of this document.

EXAMPLE 2

The following program uses a routine called putchar_ from the system library in order to display a character on the SuperPET's display screen.

Creating The Program

ex2.asm

```
;example 2
        xref putchar_           ; reference to system routine

        ldb #'a                 ; display 'a'
        jsr putchar_            ; on the screen
        swi
        end
```

Notes:

1. 'xref' defines an external reference. A module needs an xref declaration if it is used in the current program while its definition is in some other module or program.

2. putchar_ is the name of an external routine. The '_' is the left-pointing arrow on the keyboard and indicates that the routine is from the system library. The character to be displayed by putchar_ must be placed in the least significant byte of the D accumulator (B accumulator) before the routine is called.

Assembling The Program

ex2.lst

```
0000                    ;example 2
0000                            xref putchar_
0000
0000 C6  61                     ldb #'a
0002 BD  00  00                 jsr putchar_
0005 3F                         swi
0006                            end
0006
```

Notes:

1. The address for putchar_ in the line labelled 0002 is not known at assembly time and so is temporarily left as 0000.

Creating The Linker File

ex2.cmd

```
"ex2"
org $1000
include "disk/1.watlib.exp"
"ex2.b09"
```

Notes:

1. The 'include "disk/1.watlib.exp"' line is needed because a system library routine is to be used. The watlib.exp file, which should be on a diskette in disk drive 1, tells where in rom (read-only-memory) each system library routine can be found. (wat stands for Waterloo, lib for library, and exp for exports.) If this file is on a diskette in drive 0 then the 'disk/1.' designation should be deleted.

EXAMPLE 3

The following program displays a string of characters on the screen.

Creating The Program

ex3.asm

```
;example 3
            xref putchar_          ; reference to system routine

            ldx  #string           ; put address of string in X
next        ldb  ,x+               ; load character into B
            beq quit               ; quit if null byte
            pshs x                 ; save X
            jsr putchar_           ; display character
            puls  x                ; restore X
            bra next               ; repeat loop
quit        swi

string      fcc "hello"
            fcb 0
            end
```

Notes:

1. 'string fcc "hello"' defines a string of characters beginning at the address indicated by the label string. The fcc is an assembler directive meaning form constant character string.

2. 'fcb 0' puts a zero byte at the end of the string "hello" and is a convention used to indicate the end of the string. The fcb is also an assembler directive and means form constant byte.

3. Branching to the label next with the command 'bra next' puts the program into a loop.

4. Each iteration of the loop causes a character to be displayed.

5. The program exits from the loop when the null character at the end of the string is reached.

6. Index register X is used to store the address of the next character to be displayed. Its contents are pushed on the stack ('pshs x') before putchar_ is called and pulled off the stack ('puls x') afterwards because it is possible that putchar_ uses index register X as a work area. If a save and restore are not done, the character address could be lost.

EXAMPLE 4

The following program introduces some structured programming features available in the assembly language.

Creating The Program

ex4.asm

```
;example 4
            xref putchar_           ; reference to system routine

            ldx  #string            ; put address of string in X
            loop                    ; loop
               ldb,x+               ;    load character into B
               quifeq               ;    quit if null byte
               pshsx                ;    save X
               jsr putchar_         ;    display character
               pulsx                ;    restore X
            endloop                 ; endloop
            swi

string      fcc "hello"
            fcb $0d
            fcb 0
            end
```

Notes:

1. The loop/endloop control structure replaces the label next and 'bra next' from example 3. 'quif eq' replaces 'beq quit' and the label quit. quif means quit if the condition code being tested is set. The possible condition codes are:

CC - carry clear
CS - carry set
EQ - equal
GE - greater than or equal to
GT - greater than
HI - unsigned greater than (high)
HS - unsigned greater than or equal to
LE - less than or equal to
LO - unsigned less than (low)
LS - unsigned less than or equal to
LT - less than
MI - less than
NE - not equal
PL - greater than or equal to (plus)
VC - overflow clear
VS - overflow set

---

2. $0d is a 'carriage-return'. When example 4 is run, the string "hello" will now appear on a line all by itself because, after it is displayed, it will be followed by the 'carriage-return'.

Assembling The Program

ex4.lst

```
0000                          ;example 4
0000                                    xref putchar_
0000
0000 8E  00  11                         ldx #string
0003                                    loop
0003 E6  80                                 ldb ,x+
0005 27  09                                 quif eq
0007 34  10                                 pshs x
0009 BD  00  00                             jsr putchar_
000C 35  10                                 puls x
000E 20  F3                             endloop
0010 3F                                 swi
0010
0011 68  65  6C  6C  string             fcc "hello"
0016 0D                                 fcb $0d
0017 00                                 fcb 0
0018                                    end
```

Notes:

1. This listing indicates how loop, endloop, and quif are handled: loop is ignored, endloop is replaced by bra, and quif eq is replaced by beq to the instruction after endloop.

EXAMPLE 5

The following program uses two system library routines: putchar_ and putnl_.

Creating The Program

ex5.asm

```
;example 5
            xref putchar_          ; references to
            xref  putnl_           ; system routines

            ldx  #string           ; put address of string in X
            loop                   ; loop
                ldb,x+             ;    load character into B
                quifeq             ;    quit if null byte
                pshsx              ;    save X
                jsr putchar_       ;    display character
                pulsx              ;    restore X
            endloop                ; endloop
            jsr putnl_             ; skip to a new line
            swi

string      fcc "hello"
            fcb 0
            end
```

Notes:

1. putnl_ is a routine which skips to a new line on the display screen.

EXAMPLE 6

The following program replaces the loop/endloop with a loop/until control structure.

Creating The Program

ex6.asm

```
;example 6
            xref putchar_          ; references to
            xref  putnl_           ; system routines

            ldx  #string           ; put address of string in X
            ldb  ,x+               ; load character into B
            loop                   ; loop
               pshsx               ;    save X
               jsr putchar_        ;    display character
               pulsx               ;    restore X
               ldb,x+              ;    load character into B
            until  eq              ; until null byte in B
            jsr putnl_             ; skip to a new line
            swi

string      fcc "hello"
            fcb 0
            end
```

Notes:

1. This program is not equivalent to the previous one in that this one would not work if "hello" was replaced by the null string. The check for the null character here comes after putchar_ has already been called once.

EXAMPLE 7

The following program uses the system library routine getchar_ to input a character from the terminal and then, using the if/else/endif construct, decides which of two strings to display.

Creating The Program

ex7.asm

```
;example 7
            xref getchar_           ; references to
            xref putchar_           ; system routines
            xref putnl_

            jsr getchar_            ; read character
            cmpb #'h
            if eq                   ; if equal to 'h'
                ldx #hello          ;   load address of "hello"
            else                    ; else
                ldx #goodbye        ;   load address of "goodbye"
            endif                   ; endif
            loop                    ; loop
                ldb,x+              ;   load character into B
                quif eq             ;   quit if null byte
                pshs x              ;   store X
                jsr putchar_        ;   display character
                puls x              ;   restore X
            endloop                 ; endloop
            jsr putnl_              ; skip to a new line
            swi

hello       fcc "hello"
            fcb 0
goodbye     fcc "goodbye"
            fcb 0
            end
```

Notes:

1. The if/else/endif is a useful construction for selection which can improve the readability of a program. The if is used with condition codes in a manner similar to the quif. The reader might wish to examine the '.lst' file for this example to see how these constructs translate into machine language.

2. If the input character is an 'h' then the string "hello" is displayed; otherwise, the string "goodbye" is displayed.

3. The character which is read is returned in the least significant byte of accumulator D.

EXAMPLE 8

The following program demonstrates how to implement a separately assembled subroutine called display.

Creating The Program

ex8m.asm

```
;example 8 mainline routine
            xref initstd_           ; references to
            xref getchar_           ; system routines
            xref putnl_
            xref display            ; reference to user routine

            jsr initstd_            ; initialize standard I/O
            jsr getchar_            ; read character
            cmpb #'h
            if eq                   ; if equal to 'h'
               ldd #hello           ;    load address of "hello"
            else                    ; else
               ldd #goodbye         ;    load address of "goodbye"
            endif                   ; endif
            jsr display             ; display string
            jsr putnl_              ; skip to a new line
            swi

hello       fcc "hello"
            fcb 0
goodbye     fcc "goodbye"
            fcb 0
            end
```

Notes:

1. This program is equivalent to Example 7, except that the portion of the program which displayed strings is now in a separate subroutine called display.

2. The subroutine display must be mentioned in an xref command.

3. A new system routine, initstd_, has been introduced which should be used to open the screen and keyboard as files. It is good programming practice to use initstd_ even though this particular system opens these two files for its own use anyway. That is why it was not necessary to call initstd_ in previous programs.

ex8s.asm

```
;example 8 subroutine
          xdef  display          ; definition for external use
          xref  putchar_         ; reference to system routine

display   tfr  d,x               ; transfer D to X
          loop                   ; loop
             ldb ,x+             ;    load character into B
             quifeq              ;    quit if null byte
             pshsx               ;    save X
             jsr putchar_        ;    display character
             pulsx               ;    restore X
          endloop                ; endloop
          rts
          end
```

4. In order for the routine display to be accessed as an external reference by the mainline program, it has to be declared as an xdef in the subroutine. xdef stands for external definition.

5. Subroutines must have an rts command to return control to the calling program. Execution continues with the instruction following the jsr command which called the subroutine.

Creating The Linker File

ex8.cmd

```
"ex8"
org $1000
include "disk/1.watlib.exp"
"ex8m.b09"
"ex8s.b09"
```

Notes:

1. Both "ex8m.b09" and "ex8s.b09" must be included in the linker command file so that they will be included in the "ex8" load module.

EXAMPLE 9

The following program demonstrates passing a parameter to a subroutine on the stack instead of in the accumulator.

Creating The Program

ex9m.asm

```
;example 9 mainline routine
          xref initstd_          ; references to
          xref getchar_          ; system routines
          xref putnl_
          xref display           ; reference to user routine

          jsr initstd_           ; initialize standard I/O
          jsr getchar_           ; read character
          cmbp #'h
          if eq                  ; if equal to 'h'
             ldx #hello          ;    load address of "hello"
          else                   ; else
             ldx #goodbye        ;    load address of "goodbye"
          endif                  ; endif
          pshs x                 ; push address onto S
          jsr display            ; display string
          jsr putnl_             ; skip to a new line
          swi

hello     fcc "hello"
          fcb 0
goodbye   fcc "goodbye"
          fcb 0
          end
```

Notes:

1. This program is equivalent to that in example 8.

2. The address of the string to be displayed is loaded into index register X and is then pushed onto the stack.

ex9s.asm

```
;example 9 subroutine
          xdef  display          ; definition for external use
          xref putchar_          ; reference to system routine

display   puls  y                ; pull return address into Y
          puls  x                ; pull string address into X
          pshs  y                ; push return back onto S
          loop                   ; loop
             ldb,x+              ;    load character into B
             quifeq              ;    quit if null byte
             pshsx               ;    save X
             jsrputchar_         ;    display character
             pulsx               ;    restore X
          endloop                ; endloop
          rts
          end
```

Notes:

1. When a jsr is executed, the return address is stored on the top of the stack. This address is removed from the stack by the rts command.

2. The 'puls y' instruction pulls the return address from the stack into index register Y. 'puls x' pulls the address of the string to be displayed from the stack into index register X. Then the return address is pushed back onto the stack by 'pshs y'.

EXAMPLE 10

The following program uses 'lds' to set up its own stack.

Creating The Program

ex10m.asm

```
;example  10 mainline routine
          xref initstd_          ; references to
          xref getchar_          ; system routines
          xref putnl_
          xref display           ; reference to user routine

          lds #$0fff             ; initialize S pointer
          jsr initstd_           ; initialize standard I/O
          jsr getchar_           ; read character
          cmpb #'h
          if eq                  ; if equal to 'h'
             ldx #hello          ;    load address of "hello"
          else                   ; else
             ldx #goodbye        ;    load address of "goodbye"
          endif                  ; endif
          pshs x                 ; push address onto S
          jsr display            ; display string
          jsr putnl_             ; skip to a new line
          swi

hello     fcc "hello"
          fcb 0
goodbye   fcc "goodbye"
          fcb 0
          end
```

Notes:

1. 'lds #$0fff' starts the stack at $0fff instead of using the system stack which is only about 40 bytes long. Use of the system stack is dangerous since it could easily overflow and destroy system code.

2. The display routine is the same as the display routine in Example 9.

3. The 6809 can have two stacks by using the S register and the U register as stack pointers. The U register is manipulated in the same way as the S register. The U register is also used in bank-switching which is discussed in some later examples.

EXAMPLE 11

The following program employs another method of passing a parameter on the stack.

Creating The Program

ex11m.asm

```
;example 11 mainline routine
            xref initstd_           ; references to
            xref getchar_           ; system routines
            xref putnl_
            xref display            ; reference to user routine

            lds #$0fff              ; initialize S pointer
            jsr initstd_            ; initialize standard I/O
            jsr getchar_            ; read character
            cmpb #'h
            if eq                   ; if equal to 'h'
               ldx #hello           ;    load address of "hello"
            else                    ; else
               ldx #goodbye         ;    load address of "goodbye"
            endif                   ; endif
            pshs x                  ; push address onto S
            jsr display             ; display string
            leas 2,s                ; remove address from S
            jsr putnl_              ; skip to a new line
            swi

hello       fcc "hello"
            fcb 0
goodbye     fcc "goodbye"
            fcb 0
            end
```

Notes:

1. 'leas 2,s' adds 2 to the stack pointer for stack S. The contents of index register X are pushed onto the stack before display is called. After display, it is good programming practice to remove the stored value from the stack. The 'leas 2,s' instruction achieves this by changing the stack pointer. The value is actually still there in a physical sense, but, logically, it has been deleted since it is no longer considered part of the stack.

ex11s.asm

```
;example 11 subroutine
            xdef  display          ; definition for external use
            xref  putchar_         ; reference to system routine

display     ldx 2,s                ; pull address into X
            loop                   ; loop
               ldb ,x+             ;    load character into B
               quifeq              ;    quit if null byte
               pshsx               ;    save X
               jsr putchar_        ;    display character
               pulsx               ;    restore X
            endloop                ; endloop
            rts
            end
```

2. 'ldx 2,s' allows the parameter being passed to be accessed without having to pull the return address from the stack first and push it back on the stack afterwards.

EXAMPLE 12

In the following program, the subroutine, instead of the mainline routine, removes the parameter from the stack.

Creating The Program

ex12m.asm

```
;example 12 mainline routine
            xref initstd_           ; references to
            xref getchar_           ; system routines
            xref putnl_
            xref display            ; reference to user routine

            lds #$0fff              ; initialize S pointer
            jsr initstd_            ; initialize standard I/O
            jsr getchar_            ; read character
            cmpb #'h
            if eq                   ; if equal to 'h'
               ldx #hello           ;    load address of "hello"
            else                    ; else
               ldx #goodbye         ;    load address of "goodbye"
            endif                   ; endif
            pshs x                  ; push address onto S
            jsr display             ; display string
            jsr putnl_              ; skip to a new line
            swi

hello       fcc "hello"
            fcb 0
goodbye     fcc "goodbye"
            fcb 0
            end
```

ex12s.asm

```
;example 12 subroutine
            xdef  display          ; definition for external use
            xref putchar_          ; reference to system routine

display     ldx  2,s               ; pull address into X
            loop                   ; loop
               ldb,x+              ;    load character into B
               quifeq              ;    quit if null byte
               pshsx               ;    save X
               jsrputchar_         ;    display character
               pulsx               ;    restore X
            endloop                ; endloop
            ldd  ,s++              ; pull return address from S
            std  ,s                ; push in place of parameter
            rts
            end
```

Notes:

1. 'ldd ,s++' loads the return address into accumulator D and adds 2 to the stack pointer for stack S. The stack pointer now points to where the parameter is located so that, when the 'std ,s' instruction is executed, the parameter is deleted since the return address is stored in its place.

EXAMPLE 13

The following version of the subroutine display introduces the jmp[] indirect command. (The square brackets indicate indirection.)

Creating The Program

ex13s.asm

```
;example 13 subroutine
            xdef  display          ; definition for external use
            xref putchar_          ; reference to system routine

display     ldx   2,s              ; pull address into X
            loop                   ; loop
               ldb,x+              ;    load character into B
               quifeq              ;    quit if null byte
               pshsx               ;    save X
               jsrputchar_         ;    display character
               pulsx               ;    restore X
            endloop                ; endloop
            leas  4,s              ; remove return address & parm
            jmp  [-4,s]            ; jump indirect through return
            end
```

Notes:

1. 'leas 4,s' removes the parameters from the stack by moving the stack pointer and 'jmp [-4,s]' causes a jump indirect through the return address on the stack. This achieves a return from the subroutine without using the RTS instruction.

   - Using jump indirect is not a recommended method of returning from a subroutine. It is shown here simply as an illustration of jump indirect.

EXAMPLE 14

The SuperPET had 96K of RAM memory but only 32K is directly addressable. The remaining 64K is divided up into 16 blocks of 4K each. Each 4K block is called a 'bank' and can be accessed by hardware and software 'bank-switching'. Example 14 illustrates the implementation of bank-switching in assembly language by placing a subroutine in each of bank 1, bank 4, and bank 7.

Creating The Program

ex14m.asm

```
;example 14 test of bank-switching
        ;mainline routine
        xref  rtn1              ; references to
        xref  rtn4              ; definition for external uses
        xref rtn7
        xref  bankinit_         ; references to
        xref  initstd_          ; system routines
        xref  putnl_

        lds  #$0fff             ; initialize S pointer
        jsr  bankinit_          ; initialize bank-switching
        jsr  initstd_           ; initialize standard I/O
        jsr  rtn1               ; display '1'
        jsr  putnl_             ; skip to a new line
        jsr  rtn4               ; display '4'
        jsr  putnl_             ; skip to a new line
        jsr  rtn7               ; display '7'
        jsr  putnl_             ; skip to a new line
        swi
        end
```

Notes:

1. This program and the following subroutines would work just as well if the subroutines were placed in main memory with the mainline routine. There is nothing special about the routines themselves.

