Programmer's Reference saneFORTH™ for Win32® x86 Platforms

as of GreenArrays® arrayForth® 3 rev 03b5

By the late 1980s ATHENA had worked with FORTH, Inc. for years in systems design and implementation. ATHENA was part of the team that defined the polyFORTH model, and ATHENA implemented some of the polyFORTH systems. In 1983, ATHENA was contracted by NCR to implement polyFORTH on the NCR/32 chipset; this turned out to be a microcoded implementation, with microcode that was user extensible on live systems. At that time, FORTH, Inc. had experimented with 32-bit computers but had done them as 16-bit systems with extensions. In the case of the NCR/32 implementation that would have defeated the purpose for which the system had been contracted, so in the process ATHENA established the model that was used by FORTH, Inc. in subsequent polyFORTH systems for 32-bit architectures.

In about 1987, FORTH, Inc. had put polyFORTH on 386 PCs but by then FORTH, Inc was hosting those systems on DOS using DOS or BIOS I/O and accessing extended memory through DOS extender packages. ATHENA's customers required the higher performance obtainable from fully native systems, so ATHENA built its saneFORTH system for software development and as an operating system for production application use. Because both companies considered compliance with a common standard to be beneficial, ATHENA was allowed to produce and offer saneFORTH based on the polyFORTH architecture, and to explicitly acknowledge that pedigree of our saneFORTH systems.

Although native saneFORTH remains ATHENA's mainstay system, in the 1990s we were contracted to make a variant of saneFORTH that was hosted on Unix platforms. This proved that Unix was unable to handle the rate of context switches necessary to support the applications envisioned. However, when ATHENA learned that the execution environment and, indeed, executable file format of 32-bit Windows NT programs differed only trivially from those of the Unix 386 ABI and executable format, we made a variant to be hosted by Win32 platforms as a potentially useful tool for programming Windows boxes to address less demanding applications. This was used to reverse engineer colorForth nucleus source code for GreenArrays, Inc., and has been adopted as the platform for GreenArrays' arrayForth 3 software development environment and its GLOW integrated circuit design package.

Because of this "family tree", documentation for saneFORTH begins with the polyFORTH Reference Manual. This Reference serves as the equivalent of the CPU Supplement for x86 based saneFORTH on Win32 platforms.

Contents

1. Introduction

The following sections discuss the saneFORTH (sF) model in general, its degree of compliance with standards.

1.1 polyFORTH Compatibility

The ATHENA model for the NCR 9300 begat FORTH, Inc.'s model for the 386 which formed the basis for ATHENA's sF series for ISA, MC, EISA, PCI, and Multibus native platforms. With few exceptions, most code written for the polyFORTH systems of the 1987-88 era will run on the sF system. Notable exceptions occur in the following general areas:

- Full length names and their consequences. Linkage between user areas
- System organization and tools provided with the system
- Native I/O (contrasted with pF use of DOS I/O in most existing systems)
- Hosting by Win32 of a text based system that looks and acts like a native system.

In general, study of the polyFORTH Reference Manual will prepare you to read this one.

1.2 ANS X3.215-1994 Compatibility

The ANSI Standard defines a "Core" word set which is a subset of the basic functions in this system. It also defines a number of extensions. Those extensions which we find useful are either already present or will be in the future. Some of these future items may be delayed indefinitely if we continue to see no demand for them. Others of the extensions are orthogonal to our programming practices and cannot be implemented as specified without compromising our environment.

When the term "Standard" appears in this document with no further qualification, it shall be taken as a reference to X3.215-1994.

In a number of cases, the TC unfortunately made choices that break existing code. Other than a few glaring cases regarding global environmental assumptions, most of these instances involve changes in the meaning of well known FORTH words. Our strategy for dealing with these instances has taken the form of a naming convention. When a conventional word `JOE` has been given new meaning by the Standard, we in general provide two canonical names: One for the "traditional" usage, which will be spelled `p_JOE` in recognition of our polyFORTH origins, and another for the Standard usage, spelled `s_JOE`. Which of the two is also known colloquially as `JOE` in the normal `9 LOAD` depends on the degree to which code is broken thereby. Whenever practical the Standard interpretation will be chosen. When that will break known applications the Traditional intepretation will be used colloquially. In each case where the canonical names have been required they are documented, as is the colloquial choice and the rationale. A set of blocks is provided for establishing firm conformance with one environment or the other, and strategies for migration are described.

1.3 System Disk and Source Organization

1.3.1 Boot Areas

For 32-bit machines and emerging hardware for human interface and mass storage, the once-practical 8k bytes assumed by the polyFORTH model proved inadequate had to be enlarged. Others placed the boot out on the disk somewhere, which led to severe reconciliation problems because where "out on the disk" actually was depended on the application. Instead, we made the tradeoff in favor of reserving a boot and nucleus area at the start of each major chunk of disk, so that a copy would follow each system and each backup in a known place. 60 blocks proved sufficient so in a normal environment OFFSET is placed 60 blocks beyond the start of a given chunk containing code.

1.3.2 Larger-Scale Disk Organization

For our current practices, see Mass Storage Management in the next chapter.

1.3.3 Index page commitment

The full system is distributed as a 4800 block drive to be interpreted with OFFSET at 60 and SHADOWS at 2400. Index pages have been rearranged as of 4h; committed as follows:

Status	Page	sF-4g ***released***	sF/NT-00h	RMS 9b	sF-5b ***released***	sF/NT.00i	RMS 9b2
	-60	***Boot***	==	==	***Boot***	==	==
	0	***9-LOAD***	==	==	***9-LOAD***	==	==
	60	***Utility, I/O***	==v	==	***Utility, I/O***	==v	==
	120	***Utility, I/O***	==v	==	***Utility, I/O***	==v	==
	180	***Nucleus***	==	==	***Nucleus***	==	==
	240	***Target, I/O***	==v	==	***Target, I/O***	==v	==
	300	***I/O***	==v	==	***I/O***	==v	==
	360	***Math***	==	==	***Math***	==	==
	420	***Files***	==	==	***Files***	==	==
	480	***Utilities***	==v	==	***Utilities***	==v	==
Floppy (1 TUPLE) only	540	*(Target Output)*			*(Target Output)*		
	540	***Networking***	Make .exe	==	***Networking***		==
	600	***IP - UDP***	API Tests	==	***IP - UDP***		==
	660	***TCP***	(==)	==	***TCP***		==
	720	***Upper Level***	(==)	==	***Upper Level***		==
	780	***Drivers***	(==)	==	***Drivers***		==
	840	***Drivers***	(==)	==	***Drivers***		==
	900	***X Client***	(==)	==	***Drivers***		*free*
	960	***X GUI***	(==)	==	***Net print, test***		==
	1020	***LAN Analysis***	PE Exam	*empty*	***LAN Analysis***		==
	1080	***clusterFORTH***	(==)	**Extens/v**	free		*free*
Floppy (2 TUPLE) only	1140	*(Target Output)*			*(Target Output)*		
	1140	Old MIS	Disk dir for 480	*empty*	File Access		*free*
	1200	iF/VGA	(==)	*empty*	AFS		*free*
	1260	***Word Processor***	*empty*	*empty*	Services		*free*
	1320	MTA In Work	*empty*	*empty*	MTA		*free*
	1380	People/Calendar	*empty*	**clusterFORTH/v**	MTA		*free*
	1440	Cryptography	==	Matrix UDP ulp	Cryptography		*free*
	1500	***Extensions***	==	*empty*	***Extensions 1***	==	==
	1560	Misc Tools	==	*empty*	***Extensions 2***		==
	1620	Books	*empty*	*empty*	free		*free*
	1680	Graphics demos	(==)	*empty*	free		*free*
	1740	***Conc/Floppies***	==	*empty*	free		*free*
	1800	iF/V16	(==)	*empty*	***X Client***		==
	1860	Graphics	(==)	*empty*	***X GUI***		==
	1920	Graphics	(==)	GUI 1/2	free		***GUI 1/2***
	1980	Lab Stds	(==)	GUI 2/2	free		***GUI 2/2***
	2040	Lab Stds	(==)	*empty*	free		XIE Testing
	2100	Trees/uMap	(==)	Wks I/O	People/Calendar		***Wks I/O***
	2160	***SDS Files***	==	Loop I/O	**Word Processor**		***Loop I/O***
	2220	AFS In Work	SCSI diags	*empty*	Books		***Matrix UDP ulp***
	2280	***Bisync***	(==)	*empty*	***clusterFORTH***		**clusterFORTH/v**
Normal (4 TUPLE)	2340	***Target Output***	==	==	==	==	==

Please note that the above organization only applies to general purpose saneFORTH development systems. The version underlying GreenArrays software (arrayForth 3 and GLOW) is considerably edited and redacted for that use.

1.4 Related Documents and Standards

FORTH, Inc. polyFORTH Reference Manual

FORTH, Inc. Intel 80386 CPU Supplement to the polyFORTH Reference Manual

ANSI X3.215-1994 Programming Language FORTH

IBM Personal Computer AT Technical Reference (Generic ISA)

IBM Personal System/2™ Model 80 Technical Reference (Generic Microchannel)

PCI Local Bus Specification, Revision 2 [PCI SIG]

PCMCIA Standards, Release 2 [PCMCIA]

80386 Programmer's Reference Manual [Intel]

80386 System Software Writer's Guide [Intel]

Intel486™ Microprocessor Family Programmer's Reference Manual [Intel]

Pentium™ Architecture and Programmer's Manual [Intel]

Supplement to the Pentium™ Processor User's Manual, Revision 3 [Intel Secret]

MC146818 Real Time Clock Plus RAM (RTC) Data Sheet [Motorola]

Additional hardware references as relevant

Windows 32-bit API Reference

Unpublished documentation on COFF and PE formats

Unpublished documentation on implementing Structured Exception Handling in programming languages

2. Architecture

Right now we are a simple Win32 program which starts in the Console environment. The program is packaged as a PE executable origined at 10000000 hex, and will *not* load on early implementations of win32s (such as the one on old Windows 3.1) because it is not relocatable. It does, however, run fine on all versions of Windows NT, 2000, XP, Vista, 7, 8, 8.5, and Windows 10. Linkage to the system is through static DLL imports sufficient to interact with terminal, disk, and the DLL search mechanism. Further linkages are made during HI by true dynamic linking. It is ironic that by engaging in true dynamic linking (as contrasted with static, executable building time linkage), our nucleus has been tagged as malevolent code by Windows Defender.

2.1 Installation

The basic installation procedure for this system is to expand the distribution zip file into a directory of its own, preferably retaining directory structure in the process since there are documentation subdirectories which are organizationally useful. It is suggested that the ZIP be extracted into a directory on the system drive called Program Files\ATHENA\sF-NT although the "virtualization" done by Windows on 64 bit systems makes updated files, such as our source images, in Program Files difficult to find. We now recommend something more like C:\ATHENA\SF-NT .

The following files are distributed:

1. sfxxx.exe — The main executable for the FORTH system.
2. 4thdisk — Source and shadows, 4800 blocks.
3. doc/ — A directory containing this document and other references.
4. h/ — A directory containing the INCLUDE files whose contents we depend upon.

Once you have expanded the files, make a copy of 4thdisk in the same directory and call it 4thback. This file is opened by the default HI as 4800 block backup area origined at absolute block 24000.

Execute sfxxx.exe by any means (file/run, doubleclick in file manager, invoke in a console window, make an icon) using the directory in which it lies as the working directory.

When the hi prompt appears, type HI .

2.2 Entry from and calls to Win32

Our program is called as a procedure, with entry point at the start of the "text" segment (at 10001000), at which we have placed a jump to the cold start entry of the program proper, which lies at `0PROM 8 +` or 10003008. We save the Win32 stack pointer of the main thread in `u.SP` after pushing some relevant things there. Here is what that stack looks like; the address of its top, labeled `TSP` below, is saved in the user variable `t.SP` where `t.SP` refers to the stack of the thread this task runs on.:

	+2C	Return address (not used; we use u.ExitProcess)	
Reg 7(W)	+28		Registers on entry
Reg 6(I)	+24		"
Reg 5(R)	+20		"
Reg 4(S)	+1C		"
Reg 3	+18		"
Reg 2	+14		"
Reg 1	+10		"
Reg 0	+0C		"
	+8	Exception Handler	
	+4	Registration structure	"exs"
TSP	+0	Thread's FORTH stack pointer save area (was p.SP)	t.SP @
		+3 +2 +1 +0	

When calling DLL entry points, we must hop back onto the stack Win32 gave our thread. We build a call frame below `TSP` and save our FORTH stack pointer at `TSP` , then call the routine in question with the address of `TSP` in "EBP" (Register 5, our R)

which is dutifully saved and restored by all library routines. Thus, the system call mechanism is reentrant with respect to threads, and we are able to identify what FORTH task we are by restoring our stack pointer from TSP on return.

2.3 Defining Library Routines

Since Windows supports dynamic linking, we need not write anything down for accessing a library routine until we determine that we actually need to call it. In this regard we are probably one of very few programming environments in which the linking is *truly* dynamic ... which is to say that a new linkage to a library routine can be made by operator request at any time.

The procedure is to write a section of code that identifies a DLL, defines entry points to be accessible in that DLL, and closes when done with that one. Compilation of that section results in a FORTH word for each entry point, set up to accept the right number of arguments and call the appropriate routine. The DLL will also have been loaded if it was not already mapped into our memory context.

2.4 Nucleus Structure

The nucleus is target compiled in a fairly conventional way. The .exe file is constructed by a separate utility program, described later. For the initial bring-up, targeting and construction were performed on a native sF system and transferred to the NT computer using FTP.

The system can reproduce itself on the NT system, using conventional functions such as:

```
COMPILER LOAD  NT LOAD     (to recompile)
TESTING LOAD  TRY          (to run recompiled system)
RELOAD                     (to warm restart current system)
BYE                        (to terminate process)
```

During target, W0 is the origin of the executable code image; this is the same location as 0PROM. We may need to move 0RAM elsewhere and in fact we may need to make the entire program be initialized data for that matter.

2.4.1 System dependencies in the Nucleus

The initial nucleus boot depends on as little as practical. It imports only the following system calls:

```
4 7 <DLL CRTDLL.dll
5    (  0) +ENT _open  +ENT _close  +ENT _read  +ENT _write
6    (  4) +ENT _lseek  +ENT _getch  +ENT _errno
7 3 <DLL KERNEL32.dll
8    (  8) +ENT GetModuleHandleA  +ENT GetProcAddress
9    ( 10) +ENT GetLastError
```

KERNEL32.dll is attached because the system depends upon it; the call to our entry point is vectored in user mode through a glue routine in that DLL, and the system assumes that we have included it in our import table so that it will be mapped into our address space. In addition, we import the minimum hooks to allow us to dynamically link to the rest of the Universe during HI.

CRTDLL.dll is the runtime C library. It's used for normal disk I/O and for access to the console in simple file mode

These imports translate to the following interface calls:

```
1 0  2 1 UX u.open ( flg ^string - fildes)
2    1 1 UX u.close ( fildes - err)
3    3 1 UX u.read ( lng ^buf fildes - lng)
4    3 1 UX u.write ( lng ^buf fildes - lng)
5    3 1 UX u.lseek ( whence offset fildes - filptr)
6    0 1 UX u.getch ( - c)
7    0 1 UX 'errno
8 4+
```


```
 9    1 1 UX u.GetModuleHandleA ( ^string - han)
10    2 1 UX u.GetProcAddress ( ^string han - a)
11    0 1 UX u.GetLastError ( - u)
```

There are in addition several values abstracted from Include files:

```
11 ~ 0 CONSTANT u.O_RDONLY      ~ 1 CONSTANT u.O_WRONLY
12                      HEX ~ 8000 CONSTANT u.O_BINARY
13 ~ 2 CONSTANT u.O_RDWR        ~ 0 CONSTANT u.SEEK_SET
14 ~ 0 CONSTANT u.STDIN_FILENO  ~ 1 CONSTANT u.STDOUT_FILENO
```

The sources of these values are:

- **FCNTL.H** for the O_ values.
- **STDIO.H** for SEEK_SET
- **VC++ Library Reference** for STDIN and STDOUT file numbers. These values are well known and seem to be safe assumptions regardless of platform.

The above interface is, as it turns out, almost identical to the minimum set for SVR4 unix. The only significant differences are use of **u.getch** instead of read for the console (read worked on unix because we were able to **stty** the operator console into a nice raw character source; there seems to be no comparable facility in this API) and another interesting difference. Unix file opens are by default done in binary mode. Using the NT lilbrary however the default is ASCII (with transformation of cr/lf to lf on reads) so it is necessary to explicitly indicate **u.O_BINARY** when opening files.

2.5 Default HI ingredients

Presently, the system constants and entry points are a bit scattered. The values referenced in SEH should be concentrated with other like things so that there is a single place to find all of the default dependencies.

2.5.1 System Dependencies in HI

The Structured Error Handling mechanism is loaded early by block 9 so that faults and zero divides may be handled. This mechanism depends on Win32 SEH conventions. The only linkage is through FS:0 which is the head of the SEH handler registration block stack.

Should any dependency create a problem, you can say HOME EDITOR instead of HI and will be able to use a cruddy but functional editor to investigate and repair things.

3. Critical Functions and Structures

3.1 Memory Allocation

See above for context. See the later section Building Executables for the exact memory allocation in Windows.

3.2 Major Level Identification

In general, our system changes generate few upward compatibility problems. There are some few cases, however, in which running a new system with an old nucleus, or vice versa, will have catastrophic effects. At 4d level, a new indicator was added to the system:

`mLvl ( - n)` is a CONSTANT whose value is defined in the nucleus and which changes whenever a system update breaks upward compatibility, requiring careful synchronization between the nucleus and the rest of the system; and, in relatively rare cases, small areas of the application.

The presumed value of mLvl for all systems earlier than 4d is zero. The values that have been assigned since are as follow:

1. Introduced at 4d. The dispatcher loop now links user areas with a much faster jump instruction, but the relatively small amount of existing code which alters or walks this loop is broken by the change. To use a 4d or later nucleus the system and application must have the required changes. Once changed, this code will not work correctly on a nucleus earlier than 4d.

2. Introduced at 5a (never packaged as a formal release). Phase 2 Logical Device support added to nucleus. In theory, systems designed for mLvl=1 will run on a nucleus of mLvl=2, but the test for mLvl made in those systems was written to demand 1. To attempt such a combination that test must be altered.

3. Introduced at full 5b. USER area has been reorganized to that P2LDV variables and flags may be properly initialized by CONSTRUCT in an environment where most if not all terminal tasks, including those that create other tasks, use P2LDV.

3.3 Low Level Data Base

3.3.1 System Variables

When behaviors are vectored using system variables, we assume it is obvious to the reader that task specific behavior may be introduced by defining system behavior which then vectors on user variables you have defined. Since this should be obvious it will not be reiterated.

3.3.2 User Variables

3.3.2.1 FLG

Within the single logical device model compatible with that of FORTH, Inc, this cell contains state variables and parameters for the one (and only) character oriented device with which the task communicates. It should only be used for information pertinent to the *character device* as opposed to the *application.* It should be instantiated per logical device, not per task, eventually.

FLG

+0	Function Code							
+1	XOFF	CSI special	TCP device	HDX always	no spont	kibbitz	LDV user	HDX
+2	CSI temp							
+3	CSI temp							
	7	6	5	4	3	2	1	0

LDV is set if this task uses interim Logical Device structure. *In old RMS systems, this bit was used to indicate that status line code had permission to interrupt an EXPECT while the user was typing. Declared obsolete in 1995 but the feature had reappeared in RMS R9d1d2 of 17 Jan 1999 so beware.*

no spont is set to prevent RMS status line code from interrupting other terminal operations.

kibbitz is an obsolete function that was defined on PDP-11; the bit assignment was carried over into the 386 model but has never been used.

CSI Special is set to enable incoming CSI (1B/5B or 9B) sequence processing.

