saneFORTH™

Programmer's Reference
Release 5g *preliminary*

ATHENA's Generic Model for x86

The reader should be familiar with FORTH, Inc's polyFORTH 32-bit model. ATHENA essentially defined that model for and in cooperation with FORTH, Inc. in 1983 when ATHENA implemented the microcoded FORTH system for the NCR 9300 series computers and edited the polyFORTH Reference Manual to describe the 32-bit model. That model was followed in the implementations for the Intel 386 and DEC VAX. That model forms the foundation for this system.

The reader should also be acquainted with ANS FORTH, X3.215-1994. As far as that document goes, which does not include such essential features as multiprogramming or I/O, it is a good foundation for definitions of basic words and for background on innovative, but extremely useful, features such as exception handling (CATCH/THROW).

This document is under construction and this warning will be carried forward in future editions until it has been completed.

Contents

1. Introduction

This is the primary Programmer's Reference Manual for ATHENA's native saneFORTH system running on x86 processors in 32-bit mode. It has been composed and maintained as time was available; hence there will inevitably be areas that are not exposed well, or indeed at all.

1.1 Related Publications

- **SFPRM** ***saneFORTH Programmer's Reference*** (this manual) describes features of ATHENA's system for x86 (IA32) computers configured as PCs.
- **SFIG** ***Installation Guide for saneFORTH*** covers details of configuring the basic system.
- **sFW32PR** ***Programmer's Reference for sF/NT*** covers details of sF on the Win32 platform.
- **IPPR** ***Programmer's Reference for ATHENA TCP/IP Package*** covers use of the networking code.

1.2 Documentation Conventions

1.2.1 Numbers

Numbers are written in decimal unless otherwise indicated. Hexadecimal values are indicated by explicitly writing “hex” or by preceding the number with the lowercase letter “x”. Note that as of 5b this same convention works in the interpreter and compiler.

1.2.2 Bit Numbering

Binary numbers are visualized as a horizontal row of bits, numbered consecutively right to left in ascending significance with the least significant bit numbered zero. Thus bit *n* has the binary value 2^n.

1.2.3 Ranges

Standard mathematical range set notation is used with square brackets; inward square bracket is inclusive, outward is exclusive. For example [5..7] denotes the set of values {5,6,7}. [5..7[denotes {5,6} while]5..7[denotes {6,7} and]5..7[denotes {6}. The FORTH word `WITHIN (n l h - t)` is true if `n` is an element of [l..h[.

1.3 Provenance

1.3.1 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

1.3.2 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.

Eventually this section will document major differences, while an annotated version of the TC's final draft will be made available to document compliance in detail.

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.

1.3.3 System Disk and Source Organization

1.3.3.1 Boot Areas

For 32-bit machines and emerging hardware for human interface and mass storage, it proved necessary to enlarge the Nucleus considerably beyond the once-practical 8k bytes assumed by the polyFORTH model. 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. Sixty blocks seemed sufficient so in a normal environment OFFSET is places sixty blocks beyond the start of a given chunk containing code.

1.3.3.2 Larger-Scale Disk Organization

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

1.3.3.3 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-5e ***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**	**IPSec/IKEv1**		*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*	PKI		*free*
	1680	Graphics demos	(==)	*empty*	AHCI-SATA		*free*
	1740	***Conc/Floppies***	==	*empty*	USB, other I/O		*free*
	1800	iF/V16	(==)	*empty*	***X Client***		==
	1860	Graphics	(==)	*empty*	***X GUI***		==
	1920	Graphics	(==)	GUI 1/2	PDF Printing		***GUI 1/2***
	1980	Lab Stds	(==)	GUI 2/2	i350/211/219 gbe		***GUI 2/2***
	2040	Lab Stds	(==)	*empty*	Turnkey, attacks		XIE Testing
	2100	Trees/uMap	(==)	Wks I/O	People/Calendar		***Wks I/O***
	2160	***SDS Files***	==	Loop I/O	Full native USB		***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***	==	==	==	==	==

