68000

Why use a New OS & the 68000?

by Joe Bartel, Hawthorne Technology

Why Work With a New Operating System?

The small computer market is caught between two ruts today. On the small side is the PC and on the large side is Unix. The other players missed the boat by having a great (or so they thought) interface with nothing behind it to do any useful work. To be PC compatible is a dead end. The system is a kludge.

As developers try to squeeze the last bit of performance from the PC there will be problems. It is true that there are several million PCs in the world today. This doesn't mean there is a good market. Because the market is so large, it is hard (and expensive), for a small company to make themselves heard. There are public domain or low priced programs for every common application that anyone wants. These are hard to compete with. The pressure is to continue lowering prices while cutting profits. A business person needs to look at what point he can no longer afford to remain in this kind of market.

To break out of this rut a new system architecture is needed. Use the PC and clones where they fit but start to forge ahead in new directions. This doesn't mean trying to run a PC program on another machine. It is possible to emulate an 8086 on a 68000 but a full PC emulation is not worth while. In every case so far the emulation costs more than a PC clone. The interchange of disks on the other hand is very economical and easy to do. This protects the investment in data and makes it possible to add new machines without giving up the old ones.

The first step to a new architecture is to have a new operating system. It must be indpendent of a particular piece of hardware. This doesn't mean an operating system that can run on any processor. It means not being tied to a limited set of hardware like MS-DOS got tied to the PC hardware. The second step is to separate the application programs from the operating system itself. To use networks or multiple processors there must be a clear distinction between the logical and physical structure of the machine. To do otherwise would be to set a limit on what can be done with the operating system.

Bit map graphics and mice are good in some cases but to hobble an entire system with tricks that are not often needed or used is bad. The original use of mice was to allow people who knew little about computers to retreive information from them. They were not ones who had to put information into the computer or the more experienced users who want low cost and high performance. The operating systems like Mac and Atari are complex to the point where they hinder the development of new programs rather than helping. The windows that Microsoft has to sell are no better. Look at any stock broker, they have mulitple screens for dealing with different pieces of information at the same time, not tiny windows on a single screen.

A very promising area to look at for the future is multiple processor machines. With them, when more users are added to a system, more processing power is added also. This makes it possible to have multiple access without the slow down problems associated with trying to share a single CPU among many users. For cost sensitive or low performance users the muliple user approach can be used for lowest cost. For applications where high performance is important multiple processors can be used. If the operating system is independent of the hardware then the same program can be used in both cases.

Another area where multiple processors can be used to advantage is to split the operating system into component parts. For example the file management system can be duplicated for each disk in the system. Then when opening a file on disk A there would be no operating system overhead imposed on the system running disk B. If a disk is not involved then it would take no part in the activity. This allows large numbers of users to all access files at high speed if the load is balanced among different disks. A remote disk and file system can be like a new resource that can be easily added and integrated into a system. A company system can start small and grow to almost any size without requiring that the existing parts be replaced.

An individual workstation can have graphics and icons or not as need or tastes dictate. This will allow some users to access the system with icons but not impose that structure on other users of the system. It also means that some users could have windows and others could have more than one screen. Some users could have a local floppy disk or printer too. This approach to things opens a wide area of possible designs for working.

It is time to start planning for the future while the present generation of computers is still adequate for today. If we don't start now we won't have the next generation when we need it. At Hawthorne Technology we are working on new ways of doing things. All of our programs are compatible with K-OS ONE at the system call level. Our hardware varies a lot. We even use PC Clones for some things. But any program that uses K-OS ONE system calls to access the hardware, and doesn't depend on special terminals, will run on any K-OS ONE system. We intend to keep this compatibility in the future for all systems whether distributed, multitask or single task. You can join us in this by using K-OS ONE or by writing applications to run with it. The number of people using K-OS ONE is increasing every day. There is a growing market for Languages and application software. Anyone interested in doing a package should contact us. We will help out in any way we can.

Why Use a 68000?