Most other fields and bits above are interpreted in a device specific way. The HDX and TCP related flags apply generally.

manipulation with ~FLG

3.3.2.2 dvCON dvSEL dvERR and dvACT

These variables are only used by tasks that use Logical Device conventions at Phase 2 or higher. Each of these cells contains either zero, indicating none or default, or a logical device instance handle. When the Phase 2 LDV bit is set in UFLG these cells have the following meanings:

dvCON must be nonzero and must reference a valid, usable handle. This device is used by the `QUIT` loop for expecting terminal input and for saying "ok".

dvSEL identifies the device used during normal task execution outside the `QUIT` loop. If zero, the `dvCON` handle is used for that purpose instead.

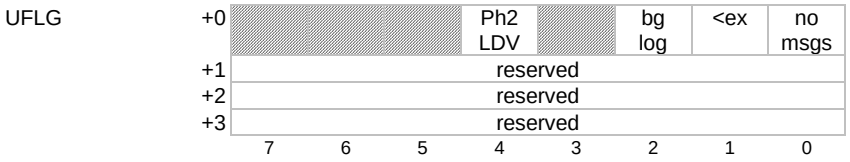
dvERR identifies the device used to display exception messages (as in, between `<ex` and `ex>`). If zero, the `dvCON` handle is used for that purpose instead.

dvACT identifies whichever of these handles is instantaneously active. Normally managed by the system.

As of level 4h, this mechanism is only under construction and is subject to change.

3.3.2.3 UFLG

This cell contains state variables and parameters conditioning operating modes of each task. Assignments are controlled by ATHENA. The default value of this cell is zero unless indicated otherwise.



Ph2 LDV is set to indicate that this task's character oriented device usage conforms fully with Logical Device conventions at Phase 2 transition level. At Phase 2 conformance level, `'TYPE` and `'EXPECT` are not changed to select devices; instead, device selection is controlled by `dvACT dvCON dvSEL` and `dvERR` with protection for `d_DEVICE`. However, the usual user area data base such as `L# C# 'PERS` is still used conventionally and this usage will not be altered until Phase 3 of transition.

bg log is set if task uses standard background error log file.

<ex is set while generating error messages within `<ex` and `ex>` . Inhibits recursion in exception editing and inhibits `THROW` due to character I/O exceptions.

no msgs is set to inhibit spontaneous displays on exceptions detected by the system. It should be set briefly surrounding sections of code that are expected to generate exceptions which will be dealt with via exception handling. For example, error messages resulting from disk troubles or hardware faults are suppressed. See the section on **Exception Handling** elsewhere in this document.

3.4 Task Management and Data Base

The early 386 dispatcher used long jumps for links between user areas, as follows:

STATUS	+0	Abs addr of next STATUS (lo)	EA (dir inter JMP)	2E (IS sg)	Asleep =WAKE
			D7FF (W PIP)		
	+4	Segment (=8)	Abs addr of next STATUS (hi)		
	+8	Saved Stack Pointer when not running			
S0	+12	Base of Parameter Stack; Origin of Text Input Buffer			
		+3	+2	+1	+0

As of sF4d-ISA, the changes made in several earlier custom systems (and the Unix system) have been made such that this is now a direct jump with relative displacement:

STATUS	+0	Addr of next STATUS minus this (STATUS+6) (lo)	E9 (dir intra JMP)	2E (IS sg)	Asleep =WAKE
			D7FF (W PIP)		
	+4	Reserved	High part of that difference		
	+8	Saved Stack Pointer when not running			
S0	+12	Base of Parameter Stack; Origin of Text Input Buffer			
		+3	+2	+1	+0

The motivation for doing this was performance. Each idle task on the round robin costs one jump per `PAUSE` by all other tasks. The base clock ratios for the old versus the new jump are: 27:7 (i386), 17:3 (i486), 3:1 (Pentium). Clearly, with a large number of tasks in the system the intersegment jump instruction was a major source of overhead. For example, the following timings were obtained before and after this change:

	i386/25 sF - 44 tasks		i386/25 RMS - 90 tasks		i486dx2/66 RMS - 89 tasks		Pentium/90 RMS - 90 tasks	
Test Condition:	old	new	old	new	old	new	old	new
PAUSE (µSec)	81.5	38.7	129.75	45.94	45.11	23.35		
Target own sys to RAM (Sec)	7.708	5.767	10.235	8.808	2.720	2.349		
Load & initialize RMS (Sec)			123.02	114.12	42.561	40.121		

The only areas in the system that are affected by this choice are: Definition of the multiprogrammer's dispatcher; initial RAM for the `OPERATOR` user area; task creation, specifically `BUILD`; and the task counting logic in block 508. Certain applications, such as RMS, also contain instances in which application code has needed to walk or alter the roundrobin (something tha occurs only very rarely in our applications). To facilitate their conversion, the following definition has been added:

`^TASK ( a - a)` Given the `STATUS` address of a task, returns the `STATUS` address of its follower on the dispatcher loop. For backward compatability, the following definition may be added to older systems. *It should only be used to facilitate application conversion. The O/S code in 0..540 must track nucleus level on making the transition between 4c and earlier systems, and 4d and later.*

```
: ^TASK ( a - a)   2+ @ ;
```

3.5 Multiprogramming in Win32 Environment

This system has most of the hooks necessary to create a "courteous" round robin within what the O/S thinks is a single task, but it turns out that Win32 does not really support asynchronous I/O. Consequently there is little point in going that route. Instead, as of sFNT01a, there is a new mechanism whereby a TERMINAL task can be commanded to convert itself into a service thread that has its own structured exception handler and which is given work through an EXECUTE vector in its user area, with two Windows event objects to synchronize it (also connected to its user area.) The system call mechanism has been upgraded so that it may be safely used by multiple threads, switching onto the appropriate thread's stack to make the system calls. (Note that, empirically, one thread cannot see another's Windows stack.)

The relevant functions ***will be described later***.

3.6 General Exception Handling

sF supports the ***Exception Word Set*** defined in X3.215-1994. The implementation is fully compliant.

3.6.1 Migration Issues

The definition of ABORT" as given in the Standard will break some applications. This is because the Standard specifies that ABORT" shall display no message whatsoever if there exists an exception frame on the exception stack. Current application usage assumes that *any* section of code may be protected by a CATCH and that diagnostics, if any, will be produced before it executes a THROW for any reason. In addition, the Standard version does not specify display of the last word interpreted.

Accordingly, we provide the canonical names a_ABORT" and p_ABORT" with the colloquial choice being the Traditional version. The system will explicitly use the traditional canonical form in all cases so that its behavior is predictable while the application may choose to use the Standard form colloquially after all inconsistencies have been resolved. System conversion will be staged to avoid creating migration problems between releases.

3.6.1.1 Migration Strategy

The entire question of diagnostic messages in the presence of an exception is practically beyond the scope of the Standard. After considering the matter carefully and discussing it with a number of people, we have concluded that in the majority of cases where exceptions have to do with truly bogus events such as faults or device errors it is vastly simpler to describe the problem at the point where it is detected than it would be to pass parameters describing the problem up the structure of exception handlers expecting that eventually they might be edited. The resource management problems the latter implies are significant, particularly in cases where exceptions occur while handling a previous exception.

In addition, as we have all discovered, ABORT" is appropriate only in certain circumstances. Certainly its traditional behavior of displaying the last word parsed, and no other information besides a static string, circumscribes its usefulness. In common usage we often find that exception displays consist of a variety of output which may, or may not, have ABORT" as its last act depending on whether having the last parsed word displayed actually makes sense. Indeed, it is not at all uncommon to find code which displays a great deal of data and then says ABORT because the action of ABORT" is both inadequate and excessive. Unfortunately, the Standard behavior of ABORT" is inconsistent with such realities since ABORT" is supposed to know whether any exception frames exist, whereas the hard coded displays preceding an ABORT have no such automatic sensitivity nor do they have any standard way of testing for the existence of an exception frame. Neither do *any* of these cases have any way of determining whether the exception frames in question will be capable of doing anything sensible about this particular exception. For these reasons we see no benefit to providing a general tool to test for the existence of exception frames, specifically because the relevance of an exception frame to the appropriate low level action in an exception condition is indeterminate.

This train of thought has led, in various discussions, to the conclusion that in most cases the display or logging of exception details should be handled at the lowest levels, and *then* the exception should be passed upward with a THROW. We have therefore introduced several tools for managing this detail display.

3.6.1.2 New Features

The exception handler has been applied to *all* system generated ABORT conditions. These include all cases in which an exception or error arises that cannot be described to the caller in terms of status, such as primitive functions like BLOCK. Since several of these arise in the nucleus, appropriate support must exist at a low level so that display action is controllable. The long term solution to these problems should be a return to the object oriented DEVICE construction which ATHENA used in the seventies prior to reconciliation with the FORTH, Inc. model for character device I/O. The short term solution has been designed to evolve smoothly into that form. The following inner mechanism has been created and may be used in application code for the same purpose:

UFLG Bit 0 (value 1) is set to indicate that "low level diagnostic messages" are not desired. In both the long and short runs this flag means that the application wants neither the operator nor any log file to accumulate noise consequent to exceptions of any sort whatsoever that might arise during operation. The implication is that the caller is willing to handle and appropriately deal with *any* exceptional conditions that might arise. All system code

that would in the traditional sense have aborted with a diagnostic message now throws an error silently in the presence of this flag.

<ex is vectored through the system variable **'<ex** which is delivered as zero. Used by all system exception detectors.

ex> is vectored through the system variable **'>ex** which is delivered as zero. Used by all system exception detectors.

sF_'ex_xxx is the naming convention for a set of system variables which contain vectors for system handling of the exceptional condition denoted by the controlled term **xxx** . This convention is applied selectively within the nucleus, and its purpose is to allow insertion of additional diagnostic features beyond the scope of those that can be implemented in terms of the character formatting normally done. The usage rules specific to each member of this class are documented separately. In general, the default vector value will be **sF_ex_xxx** .

3.6.1.3 Usage in the Nucleus

Whenever the system encounters a condition that would traditionally have been handled by a simple ABORT" we instead execute the appropriate sF_'ex_xxx vector. As of the base 4d release, all instances of ABORT" were left alone so that they refer to the canonical p_ABORT" function. This takes care of UFLG and hook sensitivities for the majority of system generated exceptions. In subsequent updates those relatively few exceptions that display more than the conventional ABORT" does will be converted to fully cover their displays with the flag and hooks. The default handling routines will in general be structured as follows:

```
: sF_ex_xxx   <ex IF
      {display appropriate remarks}  ex>  THEN  n THROW ;
```

For example, the default handling routines for p_ABORT" are structured as follows:

```
207
 0    ( Hooks)
 1 : <ex ( - t)   UFLG @ 1 AND 0=  '<ex @EXECUTE ;
 2 : ex>   'ex> @EXECUTE ;
 3
 4 : sF_ex_abort" ( a n)   <ex IF  HERE COUNT 1+ TYPE
 5       SWAP COUNT TYPE  CR  BLK 2@ DUP IF
 6          2DUP SCR 2!  THEN 2DROP  ex>  THEN THROW ;
 7
 8 | : p_abort" ( t)   ?R@  -2 sF_'ex_abort" @EXECUTE ;  RECOVER
 9
10 CODE ABORT   -1 # PUSH   ' THROW JMP
11
12 FORTH : p_ABORT"   COMPILE p_abort"  34 STRING ;   TARGET
13 FORTH : ABORT"   COMPILE p_abort"  34 STRING ;   TARGET
```

The default behavior is to generate human readable diagnostics conditioned by $UFLG_0$ and basically ABORT but with the option of application handling of the specific condition ***n*** . The OEM can customize this in two general ways.

The formatted characters describing the exception may be diverted for purposes such as event logging by inserting vectors for the behavior of <ex and ex> . These might, for example, condition the output device to direct its characters toward a buffer destined for an event log. At this level you will be dealing with all exceptions as a class.

The other general strategy is to intercept the entire procedure at the sF_'ex level ; this allows you to write situation specific information gathering agents and to dispose of the data thus obtained in any way you need to.

In terms of stability, your code will become progressively more sensitive to system changes as you take each of the steps above. In particular be warned that beyond manipulation of UFLG your code will be exposed to update requirements consequent to system changes if you use any of the mechanisms beyond UFLG itself. The format of the displayed messages, not to mention

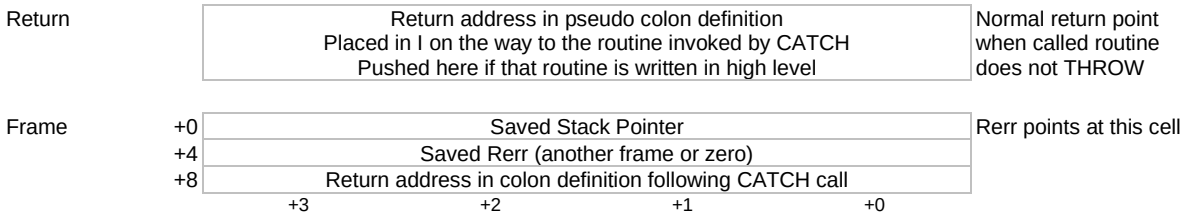
the environment in which low level exception handling routines run, is subject to change with any update. The control this gives you offsets the sensitivity and it is your tradeoff to make as you find appropriate and necessary.

3.6.1.4 Application Usage

The UFLG semantics are stable in the long term as is the significance of <ex and ex> , although the mechanisms and methods for diverting character oriented output are subject to change. Application use of these features is encouraged where appropriate, but the routines stuffed into the vectors will be sensitive to system evolution.

3.6.2 Implementation

Each time a definition is invoked by CATCH an Exception Frame is pushed onto the Return Stack. A user variable, **Rerr** , holds the address of the topmost registered frame, or zero if there is no currently active registered Exception Frame for this task. The Exception Frame is structured as follows:



Exceptions are posted by invoking THROW as described in the Standard. THROW examines Rerr and if Rerr is nonzero the stacks are unraveled to the CATCH point defined by that frame, with results as specified in the Standard. Rerr is updated with the next outer saved value from the Exception Frame. If THROW finds that Rerr is zero when it is invoked, system defined default behavior is taken. This behavior is to silently execute the task's IDLE behavior if the exception code lies within the range for Standard codes, and if we deem appropriate a subset of the system defined codes as well. For all user codes and some of the system defined codes, the default handler produces a message identifying the unhandled THROW code and then invokes the task's idle behavior. This default behavior may be extended by the system during HI. The the default handler is structured as follows:

```
HOST HERE >R  ]   HERE !  SUPPLANT  HERE @ DEH @EXECUTE
   DUP -255 U< IF  CR ." Exception " DECIMAL .  THEN abort [
```

When a definition executed by CATCH exits normally, it actually returns to a pseudo-colon definition whose address was pushed onto the return stack on top of the Exception Frame. This pseudo definition consists of a single compiled reference to a structure equivalent to a CODE definition, which in turn deletes the Exception Frame, restores the stacked Rerr and returns to the caller of CATCH with zero pushed onto the stack.

The hooks employed in the nucleus are relatively simple.

DEH In addition to the system variables, **Rerr <ex ex>** discussed earlier, DEH is created as a vector for extending the default exception handler defined in the nucleus. DEH is intended for use by system software only, primarily to definitively solve problems that require action. An example of such a problem is loss of terminal connectivity with no other action available as defined by the application. DEH *is not available for application use.* Default exception handling by the application, if required, should be addressed in the main program such as for example a redefinition of the QUIT loop using the task's 'IDLE .

abort is the new name for the old ABORT which simply jumps at high level to the address in 'IDLE . This function should *only* be used in cases where it is truly essential to bypass the exception handling mechanism. We expect that such uses will in general be restricted to the internals of the exception handling mechanism within the operating system itself. Since it can be reproduced in an application if need be, the definition has no head.

3.6.3 Windows Traps

This system uses Structured Exception Handling in a language specific way. Our specific way is to abort the task with a message and a dump of status information, throwing an exception (-2). The dump is as follows:

```
FAULT:
   10003321          0          0   C0000005  < 1001D014 >
                              Windows code
   1000C468      32464          0          2  < 1001D024 >
          0          0          0      1007F  < 1001D058 >
        27F          0          0          0  < 1001D068 >
          0          0       FFFF          0  < 1001D078 >
          0          0          0          0  < 1001D088 >
          0          0          0          0  < 1001D098 >
          0          0          0          0  < 1001D0A8 >
          0          0          0          0  < 1001D0B8 >
          0          0          0          0  < 1001D0C8 >
         2B    1438814          0          0  < 1001D0D8 >
         7B         2B         2B         53  < 1001D0E8 >
     W (7)

          0   10004420   6FFFF900   10004430  < 1001D0F8 >
         R1         R2       U (3)      I (6)

         23   10003321   6FFFF8F8   1000442C  < 1001D108 >
         CS         PC       R (5)         R0

      324F6         2B   6FFFF6FC      10202  < 1001D118 >
                              S (4)        flg
```

As is conventional in polyFORTH systems, saneFORTH treats zero divide as a non-problem in those cases where the coding sequence is predictable by returning zero results.

3.6.4 THROW Codes used

The Standard prescribes assignments for specific exceptions in the range [-255..-1], and reserves values in [-4095..-256] for assignment by our system. The values actually generated by released system software in each range are documented here.

Standard Codes	-1	ABORT Some system exceptions generate ABORT	Exception display vector name:
	-2	ABORT" Many system exceptions generate ABORT"	sF_'ex_abort"
	-4	Stack Underflow [Stack empty]	Treated as ABORT" (-2)
	-8	Dictionary Overflow [Dictionary full]	Treated as ABORT" (-2)
	-10	Unhandlable Divide by Zero	
	-13	Undefined Word [?]	Treated as ABORT" (-2)
	-15	Invalid FORGET [Can't]	Treated as ABORT" (-2)
	-33	Block Read exception [Read error]	
	-34	Block Write exception [Write error]	
	-35	Invalid block number	Treated as ABORT" (-2)
ex_IFP	-36	Invalid file position	Also ior code
ex_FIO	-37	File I/O Exception	Also ior code
ex_NXF	-38	Nonexistent File	Also ior code
	-55	Floating-point unidentified fault	
	-57	Exception in sending or receiving a character Generated on NVT disconnect	No error messages (by definition!)

These format messages before throwing and are not identified in the Standard.

System Defined			
Unassigned	-1023	Unassigned	
ex_INHB	-277	Directory Inhabited	
ex_ILL	-276	Illegal Function	
ex_PATH	-275	Invalid Path	
ex_NAME	-274	Illegal Name	
ex_VIOL	-273	Permission Violation	
ex_FULL	-272	File System Full	
ex_RUPT	-271	Corrupt File System	
ex_RES	-270	Inadequate File Resources (handles, etc.)	
ex_HAN	-263	Invalid Handle	
ex_MDEXCEED	-262	Depth of Multilevel Index exceeds limit	
ex_MCORRUPT	-261	Corrupt Multilevel Index	
	-260	Terminal Input Timeout	Silent
	-259	Irrecoverable delivery problem in transparent low level net char I/O	Silent (logically necessary!)
	-258	File pkg: File Full	
	-257	File pkg: Reference outside file	
	-256	Hardware Trap or Fault	

These are normally thrown without any message formatting

System Defined			
TC_E	-4095	TCP/IP Inadequate Resources	TCP/IP Socket operations
TC_E + 1	-4094	Illegal in present state or context	
TC_E + 2	-4093	Timeout (including application level)	
TC_E + 3	-4092	Error	
TC_E + 4	-4091	Reset	
TC_E + 5	-4090	Refused	
TC_E + 6	-4089	Unexpected Disconnection	
Unassigned	-4088		*eight values*
XC_E	-4080	TCP/IP Inadequate Resources	X Client operations
XC_E + 1	-4079	Illegal in present state or context	Messages may be formatted for these; ordinarily, X Client tasks use the XLOG facility to record such messages.
XC_E + 2	-4078	Timeout (including application level)	
XC_E + 3	-4077	Error	
XC_E + 4	-4076	Reset	
XC_E + 5	-4075	Refused	
XC_E + 6	-4074	Unexpected Disconnection	
Unassigned	-4073		*eight values*

3.7 Facility Management

Sharable, queued and registered facilities forthcoming.

