K-OS ONE Operating System

For the 68000

Hawthorne Technology

THE K-OS ONE OPERATING SYSTEM

Distribution Package

Hawthorne Technology
8836 S.E. Stark
Portland, Or 97216

(503) 254-2005

* * * PRODUCT DISCLAIMER * * *

* * * * * * * * * * * * * *

K-OS ONE OPERATING SYSTEM PACKAGE

MANUAL LAYOUT

K-OS ONE OPERATING SYSTEM

INSTALLATION

Hawthorne Technology
8836 S. E. Stark
Portland, Or 97216

(503) 254-2005

INSTALLATION OF K-OS ONE

These instructions tell how to go about installing the K-OS ONE operating system. This can be done on a system that is already running an HTPL compiler, or on a system that has some sort of monitor ROM.

In bringing up the K-OS ONE operating system, there are two different scenarios possible. The first is bringing it up on a completely clean machine that has no operating system, and where you do not have access to an operating version of the K-OS ONE operating system. The second case is where the initial version has been installed and is operational, or where there is access to an operating version of K-OS ONE.

SYSTEM REQUIREMENTS

The hardware requirements of the K-OS ONE system are minimal. There are three things that are needed.

A 5 1/4 inch double sided, double density, floppy disk drive is necessary to read the operating system and to read the system file. The operating system that is supplied assumes that the drive will be an IBM compatible, 360K, DSDD.

The second thing that is needed is a console device that can read and write characters to the command console.

The third thing is having access to the trap one interrupt.

To bring up the operating system originally from binary, the binary copy is loaded into memory and the necessary routines are patched into the hardware, or applied into PROM. After the binary copy is loaded, vectors to these services are placed in fixed locations that are relative to the beginning of where the operating system is loaded. The operating system is position independent, and assumes that vectors to these services will be located immediatly before its location.

After the binary copy of the operating system is loaded, and the necessary vectors are patched in, you can begin execution of the operating system by branching to the location at which it was loaded.

At this point, the user has a system with a single floppy disk, that has a minimal RAM disk, and a console device.

Now it is possible to edit and modify the HTPL source code of the operating system, and the assembly language operating system run time support, so that other drivers and other disks can be patched in. At this point the user can begin to build a customized system for their particular hardware. Other devices and disks can be added.

Bringing up the binary operating system involves studying the requirements of a driver for the generalized system. Sample drivers are provided in assembly language or in HTPL to provide examples for writing routines that provide these functions.

On the operating system itself, there are three areas where linkages to the underlying hardware have to be made and in the production version of the operating system, these are found in a small include file that is labeled OSRTL.HEX. This is a program that was written in assembly language in the format prescribed for linkable object in the HTPL section of the manual. It was assembled using the assembler and then linked in when the operating system was re-compiled.

These three areas are:
  1. The trap one linkage to the underlying hardware.
    a. Initialization of a set of register locations so that when a trap one is executed by a user program or by the command processor, the linkage can be achieved back to the HTPL program. This is necessary because when the HTPL program is executing, certain registers have to be pointing to certain areas.

    b. The trap one processor itself. You must save the user's registers so that a return from a utility user program, the registers used by the HTPL operating system are loaded and a branch is made to the entry point in the operating system itself. After the branch is made to the entry point of the operating system, the HTPL program will take care of servicing what ever request is made from the operating system. It will exit through an assembly language routine which then restores the user registers as they were when the operating system was invoked. Because the operating system uses a parameter block in memory, the registers of the user program don't have to be set up. The only register that is passed from the user program to the operating system is register A0. A copy of A0 is made and is pushed on the evaluation stack which is register A4.

  2. Character device drivers for user devices such as printers or CRTs which normally communicate a character at a time. These are sometimes refered to as stream devices. There are 5 entry points for each stream device. For many devices, some of the entry points are not needed and can be dummy values. They are used to maintain status and don't perform any function.

  3. Block addressable devices, commonly disk or something where a portion of the device can be addressed and where data transfer occurs in blocks. The standard block size is 512 bytes. .pa
There are 5 entry points on each device.

1. Get a character or byte from the device.
2. Put a character or byte to the device.
3. Initialize the device.
4. Get the status of the device.
5. Send a control action to the device.

The operating system assumes that these will be simple status loop type devices where after it is called, it will wait until a character is ready, get the character, and return to the operating system or to the function that called it.

The purpose of the status call is primarily to find out if the character is available before the character input routine is called. This way, if no character is available, the system will not get tied up in some kind of a loop.

On a printer, the reason for the status is to determine if the device would be capable of accepting a character so that a time out can be created in the event that a write to printer is attempted when the printer is not ready.

The different entry points for bringing up the binary copy of the operating system are:

1. Receiving a character or byte.
2. Sending a character or byte.
3. Initialize a driver.

The status is not required for the simple command processor. It only requires the first three entry points.

There is a routine to read a character from a printer. In most cases this will be a dummy that won't return anything, but it simplifies the coding by making all of the stream devices alike.

The actual device driver can be implemented by a status and wait loop for systems that don't require high performance, or it can be done through an interrupt driven device where characters are placed in a queue and are returned as needed. In the case of a character queue, the status call reports whether or not a character is in the queue waiting to be delivered.

In most cases, when a character is received, the parity is striped and 7 bit ASCII is used. When characters are sent to a device, 8 bits are sent so graphics type devices or graphics characters can be used.

For a disk device there are 5 entry points.

1. Read a record.
2. Write a record.
3. Get disk status.
4. Control the disk.
5. Flush disk buffers.

The routines that work with the disk type devices or block devices, assume a record transfer size of 512 bytes. This is the physical block size for the standard MS-DOS 360K disk. If the physical block size on the device is other than 512 bytes, the disk driving routine has to translate the actual physical block to a 512 byte block.

The disk routines assume contiguous disk with records starting at zero and going to the maximum number on the disk. As a result of this, it is the responsability of the low level disk drive routine to determine the number of sectors per track, the number of sides, and to make the appropriate translation of a logical block to a track and sector number that can be read or written.

The disk status is primarily used to determine whether or not the media has been changed since the last call. On a floppy disk, this will usually be reported as not available. On a hard disk, it will generally be reported that the media has not been changed. The purpose of this on a floppy disk is that when a directory read or write is required, such as when opening or closing a file, or allocating space, it can be checked to see if the disk is always the same. This will cause the FAT (file allocation table), and the directory to be reloaded from the disk each time.

For a higher through-put, (though slightly less secure), the floppy disk can report that the media has not changed. This would mean that the FAT and the directory would not have to be read each time, but in the case that the diskette had actually been changed, the data would be fouled up.

The disk control on the present version of the operating system is not being used.

Flushing the buffers is not being used in the current version of the operating system. The reason for flushing the buffers is to inform the disk driving routine that the buffers should no longer be considered available. This would be used when the physical size is not 512 bytes.

The initialization routine is normally called on power up or when specifically requested by a user program. The purpose of this is to define the characteristics (number of start bits, stop bits) of the device as is necessary on most micro-processor UARTs.

Because the allocation of data on an MS-DOS disk is complex, it may be convenient to prepare a less complex disk. If a disk is formatted without a volume name and no system files, the first file will start at sector 12. If there is only one file on the disk, and the disk was empty when the file was placed there, the sectors for the file will be contiguous and in order. This may make it easier to write a boot loader.

On a standard PC compatible disk, the tracks are numbered from 0 to 39. The sectors are numbered from 1 to 9. Side 0 is allocated first, then side 1. All sectors are 512 bytes. The K-OS ONE system treats the disk as contiguous sectors from 0 to 719.

The boot ROM can be located anyplace, but is generally located at memory location zero because that is where the vectors are for the 68000. The K-OS ONE operating system is located some place in low memory, with the command processor located next to it. It is optional whether the command processor is reloaded each time a user program is executed or whether the command processor stays resident in memory. If it stays resident in memory, extra memory will be used up, but it would be much faster to switch from one command to the next.

The command processor is an HTPL program that can be easily modified. The version supplied has no batch processing in it, but the hooks are available so batch processing can be easily added. The command processor would normally be located higher in memory than the operating system, but it can be located anyplace.

The user program will usually load in to a location after the command processor, but it can be located anyplace.

A general memory map of the system will look something like this diagram:

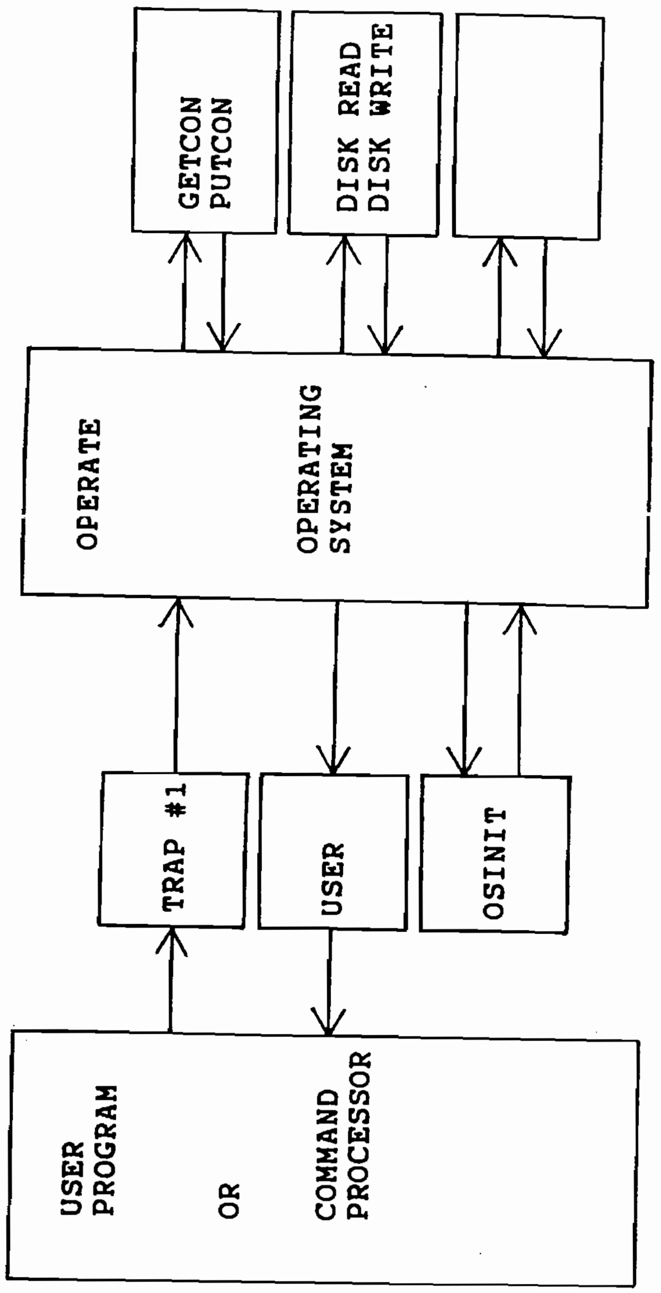
```
     PROM PATCH OPERATE BUFFERS COMMAND USER
    |----|-----|-------|-------|-------|----|
    |    |                     |<-16k->|
    |    |<---------- < 64k ---------->|
```