1.3.4 Other Related Documents and Standards

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]

2. Critical Functions and Structures

2.1 Memory Allocation

As of the 4d level for PC compatible boxes, the system variable area which previously lay at 300 hex has been moved to the end of the 32k boot image area. The space down low was becoming tight, and we had previously made the decision to place a number of system variables in the PROM image (for example, the soft write protection table) simply because there was not sufficient room down low. At the same time we chose to leave `0PROM` set at 400 hex since this will allow old and new versions of `TESTING` to interoperate without creating programmer traps, at a cost of 256 bytes. The resulting default memory map in released systems is as follows:

	Origin (hex)	Usage	Size (hex)
	100000	The rest of RAM. HERE skips to this point during 9 LOAD.	to MAXRAM
	80000	"The Hole" On worst case ISA boxes starts at 80000 which is what the system defaults to. Set to A0000 if not a problem.	512k worst case 384k on most boxes
	7FC00	Single default block buffer	400
STATUS	7F900	OPERATOR's User Area	300
HERE	8400+v	OPERATOR's Dictionary after nucleus boot	77500-v
0RAM	8400	System Variable Area	v
0PROM	400	Nucleus PROM Image	8000
	300	Reserved	100
	200	Global Descriptor Table (GDT)	100
	0	Interrupt Descriptor Table (IDT)	200

If booting and running easily on arbitrary boxes is not a concern, increase low memory limit to A0000 in block 181. If the default user area is not large enough, or if you need a deeper return stack, change `#U` or `#R` in block 181. If you must recover the 256 unused bytes in low memory, change `0PROM` in block 181 and don't forget to maintain `TESTING` and the target output comparison code in block 470. However, if you are short of memory it is more productive, and less trouble, to change `0RAM` in block 181 since this has no effect on other utilities and should make no difference to the applications. Naturally, you must retarget after chaning any of these parameters in block 181.

Automated memory sizing is on our list of things to do, as is mechanism for dealing with boxes supporting address widths larger than 24 bits but which leave a second hole for the BIOS PROMs just under the 16 meg point around FF0000. As of version 4h, the upper limit of usable RAM is specified by the constant **`MAXRAM`** defined in block 2, as opposed to editing changes in block 50. When block 50 is loaded, to skip over the "hole", we check for the existence of the last cell of memory implied by `MAXRAM` and abort if it is not read/writable.

2.2 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.

2.3 Low Level Data Base

2.3.1 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.

2.3.2 User Variables

2.3.2.1 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

2.3.2.2 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.

2.3.2.3 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.

UFLG

	7	6	5	4	3	2	1	0
+0				Ph2 LDV	-MAN	bg log	<ex	no msgs
+1	reserved							
+2	reserved							
+3	reserved							

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.

-MAN set to indicate this task is unmanned, such that it is futile to ask for guidance from the operator.

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.