3.8 Access to the Win32 API

The code starting in block 144 creates the necessary mechanism to dynamically load DLL's and to resolve their exported entry points. Read the shadows carefully in here. This section is subject to substantial change as we go along, but the methods are solid.

3.8.1 Console I/O

When the system boots we treat the console as *almost* a straight file device using stdin and stdout. Unfortunately there is no analog to the unix **stty** directive for the console and as a result we have had to use a console specific call, rather than the c library _read, to obtain input from the keyboard. Otherwise the console I/O could be initialized for any desired redirection of stdin and stdout. In the context of a unix system, this is all we'd need since addition of a simple VT220 personality will interoperate with most agents we could talk to. Console mode programs may still have full use of the WIndows User Interface if they wish it, fully supported by sF.

However, the Win32 console does not support any control functions beyond CR/LF/TAB/BELL/BS, so its simple boot console acts like glass TTY personality, making the editor rather unpleasant to use. Therefore, in the 9 LOAD we define a new w32con personality that uses the direct console kernel32 calls for screen manipulation. The efficiency is still not as good as we'd like and when time permits we'll see about making typing in general faster (which also means CLEAN and PAGE). In addition, we define a new \EXPECT\ routine for the console which presents an underscore cursor and erases characters. The Win32 console cursor is nearly unusable. While at it, we set attributes to give a familiar color scheme for sF users.

This console should adapt to resizing of the window and of the buffer (which you can alter independently from the dropdown menu.) Characteristics of both are interrogated whenever we invoke PAGE. L/P is set for whatever the window shows, and PAGE sets us for the bottom of whatever buffer you have defined. Tabbing is always done relative to this window position. If the screen scrolls, what you saw moves "up" and can be retrieved by the scroll bar. Data input *does not* change the window position in the buffer.

3.8.2 Thread Messages

3.8.3 Window Messages

Window messages are delivered to a callback routine running in the context either of the window owning thread, or of at least the process owning that thread. The manual is ambiguous about this as it is also ambiguous about entitlement to use any handles. Window messages come both from the thread message loop and directly from other sources. A typical sequence of messages seen at this level is the following:

```
EVM                                                  WM_
     12F9AC          0         24    24403B2  GETMINMAXINFO
     12F9A4          0         81    24403B2  NCCREATE
     12F9CC          0         83    24403B2  NCCALCSIZE
     12F988          0          1    24403B2  CREATE
          0          1         18    24403B2  SHOWWINDOW
     12F9B8          0         46    24403B2  WINDOWPOSCHANGING
     12F9B8          0         46    24403B2  WINDOWPOSCHANGING
          0          1         1C    24403B2  ACTIVATEAPP
          0          0         86    24403B2  NCACTIVATE
         (0          0         37      OLD    QUERYDRAGICON
     12EFA4        1FE          D    24403B2  GETTEXT
          0          1          6    24403B2  ACTIVATE
   C000000F          1        281    24403B2  ???
```

```
         0          0          7    24403B2  SETFOCUS
         0          1         85    24403B2  NCPAINT
        (0          0         37      OLD    QUERYDRAGICON
    12EFA4        1FE          D    24403B2  GETTEXT
         0   63012C62         14    24403B2  ERASEBKGND
    12F9B8          0         47    24403B2  WINDOWPOSCHANGED
   2DC03F4          0          5    24403B2  SIZE
    500038          0          3    24403B2  MOVE
         0          0          F      NEW    PAINT
         0          2         7F    24403B2  GETICON
         0          0         7F    24403B2  GETICON
         0          1         7F    24403B2  GETICON
         0          0         7F    24403B2  GETICON
```

The following table lists all the messages we have both seen and studied. Responses shown are those of the default message handler unless bold and italicized.

WM_ name	code	wparam	lparam	resp	
GETMINMAXINFO	24		^struc	0	max origin, size, drag size; min drag includes decoration
NCCREATE	81		^struc	1	Precedes create. Just copy of create args.
NCCALCSIZE	83	?RECTS	^size or coords	0	new size or posn info,
CREATE	1		^struc	0	After creation, before visible.
SHOWWINDOW	18	1=show 0=hide	Reason flags	0	Heads up on action about to be taken
WINDOWPOSCHANGING	46		^struc	0	New position/size and place in Z order
ACTIVATEAPP	1C	1=activating 0=deactivate	threadID of owner of window affected	0	Informative.
NCACTIVATE	86	1=active			Nonclient (Title bar) state needs to be changed
QUERYDRAGICON	37			han	Return a handle for non default drag icon
GETTEXT	0D	#bytes	^buff	n	Asking us for string assoc window. Return with -DFLT
ACTIVATE	6	hwprevious	lo=action code hi=minimized	0	Informative.
???Unknown	281				IME_SETCONTEXT?' u.
SETFOCUS	7		whprev	0	Informative. We have keyboard focus.
NCPAINT	85				??? Presumably nonclient paint
ERASEBKGND	**14**	**hDC**		**1**	**Commands us to erase background**
WINDOWPOSCHANGED	47		^struc	0	Generates WM_SIZE and WM_MOVE if defaulted.
SIZE	5	why	y|x	0	Size has changed
MOVE	3		y|x	0	Position has changed
PAINT	0F				
GETICON	7F				

Things one may wish to consider processing in future:

NCCALCSIZE can constrain manual resize or position and keep window onscreen and visible; can also specify which part of rectangle is valid and where to put it.

WINDOWPOSCHANGING can alter the proposed action by changing parameters on the way through

GETTEXT reply to this if we monkey with window title.

3.8.4 Windows File System

ANS Forth does include a vocabulary designed as a portable access layer atop arbitrary file system implementations, and we do implement it in some of our systems. In the particular case of sF/x86 on Windows, there are enough variations of file operations in the Win32 API that we have not yet stabilized on a best mapping, nor have we concluded whether there actually is a single best mapping. For the present, our uses of the Win32 file system operators, other than when used with BLOCK , are direct.

3.8.5 Winsock Support

Preliminary Winsock interface lies in 600ff. The intent here is to allow full enough access to permit our interfacing with normal TCP sockets.

3.9 Mass Storage Management

3.9.1 Block Address Space Management

By default, the system supports 32-bit unsigned block numbers. Everything in this section may be changed if necessary for an application, but these conventions are included in the system and in most of our applications.

3.9.1.1 Local Machine Block Address Space

For the local machine, disk class mass storage begins at absolute zero, where zero is the origin of the first file mapped into the DRIVES table. This system presently uses the extended DRIVES table as made for the unix system, allowing any accessible file (local or network) to be mapped into the BLOCK address space. The part of the address space used and the part allowed for access are declared separately (this is important so that files don't inadvertently grow due to erroneous writes, and to allow sensible boundaries.) The entries are:

DRIVES					
	+0	Address space allocation in blocks			-1 for stopper entry
	+4	Usable space in blocks (may be zero)			
	+8	reserved			
	+12	file descriptor usable by c library			
		+3	+2	+1	+0

Block 148 is an example of its use. The current nucleus has room for eighteen files.

Depending on how you open each file when mapping them in, you can implement "hard" write protection on a file-by-file basis by opening some as read-only. Soft write protection has finer granularity, is easier to change when needed, but also has limited scope (by default, only the first 318,000 blocks (318 MB) may be soft protected.

3.9.1.2 Managing Local Address Space

We organize our disks into 1200-block ***partitions***. Four partitions constitute a ***drive*** of 4800 blocks, normally forced to reside on a four-partition boundary. We have learned that it's simplest to organize very large code bases into multiple drives, but some customers have preferred to use a *hunk* of 20 partitions, or 24000 blocks. A ***unit*** is a logical or physical device at the level of the **DRIVES** table and will always be allocated a size in address space which is divisible by the largest relevant chunk size (drive or hunk) that will be used, to facilitate auditing.

3.9.1.2.1 OFFSET Management

BLOCK automatically adds the user variable **OFFSET** to the argument given. This facilitates changing one's point of view, most typically between alternate software images. **SHADOWS** is a user variable specifying the current distance in blocks between source and shadow areas. While **OFFSET** and **SHADOWS** may be manipulated manually, there are several words that facilitate this:

FLEX (- n) exists but is not meaningful on Win32 systems..

TUPLE (n) Sets **SHADOWS** to the value **n** * 600.

PART (n) Sets **OFFSET** to the value 1200**n** with **SHADOWS** at 600.

DRIVE (n) Sets **OFFSET** to the value 4800**n**+60 with **SHADOWS** at 2400. Aborts if part **n** is not divisible by 4. Special values are **-1** which sets **OFFSET** at **FLEX**+60 with **SHADOWS** at 1200, and **-2** when the system has RAM disk; **OFFSET** is at the origin of RAM disk + 60 with **SHADOWS** set at 2400.

HUNK (n) Sets **OFFSET** to the value 24000**n**+60 with **SHADOWS** at 12000. Aborts if part **n** is not divisible by 20.

SYS (n - n) Returns a block number that may be used with the current **OFFSET** to address the block that would have been meant as **OFFSET** was set during the **9 LOAD** .

3.9.1.3 Current Large Scale Practices

There is a trade-off between the needs of a running application, of having multiple storage units on a box, of being able to readily access network disk, and of avoiding traps by making it too easy for a user to err in manipulating block numbers. Our

solution to this is as follows (each chunk size is normally located in BLOCK address space on an absolute origin that is zero mod the chunk size):

Drives: All ATHENA source is organized in **DRIVE**s of 4 **PART**s and are placed on absolute boundaries zero mod 4800 in **BLOCK** address space. For things like network routers this is enough to be a full working environment.

Hunks: The next level up is in 24000 block hunks sufficient to contain multiple sources or larger applications, and working data. We normally allocate these in pairs, with the first being active and the second being a backup of the working HUNK; hence the default values in **DISKING** . We normally access source code in these areas either as **DRIVE**s or as a sequence of 4-**PART** source and shadow tuples; the formal **HUNK** tuple is only used by some customers. A typical complete system will have at least two hunks of code and some amount of space for data base such as concordance, file systems for servers, and so on.

3.9.2 Buffer Pool Management

The nucleus defines a minimal pool of a single buffer. This pool is replaced with a much larger one during HI.

'PREV is a system variable that contains the address of the PREV table.

PREV (– a) returns the current starting address of the PREV table.

NB (– a) returns the address of the NB field in the first PREV table entry.

The PREV table currently consists of a single entry as follows:

PREV	+0	Address of MRU Buffer Descriptor				Forward link
	+4	Address of LRU Buffer Descriptor				Backward link
NB	+8	Number of Buffer Descriptors				
	+12	Reserved for buffer size information				
	+16	Address of Descriptor Table Origin				
	+20	Address of Descriptor for ?UPDATED				
		+3	+2	+1	+0	

The descriptor table is a set of consecutive 20-byte Buffer Descriptors, as follows. The entries are bidirectionally linked with the PREV table acting as a list header. The forward links begin with the most recently used descriptor and progress in order of decreasingly recent access, with the least recently used descriptor always at the end of the list.

Descriptor	+0	Addr of next less recently used descriptor (or PREV if last on list)				Forward link
	+4	Addr of next more recently used descriptor (or PREV if first on list)				Backward link
	+8	STATUS address of task last UPDATEing this buffer Zero if buffer is not UPDATEd				
	+12	Absolute block number residing in this buffer -1 if buffer contains no valid data				
	+16	Address of the physical block buffer				
		+3	+2	+1	+0	

3.9.2.1 Vocabulary for Buffer Pool Management

Most of our functions are compliant with the Standard or are system specific extensions. Critical compliance review will take place later; certainly all multiprogramming implications and usage rules are extensions to the Standard. Usage rules have been normalized for safe operation with buffers larger than 1024 bytes.

3.9.2.1.1 Basic Access Operators

BLOCK (n – a) Returns the address of a buffer containing the data addressed by the given block number, relative to the user variable OFFSET . The argument is added to OFFSET as a signed number and overflows are ignored. The address is only valid until the next explicit or implicit PAUSE. Device errors are thrown as exceptions. A disk write may be necessary to free a buffer for use, and any errors in such a write are treated as exceptions for the task calling BLOCK.

BUFFER (n – a) Returns the address of a buffer identified as the given block. Identical to BLOCK, except that the system may at its option skip reading data from mass storage. The destination block *is not* marked as having been updated.

COPY (s d) The source block ***s*** is copied into the destination block ***d*** . Both numbers are relative to OFFSET and the operation is actually implemented by accessing the source with BLOCK, copying the data to an intermediate buffer, and moving the data into memory obtained with BUFFER for the destination. The destination is UPDATEd. This procedure is required to support long buffers, and for the same reason the old IDENTIFY operator has been deprecated (and deleted from the system).

UPDATE The buffer returned by the most recent BLOCK or BUFFER call is marked as having been updated using the identity of the calling task. It will later be written to mass storage when explicitly committed or when the buffer is needed for another use, whichever comes first, unless the commitment is withdrawn first.

3.9.2.1.2 Committing Data to Mass Storage

There are three degrees of commitment. Experience has proven that the "softest" of these should be called FLUSH since it is the one which is both most efficient and most appropriate in the majority of cases. This differs from the Standard's prescription. For full compliance, it is necessary to define FLUSH as a synonym for MEDIA-CHANGE .

MEDIA-CHANGE Commits all buffers in the system and marks all buffers as empty. This is the most severe of the commitment functions, since it writes all updated data regardless of which processes did the updating, and it further requires that all subsequent BLOCK references re-read data from mass storage. Its intended use is to prepare the system for changing removable media. On busy systems with large buffer pools this can take a good amount of time.

FLUSH! Is synonymous with MEDIA-CHANGE for easier typing.

SAVE-BUFFERS Nondestructively commits all updated buffers in the system to mass storage, leaving their identities set. This is not frequently used, largely because it is considerable overkill for the sort of commitment that is appropriate when committing a transaction.

FLUSH Nondestructively commits all buffers *that were most recently updated by the calling task.* Intended use is for committing data base updates made in a single transaction by the caller, such as altering data base records or indices, or editing a source block.

3.9.2.1.3 Withdrawing Commitment

In an active system, it is not necessarily *possible* to withdraw the commitment of an updated block, since it may have been written due to multiple block accesses by the current task, by block accesses of other tasks, or by explicit commitment functions previously performed by some task. However, with large buffer pools and the general tendency to use the above version of FLUSH for committing transactions, this capability is somewhat more deterministic than it would be if, for example, FLUSH! or SAVE-BUFFERS were routinely used to commit transactions.

We have designed the withdrawal mechanisms to address two quite different needs. In one case, a task may have used a "scratch file" for temporary storage during its operation. When the task is done with its operations on such a file, the data contained in the file are no longer relevant. Ordinarily, the system will eventually write all the temporary data to mass storage anyway, despite its irrelevance. To avoid this, we provide a "hinting" mechanism whereby a task may indicate that it no longer cares about the data in a given range of disk blocks, so that the system may exercise discretion in avoiding unnecessary disk operations. This need can be addressed *deterministically* since the application does not care whether the data have been written or not; therefore, the withdrawal of commitment in this case is satisfied whether or not the data have actually been written.

In the other case, a *programmer* may have made a tragic blunder and would like to un-do as much damage as possible. This is *not* something we can guarantee deterministically, of course, since the data may already have been written by the time the programmer discovers the mistake. The functions provided for use in this case do the best they can, but no guarantees are made.

EMPTY-BUFFERS is suitable for ***emergency use only.*** Marks all buffers empty, regardless of their contents, or whether they have been marked as updated, or which task updated them last. In an active system this will very likely result in

data base corruption which may not be detected immediately and which may cause indeterminate system or application malfunction at some later time. ***Do Not Use this function on a "live" system; if you do, all data that system may have been updating should be regarded as having been compromised just as though a power failure or system crash had occurred.***

-UPDATE (l h – n) Marks all buffers in the *absolute* block number range `[l..h[` which were most recently updated by the calling task as not having been updated, with their block number identities unchanged. This is intended for use within programs to withdraw commitment of scratch files that may be reused later; the identification of the buffers will save additional mass storage reads if they're still around by the time the file is next needed, and costs nothing extra if this does not turn out to be the case. `-UPDATE` should not be used on areas whose prior contents on mass storage are interesting for any reason, since the buffers are not synchronized with the data. For example it should not be used on program source areas.

-BUFFERS (l h – n) Marks all buffers in the *absolute* block number range `[l..h[` which were most recently updated by the calling task as empty. This is intended for use within programs to attempt rectification of mistakes.

UNDO Tries to withdraw commitment of the current `EDITOR` block identified by `SCR`. It uses `-BUFFERS` and is not guaranteed to accomplish anything. If it cannot, displays the message "Too late." Note that if some other task has been editing the block more recently, this function does nothing. Intended for interactive use.

UNDO! Tries to withdraw commitment of *all* uncommitted buffers that were most recently updated by the calling task. This function is intended for interactive use and is equivalent to the phrase `0 -1 -BUFFERS`. Displays the number of writes prevented.

3.9.3 Soft Write Protection

The nucleus supports soft write protection in units of 1200 block partitions, on 1200 block boundaries. This is implemented via a bit table in which each bit indicates whether its corresponding partition is protected (1) or write enabled (0). The table is inside the nucleus and its size is finite. #Lok is the number of partitions covered (bits allocated). Anything outside this range is write enabled. The vocabulary for managing the write protect table is as follows:

WHERE displays the bit table, with 1 indicating write protected, 0 indicating write enabled.

DISABLE disables writing on all partitions controlled by the bit table.

GLOBAL enables writing on all partitions..

+WRT (l h) Enables writing of partitions in `[l..h[` .

-WRT (l h) Protects partitions in `[l..h[` against writing.

3.9.4 Utilities

3.9.4.1 DISKING Utility

The standard **DISKING** utility of polyFORTH has been considerably augmented in saneFORTH systems. Improvements have been made in safety and a considerable source auditing capability has been added.

BLOCKS (s d n) works correctly when the source and destination ranges overlap (starting at the end when moving downward.)

+BLOCKS (s d n) uses **BLOCKS** twice to move corresponding shadows along with source.

+MATCHES (s d n) uses **MATCHES** twice to compare corresponding shadows along with source.

OBLITERATE (l h) wipes blocks **[l..h[** with spaces.

+OBLITERATE (l h) also wipes corresponding shadows.

MATCHING (s d) sets the distance **SEP** between two areas to compare for auditing. **HEAD** holds the upper limit of the area to be compared in the lower of the two areas.

TO (n) sets exclusive endpoint **HEAD** for auditing (default appropriate for system in use).

V displays the current block with differences highlighted.

W toggles between current and other block with differences highlighted.

C toggles but does not display differences.

G advances to the next block with differences. Hit the enter key to punch out.

GIVE replaces the other block with the one you are looking at.

TAKE replaces block you are looking at with the other block.

Z used for accepting changes, equivalent to **GIVE G** except that it aborts if used while looking at the "other" area (the higher block range).

QX AX SX NX BX OX are all overloaded to highlight blocks with differences.

4. Basic Entitlements

4.1 Clock and Calendar

Denied access to hardware in Win32, we must make do with system APIs.

4.1.1 Time of Day and Date

Time and date are obtained from the system. The FORTH API is conventional, and @TIME is corrected for local time zone. The exact choice of system facilities is subject to change, but Win32 gives one few choices for date and time in a form suited for computation. The vocabulary for these operations is as follows:

HOURS (hh:mm) given a double number on the stack in the indicated format, sets the internal time of day and also sets the clock chip.

@TIME (- n) returns the time in ***day clock units***. In addition, maintains internal date rollover; when consistent date and time are needed, `@TIME` should be executed before fetching the date.