Most of the time it is not easy leaving an old processor and going to a new one. On the old processor you are an expert and know all the small things that can and will go wrong. When you switch to a new processor you have to start all over again. So why switch?

The 8 bit machines are limited and there is little room to grow with them. The 68000 on the other hand has enough growth potential to last for many years to come. There are other 32 bit processors, but none of the others offer the same advantages as the 68000.

After you start working with it, you will find that building hardware with the 68000 is as easy or in many cases easier than building the same thing with an 8 bit processor. For small controller projects there is even an 8 bit bus version (the 68008) that comes in a 48 pin package. The 68000 and 68008 both have an E clock signal output that allows you to directly connect any peripheral device from the 68xx or 65xx families.

In most cases, the cost of doing the software for a project is many times the cost of the hardware. All of the software cost has to be paid for before the first unit is shipped. Hardware is paid for as units are sold. The 68000 is easier to program than the smaller 8 bit machines. The mistake many people make is the idea that just because you have 16 registers you have to use all of them. You don't. Just use the parts that you want and ignore the rest. After you have some experience you can start using the other commands and addressing modes.

Inline code and programs in general can be as small for the 68000 as for any 8 bit machine. The reason many current programs are so large is that they were written for the 8086 processor family, which is sloppy and many of them are written in higher level languages with compilers that don't generate very good code. What was several lines of code for a Z-80 or 6502 can often be done with a single instruction in the 68000.

The main limitation for the 8 bit machines is the small memory space and the small size of the registers available. If you want to work with more than 64k of memory you have to have registers that are big enough to hold an address. This means that without a full 32 bit register you will spend lots of time and effort working on address pointers that sould be trivial.

The 68000 is much faster than an 8 bit machine for arithmetic and full size pointer manipulation. For simple 8 bit operations like those encountered in text editors it is true that the Z-80 is very hard to beat. But if you need more memory to edit a large file or a lot of features are added to the editor, things become difficult. With a large address space you don't have to page parts of the program into memory from the disk. For spread sheets and arithmetic, the larger register size of the 68000 is faster by far.

When you look at the small cost difference between the 68000 and the older machines the choice becomes easier. Keep the 8 bit machines for existing products or for very high volume or where there is not much programming involved but for new products go with the 68000. It is easier to program, faster, and has more of a future.

Starting With HTPL

Welcome to HTPL programming. If you are familiar with Pascal, Modula or Forth then HTPL will have many parts that you already know. From Forth we borrowed the use of RPN notation for expressions. From Pascal and Modula we borrowed a structure. HTPL is good for writing small, fast programs. It can also be extended to fit any special needs in other programming areas.

To start learning any new language it helps to see a complete example in that language. The example can then be related to the same program in a language you are more familiar with. This example is a simple but complete HTPL program that displays "Hello World!" on the console.

The first line is a comment. When anything is placed in parenthesis in an HTPL program it is treated as a comment and ignored. The word "root" indicates that this is not an overlay, and tells the compiler to include the runtime library with the generated object code. The word "program" tells where the program will start executing when it is run. The main part of the program continues until the first "end". The second "end" indicates the end of the entire file being compiled. The words in the double quote marks are a string constant. When a string constant is encountered the contents of the string are saved in a data area and the address of the string is placed on the evaluation stack. The word "sprint" is a call to a run time library routine to print the null terminated string. The 13 is the numeric value of a carriage return character. It is pushed on the stack. The word "putc" is another library routine that prints the low 8 bits of the top of the stack as a single character. The "10 putc" sends out a linefeed character.

Sample Program

```
root
program
  "Hello World!" sprint
  13 putc 10 putc
  end
end
```

This is a complete HTPL program. When it is run it will display "Hello World!" on the terminal. Most of the tokens are refered to as words in HTPL just like in Forth.

Editor's Note: When this file is compiled, the executable BIN file including the run time library is only 1,446 bytes. This is much smaller than a similar Pascal or C program.