2. In order for bank-switching to occur, it is necessary for the system routine bankinit_ to be xref'ed and called at the beginning of the program. bankinit_ sets up a program in main memory so that it is possible to jump from main memory to a bank and back again.

3. The U register is used to manage a stack for bank-switching and is not normally accessible to the user program if bank-switching is being used.

ex14rtn1.asm

```
;example 14 subroutine rtn1 which
;              operates from bank 1
              xdef rtn1                ; definition for external use
              xref putchar_            ; reference to system routine


rtn1          ldb #'1                  ; load '1' into B
              jsr putchar_             ; display character
              rts
              end
```

ex14rtn4.asm

```
;example 14 subroutine rtn4 which
;              operates from bank 4
              xdef rtn4                ; definition for external use
              xref putchar_            ; reference to system routine

rtn4          ldb #'4                  ; load '4' into B
              jsr putchar_             ; display character
              rts
              end
```

ex14rtn7.asm

```
;example 14 subroutine rtn7 which
;           operates from bank 7
            xdef rtn7               ; definition for external use
            xref putchar_           ; reference to system routine

rtn7        ldb #'7                 ; load '7' into B
            jsr putchar_            ; display character
            rts
            end
```

Creating The Linker File

ex14.cmd

```
            "ex14"
            org $1000
            include "disk/1.watlib.exp"
            "ex14m.b09"
            bankorg $9000           ;optional
            banksize $1000          ;optional
            bank 1
            "ex14rtn1.b09"
            bank 4
            "ex14rtn4.b09"
            bank 7
            "ex14rtn7.b09"
```

Notes:

1. The first four lines of the '.cmd' file are the same as usual and cause the mainline routine to be loaded at address $1000 in main memory.

2. The 'bankorg $9000' directive is optional since $9000 is the default value for the origin of the bank addresses. Only one bank can be accessed at a time.

- $9000 to $9fff acts as a 4K window which allows one bank to be seen while the rest remain hidden. If a different bank is needed, the address of the current bank is switched out and a new one switched in. This change of addressing is referred to as 'bank-switching'.

3. The 'banksize $1000' directive is optional as well since $1000 (4K) is the default value for the size of the bank.

4. 'bank 1' causes 'ex14rtn1.b09" to be placed in bank 1. 'bank 4' causes "ex14rtn4.b09" to be placed in bank 4. 'bank 7' causes "ex14rtn7.b09" to be placed in bank 7.

   - More than one module could be placed in a single bank simply by specifying more than one module name after the bank directive. The user must be aware of the total size of all routines so that the routines plus the special bank-switching code do not exceed 4K bytes for any bank.

5. The '.cmd' file may contain comments which are preceded by semi-colons.

EXAMPLE 15

The following program shows how to pass data between programs stored in different banks. The data is normally passed using the main memory.

Creating The Program

ex15m.asm

```
;example 15 test of bank-switching
;              with data in a bank
;mainline routine
            xdef  message1          ; external use of string
            xref  rtn1              ; references to
            xref  rtn4              ; definition for external uses
            xref rtn7
            xref bankinit_          ; references to
            xref  initstd_          ; system routines
            xref putnl_

            lds #$0fff              ; initialize S pointer
            jsr bankinit_           ; initialize bank-switching
            jsr initstd_            ; initialize standard I/O
            jsr rtn1                ; display "hello ... 1"
            jsr putnl_              ; skip to a new line
            jsr rtn4                ; display "hello ... 4"
            jsr putnl_              ; skip to a new line
            jsr rtn7                ; display "hello ... 7"
            jsr putnl_              ; skip to a new line
            swi

message1    rmb 17
            fcb 0
            end
```

Notes:

1. Data is passed from one bank to another by passing it from one bank to main memory and then onto the other bank.

2. The mainline routine which will be loaded into main memory calls subroutines rtn1, rtn4, and rtn7 which will be loaded into bank 1, bank 4, and bank 7 respectively.

3. Subroutines rtn1, rtn4, and rtn7 call subroutine getmessage which will be loaded into bank 10.

4. Subroutines rtn1, rtn4, and rtn7 wish to have access to the character string stored beginning at the location labelled "message" in subroutine "getmessage". They cannot access message directly because each bank occupies the same address space and so only one bank is directly accessible at any moment in time.

5. The message1 label is xdef'ed in the mainline program so that it can be xref'ed by the subroutines, including subroutine getmessage.

6. Subroutine getmessage copies the string of characters from the location message in routine getmessage to the location message1 in the mainline program.

7. Subroutines rtn1, rtn4, and rtn7 access the string of characters located at message1 in main memory.

ex15getm.asm

```
;example 15 subroutine to handle
;               data in bank switched
;               memory
                xdef getmessage         ; definition for external use
                xref message1           ; reference to system label

getmessage      lda 2,s                 ; load character into A
                sta number              ; store character
                ldx #message            ; load address of message
                ldy #message1           ; load address of message1
                loop                    ; transfer string
                   lda,x+               ;   "hello from bank x"
                   quifeq               ;   from message to
                   sta,y+               ;   message1 in the
                endloop                 ; mainline routine
                ldd ,s+                 ; load return address &
                std ,s                  ; store in place of parameter
                rts

message         fcc "hello from bank "
number          fcc "x"
                fcb 0
                end
```

Notes:

1. The getmessage module stores a character whose address is passed as a parameter on the stack at location "number".

2. getmessage copies the 18 characters beginning at location message and ending at location number to the location labelled message1 in the mainline program. (message1 is xdef'ed.)

ex15rtn1.asm

```
;example 15 subroutine rtn1 which
;              operates from bank 1
;              and sends a message from
;              another bank
            xdef rtn1              ; definition for external use
            xref getmessage        ; references to
            xref display           ; user routines
            xref message1          ; reference to string

rtn1        ldb #'1                ; load '1' into B
            pshs b                 ; push '1' onto S
            jsr getmessage         ; transfer message
            ldx #message1          ; load address of message1
            pshs x                 ; push address onto S
            jsr display            ; display message
            rts
            end
```

Notes:

1. The character '1' is pushed onto the stack and passed as a parameter to subroutine getmessage.

2. After getmessage has copied its message to message1 in main memory, rtn1 pushes the address of message1 onto the stack and calls the display routine to put the message on the screen. The display routine is not shown in this example because it is the same routine as in Example 12.

ex15rtn4.asm

```
;example 15 subroutine rtn4 which
;              operates from bank 4
;              and sends a message from
;              another bank
              xdef rtn4              ; definition for external use
              xref getmessage        ; references to
              xref display           ; definition for external uses
              xref message1          ; reference to string

rtn4          ldb #'4                ; load '4' into B
              pshs b                 ; push B onto S
              jsr getmessage         ; transfer message
              ldx #message1          ; load address of message1
              pshs x                 ; push address onto S
              jsr display            ; display message
              rts
              end
```

Notes:

1. The character '4' is pushed onto the stack and passed as a parameter to subroutine getmessage.

2. getmessage and display are called as in rtn1.

ex15rtn7.asm

```
;example 15 subroutine rtn7 which
;              operates from bank 7
;              and sends a message from
;              another bank
            xdef rtn7              ; definition for external use
            xref getmessage        ; references to
            xref display           ; definition for external uses
            xref message1          ; reference to string

rtn7        ldb #'7                ; load '7' into B
            pshs b                 ; push '7' onto S
            jsr getmessage         ; transfer message
            ldx #message1          ; load address of message1
            pshs x                 ; push address onto S
            jsr display            ; display message
            rts
            end
```

Notes:

1. The character '7' is pushed onto the stack and passed as a parameter to subroutine getmessage.

2. getmessage and display are called as in rtn1.

Creating The Linker File

ex15.cmd

```
"ex15"
org $1000
include "disk/1.watlib.exp"
"ex15m.b09"
"ex12s.b09"
bankorg $9000          ;optional
banksize $1000         ;optional
bank 1
"ex15rtn1.b09"
bank 4
"ex15rtn4.b09"
bank 7
"ex15rtn7.b09"
bank 10
"ex15getm.b09"
```

NOTES:

1. The linker file is similar to the one in Example 14.

EXAMPLE 16

The following program uses a subroutine read and a subroutine display to read a line of characters from the keyboard and then redisplay it on the screen.

Creating The Program

ex16.asm

```
;example 16 mainline routine
            xref initstd_          ; references to
            xref  putnl_           ; system routines
            xref display           ; references to
            xref read              ; user routines

            lds #$0fff             ; initialize S pointer
            jsr initstd_           ; initialize standard I/O
            loop                   ; loop
                ldx#storage        ;   load address
                jsrread            ;   read string
                ldx#storage        ;   load address
                ldb,x              ;   load character into B
                quifeq             ;   quit if null string
                jsrdisplay         ;   display string
                jsrputnl_          ;   skip to a new line
            endloop                ; endloop
            swi

storage     rmb 41
            end
```

Notes:

1. The number of characters in the input line cannot be more than 40.

   - 41 bytes are reserved beginning at the location labelled 'storage'.

   - 1 byte is required to store the end-of-line character. (The read subroutine stores 0 to indicate the end of the character string.)

2. The address of 'storage' is placed in index register X and is passed to subroutines read and display.

   - The read subroutine will read characters from the keyboard and will store them beginning at the address in index register X. Characters will be read and stored until a carriage-return is entered. If more than 40 characters are entered, the extra characters will be stored in locations not intended for that purpose and this could corrupt the program.

   - The display subroutine will display characters stored beginning at the address in index register X. Characters will be displayed until the end-of-line character is reached.

3. After the line of characters has been displayed, putnl_ is called.

4. Lines are read and displayed until a null line is entered.

ex16sr.asm

```
;example 16 subroutine read
            xdef read               ; definition for external use
            xref getchar_           ; reference to system routine

read        loop                    ; loop
               pshsx                ;   save X
               jsrgetchar_          ;   read character
               pulsx                ;   restore X
               cmpb #$0d            ;   carriage return?
               quifeq               ;   if yes, then quit
               stb,x+               ;   store character
            endloop                 ; endloop
            lda #0                  ; 0 is null byte
            sta ,x                  ; store as end-of-string
            rts
            end
```

Notes:

1. The read subroutine gets and stores characters until the character entered is equivalent to $0d which is a carriage-return.

2. The $0d could be stored and used as the end-of-line character, but in order to be consistent with the convention introduced earlier, a 0 is stored instead.

ex16sd.asm

```
;example 16 subroutine display
            xdef  display           ; definition for external use
            xref putchar_           ; reference to system routine

display     loop                    ; loop
               ldb,x+               ;    load character into B
               quifeq               ;    quit if null byte
               pshsx                ;    save X
               jsr putchar_         ;    display character
               pulsx                ;    restore X
            endloop                 ; endloop
            rts
            end
```

EXAMPLE 17

Example 17 demonstrates how to create and use one's own personal library of subroutines. Two routines, read and display, have been created with object files called ex16sr.b09 and ex16sd.b09 respectively. Now, a personal library of useful routines is to be created which can be included in any load module by a single statement in the '.cmd' file.

Creating The Library File

ex17.lib

```
"ex16sr.b09"
"ex16sd.b09"
```

Notes:

1. The name of the library file can be anything. It does not have to end with the suffix '.lib'.

2. The library file must contain the quoted names of the '.b09' files for the library subroutines.

Creating The Linker File

ex17.cmd

```
"ex17"
org $1000
include "disk/1.watlib.exp"
"ex16m.b09"
include ex17.lib
```

Notes:

1. 'include ex17.lib' is effectively the same as saying
   "ex16sr.b09"
   "ex16sd.b09"
   which is what is contained in the '.cmd' file for Example 16. Thus, running the linker with 'ex17.cmd' will include the read and display subroutines.

EXAMPLE 18

Example 18 inputs from the keyboard and outputs to the printer.

Creating The Program

ex18m.asm

```
;example 18 mainline routine
            xref initstd_           ; references to
            xref openf_             ; system routines
            xref closef_
            xref putnl_
            xref fputnl_
            xref display            ; references to
            xref fdisplay           ; definition for external uses
            xref read

            lds #$0fff              ; initialize S pointer
            jsr initstd_            ; initialize standard I/O
            ldd #mode               ; load address of file mode
            pshs d                  ; push file mode onto S
            ldd #outfile            ; load address of filename
            jsr openf_              ; open file
            leas 2,s                ; remove file mode from S
            std outptr              ; file control block address
            if ne                   ; if file opened okay
                loop                ;   loop
                    ldx#storage     ;     load address
                    jsrread         ;     read string
                    ldx#storage     ;     load address
                    ldb,x           ;     load character into B
                    quifeq          ;     quit if null string
                    lddoutptr       ;     load file control block
                    jsrfdisplay     ;     display string
                    lddoutptr       ;     load file control block
                    jsrfputnl_      ;     skip to a new line
                endloop             ;   endloop
                lddoutptr           ;   load file control block
                jsrclosef_          ;   close file
```

```
            else                ; else
               ldx#errmsg       ;   load address of errmsg
               jsrdisplay       ;   display string
               jsrputnl_        ;   skip to a new line
            endif               ; endif
            swi

outfile     fcc "printer"
            fcb 0
mode        fcc "w"
            fcb 0
outptr      rmb 2
storage     rmb 41
errmsg      fcc "open error"
            fcb 0
            end
```

Notes:

1. The printer is treated as a file. A file must be opened before it is used and closed afterwards. For this purpose, the system routines openf_ and closef_ are provided.

2. fputnl_ is the same as putnl_ except that it allows a particular file to be specified. putnl_ uses the standard output device which initstd_ defines as the screen.

3. fdisplay is a new user routine which works like display except that it outputs to a specified file instead of to the standard output device.

4. Before openf_ is called, the address of the access mode for the file must be pushed on the stack and the address of the file's name must be placed in the accumulator.

5. A pointer to the file control block is returned in accumulator D if the file is opened properly; otherwise, zero is returned.

6. If the printer file is not opened, the error message "open error" is displayed on the screen using display and putnl_.

7. If the printer file is opened, read is used to input a string of characters from the keyboard and fdisplay is called to display the characters on the printer.

8. Lines are read and displayed until a null line is entered.

9. This program uses the subroutine "read" from Example 16.

ex18sf.asm

```
;example 18 subroutine fdisplay
            xdef fdisplay          ; definition for external use
            xref fputchar_         ; reference to system routine

fdisplay    std outptr
            loop                   ; loop
               ldb,x+              ;   load character into B
               quifeq              ;   quit if null byte
               pshsx               ;   save X
               pshsd               ;   push character onto S
               lddoutptr           ;   load file control block
               jsrfputchar_        ;   display character
               leas2,s             ;   remove parameter from S
               pulsx               ;   restore X
            endloop                ; endloop
            rts

outptr      rmb 2
            end
```

Notes:

1. fdisplay is passed the pointer to the printer file in accumulator D.

2. fdisplay passes this pointer to the system routine fputchar_.

Creating The Library File

ex.lib

```
"ex16sr.b09"
"ex16sd.b09"
"ex18sf.b09"
```

Creating The Linker File

ex18.cmd

```
"ex18"
org $1000
include "disk/1.watlib.exp"
"ex18m.b09"
include "ex.lib"
```

Notes:

1. 'include "ex.lib"' is effectively the same as saying

```
"ex16sr.b09"
"ex16sd.b09"
"ex18sf.b09".
```

- The object modules "ex18m.b09", "ex16sr.b09", "ex16sd.b09", and "ex18sf.b09" will all be linked into the "ex18" load module.

EXAMPLE 19

Example 19 inputs from the keyboard and outputs to a diskette file.

Creating The Program

ex19m.asm

```
;example 19 mainline routine
            xref initstd_          ; references to
            xref openf_            ; system routines
            xref closef_
            xref putnl_
            xref fputnl_
            xref display           ; references to
            xref fdisplay          ; definition for external uses
            xref read

            lds #$0fff             ; initialize S pointer
            jsr initstd_           ; initialize standard I/O
            ldd #mode              ; load address of file mode
            pshs d                 ; push file mode onto S
            ldd #outfile           ; load address of filename
            jsr openf_             ; open file
            leas 2,s               ; remove file mode from S
            std outptr             ; file control block address
            if ne                  ; if file opened okay
               loop                ;   loop
                  ldx#storage      ;   load address
                  jsrread          ;   read string
                  ldx#storage      ;   load address
                  ldb,x            ;   load character into B
                  quifeq           ;   quit if null string
                  lddoutptr        ;   load file control block
                  jsrfdisplay      ;   write string
                  lddoutptr        ;   load file control block
                  jsrfputnl_       ;   skip to a new line
               endloop             ;   endloop
               lddoutptr           ;   load file control block
               jsrclosef_          ;   close file
```

```
            else                ; else
                ldx#errmsg      ;   load address of errmsg
                jsrdisplay      ;   display string
                jsrputnl_       ;   skip to a new line
            endif               ; endif
            swi

outfile     fcc "tempfile"
            fcb 0
mode        fcc "w"
            fcb 0
outptr      rmb 2
storage     rmb 41
errmsg      fcc "open error"
            fcb 0
            end
```

Notes:

1. This program is identical to that in example 18 except that the name of the output file is different. Instead of displaying the string of characters on the printer, it is to be written to a diskette file called "tempfile". (All filenames are assumed to refer to diskette files if no device is specified.)

   - "tempfile" may be examined using the editor.

2. If "tempfile" is to be placed on the diskette in disk drive 1 then the character string "disk/1.tempfile" should be assigned to outfile. By default, "tempfile" refers to disk drive 0.

4. The program uses the subroutine "read" from Example 16.

EXAMPLE 20

The following program uses the read routine to read two different strings into memory and then uses the system routine streq_ (string equality) to compare the two strings.