@T/MS (- n n) **@T/SEC (- n n)** return ratios *defining day clock units* per millisecond, and per second, respectively.

T/DAY (- n) is the number of day clock units per day.

(TIME) (n - a n) formats a time in day clock units as `hh:mm:ss`

.TIME (n) displays a given time in day clock units as `hh:mm:ss` with a trailing space.

TIME displays the current time as formatted by `.TIME`

4.1.2 Unix Time Stamps

Unix time stamps occur in various network protocols. They are 32 bit numbers counting seconds since midnight UTC at the start of 1 January 1900. These time stamps will roll over the 32 bit boundary on 7 February, 2036. Meanwhile this time unit is a standard that we support.

[time] (t d - s) given @time and MJD, returns seconds since 0000Z 1/1/00. This is the time format used by TCP and UDP TIME function. sF's belief that 1900 is leap is corrected. The input arguments are in local time, requiring that `TZONE` be set properly for this to work.

[now] (- s) returns the present time in those units. Also depends upon `TZONE` .

4.1.3 Elapsed and Free Running Time

Two versions are supplied. The minimal version uses the old Win16 compatible tick count (32 bits of freerunning milliseconds) because this will evidently be supported on Chicago. Block 57 has a version which uses the "Performance timer" which is only available on NT. Interestingly enough, on ISA platforms this is identical to the timer we provide in sF, namely an extension of the freerunning high speed counter chip, giving better than 1 uS resolution; the Windows API does not work around the register updating problem and so, unlike native sF, Windows is capable of giving us time stamps that have negative first differences. The overhead of the high res version is worse than is that of the low res by a factor of six or so. Therefore we use the low res timer for MS and the high res for COUNTER and TIMER at present.

4.1.4 Calendar

Our calendar uses "Modified Julian Dates" (MJD) represented internally as the number of days since 31 December 1899. This is the same convention which is used in the FORTH, Inc. systems but which is mistakenly documented as days since 1 January 1900. The date conversion algorithms have always treated every century year as a leap year, which is incorrect in that only century years divisible by 400 are leap years, and 1900 was not a leap year although 2000 is. This "bug" has been retained because it has existed for twenty years, has been used on all our platforms and applications, and many existing files contain dates represented in this way. Conversions are accurate from 1 March 1900 through 28 February 2100.

MJD values may be stored in 16 bit halfcells but will only represent dates up through 5 June 2079. The day of the week may be determined by taking MJD modulo 7, in which case zero corresponds to Sunday. This mapping is valid throughout the accurate conversion range cited above.

The system is delivered with default date conversions set for the m/d/y model. Block 42 holds the alternative d mmm y conversion routines. The application interface for the default suite is as follows:

M/D/Y (d – n) Converts an external date to MJD form. The external date is a double number with decimal position sensitive coding. It is intended for use with dates entered in decimal and processed by normal number conversion. Viewed in decimal, the input numbers are of the form `mmddyy` or `mmddyyyy` where leading zeroes are required in the day and year fields but not in the month field. These are, by convention, represented in ASCII as `m/dd/yy` or `m/dd/yyyy` but any other method of entering or generating the double number is acceptable. The two forms are discriminated by the sizes of the numbers; anything greater than the decimal value 123199 is assumed to be in the four-digit year form. Years in the two-digit year form are assumed to be in the 20th century.

(MDYY) (n – a n) Formats an MJD date in ASCII using the picture `m/dd/yy` where the month is displayed as one or two digits, month and year as two always, and the year shown is the low order two digits of the actual year regardless of century. Caller's `BASE` is guaranteed saved and restored.

(YYYY) (n – a n) Formats an MJD date in ASCII using the picture `m/dd/yyyy` in the same way as does `(MDYY)` except that the actual year is shown. Caller's `BASE` is guaranteed saved and restored.

(DATE) (n – a n) Is normally a synonym for the canonical form `(MDYY)` and is the default date conversion for the system and existing applications. At some time in the future we may change `(DATE)` to use `(YYYY)` but only if this can be done without breaking applications.

.DATE (n) Displays an MJD in `(DATE)` format with a trailing space.

TODAY (– a) Is a system variable containing the current date in MJD form. Date rollover is normally done within the `@TIME` function, so it is advisable to execute `@TIME` before fetching the date. Many live systems interrogate the time regularly so that this is not necessary. Others generally use date and time for stamping functions, in which case by reading the time before accessing the date rollover is assured.

DATE Displays `TODAY` in `(DATE)` format with a trailing space.

4.2 Capsules

This is a dictionary organization tool. A CAPSULE is a word that, when invoked, makes a given dictionary environment available through vocabulary linkage and any other necessary mechanisms such as memory management or overlaying. Unlike simple vocabulary selection, when a CAPSULE is not selected none of its vocabulary is searched. This allows large bodies of code, such as TCP/IP packages or the arrayForth tools, to be present in memory but to have little or no effect on dictionary search time. On the x86 hardware, our implementation is very simple, using the following vocabulary:

'CAPSULE (-a) a USER variable containing the execution token (XT) of the current capsule.

CAPSULE (_) defines a capsule that stores the vector of dictionary head-links and search selectors within which its words may be found. When executed, a capsule definition makes itself current and executes EMPTY.

EMPTY releases temporary dictionary space and restores search head-links from the current capsule definition.

CAP (_) Saves the current dictionary head-links in the named capsule definition.

IMPORT (_cap _word) defines an alias for _word, as defined in capsule _cap, in the current dictionary.

GOLD a capsule containing the basic system, before application code has been compiled.

4.3 Logical Devices

In the polyFORTH model there is in general a direct mapping between tasks and character oriented devices. Each device has a task dedicated to servicing it. Interrupt code is bound to this task and the data base used for managing the device resides in the USER area of the task, accessible to and maintained by interrupt and task code as appropriate. The methods for operating on the device are vectored through the USER area, specifically `'TYPE 'EXPECT 'CR 'PAGE 'MARK 'TAB` and `'CLEAN`. When a task executes a particular personality by name, the latter five USER variable vectors in that set are stuffed with addresses from the personality definition. The address of the personality definition itself is stored into the USER variable `'PERS` where it may be used to restore those five variables at any time by saying `'PERS @EXECUTE`. In addition there is a data field in each personality definition that we use to identify whether the device is a terminal or printer, and which type of terminal or printer it is. This field has been used by Matrix and perhaps by BMC in selecting additional behaviors (methods) on an ad hoc basis.

4.3.1 Phase 1 Logical Devices

In September 1995 we developed basic Network Printing capabilities including straight TCP ports, LPD servers, and SMTP text only mail. To make this practical, Phase 1 Logical Devices were created. These retained the coupling between one task and one printer, but created the foundation for eventual decoupling. A network printer is an ***instance*** of a ***class***. The instance has a single handle (address) which is stored into a new USER variable **`d_DEVICE`** that denotes the ***currently selected device***. This provides access to any amount of data (instance variables) that are required to operate a device of the class in question so this need not be allocated in every USER area. It also provides access to a set of methods which are appropriate for the class. Naming an instance variable such as `d_CLAS` returns the address of that variable for the currently selected device; naming a method such as `d_TYPE` executes that method on the currently selected device.

In practice, and taken this far, Phase 1 Logical Devices are little more than a means for extending the USER area. The only device methods actually used by any Phase 1 code are `d_TYPE` and `d_EXPECT` which are stored into `'TYPE` and `'EXPECT` for tasks using their dedicated Logical Devices, and `d_OK` or `d_CLOSE` that are used in various systems by `OK` or other words that effectively disconnect from net printers and let the last page be printed. All the other functions that are defined by Personalities have, on Phase 1 Logical Devices, simply presented characters to `TYPE` for transmission to the device.

The only use of Phase 1 Logical Devices has been in Network Printers. They did not create any new capability; the normal means of using a printer remained the practice of using `SEND` or `ACTIVATE` as a means to compel the printer's task to actually produce the output. BMC continued to use a cumbersome method of using `ACTIVATE` to have the printer's task do the actual work for every `TYPE` operation of the terminal task desiring the printing; while this works, it is very cumbersome and makes exception handling exceedingly difficult.

Note that even Telnet sessions are not implemented as Logical Devices. The USER variable DEVICE holds the socket handle for the Telnet connection, and connection variables are all embedded in the socket structure.

4.3.2 Phase 2 Logical Devices

In July through October of 1997 we implemented the sF Server environment including FTP, SMTP, POP3 and SMTP out services and the full set of processing required of an RFC822 Mail Transfer Agent (MTA). This effort finally forced the issue on a single task needing to communicate with multiple character oriented interfaces. For example, a service may need to type and/or expect with two TCP sockets (control and data), read from multiple configuration files, and type into multiple log files. There *were* no tasks associated with any of these things, so the option of adopting the cumbersome method used by polyFORTH was not even an option. Phase 2 Logical Devices (P2LDV) were the result.

Phase 2 does not change the structure or definition of logical devices themselves. It is the* TASK *that is different. If a task is set up to use P2LDV, it may no longer use anything else. Simply revectoring `'TYPE` and `'EXPECT` is insufficient and will most likely lead to spectacular results. The P2LDV task gains many benefits in exchange for this sacrifice.

In the following sections you will learn how to create logical devices, how to use them, and how to introduce them into your existing applications.

4.3.3 Defining Logical Devices

Each Logical Device is an ***instance*** of a ***class*** of devices. These concepts are familiar to anyone acquainted with "object oriented programming". The class structure used here is created specifically for I/O devices and differs from structures designed for other purposes, such as the class/instance structure defined in our X Client subsystem. Each instance is identified by a ***handle*** which is the address of a structure in memory.

4.3.3.1 Data Structures

Each task has several items in its User Area that are relevant for Logical Devices:

FLG	LDV	Bit set if task uses Logical Devices
UFLG	PH2	Bit set if task uses P2LDV. For more info see 3.3.2.3
DEVICE	Used to differentiate devices in the case of single instance legacy wrappers. For example, TCP socket handle for NVT and Telnet wrappers.	
d_DEVICE	Handle (address) of device instance structure whose variables and methods are immediately accessible and effective at the present instant. Application sets it deliberately by using !DEV for access to the innards of a given logical device. It is saved and restored by operations like TYPE.	Saved and restored by normal I/O methods
dvACT	Handle of device that normal functions TYPE EXPECT CR PAGE MARK TAB CLEAN and certain others swap temporarily into d_DEVICE while operating.	"Active device"
dvCON	Handle used by interpreter for EXPECT and "ok". Default for devices below.	"Console device"
dvSEL	Handle used for operations during execution. If zero, console device is used	"Selected device"
dvERR	Handle used for system error messages. If zero, console device is used.	"Error device"

+3 +2 +1 +0

In addition, at this time the USER variables `C#` and `L#` are updated by `TYPE`, `CR`, `PAGE` and `TAB` on all devices, `P#` and `L/P` on some; if they matter, these should be saved and restored across operations on other devices.

4.3.3.1.1 Class Table

Each Class has a name returning the address of a class definition structure in memory:

Class	+0	Number of bytes of instance variables
	+4	Number of methods in following table
	+8	Table of execution tokens, one cell per method, indexed by method number. Undefined methods are zero.
	…	

+3 +2 +1 +0

An undefined method is one whose number is outside the table of a given class, or whose cell in that class definition is zero. When an undefined method is executed, its behavior is the default which is defined as part of declaring the method's name. In this system, every method is given an unique serial number when its name is declared; every class has a table big enough to hold a definition of any method that was in existence when the class was created.

4.3.3.1.2 Device (Instance) Table

Each Instance has a handle that is the address of an instance definition structure in memory, consisting of the variables defined for its class. The following ***universal variables*** are defined for `dc_ROOT` and are present for every class including `hNULL`:

d_CLAS	+0	Address of the class of which this is an instance	Also named d_ORG
d_PERS	+4	Address of a personality table (used in lieu of 'PERS for P2LDV)	
d_FAC	+8	Facility variable available for this instance	No standard usage.
d_C#	+12	Traditional character position (X coordinate)	In user area before full
d_L#	+16	Traditional line number (-Y coordinate)	P2LDV
d_L/P	+20	Traditional Lines per Page	implementation.
	+24	Any additional variables are defined by the class d_CLAS	
	…		

+3 +2 +1 +0

4.3.3.2 Defining Classes

Classes are declared in an hierarchy of inheritance. Here is the coding pattern for building a new class:

```
1  :d_M <newmethodname> <default behavior> ;
2  <more new method declarations>

3  <basis> d_LIKE   ( starting from the class <basis>)

4     n d_v <variablename>   ( addl instance variable n bytes long)
5     <more d_v declarations>

6  <FORTH definitions for any routines to be assigned to methods>

7  d_CLASS <newclassname>

8     d_IS <routine> <method>
9     <more method assignments>
```

The key elements are as follow:

1. Declare new methods, if any, as shown in lines 1 and 2. Each `:d_M` definition declares a new method name (and assigned number). The remainder of the definition is the default behavior for the new method. This is what the method will do if it is executed against a device instance whose class does not have an explicit assignment for the method. Method defaults must be universal, meaning that they will behave reasonably and safely when applied to an instance of any class. Therefore, no default method may depend on nor use any instance variable other than the *universal variables* identified above.

2. Start the new class definition based on some exiting class as shown in line 3. The new class will inherit all of the instance variables defined for the basis class, and will also inherit any method assignments that are assigned for the basis class. The method table for the new class will have enough space to make assignments for every method that has been declared, system wide, as of the time the new class is being declared (hence the requirement to declare new methods before defining the new class that will use them.) Note that `d_LIKE` leaves a number on the stack which is used and maintained by `d_v` and is consumed by `d_CLASS` .

3. Define any new instance variables as shown in lines 4 and 5. `d_v` is like any other normal structure defining word. It is not absolutely necessary that their names be unique but it is a very bad idea to overload them.

4. Define any new FORTH words that will be assigned to methods for this class as noted in line 6, particularly necessary if they refer to any newly defined instance variables.

5. End the new class definition by giving it a name with `d_CLASS` .

6. Immediately, *before making any other definitions*, assign behaviors to any methods that should do something other than their defaults or the behaviors assigned in the basis class. Each phrase starting with `d_IS` makes the FORTH word that follows be the behavior of the following method name for the class just created.

Experience has taught that although class and instance structures with inheritance of variables (attributes) and of methods can be extremely clear and simple to work with, this is only true if we are very careful to never muddy the semantics we are creating as we define these things. Avoid overloading variables, and avoid giving the same variable different meanings in different classes.

Even more importantly, we must keep the semantics of methods clean and consistent. If the purpose of a method is to initialize an instance one time when the instance is created, we must not succumb to any temptation to find other uses for it. Bite the bullet and make a new method rather than changing the meanings of the words for different classes. What the word does may vary with class, and certainly how it does it; but keep things clean so that there is no variation in why we would do it and what

we expect to achieve by doing it. The way to access and nest methods of basis classes is to explicitly invoke their behavioral routines rather than to try and torque the syntax around to access them by method names.

Take this to heart; environments in which these principles are ignored become deadly programmer traps full of latent bugs.

4.3.3.3 The dc_ROOT class and hNULL device

The ultimate foundation class for Logical Devices is named `dc_ROOT`. It defines the universal variables identified above: `d_CLAS d_PERS` and `d_FAC`. The following universal methods are defined and their default behaviors apply to `dc_ROOT` and any class that does not prescribe or inherit any other behavior:

`d_INIT` Initialize a new instance immediately after it has been created (invoked automatically by `d_MAKE` described below.) Typically this means to initialize instance variables if the default presetting, to zero, is not suitable for them. The default behavior is no-op.

`d_TYPE ( a n)` The basic `TYPE` function for this class. May or may not be used for control functions depending on the class; and there may be lower level functions for complex devices such as PDF, but those lower level functions are implemented by other means than class or method nesting. Default is to discard arguments and do nothing.

`d_CR` is included in case a low level new-line function that does not involve personality is useful. So far this has not proven useful and the word has never been employed, so it may eventually be deprecated. Default is to type `0D` and `0A` using `d_TYPE` method.

`d_expect ( a n f)` The basic low level `EXPECT` function with argument that can select optional behaviors like `STRAIGHT ?KEY` or `EMIT/STRAIGHT`. Default behavior is to discard arguments and put the calling task to sleep.

`d_OPEN` initiates communication with the device, if relevant. Default is no-op.

`d_CLOSE` ends communication with the device, if relevant. Default is no-op.

`d_OK` ends communication with a device such as a page printer that may require a specific closure of the last page or other similar situation in which display of characters is deferred until some event such as the end of a page. Default is to invoke personality specific `PAGE` function and then the `d_CLOSE` method.

`d_!EX ( n)` sets an exception code to be thrown (if nonzero) on connectivity problems.

`d_hDUP ( h|0 ior)` replicate the handle for the selected instance if appropriate for this class. Default behavior is to return current instance's handle. For classes requiring it, makes a new instance. *Not used currently.*

`d_hCLO ( ior h|0)` ends use of handle for selected instance. Default behavior is to simply return the handle. For classes requiring it, the handle is annihilated and zero is returned. *Not used currently.*

`?d_DV ( - t)` returns dirty true if calling task uses logical devices.

`?d_P2 ( - t)` returns dirty true if calling task uses P2LDV.

`hNULL` is an instance of this device class, with all its methods defaulted and with `GLASS` for a personality. It may be freely shared and used.

4.3.3.4 Defining Instances

This may be done in one of a number of ways depending on the circumstances. All methods depend upon

`d_MAKE ( class – dh)` Given the address of a class definition, as is returned by its name, creates in the dictionary an instance of that class, returning its handle. The entire allocation of instance variables is cleared to zero, after which it is initialized as appropriate to the class by executing the `d_INIT` method for that class.

Here are some usage examples:

```
1  CREATE MYNULL  dc_ROOT d_MAKE  DROP
2  CREATE FUNNY  dc_ROOT d_MAKE d_DEVICE @!  <setup>  !DEV
3  dc_ROOT d_MAKE  <store handle into a table or use directly>
```

In line 1 we simply make a named instance with nothing special about it.

In line 2 we perform some setup operations on the device which are not covered by its `d_INIT` .

In line 3 we might be making a pool of devices, such as email or PDF printers.

4.3.4 Operating with Logical Devices

A P2LDV task may designate handles for up to three Logical Devices: ***Console***, ***Selected***, and ***Error***. The *Console* device is used by the Interpreter for expecting input and typing "ok". The *Selected* device is used while executing code directed by the interpreter; if the handle is zero, the console device is used. The *Error* device is used for exception display; if the handle is zero, the console device is used. For normal FORTH usage no attention is required at all; simply interact with the interpreter normally. It is generally unnecessary to operate directly on the USER variables to control this because there are words to use instead:

!DEV (dh) makes the given device handle, which must be valid and nonzero, effective during normal code execution by setting `d_DEVICE`, so that instance variables and any explicitly invoked methods are those of that device. The value in `d_DEVICE` is saved and restored during ordinary operations including `TYPE EXPECT CR PAGE MARK TAB` and `CLEAN` so that a device instance may be inspected at leisure without losing the handle on I/O.

>SEL makes the selected device ***active*** by copying the handle from `dvSEL` into `dvACT` (the handle from `dvCON` is used if `dvSEL` is zero.)

>CON makes the console ***active*** by copying the handle from `dvCON` into `dvACT` .

ACTIVE (dh|0) sets `dvSEL` to the value given, either selecting the given device or, if zero, the console for character I/O at execution, and makes it ***active*** using `>SEL` .

Note that `>SEL` `>CON` and `ACTIVE` do not change `d_DEVICE` . If task code wishes to directly manipulate a logical device instance it's necessary to use `!DEV` either instead of or in addition to these functions.