HTPL Compile and Run

To compile and run an HTPL program you first write the program using any editor. The compiler assumes that all characters have the high bit a zero. The output of the compiler is an executable binary file.

To compile a program type HTPL at the command line prompt. After the compiler is loaded it will prompt for the name of the first input file. Next it will prompt for the name of the output file. Any extension can be given for the output file but the command processor will only try to load and execute files that have the extension ".BIN". Next you will be prompted for options. If you enter an "N", there will be no listing of the source code as the program is compiled. If you put an 'S", there will be no symbol table listing after the program is compiled. The options can be given in any order. The complier reads the source program and any files involved twice. The run time library hex file "HTPLRTL.HEX" must be on the default drive for the compiler to find it. An overlay doesn't include the runtime library so it is not needed. You can include as many source files as you want at compile time so each source file can be kept small to be easier to edit.

Stack Notation

The commands in the manual have a comment describing the stack before and after the call to the routine. This is necessary because in a stack oriented programming environment the programmer has to keep track of the stack. Errors in the size of the stack is perhaps the most common kind of error made.

The letters or words before the "--" are the contents of the stack before the call. The top of the stack is on the far right hand side. The words or letters after the "--" are the contents after returning from the routine or after the word is executed. If a word appears before and not after it has been used up. The number of items before and after the call indicate how the stack will grow or shrink when the routine is run. If a routine calls itself then this can be used to estimate how many levels of stack will be required to run the program.

```
EXAMPLE:    a b -- c

     Shows the stack change:   b                 top
                               a       c
                              ---     ---        bottom
                             stack   stack
                             before  after
```

What is RPN, and Why Would You Want to Use It?

There are three different ways to mix operators and the things they operate on: prefix, infix, and postfix. These simply mean that the operator comes before the operands, between the operands, or after the operands. Most of the common languages like BASIC, C or PASCAL are infix languages. LISP is the only common prefix language. Postfix languages, refered to as RPN (Reverse Polish Notation), are represented by Forth, PostScript, and HTPL. The Teco editor used RPN. Adding machines all use RPN and most printing desk calculators use RPN. Each notation has its adherents. So why use RPN?

The compiler for an RPN language is smaller and simpler than the compiler for an infix algebraic language. A large portion of most compilers is a syntax analysis routine that converts the source language to an internal RPN format. If the source is RPN this step is eliminated. When a subscripted variable is referenced a lot of code needs to be generated to calculate the address to use. In RPN these calculations are explicit rather than hidden. For expressions, all an RPN compiler needs to do is push any operand on the evaluation stack and call or generate code for any operator.

In an RPN language, user created operators look the same as the built in operators. When a subroutine package is used to extend an infix language the subroutine calls are very different from the built in operators. If the extensions look the same as the built in operators they are easier to use and the whole program has a more natural look about it. It is easy to create a special set of words for graphics, statistics, mathematics or data base programs. By the time a conventional language has been extended very far it starts to look more like LISP than whatever it started out as. An RPN language in contrast looks the same no matter how far it is extended.

An RPN language is much simpler to learn than an algebraic language. There are no rules of associativity or precedence. The operations are done in the order specified. In the C language there are 14 levels of precedence. Some associate left to right and some the other way. With RPN languages things are much simpler. If it is data it goes on the stack. If it is an action word, the action happens.

In an RPN language the programmer has more control over what kind of code is produced. The sequence of operations is given by the source code. You don't have to worry about the compiler rearranging the order to get better code. Even when using an optimizing compiler you are assured that the operations will be executed in the sequence given.

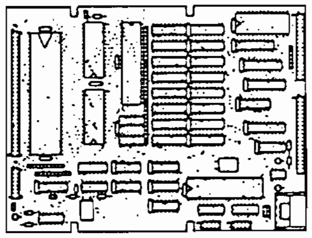
RPN languages are more flexible with the way arguments are passed to subroutines. You can pass parameters by value and parameters by reference in a single call.