Creating The Program

ex20m.asm

```
;example 20 mainline routine
            xref initstd_           ; references to
            xref putnl_             ; system routines
            xref streq_
            xref display            ; references to
            xref read               ; definition for external uses

            lds #$0fff              ; initialize S pointer
            jsr initstd_            ; initialize standard I/O
            ldx #string1            ; load address1
            jsr read                ; read string
            ldx #string2            ; load address2
            jsr read                ; read string
            ldd #string2            ; load address2
            pshs d                  ; push address onto S
            ldd #string1            ; load address1
            jsr streq_              ; compare two strings
            leas 2,s                ; remove string2 address from S
            if ne                   ; if strings are equal
               ldx#true             ;    load address of "TRUE"
            else                    ; else
               ldx#false            ;    load address of "FALSE"
            endif                   ; endif
            jsr display             ; display "TRUE" or "FALSE"
            jsr putnl_              ; skip to a new line
            swi

string1     rmb 41
string2     rmb 41
true        fcc "TRUE"
```

```
            fcb 0
false       fcc "FALSE"
            fcb 0
            end
```

Notes:

1. The system routine streq_ compares two strings for equality. Before streq_ is called, the address of one string must be placed on the stack and the address of the other string must be placed in accumulator D.

2. If the two strings are equal, streq_ returns a non-zero value in accumulator D. If they are not equal, zero is returned.

   - If the two strings are equal then the word "TRUE" is displayed on the screen using the display subroutine; otherwise, the word "FALSE" is displayed.

3. The program uses the subroutine "read" from Example 16.

EXAMPLE 21

The following program uses the read routine to read two different strings into memory. The system routine length_ is used to get the length of each string and, if the lengths are equal, the system routine equal_ is called to compare the two strings.

Creating The Program

ex21m.asm

```
;example 21 mainline routine
            xref initstd_          ; references to
            xref putnl_            ; system routines
            xref length_
            xref equal_
            xref display           ; references to
            xref read              ; user routines

            lds #$0fff             ; initialize S pointer
            jsr initstd_           ; initialize standard I/O
            ldx #string1           ; load address1
            jsr read               ; read string
            ldd #string1           ; load address1
            jsr length_            ; get length of string1
            std length1            ; store length at length1
            ldx #string2           ; load address2
            jsr read               ; read string
            ldd #string2           ; load address2
            jsr length_            ; get length of string2
            cmpd length1           ; compare lengths
            if eq                  ; if lengths are equal
               pshsd               ;   push length onto S
               ldd#string1         ;   load address1
               pshsd               ;   push address onto S
               ldd#string2         ;   load address2
               jsrequal_           ;   compare strings
               leas4,s             ;   remove parameters from S
               if ne               ;   if strings are equal
                  ldx#true         ;      load address of "TRUE"
               else                ;   else
                  ldx#false        ;      load address of "FALSE"
```

```
            endif               ;   endif
          else                  ; else
            ldx#false           ;   load address of "FALSE"
          endif                 ; endif
          jsr display           ; display "TRUE" or "FALSE"
          jsr putnl_            ; skip to a new line
          swi

string1   rmb 41
string2   rmb 41
true      fcc "TRUE"
          fcb 0
false     fcc "FALSE"
          fcb 0
length1   rmb 2
          end
```

Notes:

1. Given the address of a string in accumulator D, length_ returns the number of characters in the string. The result is returned in accumulator D.

2. equal_ compares 'n' characters of memory for equality. Before equal_ is called, 'n' must be pushed on the stack followed by the address of one string with the address of the other string being placed in accumulator D.

3. If the first 'n' characters of the two strings are equivalent, equal_ returns a non-zero value in accumulator D; otherwise, zero is returned.

4. In the above program, the two strings are checked for exact equality. If their lengths are not equal, it is assumed that the strings are not equal. If the lengths are equal, equal_ is called.

   - If the two strings are equal then the word "TRUE" is displayed on the screen using the display subroutine.

   - If the two strings are not equal then the word "FALSE" is displayed.

5. The program illustrates how control constructs can be nested. One if/else/endif construct is nested within another.

EXAMPLE 22

The following program uses a subroutine called bdisplay to display a binary integer on the screen.

Creating The Program

ex22m.asm

```
;example 22 mainline routine
            xref initstd_          ; references to
            xref  putnl_           ; system routines
            xref bdisplay          ; reference to user routine

            lds #$0fff             ; initialize S pointer
            jsr initstd_           ; initialize standard I/O
            ldb number             ; load 25 into B
            jsr bdisplay           ; display 25 in binary
            jsr putnl_             ; skip to a new line
            swi

number      fcb 25
            end
```

Notes:

1. The number 25 is loaded into accumulator B and then bdisplay is called to display 25 as a binary number on the screen.

   - The number displayed will be 00011001.

   - bdisplay will only work for numbers which can be represented in 8 bits.

ex22sb.asm

```
;example 22 subroutine bdisplay
            xdef  bdisplay          ; definition for external use
            xref display            ; reference to user routine

bdisplay    lda #8                  ; load 8 into A
            loop                    ; loop
               aslb                 ;   shift B left
               ifcc                 ;   if carry clear
                  ldx#char0         ;     load address of "0"
               else                 ;   else
                  ldx#char1         ;     load address of "1"
               endif                ;   endif
               pshsd                ;   save D
               jsrdisplay           ;   display "0" or "1"
               pulsd                ;   restore D
               deca                 ;   decrement A by 1
               quifeq               ;   quit if A equals 0
            endloop                 ; endloop
            rts

char0       fcc "0"
            fcb 0
char1       fcc "1"
            fcb 0
            end
```

Notes:

1. 8 bits are to be displayed and so the loop must be executed 8 times. The number 8 is loaded into accumulator A and every time the loop is completed, 1 is subtracted from accumulator A. When accumulator A contains 0, the loop ends.

2. Accumulator B contains the number to be displayed. At the beginning of the loop, an arithmetic shift left is performed on accumulator B. This will shift the leftmost bit of the B accumulator into the carry bit.

   - If the carry flag is set then a 1 was shifted into the carry bit and so the character "1" is displayed on the screen.

- If the carry flag is clear then a 0 was shifted into the carry bit and so the character "0" is displayed on the screen.

EXAMPLE 23

The following program uses a subroutine called hdisplay to display an integer on the screen in hexadecimal.

Creating The Program

ex23m.asm

```
;example 23 mainline routine
            xref initstd_           ; references to
            xref  putnl_            ; system routines
            xref hdisplay           ; reference to user routine

            lds #$0fff              ; initialize S pointer
            jsr initstd_            ; initialize standard I/O
            ldb number              ; load 25 into B
            jsr hdisplay            ; display 25 in hexadecimal
            jsr putnl_              ; skip to a new line
            swi

number      fcb 25
            end
```

Notes:

1. The number 25 is loaded into accumulator B and then hdisplay is called to display 25 as a hexadecimal number on the screen.

   - The number displayed will be 19.

   - hdisplay will only work for numbers which can be represented in 8 bits.

ex23sh.asm

```
;example 23 subroutine hdisplay
            xdef  hdisplay          ; definition for external use
            xref putchar_           ; reference to system routine

hdisplay    clra                    ; clear A
            pshs  d                 ; save D
            lsrb                    ; shift B right 4 times
            lsrb                    ; in order to have D
            lsrb                    ; contain value of
            lsrb                    ; first hexadecimal digit
            tfr  d,x                ; transfer D to X
            ldb  hchars,x           ; load hex digit into B
            jsr putchar_            ; display first hex char
            puls  d                 ; restore D
            andb #$0f               ; value of 2nd hex digit
            tfr  d,x                ; transfer D to X
            ldb  hchars,x           ; load hex digit into B
            jsr putchar_            ; display second hex char
            rts

hchars      fcc "0123456789ABCDEF"
            end
```

Notes:

1. The value to be displayed is in accumulator B when hdisplay is called.

   - Accumulator A is cleared so that accumulator D is equivalent to accumulator B.

   - Accumulator D is pushed on the stack to save it.

   - Accumulator B is logically right shifted four times so that accumulator D contains a number between 0 and 15 which is to be represented as the first of two hexadecimal digits.

2. Accumulator D is transferred to index register X. The contents of index register X act as an offset from hchars so that the appropriate hexadecimal character is loaded into accumulator B.

- putchar_ is called to display the character on the screen.

3. The value of accumulator D saved on the stack earlier is restored.

   - Accumulator B is logically ANDed with $0F so that accumulator D contains a number between 0 and 15 which is to be represented as the second hexadecimal digit.

4. Accumulator D is transferred to index register X. The contents of index register X act as an offset from hchars so that the appropriate hexadecimal character is loaded into accumulator B.

   - putchar_ is called to display the second character on the screen.

EXAMPLE 24

The following program demonstrates how to use the system routine printf_ which is a facility to provide formatted output to the screen.

Creating The Program

ex24m.asm

```
;example 24 mainline routine
            xref initstd_           ; references to
            xref printf_            ; system routines

            jsr initstd_            ; initialize standard I/O
            ldb char1               ; load 'a' into B
            pshs d                  ; push 'a' onto S
            ldd #string1            ; load address1
            jsr printf_             ; display string1 formatted
            ldd number              ; load 25 into D
            pshs d                  ; push 25 onto S
            ldd #string2            ; load address2
            jsr printf_             ; display string2 formatted
            ldd #string             ; load address
            pshs d                  ; push address onto S
            ldd number              ; load 25 into D
            pshs d                  ; push 25 onto S
            ldd #string3            ; load address3
            jsr printf_             ; display string3 formatted
            swi

string1     fcc "printf_ can be used to display characters"
            fcc " such as %c.%n"
            fcb 0
string2     fcc "it can also be used to display decimal "
            fcc "numbers such as %d.%n"
            fcb 0
string3     fcc "or hexadecimal numbers such as %h or "
            fcc "strings such as %s.%n"
            fcb 0
char1       fcb 'a                  ; substitution value for %c in string1
```

```
number  fdb 25                  ; substitution value for %d in string2
                                ; and %h in string3
string  fcc "'This is a string.'" ; substitution value
                                        ; for %s in string3
        fcb 0
        end
```

Notes:

1. For printf_, a string to be displayed is to be specified. This string may contain the following escape characters for special formatting:

   %n - insert the new line character
   %c - insert a character
   %d - insert a decimal number
   %h - insert a hexadecimal number
   %s - insert a string

2. Up to 6 substitution values may be specified, not counting new line characters which do not need to be specified.

   - The substitution value for %c is a character.

   - The substitution values for %d and %h are a decimal number one word in length and a hexadecimal number one byte in length respectively.

   - The substitution value for %s is the address of a string.

3. The substitution values are to be pushed onto the stack with the last substitution value being pushed on first.

4. The address of the string to be displayed is to be placed in accumulator D.

   - Then, printf_ may be called.

EXAMPLE 25

The following program demonstrates how to use macros. The macro "prints" displays a string on the screen, preceded by forty blank spaces if accumulator B contains 'r'.

Creating The Program

ex25m.asm

```
; example 25 mainline routine
            xref initstd_           ; references to
            xref putchar_           ; system routines
            xref putnl_
            xref itos_
            xref display            ; reference to user routine

prints      macr
            cmpb  #'r               ; compare B to 'r'
            if eq                   ; if B equals 'r'
               ldy#40               ;   load Y with 40
               pshsx                ;   save X
               loop                 ;   loop
                  ldb#' '           ;     load ' ' into B
                  pshsy             ;     save Y
                  jsrputchar_       ;     display blank
                  pulsy             ;     restore Y
                  leay-1,y          ;     decrement Y by 1
               untileq              ;   quit if Y equals 0
               pulsx                ;   restore X
            endif                   ; endif
            jsr display             ; display string
            endm                    ; end of macro "prints"

            jsr initstd_            ; initialize standard I/O
            jsr putnl_              ; skip to a new line
            ldx  #leftstr           ; load address of leftstr
            ldb  #'l                ; on left side
            prints                  ; display "ODD"
            ldx  #rightstr          ; load address of rightstr
```

```
            ldb  #'r            ; on right side
            prints              ; display "EVEN"
            jsr putnl_          ; skip to a new line
            swi

leftstr     fcc "ODD"
            fcb 0
rightstr    fcb 8
            fcb 8
            fcb 8
            fcc "EVEN"
            fcb 0
            end
```

Notes:

1. This program leaves a blank line, displays the heading "ODD" ... EVEN" with "ODD" beginning at the left side of the screen and "EVEN" beginning in the middle of the screen.

2. prints is a macro which displays a string whose address is in index register X. If accumulator B contains the character 'r', the string is preceded by 40 blank characters.

   - A macro begins with a label which is the macro name followed by the assembler directive 'macr'. In this case, the macro name is 'prints'.

   - A macro ends with the assembler directive 'endm'.

   - A macro is called by using the macro name as an instruction. Wherever the macro is called, the assembler replaces the macro call with the body of the macro. Thus, the body of the macro occurs only once in the assembly language source code but, in the object code, it occurs as many times as the macro is called.

   - A macro name must not be the same as an assembly instruction or assembler directive. If it is, the macro will be ignored.

3. The heading is created by displaying "ODD", displaying 40 blanks, displaying 3 backspaces, and displaying "EVEN".

- $08 is a backspace in ASCII.

Assembling The Program

ex25m.lst

```
0000                        ;example 25
0000                                xref initstd_
0000                                xref putchar_
0000                                xref putnl_
0000                                xref itos_
0000                                xref display
0000
0000                        prints  macr
0000                                cmpb #'r
0000                                if eq
0000                                    ldy #40
0000                                    pshs x
0000                                    loop
0000                                        ldb #'
0000                                        pshs y
0000                                        jsr putchar_
0000                                        puls y
0000                                        leay -1,y
0000                                    until eq
0000                                    puls x
0000                                endif
0000                                jsr display
0000                                endm
0000
0000 BD 00 00                       jsr initstd_
0003 BD 00 00                       jsr putnl_
0006 8E 00 BA                       ldx #leftstr
0009 C6 6C                          ldb #'l
000B                                prints
000B C1 72                  +        cmpb #'r
000D 26 15                  +        if eq
000F 10 8E 00 28            +            ldy #40
0013 34 10                  +            pshs x
```

```
0015                        +           loop
0015 C6  20                 +               ldb #'
0017 34  20                 +               pshs y
0019 BD  00  00             +               jsr putchar_
001C 35  20                 +               puls y
001E 31  3F                 +               leay -1,y
0020 26  F3                 +           until eq
0022 35  10                 +           puls x
0024                        +         endif
0024 BD  00  00             +         jsr display
0027 8E  00  BE                     ldx #rightstr
002A C6  72                         ldb #'r
002C                                prints
002C C1  72                 +         cmpb #'r
002E 26  15                 +         if eq
0030 10  8E  00  28         +           ldy #40
0034 34  10                 +           pshs x
0036                        +           loop
0036 C6  20                 +               ldb #'
0038 34  20                 +               pshs y
003A BD  00  00             +               jsr putchar_
003D 35  20                 +               puls y
003F 31  3F                 +               leay -1,y
0041 26  F3                 +           until eq
0043 35  10                 +           puls x
0045                        +         endif
0045 BD  00  00             +         jsr display
0048 BD  00  00                     jsr putnl_
004B 3F                             swi
004B
004C 4F  44  44             leftstr fcc "ODD"
004F 00                             fcb 0
0050 08                     rightstr fcb 8
0051 08                             fcb 8
0052 08                             fcb 8
0053 45  56  45  4E                 fcc "EVEN"
0057 00                             fcb 0
0058                                end
0058
```

Notes:

1. The macro definition does not generate any object code.

2. The macro calls are all replaced by the body of the macro and object code for the macro body is generated each time there is a macro call.

   - In this case, the macro 'prints' is called two times. The insertion of a macro body is indicated by plus signs.

EXAMPLE 26

The following program provides an example of conditional assembly directives. The program assembles one set of instructions for an eighty-column screen microcomputer and a different set of instructions for a forty-column screen microcomputer. (The user is responsible for which set of instructions is assembled.)

Creating The Program

ex26m.asm

```
; example 26 mainline routine
screen    equ 80
          xref initstd_          ; references to
          xref putchar_          ; system routines
          xref putnl_
          xref itos_
          xref display           ; reference to user routine

prints    macr
          cmpb  #'r              ; compare B to 'r'
          if eq                  ; if B equals 'r'
             ifeq%0-80           ;   if %0 equals 80
                ldy#40           ;     load Y with 40
             endc                ;   end of cond. if
             ifeq%0-40           ;   if %0 equals 40
                ldy#20           ;     load Y with 20
             endc                ;   end of cond. if
             pshsx               ;   save X
             loop                ;   loop
                ldb#' '          ;     load ' ' into B
                pshsy            ;     save Y
                jsrputchar_      ;     display blank
                pulsy            ;     restore Y
                leay-1,y         ;     decrement Y by 1
             untileq             ;   quit if Y equals 0
             pulsx               ;   restore X
          endif                  ; endif
          jsr display            ; display string
          endm                   ; end of macro "prints"
```

```
            jsr initstd_          ; initialize standard I/O
            jsr putnl_            ; skip to a new line
            ifeq (screen - 80)    ; if screen equals 80
               ldx#scr80          ;    load address of scr80
            endc                  ; end of cond. if
            ifeq (screen - 40)    ; if screen equals 40
               ldx#scr40          ;    load address of scr40
            endc                  ; end of cond. if
            ldb  #'l              ; on left side
            prints screen         ; display scr80 or scr40
            jsr putnl_            ; skip to a new line
            jsr putnl_            ; skip to a new line
            ldx  #leftstr         ; load address of leftstr
            ldb  #'l              ; on left side
            prints screen         ; display "ODD"
            ldx  #rightstr        ; load address of rightstr
            ldb  #'r              ; on right side
            prints screen         ; display "EVEN"
            jsr putnl_            ; skip to a new line
            swi

scr80       fcc "This is an 80-column machine."
            fcb 0
scr40       fcc "This is a 40-column machine."
            fcb 0
leftstr     fcc "ODD"
            fcb 0
rightstr    fcb 8
            fcb 8
            fcb 8
            fcc "EVEN"
            fcb 0
            end
```

Notes:

1. This program is the same as that in Example 25 with a few small additions.

   - Since screens on some microcomputers are only forty columns wide instead of eighty, this program allows for that by using conditional assembler directives.