For a Non-Resident command processor, the memory map will look like this:

```
       PATCH OPERATE BUFFERS USER/COMMAND
      |-----|-------|-------|------------|
      | 100H|               |
            |<-Entry point  |<-Set by OSINIT
```

1. ROM Bios can be anyplace.
2. If command processor is resident, it can be located anyplace.

SYSTEM BLOCK DIAGRAM



SAMPLE ROUTINES

These are sample routines for OSINIT, TRAP1 and USER. They will probably have to be changed for your hardware.

```
;---------- INITIALIZE TRAP REGISTER SAVE
OSINIT MOVE.L   (A4)+,D0        ;SYSTEM HEAP POINTER
       MOVE.L   (A4)+,SYSPC     ;INIT REG. SAVE AREA
       MOVE.W   #2000H,SYSSR
       MOVEM.L  #0FFFFH,SYSREG
       MOVE.L   #220000H,D1     ;RESIDNT CMD PROCESS
       MOVE.L   #240000H,D0     ;USER PROGRAM SPACE
       MOVE.L   D1,-(A4)        ;CMDSTRT (0=NON RES)
       MOVE.L   D0,-(A4)        ;USTART ()=SYS HEAP)
       RTS

;---------- ENTER TRAP PROCESSOR
TRAP1  MOVE.W   (SP)+,USRSR     ;SAVE USER STATE
       MOVE.L   (SP)+,USRPC
       MOVEM.L  #0FFFFH,USRREG
       MOVEM.L  SYSREG,#0FFFFH ;LOAD SYSTEM STATE
       MOVE.L   SYSPC,-(SP)
       MOVE.L   USRA0,-(A4)     ;LOAD PARAM POINTER
       CLR.L    D7              ;BYTE CONVERSION REG
       RTS                      ;GOTO TRAP PROCESSOR

;---------- ENTER USER PROGRAM
USER   MOVEM.L  USRREG,#0FFFFH ;LOAD USER STATE
       MOVE.L   USRPC,-(SP)
       MOVE.W   USRSR,-(SP)
       RTE                      ;GO TO USER PROGRAM
```

PATCH ROUTINES FOR K-OS ONE

These are routines that you must (some routines are optional) provide to run K-OS ONE on your hardware.

OSINIT   (#trap #heap -- cmdstrt ustart)

This routine sets up an initialize area that allows the trapl vector to reload the registers needed for htpl. It also returns the starting address for the command processor and the user program area. If the command processor is not resident, a 0 should be returned.

USER   (--)

This reloads the user registers and returns to the user program.

CLOCKSET (#systime #sysdate --)

This sets a pointer for the realtime clock, if there is one.

EXSET   (#adrs --)

This sets up an exception vector for the ROM to branch to if any exception occurs.

CONSOLE DRIVER LINKS

GETCON   (-- char)

This gets a character from the console, and returns it on the stack.

PUTCON   (char --)

This takes the character from the top of the stack and sends it out to the console.

INITCON (--)

This initializes the console.

STATCON (-- status)

This gets the status of the console.

CNTRLCON (#adrs size -- status)

This sends a control string to the console and returns the status.

VECTORS FOR PRINTER

GETPRN (-- char)

This gets a character from the printer, and returns it on the stack. In most cases it will return a null.

PUTPRN (char --)

This takes the character from the top of the stack and sends it out to the printer.

INITPRN (--)

This initializes the printer.

STATPRN (-- status)

This gets the status of the printer.

CNTRLPRN (#adrs size -- status)

This sends a control string to the printer and returns the status.

DISK 1

DSKREAD1 (rec #buf -- status)

Read a record from disk 1.

DSKWRITE1 (rec #buf -- status)

Write a record to disk 1.

DSKSTAT1 (-- status)

Get the status of disk 1.

DSKCNTRL1 (#adrs size -- status)

Send a control string to disk 1.

DSKFLSH1 (-- status)

Flush disk 1.

PATCH ROUTINE OFFSET VALUES

PATCH EQU 0 - 100H

ROUTINE	OFFSET
OSINIT	+0
USER	+6
CLOCKSET	+12
EXSET	+18
WAIT	+24
GETCON	+30
PUTCON	+36
INITCON	+42
STATCON	+48
CNTRLCON	+54
GETPRN	+60
PUTPRN	+66
INITPRN	+72
STATPRN	+78
CNTRLPRN	+84
DSKREAD.1	+90
DSKWRITE.1	+96
DSKSTAT.1	+102
DSKCNTRL.1	+108
DSKFLSH.1	+114

STATUS WORD DEFINITIONS

FCB STATUS

BIT	MEANING
0	
1	FILE MODIFIED
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	TEMP

DCB STATUS

BIT	MEANING
0	FAT CLEAN
1	FAT MODIFIED
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

BUF STATUS

BIT	MEANING
0	BUFFER CLEAN
1	BUFFER MODIFIED
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	
D	
E	
F	

CHANNEL STATUS

BIT	MEANING
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
A	
B	
C	FILE
D	DEVICE
E	READ
F	WRITE

K-OS ONE OPERATING SYSTEM

CHANNEL

	Offset	Parameter
DC.W	+0	Flag
DC.L	+2	DCBpn
DC.L	+6	FCBpn
DC.L	+10	RWP
DC.L	+14	CAB

DCB DEVICE CONTROL BLOCK - TERMINAL

	Offset	Parameter
DC.L	+0	Get Routine
DC.L	+4	Put Routine
DC.L	+8	Initialize Routine
DC.L	+12	Status Routine
DC.L	+16	Control Routine

DCB DEVICE CONTROL BLOCK - DISK

	Offset	Parameter
DC.L	+0	Read Routine
DC.L	+4	Write Routine
DC.L	+8	Status Routine
DC.L	+12	Control Routine
DC.L	+16	Flush Routine
DC.W	+20	Status
DC.W	+22	BSF
DC.W	+24	Fatsize
DC.B	+26	Unit
DC.B	+27	Type
DC.L	+28	Fatpointer
DC.W	+32	Dirstart
DC.W	+34	Datastart
DC.W	+36	Fatstart

FCB FILE CONTROL BLOCK

	Offset	Parameter
DC.B	+0	8 Character Filename
DC.B	+8	3 Character Ext
DC.B	+11	1 Byte Attribute
DC.W	+22	Time
DC.W	+24	Date
DC.W	+26	Start
DC.L	+28	Size
DC.W	+32	Status
DC.W	+34	Users
DC.W	+36	Bufcount
DC.L	+38	BCBpn
DC.W	+42	Unit
DC.W	+44	Dirrec
DC.W	+46	Dirndx

BCB BUFFER CONTROL BLOCK

	Offset	Parameter
DC.W	+0	Bufstat
DC.W	+2	Record
DC.L	+4	Buffer

(Repeat once for each FCB Bufcount.)

K-OS ONE
COMMAND PROCESSOR

USER'S MANUAL

Hawthorne Technology
8836 S.E. Stark
Portland, Oregon 97216

(503) 254-2005

K-OS ONE
COMMAND PROCESSOR

SUPPLIED COMMANDS

BIN2HEX	Convert Binary to Hex
CHDIR	Change Directories
COPY	Copy File
DATE	Display / Set Date and Time
DEL	Delete File
DIR	List Directory
DUMP	Dump a File
FORMAT	Format a Disk
HEX2BIN	Convert Hex to Binary
MKDIR	Make a Directory
PRINT	Print
RENAME	Rename a File
RMDIR	Remove Directory
TYPE	Display Contents of a File

These are the commands that are executed by the standard K-OS ONE command processor. If you use a custom command processor then the commands will be different. The main purpose of the command processor is to give direct access to services and functions provided by the operating system. Application programs can access these services directly and in some cases these services will be available to the user through the application.

If other commands are required, they are easy to add using the existing commands as examples.

The command processor is written in HTPL like any user program. The source code is provided to make it easy to change.

No batch processing is included with this release. There are system commands that would make it easy to add batch or automatic files.

Command Format

K-OS ONE Command Processor commands are discribed in this section. The syntax of the commands will be discribed as follows:

The parts of the examples shown in UPPER CASE must be used in the format shown.

The parts of the examples shown in lower case represent information to be supplied by you when you use the command.

When using the commands, it does not matter if upper case or lower case letters are used.

Parentheses () indicate optional information that you could supply.

Punctuation in a command must be used as shown in the examples.

When a file name is to be used by a command, the selected disk drive will be assumed as a default unless you specify a drive along with the file name.

BIN2HEX CONVERT BINARY TO HEX

BIN2HEX will convert the content of a binary file (source) to hex, and put the result in the destination file. The source file is not affected.

```
BIN2HEX source destination
```

CHDIR CHANGE DIRECTORY

This command is used to change which directory will be considered the default directory. The default directory is the one that is searched when no directory or disk is specified.

```
CHDIR newdirectory
```

The command given without a directory name will respond by displaying the name of the current directory.

COPY COPY

This command is used to copy from one file or device to another.

```
COPY source destination
```

If the file identification does not specify the drive, the selected drive is used as a default.

DATE DATE / TIME CHANGE / DISPLAY

This command is used to display or change the system date or time. To leave the date and time unchanged, press return.

```
DATE
```

To change the date or time, you enter the new values.

```
DATE newdate newtime
```

DEL — DELETE

This command is used to delete or erase files from the system. When a file has been deleted the space that it used will be made available for reuse by the rest of the system.

```
DEL  filename
```

DIR — DIRECTORY

This command is used to display the contents of the currently selected directory or for a directory or disk that is specified. For each file the size and date are also shown.

```
DIR
```

Displays the current directory.

```
DIR B:
```

Displays the directory on drive B, no matter which drive is selected.

```
DIR \TRUNK
```

Displays the contents of the directory 'TRUNK'.

DUMP — DUMP FILE

The dump command can be used to look at a binary file. The file is displayed, one screen at a time. The hex value is displayed on the left, and the ASCII translation of it is displayed to the right.

```
DUMP filename
```

FORMAT — FORMAT DISK

This command is used to format a new disk or to re-format a used one. To format a disk, it is always safer to specify which disk you want the format to take place on.

```
A> FORMAT  B:
```

This prevents formating your system disk on the default drive.

When you format a disk, a pattern is written on the disk, totally distroying anything that may have been on the disk.