A couple of additional functions are sometimes handy, and are in fact used in the vectored routines for TYPE etc to change d_DEVICE temporarily while executing the appropriate methods:

>DEV< (dh|0 - dh) swaps the given handle, or `hNULL` if it is zero, into `d_DEVICE` and saves the original value of `d_DEVICE` on the data stack so that the instance may be manipulated without selecting it into any of the normal USER variables or affecting `TYPE` et al. After the manipulation is done `!DEV` is required to restore the original handle.

dUSING (dh|0 >r) swaps the given handle, or `hNULL` if it is zero, into `d_DEVICE` and saves the original value of `d_DEVICE` on the return stack so that the instance may be manipulated without selecting it into any of the normal USER variables or affecting `TYPE` et al. After the manipulation is done `R> !DEV` is required at the same return stack level.

d'TUBE and **d'PRNT** return the addresses of the one or two words in the active Personality that encode device type. When used by a normal task these are in the personality that `'PERS` points to. For a P2LDV task, they are in the personality that is addressed by `d_PERS` of the ***active*** device.

Since file handles from our File System may be used as devices, additional methods and functions are defined for manipulating handles. These functions and methods may be used on regular logical devices without causing any harm.

d_hDUP (- h|0 ior) is a method that may be used on any device to return a duplicate of its handle that may be used independently of the original, making a new instance if necessary. For simple instances this is merely a copy of the address of the instance with zero `ior`. For file handles, the method may create a new and separate handle for the same file description. If `ior` is nonzero, it means that the operation failed either because it is forbidden or because of resource exhaustion. Default behavior is no-op.

d_hCLO (- ior h|0) is a method that indicates the handle is no longer needed, its use being ended. For simple instances nothing is done and a zero is returned for `ior` with the handle on top of the stack. For file handles, the handle is freed and zero is returned. Default behavior is no-op.

hDUP (h – h'|0 ior) duplicates the given handle, throwing an exception if it was zero.

hCLO (h – ior h|0) releases the given handle, returning its address if a simple instance. If the given handle is zero, returns two zeroes.

4.3.5 Transitioning to P2LDV

The biggest difference between a task that has been set up to use P2LDV and a normal task is that the vectors in the USER area for the seven classical functions EXPECT TYPE CR PAGE MARK TAB and CLEAN become static, and never changed. The names of the routines that go into these vectors are p2EXPECT through p2CLEAN respectively. Their functions are to execute the d_EXPECT and d_TYPE methods of the *active* device for EXPECT and TYPE, while the functions of the rest are to execute the words that are listed in the Personality that is linked to the *active* device. d_DEVICE is protected by each of these functions.

In order to use P2LDV a task must do the following; examples will be found in the system source code:

1. Obtain a suitable instance handle for the task's console, which may be hNULL but which must be nonzero.
2. Make any required changes to that instance handle, such as copying the contents of 'PERS into d_PERS .
3. Store that handle into dvCON (and ensure dvSEL and dvERR are either zero or valid handles)
4. Execute >P2LDV which makes the appropriate handle active, writes the p2 versions of the 7 classical functions into the USER area vectors, and turns on UFLG bit 4 to indicate that the task uses P2LDV as well as FLG bit 9 in case the task had not previously used Phase 1.

If the task uses a legacy device that is not an instance of an LDV class, it may use an instance of a wrapper class **dc_LEG** . There are rules: dc_LEG must be instantiated once for each old pF style character driver (e.g. each requiring a distinct combination of 'TYPE 'EXPECT and Personality). The TYPE and EXPECT vectors for the driver must be stored into the instance's dl_'TY and dl_'EX . Other methods come from the Personality. There is a very significant limitation: A task may use only one Legacy device because all the other variables and flags such as DEVICE PTR CTR FLG and so on which hold driver state and context information are still in the USER area. In order for a task to use two legacy devices, at least one of them must be converted to be a genuine Logical Device.

These wrapper instances already exist:

hlNVT for TCP Network Virtual Terminal tasks, socket in DEVICE , personality GLASS

hlTEL for Telnet tasks, socket in DEVICE , personality TNVT

hlVGA for OPERATOR task, many USER variables, personality EGA

Note that in cases where system facilities are affected by P2LDV, such as PERSONALITY and OK , the words have been altered to check the FLG and UFLG states, acting accordingly. Thus after a task has executed >P2LDV its character I/O vectors are no longer changed when a PERSONALITY is executed, and the address stored is not 'PERS but d_PERS .

Additional tools used in creating tasks that will use P2LDV are as follow:

>P2 (tdb) activates the given task to elaborate itself as a P2LDV task, given that d_DEVICE and 'PERS have been set. The latter is redundant and will soon be deprecated as a trap.

d_term (dv n _ _ - dv a) makes a P2LDV terminal task set up to use device dv and the named personality and idle behavior. n is dictionary size. Returns address of TDB for newly constructed task.

d_TERMINAL (dv n _ _ _ - dv') makes a named P2LDV task given the args for d_term . Leaves its device selected with own task device on stack!

4.3.6 Phase 3 Logical Devices

These will only be implemented if a truly compelling reason has emerged, because they would break too much code for other sites and would intrinsically require rewrite of all character drivers into P2LDV compatible form. When it *is* implemented, the goals will be several:

1. Make USER variables like `L# C# P# L/P TOP CURSOR ATTRIBUTE` instance variables. Problems are their direct use in drivers and interrupt code for assorted devices, not all of which code was previously under ATHENA control. (In P2LDV, it was necessary to anticipate this by making C# "smart" about the flags, returning the USER variable if not using logical devices, the d_C# attribute of the *selected* device if phase 1, or the d_C# attribute of the *active* device if phase 2.)
2. Integrate the structures of socket handles so that they are in effect logical device instances themselves. This has already been done with file handles.
3. Clean up the distinction between a communication channel and a device protocol; presently the personality is trying to do a little of both.
4. Eliminate all other USER variables like `DEVICE` that are employed by old character drivers.
5. Eliminate USER vectors for classical functions. This then makes the set of device floated functions an open set with no need to mess with USER area or retrofit defaults in order to make new classes and operate on them in that same way.
6. Provide for layering of logical devices as implied by PDF over MIME over RFC822 mail over TCP. This would most likely be the "compelling reason".

4.4 Disk Directory

A simple mechanism for naming disk areas has been used for years and has proven adequate. It depends on the notion of putting some interpretable text at the start of each area of disk containing one or more named areas and parsing those text blocks as is done by the following words; see the code for the set of possible text locations on this system:

.INDEX parses all of these text areas and produces a directory listing, as for example:

```
     File             Origin  Records  Bytes  Blocks      At    Gap

     .Working-Sys....      0    10800   1024   10800
     OWL-LOGO          19284      204     32       8   10800   8484
     EARTH-SHORT       19292     8132      8      64
     CHARACTERS        22800       78   1024      78   19356   3444
     COLOR-TABLE       22878    32768      2      64
     ................  24000        0   1024       0   22942   1058
     .Concordance.... 240000        0   1024       0   24000 216000
     conc-WORDS       240000    60000     32    1875
     conc-CRUD        241875      320     32      10
     conc-XREF        241885   614400      4    2400
     ................ 264000        0   1024       0  244285  19715
                                                      264000-259200
```

NAMED (_ - n) returns the starting absolute block number of the given named area.

Named (_ - b r n o) returns file declaration parameters for the given named area. (Not all areas are organized or accessed as data base files; this word is only for use with those that are.)

4.5 Fixed Point Arithmetic

Because all mainline x86 processors have supported the numeric (floating point) instruction set, beginning with the 80387 coprocessor for the 80386, and with internal support in the main CPU since the 80486, and because in most cases the numeric instructions are actually competitive in performance with fixed point, this system depends on fixed point software math routines less than do most of the polyFORTH systems. Our standard basic fixed point arithmetic is present, configured for 30-bit fractions and the usual range of [-2..+2[as well as support for 30-bit angles because fixed point is much better suited for things like celestial mechanics than is floating. This code is loaded along with and may be found co-located with the floating arithmetic.

4.6 Floating Point Arithmetic

FLOATING compiles the floating and fixed point arithmetic and extensions, making them part of GOLD , the first time it's executed per boot. Thereafter FLOATING is a no-op.

This is ATHENA's implementation which makes direct use of the 8-element floating point stack, rather than simulating a floating point stack in memory, therefore maintaining full precision at a much lower cost than is levied by the latter. It is a constraint, but has proven to be a good trade-off; for example, we used it to successfully fight a simulated nuclear war, losing only Minot ND and Los Angeles CA. The code may be found in the index page at 360 and is well shadowed.

4.7 Data Base Management

The Data Base Management package provides mechanism for managing the content of structured files. The most basic structure is a flat file with fixed length blocked records. Based on that structure, both ordered and multilevel ("B-tree") indices are supported.

4.7.1 Flat Files

Flat files are declared and manipulated in a manner consistent with that in classical polyFORTH systems.

4.7.2 Ordered Indices

These also are consistent with polyFORTH.

4.7.3 Multilevel Indexing Package

The Multilevel Indexing Package is normally available for each FORTH system supported by ATHENA, is currently recommended technology, and is actively supported. Applications with relatively little file sharing may generally use it as supplied. Large or complex environments will often need application specific adaptations, as discussed later.

4.7.3.1 Purpose

Ordered indexes (OI's), while suitable for small files and simple applications, become intolerably inefficient when the files become large or many users share them. The number of block calls and potential disk accesses required to search for a key in an OI is basically the log to base two of the number of keys. Thus for one million keys about 20 block calls are needed. Assuming 12 byte keys, so that 64 keys fit in a block, it is likely that fourteen or fifteen of these block calls will read the disk. This might take perhaps 280 mS on an idle system, and lots more if there is heavy disk activity. Updating this particular index is far worse; the example just discussed is on the order of 15K blocks, and a random insertion or deletion will need to read and write a mean of 50% of that. So we would be on average doing 7500 disk reads and writes per insert or delete, or nearly four minutes on an idle system.

One of our clients has tried using two levels of OI's; in the above case, *with well balanced indexes,* any particular operation would need to deal with two files of about 1,000 keys (15 blocks) each. This would tend to reduce search time to ten block calls, perhaps 4 or 5 hitting the disk, in each of the levels for a total of 8 or 10 disk operations and perhaps 160 to 200 mS for a search in an idle system, up to twice as good. Insertions and deletions fare far better; the probability of needing to maintain the high level index is infinitessimal, so in general we're talking about reading and writing 7.5 blocks per operation for a total of perhaps 210-300 mS in an idle system.

While this was a great improvement, it added a good deal of complexity to the application. Further, file imbalance and overflow opportunities are extremely great, with attendant additional complexity and lost time when those things occur. Finally, even in this structure we are still looking at something like ten disk reads for a search and fifteen disk operations for an insert or delete. In a heavily loaded system with lots of disk activity, the search and insert times can easily grow to several seconds due to contention for the disk.

Another of our clients with a real-time application was experiencing severe index performance problems in 1986, and ATHENA invented the Multilevel Indexing package as a relatively simple and robust solution to them. Simply stated, this package supports a radix-n search (where n is the number of keys per block) as opposed to radix 2 in the OI case. Levels are automatically added as needed. In the above case, with 64 active keys we would need one block and one level. With 64*64 or up to 4096 keys, we would need 65 blocks and two levels. At one million keys, we would have a minimum of 15875 blocks and four levels. We could handle a maximum of 16 million keys before a fifth level was necessary.

As will be seen later in the implementation description, this means that a search for any of the million records would take a maximum of four disk accesses, better than half what's required even with two layers of OI's and three to four times fewer than required in a single OI. This reduces search time exposure to disk loading by a commensurate factor. More importantly, *the majority of updates require only writing a single disk block.* This vastly reduces insert times, especially in heavily loaded systems.

It has been our experience, even in systems with good support for sharable facility control of files and particularly indexes, that insertion and deletion operations requiring many disk hits have catastrophic effects on system performance. This follows because (a) there is a potentially great multiplier on disk operation time produced by contention for the disk channel, and (b) insertion and deletion are essentially indivisible operations requiring exclusive use of the index, so that *other terminals* lose considerable search performance when insertions and deletions are going on.

Thus, the purpose of this package is to reduce the number of disk operations for index searching and maintenance to an absolute minimum while keeping the structure relatively simple to use and maintain. After the package had been implemented we discovered that what we had accomplished was essentially to independently discover "B-trees." Actually, though, this package is simpler in structure than is the one described by Knuth (all data references occur at the same level of the index) and simultaneously embodies many of the possible enhancements he suggests at the end of his discussion. Since our design is *not* literally that of the "B-tree" we do not call it that.

4.7.3.2 Implementation

A multilevel index lives in a standard, random access data file. Records are described, referenced, and accessed normally. Space is, however, allocated in units of *blocks* rather than individual records. ***Block zero*** of each index file is used entirely for administrative purposes. Remaining blocks are either free, or are filled with key records.

Each ***Key Record*** in the file contains a 32-bit **`LINK`** field at the beginning, followed by a fixed length ***Key Value***. On some implementations the hardware virtually compels record alignment on two- or four-byte boundaries for efficiency and simplicity, so in general the keys will need padding to the correct modulus. (The package as distributed for the 386 works in 16 bit units, requiring even key length with any character strings byteswapped.) *Two key values, namely all binary zeroes and all binary ones, are **absolutely reserved** and **may not** be created **under any circumstances** by the application.*

Code using this package is responsible for the contents of key records that actually reference data. Key values ***must*** guarantee never to use the two reserved values. In addition, *the `LINK` field **must** contain* a ***positive, nonzero*** *value.* Beyond that, key and link values are entirely the property of the application. The application presents these keys to the package and speaks in terms of searching, inserting, and deleting them.

The package is itself responsible for placing and maintaining these user-created keys on disk, and for maintaining the structures necessary to find them. The application can largely ignore these matters. However, certain patterns of use can lead to fragmentation of the file, causing slightly increased search times and potential premature exhaustion of space. Utilities discussed below are available to monitor and deal with these situations, but the basic package does *not* automatically deal with them in real time to avoid surprising the application with unexpectedly great latencies. See also the Usage Recommendations below.

4.7.3.2.1 Declaration and Initialization

Index files are declared in the usual way with **`FILE`**. The `LIM` value ***must*** specify a number of records that exactly fills all of the blocks allocated for the file. That is to say that it ***must*** be divisible by `R/B`. The `B/R` value ***must*** be larger than four, rounded up to the next larger even or modulo four value as mandated by the package version in use (even in the case of the 386 distribution). The key field length as regarded by the package is always four less than the value given for `B/R`.

The word **`+nARY`** should be used immediately after `FILE` is declared. In some systems, `+nARY` may allot additional space in the file description for use by the package. In all systems, it serves the additional diagnostic purposes of checking the `LIM` and `B/R` values for legality.

Index files are normally initialized using **`N-INITIALIZE`**. In addition to the normal sort of initialization done for flat files, this word sets up *Block Zero* and the first ***Key Block*** so that the file describes a properly empty index. The initial conditions of these two blocks are described in detail in the following sections.

Alternatively, it may be necessary to rebuild an index in a hurry. There are basically two ways to do this. The most straightforward is to simply re-initialize the index file using `N-INITIALIZE` and then insert all of its keys using

normal mechanisms. For larger files, it is a bit quicker to build a flat file of keys and sort them. To support this process, a utility is provided that finishes the process of converting such a sorted flat file into a usable multilevel index.

4.7.3.2.2 Block Zero

Block zero contains, at minimum, **AVAILABLE** (as a 32-bit value) and a new field called **ANCHOR** . ANCHOR consists of a 32-bit record number followed by a nominally 16-bit level count.

AVAILABLE will always contain a value divisible by R/B since space is always allocated in units of blocks. As is usually the case with flat files, AVAILABLE will ordinarily contain the starting record number of the last allocation performed. When an index has been initialized, AVAILABLE will contain the value R/B since block 1 of the file will have been allocated as a key block.

The record number in ANCHOR will be the first record number of some block and will therefore always be divisible by R/B . The block it points to will ***always*** be the *top level index block.* This is the first block that must be searched to find the actual data reference for a particular key. The level count in **ANCHOR 4+** is the number of levels of blocks that must be searched before getting to a block containing keys that reference data. Other than block zero, the total number of key blocks that must be examined to evaluate a key to a LINK will be one greater than this level count. When an index has been initialized, ANCHOR will contain the value R/B meaning that the top level index block is block 1. The level count in ANCHOR 4+ will be zero since block 1 is also a *bottom level index block* containing actual data referencing keys.

In some systems, a copy of ANCHOR is maintained in the file definition (or elsewhere that occurs only once per file) so that Block Zero need not be accessed to do a search. In such cases, the word ANCHOR will return the address of this place in memory. *This should not be an issue, since user code has absolutely no business referencing any of this package's internal data base.*

4.7.3.2.3 Key Blocks

Free blocks, meaning blocks that are not part of the active structure, are marked as free in whatever way is supported by the file package in use. Beyond that we do not specify. *One reason why the LINK value of zero is reserved is that some of these systems mark free blocks with a zero LINK value.*

Active blocks, meaning blocks that *are* part of the structure, contain exactly R/B key records. The key records within a block always occur in ascending key value collating sequence order, where the key values are considered to be long unsigned numbers. Any unused key records in a block are grouped together at the end of the block and are filled with ***Stoppers.*** A Stopper in this structure is a key record whose LINK field is all zero and whose Key Value is all binary one's. The all one's value is chosen since it is the largest possible and facilitates any search paradigm. The zero LINK value is chosen since it's already reserved and is far cheaper to test for than is the all one's key value.

There is no such thing as an Active block that contains nothing but Stoppers. For one thing this would violate the rules since in that case the block's first LINK field would contain zero, making it look like a Free block in some systems. For another thing it would be a gross violation of structural rules since as we will see later this would make it impossible for the structure to contain any references to the block! What happens when the last active key in a block is deleted will be discussed later on.

An index *always* contains at least one active key block, even if no user created keys are present. This implies that there is always at least one non Stopper key in the index. This particular key, called the ***Zero Key,*** has the largest positive number in its LINK field and a Key Value of all zeroes. The all zero key value is reserved to protect this mandatory Zero Key; *if you insert or delete a key with this value, you can destroy the integrity of the index structure.* The particular LINK value chosen is not actually reserved, but its use is discouraged since some systems may use this value to mark keys for future deletion. When the index is initialized, its one and only bottom level key block will contain just a Zero Key record, followed by Stoppers.

It is entirely up to you whether a particular index will enforce conventions about unique key values. As is usually the case, indexes that contain potential replications of identical key values require more work to maintain and use than do

ones with unique keys. This system's behavior with respect to identical keys is deterministic, and as will be discussed later there are tools to help with managing them. See also Usage Recommendations below.

There are just two sorts of Active Key blocks that can exist in an index: ***Bottom Level Index Blocks***, whose `LINK` fields contain user supplied values such as data file references that comply with the rules stated previously; and ***High Level Index Blocks***, whose keys refer to other Index Blocks. High Level blocks are created and managed only by the package, and are its property. The two kinds of index blocks are distinguishable by their `LINK` values. All `LINK` fields for active key records in Bottom Level blocks are *positive and nonzero*, while all `LINK` fields for those in High Level blocks are *negative*. (Stoppers look the same in both kinds of blocks.) To illustrate how this works, let's go through the procedures for searching, insertion, and deletion in order and in that way all of the special structural cases will be revealed as they occur.

4.7.3.2.4 Search Operations

To search for a given key value, begin by placing that key value at offset 4 (past `LINK`) in `WORKING`. ***Never,*** *under any circumstances, search for a key whose value is all ones!* If padding is required, ensure that you do so in a deterministic way every time a key is placed here for any purpose since the padding is logically part of the key and is compared. Store the key in `WORKING` as you usually would for its data type so that byteswapping or other manipulations conventional in your system are done. The key record in `WORKING` is protected by the package except where specifically noted below.

The simplest, fastest, and least restrictive search is done by **`N-ARY (- n t)`**. It searches for an exact match with the key given. If no match is found, `N-ARY` returns a garbage number and false. If exactly one matching key is found, `N-ARY` returns that key's `LINK` value and true. If the key was not unique in the index, the `LINK` returned by `N-ARY` is that of the *last* matching key in the index. `N-ARY` requires at least shared use of the index. Its intended use is simply to find a data reference in an index that presumably contains unique keys. This search leaves behind no information that may be used for further operations on the index, even to the point that `R#` is garbaged.