2. The prints macro displays a string either beginning at the extreme left of the screen or beginning at the middle.

   - The middle of an 80-column screen will be column 41 while the middle of a 40-column screen will be column 21.

   - Thus, in prints, index register Y should be loaded with 40 if the screen has 80 columns or with 20 if the screen has 40 columns.

3. This program is set up for an 80-column screen.

   - In the first line of the program, the parameter "screen" is equated (by EQU) to the number 80 in order to indicate an 80-column screen. The statement "screen equ 40" would change the value of "screen" and indicate a 40-column screen.

4. The program uses the conditional assembler directive ifeq both in the macro and in the main body of the program.

5. In the main body of the program, the statements "ifeq (screen - 80)" and "ifeq (screen - 40)" are used to test the value of screen. If the expression "(screen - 80)" is equal to zero at assembly time then the statement "ldx #scr80" is assembled. If the expression "(screen - 40)" is equal to zero at assembly time then the statement "ldx #scr40" is assembled.

6. In the macro, the assember directive "ifeq" is used in the statements "ifeq %0-80" and "ifeq %0-40". The notation "%0" stands for the first argument of the macro. In this program, the first and only argument is "screen". Macros may have many arguments with the notation "%n" indicating the (n+1)st argument.

   NOTE: *The '%' is a backslash on the keyboard.*

7. The conditional assembler directives available are IFEQ, IFGE, IFGT, IFLE, IFLT, IFNE, IFC, and IFNC.

   These directives are explained in the section on assembler directives.

Waterloo 6809 Assembler

Reference Manual

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various user. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 2

EDITOR

The purpose of the text editor, the Waterloo microEDITOR, is to create and maintain program files and source data files. While it is a line-oriented editor, it does have full-screen support. The usual commands for performing such tasks as adding, deleting, and changing text are available and, as well, special program function (PF) keys are provided. These keys can be pressed instead of entering some of the more frequently used commands. Other features include the ability to make global changes and the facility to repeat and edit the last command issued. (See the Waterloo microEDITOR manual for further explanation.)

Chapter 3

ASSEMBLER

The Waterloo 6809 Assembler will assemble normal Motorola 6809 assembly language syntax and directives along with macros, pseudo opcodes for structured programming control, and long label names. Definitions from separate files may also be included. The assembler produces object modules which can then be linked into executable load modules.

Method of Operation

The filename of each assembly language program to be assembled must be suffixed by '.asm'.

In order to assemble the program, enter 'a' when selecting from the menu.

When asked to 'Enter filename:', enter the name of the assembly languag e program file without including the '.asm' suffix. The suffix is assumed and, if it is included accidentally, the assembler will try to assemble a file with a '.asm.asm' suffix.

The program will then be assembled. The 'Assembler completed' message indicates the end of the assembly. Pressing 'RETURN' will cause a return to the menu.

Files Produced By The Assembler

The assembler produces two new files with names identical to the assembly language program filename except for the suffixes.

The '.lst' file contains a copy of the original program with its machine language translation displayed beside it on the left hand side. This file could be useful in debugging. If any errors occur in the assembly process, they will be marked by appropriate messages in the corresponding '.lst' file. The message before the 'Assembler completed' line will indicate how many diagnostics messages have occurred.

The '.b09' file contains the object code created by assembling the source code. Provided that there were no assembly errors, this object module can now be linked by the linker into a load module. At this point, the origin of the object code is at zero. The linker is responsible for its relocationto an absolute address.

Chapter 4

6809 ARCHITECTURE & INSTRUCTIONS

Registers

Both registers and memory need to be addressed. The available registers are: accumulator A (A), accumulator B (B), accumulator D (D), index register X (X), index register Y (Y), the hardware system stack pointer (S), the user stack pointer (U), the program counter (PC), the direct page register (DP), and the condition code register (CC).

1. Accumulators (A, B, and D)

Accumulators A and B are general-purpose 8-bit accumulators which are used for arithmetic calculations and manipulation of data. Accumulator D is a 16-bit double accumulator which consists of accumulator A and accumulator B concatenated together with the A register as the most significant byte.

2. Index Registers (X and Y)

Index registers X and Y are used in indexed addressing mode (to be discussed later). The 16-bit address in an index register is added to an optional offset to provide the instruction's effective address. During some indexed mode operations, the contents of the index register may be automatically post-incremented or pre-decremented.

3. Stack Pointers (S and U)

The hardware system stack pointer (S) is used automatically by the processor during subroutine calls and interrupts. The user stack pointer (U) is controlled exclusively by the programmer. This allows arguments to be passed to and from subroutines easily. The stack pointers point to the last used byte on the stack. S and U may be used as index registers while also supporting 'push' and 'pull' instructions.

4. Program Counter (PC)

The program counter is used by the processor to point to the address of the next instructionto be executed. With relative addressing, it is also used as an index register.

5. Direct Page Register (DP)

The direct page register forms the most significant byte of an instruction's effective address when direct addressing is being used. It is concatenated with the byte following the direct mode opcode.

6. Condition Code Register (CC)

The condition code register defines the current state of the processor flags. The bits in the condition code register are: the carry flag (C), the overflow flag (V), the zero flag (Z), the negative flag (N), the interrupt request mask flag (I), the half-carry flag (H), the fast interrupt request mask flag (F), and the entire state saved on stack flag (E).

The bits are numbered from right to left (7-6-5-4-3-2-1-0).

Carry (C - bit 0)
The carry flag is usually the carry from the binary arithmetic operation. C is also used to represent a 'borrow' in instructions involving subtractions such as CMP and is the complement of the carry.

Overflow (V - bit 1)
The overflow flag is set to a one by an operation which causes a signed two's complement arithmetic overflow.

Zero (Z - bit 2)
The zero flag is set to a one by an operation whose result is exactly zero.

Negative (N - bit 3)
The negative flag contains the value of the most significant bit of the preceding operations's result.

Interrupt Request Mask (I - bit 4)
The interrupt request mask flag is set to a one when the processor is to be prevented from recognizing interrupts from the interrupt request line.

Half-Carry (H - bit 5)
The half-carry flag is used to indicate a carry from bit 3 as a result of an 8-bit addition.

Fast Interrupt Request Mask (F - bit 6)
The fast interrupt request mask flag is set to a one when the processor is to be prevented from recognizing interrupts from the fast interrupt request line.

Entire State Saved On Stack (E - bit 7)
The entire state saved on stack flag is set to a one to indicate that the complete machine state (all the registers) was stacked and not just the program counter and condition code register.

Addressing Modes

1. Inherent

In inherent addressing, the opcode of the instruction contains all the address information necessary and there are no addressing options.

Ex: MUL ; Multiply A by B and store in D.

2. Accumulator

Accumulator addressing mode involves those instructions which operate on an accumulator.

```
Ex:     CLRA            ; Clear accumulator A.
        CLRB            ; Clear accumulator B.
```

3. Immediate

In immediate addressing, the data to be used in the instruction immediately follows the instruction's opcode so that the effective address of the data is the location following the opcode. There are both 8-bit and 16-bit immediate values depending on the size of argument specified by the opcode. Immediate values are known at assembly time.

```
Ex:     LDX  #CR            ; Load address of CR into X.
        LDB  #7             ; Load number 7 into B.
        LDA  #$F0           ; Load hexadecimal number F0 into A.
        LDB  #%11110000     ; Load binary number 11110000 into B.
        LDX  #$8004         ; Load hexadecimal number 8004 into X.
```

Note: # signifies immediate addressing; $ signifies a hexadecimal value; and % signifies a binary value.

4. Absolute (Immediate Indirect)

Absolute addressing refers to an exact 16-bit location in the memory addressing space and is especially useful for I/O transactions with peripherals. Another name for absolute addressing is immediate indirect addressing. There are three program-selectable modes of absolute addressing: direct, extended, and extended indirect.

- (a) Direct

With direct addressing, one byte of address follows the opcode. This byte specifies the lower 8 bits of the effective address. The upper 8 bits are supplied by the direct page register. Only 256 locations (one 'page' of memory) can be accessed without redefining the direct page register. The direct page register is set to $00 and page 00 is normally the only page for which direct addressing is used. By loading a register, such as accumulator A, with a value and then transferring that value to the direct page register, the direct page register may be reset. Caution is recommended with this procedure, however, since the system library routines will not work properly if the direct page register does not contain zero. Direct addressing is specified by specifying a one-byte address in the instruction. Since only one byte of address is required in direct addressing, this mode requires less memory and executes faster than other absolute or indexed modes. Note that indirection is not allowed with direct addressing.

```
Ex:
XX      EQU $40
        LDB $30          ; Load contents of $0030 into B.
        LDA #XX          ; Load A with $40.
        TFR A,DP         ; Reset DP to $40.
        LDB $10          ; Load contents of $4010 into B.
        CLRA             ; Load A with $00.
        TFR A,DP         ; Restore DP to $00.
```

- (b) Extended

In extended addressing, the contents of the two bytes immediately following the opcode fully specify the 16-bit effective address used by the instruction.

```
Ex:     STX MOUSE       ;Store X into locations MOUSE & MOUSE+1.
        LDD $2000       ; Load contents of $2000 & $2001 into D.
```

- (c) Extended Indirect

In extended indirect addressing, the contents of the two bytes immediately following the opcode specify a 16-bit absolute address from which to recover the effective address to be used by the instruction.

```
Ex:     LDA [CAT]       ; Effective address is address contained
                        ;in CAT & CAT+1. Load contents into A.
        LDX [$FFFE]     ; Effective address is address contained
                        ;in $FFFE & $FFFF. Load contents into X.
```

Certain instructions (SWI, SWI2, SWI3) and the interrupts use an inherent absolute address to function similarly to extended indirect addressing mode and are said to have 'absolute indirect' addressing.

5. Register

Register addressing involves those instructions which operate on the various registers available.

```
Ex:     TFR X,Y         ; Transfer contents of X into Y.
        PSHS A,B,X,Y    ; Push A, B, X, and Y onto S stack.
```

6. Indexed (Register Indirect)

In all indexed addressing, one of the pointer registers (X, Y, U, S, and sometimes PC) is used in a calculationto obtain the effective address of the instruction's operand. Various types of indexing are available: constant-offset indexed, constant-offset indexed indirect, accumulator indexed, accumulator indexed indirect, auto-increment, auto-increment indirect, auto-decrement, and auto-decrement indirect.

- (a) Constant-Offset Indexed

In constant-offset indexed addressing, an optional two's complement offset (up to 16 bits long) is added to the contents of a register to form the effective address of the instruction's operand. (In zero-offset indexed mode, the offset either is left out entirely or is zero. This mode is the fastest indexing mode since the selected pointer register contains the address of the data to be used and no addition needs to be done]

```
Ex:     LDA ,X          ; Effective address is address in X.
                        ; Load contents into A.
        LDB 0,Y         ; Effective address is address in Y.
                        ; Load contents into B.
        LDX 4,S         ; Effective address is 4 plus contents
                        ; of S. Load contents into X.
        LDY -4,U        ; Effective address is -4 plus contents
                        ; of U. Load contents into Y.
        LDA 17,PC       ; Effective address is 17 plus contents
                        ; of PC. Load contents into A.
        LDA THERE,PC    ; Effective address is formed by adding
                        ; a constant, which is the difference
                        ; between the PC and the address THERE,
                        ; to the PC. Load contents into A.
```

- (b) Constant-Offset Indexed Indirect

As with all indirect addressing, constant-offset indexed indirect addressing functions in two stages. First, an indexed address is formed by temporarily adding the offset-value to the contents of the selected register. Second, this address is used as a pointer to the instruction's effective address.

```
Ex:     LDB [0,Y]       ; Effective address is contents of
                        ; address pointed to by address in Y.
                        ; Load contents into B.
        LDY [-4,U]      ; Effective address is contents of
                        ; address pointed to by -4 plus contents
                        ; of U. Load contents into Y.
        LDA [THE,PC]    ; Effective address is contents of the
                        ; address pointed to by the addition of
                        ; the difference between the address of
                        ; THE and the contents of the PC to the
                        ; PC. Load contents into A.
```

- (c) Accumulator Indexed

In accumulator indexed addressing, the contents of an accumulator (A, B, or D) is added to the contents of a register to form the effective address of the instruction's operand.

```
Ex:     LDA B,Y         ; Effective address is formed by adding the
                        ; contents of B to the contents of Y.
                        ; Load contents into A.
        LDX D,Y         ; Effective address is formed by adding the
                        ; contents of D to the contents of Y.
                        ; Load contents into X.
        LEAX B,X        ; Effective address is formed by adding
                        ; the contents of B to the contents of X.
                        ; Load effective address into X.
```

- (d) Accumulator Indexed Indirect

Like all indirects, accumulator indexed indirect addressing functions in two stages. First, an indexed address is formed by temporarily adding the contents of the selected accumulator to the contents of the selected register. Second, this address is used as a pointer to the instruction's effective address.

```
Ex:     LDA [B,Y]       ; Effective address is contents of address
                        ;pointed to by address formed by adding the
                        ; contents of B to the contents of Y.
                        ; Load contents into A.
        LEAX [B,X]      ; Effective address is contents of address
                        ;pointed to by address formed by adding the
                        ; contents of B to the contents of X.
                        ; Load effective address into X.
```

- (e) Auto-Increment

Auto-increment addressing uses the value in the selected pointer register (X, Y, S, or U) to address a one- or two-byte value in memory. The register is then incremented by one (single +) or two (two +'s). No offset is permitted.

```
Ex:     LDA ,X+         ; Effective address is address in X.
                        ;Load contents into A. Increment X by one.
        LDY ,Y++        ; Effective address is address in Y.
                        ;Load contents into Y. Increment Y by two.
        LDB ,S+         ; Effective address is address in S.
                        ; Load contents into B. Increment S by 1.
        LDX ,U++        ; Effective address is address in U.
                        ;Load contents into X. Increment U by two.
```

- (f) Auto-Increment Indirect

Auto-increment indirect addressing uses the value in the selected register (X, Y, S, or U) to point to an address value in memory. This value is used as the instruction's effective address. The register is then incremented by two (two ++'s). The indirected increment by one is not permitted and neither are offsets.

```
Ex:     LDY [,Y++]      ; Effective address is contents of address
                        ; pointed to by address in Y. Load
                        ; contents into Y. Increment Y by 2.
        LDB [,S++]      ; Effective address is contents of address
                        ; pointed to by address in S. Load
                        ; contents into B. Increment S by two.
```

- (g) Auto-Decrement

Auto-decrement addressing first decrements the selected register (X, Y, S, or U) by one (single -) or two (two -'s). The resulting value in the register is then used as the instruction's effective address. No offset is permitted.

```
Ex:     LDA ,-X         ; Decrement X by one. Effective address is
                        ; address in X. Load contents into A.
        LDY ,--Y        ; Decrement Y by two. Effective address is
                        ; address in Y. Load contents into Y.
        LDB ,-U         ; Decrement U by one. Effective address is
                        ; address in U. Load contents into B.
        LDX ,--S        ; Decrement S by two. Effective address is
                        ; address in S. Load contents into X.
```

- (h) Auto-Decrement Indirect

Auto-decrement indirect addressing first decrements the selected register by two (two -'s). The resulting value in the register is then used as a pointer to an address value in memory. This value is used as the instruction's effective address. Auto-decrement by one indirect is not permitted and neither are offsets.

```
Ex:     LDY [,--Y]      ; Decrement Y by 2. Effective address is
                        ; contents of address pointed to by address
                        ; in Y. Load contents into Y.
        LDB [,--U]      ; Decrement U by 2. Effective address is
                        ; contents of address pointed to by address
                        ; in U. Load contents into B.
```

7. Relative

Relative addressing, also known as short relative addressing, adds the value of the immediate byte of the instruction (an 8-bit two's complement value) to the program counter to produce a new absolute address in the program counter. This addressing mode is always position independent. Only part of memory can be reached with (short) relative addressing.

```
Ex:     BEQ CAR        ; If Z bit is set then branch to CAR.
        BRA BAT        ; Branch unconditionally to BAT.
        .
        .
        .
CAR ...
BAT ...
```

8. Long Relative

Long relative addressing adds the value of the two immediate bytes of the instruction (a 16-bit two's complement value) to the program counter to produce a new absolute address in the prgram counter. This addressing mode is always position independent. All of memory can be reached with long relative addressing.

```
Ex:     LBNE RAT       ; If Z bit is clear then long branch to RAT.
        LBRA CAT       ; Long branch unconditionally to CAT.
        .
        .
        .
RAT ...
CAT ...
```

Assembly Language Instructions

ABX - add accumulator B (unsigned) to index register X

Description: ABX

The 8-bit unsigned value in accumulator B is added to the 16-bit value in index register X and the sum is stored in index register X.

ADC - add carry bit and memory byte to accumulator A or B

Description: ADCA P (or ADCB P)

The contents of the carry flag and the 8-bit value addressed by P are added to the 8-bit value in accumulator A (or B) and the sum is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

H - Set IFF there is a carry from bit 3.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 8-bit two's complement overflow.
C - Set IFF there is a carry from bit 7.

ADD - add memory byte to accumulator A or B

Description: ADDA P (or ADDB P)
The contents of the 8-bit value addressed by P are added to the 8-bit value in accumulator A (or B) and the sum is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
H - Set IFF there is a carry from bit 3.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 8-bit two's complement overflow.
C - Set IFF there is a carry from bit 7.

ADD - add 16 bits of memory to accumulator D

Description: ADDD P
The contents of the 16-bit value addressed by P is added to the 16-bit value in accumulator D and the sum is stored in accumulator D.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
N - Set IFF bit 15 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF there is a 16-bit two's complement overflow.
C - Set IFF there is a carry from bit 7 when the most significant byte (MS Byte) is operated on.

AND - logical and memory byte to accumulator A or B

Description: ANDA P (or ANDB P)

Each bit of the 8-bit value addressed by P is logically anded with the corresponding bit of the 8-bit value in accumulator A (or B) and the result is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Cleared.