HEX2BIN — CONVERT HEX TO BINARY

HEX2BIN will convert the contents of a hex file to binary and put the result in the destination file.

```
HEX2BIN source destination
```

MKDIR — MAKE DIRECTORY

This command is used to create a new directory or sub-directory for a disk. If no existing directory name is specified, the sub-directory you make will be under your root directory.

```
MKDIR \TRUNK
```

will make a sub directory 'TRUNK'.

After the directory named TRUNK exists,

```
MKDIR \TRUNK\BRANCH
```

will make a directory BRANCH that is under TRUNK. Using this method, you can create a structure of directories.

PRINT **PRINT**

This command is used to send a text file to the system printer. It will expand tabs and assume a tab stop every 8 positions. A top of form will be executed after every 55 lines of print.

```
          PRINT  filename
```

REN **RENAME**

This command is used to change the name of a file.

```
          REN  oldname newname
```

RMDIR **REMOVE DIRECTORY**

This command is used to remove a directory from the system. The directory to be removed must be empty, (have no files in it).

```
          RMDIR  directory name
```

TYPE **DISPLAY CONTENTS OF A FILE**

This command is used to display the ASCII contents of a file on the console. Tabs will be expanded, with a tab stop every 8 charactors.

```
          TYPE  filename
```

THE 68000 OPERATING SYSTEM

K-OS ONE

PROGRAMMER'S MANUAL

Hawthorne Technology
8836 S. E. Stark
Portland, Or 97216

(503) 254-2005

68000 OPERATING SYSTEM

K-OS ONE   PROGRAMMER'S MANUAL

INDEX

SYSTEM CALLS

I/O MANAGEMENT

1	TEST I/O
2	READ
3	WRITE
4	POSITION
5	OPEN CHANNEL
6	CREATE FILE
7	CREATE TEMPORARY FILE
8	CLOSE CHANNEL
9	DELETE FILE
10	CONTROL DEVICE
11	MAKE DIRECTORY
12	GET / CHANGE CURRENT DIRECTORY
13	DELETE DIRECTORY
14	START DIRECTORY SEARCH
15	FIND NEXT FILE
16	LOCK / UNLOCK BLOCK
17	GET / SET - FILE DATE & TIME
18	DUPLICATE CHANNEL
19	GET / SET - FILE ATTRIBUTES
20	RENAME FILE
21	GET FREE SPACE ON DISK
22	RAW READ DISK
23	RAW WRITE DISK

NON DISK

41	SET - CONTROL-BREAK ADDRESS/ACTION
42	GET / SET - TIME & DATE
43	GET DOS ID
44	GET / RELEASE / INQUIRE ABOUT MEMORY SIZE

PROGRAM CONTROL

51	GET LAST TERMINATE CODE
52	GET AND EXECUTE PROGRAM
53	TIME DELAY WAIT
54	TERMINATE PROGRAM
55	GET COMMAND LINE

BATCH CONTROL

61	GET NEXT LINE FROM .BAT FILE
62	GET .BAT CONTROL LINE
63	START .BAT FILE PROCESSOR

SYSTEM CALLS

DESCRIPTION

The 68000 operates in one of two different states, user or supervisor. When in the user state, certain commands are not available. Also, when in user state, the memory is mapped from a logical address to a physical address that is different. None of the I/O devices can be accessed from a user state directly. The code for the operating system or any other user cannot be accessed.

System Calls are used to perform operating system level functions in a straight forward manner. The appropriate parameters are loaded into the parameter block, the call is issued with A0 pointing to the parameters, the result is found in the pre-specified location. In most cases error codes will describe the resulting success or mode of failure.

The system calls are the only way for a user program to comunicate with the world or with mass storage. The command processor forms the user interface and runs like any other user program. Calls are included to ease the task of writing a customized command processor. Also included are services that most users will need and could be difficult or take up lots of memory.

This operating system uses a parameter block to describe the operation requested instead of registers or the stack. When registers or the stack are used it is more difficult to have more than one supervisor operation active at the same time. With more than one operation active the overall through-put of the system can be increased and it will be more responsive to user requests. This is more important with comercial operations that are I/O intensive than with operations that are computation intensive.

TEST I/O   Command = 1

This command is used to test the status of an I/O channel. The status bits indicate the following:

Status Bit	Indication
15	1 = Command processed
14	1 = Error -- ignore other bits
13	
12	
11	
10	
9	Device needs reset or setup
8	Device is busy
7	RI -- Ring (phone call) / optional
6	DSR - They are ready / optional
5	CTS - I can send /optional
4	
3	EOF
2	^C or Break
1	Transmit Ready
0	Charactor Available

```
       Offset  Parameter
----   ------  --------------
DC.W     +0    Command
DC.W     +2    Status
DC.W     +4    Channel
```

Return Status

READ Command = 2

Read from a device or a disk file. This call reads from a disk file or device that has been previously opened. (For random access, the POSITION command can be used in conjunction with the READ command for reading from specific locations.)

The requested number of bytes of data are read from the specified disk file into the buffer. If more than one byte is read from a device then the input stream will be edited and echoed and at most one line of input will be transfered. The actual number of bytes read will be returned.

	Offset	Parameter
DC.W	+0	Command - Options
DC.W	+2	Status
DC.W	+4	Channel number
DC.W	+6	Number of bytes to read
DC.W	+8	Number of bytes actually read
DC.L	+10	Pointer to data buffer

Options

ECHO - if device
EDIT - if device

WRITE Command = 3

Write to a disk file or device. This call writes to a disk file or device that has been previously opened. (For random acces, the POSITION command can be used in conjunction with the WRITE command for writing to specific locations.)

The requested number of bytes of data are written from the buffer to the specified disk file or device. If the area being written to is locked by another task, the write request is rejected. If a write request is rejected because of a file lock it should be tried again later.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.W	+4	Channel number
DC.W	+6	Number of bytes to write
DC.W	+8	Number of bytes written
DC.L	+10	Pointer to data buffer

POSITION   Command = 4

The position command is used to move the file pointer. The move is specified as an offset from the beginning of the file, from the current location, or back from the end of the file.

	Offset	Parameter
DC.W	+0	Command - Options
DC.W	+2	Status
DC.W	+4	Channel number
DC.L	+6	Location pointer offset

Options

0	From start
1	From current
2	From end

OPEN CHANNEL   Command = 5

Open a disk file or device. This call opens a device or an existing file for use. The channel used to access the file or device is specified by the user or created by the system if a channel of 0 is specified. A file or device can be open for more than one channel at a time.

	Offset	Parameter
DC.W	+0	Command - Options
DC.W	+2	Status
DC.W	+4	Channel number
DC.L	+6	Pointer to file name

Options

Shared read
Exclusive read
Shared write
Exclusive write

CREATE FILE Command = 6

Create a disk file. This call creates a file but dosen't open it for use. If the file already exists, it's length is changed to 0.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file name
```

CREATE TEMPORARY FILE Command = 7

This command is used to create and open a temporary file on the designated disk. The name is used only by the system and will be unique for each call. This call creates the file and opens a file channel to it. When the file channel is closed the file will be automatically deleted.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Channel number
DC.L    +6    Pointer to disk name
```

CLOSE CHANNEL   Command = 8

Close a device or disk file. This call closes a previously opened file. Any partial buffer is sent to the device or disk file. When a task ends, any channels that were left open are closed. When a task closes a file channel any locked records are unlocked. If a shared file is closed the close command affects only the task making the call. If a temporary file is closed, it is automatically deleted. Because of the shared nature of the disk system a close call should not be used to insure that buffers are flushed to the disk.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Channel number
```

DELETE FILE   Command = 9

Delete a disk file. This call deletes a file from the directory, releasing the space previously occupied by that file.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file name
```

CONTROL DEVICE Command = 10

The control command sends control information to the device. The information is not transmitted but is used to control or change the characteristics of the device or its buffer.

```
        Offset  Parameter
----    ------  --------------
DC.W      +0    Command
DC.W      +2    Status
DC.L      +4    Channel number
DC.W      +8    Number of bytes to send
DC.L     +10    Pointer to control string
```

NOTE: This can format or initialize a disk

MAKE DIRECTORY Command = 11

The make directory command is used to start a new directory. A new directory starts with only two entries in it. Itself and the parent directory.

```
        Offset  Parameter
----    ------  --------------
DC.W      +0    Command
DC.W      +2    Status
DC.L      +4    Pointer to new directory name
```

GET / CHANGE CURRENT DIRECTORY Command = 12

This command is used to designate a new default directory. When an incomplete file reference is made the default directory will be the one that is searched for the requested file. If the directory name pointer points to a null, the path of the current directory is copied. If the pointer points to a name, it becomes the new current directory.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to new directory name
```

DELETE DIRECTORY Command = 13

The delete directory command is used to remove a directory from the system. The root directory on a disk cannot be deleted. In order for a directory to be deleted it must not contain any files or subdirectories.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to directory name
```

START DIRECTORY SEARCH   Command = 14

The START DIRECTORY SEARCH command is used to implement the user directory function. A path name for a directory is specified. When the directory is found, information about the first file is returned. All files and sub-directories are reported. Finding a specific file or directory is left to the calling program. The directory is opened and positioned so FIND NEXT can can retrieve subsequent files.

A pointer to NULL STRING and NIL POINTER specify the default directory.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to directory name to search
```

FIND NEXT FILE   Command = 15

The find next command is used to continue the printing or searching of a directory. Any command that uses any directory (such as open or close), will set the pointer to not valid.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file information
```

```
Note:     File Information:

               Name        12 BYTES
               Attributes    .L
               Time          .L
               Date          .L
               Size          .L
```

LOCK / UNLOCK BLOCK Command = 16

The lock/unlock command is used to lock a specified portion of a file for exclusive writing by a single user. Other users can read the locked area but only the owner can write to it. This is needed to prevent collisions when multiple users are updating a shared file. A number of bytes specified as size starting at the current pointer is locked or unlocked. To prevent collisions in a multiuser file update the following sequence should be used: lock,read,update,write,unlock. By convention if the first byte in a file is locked then more records are being added to the file.

```
      Offset  Parameter
----  ------  ---------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Channel number
DC.W    +6    Number of bytes to lock/unlock
```

GET / SET - FILE DATE & TIME Command = 17

This command is used to set the time and date that a file was read or written. Normally this is set when the file is last written to and is closed. If the new date is zero, the file's current date is returned and not altered. If the new time is zero, the file's time is returned and not altered.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file name
DC.L    +8    New time
DC.L   +12    New date
```

DUPLICATE CHANNEL Command = 18

This is used to create a duplicate of a file channel. A new channel is created by the operating system to refer to the same file or device as an existing channel.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Old channel number
DC.L    +6    New channel number
```

GET / SET - FILE ATTRIBUTES Command = 19

This command is used to get the attributes of a file or to set the attributes.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file name
DC.L    +8    File attribute bits
```

RENAME FILE   Command = 20

The rename command is used to change the name of a file that already exists.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to old file name
DC.L    +8    Pointer to new file name
```

GET FREE SPACE ON DISK   Command = 21

This command returns the number of free blocks that are on the disk that can be allocated by the directory. Caution should be used in interpreting the results because another task could use some or all of the space before the program making the request can.