The items on the stack become abstract items. They can be used as values or addresses. They can be used as byte pointers, word pointers, long pointers, pointers to structures, or pointers to

68000 SINGLE BOARD COMPUTER

$395.00

32 bit Features / 8 bit Price



-Hardware features:
- 8MHZ 68000 CPU
- 1770 Floppy Controller
- 2 Serial Ports (68681)
- General Purpose Timer
- Centronics Printer Port
- 128K RAM (expandable to 512K on board.)
- Expansion Bus
- 5.75 x 8.0 Inches Mounts to Side of Drive
- +5v 2A, +12 for RS-232
- Power Connector same as disk drive

Add a terminal, disk drive and power, and you will have a powerful 68000 system.

-Software Included:
- K-OS ONE, the 68000 Operating System (source code included)
- Command Processor (w/source)
- Data and File Compatible with MS-DOS
- A 68000 Assembler
- An HTPL Compiler
- A Line Editor

ASSEMBLED AND TESTED ONLY **$395.00**

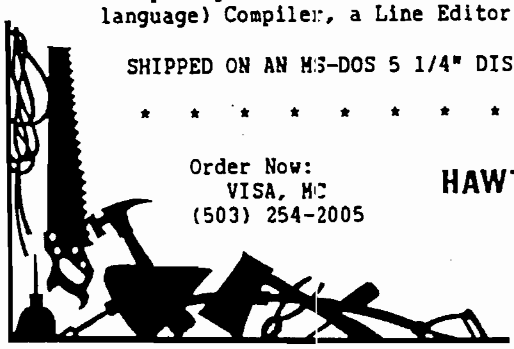
* * * * * * * * * * * * * * * * *

K-OS ONE, 68000 OPERATING SYSTEM

For your existing 68000 hardware, you can get the K-OS ONE Operating System package for only $50.00. K-OS ONE is a powerful, pliable, single user operating system with source code provided for operating system and command processor. It allows you to read and write MS-DOS format diskettes with your 68000 system. The package also contains an Assembler, an HTPL (high level language) Compiler, a Line Editor and manual.

SHIPPED ON AN MS-DOS 5 1/4" DISK. **$50.00**

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



Order Now:
VISA, MC
(503) 254-2005

HAWTHORNE TECHNOLOGY

8836 S.E. Stark
Portland, Or 97216

strings. Different numbers of parameters can be used by the called routine depending on what is found on the stack. A procedure can return a varying number of results depending on what happened. Conventional languages don't offer this kind of flexibility. The 68000 is a very good processor to use with an RPN language. All eight of the address registers can be used as stack pointers. In the other contending micros you have only a single stack pointer and that is used for return addresses. The 68000 also has a very effective set of opcodes that make for small efficient programs.

HTPL has very low overhead on procedure calls. In HTPL there is only a BSR or JSR to get to the procedure and an RTS to get back from the procedure. Any arguments used by the procedure are found on the evaluation stack. This means that there is no need for an explicit transfer of arguments.

Many new languages like Post Script are using RPN because as a subject gets more abstract the use of a stack to hold operands becomes more convenient. The algebraic languages were derived from math equations. When computing is less numeric in nature, it is useful to have a stack for a short term memory to hold what is being worked on.

There are not many books or articles on theory for RPN languages. In many cases this is because writers write about things that are easy to write about. If you look at any book on compilers you find good coverage of syntax and very little coverage of code generating. If you write a compiler you spend lots of time on the code generating and relatively little on the syntax.

Why Not Forth?

Because Forth is the best known of the current RPN languages many of it's quirks are assumed to be in all RPN languages. While some of these disadvantages may be true with Forth, they are not neccessarily true about all RPN languages.

RPN and threaded code are not the same thing. RPN is a way for a programmer to describe the problem to the computer. Threaded code is a technique for generating object code. Threaded code has been popular for Forth on microprocessors because it allows you to create a very fast interpreted instruction set. For 32 bit machines like the 68000 there is no real need to use threaded code.