AND - logical and memory immediate byte to CC register.

Description: ANDCC #XX

The contents of the 8-bit immediate value are logically anded with the 8-bit value in the condition code register and the result is stored in the condition code register.

Condition Codes:

The condition codes will be cleared or set depending on the result stored in the condition code register.

ASL - arithmetic shift left accumulator or memory

Description: ASLA, ASLB; ASL P

All bits of the 8-bit operand are shifted one place to the left. Bit 0 is loaded with a zero. Bit 7 is shifted into the carry flag.

Address Modes for P: direct, indexed, extended

Condition Codes:

H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF one and only one of bits 6 and 7 of the original operand is set; otherwise cleared.
C - Set IFF bit 7 of the original operand is set; otherwise cleared.

ASR - arithmetic shift right accumulator or memory

Description: ASRA, ASRB; ASR P

All bits of the 8-bit operand are shifted one place to the right. Bit 7 is held constant. Bit 0 is shifted into the carry flag.

Address Modes for P: direct, indexed, extended

Condition Codes:

H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
C - Set IFF bit 0 of the original operand is set; otherwise cleared.

BCC - branch on carry clear

Description: BCC dd; LBCC DDDD

If the carry flag (C bit) is set then a branch does not occur. If the carry flag is clear then a branch does occur. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBCC DDDD indicates a long branch on carry clear which uses long relative effective addressing and BCC dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BCS - branch on carry set

Description: BCS dd; LBCS DDDD

If the carry flag (C bit) is clear then a branch does not occur. If the carry flag is set then a branch does occur. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBCC DDDD indicates a long branch on carry set which uses long relative effective addressing and BCC dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BGE - branch on greater than or equal to zero

Description: BGE dd; LBGE DDDD

A branch does not occur unless both the negative flag (N bit) and the overflow flag (V bit) are set or both the negative and overflow flags are clear. Having both flags clear means that the value of a valid two's complement result is greater than or equal to zero. Having both flags set means that an overflow occurred but, if it hadn't the value of the two's complement result would be greater than or equal to zero. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBGE DDDD indicates a long branch on greater than or equal to zero which uses long relative effective addressing and BGE dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BGT - branch on greater than zero

Description: BGT dd; LBGT DDDD

A branch does not occur unless (a) both the negative flag (N bit) and the overflow flag (V bit) are set or both the negative and overflow flags are clear and (b) the zero flag (Z bit) is clear. Having both the negative and overflow flags clear means that the value of a valid two's complement result is greater than or equal to zero. Having both the negative and overflow flags set means that an overflow occurred but, if it hadn't, the value of the two's complement result would be greater than or equal to zero. Having the zero flag clear means that the value of the two's complement result is non-zero. Thus, a branch occurs if the value of the two's complement result is greater than zero. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBGT DDDD indicates a long branch on greater than zero which uses long relative effective addressing and BGT dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BHI - branch on higher

Description: BHI dd; LBHI DDDD

A branch does not occur unless both the carry flag (C bit) and the zero flag (Z bit) are clear. The 'branch on higher' instruction is the same as the 'branch on greater than zero' except that it compares unsigned values and 'branch on greater than zero' compares signed values. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBHI DDDD indicates a long branch on higher which uses long relative effective addressing and BHI dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BHS - branch on higher or same

Description: BHS dd; LBHS DDDD

A branch does not occur unless the carry flag (C bit) is clear. The 'branch on higher or same' instruction is the same as the 'branch on greater than or equal to zero' except that it compares unsigned values and 'branch on greater than or equal to zero' compares signed values. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBHS DDDD indicates a long branch on higher or same which uses long relative effective addressing and BHS dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BIT - bit test (ANDing memory byte with accumulator A or B)

Description: BITA P (or BITB P)

Each bit of the 8-bit value addressed by P is logically anded with the corresponding bit of the 8-bit value in accumulator A (or B). The condition codes are modified to reflect the result but the result is not stored anywhere. The accumulator and the memory byte are both left unchanged.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Cleared.

BLE - branch on less than or equal to zero

Description: BLE dd; LBLE DDDD

A branch does not occur unless (a) one and only one of the negative (N bit) and overflow (V bit) flags are set or (b) the zero flag (Z bit) is set. Having the negative flag set and the overflow flag clear means that the value of a valid two's complement result is less than zero. Having the negative flag clear and the overflow flag set means that an overflow occurred but, if it hadn't, the value of the two's complement result would be less than zero. Having the zero flag set means the value of the result is zero. Thus, a branch occurs if the value of the two's complement result is less than or equal to zero. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBLE DDDD indicates a long branch on less than or equal to zero which uses long relative effective addressing and BLE dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BLO - branch on lower

Description: BLO dd; LBLO DDDD

A branch does not occur unless the carry flag (C bit) is set. The 'branch on lower' instruction is the same as the 'branch on less than zero' except that it compares unsigned values and 'branch on less than zero' compares signed values. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBLO DDDD indicates a long branch on lower which uses long relative effective addressing and BLO dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BLS - branch on lower or same

Description: BLS dd; LBLS DDDD

A branch does not occur unless either the carry flag (C bit) or the zero flag (Z bit) is set. The 'branch on lower or same' instruction is the same as the 'branch on less than or equal to zero' except that it compares unsigned values and 'branch on less than or equal to zero' compares signed values. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBLS DDDD indicates a long branch on lower or same which uses long relative effective addressing and BLS dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calcualated by the assembler.)

BLT - branch on less than zero

Description: BLT dd; LBLT DDDD

A branch does not occur unless one and only one of the negative (N bit) and overflow (V bit) flags are set. Having the negative flag set and the overflow flag clear means that the value of a valid two's complement result is less than zero. Having the negative flag clear and the overflow flag set means that an overflow occurred but, if it hadn't, the value of the two's complement result would be less than zero. Thus, a branch occurs if the value of the two's complement result is less than zero. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBLT DDDD indicates a long branch on less than or equal to zero which uses long relative effective addressing and BLT dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BMI - branch on minus

Description: BMI dd; LBMI DDDD

A branch occurs if the negative flag (N bit) is set. Thus, this branch occurs if the value of the two's complement result is less than zero without taking into account that the result might be invalid due to an overflow. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. BMI indicates a short branch on minus which uses relative effective addressing and LBMI indicates a long branch which uses long relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BNE - branch on not equal

Description: BNE dd; LBNE DDDD

If the zero flag (Z bit) is set then a branch does not occur. If the zero flag is clear then a branch does occur. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. LBNE DDDD indicates a long branch on equal which uses long relative effective addressing and BNE dd indicates a short branch which uses relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BPL - branch on plus

Description: BPL dd; LBPL DDDD

A branch occurs if the negative flag (N bit) is clear. Thus, a branch occurs if the value of the two's complement result is greater than or equal to zero without taking into account that the result might be invalid due to an overflow. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. BPL dd indicates a short branch on plus which uses relative effective addressing and LBPL DDDD indicates a long branch which uses long relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BRA - branch always

Description: BRA dd; LBRA DDDD

BRA dd cause an unconditional short branch which uses relative effective addressing and LBRA DDDD causes an unconditional long branch which uses long relative effective addressing. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. The branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BRN - branch never

Description: BRN dd; LBRN DDDD

BRN dd and LBRN DDDD are essentially 'NO-OP' (no operation) instructions. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BSR - branch to subroutine

Description: BSR dd; LBSR DDDD

dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. BSR dd indicates a short branch to subroutine which uses relative effective addressing and LBSR DDDD indicates a long branch which uses long relative effective addressing. The program counter is pushed onto the stack in order to act as the return address when program control is returned to the calling program from the subroutine. The branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BVC - branch on overflow clear

Description: BVC dd; LBVC DDDD

A branch does not occur unless the overflow flag (V bit) is clear. Thus, a branch occurs if the value of the two's complement result is valid. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. BVC dd indicates a short branch on overflow clear which uses relative effective addressing and LBVC DDDD indicates a long branch which uses long relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

BVS - branch on overflow set

Description: BVS dd; LBVS DDDD

A branch does not occur unless the overflow flag (V bit) is set. Thus, a branch occurs if the value of the two's complement result is invalid. dd and DDDD are memory immediate values. dd is an 8-bit offset and DDDD is a 16-bit offset. BVS dd indicates a short branch on overflow set which uses relative effective addressing and LBVS DDDD indicates a long branch which uses long relative effective addressing. A branch is accomplished by adding the memory immediate value to the contents of the program counter and storing the sum in the program counter. (A branch to a label is equivalent to a branch to an offset as the offset to the label is calculated by the assembler.)

CLR - clear accumulator or memory

Description: CLRA, CLRB; CLR P
All bits of the 8-bit operand are cleared to zero.

Address Modes for P: direct, indexed, extended

Condition Codes:
N - Cleared.
Z - Set.
V - Cleared.
C - Cleared.

CMP - compare memory byte to accumulator A or B

Description: CMPA P (or CMPB P)
The 8-bit value addressed by P is compared to the 8-bit value in accumulator A (or B) by subtracting the value addressed by P from the value in the accumulator. The condition codes are modified to reflect the result but the result is not stored anywhere. The accumulator and the memory byte are both left unchanged.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 8-bit two's complement overflow.
C - Set IFF there is NO carry from bit 7.

CMP - compare 16 bits of memory to a 16-bit register

Description: CMPD P; CMPX P (or CMPY P); CMPU P; CMPS P
The contents of the 16-bit value addressed by P is compared to the 16-bit value in the designated register by subtracting the memory value from the register value. The condition codes are modified to reflect the result but the result is not stored anywhere. Neither the register nor the memory value are changed.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 16-bit two's complement overflow.
C - Set IFF there is NO carry from bit 7
when the MS byte is operated on.

COM - complement accumulator or memory

Description: COMA, COMB; COM P
Each bit of the 8-bit operand is logically complemented. Each zero bit becomes a one and each one bit becomes a zero.

Address Modes for P: direct, indexed, extended

Condition Codes:
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Cleared.
C - Set.

CWAI - clear and wait for interrupt

Description: CWAI #$XX

The memory immediate byte is ANDed with the condition code register and the result is stored in the condition code register. The 'entire state saved on stack' flag (E bit) is set. Then the entire machine state is stacked on the hardware system stack: the program counter, the user stack pointer, index register Y, index register X, the direct page register, accumulator B, accumulator A, and the condition code register. When a (non-masked) interrupt occurs, no further machine state will be saved before vectoring to the interrupt handling routine.

Condition Codes:

The condition codes may be cleared by the ANDing of the memory immediate byte with the condition code register.

DA - decimal addition adjust on accumulator A

Description: DAA

The sequence of a single-byte add instruction on accumulator A (either ADDA or ADCA) followed by a DAA instruction results in a binary coded decimal (BCD) addition with appropriate carry flag. Both values to be added must be in proper BCD form which means that all four nybbles must have values between 0 and 9. Multiple-precision additions must add the carry generated by this DAA instruction into the next higher digit during the add operation immediately prior to the next DAA. The DAA instruction adds one of the binary values 00000000, 00000110, 01100000, and 01100110 to accumulator A. Which value is chosen depends on the following: if (a) the half-carry flag (H bit) is set or (b) the value of the least significant nybble (LSN) in accumulator A is greater than 9, then the correction value to be added has 0110 in its LSN; if (a) the carry flag (C bit) is set or (b) the value of the most significant nybble (MSN) in accumulator A is greater than 9 or (c) the value of the MSN is greater than 8 and the value of the LSN is greater than 9, then the correction value to be added has 0110 in its MSN.

Condition Codes:

N - Set IFF the MSB of the result is set.
Z - Set IFF all bits of the result are clear.
V - Undefined.
C - Set if there is a carry from bit 7 or if
the carry flag was set before the operation.

DEC - decrement accumulator or memory by one

Description: DECA, DECB; DEC P
The value 1 is subtracted from the operand.

Address Modes for P: direct, indexed, extended

Condition Codes:
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF the original operand was 10000000.

EOR - exclusive or memory byte to accumulator A or B

Description: EORA P (or EORB P)
Each bit of the 8-bit value addressed by P is exclusive ored with the corresponding bit of the 8-bit value in accumulator A (or B) and the result is stored in accumulator A (or B).

Address Modes for P: direct, extended, immediate, indexed

Condition Codes:
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of result are clear.
V - Cleared.

EXG - exchange registers

Description: EXG R1, R2

The contents of register R1 are exchanged with those of R2. Registers may only be exchanged with registers of like size. 8-bit registers are to be exchanged with 8-bit registers and 16-bit registers are to be exchanged with 16-bit registers. The possible 8-bit registers are A, B, CC, and DP and the possible 16-bit registers are D, X, Y, U, S, and PC.

Condition Codes:

The condition codes are not affected unless one of R1 and R2 is the condition code register (CC).

INC - increment accumulator or memory by one

Description: INCA, INCB; INC P

The value 1 is added to the operand.

Address Modes for P: direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF the original operand was 01111111.

JMP - jump to effective address

Description: JMP P

Program control is transferred to the location equivalent to the effective address by loading the program counter with the effective address indicated by P.

Address Modes for P: direct, indexed, extended

JSR - jump to subroutine at effective address

Description: JSR P

The program counter is pushed onto the hardware stack in order to act as the return address when program control is returned to the calling program from the subroutine. Program control is transferred to the location equivalent to the effective address by loading the program counter with the effective address indicated by P.

Address Modes for P: direct, indexed, extended

LD - load memory byue into accumulator A or B

Description: LDA P (or LDB P)

The contents of the 8-bit value addressed by P are loaded into accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the loaded data is set.
Z - Set IFF all bits of the loaded data are clear.
V - Cleared.

LD - load 16 bits of memory into a 16-bit register

Description: LDD P; LDX P (or LDY P); LDS P; LDU P

The contents of the 16-bit value addressed by P is loaded into the designated register.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

N - Set IFF bit 15 of the loaded data is set.
Z - Set IFF all bits of the loaded data are clear.
V - Cleared.

LEA - load effective address

Description: LEAX P (or LEAY P); LEAS P; LEAU P
Form the effective address to data using the memory addressing mode and load that address, not the data itself, into the pointer register designated by the instruction (eg. X for LEAX).

Memory Addressing Mode for P: indexed

Condition Codes:
Z - Set IFF all bits of the result are clear and the instruction was LEAX or LEAY.
If the instruction was LEAS or LEAU then the Z bit is not affected.

LSL - logical shift left accumulator or memory

Description: LSLA, LSLB; LSL P
All bits of the 8-bit operand are shifted one place to the left. Bit 0 is loaded with a zero. Bit 7 is shifted into the carry flag. The LSL instruction is the same as the ASL instruction.

Address Modes for P: direct, indexed, extended

Condition Codes:
H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF one and only one of bits 6 and 7 of the original operand is set; otherwise cleared.
C - Set IFF bit 7 of the original operand is set; otherwise cleared.

LSR - logical shift right accumulator or memory

Description: LSRA, LSRB; LSR P

All bits of the 8-bit operand are shifted one place to the right. Bit 7 is loaded with a zero. Bit 0 is shifted into the carry flag.

Address Modes for P: direct, indexed, extended

Condition Codes:

N - Cleared.
Z - Set IFF all bits of the result are clear.
C - Set IFF bit 0 of the original operand is set;
otherwise cleared.

MUL - multiply (unsigned) accumulators A and B

Description: MUL

The unsigned binary numbers in accumulators A and B are multiplied together and the result is stored in the D accumulator. Unsigned multiply allows multiple-precision operations.

Condition Codes:

Z - Set IFF all bits of the result are clear.
C - Set IFF bit 7 of accumulator B of the result is set.

NEG - negate accumulator or memory

Description: NEGA, NEGB; NEG P

The operand is replaced by its two's complement. (10000000 is replaced by itself and the V bit is set. 00000000 is replaced by itself and the C bit is cleared]

Address Modes for P: direct, indexed, extended

Condition Codes:

H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF the original operand was 10000000.
C - Set IFF there is NO carry from bit 7.

NOP - no operation

Description: NOP

This is a single-byte instruction that only causes the program counter to be incremented. No other registers or memory contents are affected.

OR - inclusive or memory immediate byte toaccumulator A or B

Description: ORA P (or ORB P)

Each bit of the 8-bit value addressed by P is inclusive ored with the corresponding bit of the 8-bit value in accumulator A (or B) and the result is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Cleared.

OR - inclusive or memory immediate byte to CC register

Description: ORCC #XX

The contents of the 8-bit immediate value are inclusive ored with the 8-bit value in the condition code register and the result is stored in the condition code register.

Condition Codes:

The condition codes will be cleared or set depending on the result stored in the condition code register.

PSHS - push registers on the system hardware stack

Description: PSHS register list

Any registers specified are pushed onto the system hardware stack. The possible registers are the PC, U, Y, X, DP, B, A, and CC registers. (The hardware stack pointer itself cannot be specified] A register list consists of the register names separated by commas. The registers are always pushed on the stack in the following order: program counter, user stack pointer, index register Y, index register X, direct page register, accumulator B, accumulator A, and condition code register. The assembler converts the list of registers to an 8-bit immediate memory value whose 8 bits correspond to the 8 registers. Which registers are pushed depends on which bits of the memory immediate value are set. Bit 7 corresponds to the program counter, bit 6 to the user stack pointer, bit 5 to index register Y, bit 4 to index register X, bit 3 to the direct page register, bit 2 to accumulator B, bit 1 to accumulator A, and bit 0 to the condition code register.

PSHU - push registers on the user stack

Description: PSHU register list

Any registers specified are pushed onto the user stack. The possible registers are the PC, S, Y, X, DP, B, A, and CC registers. (The user stack pointer itself cannot be specified] A register list consists of the register names separated by commas. The registers are always pushed on the stack in the following order: program counter, hardware stack pointer, index register Y, index register X, direct page register, accumulator B, accumulator A, and condition code register. The assembler converts the list of registers to an 8-bit immediate memory value whose 8 bits correspond to the 8 registers. Which registers are pushed depends on which bits of the memory immediate value are set. Bit 7 corresponds to the program counter, bit 6 to the hardware stack pointer, bit 5 to index register Y, bit 4 to index register X, bit 3 to the direct page register, bit 2 to accumulator B, bit 1 to accumulator A, and bit 0 to the condition code register.