A pointer to NULL STRING and NIL POINTER specify the default directory.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Number of bytes available
DC.L    +8    Pointer to directory name
```

RAW READ DISK

Command = 22

This command reads a 512 byte block from UNIT, at RECORD NUMBER, into BUFFER. Unit 0 (zero) is RAM Disk, Record 0 (zero) is the first sector on the disk.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.B	+4	UNIT Number
DC.B	+5	Reserved
DC.W	+6	RECORD Number
DC.L	+8	POINTER to BUFFER

RAW WRITE DISK

Command = 23

This command writes a 512 byte block from UNIT, at RECORD NUMBER, into BUFFER. Unit 0 (zero) is RAM Disk, Record 0 (zero) is the first sector on the disk.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.B	+4	UNIT Number
DC.B	+5	Reserved
DC.W	+6	RECORD Number
DC.L	+8	POINTER to BUFFER

NON DISK

SET - CONTROL-BREAK ADDRESS / ACTION

Command = 41

This command is used to specify the action to take when an exception is detected. An address can be specified to branch to or the operating system can abort the program that is running. If the address is 0, the program is terminated. Otherwise control is transfered to the specified address.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.W	+4	Vector number
DC.L	+6	Address of routine
DC.L	+10	Address of info

GET / SET - TIME & DATE   Command = 42

This command is used to get the system time or to set the system time. In general a user program should only get the time and not set it. A startup routine can be used to set the system time. If date is 0, the current date is returned and not altered. If time is 0, the current time is returned and not altered. If bit 8 of the command is set, the time is returned in milliseconds from boot and not altered.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.L	+4	Time
DC.L	+8	Date

Time -- Long Word, Byte Format:	HH MM SS 00
Date -- Long Word, Byte Format:	YY MM DD 00

Options: Command[8] =1   Return time since boot in Milliseconds.

GET DOS ID   Command = 43

This command returns a unique 64 bit value that is the serial number for the operaing system. This number is hard to change and can be used to lock valuable software to only one system. A program that is protected in that manner can be freely copied but can only be used with a single system. Because the operating system separates the user from the hardware the exact hardware that is being used is unknown to any program. Because all interface with the system is defined in terms of system calls, the operating system itself may have many significant differences but work in the same manner.

	Offset	Parameter
DC.W	+0	Command
DC.W	+2	Status
DC.L	+4	High 32 bits of ID
DC.L	+8	Low 32 bits of ID

GET / RELEASE / INQUIRE ABOUT MEMORY SIZE

Command = 44

This command is used to do some related things. First it can be used to find out how much memory is available to the requesting program. The second use is to change the amount of memory that is available. More memory can be requested or some existing memory given up. The system will try to comply by allocating or releasing pages of memory. If the number of bytes requested =0, the current size is returned. The size of a memory page and the number of pages available may vary from one system to another.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Number of bytes for new
                size requested
DC.L    +8    Number of bytes allocated for task
```

PROGRAM CONTROL

GET LAST TERMINATE CODE   Command = 51

This command is used to get the termination code of the last program to execute for this user. This is used to signal to the batch processor any errors or special conditions that may have occured so the batch processor can take any required action.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Terminate code
```

GET AND EXECUTE PROGRAM   Command = 52

This command is used to get and execute a new program in this address space. The current program is overlayed.

Any open file channels will be passed to the new program. This includes temporary files as well as permanent ones.

Either this task space can be used to hold the new program or a new task space can be requested if there is space available.

```
        Offset  Parameter
----    ------  --------------
DC.W     +0     Command
DC.W     +2     Status
DC.L     +4     Memory size requested for new task
DC.L     +8     Pointer to file name that
                  has program
DC.L    +12     Pointer to command line for
                  new program
```

TIME DELAY WAIT   Command = 53

This command is used to delay a task for a period of time. If zero delay is specified the effect is to forfeit the remaining part of the current time slice for use by other tasks. This is used instead of a delay loop because it does not waste processor time and the delay is more predictable. The delay is specified in milliseconds but the exact delay is determined by the length of a time slice and how many other tasks are using the system.

```
        Offset  Parameter
----    ------  --------------
DC.W     +0     Command
DC.W     +2     Status
DC.L     +4     Time delay in milliseconds
```

TERMINATE PROGRAM Command = 54

This command is used to terminate the currently running program and return control to the command processor that was specified. A value can be returned to make the command processor aware of any unusual conditions or the reason for termination. As an option the file channels can be left open. This allows a temporary file to be passed back to the starting program.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Termination code for this task
```

GET COMMAND LINE Command = 55

This command is used to get a .CMD control line. The command line will contain any actual parameters used.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Number of bytes read
DC.L    +6    Pointer to command line buffer
```

BATCH CONTROL

GET NEXT LINE FROM .BAT FILE Command = 61

This command gets the next line of the batch file for this task and places it in a buffer. If no batch file is active for this user, an error is noted. This command is used to construct batch file processing programs.

```
        Offset  Parameter
----    ------  --------------
DC.W      +0    Command
DC.W      +2    Status
DC.W      +4    Number of bytes read
DC.L      +6    Pointer to buffer
```

GET .BAT CONTROL LINE   Command = 62

This command is used by batch file processors to get the command line that was used to start the batch file. The command line will contain any actual parameters used.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.W    +4    Number of bytes read
DC.L    +6    Pointer to command line buffer
```

START .BAT FILE PROCESSOR   Command = 63

This command is used to start a batch file process for this user. A word is used to point to the name of the batch file to be processed. Another word is used to point to a buffer with at most 256 characters that will be used as the command line for the batch file processor.

```
      Offset  Parameter
----  ------  --------------
DC.W    +0    Command
DC.W    +2    Status
DC.L    +4    Pointer to file name
DC.L    +8    Pointer to command line
```

A TABLE OF ERROR CODES

CODE	MEANING
00	Normal Exit
01	Invalid Channel Number
02	Channel Not Open for Read
03	Channel Not Open for Write
04	No Channel Available
05	No File Control Block Available
06	Invalid Unit Number
07	File Not Found
08	Disk Full
09	Directory Full
0A	Attempt to Read Beyond End of File
0B	Attempt to Postition Beyond End of File
0C	Invalid Device
0D	File in Use
0E	Device in Use
0F	System Error
10	Read Error
11	Write Error
12	File Exists
13	Directory Not Found
14	Directory Not Empty
15	File is Read Only
16	Attribute Conflict
17	Request Denied

* * HTPL * *

HAWTHORNE TECHNOLOGY PROGRAMMING LANGUAGE

USER MANUAL

Hawthorne Technology
8836 SE Stark
Portland, Or 97216

(503) 254-2005

HTPL USER MANUAL

INTRODUCTION

HTPL is a language created in 1986 at Hawthorne Technology to be used with the 68000 micro-processor. The main design goal was to have a language that would be economical to implement, would produce efficient code, and make it easy to write menu driven, overlayed programs. A second goal was to create a language that could be used to implement part of the operating system or at least the user utilities.

HTPL was influenced by several existing languages. The most important was FORTH because it is small and generates very good object code for the size of the compiler required. The other major influence was PASCAL which was used to implement the first version of the compiler under HS-DOS. Pieces of other languages are also present.

The expressions in HTPL are reversed polish like FORTH. This allows the programmer to optimize the code produced and allows the compiler to have an extremely simple code generator section. While FORTH is good for expressions the control structures are very awkward and limited. Most FORTH systems are oriented to being interactive and handicap compiled implementations. While many systems can be written without local variables or goto statements almost every major programming project needs one or both of these. Instead of being dogmatic, the practical approach was taken and labels were allowed.

Pascal, Modula, C and other similar languages offer good control structures that can be generated efficiently. These were the models used for control. These languages suffer from excessive overhead for procedure calls and it is difficult to generate good object code from a small compiler.

The HTPL language is not intended to be good for every purpose. It can be used to implement part of the utilities for the HT-21 and for many simple user applications. For large programs or for complicated programs a standard language might be better.

LANGUAGE DESCRIPTION

The HTPL language is composed of different elements. There are some reserved symbols with special meaning and a few reserved words. In general all items can be described as numbers, words, comments, and string constants. Some are defined by the user and some are pre-defined.

HTPL is an RPN language. This means that all objects that are operated on must be pushed on an evaluation stack and then the operation named. There are no rules of associativity or precedence in HTPL. All values are placed on the stack when encountered. All procedures are called when referenced. The compiler will not change the sequence in which actions are performed.

A user defined word is any group of up to 12 printable characters that start with something that is not defined as a special prefix or a digit. Because HTPL is not case sensitive, all letters used are converted to upper case in the internal symbol table. User defined words are of three types:

```
variables
labels
procedures
```

A variable is the label of a data item in either common or local storage. Variables are 1, 2 or 4 bytes long and are defined with byte, word or long statements.

Labels are declared with a label statement and are given a defined value when they appear in the program following a colon (:). Labels are used as the targets for goto statements.

Procedures are defined with a procedure statement. A procedure is a collection of instructions that produces a result. It begins with:

```
proc name
  (and continues until a matching)
end  (is encountered)
```

Arguments are passed to procedures on the stack or through variables. Procedures can be referenced before they are defined. Any undefined word on pass one is assumed to be a forward reference to a procedure. Procedures don't have types.

A variable is always used with one of three prefixes:

@ !

When # appears with a variable the 32 bit address of the varaible is pushed on the evaluation stack. The # prefix may also be used with procedure names or labels.

```
  #VAR  -  Address of VAR is pushed on stack
  #PRO  -  Address of PRO is pushed on stack
```

When @ appears with a variable, the value stored in the variable is expanded to 32 bits, if necessary, and pushed on the evaluation stack. Words are sign extended but bytes are not. The high order bits of a byte variable are set to 0.

```
  @VAR  -  Value stored in VAR is pushed on stack.
```

When ! appears with a variable, the top 32 bit value on the evaluation stack is truncated, if necessary, and stored in the variable.

```
  !VAR  -  Top value from stack is stored in VAR.
```

PRE-DEFINED WORDS

There are two types of predefined words:

structure and macro

A structure word is used to define words or to control the sequence of operation.

A macro word is like an operator and generates inline code. Other than the size of code generated there is no difference between a macro and a procedure call.