2.4 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+ @ ;
```


2.5 Exception Handling

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

2.5.1 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.

2.5.1.1 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.

2.5.1.2 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** .

2.5.1.3 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

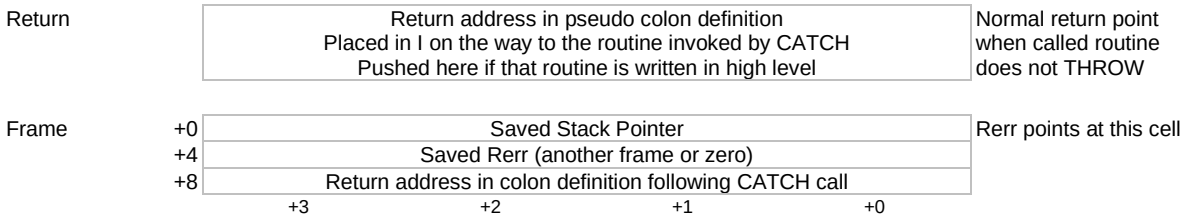
the environment in which low level exception handling routines run, is subject to change with any update. The control this gives you offsets the sensitivity and it is your tradeoff to make as you find appropriate and necessary.

2.5.1.4 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.

2.5.2 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.

2.5.3 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	
TC_E + 7	-4088	Irresolvable DNS/[ip]/host to IP address	Code only used in npRESOLVE
Unassigned	-4087		*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
XC_E + 2	-4078	Timeout (including application level)	for these; ordinarily, X Client
XC_E + 3	-4077	Error	tasks use the XLOG facility
XC_E + 4	-4076	Reset	to record such messages.
XC_E + 5	-4075	Refused	
XC_E + 6	-4074	Unexpected Disconnection	
Unassigned	-4073		*eight values*

2.6 Facility Management

Sharable, queued and registered facilities forthcoming.

2.7 Bootstrapping

Initial (power on) bootstrap procedures vary depending on whether the system has sF in nonvolatile memory as is common with embedded systems. In general, bootstrapping requires placing an executable copy of the nucleus into memory (where it may already exist); initializing system memory and data structures; initializing hardware; and, optionally, commencing the interpretation of source to load and run an environment and/or an application. "Rebooting" refers to repeating this process and is supported in PROM based systems just as well as it is in those which must obtain the nucleus from other storage media.

The system may be rebooted under program, operator, or keyboard control. *Since these methods may be used in cases of system malfunction, they intentionally do not FLUSH the block buffers; you must do this explicitly if it is desired.*

2.7.1 Programmed facilities

The word **RELOAD** performs a soft reboot of the system. The nucleus image is preserved and reused; otherwise, system and user variables are initialized as they are on any other boot. This is, usually, the preferred method for rebooting since it is fast and in addition it is more useful for debugging purposes since, unlike the BIOS boot, it does not clear all of memory before beginning the boot process. There are other differences between the function of RELOAD and that of a BIOS boot. Most notably, the hardware reset line is not asserted. The only hardware initialization that is performed is that which is done directly to hardware registers and other resources, typically performed before enabling interrupts or DMA on the devices.

There are cases in which RELOAD is by itself unsafe. For example, DMA devices may still be operating after a RELOAD and this can lead to problems during program construction and compilation. In addition, device FIFO's and other resources will very likely be found in states other than those which follow a hardware reset. Since RELOAD is very useful, most of our I/O routines are coded to take extra effort during initialization in recognition of these states. However, for things like DMA devices that is not enough. Further, some applications must for good reasons patch the nucleus, and these patches survive a simple RELOAD with the result that the newly warm booted nucleus may contain references to code or data that lie somewhere out in memory where they were created by the prior boot.

'reload is a system variable, added at release 4f, containing the address of the machine code that is executed by RELOAD. As part of warm or cold boot initialization, this variable is pointed at the default warm start code. If the application creates situations requiring additional cleanup activity that should be performed in the course of a warm boot, it may incrementally prepend these additional activities to the functions of RELOAD by using 'reload . These activities must all be described in terms of machine code, and it is vitally important that they be written in such a manner that they will always operate deterministically and always succeed. An example follows in which the application is about to patch the code field of ?ABSENT to point at application specific code. Before applying the patch, the application could include something like the following:

```
ASSEMBLER BEGIN ... ' ?ABSENT 4- @ #  ' ?ABSENT 4- MOV
   'reload @ JMP  'reload !
```

The result of the above is that this correction to the code field for ?ABSENT becomes the first thing done by RELOAD. The jump at the end of the routine transfers to whatever other activities RELOAD had previously been configured to perform.

The word **COLD-BOOT** attempts to produce the same effect as does pressing the RESET button on the computer. It does this by instructing the keyboard processor to assert the system reset line. Unfortunately, the exact effects of asserting this line are processor dependent. For example, on many processors it asserts the RESET signal on the motherboard but does not do anything on the ISA bus. On such machines, peripherals in ISA slots will not be reset and this may lead to nondeterministic side effects such as DMA devices still enabled, or as is the case with some Ethernet boards, the 16-bit ISA memory access signal being asserted for a 128k block of "hole" memory which contains 8-bit BIOS ROM's that we do not need for RELOAD but which the BIOS needs for its processing. In such cases, COLD-BOOT may be unusable. Moreover, the semantics of the keyboard processor's RESET signal have been observed to lead to unpleasant results on some PCI motherboards, such that the BIOS will

attempt to boot without properly maintaining or consulting the PCI configuration. In such cases, we have seen some BIOS's which simply fail to find a boot device, while one AST box went so far as to decide this meant it had to move its PCI devices around. Thus, while `COLD-BOOT` may be useful, you should only employ it on operational machines that have been generically proven to react properly to the keyboard processor's RESET signal. Strange as it may seem, `RELOAD` is a far more deterministic reboot method, and on many machines it is better to use `TESTING LOAD TRY` than to use `COLD-BOOT`.

In release 5b, `COLD-BOOT` was made more robust by changing the definition of `'reload` slightly so that it could be called by both `RELOAD` and `COLD-BOOT` as a subroutine (with interrupts prevented). This addresses the `COLD-BOOT` problems resulting from failure to shut down activity on bus mastering devices. It does not, of course, help when the problem is a brain damaged BIOS as noted above. Fortunately, the PCI BIOS writers seem to have cleaned this up such that problems of that type have not been observed since the mid 1990's. The means for extending `'reload` shown above are unchanged and no code changes are required in any existing extensions.

In release 5e, we first found a box in which the BIOS failed to reproduce some control functions of the keyboard processor such as the reset function. Research revealed that there is an I/O register CF9 which has been supported by all Intel PCI chipsets from the first PIIX through all ICHex and PCHes such that writing hex 0E into that register performs a hard reset, including in later boxes a simulated power cycle. As of 5e, `COLD-BOOT` now does both the keyboard reset and the CF9 reset, thus working on all known boxes.

2.7.2 Keyboard Interrupt Facilities

To facilitate emergency recovery from system, application, or task hangs, the keyboard interrupt code makes special provision for handling of certain keystrokes. It is important to realize that this is *not* the Microsoft "control-alt-delete" business, which is unavailable in our systems. The processing is *performed by keyboard interrupt code,* so if the system has crashed with interrupts prevented these keystrokes will not work. In fact, no response to these is the best *evidence* of that sort of problem. Each of the keystrokes should be used with great caution, as described below. If you do not wish to make these functions available to the operator in an operational environment, the phrase `2 Keys 4+ C!` disables them.

`Ctrl-ESC` causes the interrupt code to prevent interrupts and execute the function of **`RELOAD`**. No block buffers are saved, but all of the specified cleanup work is done.

`Ctrl-Alt-ESC` causes the interrupt code to attempt a **`COLD-BOOT`**. What happens next depends on the characteristics of the motherboard and BIOS in use, but if all is well the BIOS will reboot the system.