PULS - pull registers from the system hardware stack

Description: PULS register list

Any registers specified are pulled from the system hardware stack. The possible registers are the PC, U, Y, X, DP, B, A, and CC registers. (The hardware stack pointer itself cannot be specified] A register list consists of the register names separated by commas. The registers are always pulled from the stack in the following order; condition code register, accumulator A, accumulator B, direct page register, index register X, index register Y, user stack pointer, and program counter. The assembler converts the list of registers to an 8-bit immediate memory value whose 8 bits correspond to the 8 registers. Which registers are pulled depends on which bits of the memory immediate value are set. Bit 7 corresponds to the program counter, bit 6 to the user stack pointer, bit 5 to index register Y, bit 4 to index register X, bit 3 to the direct page register, bit 2 to accumulator B, bit 1 to accumulator A, and bit 0 to the condition code register.

Condition Codes:

The condition codes may be changed if the condition code register is pulled from the stack but otherwise they are unaffected.

PULU - pull registers from the user stack

Description: PULU register list

Any registers specified are pulled from the user stack. The possible registers are the PC, S, Y, X, DP, B, A, and CC register. (The user stack pointer itself cannot be specified] A register list consists of the register names separated by commas. The registers are always pulled from the stack in the following order: condition code register, accumulator A, accumulator B, direct page register, index register X, index register Y, user stack pointer, and program counter. The assembler converts the list of registers to an 8-bit immediate memory value whose 8 bits correspond to the 8 registers. Which registers are pulled depends on which bits of the memory immediate value are set. Bit 7 corresponds to the program counter, bit 6 to the hardware stack pointer, bit 5 to index register Y, bit 4 to index register X, bit 3 to the direct page register, bit 2 to accumulator B, bit 1 to accumulator A, and bit 0 to the condition code register.

Condition Codes:

The condition codes may be changed if the condition code register is pulled from the stack but otherwise they are unaffected.

ROL - rotate left accumulator or memory

Description: ROLA, ROLB; ROL P

All bits of the 8-bit operand are shifted one place to the left. The carry flag is rotated into bit 0 and bit 7 is rotated into the carry flag.

Address Modes for P: direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF one and only one of bits 6 and 7 of the original operand is set; otherwise cleared.
C - Set IFF bit 7 of the original operand is set; otherwise cleared.

ROR - rotate right accumulator or memory

Description: RORA, RORB; ROR P

All bits of the 8-bit operand are shifted one place to the right. The carry flag is rotated into bit 7 and bit 0 is rotated into the carry flag.

Address Modes for P: direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
C - Set IFF bit 0 of the original operand is set; otherwise cleared.

RTI - return from interrupt

Description: RTI

The condition code register is pulled from the system hardware stack. The rest of the saved machine stack is recovered from the hardware stack and control is returned to the interrupted program. If the recovered 'entire state saved on stack' flag (E bit) is clear then only the program counter needs to be pulled from the stack. If the E bit is set then all the following are pulled from the stack: accumulator A, accumulator B, direct page register, index register X, index register Y, user stack pointer, and program counter.

Condition Codes:

The condition code register is recovered from the stack.

RTS - return from subroutine

Description: RTS

Program control is returned from the subroutine to the calling program by pulling the return address from the stack into the program counter.

SBC - subtract carry bit and memory byte from accumulator A or B

Description: SBCA P (or SBCB P)

The contents of the carry flag and the 8-bit value addressed by P are subtracted from the 8-bit value in accumulator A (or B) and the result is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:

H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 8-bit two's complement overflow.
C - Set IFF there is NO carry from bit 7.

SEX - sign extended

Description: SEX

A two's complement 8-bit value in accumulator B is transformed into a two's complement 16-bit value in accumulator D. If bit 7 of accumulator B is set then accumulator A is loaded with 11111111. If bit 7 of accumulator B is clear then accumulator A is loaded with 00000000.

Condition Codes:

N - Set IFF accumulator A is loaded with 11111111.
Z - Set IFF all bits of accumulator D are clear.

ST - store accumulator A or B into memory byte

Description: STA P (or STB P)
The contents of accumulator A (or B) are stored into the memory location addressed by P.

Address Modes for P: direct, indexed, extended

Condition Codes:
N - Set IFF bit 7 of the stored data is set.
Z - Set IFF all bits of the stored data are clear.
V - Cleared.

ST - store 16-bit register into 16 bits of memory

Description: STD P; STX P (or STY P); STS P; STU P
The contents of the designated register are stored into the consecutive memory locations addressed by P and P+1.

Address Modes for P: direct, indexed, extended

Condition Codes:
N - Set IFF bit 15 of the stored data is set.
Z - Set IFF all bits of the stored data are clear.
V - Cleared.

SUB - subtract memory byte from accumulator A or B

Description: SUBA P (or SUBB P)
The contents of the 8-bit value addressed by P are subtracted from the 8-bit value in accumulator A (or B) and the result is stored in accumulator A (or B).

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
H - Undefined.
N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 8-bit two's complement overflow.
C - Set IFF there is NO carry from bit 7.

SUB - subtract 16 bits of memory from accumulator D

Description: SUBD P
The contents of the 16-bit value addressed by P is subtracted from the 16-bit value in accumulator D and the result is stored in accumulator D.

Address Modes for P: immediate, direct, indexed, extended

Condition Codes:
N - Set IFF bit 15 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Set IFF 16-bit two's complement overflow.
C - Set IFF there is NO carry from bit 7.

SWI - software interrupt

Description: SWI

The 'entire state saved on stack' flag (E bit) is set. Then the entire machine state is stacked on the hardware system stack: the program counter, the user stack pointer, index register Y, index register X, the direct page register, accumulator B, accumulator A, and the condition code register. The 'interrupt request mask' flag (I bit) and the 'fast interrupt request mask' flag (F bit) are set. Then control is transferred through the SWI vector by loading the program counter with the contents of locations FFFA and FFFB.

SWI2 - software interrupt 2

Description: SWI2

The 'entire state saved on stack' flag (E bit) is set. Then the entire machine state is stacked on the hardware system stack: the program counter, the user stack pointer, index register Y, index register X, the direct page register, accumulator B, accumulator A, and the condition code register. Then control is transferred through the SWI2 vector by loading the program counter with the contents of locations FFF4 and FFF5.

SWI3 - software interrupt 3

Description: SWI3

The 'entire state saved on stack' flag (E bit) is set. Then the entire machine state is stacked on the hardware system stack: the program counter, the user stack pointer, index register Y, index register X, the direct page register, accumulator B, accumulator A, and the condition code register. Then control is transferred through the SWI3 vector by loading the program counter with the contents of locations FFF2 and FFF3.

SYNC - synchronize to external event

Description: SYNC

When a SYNC instruction is executed, the computer enters a SYNCing state, stops processing instructions, and waits on an interrupt. When an interrupt occurs, the SYNCing state is cleared and processing continues. If the interrupt is enabled and the interrupt lasts 3 cycles or more, the processor will perform the interrupt routine. If the interrupt is masked or is shorter than 3 cycles long, the processor simply continues to the next instruction.

TFR - transfer register to register

Description: TFR R1, R2

The contents of register R1 are loaded into register R2. Registers may only be transferred between registers of like size. 8-bit registers are to be transferred to 8-bit registers and 16-bit registers are to be transferred to 16-bit registers. The possible 8-bit registers are A, B, CC, and DP and the possible 16-bit registers are D, X, Y, U, S, and PC.

Condition Codes:

The condition codes are not affected unless R2 is the condition code register (CC).

TST - test accumulator or memory

Description: TSTA, TSTB; TST P

The overflow flag is cleared and the negative and zero flags are set according to the contents of the 8-bit operand.

Address Modes for P: direct, indexed, extended

Condition Codes:

N - Set IFF bit 7 of the result is set.
Z - Set IFF all bits of the result are clear.
V - Cleared.

Assembler Directives

The Assembler directives are instructions to the Assembler, rather than instructions to be directly translated into object code. This section describes the directives that are recognized by the Waterloo 6809 Assembler. Detailed descriptions of each directive are arranged alphabetically. The notations used here are:

(* *) Contains a list of elements, one of which must be selected. Each choice will be separated by a vertical bar. For example, (*IFC|IFNC*) indicates that either IFC or IFNC must be selected.

[ ] Contains an optional element. If one of a series of elements may be selected, the available list of choices will be contained within the brackets. Each choice will be separated by a vertical bar. For example, [LIST|NOLIST] indicates that either LIST or NOLIST may be selected.

XYZ The names of the directives are printed in capital letters. The required parts of directive operands will also be printed in capital letters. All elements outside of the angle brackets (<>) must be specified as-is. For example, the syntactical element [<expr>,] requires the comma to be specified if the optional element <number> is selected.

< > The element names are printed in lower case and contained in angle brackets. The following elements are used in the subsequent descriptions:

<comment>	A statement's comment field
<label>	A statement label
<expr>	An Assembler expression
<filename>	A diskette filename
<string>	A string of ASCII characters
<sym>	An Assembler symbol

Assembler Expressions

An assembler expression is a combination of symbols, constants, algebraic operators, and parentheses following the conventional rules of algebra. An expression specifies a value which is to be used as an operand. Relocatable or externally defined symbols may be used in an expression but relocatable symbols or expressions cannot be operated on except for addition or subtraction.

Parenthetical expressions are evaluated first, with the innermost parentheses being evaluated before any outer ones. Evaluation takes place from left to right. Operators can operate on numeric constants, single character ASCII literals, and symbols.

The operators are:

-	unary minus
>	(monadic) low byte
<	(monadic) high byte
>	(dyadic) shift right
<	(dyadic) shift left
*	multiplication
/	division
+	addition
-	subtraction
&	logical AND
!	inclusive OR
↑	exclusive OR (↑ is the up-arrow on the keyboard)

Comment

Description: ; [<string>]

A comment is begun by a semi-colon and consists of everything from the semi-colonto the end of the line.

Include

Description: ;INCLUDE < <filename> >

The ;INCLUDE directive causes the contents of the specified diskette file to be included in the program in place of the directive. The file is specified by a filename enclosed by angle brackets. If the filename does not contain a disk drive designation, the default is drive 0.

DSCT - Data Section

Description: DSCT [<comment>]

The DSCT directive is accepted by the assembler but does not do anything.

END - End of Source Program

Description: END [<comment>]

The END directive indicates that the logical end of the source program has been encountered.

ENDC - End of Conditional Assembly

Description: ENDC [<comment>]

The ENDC directive is used to signify the end of the current level of conditional assembly (see IFxx).

ENDM - End of Macro Definition

Description: ENDM [<comment>]

The ENDM directive is used in a macro definition (see MACR). Its presence indicates the end of the macro definition.

EQU - Equate Symbol to a Value

Description: <label> EQU <expr> [<comment>]

The EQU directive assigns the value of the expression in the operand field to the label. The EQU directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program. The expression cannot contain any external references or undefined symbols. The expression may, however, be relocatable.

FAIL - Programmer Generated Error

Description: FAIL

The FAIL directive will cause an error message to be printed by the Assembler. The total error count will be incremented as with any other error. The FAIL directive is normally used in conjunction with conditional assembler directives for exceptional condition checking. The assembly proceeds normally after the error has been printed.

FCB - Form Constant Byte

Description:
[<label>] FCB (*<expr>[,<expr>,..,<expr>]*)[<comment>]

The FCB directive may have one or more operands separated by commas. The value of each operand is truncated to eight bits, and is stored in a single byte of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression.

FCC - Form Constant Character String

Description:
[<label>] FCC "<string>"[<comment>]

The FCC directive is used to store ASCII strings into consecutive bytes of memory. Any of the printable ASCII characters can be contained in the string. The string is delimited by quotes.

FDB - Form Double Byte Constant

Description:
[<label>] FDB (*<expr>[,<expr>,..,<expr>]*)[<comment>]

The FDB directive may have one or more operands separated by commas. The 16-bit value corresponding to each operand is stored into two consecutive bytes of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression.

IFxx - Conditional Assembly Directives

Description: (*IFC|IFNC*) <string 1>,<string 2>
or
(*IFEQ|IFGE|IFGT|IFLE|IFLT|IFNE*) <expr> [<comment>]

The IFxx directives are used to conditionally assemble a section of a source program. The portion of the source program following the IFxx directive up to the next ENDC directive is conditionally assembled, depending on the result of the string comparisons (first form) or depending on the value of the expression in relationto the condition (the second form).

The IFC directive will cause the subsequent statements to be assembled if the two strings compare. The IFNC directive will cause the subsequent statements to be assembled if the two strings do not compare. In either case, if the condition is not met (comparison in the first case, and no comparison in the second case), the subsequent statements will be excluded from the assembly. The beginning of <string 1> is the first non-blank, non-comma character after the IFxx directive. The end of <string 1> is the last character before the first comma. The beginning of <string 2> is the first character after the first comma. The end of <string 2> is the last character before the end of the source line. Thus, if the first form of the IFxx directive is used, no comment can appear on the source statement. Both <string 1> and <string 2> can be null. <string 1> will be null if only a comma is specified after the IFxx directive. <string 2> will be null if nothing is found after the comma.

If the second form of the IFxx directive is used, the subsequent statements will be assembled if the expression is:

IFEQ -- equal to zero
IFGE -- greater than or equal to zero
IFGT -- greater than zero
IFLE -- less than or equal to zero
IFLT -- less than zero
IFNE -- not equal to zero

If the condition is not met, the subsequent statements will be excluded from the assembly.

MACR - Macro Definition

Description: <label> MACR [<comment>]

The MACR directive is used to define a macro. All statements following the MACR directive up to the next ENDM directive become a part of the macro definition. The required label is the symbol by which the macro will subsequently be called. The MACR directive is one of the directives that assigns a value other than the program counter to the label. Macro names must not be names of existing instruction mnemonics, root mnemonics (e.g., SUB, EOR, ADD, etc], or Assembler directives. Macro definitions may not be nested -- that is, another MACR directive cannot be encountered before the ENDM directive.

NAM - Assign Program Name

Description: NAM [<string> [<comment>]]

The NAM directive is accepted by the assembler but does not do anything.

OPT - Assembler Output Options

Description: OPT [LIST|NOLIST]

OPT LIST will cause the listing to be printed (in the '.lst' file) from this point on until an OPT NOLIST directive is encountered. OPT NOLIST will cause the listing to be turned off from this point on (including the OPT NOLIST directive) until an OPT LIST directive is encountered. OPT LIST is the default if neither is specified.

ORG - Set Program Counter to Origin

Description: ORG <expr> [<comment>]

The ORG directive changes the program counter to the value specified by the expression in the operand field. Subsequent statements are assembled into memory locations starting with the new program counter value. If no ORG directive is encountered in a source program, the program counter is initialized to zero.

PAGE - Top of Page

Description: PAGE

The PAGE directive causes the Assembler to advance the paper to the top of the next page.

PSCT - Program Section

Description: PSCT (<comment>)

The PSCT directive is accepted by the assembler but does not do anything.

RMB - Reserve Memory Bytes

Description: [<label>] RMB <expr> [<comment>]

The RMB directive causes the location counter to be advanced by the value of the expression in the operand field. This directive reserves a block of memory the length of which in bytes is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any external references, forward references, or undefined symbols. The value of the expression cannot be relocatable.

SET - Set Symbol to a Value

Description: <label> SET <expr> [<comment>]

The SET directive assigns the value of the expression in the operand field to the label. The SET directive functions like the EQU directive. However, labels defined via the SET directive can have their values redefined in another part of the program (but only through the use of another SET directive). The SET directive is useful in establishing temporary or re-usable counters within macros. The value of the expression cannot be relocatable.

TTL - Initialize Page Heading

Description: TTL [<string>]

The TTL directive is accepted by the assembler but does not do anything.

XDEF - External Symbol Definition

Description: XDEF <sym>[,<sym>,...,<sym>] [<comment>]

The XDEF directive is used to specify that the list of symbols is defined within the current source program, and that those definitions should be passed to the Waterloo 6809 Linker so that other programs may reference these symbols. If the symbols contained in the directive's operand field are not defined in the program, an error will be generated.

XREF - External Symbol Reference

Description: XREF <sym>[,<sym>,...,<sym>] [<comment>]

The XREF directive is used to specify that the list of symbols is referenced in the current source program, but is defined (via XDEF directive) in another program.

If the XREF directive is not used to specify that a symbol is defined in another program, an error will be generated, and all references within the current program to such a symbol will be flagged as undefined.

Chapter 5

STRUCTURED PROGRAMMING STATEMENTS

There are three types of structured programming statements: if statements, guess statements, and loop statements. If statements and guess statements provide a means of selection where there are two or more code sequences of which only one is to be executed. Loop statements make it possible to repeat certain sequences as many times as necessary.

If Statement

The if statement has two possible forms: if/endif and if/else/endif.

(a) The if/endif statement has the following form:

```
IF <condition>
.
.
.
ENDIF
```

If the specified condition is true then execute the statement(s) between the IF <condition> and the ENDIF; otherwise do not execute them but continue execution with the statement following the ENDIF.

<condition> is one of: CC, CS, EQ, GE, GT, HI, HS, LE, LO, LS, LT, MI, NE, PL, VC, and VS. <condition> will be more fully explained later in this section.

(b) The if/else/endif statement has the following form:

```
IF <condition>
.
.
.
ELSE
.
.
.
ENDIF
```

If the specified condition is true then execute the statement(s) between the IF <condition> and the ELSE; otherwise execute the statement(s) between the ELSE and the ENDIF. Then, continue execution with the statement following the ENDIF.

Guess Statement

The guess statement has many possible forms:
guess/endguess,
guess/admit/endguess,
guess/admit/admit/endguess,
guess/admit/admit/admit/endguess, ...