A comment is enclosed in parentheses ( ) .

(This is a comment.)

No code is generated from a comment. A comment can occur anywhere a space can occur. Comments do not nest so care must be taken when commenting out large sections of programs.

A plain number is always pushed on to the stack. There are decimal numbers that contain only digits, and hex numbers that contain digits and the letters A to F. A hex number starts with a digit and ends with the letter H, or starts with a $. Because of the syntax of the language there are no negative numbers. If a negative number is needed then use the positive value and the negate procedure.

A character constant is enclosed in single quotes and can be from one to four characters long. The charcters are right justified in a long word and any high order bits are set to 0. A string starts with double quote marks ("). When a string appears in an expression, space is allocated for the string in the common storage area and the address is pushed on the stack.

STRUCTURE WORDS

ROOT

This tells the compiler that we are compiling the root section of a set of a system. When the root is compiled the runtime library is included in the code and initialize code is generated for variables in common.

OVERLAY

This tells the compiler that we are compiling an overlay for a system. When an overlay is compiled no runtime library is generated and no initialization code is output for variables in common. The common definitions are for reference only and should be the same as the common definitions of the root or an error may result.

MODULE

This marks the start of local variables and procedures. Code is generated for a module whether or not it is root or overlay. Local variables are not initialized. Storage for local variables is on the heap. If the heap is to be used then space for local variables must be explicitly allocated.

PROGRAM

This marks the start of the program for either the root or an overlay. This is where execution will start after the object code has been loaded into memory. A program ends when a matching END statement is found.

LINK

This links in a hex code module that has been generated by the assembler. Procedures can be defined when linked, but variables can not. More than one procedure can be in a module. A module can have its own internal structure.

PROC

This marks the start of a user defined procedure. A procedure continues until a matching END is found. A procedure is called by simply using its name. Arguments are passed to a procedure on the stack and results can be returned the same way.

BYTE / CHARACTER

This is used to define user variables. If a module statement hasn't been encountered then the variables will be in common. After a module statement is encountered the variables will be local. A byte occupies one byte of memory. All the variables defined in a single byte statement will be placed in contiguous bytes. The first byte will be on a word boundary. At the end of the byte statement a null byte will be added if needed so that the next code will be on a word boundary.

INTEGER / WORD

This is used to define user variables. If a module statement hasn't been encountered then the variables will be in common. After a module statement is encountered the variables will be local. A word occupies two bytes of memory. All the variables defined in a single word statement will be placed in contiguous bytes on word boundaries.

LONG

This is used to define user variables. If a module statement hasn't been encountered then the variables will be in common. After a module statement is encountered the variables will be local. A long occupies four bytes of memory. All the variables defined in a single statement will be placed in contiguous bytes on word boundaries.

LABEL

This is used to define labels for use with the GOTO statement or COMPGO words. The LABEL words tell the compiler that these will be labels but the labels don't have values until they are encountered in the body of the code.

WHILE DO END

This sequence is used to execute a block of code while a specified condition is true. WHILE marks the start of the loop where control will return after the loop is executed. DO uses the top value of the stack to decide whether or not to execute the block. One item is removed by the DO. When the END is encountered control branches back to the WHILE. As long as the expression between the WHILE and the DO is true the code between the DO and the END will be executed.

REPEAT UNTIL.

This sequence is used to execute a block of code until a specified condition is true. REPEAT marks the place where control will return to at the end of the loop. UNTIL uses the top item of the stack to determine whether control will pass back to the REPEAT or whether the program will continue.

IF THEN END
IF THEN ELSE END
IF THEN ELSIF THEN END

These words are used to select which one of several blocks of code will be executed. IF is a polite word, it doesn't do anything but it makes the code easier to read. IF should be used to mark the start of the expression that will be tested. THEN uses the top item on the stack to determine if the code following THEN will be executed. If the top value of the stack is true the code following THEN will be executed, if not the code following the ELSE will be. END marks the end of the selection.

ELSIF THEN

These words are used to try a new condition in a IF THEN END sequence. When ELSIF is encountered the expression between ELSIF and THEN is evaluated. If true, the code after the THEN is executed; if not, then the code after the next ELSE or ELSIF is executed. Only one of the THEN, ELSIF-THEN, ELSE blocks will be executed.

CASE [] [] ELSE END

This is used to select one of several alternatives. When CASE is found, the top value on the evaluation stack becomes the test value. When the END is encountered the value is removed from the stack. When a [is found the selection value is compared to the value in [] and if equal the code following the [] will be executed. If not equal then the next [] will be tested. If no [] is found equal when ELSE is encountered then that block will be executed.

RETURN

The return statement is used to return from a procedure at some place other than at the end. When leaving a procedure care must be taken to be sure the evaluation stack is left in a predictable state. The return stack in HTPL is used only for returns and not for any other control. The only control structure that uses the evaluation stack is the case statement which uses one level.

GOTO

The goto statement is used to transfer the execution of the program to some point other than in a direct line. The goto statement is always followed by a label. Control is transfered to the statement at that label and execution procedes from that point.

ASSEMBLY LANGUAGE MODULES

It is possible to create assembly language modules and link them into HTPL programs. This provides the programmer with a great deal of flexibility.

To create a procedure module to link to your HTPL program you write the module in the required format and assemble it into an Intel format HEX file. The procedure is then linked into the program using the following command:

```
link 'filename.hex'
```

The filename should be the name you have assigned to the hex file of your assembly language module. The .hex extension must always be given. It is not assumed by the compiler.

A simple RTS should be used to exit an assembly language procedure in a module.

* * * * FORMAT FOR LINK.HEX MODULES * * * *

```
        ORG       0
        DC.W      SIZE
        DC.W      ENTCNT
        DC.B      'FIRSTNAME    '
        DC.W      FIRST
        DC.B      'SECONDNAME   '
        DC.W      SECOND

        ORG       0
        (actual code)

SIZE:   EQU       $
        END
```

Always ORG the code at 0.

The SIZE of the module must be even.

ENTCNT is the entry count. The number of entries in the module.

'FIRSTNAME   ' is the name of the first entry in the module. It is padded with spaces to a full 12 charactors. Only uppercase letters should be used for the names of procedures. HTPL is not case sensitive and converts all letters to uppercase while it is compiling a program.

FIRST is the address of the FIRST routine in the module.

The runtime library HTPLRTL.HEX is almost the same as any other .HEX file. The first 16 bytes will be filled in by the compiler to provide sizes for setting registers. The initialization code that comes before the first routine should not be changed or the program will not setup properly.

There are some registers that are used by HTPL and must be preserved when creating assembly language modules to link into HTPL programs or when modifying the runtime library file HTPLRTL.HEX.

REGISTER	RESTRICTED USE
D7	With the exception of the low byte, D7 must remain 0 (zero).
A3	Points to beginning of runtime library, must be preserved.
A4	Evaluation stack pointer.
A5	Points to local variables and must be preserved.
A6	Points to common variables and must be preserved.
A7	Return stack, must preserve.

STANDARD LIBRARY WORDS

These words are included in the runtime library that is used by the compiler when a root is being compiled. Some of these are in the runtime library and some are macros. A macro generates in line code to perform its function while a library routine generates a call to the routine. In general library words and user defined procedures act the same.

For each library word the word is given, the stack configuration before and after, and a short description. All stack items are 32 bits long. For higher performance several procedures can be written in assembly code and linked in using the link command.

Many of these routines are used in the operating system or the command processor supplied with the system. Both of those should be studied to see how these routines are used. If more information on the exact nature or behavior of any run time routine is needed you should look at the source code for HTPLRTL.

These routines are currently supplied with the HTPL compiler runtime library. As the language changes new library routines will be published. Any routine that is needed but not included can be built from standard words or can be created using an ASM file and linked in. If there are small routines that you use frequently then they can be added to your version of HTPLRTL and the file reassembled. We often debug a piece of code first in HTPL using other routines. We then recode it in assembly language form for greater speed and smaller size.

MACRO WORDS

This list contains the words in the library that are macro words.

ABORT	NIP	=0
AND	NOT	+
ATCALL	OR	-
ATGO	OVER	=
DECR	PICK	<>
DROP	RESTORE	<
DUP	ROT	<=
INCR	SAVE	>
NDXL	SHL	>=
NDXW	SHR	+2
NEGATE	SWAP	+4
	TUCK	-2
		-4
		+1
		-1
		<>0

RUNTIME LIBRARY WORDS

This list contains the words that are in the HTPLRTL.HEX runtime library.

ALLOCATE	GETC	RANGE
APRINT	GETLINE	RECLAIM
BFPRINT	IGET	RELEASE
BIPRINT	IPRINT	RESET
CCOMP	LENGTH	SPACES
CONCAT	MARK	SPRINT
COPY	MOVEC	TRAP
FGET	NEWHEAP	*
FILL	PUTC	*/
FPRINT	EXIT	MOD

LIBRARY WORDS

DUP (a -- a a)

Duplicate top item on evaluation stack.

DROP (a --)

Delete the top item on the stack.

NIP (a b -- b)

Like drop but delete the next to the top item.

ROT (a b c -- b c a)

Rotate the top three items on the evaluation stack.

SWAP (a b -- b a)

Swap the top two items on the stack.

OVER (a b -- a b a)

Copy the next to the top item.

TUCK (a b -- b a b)

Like over but copies the top item on the stack.

PICK (a -- a)

This is used for random access of an item on the evaluation stack. The sequence 1 PICK is like DUP and the sequence 2 PICK is like over.

AND (a b -- c)

Logical and's the top two items on the stack.

OR (a b -- c)

Logical or's the top two items on the stack.

NOT (a -- b)

Does a 1's compliment of the top item on the stack.

RANGE (a b c -- d)

Compare the third item on the stack to see if it is between the top two items. The top item is the high limit, the next is the low limit, and the third is the item being tested. The result is true if the item is in range and false if it is not.

NEGATE (a -- b)

Does a 2's compliment of the top item on the stack.

SHL (a b -- c)

Shift left, the next to the top item on the stack, using the top item on the stack for the number of bits to shift.

SHR (a b -- c)

Shift right, the next to the top item on the stack, using the top item on the stack for the number of bits to shift.

```
+  ( a b -- c )
-  ( a b -- c )
*  ( a b -- c )
/  ( a b -- c )
```

These are the normal arithmetic operations. All operations are done with 32 bit numbers. They use the top two items on the stack and put the result on the stack.

*/ (a b c -- d)

This is used to calculate a scaled product using a larger intermediate result. d= (a*b)/c The intermediate result is 64 bits long.