`Alt-ESC` causes the interrupt code to force a hardware **TRAP**, just as though the code which it interrupted executed an illegal instruction. This is a powerful but dangerous function. If task code is interrupted by the keyboard, then that task will take a trap, and depending on state variables it may attempt to format a trap display and throw an exception, which by default is equivalent to an abort. Note that this task will not necessarily be the `OPERATOR` task. If all tasks are idle, the interrupt may take place during the dispatcher loop in which case the last task executed will be trapped. If the keyboard interrupts some other piece of interrupt code, such as network, which is running with interrupts allowed, then the results will depend upon whether that interrupt code has left registers `U` and `S` intact. Since `U` is often changed by such interrupt code, the results of the trap will be nondeterministic. For all of these reasons, there are in general only two cases in which it is useful to employ `Alt-ESC`. If you are testing new code in the `OPERATOR` task and believe that this code is in a CPU loop that does not `PAUSE`, you will most likely trap that code cleanly with `Alt-ESC`. Or, if you have enabled the panic dump mechanism, you may expect a clean and usable panic dump (along with the immediate halt of all system operations). In most other cases, the results of `Alt-ESC` will be difficult to predict or analyze, may compromise the system's integrity, and should in general be avoided. In particular, using `Alt-ESC` in other cases that you may well mess the system up enough that it can no longer process `Ctrl-ESC`, even though it might have been able to before you did that.

2.7.3 ISA/PCI PC Bootstrap Routines

Three bootstrap routines are available for reading nucleus images from BIOS supported media. Two are BIOS device specific, and are preprogrammed as to which device contains the nucleus and what geometry (or, in the case of many mass storage devices, what *virtual BIOS geometry*) to use with the BIOS calls. The third always reads the nucleus from the same BIOS device

as the boot routine was read from, and adapts to the *virtual geometry* reported in response to a BIOS query. All three are designed to be loaded by the BIOS as the level zero boot sector (absolute sector zero) on whichever medium they reside.

According to original IBM BIOS documentation, sector zero of the boot medium is read into memory at `7C00` and control is transferred with `CS=0 IP=7C00 DL(2 lo)=logical boot device.` So far this documentation has proven to be accurate.

All of the boot routines depend on the BIOS for reading the nucleus, mainly because of the difficulty of fitting code to talk to the devices into 512 bytes. Over the years the (undocumented) memory needed by the BIOS to perform mass storage operations has changed once, requiring that the nucleus be read into memory above 64k to avoid further encroachments upon low memory by the BIOS. This occurred at about the time the 440BX PCI chipsets were introduced, which is why some of our older boot routines will not work on some late model mother boards using late model BIOSes.

The central problem in talking to the BIOS about mass storage is that the only publicly documented interface we know of is the basic read-sectors function, to which the disk address is specified in terms of cylinder, head, sector. This is the case even with inherently linearly addressed devices such as SCSI disks and LBA (Logical Block Addressing) IDE devices. In order to read a nucleus, 40k as of sF/2a (to accommodate the microcode for Symbios chips, for example), it is necessary that the boot routine agree with the often virtual geometry used by the BIOS at least insofar as number of sectors per track and, if that parameter is small enough as is the case with floppies, the number of heads, is concerned. This can be problematic when using SCSI BIOS extensions conforming with SCSI-2 and SCSI-3 standards since those prescribe that the BIOS query the drive for size and adjust the virtual geometry by an algorithm. Of the three supported boot routines, only the Universal boot is capable of adapting to this last situation.

2.7.3.1 Floppy Boot Routine

2.7.3.2 Hard Disk Boot Routine

2.7.3.3 Universal Boot Routine

Like the others, this routine begins with an unconditional jump past a set of local variables. During execution register `W` points at this 2-byte jump instruction and the variables are indexed (all halfcells) as follows:

```
 2 W)  Sectors per track = max 1-relative sector number
 4 W)  Number of sectors to read (one sector more than nuc size)
 6 W)  Nucleus entry point
 8 W)  Number of heads per cyl (not max head number as they give)
0A W)  hhhh hhhh dddd dddd  Start head & drive (d hi 1 => hard)
0C W)  llll llll hhss ssss  lo/hi start cyl, sector (s norm. 1)
0E W)  Save area for reg 1 returned by read drive parameters.
```

The first action of the boot routine is to copy itself (all 512 bytes) from `7C00` to the designated execution area `'BOOT` which is normally `F000`, jump up to continue executing in that area, initialize `W` as pointing to that area's origin, and leave registers `DS`, `ES`, and `1` all zero, and interrupts allowed. This is done so that the boot routine will be out of the way when it later moves the nucleus down to low memory (note that this limits nucleus image size to 59 Kbytes when the system origin is at `0400` as it is with the sF/2a.)

Once operating, the Universal Boot at version 1e provides for displaying hex numbers on the VGA screen as snapshots for diagnosis of any future BIOS I/O problems. The values displayed are in order:

```
- R2 from boot entry with logical device number in low byte
- R2 from read parameters hhhh hhhh nnnn nnnn
- #heads derived from that R2 value
- R1 from read parameters cccc cccc ccss ssss
- R1 masked to #sectors
- R0 status from each sector read operation (high byte)
```

We next OR the logical boot device number from register 2 with the contents of the start drive number at +0A in the boot block, normally zero. This is a kludge to permit continued use of Intel AN430TX motherboards with the Universal Boot. A considerable number of these have been deployed in the field, and as of the time Intel ceased providing BIOS maintenance there was a bug in the BIOS such that on hard disk boot it gives a logical boot device number of 00 for C: where the correct, standard value is 80 hex. If a Universal Boot image is written to a hard disk that might be used with one of these motherboards, the value in +0A should be hex 80.

2.7.3.3.1 Universal Boot communication area

The universal boot writes the second sector of the boot block into memory starting at 200 hex. Much of this is overlaid by the IDT and GDT areas, but the area from 2B0 to 400 is presently free.

At the end of the boot process, the boot routine overwrites several of these cells with data obtained from the BIOS pertaining to the boot device and legacy PC hardware configuration. The rest survives from the text of the boot block and may be examined by the nucleus.

The layout of this space is as follows. "*" indicates values that are stored by the boot routine. "h" indicates a halfcell value.

```
   3E0* Drive number from which booted acording to BIOS
   3E4* Sectors per track on boot dev according to BIOS...
   3E8* Number of heads on boot dev
   3EC* Max 0 rel cyl number  llll llll hhss ssss -> h|l

   3F0  If nonzero, overrides load block when autoloading.
   3F4  Overrides #dd if latter is zero.

h 3F8* Value from BIOS mem 11A
h 3FA* Value from BIOS mem 118
h 3FC* Equipment flags from BIOS 11 INT.
h 3FE* Equipment flags from BIOS 11 INT.
```

2.7.3.4 URAM Boot Routine

This boot routine is built on the Universal Boot and may be used on any BIOS bootable medium other than optical disk. It's intended to install a system onto a machine that has no floppy disk controller. To accomplish this there are several major differences from the normal sF boot medium and mechanism.

The nucleus boots normally using the BIOS but must be configured for RAM disk. When RAM disk is enabled, the system disk blocks from 5,700,000 to 6,000,000 onto RAM starting at absolute hex address 200000 (2 megs.) -2 DRIVE sets OFFSET to use this area. The nucleus is configured for a given amount of space in this area, and the boot is configured to load that much additional data from the boot medium using the BIOS. Thus, the system may perform a 9 LOAD from RAM disk, and may have full entitlements depending on how much is loaded; this should be enough to explore the machine, access the hard disk, copy the system onto it, configure for network interfaces, and so on, without actually operating on the boot media as such.

The RAM disk area begins at 200000 instead of 100000 because we have encountered BIOS that refuse to read from the boot medium into the 1 meg (or perhaps any odd meg) areas. If you are booting with more than 1000 blocks of RAM disk be sure to check that block 1024 does not look like block 0.

2.7.3.4.1 Making URAM Boot Media

Let's make a bootable USB flash that loads 4800 blocks into RAM disk. This requires three steps.

2.7.3.4.1.1 Configure and Compile Nucleus and Boot

In block 182, change `?_rd` from 0 to 1. This enables compilation of the RAM disk code (291) in the nucleus and enables mapping of the RAM disk at `DRAM` (normally 5700000) in blocks 264 for basic system and 773 for network. In block 50, when the RAM disk is present indicated by `DRAM` defined, HERE jumps to hex 700000 instead of 100000, leaving room for 5 megs of RAM disk (this position is set in block 50.) You will probably also want to ensure that `?auto` is set to zero in this nucleus. Compile this nucleus `COMPILER LOAD ISA LOAD` as usual.

Configure the URAM boot routine for the amount of RAM disk to be copied to RAM disk. In block 102 there is a `CONSTANT` called `#bs` (boot sector count) which is calculated by doubling a literal number of blocks. For this configuration that number should be 4800. Compile this boot routine and combine with the nucleus just made in the boot area of the current system image by saying `TESTING LOAD URAM BOOT INSTALL .`

2.7.3.4.1.2 Transfer 4800 Blocks to sF/NT

This is usually done with FTP. You may want to simplify block 9 to avoid needing to edit it initially. For example, don't load the concordance, normally done at the end of block 9.