Incremental compiling is also a technique that is often associated with RPN languages. This was a technique used to create an interactive environment without the slowness of a conventional interpreter. Many RPN languages are now compiled.

As you can see, RPN languages do not need to be feared. The weak points of popular RPN languages have given this method a bad name. One it does not rightly deserve. It may seem like an unnatural method at first. This is due to early mathamatics training. Anyone who has learned to use a 10 key adding machine has learned to use RPN with postfix operators. Most adding machine operators wouldn't recognize the terms, but after their first couple of weeks training, they don't even think about the order they enter the information into the machine. Ask someone you know who uses a 10 key by touch, what order they put the information into the machine. If they don't have a machine they can try it on to find out, they will have to think it through keystroke by keystroke. The actions have become automatic. It isn't so unnatural after all.

HTPL Programming Techniques

When starting to use any new language there are lots of little tricks and techniques that you learn to make it easier and faster to write programs. Some of these are very dependent of the kinds of programs that are being written and some are of a much more general nature. For many languages there are collections of algorithms. These can be used to write a good sort program or to manipulate a data structure.

HTPL is a stack oriented language. If a stack is not a familiar thing, code can be produced by making a literal tranlation of simple algebraic code. Frequently used code sequences can be given a name and made into a procedure. The use of many small procedures results in a slight speed penalty but tends to make the object code generated a little smaller.

Because a stack is so easy to use, there is a tendency to try to do too many things on the stack and save too many values there. You should avoid ever having more than four values on the stack at any time. Saving a value in a temporary variable is not that hard. Having the wrong number of items on the evaluation stack is probably the most common error that occurs in RPN languages. The reset command is used to reset the return and evaluation stacks. Sometimes I use reset as a safty mechanism. When I'm not sure if the stacks are OK I use it to force them into a known good condition.

Strings and Characters

Many of the program modules that are used in the K-OS ONE system deal with strings or character manipulation. In the early days of computing, the processing of numbers was most important. Now it is more important to be able to easily manipulate characters. The stack is used to pass single characters or the address of a string. HTPL uses the C convention where a string is a group of bytes that ends with a null or zero byte. A string is referenced by pointing to the first charcter of it. Also by convention, a valid pointer can never be equal to 0. That is refered to as a null pointer and it means that it doesn't point to anything. Because all stack values are 32 bits long, a string can be kept any place in memory.

The first two routines below deal with adding a character to the end of a string that is being built. In both cases a pointer is used to save the character then the pointer is incremented so it will be ready for the next use. In some cases it is possible to have characters and pointers on the stack. In most cases however it will be easier if either the character or the pointer is in memory. A short assembly routine can be added to the run time library to do many of these things if they are used a lot.

In the first case, the destination pointer is on the stack. The item is placed on the stack. Then the over is used to make a copy of the address to store the character. The !1 uses the character and the copy of the pointer. The +1 then increments the pointer for next time.

In the second case, the destination pointer is in a variable and the character is on the stack. First we get the pointer on the stack. We then duplicate the pointer so that we will have a copy of the pointer as it is. We increment the top copy and store it back in the variable that holds the pointer. The other copy of the pointer that wasn't incremented is used by !1 to store the character.

```
Add a character to a string:

        1. if the destination pointer is on the stack:

                @item over !1 +1
```

```
            2. if the item is on the stack:

                    @pntr dup +1 !pnter !1

Get the next character using a pointer and return it:
  @pntr dup +1 !pntr @1

Change lowercase letters to uppercase letters:
  if dup 'a' 'z' range then 32 - end

Looking for first space, (pointer is on stack):
  while dup @1 ' ' <> do +1 end
```

■

Reader's Feedback

(Continued from page 4)

take a 64 pin header and replace the Z80 with a HD64180Z, if the HD64180Z will run at 4 MHz. I think my 64K chips will baulk at 6 MHz.

If this works I plan to remove the 64K chips later and replace them with 256K chips and bring the system up to 6 MHz along with new ROMS.