MOD (a b - c)

This returns the remainder from a division rather than the quotient.

```
=  ( a b -- c )
<> ( a b -- c )
<  ( a b -- c )
<= ( a b -- c )
>  ( a b -- c )
>= ( a b -- c )
```

These are the normal compare operations. All values are treated as 32 bit signed integers. They use the to two items on the stack and put the result on the stack.

=0 (a -- a)

This tests the top item on the stack and returns true if the value is 0.

<>0 (a - b)

This macro word tests the top item on the stack to see if it is non zero. If it is not zero the result is true.

TRUE (-- tt)

This pushes $FFFFFFFF on the stack.

FALSE (-- 0)

This routine pushes a zero on the stack.

+1 +2 +4 (a -- a)

Add 1, 2, or 4 to the top item on the evaluation stack and leave the result.

DBLINC (a b - a b)

This adds one to the top two items on the stack. This is the same as: +1 swap +1 swap

-1 -2 -4 (a -- a)

Subtract 1, 2 or 4 from the top item on the evaluation stack and leave the result.

DBLDEC (a b - a b)

This routine decrements the top two items on the stack. This is the same as: -1 swap -1 swap

!1 !2 !4 (a b --)

Store the next item on the stack using the top item as the address. All items are trimed to the proper size before they are stored.

@1 @2 @4 (a -- b)

Load 1, 2, or 4 bytes from memory using the top of the stack as the source address and push on the evaluation stack. Bytes are expanded to 32 bits by adding high order zeros. Words are sign extended.

NDXW (a b -- c)

This is used to calculate the address of an element in a word array. The top of stack has the base address of the array, the next item has the index value. The index value is multiplied by two and added to the base value forming the address.

NDXL (a b -- c)

This is used to calculate the address of an element in a long array. The top of stack has the base address of the array, the next item has the index value. The index value is multiplied by 4 and then added to the base address. The resulting address is left on the stack.

IPRINT (a b --)

This displays the next item on the stack as an integer using the number of spaces given by the top item.

BIPRINT (a b c --)

This converts the next item on the stack, as an integer, to a buffer. The top item on the stack gives the number of spaces to be used. The third item points to the buffer.

FPRINT (a b c --)

This displays the third item on the stack as a pseudo floating point value. The top item is how many digits to the right of the decimal point and the next item is how may places for the entire number.

BFPRINT (a b c d --)

This prints the third item on the stack as a pseudo floating point value. The top item is how many digits to the right of the decimal point and the next item is how may places for the entire number. The fourth item is the pointer to the buffer.

SPRINT (a --)

Print the string whose address is given as the top item of the stack. A string is assumed to terminated with a null byte.

APRINT (a b --)

This prints a group of characters of bytes including nulls. The top item is the count of how many bytes to send and the next item is the address of where they will come from.

IGET (a -- b)

Get an integer from the input stream. The top item on the stack tells how many characters to use. The result is left on the stack.

FGET (a b -- b)

Get a value from the input stream as a pseudo floating point number and store it on the stack as a 32 bit binary integer. The top item on the stack tells how many places to the right of the decimal point. The next item tells how many charactors there are in the entire number.

CRLF (--)

Send out a carriage return linefeed sequence.

SPACES (a --)

Prints a groups of spaces on the console.

GETC (-- a)

Get the next character from the console but do not echo it.

PUTC (a --)

Send the top item on the stack to the console as a character. Getc and putc are used in combination to create interactive user programs.

GETLINE (a b -- a)

Get a line of characters from the console. All characters are echoed and simple inline editing as defined by the operating system is permitted. The top value on the stack is the maximum number ofcharacters to read in. The next to the top item is the address of a buffer to put the characters in. The procedure will return the actual number of characters read.

ABORT (--)

This causes the program to be restarted and all stacks to be reset to their original values.

EXIT (a --)

This turns control of the system over to the command processor (operating system). The top item on the stack contains the terminate code.

RESET (--)

This word resets the return and evaluation stacks but does not restart the program. The reset command is used to set the stacks to a known condition after a branch or an error.

ALLOCATE (a -- a)

This is used to allocate memory from the heap. The heap is just above the stacks and grows from low to high in memory. The routine allocates the amount of storage requested by the top stack value if possible. If storage is allocted then a pointer to it is returned. If not enough memory remains then a null pointer is returned.

RECLAIM (a --)

This is used return a block of memory that was gotten by the allocate procedure.

MARK (-- a)

This returns a pointer to the current top of the heap.

RELEASE (a --)

This sets the heap pointer to the value on top of the stack.

NEWHEAP (--)

This resets the heap to its initial value after being used or if its state is unkown.

SETLOCAL (pp --)

This uses the top of stack value to set a pointer to local data.

GETLOCAL (-- pp)

This routine returns the base pointer (A5) used to access loacl data.

LOCALSIZ (-- ss)

This routine returns the size of the local data area that was delared by the program.

SAVE (-- a)

This copies the evaluation stack pointer to the evaluation stack.

RESTORE (a --)

The top item on the current evaluation stack replaces the evaluation stack pointer.

MOVEC (a b c --)

This routine moves bytes from left to right anyplace in memory. The parameters are from address, destination address, and number of characters to move.

MOVER (src dst cnt --)

This routine moves characters from right to left. The source and destination pointers must point to the right hand end of the fields to be moved. The count is the number of bytes to move.

FILL (a b c --)

This routine is used to fill a section of memory with a constant byte. The first parameter (a) is the address to start fillint, the next is the byte to fill with, the last (top of stack) is a count of how many bytes to fill.

TRAP (a --)

The trap word uses the top of the stack, puts it in A0, and does a 'trap 1'. The trap command expects the item on the stack to be pointing to a parameter block. This allows communication between the command processor and the operating system.

ATGO (pp --)

This causes an unconditional branch to the location whose address is on the top of the evaluation stack.

ATCALL (pp --)

This routine causes an uncondtional call to the location whose address is on the evaluation stack.

CONCAT (s1 s2 --)

This routine combines two strings into one by appending the second string to the first string.

COPY (s1 s2 --)

This routine makes a copy of the first string where the top stack itme is pointing.

CCOMP (p1 p2 cnt -- torf)

This routine compares two areas of memory pointed at by p1 and p2. If the two areas are equal for the number of bytes then true is returned else false is returned.

DELETE (str pos num --)

This routine deletes part of a string. The first argmuent tells how many characters to delete. The second argument tells where in relation to the start of the string is the desired part to delete.

LENGTH (a -- b)

This finds the length of a null terminated string whose address is the top item of the stack.

POS (srch obj -- pntr)

This routine searches the object string for the search string. If the string is found then its position is returned. If the search string is not found then a null pointer is returned.

SCOMP (s1 s2 -- torf)

This routine compares two strings. If they are equal then true is returned. If they are not equal then false is returned. To be equal, both strings must be the same length.

CHMATCH (str chr -- pos)

This routine scans the string until it finds a match for the byte. If a non zero byte is being searched for this routine will keep looking even if it runs past the end of the string.

HAWTHORNE TECHNOLOGY

LINE EDITOR

USER'S MANUAL

Hawthorne Technology
8836 S.E. Stark
Portland, Oregon 97216

(503)254-2005

Table Of Contents

OPERATING PROCEDURES

This editor has two major modes of operation, ENTER mode and COMMAND mode. Each line of data or command entered is activated by pressing the RETURN key. The backspace key can be used in edit mode to back up and re-work part of or all of a line. This is only good for the current line, so it must be used prior to pressing the return key. Changes on other lines must be made from command mode.

All data is entered from enter mode. The commands EN enter, IN insert, and AP append get you into enter mode. From enter mode, all data is added to the file until a line where the only charactor on the line is a '.' (period) is encountered.

On leaving enter mode, you will be in command mode. Using edit commands while in enter mode will result in the commands being entered as part of the file.

Command mode is designated by a prompt: '*', on the left margin. In command mode the edit commands are used to view or modify the file.

COMMAND MODE OPERATION

IMPORTANT NOTES:

1. In Command Mode, all two letter commands must be terminated with a Carriage Return (CR).

2. Commands are not case sensitive. They may be entered either upper or lower case letters.

3. Commands that require a parameter must have one space between the command and the parameter. If no parameter is entered, a default value that is equal to the previous condition or a value of 1 is used.

4. Commands that work with strings of charactors use delimiters to define the limits of the string. A delimiter can be any non-alpha/numeric charactor that is not contained in the text of the string itself.

5. Many of the commands function relative to the line that is considered current. The 'Current Line Pointer' is always pointing to the current line. If the CLP is at line 1, and you PRINT (display) 25 lines, the CLP is not affected. It still points to line 1. Only commands that are specified for moving the CLP will affect its position.

DESCRIPTION OF EACH COMMAND

DATA ENTRY COMMANDS

Append Command = AP

AP (return) - The append command gets you into ENTER mode. The data you enter is added at the end of the file. You remain in ENTER mode until a line with nothing but a . (period) is entered.

Enter Data Command = EN

EN (return) - The enter command gets you into ENTER mode. Each line you type while in ENTER mode is put after the last existing line. A line is completed when you press the return key. You remain in ENTER mode until a line with nothing but a . (period) is entered.

Insert Command = IN

IN (return) - The insert command gets you into ENTER mode. The lines you enter are inserted directly following the 'current' line in the file. The insertion is terminated by entering a line wiht nothing but a . (period) on it.

MOVING CURRENT LINE POINTER (CLP)

Top Command = TO

TO (return) - Positions the CLP at top of the file and displays the first line.

Bottom Command = BO

BO (return) - Positions the CLP at last line in the file, and displays it.

Here Command = HE

HE (return) - Display the current line.

Down Command = DO

DO n - Moves the pointer down "n" lines and displays the line. (Default is one line.)

Up Command = UP

UP n - Moves the pointer up "n" lines and displays the line. (Default is one line.)

Find Line Command = LN

LN n - Moves pointer to line number "n".

PRINTING (DISPLAY)

Print Command = PR

PR - Prints the entire contents of file.

PR n - Displays "n" lines, starting with the next line after the current line.

FIND DATA

Find Data Command = FI

FI /string/ - Find the group of characters specified by "string" starting from CLP +1. The string specified in the command line must be set off using delimiters. The line on which the "string" is found is displayed. The CLP is now at the found line.

Find Next Command = FN

FN (return) - Find the next occurrence of the "string" that was specified in the prior Find Data (FI) command. This "find" starts at CLP + 1 and displays the line on which the string was found.

Find All Occurrences Command = FA

FA /string/ - Find all occurrences of the "string" in the entire file and display each line on which the string was found. This can be useful in creating a cross reference of particular items in your file.

EDITING COMMANDS

Copy — Command = CO

CO begin,end - Copies the lines "begin" through "end" on to the spot just before the current line, where begin is the number of the first line to be copied and end is the last line to be copied.