(a) The guess/endguess statement has the following form:

```
GUESS
.
.
.
QUIF <condition>
.
.
.
ENDGUESS
```

Execute the statement(s) between the GUESS and the QUIF <condition>. If the specified condition is false then execute the statement(s) between the QUIF <condition> and the ENDGUESS; otherwise do not execute them but quit and continue execution with the statement following the ENDGUESS.

(b) The guess/admit/endguess statement has the following form:

(i)

```
GUESS
.
.
.
QUIF <condition>
.
.
.
ADMIT
.
.
.
ENDGUESS
```

Execute the statement(s) between the GUESS and the QUIF <condition>. If the specified condition is false then execute the statement(s) between the QUIF <condition> and the ADMIT; otherwise execute the statement(s) between the ADMIT and the ENDGUESS. Then, continue execution with the statement following the ENDGUESS.

(ii)

GUESS
.
.
.
QUIF <condition> (1)
.
.
.
ADMIT
.
.
.
QUIF <condition> (2)
.
.
.
ENDGUESS

Execute the statement(s) between the GUESS and the QUIF <condition>. If the specified condition (1) is false then execute the statement(s) between the QUIF <condition> and the ADMIT and then continue execution with the statement following the ENDGUESS. If the specified condition (1) is true then do the following: execute the statement(s) between the ADMIT and the QUIF <condition>; if the specified condition (2) is false then execute the statement(s) between the QUIF <condition> and the ENDGUESS or if the specified condition (2) is true then simply continue execution with the statement following the ENDGUESS.

(c) The guess/admit/.../admit/endguess statements all work in the same way as the guess/admit/endguess statement in (b). When a QUIF <condition> is encountered: if the condition is false then execution continues with the next statement; if the condition is true then execution continues with the statement following the next ADMIT or ENDGUESS. When an ADMIT is encountered, execution continues with the statement following the ENDGUESS.

Loop Statement

The loop statement has two possible forms: loop/endloop and loop/until.

(a) The loop/endloop statement has the following form:

```
LOOP
.
.
.
QUIF <condition>
.
.
.
ENDLOOP
```

Execute the statement(s) between the LOOP and the QUIF <condition>. If the specified condition is false then execute the statement(s) between QUIF <condition> and ENDLOOP; otherwise quit the loop and continue execution with the statement following the ENDLOOP. After the statement(s) between QUIF <condition> and ENDLOOP have been executed and the ENDLOOP has been encountered, execution continues with the statement following the LOOP. The looping process can only be terminated by the QUIF <condition> being true.

(b) The loop/until statement has the following terms:

(i)

```
LOOP
.
.
.
UNTIL <condition>
```

Execute the statement(s) between the LOOP and the UNTIL <condition). If the specified condition is false then execution continues with the statement following the word LOOP; otherwise quit the loop and continue execution with the statement following the UNTIL <condition>.

(ii)

```
LOOP
.
.
.
QUIF <condition> (1)
.
.
.
UNTIL <condition> (2)
```

Execute the statement(s) between the LOOP and the QUIF <condition>. If the specified condition (1) is false then execute the statement(s) between the QUIF <condition> and the UNTIL <condition>; otherwise continue execution with the statement following the UNTIL <condition>. After the statement(s) between the QUIF <condition> and the UNTIL <condition> have been executed and the UNTIL <condition> has been encountered, the following actions occur: if the specified condition (2) is false then execution continues with the statement following the word LOOP; if the specified condition (2) is true then execution continues with the statement following the UNTIL <condition>.

Condition

<condition> is one of: CC, CS, EQ, GE, GT, HI, HS, LE, LO, LS, LT, MI, NE, PL, VC, and VS.

CC - carry clear (carry flag clear) (identical to HS)
CS - carry set (carry flag set) (identical to LO)
EQ - equal (zero flag set)
GE - greater than or equal to zero (negative and overflow flags either both set or both clear)
GT - greater than zero (zero flag clear and negative and overflow flags either both set or both clear)
HI - higher (carry and zero flags both clear) (unsigned greater than)
HS - higher or same (carry flag clear) (identical to CC) (unsigned greater than or equal to)

LE - less than or equal to zero (zero flag set or
negative or overflow flag set but not both)
LO - lower (carry flag set) (identical to CS)
(unsigned less than)
LS - lower or same (carry flag and/or zero flag set)
(unsigned less than or equal to)
LT - less than zero
(negative or overflow flag set but not both)
MI - minus (negative flag set)
NE - not equal (zero flag clear)
PL - plus (negative flag clear)
VC - overflow clear (overflow flag clear)
VS - overflow set (overflow flag set)

NOTE: Anywhere in the previous examples where there is one QUIF <condition>, there could be two or more QUIF <condition>'s.

Example:

```
GUESS
.
.
.
QUIF CC
.
.
.
QUIF EQ
.
.
.
ENDGUESS
```

Chapter 6

LINKER

The Waterloo 6809 Linker links together an arbitrary number of relocatable object modules and produces an absolute executable load module. Subroutines from the system runtime library may also be linked in.

The Linker Command File

The linker requires a '.cmd' file which the user must create using the editor. The linker commands available for the '.cmd' file are: bank, bankorg, banksize, export, include, org, and printalv. Comments may be included provided that they are preceded by semi-colons.

The Load Module Name

The first line of this linker command file is the quoted name of the load module to be created.

"loadname"

The name of the load module to be created is 'loadname'.

The ORG Command

The ORG command indicates the address in main memory where the module is to be loaded. This line contains 'org' followed by an address such as '$1000'.

org $1000

The module is to be loaded at hexadecimal address 1000.

Each object module which is to be linked into the load module is specified by a line giving the filename in quotes.

"obj1.b09"
"obj2.b09"

Two object modules are to be linked into the load module which, at load time, will be loaded into main memory at the address specified by the 'org' line.

The BANKSIZE and BANKORG Commands

There is 32K of main memory. As well, there are sixteen 4K 'banks' of memory which provide an additional 64K. Each bank is addressed by the hexadecimal addresses 9000 through 9FFF and only one bank can be accessed at any one time. The BANKSIZE command specifies the size of the banks and the BANKORG command specifies their origin in memory. These commands are optional since the only values that can be specified are the default values: $1000 for the size and $9000 for the origin.

banksize $1000
bankorg $9000

The BANK Command

Routines may be loaded into a particular bank by using the BANK command whose format is the word 'bank' followed by a decimal number from 0 to 15. Any routines which are to be loaded into main memory should all be specified before the bank command. After the bank command, all loading will be into the designated bank until another bank command is encountered.

```
bank 1
"obj4.b09"
"obj5.b09"
bank 12
"obj6.b09"
```

Two routines are to be loaded into bank 1 and one routine is to be loaded into bank 12.

The user is responsible for making sure that the routines for each bank will fit into the allotted 4K. In making this decision, 'auto-load vectors' may also have to be taken into account. Since only one bank can be accessed at a time, one bank can not directly access another bank. Instead, there is an indirect reference created automatically by the linker using an auto-load vector. A bank's auto-load vectors contain the address and bank numbers of all routines referenced which are located in another bank. The PRINTALV command will cause the auto-load vector informationto be displayed in the '.map' file during the linking process.

The INCLUDE Command

The INCLUDE command is used in two instances.

(1) If any routines from the system library are needed, the following line must be included.

```
include "disk/1.watlib.exp"
```

The system disk is assumed to be on a diskette in drive 1 of the disk drive device; thus, 'disk/1.' is designated.

(2) A user's personal library of routines may be linked into the load module by a line containing the word 'include' followed by the quoted name of a library file.

```
include "personal"
```

The library file may contain a list of the object modules for the library routines or any other linker commands.

If "personal" contains the lines

"rout1.b09"
"rout2.b09"

then the line 'include "personal"' in the '.cmd' file causes "rout1.b09" and "rout2.b09" to be linked into the load module.

The EXPORT Command

The EXPORT command specifies a name and an address in memory with which this name is to be associated. This location can be accessed by any routine in the load module which has xref'ed the associated name.

export glovar = $0300

Location $0300 can now be accessed by any routine which contains the line 'xref glovar'. For example, routine A could store a value at glovar and this value could then be used later on by routine B.

The Linking Process

In order to start the linking process, enter 'l' when selecting from the menu.

When asked to 'Enter filename:' enter the name of the linker command file without including the assumed '.cmd' suffix.

The linking will then be done. The 'Linker completed' message indicates the end of the linking process. Pressing 'RETURN' will cause a return to the menu. The linker produces two or three new files with names identical to the load module name specified in the first line of the '.cmd' file, except for added suffixes.

The '.mod' file is the executable load module which is to be loaded and executed by the monitor.

The '.map' file contains information showing how the object modules are mapped into the load module.

The '.exp' file contains the names and addresses of any routines which are xref'ed in the source program. If a system library routine is xref'ed then the '.cmd' file must contain an 'include "disk/1.watlib.exp"' line. This line will cause all the system library routines to be included in the '.exp' file. A '.exp' file will not be created if there are no export routines (and if the '.cmd' file does not contain the 'include "disk/1.watlib.exp"' line).

Chapter 7

MONITOR

The monitor primarily serves to load and execute load modules but is useful in debugging programs as well. There are several monitor commands: b (bank), c (clear), d (dump), f (fill), g (go), l (load), m (modify), p (passthrough), q (quit), r (registers), s (stop), and t (translate).

In order to enter the monitor, select 'm' from the menu.

Whenever the monitor is ready to receive a command, it prompts with a right angle bracket ('>').

In order to load a program, enter the command 'l' followed by the name of the load module (the '.mod' file).

```
>l prog.mod
```

This will load the module into memory at the address specified by the 'org' line in the '.cmd' file used by the linker to create the load module.

In order to run the program, enter the command 'g' followed by the address in hexadecimal of the loaded module.

>g 1000

This will start execution at hexadecimal address 1000 which is where the program's first instruction should be located.

When the program is finished executing, control should be returned to the monitor. Entering the command 'q' will cause a return to the menu.

>q

The following paragraghs list the monitor commands.

Bank

b <n>

There are 16 banks. <n> should be a hexadecimal number between 0 and F.

>b a

Bank 10 can now be accessed by such commands as d<ump> and m<odify>.

Clear

c

'c' clears all breakpoints. (Breakpoints are set by s<top>.)

>c

Dump

d <xxxx>
or
d <xxxx>-<yyyy>
or
d <xxxx>.<yy>

The dump command always displays 8 bytes per line. The number of bytes displayed will therefore often be greater than the number of bytes requested.

'd <xxxx>' will display the hexadecimal contents of the hexadecimal addresses from <xxxx> to (<xxxx> + 7).

>d 1000

The contents of the bytes from 1000 to 1007 will be displayed.

'd <xxxx>-<yyyy>' will display the hexadecimal contents of the hexadecimal addresses from <xxxx> to <yyyy>.

>d 1000-1020

The contents of the 40 bytes from 1000 to 1027 will be displayed.

'd <xxxx>.<yy>' will display the hexadecimal contents of the hexadecimal addresses from <xxxx> to (<xxxx> + <yy>).

>d 30.28

The contents of the bytes from 0030 to at least 0058 will be displayed.

Eight bytes per line followed by the ASCII translation are displayed. The contents of memory may be modified by cursoring to the value of a byte displayed on the screen and typing over it with a new value. After a dump line has been changed, 'RETURN' must be pressed in order for the changes in that line to be entered.

Fill

f <xxxx>-<yyyy> <aa>
or
f <xxxx>.<yy> <aa>

'f <xxxx>-<yyyy> <aa>' will fill the hexadecimal addresses from <xxxx> to (<yyyy>) with the hexadecimal value <aa>.

>f 1000-1020 0

The 32 bytes from 1000 to 1020 will be filled with 00.

'f <xxxx>.<yy> <aa>' will fill the hexadecimal addresses from <xxxx> to (<xxxx> + <yy> - 1) with the hexadecimal value <aa>.

>f 1000.30 ff

The 48 bytes from 1000 to 102f will be filled with ff.

Go

g <xxxx>

This will start the execution of instructions in sequential order beginning with the instruction at hexadecimal address <xxxx>. If no <xxxx> address is specified, the default is the current value of the program counter. Execution will continue until a software or hardware interrupt occurs to return control to the monitor. The g instruction will usually be preceded at some point by a l<oad> instruction which loads a program into memory.

>g 1000

Load

l <loadmodule>

This will load the specified load module into memory at the address designated by the 'org' line in the '.cmd' file used by the linker in creating the load module. In order to run the program, the g<o> command will then need to be entered.

>l prog. mod

Modify

m <xxxx> <aa>,<bb>,<cc>,...

This will modify memory beginning with the hexadecimal address <xxxx>. Address <xxxx> will be set to the value <aa>; address (<xxxx> + 1) will be set to the value <bb>; address (<xxxx> + 2) will be set to the value <cc>; et cetera.

>m 20 22 ff

The byte at address 0020 will be set to 22 and the byte at address 0021 will be set to ff.

Passthrough

p

The p command causes the system to go into terminal passthrough mode. All input which is entered will be sent to the host computer. All output from the host computer will be displayed on the screen. To get out of host passthrough mode, the 'STOP' key must be pressed.

>p

Quit

q

To get out of the monitor, the q command must be entered. This will cause a return to the menu.

>q

Registers

r

The r command will display the contents of the registers on the screen. The registers displayed are the PC (program counter), D (accumulator D), X (index register X), Y (index register Y), U (user stack pointer), S (system stack pointer), CC (condition code register), and DP (direct page register). The contents of a register may be modified by cursoring to the displayed value, typing over it, and pressing 'RETURN'.

>r

Stop

s
or
s <xxxx>

's' will display the hexadecimal addresses at which any breakpoints have been set.

>s

's <xxxx>' will set a new breakpoint at hexadecimal address <xxxx> and will also display the hexadecimal addresses at which any breakpoints are set.

>s 1111

A breakpoint will be set at 1111.

If a breakpoint has been set and then the g<o> command is entered to run a program, execution will stop just before the instruction at the breakpoint address is executed. Control will be returned to the monitor and the contents of the registers will be displayed. Setting a breakpoint is like inserting a software interrupt at the breakpoint address. A maximum of four breakpoints can be set at one time. The c<lear> command will clear any breakpoints set.

Translate

t <xxxx>
or
t <xxxx>-<yyyy>
or
t <xxxx>.<yy>

't <xxxx>' will display the assembly instruction contained in the hexadecimal address <xxxx>.

>t 300
The assembly instruction at 0300 will be displayed.

't <xxxx>-<yyyy>' will display the assembly instructions contained in the hexadecimal addresses from <xxxx> to (<yyyy>).

>t 1000-1200

The assembly instructions between 1000 and 1200 will be displayed.

't <xxxx>.<yy>' will display the assembly instructions contained in the hexadecimal addresses between <xxxx> and (<xxxx> + <yy> - 1).

>t 1000.30

The assembly instructions between 1000 and 102f will be displayed.

Chapter 8

SYSTEM LIBRARY REFERENCE MANUAL

All routines in the system library have names ending in an underbar (for example, ISALPHA_). On the keyboard, the underbar is a left-pointing arrow.

The first parameter is always passed in accumulator D. Other parameters are passed on the stack with the (n)th parameter being pushed on before the (n-1)st parameter. For example, before calling a routine that expects three parameters, the third parameter P3 must be pushed on the stack followed by the second parameter P2 and the first parameter P1 must be placed in accumulator D. Two bytes per parameter must be pushed onto the stack even if the parameter could be contained in one byte.

Any parameters pushed on the stack must be pulled off the stack by the calling program. (The LEAS command is useful for this.) Parameters should not be reused because some routines will modify them.

Results are always passed back in accumulator D.

Strings are a collection of characters terminated by a null byte. A null byte is represented by 00.

FALSE is represented by a zero value while TRUE is represented by a non-zero value. System routines which return TRUE/FALSE results set the condition codes appropriately so the calling program can perform EQ/NE tests.

Manipulation of Character Strings and Numbers

There are a large number of routines to manipulate strings and to convert strings to numbers and numbers to strings.

ISALPHA_

ISALPHA_ checks P1 to see if it is an alphabetic character and, if it is, returns TRUE. If P1 is not an alphabetic character, FALSE is returned.

P1 - character to check

Result - TRUE/FALSE

ISDELIM_

ISDELIM_ checks P1 to see if it is a delimiter character and, if it is, returns TRUE. If P1 is not a delimiter character, FALSE is returned. A delimiter is any character which is neither alphabetic nor numeric.

P1 - character to check

Result - TRUE/FALSE

ISDIGIT_

ISDIGIT_ checks P1 to see if it is a numeric character and, if it is, returns TRUE. If P1 is not a numeric character, FALSE is returned.

P1 - character to check

Result - TRUE/FALSE

ISLOWER_

ISLOWER_ checks P1 to see if it is a lower-case alphabetic character and, if it is, returns TRUE. If P1 is not a lower-case alphabetic character, FALSE is returned.

P1 - character to check

Result - TRUE/FALSE

ISUPPER_

ISUPPER_ checks P1 to see if it is an upper-case alphabetic character and, if it is, returns TRUE. If P1 is not an upper-case alphabetic character, FALSE is returned.

P1 - character to check

Result - TRUE/FALSE

LOWER_

LOWER_ converts a character specified by P1 to lower-case.

P1 - character to convert to lower-case

Result - lower-case character

UPPER_

UPPER_ converts a character specified by P1 to upper-case.

P1 - character to convert to upper-case

Result - upper-case character

ZLOSTR_

ZLOSTR_ converts the characters in the string addressed by P1 to lower-case.

P1 - address of the string

ZUPSTR_

ZUPSTR_ converts the characters in the string addressed by P1 to upper-case.

P1 - address of the string

COPY_

COPY_ copies memory as addressed by P1 to memory addressed by P2 for a length specified by P3.

P1 - address of the memory to be copied
P2 - address of the destination in memory
P3 - length to copy

COPYSTR_

COPYSTR_ copies the string addressed by P1 to memory addressed by P2.

P1 - address of the string to be copied
P2 - address of the destination in memory

PREFIXST_

PREFIXST_ adds a prefix string addressed by P1 to a string addressed by P2.