The procedure whereby `N-ARY` searches the index is as follows. The `ANCHOR` block is searched using the internal word called `-BOTTOM`, which finds the last key in the block that is less than or equal to the desired. This key's `LINK` field is inspected; if positive, we are done, and that `LINK` value is returned along with the flag indicating whether the key values matched at this last comparison. If on the other hand `LINK` is negative, this must be a High Level block. The sign bit is stripped from `LINK` and the remainder of its value is used as a record number for accessing the next lower level index block, whereupon `N-ARY` recurses and searches the new block as above. (As of level 4h, the internal search primitive also returns an indicator of index corruption that is tested within the package to throw exceptions in such cases.)

As will be seen later, the insertion rules guarantee a structure such that we will *never* look at an inappropriate block. Thus, any key block we look at during a search will *always* begin with a key less than or equal to the desired. Furthermore, the structure guarantees that any key block we look at during a search will *always* contain either an active key greater than the desired, or will contain the last key less than or equal to the desired existing at the current level.

In fact, each key value present in a High Level block is a copy of the *first* key in the referenced lower level index block. The structure is a tree, where each node can refer to as many branches as there are key records in a block. If the `ANCHOR` block is a High Level block, it is simply a list of the starting keys in each block at the next lower level. The top level always consists of a single block. Each level, whether a single block or a sequence of blocks, contains a sequenced list of keys; if multiple blocks are involved, the blocks may each contain anywhere from one through `R/B` keys. The successor for the last active key in a block at any level is always the first key in the next block at the same level; the "next block" (if any) is defined by the sequence provided through the keys at the next higher level (if any).

The convention that each high level block lists the starting keys of lower level blocks provides the guarantee of appropriateness mentioned earlier. Suppose we have a two level index whose top level index block has two keys. This means that the bottom level has two blocks. The first top level key is the Zero Key, and points to the first bottom level block whose first key is the Zero Key. All remaining active keys in the first bottom level block will be less than or equal

to the first key in the second bottom level block, and that key value will be the second one appearing in the top level block.

If the first bottom level block had key values 1, 2, 7, and 8, and if the second bottom level block had key values 20 and 30, then the two keys in the top level block would be zero and 20. If we were looking for any key from zero through 19, we'd stop on the first key at the top level and go look at the first bottom level block, where those values *must* be if they exist. If we were looking for 20, or anything greater, we'd stop on the second key at the top level and go look at the second bottom level block, where any of *those* values would lie.

Consider non unique keys. Suppose we had the same index as above, except that the bottom level had 1, 2, 3, and ten occurrences of 15 in the first block; and ten more 15's in the second block followed by 100. The top level keys would then be zero and 15. In searching for 15, we would not look at the first bottom level block at all. Instead we'd stop on the second key at the top level, which would send us to the second bottom level block, where we would in turn stop on the *last* occurrence of 15 in that block (and incidentally the last 15 in the index.)

4.7.3.2.5 Path Searching

If you plan to maintain the index by insertion or deletion, or to move about in the index sequentially, a different search word called **`n-ary (- n t)`** must be used. While its inputs and outputs are the same as are those for `N-ARY`, this word leaves "tracks" so that those operations are possible. `R#` is left pointing at the bottom level key record that was found, and a stack is maintained keeping track of each higher level key record that was used to find it. `R#` is the key found, is the key that may subsequently deleted, and is the key *following whose logical position* in *the bottom level sequence* a new key might be inserted. The stack gives us essentially the same information for each higher level.

Two types of facility protection come into play when you use this word. The first is that of the index itself. When you perform a path search, you obtain information that will only remain valid so long as the index is not updated, and this implies at least shared use of the index until you no longer need the information. If you did the path search so that you could update the index, it follows that you must acquire exclusive use of the index before performing the search. The facility protection may be released in either case after the planned operation is complete.

The second facility of interest is the path information itself. As stated before, this exists in the form of `R#` and a stack. The basic package is provided with a single stack, so it is a system wide resource and all operations involving path searches require exclusive use of this stack. Since the basic package assumes only global `ORDERED` control for all data bases, the stack is acquired through `ORDERED` as well. Depending on your application, multiple stacks can be allocated on a file by file, per task, or other basis. Application needs in this area vary widely; see the Adaptations section later on for more information. Regardless of how stack management is implemented in your system, the stack used for a particular path search must be exclusively owned until the purpose for which the search was done has been satisfied.

4.7.3.2.6 Sequential Searching

Once you have established a path to a particular bottom level key using `n-ary`, you may reposition yourself forward or backward in the bottom level key sequence in steps of one key at a time. **`ADV (- t)`** steps forward to the next active key in the sequence, and `-ADV (- t)` steps backward. Each of these maintains `R#` to point to the new bottom level key, having used and updated the stack as necessary to get there. *Neither of these inspects or changes anything at* `WORKING`. Each returns true if there *was* a user key to step to, in which case you may reference the bottom level key record `R#` points to directly for any desired information.

These words have "stopping power." If you are positioned to the last active key at the bottom level, `ADV` will return false and leave your position unchanged. You will still be pointing at an active record, and will still be properly set up to insert a key that correctly should follow it, or to delete the last key in the index. `ADV` will repeatedly do the same thing as many times as it is called under these circumstances.

If you are positioned at the first active user supplied key at the bottom level, `-ADV` will go ahead and position you to the Zero Key but will return false. If you use `-ADV` again while positioned at the Zero Key, it will again return false but

will leave you positioned at the Zero Key. This is the proper insertion point for a new smallest key in the index, but is a position from which you ***must never*** perform a deletion!

Sometimes you will need to find the specific key that goes with a particular LINK value, such as when deleting an index entry for a given data record from an index that doesn't enforce unique keys. **POINT (- t)** does this. To use POINT you must set up WORKING as for any normal search, and must also place the desired LINK value at WORKING. POINT does a normal path search for the desired key. If the key value found does not match, POINT returns false. If both key and LINK values match, it returns true. Otherwise POINT steps backward one key and tries again. Thus, when you get a true return from POINT you will be positioned at the exact key record you had in mind.

When POINT returns false, you will be positioned *in front of* the place at which the values in question would go. For example, using POINT with the impossible LINK value of will return false but will leave you positioned in front of any occurrences of the desired value. ADV may then be used to look at the first such key, if any; **EXACT (- t)** will be true if the key at the current position matches the desired key value in WORKING.

The word **NXT (- t)** facilitates simple but inefficient courteous sequential access to a data file LINKed to an index. To begin the process, zero all of WORKING and R#. Then, to process each data record, invoke NXT. If NXT returns true, there exists another key in the index. R# is left positioned so that you may fetch its LINK and then release the index and stack. You may then READ the data record that LINK points at, and process it as needed. When you are done processing the record, move its key value into WORKING and leave R# pointing *at the data record*. NXT will move R# to LINK, POINT at the key for the record you just processed, and ADV to the next one. Since each step does a new search, you need not prevent index insertions or deletions while processing. A word of caution, though; if the key for the data record you are processing is deleted while you're processing it, NXT will rescan all matching keys.

4.7.3.2.7 Insertion

Insertion of keys is normally a two-step process. You begin by searching for the new key using n-ary with exclusive use of the index. Without releasing the index or stack, you then store the desired LINK value in WORKING and invoke **+nary**. This latter word inserts the new key *after* the one that was found and then releases the index. Both steps are necessary; the first finds the insertion point and its path, and the second maintains the structure as needed. Note that since the normal search finds the last occurrence of identical keys, new identical keys follow old ones. By properly using both of these words you can fashion any desired sort of search-insert protocol for such purposes as ensuring unique keys, maintaining symbol tables, and so on.

If you don't care whether or not the key is already present, you can streamline things by simply building the entire new key including its LINK at WORKING and then invoke **INSERT**. This word performs both search and insert steps. *Regardless of which of these procedures you use, WORKING should be assumed garbage after the insertion is complete.*

The procedure for insertion is fairly simple in concept but involves several boundary conditions. In the normal case, the bottom level key block containing the insertion point isn't full (contains at least one Stopper). The operation is then trivially completed by moving the keys past the insertion point "up" one record in the block, and writing the new key record from WORKING into the block. The only special case here would occur if we'd changed the first key in the block, in which case we'd need to alter any references to that key in higher level blocks. However, it is *impossible* to even talk about an insertion point *preceding* the first key in a block; in any such case, we are "thinking" about a different block (the one preceding it in the sequence). Hence, *insertion never replaces the first key in any block* and hence never requires maintenance of that block's references at any higher level.

Thus, in the normal case we leave behind only one updated block. If, however, the key block containing the current insertion point is full before the insertion, we will have to make room for it. The first step is to split the key block at the current level. This is done by allocating a new block in the index file and moving the second half of the original block into it. Both blocks are topped off with Stoppers, so that the first half of the keys from the original block still lie in that block, and the second half of its keys are in the first half of the new block. We then determine which of these two

blocks should receive the key from WORKING and insert it in the appropriate one as above. To minimize buffer pool activity, the splitting operation uses a 1k memory buffer called **nBUF** with brief exclusive use through the facility **nBV** . This is a choke point, but is used so briefly that we considered it justified.

At this point we have updated two blocks and done whatever the system requires to have allocated the new block. However, the new block is not linked into any higher level structure, and so we move the first key from the new block into WORKING , move the new block's starting record number with sign set to LINK in WORKING as the key we are *now* trying to insert, and recurse up to the next higher level using the stack. By definition, since what we recorded in the stack were record numbers of intermediate higher level key records, and since *all* key blocks conform to the same sequencing rules, the insert point for the new key (first key from the new block) is properly exactly the key record at the next higher level. Therefore, we simply repeat all of the above procedure at the next higher level using the high level key constructed in WORKING . The process continues until we have made an insertion that did not require splitting a key block.

If it is rare to split a bottom level key block during insertion, it is rare squared to split the next higher level and so on. Therefore, if the normal case requires updating one disk block, the "abnormal" case typically requires updating only three and performing a block allocation. However, in the rarest of cases, we will find it necessary to recurse clear out to the ANCHOR block and discover that it is full. The final insertion boundary condition occurs when we need to split this block, since there *is* no higher level key block.

In this rarest of all cases, we must add a higher level to the index. This is done after splitting the former ANCHOR block by allocating a new block to be the new ANCHOR , initializing it with high level keys referencing the two halves of the old ANCHOR , and filling the rest of it with Stoppers. ANCHOR is changed to point to the new block, and ANCHOR 4+ is incremented to count the new level that has been added.

The notion of adding levels to the trunk rather than to the twigs of the tree is the main piece of elegance in this entire scheme and its central point of departure from what is apparently common practice in B-tree implementations. When levels are added to twigs as they overflow, two bad things happen. First, growth is vertical and vertical growth does not make much additional space for the price of the level it mandated. Vertical growth at a twig adds lots of room for keys in its immediate vicinity, but does nothing for random keys elsewhere. With the same data, twig growth costs space and levels. Second, twig growth leads to tree imbalance which in turn leads to longer search times, more complex structures, deeper stacks, and much more complex utilities.

In contrast, when levels are added to the trunk they enlarge the structure *horizontally,* creating more usable space throughout it by effectively stretching the entire structure and revealing new empty space in both axes. The tree intrinsically remains balanced, resulting in a flatter topology with fewer levels than other schemes regardless of the order in which keys are created. Having said that, we must note that while levels are minimized, *space* is consumed most rapidly when keys are generated over time in ascending or descending sequence. This follows from the splitting method; whichever "direction" new keys are travelling in, the bottom level blocks (and higher level ones too) that are left behind will not be revisited. Thus, sequential key creation tends to leave all active key blocks half full. This is the worst case condition for index creation; with any randomness at all in key creation sequence, the resulting index will have much better than 50% space utilization.

4.7.3.2.8 Deletion

You may delete any key to which you've positioned the index using path searching, or any of the other searching procedures above that maintain the stack, by executing **DELETE** . This word does not care what's in WORKING , but should be assumed to leave garbage there. It *does,* of course, ***require*** that the bottom level key to be deleted ***must*** be properly described by the stack and by R# . When completed, the index and stack are released; as with INSERT , any further operations on the index will require a new search. *It is **absolutely forbidden** to attempt deletion of the Zero Key. If you do so the index will immediately become unusable.*

The procedure for deletion is quite a bit simpler than for insertion, although it too is potentially recursive. The first step is to remove the R# key from the current level's key block, moving the rest of the block "down" one record and inserting a Stopper at the end of the block. In the normal case, the key being deleted isn't the first key in the block, so

we don't need to do anything further. Thus, in the normal case deletion, just like insertion, updates only one disk block.

If, however, we *did* delete the first key in the current block, we'll have to do *something* to the next higher level index block. (There's *guaranteed* to be one, since the first key in the Top Level block is always the Zero Key, which is illegal to delete.) There are two cases to consider.

In the more normal of the two cases, the resulting block still contains at least one active key. Thus, the only structural problem we've created by deleting its first key is to invalidate the key pointing to this block in the next higher level. If there *is* a higher level according to the stack, we simply rewrite the key value for this block in that level to reflect this block's new starting key, updating a second disk block, and we're through.

If, however, we have deleted the only key in a block, we have created an anomaly since the rules forbid empty key blocks. Therefore we make the block we're working on *Free* and recurse to the next higher level (guaranteed to exist) where we repeat this whole procedure to delete the high level key which used to refer to the block that we just freed. Thus, `DELETE` will free disk blocks almost anywhere in the tree as they become empty. However, since the Zero Key can never be deleted, *the first block at each level can never be deleted either.* This implies that `DELETE` never reduces the number of levels in the index. If you build an index five levels deep and then delete all of its keys in any order, you will end up with five active blocks, each of which contains only the Zero Key.

This deletion method is very harmonious with the horizontal growth property of the insertion method. As sections of the key address space fall into disuse their blocks are released and boundaries are realigned, leaving room for expansion at all levels into whatever parts of the address space are becoming more active. With random insertion and deletion activity, these indexes require little maintenance; they tend to remain well balanced at an appropriate ambient number of levels, and most of their blocks tend to show utilization well above 50%. For example, indexes of people's names are typically well behaved in this way. With nonrandom activity, space can become fragmented and more maintenance attention may be required; see Usage Recommendations.

4.7.3.2.9 Exceptions

As of level 4h, tests are made at several places in the package to detect corrupt index files. Upon their detection, exceptions are thrown. with values **`ex_MCORRUPT`** for apparently corrupt pointers, and **`ex_MDEXCEED`** for exceeding the configured maximum index depth. (Generally this exception will imply a key loop unless the configured limit is set much smaller than the default ten levels.) When these exceptions are thrown, facilities such as `ORDERED` may still be owned and will need to be released.

4.7.3.2.10 Utilities

Three main utilities exist for managing and maintaining multilevel indexes. None of them are reentrant. They are as follow:

4.7.3.2.10.1 Index Analysis

This utility gathers statistics regarding usage of an index and displays them. **`ENUMERATE`** quickly traverses the index, filling local arrays with descriptive data. **`SHORT`** produces a brief summary report from these arrays. A line is displayed for each active level in the index, showing total blocks and total keys for each. It also computes and displays mean keys per block and mean percent utilization of each block at each level. This is a good way to find out how well an index is behaving, particularly in a new application. **`HISTOGRAM`** displays detailed data about space utilization in key blocks. For each active level, a histogram is produced. The vertical axis is in percent utilization of space in blocks, and the horizontal axis is proportional to population of blocks at each usage level. Each histogram is preceded by a scale factor which is the number of blocks representing full horizontal scale.

4.7.3.2.10.2 Index Compression

This utility provides a single word, **`SQUISH`** . It compresses a multilevel index to minimum size and prunes unnecessary high levels in a single pass over each level. To minimize execution time, this is done by a procedure that precludes any other use of the index while it is happening. The execution times are quite short, well under a minute for even relatively large indexes.

While frequent SQUISH ing is a sure way to avoid problems with index fragmentation, we have found that many indexes are well enough behaved that this is not necessary or even desirable. Stabilized indexes tend to have plenty of insertion space everywhere, so that most maintenance actions take minimal disk operations. On the other hand, right after SQUISH ing most insertions will need to split blocks most of the time. Find out what's going on with your indexes before SQUISH ing them frequently or indiscriminately.

4.7.3.2.10.3 2-phase Rebuild

To rebuild an index the "fast" way, begin by INITIALIZE ing the index file in the normal way for a flat file. Then write a Zero Key at record R/B , followed by any additional keys as a straight sequential flat file. Do *not* write a Stopper at the end. Leave in AVAILABLE the record number of the last key in the file. An "empty" index will contain just the Zero Key as above and AVAILABLE will be R/B .

Unless the keys you wrote were intrinsically in proper key value sequence, sort the keys in the flat file using any sorting algorithm that can be *absolutely* trusted.

Finally, invoke **2-PHASE** on the file. Space will be allocated for any required high levels; keys will be abstracted into them; Stoppers will be written at the end of each level; and Block Zero will be left set up properly. The operation is quite brisk, and if you have a good sort this procedure will be faster than INSERT ing everything.

4.7.3.3 Usage Recommendations

It takes time to traverse a list of identical keys, and this must be done to find the particular key associated with a given LINK value so that it can for example be deleted. For this reason, as with other indexing methods *it is not a good idea to file default field values, such as nil, in these indexes* when that would lead to large populations of the nil value. If you *must* file such values, consider concatenating something deterministic and unique, such as data file record number, with the key field. Other such cases that can lead to trouble are things like department numbers in large groups with few departments. Such an index is cumbersome and you should consider a better indexing key such as for example department number concatenated with employee number.

In larger systems with many users, it is most discourteous to maintain even shared use of an index any longer than is absolutely necessary. This makes it difficult to do a good job of sequential processing in index order. The safest but slowest way is to use NXT as outlined above. Another method is to abstract a list of items with ADV , then release the index and process the data records in the list. When done, use NXT from the point of view of the last data record in the list and abstract another group. This is unfortunately complex. If you add update sequence numbering to your files, and have a search stack of your own, you can then implement an abbreviated NXT that only does a full search if the file's been updated while you were away. If it has not, you can safely bypass that step and use ADV . This is the simplest method we have thought of so far with this (or indeed any) indexing package.

There are some pathological nonrandom insertion/deletion sequences that can play havoc with index fragmentation. One such case might occur when the keys being indexed are assigned sequentially over time as new data records are created. If each of these keys is deleted a certain amount of time after it was created, you can see how DELETE will dismantle obsolete key ranges as they disappear, making space "ahead" with good efficiency. If, however, some small percentage of these keys last *much* longer or are *never* deleted, the "older" side of the tree will wither but will retain much deadwood in the form of nearly empty blocks that will not be deleted any time soon. If there's sufficient extra room in the file, this low utilization can go pretty far without doing much but wasting space and adding a level or perhaps two. If you don't have lots of space, indexes that act this way should be SQUISH ed on a regular schedule.

4.7.3.4 Adaptation

Since this package is often used as a retrofit into existing applications, we often need to adapt it to the conventions of those applications' data bases. Generally this requires coding changes to the package but few changes in usage beyond facility management.

Facility management capabilities vary widely among systems, and cannot be completely concealed from the application. Obviously the best case will occur in new systems using shared, queued facilities. Whenever feasible,

appropriate facility management operations occur within the operative words of the package, minimizing application sensitivity to these things. However there will always be exceptions.

We have not yet invented an optimum general solution to the problem of search stacks. They are definitely resources, but their usage depending on the application may map onto files, tasks, some combination of the two, or none of the above. Without some knowledge of the application it's not clear to us what structures should have stacks attached and how many there should be. Basically any task having shared or exclusive use of a file might want to have an active search path for that file. Since all tasks may share any file and any task may share all files, the only safe answer that can be given in ignorance is the product of tasks times files, which is clearly an absurdly large number of stacks to allocate. Thought continues on the subject, but for the present positioning and replication of these stacks is done on a case by case basis.