Delete Lines — Command = DE

DE n - Deletes "n" lines (default 1) starting with the current line.

Move — Command = MO

MO begin,end - Moves the lines "begin" through "end" to just before the current line, where begin is the number of the first line to be moved and end is the number of the last line to be moved.

New Text — Command = NE

NE (return) - Deletes all old text and restarts the editor. Clears the buffer for new text to be entered.

Replace Current Line — Command = RE

RE (return) - Replaces the current line with the next line you enter.

Truncate Last Lines — Command = TR

TR (return) - Deletes all lines after the current line.

CHANGING STRINGS

Change String(s) — Command = CH

CH /old/new/n - The characters in the new string replace those in the old string. If no value for "n" is given, the change takes place only on the current line. If a value is given for "n", the change will be made for every occurrence of the old string, within "n" lines. To delete a string of characters, instead of specifying a "new" string, use two consecutive delimiters i.e. "//". Any valid delimiter, (a non alpha-numeric character), can be used in place of "/". All changed lines will be displayed.

Caution: The charactor used as a delimiter can not be used in old or new text string.

NOTE: If the substitution applies to all occurrences in an entire file, start at the top of the file and use a number for "n" that is larger than the last line number in the file.

Change By Position — Command = CC

CC c/new/n - The "c" indicates the column number in the line at which point "new" data will start overlapping the original data. Change starts on the current line and changes "n" lines. All of the lines that are changed will be displayed.

ADMINISTRATIVE COMMANDS

Open File Command = OP

OP - Opens a disk file, reads it into memory, and then closes it. This command is used when you want to edit a file that already exists on your disk.

Save File Command = SA

SA filename - Saves the file on disk. A copy of your file is saved on disk. If the file is already on the disk, the copy you have been working on will replace the old copy on the disk.

Exit Editor Command = XX

XX - Exits the line editor and returns command to the operating system.

APPENDIX A

A TABLE OF EDITOR COMMANDS

```
DATA ENTRY
   AP - Append
   EN - Enter Data
   IN - Insert
    . - Exit Enter Mode

MOVING CURRENT LINE POINTER
   BO - Bottom
   DO - Down
   HE - Here
   LN - Goto Line
   TO - Top
   UP - UP

PRINTING (DISPLAY)
   PR - Print

FIND STRING
   FA - Find All Occurrences
   FI - Find Data
   FN - Find Next

EDITING COMMANDS
   CO - Copy
   DE - Delete Lines
   MO - Move
   NE - New Text
   RE - Replace Current Line
   TR - Truncate File Here

CHANGING STRINGS
   CC - Change by Position
   CH - Change String

ADMINISTRATIVE COMMANDS
   OP - Open File
   SA - Save File
   XX - Exit Editor
```

INDEX

HAWTHORNE TECHNOLOGY

68000 ASSEMBLER

HT68K User's Manual

Hawthorne Technology
8836 S.E. Stark
Portland, OR 97216

(503) 254-2005

TABLE OF CONTENTS

INTRODUCTION

The Hawthorne Technology HT68K Assembler is designed to run on the HT21 microcomputer system under 68K-OS. The assembler reads a source code file from a disk. It then generates a machine language object file in the Intel Hex format.

Listing of the assembly is optional. All or part of the code may be listed using the LIST and UNL (unlist) commands in the source file. The listing may be directed to the console or the printer.

Some pseudo-operations are included in the instruction set, such as storage definition instructions and listing control.

The HT68K user's manual assumes that the operator is already familiar with the processors' instruction set, the operating system, and with the general operation of assemblers. Those needing further assistance on these points should consult one of the many books available on these subjects.

OPERATING INSTRUCTIONS

Commands can be given to the assembler on the command line used to start it or can be given in response to prompts. Some options can also be given as directives in the body of the program being assembled.

When commands are given as part of the command line the file name to be assembled is given first. Any file extension can be used. There are no assumptions made. Next all the options are given.

When no file name or options are given on the command line the assembler will prompt for the file name and for the options.

The file name is the path name of the source file. The object file will have the same name with an extension of .HEX if an object is generated.

The list options are C - list to console, P - list to the printer, and N - for no listing. If N is selected then any lines that contain errors and any error messages are sent to the console. The symbol table and cross reference are sent to the same place as the listing.

Cross referencing is selected by an X. Cross referencing adds to the amount of space used by the symbol table and takes extra time and paper to print but does not significantly slow down the assembly.

To supress the generation of an object .HEX file select option O. This will cause the assembler to run slightly faster and will still detect any errors that may be present.

To prevent the symbol table from being printed use the S option. This will prevent the symbol table from being printed. Other than print time it has no effect on the speed of assembly.

C	List to console
P	List to printer
N	No listing
X	Cross reference
O	No object
S	No symbol table

EXAMPLES

```
A>HT68K MONITOR.ASH XP
```

This will assemble file MONITOR.ASH, will generate an object file MONITOR.HEX, will send the listing to the printer, and will create a symbol table and a cross reference.

```
A>HT68K
```

Because the are no command line arguments then the operator will be prompted for the file name and the options wanted.

ASSEMBLY LANGUAGE FORMAT

A standard assembly language source statement contains up to four fields in the following format:

```
LABEL        MNEMONIC        OPERANDS        COMMENT
```

There must be at least one blank character or tab between each field. The label, mnemonic, and operand fields must all be within the first 60 characters of each line.

LABEL FIELD

The label is made up of from 1 to 16 characters, the first of which must be alphabetic. It must start in the first position of the line of source code. If there is no label on a line of source code, the first space must be left blank. Optionally, the label can be followed by a colon. The label must be followed by at least one space, or by a tab character which terminates the label. When the cross assembler identifies a label, it assigns the current address to that label.

MNEMONIC FIELD

This field contains a name which represents either an assembly language directive or a program instruction. The mnemonic field is required, except where the entire line is a comment, since it describes the operation which is to be performed. It begins after the first blank space on the line and ends with a blank space.

OPERANDS FIELD

The operands specify either the memory locations of the data to be used by the instruction, or immediate values. This field begins following the last blank after the mnemonic field. The memory locations can be specified by constants, symbols, or expressions to describe any of several addressing modes available.

COMMENTS FIELD

Comments can be entered following the last blank after the operands field. If the first character position of a line contains a `;' or a `*', then the entire line is considered to be a comment. Although comments are listed in the source portion of the assembler, they have no effect on the generated object code, but are there only for the benefit of the programmer.

SYMBOLS

Symbols are used in the label field, the operator field, or the operand field. A symbol is a string of characters beginning with an alphabetic character, and contains only letters and digits.

EXPRESSIONS

Expressions are composed of symbols and constants with operators. The operators are evaluated from left to right without precedence.

CONSTANTS

Constants may be of various types (i.e. Binary, Octal, Hex, or Character). These types are denoted by the following characters:

(I) Leading:

type	symbol	example
Binary	%	%01010101
Octal	@	@125
Hex	$	$5E

(II) Trailing:

type	symbol	example
Binary	B	01010101B
Octal	Q	125Q
Hex	H	5EH

(III) Decimal

type	symbol	example
Decimal	(none)	12345

(IV) Character:

"A"
`B'
`AB'

PSEUDO OPERATORS:

The following pseudo operators are accepted by the assembler:

DC.B	Define Constant Byte
DC.W	Define Constant Word
DC.L	Define Constant Long
DS.B	Define Storage Byte
DS.W	Define Storage Word
DS.L	Define Storage Long
END	End
EQU	Equate
EVEN	Force Even
INCLUDE	Include
LIST	Listing On
ORG	Origin
PAGE	Start new page of listing
SYMOFF	Symbol Table Off
TITLE	Print Title Line
UNL	Listing Off
UNLIST	Listing Off

The following operators are used when formulating expressions:

+	Add
-	Subtract
*	Multiply
/	Divide
>	Shift Right
<	Shift Left
&	And
|	Or

DC

Define Constant initializes bytes of storage. Each operand must evaluate to an 8, 16, or 32 bit value. For each operand, bytes of storage are initialized to the value of the operand. The DC pseudo op is also used to initialize text strings. Each byte of the text occupies one byte of storage. The text is normally 7 bit ASCII with the high bit set to zero.

DC.B	Byte - 8 bits
DC.W	Word - 16 bits
DC.L	Long - 32 bits

DS

Define Storage is used to reserve a block of memory, but does not generate any object code and does not initialize the storage to any value. If a label is used, it is assigned the address of the first byte of storage reserved. This should be used to reserve RAM locations so as not to confuse a PROM programmer.

DS.B	Byte - 1 byte
DS.W	Word - 2 bytes
DS.L	Long - 4 bytes

END

The last statement in the source file must be an END statement. An END in the main file will terminate the assembly process. An END in an included file marks the end of the included file, and returns to the main line of the program.

EQU

Equate is used to assign a value to a program symbol. The symbol to be defined is placed in the label field, and the value to be assigned is placed in the operand field. The expression used must be known during pass 1, and must be defined before its first use.

EVEN

Even is used to require that the next command land on a word boundry. It checks for even. If odd, a DS 1 is inserted forcing it to be even.

INCLUDE

Include is used to include another file in the assembly code. The next line of code will be read from the included file and will continue until an END is found in the included file. An included file cannot name another included file. For large programs, a main section can be defined that includes the other parts of the program. This enables a large program to be edited in parts so that the source file does not have to be concatenated prior to assembly.

LIST

List is used to turn the listing on if it has been turned off. The default condition is for the listing to be on. When List is on, the listing will be sent to the device specified by the operator in response to the options prompt.

ORG

Origin sets the first location address for the assembly code. If no ORG statement is used, a default ORG value of 0 is assumed. Multiple ORG statements may be used to create separate blocks of object code. An ORG statement may also be used to redefine a prior memory area. The expression in an ORG statement must have a defined value during pass 1.

PAGE

If the listing is enabled, this will force the next line of the listing to start on a new page.

SYMOFF

This turns off the listing of the symbol table which appears at the end of the program listing.

TITLE

This is used to designate a title to be placed at the top of each page of the listing.

UNL or UNLIST

Unlist is used to turn the listing off. It will remain off until it is turned back on with a List directive.

ERROR MESSAGES

An error message appears on the line below any program line which contains an error. There will never be more than one error message per line, even when the line contains more than one error, so be sure to check for additional errors when correcting your programs.

FORMAT:

****** ERROR: [error name]

BAD LABEL

Label contains an invalid character.

MULTIPLE DEFINED SYMBOL

Symbol has been defined more than once.

INVALID OPCODE

Invalid character string in the OPCODE location.

SIZE MISMATCH

Number is too large for register.

INVALID REGISTER

Register is not valid in this context.

RANGE ERROR

Tried to branch to a location that is too far away.