P1 - address of the prefix string
P2 - address of the string to be prefixed

SUFFIXST_

SUFFIXST_ adds a suffix string addressed by P1 to a string addressed by P2.

P1 - address of the suffix string
P2 - address of the string to be suffixed

EQUAL_

EQUAL_ compares memory as addressed by P1 to memory as addressed by P2. TRUE is returned if there is equivalence for the length specified by P3; otherwise, FALSE is returned.

P1 - first memory address
P2 - second memory address
P3 - number of bytes to compare

Result - TRUE/FALSE

LENGTH_

LENGTH_ calculates the length of the string addressed by P1 and returns the number of characters in the string.

P1 - address of the string

Result - number of characters in the string

STREQ_

STREQ_ compares the string addressed by P1 to the string addressed by P2. TRUE is returned if they are equivalent for the length of the P1 string; otherwise FALSE is returned.

P1 - address of the first string
P2 - address of the second string

Result - TRUE/FALSE

BTOHS_

BTOHS_ converts a binary number addressed by P1 to a hexadecimal string addressed by P3. The number of bytes in the binary value is specified by P2.

P1 - address of the binary number
P2 - number of bytes in the binary value
P3 - address for the hexadecimal string

HSTOB_

HSTOB_ converts a hexadecimal string addressed by P1 to a binary number addressed by P2. The result returned is the number of bytes in the binary value.

P1 - address of the hexadecimal string
P2 - address for the binary number

Result - number of bytes in the binary value

ITOHS_

ITOHS_ converts an integer specified by P2 to a hexadecimal string addressed by P1.

P1 - address for the hexadecimal string
P2 - integer to convert

ITOS_

ITOS_ converts an integer specified by P2 to a decimal string addressed by P1.

P1 - address for the decimal string
P2 - integer to convert

STOL_

STOL_ converts a decimal string addressed by P1 to an integer. The decimal string may include a plus or minus sign but not a decimal point.

P1 - address of the decimal string

Result - binary representation of the integer

DECIMAL_

DECIMAL_ converts a decimal number addressed by P1 to an integer. P2 specifies the number of digits in the decimal number. The decimal number may only contain digits. It may not include plus signs, minus signs, or decimal points.

P1 - address of the decimal number
P2 - number of digits in the decimal number

Result - binary representation of the integer

HEX_

HEX_ converts a hexadecimal character specified by P1 to a hexadecimal byte.

P1 - hexadecimal character (0-9, A-F)

Result - binary representation of the hexadecimal character

Input/Output Routines

There are various system library routines provided for input/output. Routines such as GETCHAR_, PUTCHAR_, and PUTNL_ are for input from the standard input device and output to the standard output device. Before using those routines, the INITSTD_ routine should be called to initialize the standard devices for input/output.

Routines like FGETCHAR_, FPUTCHAR_, and FPUTNL_ are for file I/O. Before getting input from or sending output to a file, the file must be opened by the OPENF_ routine and, afterwards, the file must be closed by the CLOSEF_ routine. OPENF_ must be provided with the address of a filename string and the address of an access mode character and will return a pointer to a file control block. Any access of the file for input, output, or closing must refer to this file control block.

INITSTD_

INITSTD_ initializes the standard input and output devices for I/O.

GETCHAR_

GETCHAR_ gets a character from the terminal.

Result - input character

PUTCHAR_

PUTCHAR_ puts a character specified by P1 to the screen.

P1 - character to be output

PUTNL_

PUTNL_ causes a skip to a new line on the screen.

PRINTF_

PRINTF_ provides a means of formatting output to the screen. P1 is the address of the string to be displayed. This string can contain the following in order to cause special formatting:

%n - new line
%c - character
%d - decimal number
%h - hexadecimal number
%s - string

Wherever %n, %c, %d, %h, or %s occurs in the string, a substitution value is to be inserted in its place. %n causes a skip to a new line. %c, %d, %h, and %s must have substitution values specified by the user. Up to six such substitution values, P2 through P7, may be specified. The substitution value specified for %c must be a character; the substitution values specified for %d and %h must be integer numbers; and the substitution value for %s must be the address of a string.

P1 - format string to display
P2 - substitution value
P3 - "
P4 - "
...
P7 - substitution value

GETREC_

GETREC_ gets a record from the terminal and stores it in a buffer addressed by P1. P2 specifies the length of the buffer. The result is the number of characters read.

P1 - address of the record buffer
P2 - length of the record buffer

Result - number of characters read

PUTREC_

PUTREC_ puts a record addressed by P1 out to the screen. The length of the record is specified by P2.

P1 - address of the record buffer
P2 - length of the record buffer

OPENF_

OPENF_ opens a file whose filename string is addressed by P1. ("printer" and "keyboard" are examples of filename strings. For more information, see the System Overview: Commodore SuperPET manual.) The mode for which the file is to be opened is addressed by P2. The mode is to be one of:

"R" - read
"W" - write
"U" - update
"A" - append
"S" - store (write PRG format files)
"L" - load (read PRG format files)

If the open succeeded, OPENF_ returns the address of the file control block. If the open failed, OPENF_ returns zero.

P1 - address of the filename string
P2 - address of the file mode string
(Only one character is significant.)

Result - address of the file control block
If the open failed, the result is 0.

CLOSEF_

CLOSEF_ closes a file whose file control block is addressed by P1.

P1 - address of the file control block

FGETCHAR_

FGETCHAR_ gets a character from a file whose file control block is addressed by P1.

P1 - address of the file control block

Result - input character

FPUTCHAR_

FPUTCHAR_ puts a character specified by P2 out to a file whose file control block is addressed by P1.

P1 - address of the file control block
P2 - character to be output

FPUTNL_

FPUTNL_ causes a skip to a new record in a file whose file control block is addressed by P1.

P1 - address of the file control block

FPRINTF_

FPRINTF_ provides a means of formatting output to a file whose file control block is addressed by P1. P2 is the address of the string to be output. This string can contain the following in order to cause special formatting:

%n - new line
%c - character
%d - decimal number
%h - hexadecimal number
%s - string

Wherever %n, %c, %d, %h, or %s occurs in the string, a substitution value is to be inserted in its place. %n causes a skip to a new record. %c, %d, %h, and %s must have substitution values specified by the user. Up to six such substitution values, P3 through P8, may be specified. The substitution value specified for %c must be a character; the substitution values specified for %d and %h must be integer numbers; and the substitution value for %s must be the address of a string.

P1 - address of the file control block
P2 - format string to output
P3 - substitution value
P4 - "
P5 - "
...
P8 - substitution value

FGETREC_

FGETREC_ gets a record from a file whose file control block is addressed by P1 and stores it in a buffer addressed by P2. The length of the buffer is specified by P3. The result is the number of characters read.

P1 - address of the file control block
P2 - address of the record buffer
P3 - length of the record buffer

Result - number of characters read

FPUTREC_

FPUTREC_ puts a record addressed by P2 out to a file whose file control control block is addressed by P1. The length of the record is specified by P3.

P1 - address of the file control block
P2 - address of the record buffer
P3 - length of the record buffer

FSEEK_

FSEEK_ seeks to a record in a random file whose file control block is addressed by P1. The record number is specified by P2.

P1 - address of the file control block
P2 - record number

EOF_

EOF_ checks a file whose file control block is addressed by P1 to see if it is at end-of-file. If the file is at end-of-file, TRUE is returned; otherwise, FALSE is returned.

P1 - address of the file control block

Result - TRUE/FALSE

EOR_

EOR_ checks a file whose file control block is addressed by P1 to see if it is at end-of-record. If the file is at end-of-record, TRUE is returned; otherwise, FALSE is returned.

P1 - address of the file control block

Result - TRUE/FALSE

ERRORF_

ERRORF_ gets the error code for a file whose file control block is addressed by P1. The code returned is one of:

0 - success
2 - end-of-file
3 - error

P1 - address of the file control block

Result - error code for the file (0, 2, or 3)

ERRORMSG_

ERRORMSG_ returns the address of an I/O error message when an error has occurred. (ERRORF_ may be called to see whether or not there has been an error.)

Result - address of the I/O error message string

MOUNT_

MOUNT_ mounts a new diskette whose device string is addressed by P1.

P1 - address of the device string

RENAMEF_

RENAMEF_ renames a file whose filename string is addressed by P1. The new filename string is addressed by P2.

P1 - address of the old filename string
P2 - address of the new filename string

SCRATCHF_

SCRATCHF_ scratches a file whose filename string is addressed by P1.

P1 - address of the filename string

DIRCLOSEF_

DIRCLOSEF_ closes a directory whose file control block is addressed by P1.

P1 - address of the file control block

DIROPENF_

DIROPENF_ opens a directory whose filename string is addressed by P1 and returns the address of the file control block.

P1 - address of the filename string

Result - address of the file control block

DIRREADF_

DIRREADF_ reads a directory entry from a directory whose file control block is addressed by P1. The buffer into which to read the directory entry is addressed by P2.

P1 - address of the file control block
P2 - address of the directory entry buffer

TABSET_

TABSET_ sets ten tabulation stops. P1 is the starting address of ten words containing the column positions for the tabs.

P1 - address of the ten words containing column positions

TABGET_

TABGET_ returns the address of ten words containing the column positions for ten tab stops.

Result - address of the ten words containing tab settings

KBDISABL_

KBDISABL_ disables the keyboard scan. (It may be reenabled by KBENABL_.)

KBENABL_

KBENABL_ enables the keyboard scan. (If the keyboard scan has been disabled by KBDISABL_, KBENABL_ will reenable it.)

PASSTHRU_

PASSTHRU_ puts the terminal into host passthrough mode. Depressing the 'STOP' key will cause the routine to return.

Terminal and Serial Input/Output Routines

TBREAK_

TBREAK_ tests the terminal's 'STOP' key to see if it has been depressed. If it is has been depressed, TRUE is returned. If it has not been depressed, FALSE is returned.

Result - TRUE/FALSE

TGETCURS_

TGETCURS_ returns the current position of the cursor on the screen. The high order byte of the result specifies the row and the low order byte specifies the column.

Result - cursor position (MS byte = row & LS byte = column)

TPUTCURS_

TPUTCURS_ sets the cursor positionto the position specified by P1. The high order byte of P1 specifies the row and the low order byte specifies the column.

P1 - cursor position (MS byte = row & LS byte = column)

TSETCHAR_

TSETCHAR_ sets the screen's character set. If the value is 1, the screen is to display the ASCII character set. If the value is 2, the screen is to display the APL character set.

P1 - 1 (for ASCII) or 2 (for APL)

SIOINIT_

SIOINIT_ initializes a serial port addressed by P1 for input and/or output using the speed, parity, and stopbits characteristics specified by P2, P3, and P4 respectively.

P1 - address of the serial port
P2 - speed of the serial port
P3 - parity: even = $60
odd = $20
mark = $A0
space = $E0
P4 - stopbits: one = $00
two = $80

SBREAK_

SBREAK_ tests for a break from a serial port addressed by P1. If there is a break then TRUE is returned; otherwise, FALSE is returned.

P1 - address of the serial port

Result - TRUE/FALSE

Date and Time Routines

SETDATE_

SETDATE_ sets the date to the date string addressed by P1.

P1 - address of the date string

GETDATE_

GETDATE_ returns the address of a string containing the date. (The date should previously have been set by SETDATE_.)

Result - address of the date string

SETTIME_

SETTIME_ sets the time to the four bytes addressed by P1. The four bytes are to contain the time in hours, minutes, seconds, and sixtieths of a second.

P1 - address of the four time bytes

GETTIME_

GETTIME_ returns the time in the four bytes addressed by P1. The four bytes are to contain the time in hours, minutes, seconds, and sixtieths of a second. (The time should previously have been set by SETTIME_.)

P1 - address to put the time

Miscellaneous Routines

TABLELOO_

TABLELOO_ looks up a sequence of characters addressed by P2 in a table addressed by P1 and returns the position of the sequence in the table. If the sequence of characters is not found then zero is returned. The length of the sequence to be looked up is specified by P3. A table is a collection of strings terminated by a null string. Each string in the table is comprised of lower case characters followed by optional upper case characters. The characters being looked up are case insensitive. In order for a sequence of characters to match a string in the table, its length must be at least as great as the number of lower case characters in the string and each of its characters must match the corresponding character in the string. Thus, if a table contained the string "abcDEF", the following sequences would match that string: abc, ABCD, abcDE, and ABCDef where the case of the characters in those sequences does not matter. The sequences A, ab, and aBcDeFg are examples of sequences that would not match.

P1 - address of the table
P2 - address of the sequence of characters to look up
P3 - length of the sequence of characters to look up

Result - position of the word in the table
If the word was not found, the result is 0.

BANKINIT_

BANKINIT_ performs initializations allowing bank-switching to be used.

CONBINT_

CONBINT_ creates a connectionto an interrupt processing routine addressed by P1. (The interrupt routine may be in bank switched storage.) The type of interrupt to be handled is specified by P2 where:

2 = SWI3
4 = SWI2
6 = FIRQ
8 = IRQ
10 = SWI
12 = NMI

P1 - address of the interrupt processing routine
P2 - type of interrupt (2, 4, 6, 8, 10, or 12)

Chapter 9

RESERVED WORDS

There are several words in the 6809 assembly language which have special meanings attached to them and, thus, are referred to as reserved words. They should not be used in any context other than that indicated by the language. For example, should a user-defined label have the same name as a reserved word, the results are unpredictable and it is highly unlikely that the program will work as intended.

A	ABX	ADCA	ADCB
ADDA	ADDB	ADDD	ADMIT
ANDA	ANDB	ANDCC	ASL
ASLA	ASLB	ASLD	ASR
ASRA	ASRB		
B	BCC	BCS	BEQ
BGE	BGT	BHI	BHS
BITA	BITB	BLE	BLO
BLS	BLT	BMI	BNE
BPL	BRA	BRN	BSR
BVC	BVS		

CC	CCR	CLC	CLI
CLR	CLRA	CLRB	CLV
CMPA	CMPB	CMPD	CMPS
CPMU	CMPX	CMPY	COM
COMA	COMB	CPX	CS
CWAI			
D	DAA	DEC	DECA
DECB	DES	DEX	DPR
DSCT			
ELSE	END	ENDC	ENDGUESS
ENDIF	ENDLOOP	ENDM	EORA
EORB	EQ	EQU	EXG
FAIL	FCB	FCC	FDB
GE	GT	GUESS	
HI	HS		
IF	IFC	IFEQ	IFGE
IFGT	IFLE	IFLT	IFNC
IFNE	INC	INCA	INCB
INS	INX		
JMP	JSR		
LBCC	LBCS	LBEQ	LBGE
LBGT	LBHI	LBHS	LBLE
LBLO	LBLS	LBLT	LBMI
LBNE	LBPL	LBRA	LBRN
LBSR	LBVC	LBVS	LDA
LDAA	LDAB	LDB	LDD
LDS	LDU	LDX	LDY
LE	LEAS	LEAU	LEAX
LEAY	LIST	LO	LOOP
LS	LSL	LSLA	LSLB
LSLD	LSR	LSRA	LSRB
LSRD	LT		
MACR	MI	MUL	
NAM	NE	NEG	NEGA
NEGB	NOLIST	NOP	
OPT	ORA	ORAA	ORAB
ORB	ORCC	ORG	
PAGE	PC	PL	PSCT
PSHA	PSHB	PSHS	PSHU
PSHX	PULA	PULB	PULS
PULU	PULX		
QUIF			

RMB	ROL	ROLA	ROLB
ROR	RORA	RORB	RTI
RTS			
S	SBCA	SBCB	SEC
SEI	SET	SEV	SEX
STA	STAA	STAB	STB
STD	STS	STU	STX
STY	SUBA	SUBB	SUBD
SWI	SYNC		
TAB	TAP	TBA	TFR
TPA	TST	TSTA	TSTB
TSX	TXS	TTL	
U	UNTIL		
VC	VS		
X	XDEF	XREF	
Y			

Commodore Magazine

This bi-monthly magazine, published by Commodore, provides a vehicle for sharing the latest product information on Commodore systems, programming techniques, hardware interfacing, and applications for the CBM, PET, SuperPET, and VIC Systems. Each issue contains user application features, columns by leading experts, the latest news on user clubs, a question/answer hotline column, and reviews of the latest books and software.

The subscription fee is $15.00 for six issues per year within the U.S. and its possessions, and $25.00 for Canada and Mexico. Make checks payable to COMMODORE BUSINESS MACHINES, and send to:

Editor, Commodore Magazine
Commodore Business Machines, Inc.
681 Moore Road
King of Prussia, PA 19406

The Transactor

The Transactor, which is a monthly publication of Commodore-Canada, is primarily a technical periodical, containing pertinent hardware and software information for the CBM, PET, VIC, and SuperPET systems. Each issue features product reviews, hardware and software evaluations, and programming tips from the finest technical experts on Commodore products. Additionally, The Transactor contains general information such as product updates and trade show reports.

The subscription fee is $10.00 for six issues within Canada and the United States, and $13.00 for all foreign countries. Make checks payable to COMMODORE BUSINESS MACHINES, INC. and send to:

Editor, The Transactor
Commodore Business Machines, Inc.
3370 Pharmacy Avenue
Agincourt, Ontario, Canada M1W 2K4

This manual describes the Waterloo 6809 Assembler, Linker and Monitor systems. It provides all the details necessary to develop and debug programs written in the 6809 assembly language for the Commodore SuperPET Particular features described in the book include:

- The Motorola 6809 microcomputer system architecture found in the Commodore SuperPET including the details of the bank-switched memory
- The assembly language instructions and directives
- The Structured Programming statements in the language
- The Waterloo 6809 linker, including the details of how to produce programs residing in the bank-switched memory
- The Waterloo 6809 monitor which is used to interactively debug programs
- The Waterloo library of routines which is resident in the read-only memory and may be called to perform common functions

The book is organized into two sections. The first section contains a collection of annotated examples to be used as a tutorial. The second part contains the comprehensive details of the 6809 Assembler and development systems.

DISTRIBUTED BY

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$10.95/21908

ISBN: 0-672-21908-5