At this point I am a systems programmer. Also am learning to type and will try to learn assembly language programming. There is a lot to crowd into my later life. I am 72 now.

What do you think of the idea?

Any help you can give me will be greatly appreciated.

Hiram Desantis
1896 Keewin Ave. N.E.
Palm Bay, FL 32905

How about some of you hardware gurus giving Hiram a hand on this project.

Disk Formats

The big issue around here is disk formats. Number one is getting other people's files moved to the Macintosh network, from IBM, CP/M, HP 3½", Tandy 100 3½", etc. Number two is keeping all the files on the CP/M systems accessible when 8" SSSD seems to be obsolete and the CCS 2422 (at least mine) won't read/write the Ampro or Kaypro formats everyone seems to be using for disk exchange.

If Ampro format is the new CP/M "standard," how about publishing all the details & hints on how to read it? Or a series of programs to force the common controller chips to read it (1793, 765A, etc) and only require the proper I/O ports to be patched in?

Maybe my problem is trying to use my Teac 55G (dBM-AT style 8" compatible) drives in their 5" 300 rpm mode, instead of regular 40 track drives... Has anyone made this work?

L.A.

Jay Sage Fan

The past year has been terrific! Jay Sage is a real boost. His insight and the brilliance of his work is monumental. Also, his understanding of those of us with small TPAs (Osb 1, hard drive) is an uplift.

I have used his new "SUB" to make programs leave ZCPR33, run in full TPA, and then return to ZCPR33, fantastic!

Keep Clark Calkins writing about debugging and Thomas Hilton's educational articles.

Thanks for a terrific year.

A.W.

Z80 User

I have two systems — both Z80 based

My "main" system is a TRS-80 Model II with 256K memory, two 8" 1 Mbyte floppies, and a high resolution graphics board. I run OASIS, a multi-user OS on this computer. My other system is a Televido 806 with two 5¼" floppies, running CP/M 2.2x.

I would like to see articles on interfacing hard disks to SCSI controllers — compatibilitiy of various SCSI based controllers with various hard disks, command sets for the controllers, example drivers, comparisions of different controller/disk combinations for speed and ease of use, etc.

I would also like to see more articles on interfacing speech chips (SP0256) and sound chips (AY-3-8210/8212) to various buses, using IBM keyboards and monitors with non-IBM equipment... (Which reminds me, pin outs and signal descriptions for the IBM keyboard, monitors, and other peripherals would be extremely useful to us non- IBM types to support interfacing attempts.)

L.S.

Miscellaneous Reader Comments

How about a wire wrap video board for the IBM PC Bus?. Possibly using one of the new graphics controller IC chips. Suggest you drop CP/M & Z80.

Would like to see more hardware and software articles for MS DOS, especially AT systems.

I am renewing because you have more articles on MS DOS. I am interested in understanding MS DOS so I can write programs such as device drivers or other enhancements to MS DOS.

Like C.D.M. (Reader Feedback, Issue #27), I was afraid that you might be following in the footsteps of Communications and Electronics, which I once thoroughly enjoyed because of its blend of hardware and software material (generally not too complex for my capabilities).

I was about to drop my subscription but decided to wait for issue #27 before I made my decision. Happily (as owner of an SB-180) Jay Sage's column appeared for a third consecutive issue with promises of continuing regularly. This alone was enough to make me reconsider. Also Jon Schneider's article on the HD64180 put the icing on the cake.

Please don't forget us hardware hobbyists (expert though we may not be).

I am still interested in CP/M stuff (North Star). I recently acquired a Sage II and would be interested in an article on how to install K-OS ONE on it.

My systems are Ampro Z80 Little Board (1A), STD Homebrew, expanded Little Board on STD Bus.

Just in — Hawthorne Little Giant. It's gonna need hack ports of: VDO, Disk7/Sweep, NULU, Small C, Superzap, Fbad, MDM740, Unera, Vfiler, DDT/SID, Config, Multidsk.

I'm designing/programming boards for

(Continued on page 41)