UNDEFINED SYMBOL

The symbol used as an argument is either undefined, or was improperly defined in an EQU statement.

BAD CONSTANT

Constant is the wrong size or the wrong mode.

INVALID ADDRESS

This addressing mode is not valid for this operator.

WARNING MESSAGES

A warning message appears on the line below a program line which may not be interpreted the way the programmer had intended. In general, this is a result of either too little or too much information.

FORMAT:

```
***** WARNING: [warning name]
```

IMMEDIATE OPERAND LONGER THAN SIGNIFICANT LENGTH

The value of the immediate operand will not fit in a byte. The high order bits are discarded.

OPERAND SIZE INDETERMINATE, WORD ASSUMED

The programmer hasn't specified whether the operand(s) are byte or word sized. In most cases, word is the default size.

OPERAND SIZE INDETERMINATE, BYTE ASSUMED

In the case of IN and OUT instructions, the default size is byte.

SYMBOL LONGER THAN SIGNIFICANT LENGTH

The symbol has more than 16 characters. The trailing charaters are ignored.

ADDRESSING MODES AND DATA ORGANIZATION

OPERAND SIZE

The operand size for each instruction is either an implicit part of the instruction, or explicitly defined by the programmer. All explicit instructions support byte, word, or long word operands.

```
     BYTE  =   8 BITS
     WORD  =  16 BITS
LONG WORD  =  32 BITS
```

REGISTERS

In an instruction, the register field specifes which register number is to be used. Other parts of the instruction specify whether it is an address register or a data register and how it is to be used.
Each of the eight data registers is 32 bits wide. They support data operands of 1, 8, 16 or 32 bits. Byte operands use the low 8 bits of the register. Word operands use the low order 16 bits and long word operands occupy all 32 bits.

```
 31         23         15         7        0
 |---- ----|---- ----|---- ----|---- ----|
MSB                                       LSB
```

In bit, byte, or word operations, only the low order portion of the data register is changed. The remaining high order bits are not used or changed.

The seven address registers and an active stack pointer are 32 bits wide. They support 16 and 32 bit operands. When used as the destiniation operand, the entire address register is affected regardless of the operation size.

Registers are commonly identified as follows:

An	- Address Register (n specifies number)
Dn	- Data Register (n specifies number)
Rn	- Any Register, Address or Data
PC	- Program Counter
SR	- Status Register
CCR	- Condition Code Half of Status Register
SP	- Active Stack Pointer (either one)
USP	- User Stack Pointer
SSP	- Supervisor Stack Pointer
d	- Displacement Value
N	- Operand size in Bytes (1, 2, 4)

ADDRESSING

There are two kinds of information contained in 68000 instructions: the function to be performed and the location of the operands to perform the functions on.

The location of an operand can be specified in one of three ways:

1 EFFECTIVE ADDRESS - Use one of the effective address modes.

2 REGISTER SPECIFICATION - The number of the register is given in the register field of the instruction.

3 IMPLICIT REFERENCE - The use of specific registers is defined in certain instructions.

EFFECTIVE ADDRESS

An effective address is used by most of the 68000 instructions. The effective address specifies the location of the operand. The three groups of effective address modes are register direct, memory addressing, and special.

REGISTER DIRECT

DATA REGISTER DIRECT - In data register direct mode, the operand is in the data register that is specified by the effective address register field.

Syntax: Dn

ADDRESS REGISTER DIRECT - In address register direct mode, the operand is in the address retisterthat is specified by the effective address register field.

Syntax: An

MEMORY ADDRESSING

ADDRESS REGISTER INDIRECT - The address of the operand is in the address register specified by the register field. In all but the jump and jump to subroutine instructions, address register indirect is classified as a data reference.

Syntax: (An)

ADDRESS REGISTER INDIRECT WITH POSTINCREMENT - The address of the operand is in the address register specified by the register field. After being used, the operand address is incremented. If the operand is byte sized, the increment is by one. If it is word sized, the increment is by two. If it is long word sized, the increment is by four.

If the address register is the stack pointer, to keep the stack pointer on a word boundary, if the operand is byte sized, the stack pointer is incremented by two.

Address register indirect with postincrement is classified as a data reference.

Syntax: (An)+

ADDRESS REGISTER INDIRECT WITH PREDECREMENT - The address of the operand is in the address register specified by the register field. Before being used, the operand address is decremented. If it is byte sized, the decrement is by one. If it is word sized, the decrement is by two. If it is long word sized, the decrement is by four.

If the address register is the stack pointer, to keep the stack pointer on a word boundary, if the operand is byte sized, the stack pointer is decremented by two.

Address register indirect with predecrement is classified as a data reference.

Syntax: -(An)

ADDRESS REGISTER INDIRECT WITH DISPLACEMENT - One word of extension is required with this address mode. The address in the address register is added to the sign-extended 16-bit displacement integer of the extended word to get the address of the operand. In all but the jump and jump to subroutine instructions, address register indirect with displacement is classified as a data reference.

Syntax: d16(An)

ADDRESS REGISTER INDIRECT WITH INDEX - The address of the operand is the sum of the address in the address register, the sign-extended displacement integer in the low order eight bits of the extension word, and the contents of the index register. The reference is classified as a data reference with the exception of the jump and jump to subroutine instructions.

```
Syntax:   d8(An,Rn.W)
          d8(An,Rn.L)
```

SPECIAL ADDRESS MODES

ABSOLUTE SHORT ADDRESS - The absolute short address mode requires one word of extension. The extension word contains the address of the operand. The 16 bit address is sign extended before it is used. In all cases except the jump and jump to subroutine instruction, this mode is classified as a data reference.

Syntax: xxx.W

ABSOLUTE LONG ADDRESS - The absolute long address mode requires two words of extension. The address of the operand is developed by the concatenation of the extension words. The high-order part of the address is the first extension. The low order part of the address is the second extension word. In all cases except the jump and jump to subroutine instruction, this mode is classified as a data reference.

Syntax: xxx.L

PROGRAM COUNTER WITH DISPLACEMENT - This address mode requires one word of extension. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word. The value in the program counter is the address of the extension word. This is classifed as a program reference.

Syntax: LABEL(PC)

PROGRAM COUNTER WITH INDEX - This address mode requires one word of extension. The address is the sum of the address in the program counter, the sign-extended displacement integer in the lower eight bits of the extension word, and the contents of the index register. The value in the program counter is the address of the extension word. This reference is classified as a program reference.

```
Syntax:   LABEL(PC,Rn.W)
          LABEL(PC,Rn.L)
```

IMMEDIATE DATA - This address mode requires either one or two words of extension depending on the size of the operation. In a byte operation the operand is the low order byte of the extension word. In a word operation the operand is the extension word. In a long word operation the operand is in the two extension words. The high order 16-bits are in the first extension word and the low order 16 bits are in the second extension word.

Syntax: #xxxx

IMPLICIT INSTRUCTIONS

INSTRUCTION	IMPLIED REGISTER(S)
Branch Always (BRA)	PC
Branch Conditional (Bcc)	PC
Branch to Subroutine (BSR)	PC, SR
Check Register Against Bounds(CHK)	SSP, SR
Test Condition, Dec & Branch(DBcc)	PC
Signed Divide (DIVS)	SSP, SR
Unsigned Divide (DIVU)	SSP, SR
Jump (JMP)	PC
Jump to Subroutine (JSR)	PC, SP
Link and Allocate (JSR)	PC, SP
Move Condition Codes (MOVE CCR)	SR
Move Control Register (MOVEC)	VBR, SFC, DFC
Move Alternate Addr. Space (MOVES)	SFC, DFC
Move Status Register (MOVE SR)	SR
Move User Stack Pointer (MOVE USP)	USP
Push Effective Address (PEA)	SP
Return and De-allocate (RTD)	PC, SP
Return from Exception (RTE)	PC, SP, SR
Ret.& Restore Condition Codes(RTR)	PC, SP, SR
Return from Subroutine (RTS)	PC, SP
Trap (TRAP)	SSP, SR
Trap on Overflow (TRAPV)	SSP, SR
Unlink (UNLK)	SP
Logical Immediate to CCR	SR
Logical Immediate to SR	SR

STACKS AND QUEUES

SYSTEM STACK

The system stack pointer is in address register seven (A7). The system stack pointer is either the supervisor stack pointer (SSP) or the user stack pointer (USP). Which pointer is used is determined by the state of the S bit in the status register. Each system stack fills from high memory to low memory. Address mode -(SP) creates a new item on the active system stack. Address mode (SP)+ deletes an item from the active system stack.

On a subroutine call, the program counter is saved on the active system stack. The program counter and the status register are saved on the supervisor stack during the processing or traps and interrupts. They are restored on return.

To keep the system stack alligned properly, data entry on the stack is restricted to word boundaries. If byte data is pushed on or pulled off the stack, only the high half of the word is changed.

USER STACKS

User stacks use an address register (one of A0 through A6). They can be filled from high memory to low memory or from low memory to high. Some things to remember when implementing and manipulating user stacks are:

Using predecrement, the register is decremented before its contents are used as the pointer into the stack.

Using postincrement, the register is incremented after its contents are used as the pointer onto the stack.

Byte data must be put on the stack in pairs when mixed with word or long data so that the stack will not get misaligned when the data is retrieved. Word and long accesses must be on word boundary (even) addresses.

For stack growth from high memory to low, use:

```
     -(An)      to push data onto stack
     (An)+      to pull data off of stack
```

Register An points to the last (top) item on the stack after either a push or a pull operation.

For stack growth from low memory to high, use:

```
     (An)+      to push data onto stack
     -(An)      to pull data off of stack
```

Register An points to the next available space on the stack.

QUEUES

User queues use a pair of address registers (two of A0 through A6). Queues are pushed from one end and pulled from the other. Two registers are used for the get and put pointers.

For queue growth from low memory to high, use:

```
     (An)+      to put data into the queue
     (An)+      to get data from the queue
```

After a put operation, the put address register points to the next available space in the queue. The unchanged get address register points to the next item to remove from the queue.

After a get operation, the get address register points to the next item to remove from the queue. The unchanged put address register points to the next available space in the queue.

If using the queue as a circular buffer, check the address register and if necessary, adjust it before the put or get operation is performed. The address register is adjusted by subtracting the buffer length (in bytes).

For queue growth from high memory to low, use:

```
     -(An)-     to put data onto the queue
     -(An)-     to get data from the queue
```

After a put operation, the put address register points to the last item put in the queue. The unchanged get address register points to the last item removed from the queue. The unchanged put address register points to the last item put in the queue.

If the que is to be implemented as a circular buffer, the get or put operation should be performed first, and then the address register should be checked, and if necessary, adjusted. The address register is adjusted by adding the buffer length (in bytes).