5. Extensions and Utilities

5.1 Resident Programmer Interface

5.1.1 Editor

The resident editor is, basically, the venerable character editor which has been standard on polyFORTH systems as well as all ATHENA systems for many years. It's reentrant, and is small enough to be PROM resident on any machine that can access mass storage and compile source. There have been some recent enhancements:

5.1.1.1 Moving/copying lines from another block

The **M** function is operationally awkward because of its stack arguments. An alternative uses the normal editor mechanisms for selecting the source to be copied from. To avoid problems, it has been given the name **Y**. Unlike `M`, `Y` takes its source from the "other" block at the current positiion and advances the current position of the "other" block to its next line. Otherwise it is functionally identical to `M`. The only difference is that instead of selecting the text to be copied by specifying its block and line numbers, the selection is done by normal editor operations that one would do anyway when verifying that the stuff to copy is what one actually wants. The name may change some day in the future but not soon enough to make you crazy. Call it Yank instead of Move.

Procedurally, to grab source from line 5 of block 177:

1. Look at block 177 (`177 LIST`).
2. Select the first line to copy (`5 T`)
3. List the block into which you want to insert stuff
4. Put cursor on line under which you want to insert it (`n T`)
5. Say `Y` for each line to grab.

The only difference in procedure between this and `M` is that the steps numbered 1 and 2 above replace the operations of putting block and line numbers on the stack. Since `O` does not change the cursor positions you can toggle between the sending and receiving blocks at will with `O` during the copying operation.

This feature has proven useful and is now present in all saneFORTH systems. When the "other" block is not conveniently set up for use of Y, the following word is provided:

`SY (b n)` explicitly makes ***b*** the "other" block and ***n*** be the current line number [0..15] in block ***b***.

5.1.2 Concordance Librarian

Block 9 shows commented code for loading the concordance interrogation mechanism. To use this mechanism, the disk directory must contain valid entries for data bases named **`conc-WORDS`** **`conc-CRUD`** and `conc-XREF`, and these data bases ***must*** have been previously constructed by the Concordance utility as described later on in the Utilities section.

The concordance utility scans the source in specified ranges of blocks, building a comprehensive source concordance of all words found in those blocks. The utility ignores a short list of ubiquitous words such as colon and semicolon, but otherwise basically parses space delimited strings and sorts them. When the librarian code is loaded, the following resident functions are added, and the `EDITOR` block listing is enhanced to show the results of the current search. Current search results are not instantiated per user; there is no restriction against concurrent use of the librarian by multiple terminals, but the result set in effect for *all* terminals will always be the most recent produced by *any* terminal.

`LIB` Displays Librarian help screen.

FIND (_) Composes a result set of all references to the given word or string.

NEAR (_) Composes result set of all words or strings beginning with the given string.

HIT (_) Composes result set of all words or strings that would collide with the given string in a 3 character plus length dictionary. This is still relevant for 16-bit systems but we use full length names when resources permit.

NN and **BB** (right and left arrows on PC keyboard) move forward and backward within the blocks containing the result set. The current block number must be within `[0..38400[` relative to `OFFSET`.

Up arrow on PC keyboard is equivalent to `0 LIST`.

Down arrow on PC keyboard is equivalent to `L`.

This tool is invaluable when researching changes or corrections for large applications. It is unfortunately necessary to run the concordance utility, which takes a while, to absorb source changes into the concordance data base; however, for definitive research the advantages of holographically accessing commented and conditionally compiled code, overlays, nonresident utilities, comments, and arbitrary strings such as block numbers in `LOAD` statements, are of overriding importance.

5.1.3 Display Tools

XD (a n) dumps octet oriented data in hex, with dump relative offsets instead of absolute addresses. May be changed without notice but the basic functionality should remain the same.

.IO (a n) dumps a range of I/O address space, using 8 bit input operations.

5.2 Resident Functions

5.2.1 Conveniences

These environment-specific functions have proven useful:

BYE Orderly exit of saneFORTH as a Windows application.

WHO Added for arrayForth because it is the human interface for both sF/x86 and pF/144. Identifies which system you are talking to at the moment.

5.2.2 Hex Numbers ("xNUMBER")

After years of losing white space in source code to transitions between HEX and DECIMAL radixes, not to mention obscure bugs resulting from, for example, copying source phrases from an area in one radix into an area in the other, we finally introduced a convention that's now standard across all saneFORTH and polyFORTH systems we have created, specifically including the systems comprising GreenArrays' arrayForth 3.

When parsing a potential number, regardless of the current radix (BASE), its first character is tested and, if found to be lower case X, the rest of the number is parsed and converted as being represented in hexadecimal. This test and behavior is always performed as the first act of the routines in the 'NUMBER vector so it takes precedence over recognizing such things as floating point numbers.

This capability has proven to be very clean and useful. It does require a usage rule, namely that you should not define any word beginning with lowercase X that looks like a valid hex number. This is not because the system would misinterpret your word as a number; as usual, our systems check for valid words in the dictionary before considering that they might be numbers. However, if you define such a word and you or anyone else happens to write a hex number that is spelled the same, your word will be found and instead of a hex value you will have a bug. To see if you have any potential problems in this area, use the concordance and say **NEAR x** which will quickly reveal any potential problems.

5.2.3 Extended Data Types

Scalar arrays

`table (n)` allots and zeroes the given number of bytes. The following array types are all prezeroed in this way.

`TABLE (n _)` allots a named area of ***n*** bytes with table .

`CARRAY (n _) (i - a)` creates a named array of ***n*** bytes which is indexed by a zero relative value.

`HARRAY (n _) (i - a)` creates a similar array but of halfcells.

`ARRAY (n _) (i - a)` creates a similar array but of cells.

`2ARRAY (n _) (i - a)` creates a similar array but of double cells.

`.ARRAY (n _)` creates a named area of at least n bits with table .

`.@ (i a - t)` fetches truth value of bit i in .ARRAY a .

`.! (t n a)` stores truth value into bit i in .ARRAY a .

String arrays

A *String Array* is a data type consisting of a 1-d indexed array of strings. It's mechanized using a vector of pointers to counted strings. The vector is indexed 0-relatively; the strings may be of differing lengths.

`SARRAY (n _) (i - a n)` defines such an array, initialized with zero pointers; returns double zero for a nonexistent element. May be initialized by hand (for example, declare with zero length then comma addresses of the strings) or by using `S{` .

`S{ (n _})` defines element ***n*** of the most recent definition, which ***must*** be an `SARRAY` . Usage:

```
2 SARRAY Y/N   0 S{ No }  1 S{ Yes}
```

`[TYPE] (a n w)` displays the given string left justified in a field of width ***w*** , padding with spaces as needed. Provided here to facilitate string tables with variable width elements.

5.2.4 Miscellaneous Primitives

Loop Control

`I+ (n - n)` efficiently adds the induction variable `I` to the given value. Equivalent to `I +` .

`TOGO (-n)` returns the number of iterations remaining in the current `LOOP` including the current iteration. Equivalent to the phrase `I' I -` .

Stack Management

`DROPS (w*n n)` removes the given number of items from the stack.

Integer and Boolean Operations

`@! (n a - n)` exchanges the given number with a cell in memory.

`>4< (w - w)` interchanges the four octets of a value, reversing the "endian-ness" of the value.

`-! (n a)` subtracts the value from a cell in memory. Complementary with `+!` .

`H+! (h a)` adds a number to a halfcell in memory.

`&! (n a)` AND's the argument onto a cell in memory.

`OR! (n a)` OR's the argument onto a cell in memory.

`-&! (n a)` Clears the 1-bits of the argument in a memory cell. (Bit clear; not n AND m)

Structure Definition

`OFS (n w _ - n)` defines a named offset in a structure. The name is made a `CONSTANT` whose value is the given *n* after which *n* is advanced by the distance *w*.

Coding Tools

>> permits high level FORTH to fall into following code. The return to the caller of the high level is already made, so the code normally ends with `NEXT`. `>code` is execution. A trivial example that returns the second cell in a block:

```
: 2NDCELL ( n - n)   BLOCK >>   W POP   4 W) PUSH   NEXT
```

5.3 Elective Functions

TCP/IP Package, documented separately.

Cryptography, not documented publicly due to Arithmetic Control laws of various governments.

5.4 Utilities

5.4.1 Concordance Generator

The utility in block 1551 scans source and builds concordance files for later searching by the Librarian. This utility consumes significant memory and takes a while to run, so it is generally practical only from the `OPERATOR` terminal and only when a good deal of memory is available. The required files must exist and be sufficiently large for the source to be indexed. The range or ranges scanned are defined by the word **`SCAN`** which normally indexes only blocks 0 to 2340. You may wish to change `SCAN` by adding `TEW` or `LODE` phrases to cover your source. However, this package is designed for use with block numbers expressible in 16 bits, and the librarian mechanism as delivered limits its operations to a span of 38400 blocks.

After checking its configuration and successfully loading the utility, the phrase **`GO >LIB`** performs the complete procedure. Read shadows to understand the scanning rules used in this package; in particular, pay attention to the rules for right parenthesis. They have been designed to ensure that the important identifiers will be indexed in normal usage of code commenting, but these rules preclude indexing of words with spellings such as `X)Y`.

Once the utility has run, the files may immediately be consulted by the librarian; if the latter is already loaded, no reboot is necessary.

5.4.2 Not Documented Yet, see shadows

Single Stepper

Memory Identification

Stack Dumping

Interactive Crash Dumping

Delta Lists

3-Way Matching

Terminal Emulator

Benchmarks

Pasting source between Telnet and similar windows

5.5 Debugging Tools

5.5.1 386 Debug Registers

These are not accessible to application code in Win32, which is most unfortunate because they are extremely useful in debugging otherwise intractable problems such as wild stores.

5.5.2 Panic Dump

These are also applicable only to native systems.

6. Operations on Raw Media

Subject to Windows' cooperation, this system supports reading and writing removable media such as floppies and USB flash memories, not merely as Windows file system devices but as native, raw devices such as those used as boot media for native saneFORTH or colorForth systems. Assuming that new media are preformatted for Windows file system use, the data structures created by such formatting must be clobbered, something which requires permission from Windows. Procedures for acquiring such permission have changed over the years; the procedures herein have been tested on Windows 7 Professional, and on Windows 10 Professional up to date as of 1 Feb 2021 but without the latest "Feature Update" as of that date (Feature Updates generally break many things and so we delay applying them due to their disruption of useful work.) Once this clobbering has been done successfully, it need not be repeated.

In the following, we assume you are using the sF released with GreenArrays' arrayForth-3, available from customer support central on http://www.greenarraychips.com and have installed it as recommended.

After using this utility, the mapping of BLOCK address space to files will have been altered, so you will probably need to reload saneFORTH before resuming normal activities.

6.1 Accessing Raw Media in Windows

The term "Disk" has varying meanings in Windows depending on the media. For a 1400-block floppy, it means all 1400 blocks starting at absolute sector zero on the physical floppy. For a flash or other large disk medium, there will generally (always?) exist a partition table in Windows' world, and "Disk" will ordinarily mean the contents of a partition. In either case, Windows will normally assign a drive letter to a "Disk" and will in normally allow us to read and write anywhere in a "Disk", including its structures such as "Master Boot Records" (MBRs) and file directories, treating the "Disk" as raw storage.

For a floppy, we may thus make bootable media because it is the MBR that defines what is being booted. For flashes, we must first nuke the partition table (which begins with its own MBR, the only one Windows sees) so that we may access the physical medium at absolute zero as though it were a floppy.

The path referring to a floppy will be of the form `//./A:` where A is the drive letter, visible in File Manager.

We refer to the flash device by its physical disk number. Learn this by inspecting the graphical view in Disk Management on Windows 7..10. The left section says Disk <n> and the path `//./PHYSICALDRIVEn` denotes this entire device. If the graphical section shows a single "Healthy Primary Partition" of xxx MB RAW then the device has already been nuked so the nuking step can probably be omitted and the floppy procedure below can be used with the PHYSICALDRIVE path shown above.

6.2 Making a Native Floppy

The basic Windows file explorer, at the node variously named "Computer", "My Computer" or "This PC" depending on which version of Windows you are using, will show a list of devices and drives including their drive letters. Floppies will appear here and by convention the first floppy, whether native or USB, will be assigned drive letter A. The following assumes you found that assignment.

6.2.1 Using the NATIVEBOOT Utility

This utility is an extension of DISKING that allows you to map arbitrary files or drives into 8 UNIT or 9 UNIT between which data may be moved. You must have started sF with "run as administrator. To write a boot floppy:

1. Place a copy of the file containing a 1440 block floppy image into the root directory for sF-Win32 (this will be C:\GreenArrays\EVB002\sF if you used the recommended arrayForth-3 installation procedure.)
2. Mount a write enabled floppy into the drive you are planning to use.
3. Check the file explorer to get the drive letter assigned to this floppy. Look at "properties" which should show you that a FAT file system exists if it's a new floppy.
4. Run sF using the **arrayFORTH-3 PANIC** shortcut, and say **`HI`** .
5. Load the NATIVEBOOT utility: **`93 LOAD`**
    a. If you have not clobbered the floppy's MBR before, you must use the following procedure on Windows Vista and later:
        i. Say **`95 LOAD`** .
        ii. Say **`TRY \\.\A:`** and answer Y to question about clobbering other software. Expect it to say locked 0 unmounted unlocked .
        iii. If and only if you have done the above successfully, will Windows allow you to write on the floppy in the following steps.
    b. Say **`SRC: filename-of-floppy-image`** This will map **8 UNIT** onto that file.
    c. Say **`OLD: \\.\a:`** This will map **9 UNIT** onto the floppy.
    d. Say **`SRC DEST 1440 BLOCKS SRC DEST 1440 MATCHES`** to write & verify floppy. Note that if all you need is the boot and no source, you may copy and match only 60 blocks.

6.3 USB Flashes/Disks

This is more complicated because we must annihilate the true master boot record and partition table of the medium, which requires using a more arcane path for its access.

If the flash has already been nuked, you may get a message saying you must format before using. Cancel/ignore this message.

See code in Xa/sF-NT/Matrix shortcut Matrix BC sF01b-2 which has scripted code in 70..74 for making USB flash install media. This needs to be reconciled with my base 93..97 before committing to documentation.

6.4 Making a Bootable Native saneFORTH System

When we are writing directly to absolute zero on the physical medium, we may make an MBR that Windows will obey to load and run our code. If that code (the saneFORTH nucleus) knows how to operate on the physical boot medium, as it does for native floppy drives or ATA/SCSI hard disks, once booted the system may do things like 9 LOAD directly

from the medium, up to the point of connecting to the network and importing any desired code or data to commission a new box, using ATHENA's network disk protocol from other saneFORTH boxes or via FTP.

If the boot medium is something the nucleus does not know how to operate upon, such as USB disks or flashes, the nucleus must be changed to add RAM disk, and enough source code must be included on the boot medium to finish the desired commissioning operations. In this case, configure the nucleus is configured to talk to the native mass storage on the new box, but to use RAM disk so that the commissioning source code is read from the boot medium into RAM where the nucleus with RAM disk support may do things like 9 LOAD from the source image in RAM, copying it up to the real native medium and so on.

7. Building Executables

Unlike our earlier SVR4 Unix port, we have decided in this case to create executable files directly. This avoids many technical problems, and has the added benefit that we have *no dependencies whatsoever* on any third party licensed sofware in the process of generating our system or user applications.

We use the sF target compiler to generate Win32 PE (.exe) files. The following sections discuss the organization and implementation of mechanism to generate these files.

7.1 Target Compilation

Compile the nucleus conventionally: COMPILER LOAD NT LOAD leaves new nucleus binary in the target output area, in the index page at SHADOWS 60 -Configure the nucleus before compilation by editing block 182.

7.2 Nucleus Matching

470 LOAD PAGE HUNT shows binary differences between the "new" image (normally target output) and the "old" one (normally the one in the boot area, index page at -60). Options are available by rearranging the OLD and NEW definitions in block 470, including comparison against the nucleus actually loaded into memory and running.

7.3 Test & Install Utility

TESTING LOAD give a number of options. INSTALL replaces the nucleus binary in the boot area with the present contents of the target output area. To write a completely fresh boot file ... consisting of boot sector and nucleus binary ... say

```
TESTING LOAD  UNIV BOOT INSTALL
```

There are other options for cross-booting, setting up auto load, and so on. The phrase above causes any of these options that may have been selected before to be effectively erased.

7.4 PE Executable Format

The overall structure of a PE file, *as we make it,* is as follows (offsets and values in hex):

DOS hdr	+0	Header for DOS MZ executable which allows the stub to run in DOS. Includes pointer to start of PE Header.				Size 40
DOS stub	+40	Trivial DOS program which says "ATHENA Forth does not run in DOS mode."				Size 40
PE signature	+80	0	0	45 ("E")	50 ("P")	
COFF Header	+84	Also known as PE file header. Four main parts: File header (14), Optional standard (1C), NT specific (44), and Data Directories (80)				Size F4
Object Table	+178	28 hex per section defined. Null padded to file granule				
Image Pages	+x00	File granule boundary aligned; Most but not all listed in Object Table.				
		+3	+2	+1	+0	

7.4.1 DOS Header

All data in the DOS header are fixed. Offsets and values shown below are in hex. We still need to understand semantics of the early fields referring to page size; these are probably why our .EXE files are only accepted if their lengths are greater than or equal to 1025 bytes.

DOSHDR	+0	Bytes on last page of file 90 (why?)	Magic 5A ("Z")	Magic 4D ("N")
	+4	Number of Relocations 0	Pages in file 3 (why?)	
	+8	Min extra paragraphs needed 0	Header size, paragraphs 4 (64 bytes)	

+0C	Initial relative SS value 0	Max extra paragraphs needed FFFF
+10	Checksum 0	Initial SP value B8
+14	Initial relative CS value 0	Initial IP value 0
+18	Overlay Number 0	File Address of Reloc Table 40
+1C	Reserved	
+20	Reserved	
+24	OEM Information 0	OEM Identifier 0
+28	Rsserved	Rsserved
+2C	Rsserved	Rsserved
+30	Rsserved	Rsserved
+34	Rsserved	Rsserved
+38	Rsserved	Rsserved
+3C	e_lfanew: File Address of new exe header 80 (must be 8-byte aligned)	
	+3 +2	+1 +0

7.4.2 DOS Stub Program

The program we provide for DOS to execute simply puts out a diagnostic message and exits. It, along with the DOS header above, is generated by the following source:

```
546
 0 ( DOS hdr and stub)   HEX
 1 CREATE DOSHDR   ASSEMBLER 16BIT
 2    5A4D H,  90 H, 3 H,  0 H,  4 H,  0 H, -1 H,  0 H, B8 H,
 3       0 H, 0 ,  40 H,  0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 H,
 4       ( e_lfanew) 80 ,
 5    ( Entry)   IS PUSHS   DS POPS
 6    ( Print string)  0F # 2 MOV   9 #b 0 hi MOV   21 INT
 7    ( Term code 1)  4C01 # 0 MOV   21 INT
 8    22 STRING ATHENA Forth does not run in DOS mode."
 9       -1 ALLOT  240A0D0D ,
10    32BIT FORTH   DOSHDR 80 SIZED
11
12 HERE DOSHDR - CONSTANT /DOSHDR
```

7.4.3 COFF Header

The COFF header immediately follows the PE signature cell. Offsets and values shown in hex:

COFFHDR	+0	0	0	45 ("E")	50 ("P")	
File Header	+4	Number of Sections 7 in example		Machine ID 014C = i386 or later		
	+8	Date/time stamp, C library units: Seconds since 0000 UT 1 Jan 1970				
	+0C	Offset within COFF file of symbol table (zero for executables???)				
	+10	Number of entries in symbol table (1B7 in example; 0 in other examples)				
	+14	File Characteristics 8386 in example		Size of optional header 00E0		
Optional Hdr Standard Part	+18	Minor linker vers example = 32	Major linker vers example = 02	File State 010B for executable		
	+1C	Code Size text section or sum of such Example=0800				Sizes of granularized images in this file.
	+20	Initialized Data Size idata section or sum of such Example = 1600				
	+24	Uninitialized Data Size bss section or sum of such Example = 0400				Size granularized even though ∃ no image.
	+28	Entry Point RVA Relative to image base Example = 1543				In .text
	+2C	Base of Code Code section relative to image base Example = 1000				
	+30	Base of Data Data section relative to image base Example = 2000				
Optional Hdr NT specific	+34	Image Base (Preferred load address) Multiple of 64K Example 013E0000 vc++ 400000				
	+38	Section Alignment Must be ≥ File Alignment Example = 1000				Actually must be at least 4k (hdw page).
	+3C	File Alignment for "pages" 2^(9..16) Example = 200				A page is text for a section.
	+40	Minor O/S Version 0 for "NT version 1.0"		Major O/S version 1 for "NT version 1.0"		
	+44	Minor image version Our discretion		Major Image version Our discretion		
	+48	Minor Subsystem Version 0A for Win32 3.10		Major Subsystem Version 3 for Win32 3.10		
	+4C	Reserved				
	+50	Image Size Definition ambiguous. Example = 8000				Note 2
	+54	Header Size Includes DOS hdr, PE Hdr, Object Table Example = 0400				Note 3
	+58	File Checksum Tested for drivers, boot DLLs, server DLLs. Assume 0 default.				Note 4
	+5C	DLL flags (obsolete) Example = 0		Subsystem Code 0=Unknown; 1=Native NT; 2=Win gui; 3=Win cui; 5=OS2 cui; 7=Posix cui		finger is win cui.
	+60	Stack Reserve Size (Maximum) Example = 100000 (1Mb)				
	+64	Stack Commit Size (Preallocated) Example = 1000 (4k)				
	+68	Local Heap Reserve Size Example = 100000 (1Mb)				
	+6C	Local Heap Commit Size Example = 1000 (4k)				
	+70	Loader flags (obsolete) Example = 0				
	+74	Number of Data Directories Following 8 bytes allocated for each. Example = 10				
Data Dir 0	+78	RVA of Export Table (Example = 0)				Spec 6.3
	+7C	Length in bytes (Example = 0)				

Data Dir 1	+80	Import Table Example 5000, 2C2 = all of .idata	Spec 6.4
Data Dir 2	+88	Resource Table Example 6000, 0A9C = all of .rsrc	Spec 6.7
Data Dir 3	+90	Exception Table Example 0,0	MIPS/Alpha
Data Dir 4	+98	Security Table Example 0,0	???
Data Dir 5	+A0	Base Relocation Table (executables) Example 7000, 90 = front of .reloc	Spec 6.5
Data Dir 6	+A8	Debug Table Example 3000, 54 = front of .rdata	Spec 6.1
Data Dir 7	+B0	Copyright String Example 0,0	???
Data Dir 8	+B8	Global Pointer Register [MIPS] Example 0,0	
Data Dir 9	+C0	Thread Local Storage [TLS] Example 0,0	Spec 6.6
Data Dir 0A	+C8	Load Configuration Table Example 0,0	???
Data Dir 0B	+D0		
Data Dir 0C	+D8		
Data Dir 0D	+E0		
Data Dir 0E	+E8		
Data Dir 0F	+F0		
		+3 +2 +1 +0	

Bit assignments in the File Characteristics field are as follow:

File Char's										From File Header
	+0	little endian note 1	res for 16 bit mach	res for update	res for minim	no local syms	no line nos	xqtbl img	no relocs	
	+1	big endian note 1	res	DLL	sys-tem file	res	res for patch	no debug	32 bit mach	

Note 1: Although the spec clearly indicates that these two bits are mutually exclusive, the include file describes them both the same way, and all programs inspected had *both bits set* for 386 code. Implies bugs in linker, loader, and/or spec. In addition, executable produced by VC++ for exitwin.exe had *neither bit set.* Obviously there is confusion here.

Note 2: The 4.1 spec says this is size, in bytes, of image, including all headers; must be a multiple of Section Alignment. The "Top to Bottom" document says it is the amount of address space to reserve in the address space of the loaded executable image, specifically that it's the sum of the segments including alignment pads in memory.

It may be that they are both right. The example defines allocation for 7000 starting at relative 1000. It would seem that the initial 1000 counts. What is it for? Perhaps selected headers are laid down there. That would account for 8000 and would support a grain of truth in the spec's remarks.

Note 3: According to "Top to Bottom" this is the File Address of the start of the section bodies, in which case it also includes the DOS stub. In the example, this appears to be true.

Note 4: The spec says the algorithm is available in IMAGEHELP.DLL. "Top to Bottom" says the algorithm is proprietary and will no be published. FINGER.EXE has a value of 553A; vanilla VC++ programs show 0.

7.4.4 The Section (or Object) Table

This table starts at offset 178 within the file (or *immediately following* the optional COFF header) and consists of 40_{10} byte entries. The number of entries is given in the front of the COFF header. The order of sections is, according to the spec, arbitrary. Section numbers, according to the spec, are one relative table entry positions. The spec goes on to say that the virtual addresses of sections must be assigned in contiguous ascending order (with Section Alignment). This implies, but the spec does not say, that like sections (text, initialized data, bss) must be grouped together, so that the relevant fields in the COFF Optional Header, Standard Part, can be meaningful.

The format of a Section Table entry is as follows:

Section Table			
+0	Section Name 8 bytes ASCII, left justified, null filled		Note 1
+8	Virtual Size (loader pads with zeroes to this size)		No align rules
+0C	RVA of Section Start		Section Aligned
+10	Size of Raw Data		File align granules
+14	File Pointer to Raw Data		File aligned
+18	Pointer to COFF Relocs *Zero for executables*		
+1C	Pointer to COFF Line Numbers *Zero for executables*		
+20	Number of Line Numbers *Zero for executables*	Number of Relocs *Zero for Executables*	
	+3 +2	+1 +0	

Section Flags	+24	Memory							R	Res	Res	Linker					Content			Type					See spec 4.1 for further discussion.
		WRITE	READ	EXEC	SHARE	noPAGE	noCACHE	discardible				COMDAT	REMOVE	OVERLAY	INFO	OTHER	BSS	IDATA	CODE	COPY	noPAD	GROUPD	noLOAD	DUMMY	
		+3								+2		+1					+0								

Note 1: There is an escape clause allowing longer names. "Top to Bottom" says names have to start with period.

The example, finger.exe, had the following nonzero section table entries:

#	Name	RVA Start	Virtual Size	Raw Size	File Pointer	Section Flags	Notes
1	.text	1000	73E	800	400	60 00 00 20	Read, Execute, Code
2	.bss	2000	230	0	0	C0 00 00 80	Read, Write, Uninitialized
3	.rdata	3000	10C	200	C00	40 00 00 40	Read, Initialized
4	.data	4000	88	200	E00	C0 00 00 40	Read, Write, Initialized
5	.idata	5000	2C2	400	1000	40 00 00 40	Read, Initialized
6	.rsrc	6000	A9C	C00	1400	40 00 00 40	Read, Initialized
7	.reloc	7000	0D4	200	2000	40 00 00 42	Read, Initialized, Noload

The section table, in the examples, was followed immediately by an undocumented ASCII string identifying the linker which read "Version 2.50.4048 (NT)" and the remainder was padded with zeroes up to the first image page. Images were found at the file pointers indicated in the section table above.

The locations and sizes in the COFF header are consistent with the above. Code indeed starts at RVA 1000 and has an image size of 800. Data starts at RVA 2000. Uninitialized data has an "image" size of 400. Initialized data image size is 1600. Entry point of 1543 is in the **.text** section.

7.4.5 Code Image Pages (.text)

Some general rules apply to image pages; see Spec sections 5. and 5.1.

"Top to Bottom" on page 16 (.text) claims that the entry point, occurring in this section, is preceded *immediately* by the IAT table, consisting of a series of fixed up jump instructions. It says: *"When Windows NT executable images are loaded into a process's address space, the IAT is fixed up with the location of each imported function's physical address. In order to find the IAT in the **.text** section, the loader simply locates the module entry point and relies on the fact that the IAT occurs immediately before the entry point. And since each entry is the same size, it is easy to walk backward in the table to find its beginning."*

The entry point, RVA 1543, is 543 into .text which begins at 400 in the file. 943 in the file is in fact a 55 (EBP push) which might have something to do with reality. The immediately preceding memory certainly does not look like a table of jumps. Having inspected several .exe files and having found that the IAT is typically located in the **.idata** section, while the entry point is typically in the **.text** section, the above assertion is clear nonsense and deprecates the credibility of the rest of that paper.

7.4.6 Uninitialized Data Pages (.bss)

This is simply allocated, reserved storage. Presumably the loader zeroes it before execution. On the unix model, we had made a large **.bss** into which the talker loaded our program. We had to specifically tell the unix linker to make the section RWX and we shall have to do the same with the **.data** section that receives the code image for our sF system.

7.4.7 Initialized Data Pages

The exact loader treatment of these is not well documented. Care is essential since the virtual memory (swap) manager is certainly entitled to use the original execution load area for swapping backup of all read only entities.

Note that some of the tables pointed to by the COFF header occur inside specific ones of these sections. Note also that some of the names are acknowledged to be "magic" in the Spec. To avoid trouble perhaps we should stick with the standard names.

7.4.7.1 Read-Only Data (.rdata)

Nothing prima facie special about the section itself. However, in the example this section begins with 54 bytes of Debug Table which has one confusing entry for COFF type followed by two entries that describe the otherwise undocumented image page at 2200.

7.4.7.2 Read/Write Data (.data)

This segment type is the best candidate for preliminary testing of our system. All we need do is mark it for execute permission.

7.4.7.3 Import Section (.idata)

Refer to Spec section 6.4. The entire section is covered by the "Import Table" description in the example's Data Directory. The first part of the section consists of a sequence of twenty byte entries, one per DLL referenced at load time, called an "Import Directory." The end of the table is marked by an entry of all zeroes. A table entry is constructed as follows:

Import Dir	+0	RVA of Import Lookup Table (ILT) for this DLL	See ILT below
	+4	Time/date stamp of DLL *after bound at load time* In example, is set as though bound. Zero in fresh C++.	
	+8	Index of first "forwarder reference" in some sort of chain In example, is -1 In a fresh C++ make is 0	Undefined!
	+0C	RVA of DLL hint/name string. Format of hint/name: Halfcell index to Export Name Pointer Table for target DLL (the hint) ASCII string null terminated & null padded to be even.	
	+10	RVA of Import Address Table (IAT) *Initially identical to ILT until bound, then just virtual addresses.*	
		+3 +2 +1 +0	

Although the documentation says that the fields at +4 and +8 are updated when the DLL module is bound we have not found this to be the case. It remains a mystery how the FINGER example's .idata section came to be the way it is.

7.4.7.3.1 Import Lookup Table (ILT)

The ILT (and the initial IAT) is a sequence of single cell entries ending with a zero cell. The active entry format depends on its sign bit:

ILT/IAT					
	+0	0	RVA of Hint/Name string for DLL entry point		
	or...	1	Ordinal Number for DLL entry point *Note that not all entry points have names!*		
			+0	+1	+2	+3

7.4.7.3.2 Strange things about importation

Examination of some PE files has made it clear that the Linker knows with certainty which entry points go with which DLL's, at least insofar as USER32, GDI32, ADVAPI32, and KERNEL32 are concerned; and that it has good suspicions about what those DLLs' import tables look like. Perhaps this means that we will need a utility to strip those import tables from the DLL's for reference at some point. Additionally, there are two species of .exe elements we have seen, one of which looks as though its import tables have been bound already. This, too, is certainly food for thought.

How does the Linker know which entry points go with which DLL's? Searched the VC++ directory for some entry point and DLL names with no success. This is a mystery so far.

How does the Linker come up with hinting Export Name Pointer Table indices for DLL entries?

Why do some .exe's, such as FINGER.EXE, have the appearance of having been bound already (Date/time stamps; -1 in forwarder chain; IAT filled with Virtual Addresses instead of ILT entries) while others, such as those freshly generated by VC++, look more like the spec (zero for date/time stamp and forwarder chain; IAT identical to ILT)?

7.4.7.3.3 Importation Assumed by System

The system calls our entry point by diving through a stub contained in KERNEL32.dll. It turns out that the system simply *assumes* we will have asked it to load that DLL for us. If we fail to do so, the system traps on the way to our entry point because the above DLL is not mapped into our address space.

7.4.7.4 Export Data (.edata)

Refer to Spec section 6.3. This section is only applicable for DLL's, which we will eventually learn to make but which we do not bother with now.

7.4.7.5 Thread Local Storage (.tls)

Refer to Spec section 6.6. Not used presently, but this may prove handy in the future. It's tantamount to a user area at the level of Microsoft's subtask model (threads). Thread local storage can be allocated on the fly, but the tables in this section enable the loader to take responsibility for allocating memory. However since this is in effect only a pointer table, thread local storage may not use static addresses to access its local storage. This seems a bit strange inasmuch as a thread presumably uses a different memory map than do other threads of the same process. Wonder why they did it this way.

7.4.7.6 Relocation Section (.reloc)

Refer to Spec section 6.5. Perhaps if we are lucky we can get away without any of these for at least a while. The Data Directory entry "Base Relocation Table" describes only the first 90 hex bytes of this section, although its total length is D4.

7.4.7.7 Resource Section (.rsrc)

Refer to Spec section 6.7. The entire section is covered by the "Resource Table" entry in the example's Data Directory.

7.4.7.8 Debug Section (.debug)

Refer to Spec section 6.3. This body of data is specific to Microsoft debuggers; since sF provides its own debugging tools, and since much of this stuff is not applicable to our execution model anyway, we happily need not study it. Our PE files do not have **.debug** sections. *The example does not document inclusion of a debug section anywhere, yet it seems to have one at File Address 2200 which looks like a section image but which is only mentioned from within the debug table. It would seem that one may hide whatever junk he likes in an executable if he is careful about it.* The Data Directory entry "Debug Table" of the example refers to the **.rdata** section above.

7.5 Generating a PE file on a Windows platform

The following steps will write a Windows .exe (PE) file:

1. COMPILER LOAD NT LOAD
2. EMPTY 540 LOAD
3. upload filename.exe

The file will be produced in the working directory (such as C:\ATHENA\sF-NT\) unless you write a full path for the filename.

7.6 Crossgeneration from a Native system

We crosscompile from a regular sF system. The steps for making and trying a new system are:

1. COMPILER LOAD NT LOAD
2. EMPTY 543 LOAD
3. xx ftp (where xx is machine for testing)
4. user xxxx
5. pass yyy
6. Select directory
7. say type i
8. upload filename.exe
9. quit

8. External References

The following sections identify conceptual, informational, and programmatic dependencies with which programmers should familiarize themselves in order to use this system effectively.

8.1 Related Documents and Standards

WINNT.H From current Win32 SDK: Primary reference for PE structure.

FINGER.EXE from NT 3.5 final workstation build 807 (and several others): Verification of PE structure.

Clearer, more Comprehensible Error Processing with Win32 Structured Exception Handling Kevin Goodman, from VC++ 2.0 Technical Articles. Description of Win32 exception dispatcher data structures largely from point of view of a C programmer.

EXCEPT From MSDN Oct 94 Sample City/win32/windows NT. EH.ASM is a more illuminating, but still incomplete, illustration of language independent Structured Exception Handling. The .doc files promised in the abstract seem to be lacking on the CD.

EXCPDISP.WRI is an annotated disassembly of the *actual* exception dispatcher logic from NT Wks 3.5 which was needed to complete an understanding of Structured Exception Handling.

Microsoft Portable Executable and Common Object File Format Specification 4.1 From Oct 94 Developer Network CDROM, Specs section: Clarification (and some confusion) of PE structure.

Stepping Up to 32 Bits: Chicago's Process, Thread, and Memory Management Matt Pietrek, MSJ 1994 #8 (on the MSDN CD's) is a good sanity check for which Win32 features will be compatible between NT and Chicago.

The Portable Executable File Format from Top to Bottom From Oct 94 Developer Network CDROM, Technical Articles re Windows (32-bit) section: More clarification and confusion of PE structure.

VC++ Runtime Library Reference (doc DB52180-0693) Documentation for the C library routines in CRTDLL.dll

Win32 API Documentation is available from Microsoft in various formats, including a 5-volume set, the CDroms and the website. There are times when each is necessary. You should definitely have both if you plan to do much work on this system.

8.2 Include File Dependencies

Values and structures used to communicate with the Win32 API have been abstracted from as small a set of Include files as practical. The versions referenced are copied in the base package for comparison with those in new releases. The files used, and the information obtained from them, are as follow:

EXCPT.H True callling sequence for exception filters. The enumerated type EXCEPTION_DISPOSITION which documents *actual* return values from exception filters, which are *not* the same as those returned by C exception filter stubs to the C exception filter handler (defined values are [0|1|2|3], not [-1|0|1]).

FCNTL.H O_RDONLY and other arguments for open.

STDIO.H SEEK_SET

TIMEB.H The _timeb structure for _ftime (to get local time offset)

WINBASE.H STD_INPUT_HANDLE and friends. INVALID_HANDLE_VALUE is -1.

WINCON.H CHAR_INFO structure for content of console screen buffer. CONSOLE_SCREEN_BUFFER_INFO and other simple structures used in calling sequences.

WINDEF.H 0 is the NULL value in this system.

WINNT.H PE executable header structures. Exception report record, and exception codes appearing therein, referenced in exception handling. Context frame, referenced and altered during exception handling.

WINSOCK.H WSAData structure. hostent structure.

9. Revision History

REVISION	DESCRIPTION
190526	First draft for support of saneFORTH/x86 distributed as part of GreenArrays arrayForth 3.
210227	Document procedures for making native (raw) media such as for booting on removable media including floppies and USB flash memories.

ATHENA Programming ™

Reference Manual SFW32PR *Revised 8/05/21*

IMPORTANT NOTICE

ATHENA Programming, Inc. is a Wyoming C Corporation founded in 1966. With the exception of the retirement of its Founder, Jack Perrine, PhD, in 2013, our company has operated continuously under the same ownership and management since 1970.

ATHENA's business began with creation, maintenance and enhancement of system software for Univac mainframes such as the 1107 and 1108, working mostly in machine code and FORTRAN V. In the early 1970s we began writing code for minicomputers, and in 1975 we intentionally shifted our business exclusively to systems and applications programming in Forth, working in collaboration with FORTH, Inc. as well as serving our own customers.

Our business model has always been to work in a nurturing, co-operative way with a small number of long-term OEM customers, providing them with secure, reliable software and helping them develop and maintain their own products. ATHENA has been largely invisible to the general public because our work, and our products, have been in the background relative to those of our customers. Relationships have generally lasted until a customer went out of business or its products reached end of life, often through acquisition; our relationships with our most durable customers are approaching their fortieth anniversaries.

ATHENA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. In practice, though, ATHENA has a policy of maintaining continuity and upward compatibility in its software for as long as any current customer depends on it.

ATHENA disclaims any express or implied warranty relating to the sale and/or use of ATHENA products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

In general, we license our software to a customer without limitation on the number of active instances the customer may install or use. In general, a customer is authorized to deliver our software to its own customers, as the vehicle for operating our customers' products, with suitable limitations on reproduction or further dissemination by the end customers. In general, our customers may not resell or otherwise release our software to third parties when configured as software development systems. Thanks to our non-adversarial relationships with our customers over the decades, these general terms have been respected as a matter of Honor.

Mailing Address: ATHENA Programming, Inc., 821 East 17th Street, Cheyenne, WY 82001
Printed in the United States of America
Phone (971) 235-2385 email sales at athenaprog (tld) com