```
1. NETWORK 720 LOAD
2. <host> ftp
3. user <id>
4. pass <id>
5. say type i
6. -60 4800 >blks <name>.blk  (typically in ATH)
```

Once it's there, use Windows to move the file to a place convenient to name with a path in sF/NT.

2.7.3.4.1.3 Write 4800 Blocks to Flash

Presently doing this with GLOW system. Run it as Administrator. Check 93 and make sure it does not load the stuff at 95 yet.

```
1. Insert flash and using disk manager find out what its drive number is.
2. 93 LOAD
3. SRC: <name>.blk
4. OLD: //./physicaldrivex  ( x=7 in this case)
5. SRC DEST 4800 BLOCKS  SRC DEST 4800 MATCHES
```

If the flash in question has had partitions assigned you may need to delete them, either using Windows or by verifying the code in 95 and enabling it.

2.8 Mass Storage Management

2.8.1 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.

2.8.1.1 Local Machine Block Address Space

For the local machine, disk class mass storage begins at absolute zero, where zero is normally block 0 of the boot device. Mapping of remaining storage depends on the system level. The key infrastructure of the system is organized so that a functional system's source may be carried on a 1440-block floppy, but on mass storage devices the basic "drive" size for source code has been set at 4800 blocks for many years.

In all systems, the first local floppy drive appears at the constant **FLEX** which defaults to 3,000,000 and the second, which must have the same media type as the first, begins at **FLEX** plus selected media size, almost always 1440 blocks. **FLEX** is by convention at the upper end of local block address space.

There are two generationally different ways in which the system maps the address space between 0 and **FLEX** , as follows:

2.8.1.1.1 Before PCI-ATA Implementation

IDE systems support only one drive at zero, and are not combined with other mass storage types. The drive being used is selected by #un which is a target compilation parameter and may not be changed dynamically on a running system. The IDE code may be used with PCI-ATA host controllers but only if legacy IDE mode is supported by the chipset in use.

SCSI systems support multiple drives, with disk space described by the **DRIVES** table which has a 12 byte entry for each chunk of address space:

DRIVES

	+3	+2	+1	+0
+0	Size of chunk in blocks (-1 indicates end of table)			
+4	"yy" parameter for SCSI read/write	"xx" parameter for SCSI read/write	$\log_2$ of number of sectors per block	"cu" parameter for SCSI ops
+8	Address of SCSI locator structure to use with this chunk			

Default table has space for two entries.

See the old SCSI code for use of this table.

2.8.1.1.2 After PCI-ATA and AHCI-SATA Implementation

In systems supporting PCI-ATA mass storage, the **DRIVES** table has been replaced with the **DISKS** table which can support all of the device classes:

DISKS

	+3	+2	+1	+0
+0	Size of chunk in blocks (-1 indicates end of table)			
+4	0	0	$\log_2$ of number of sectors per block (is 1 for ATA)	SCSI c/u, ATA logical port, host number
+8	SCSI Locator address or code indicating other storage classes: 0 = Floppy, 1 = PCI Native ATA			
+12	Starting block number on physical device, normally 0			

Default table has space for ten entries.

The mechanism for performing the mapping is an extension of that in the old SCSI support, working as follows:

SEL (n nb - n ns p t | -1 0) Finds block **n** in the **DRIVES** table, returning arguments for a read/write operation and true, or -1 (for bad status) and 0 if the block number exceeds the range described in the table. The block addressing has been converted to sectors and biased from the start of the physical drive as shown.

UNIT (i - n) Returns starting block for the **DRIVES** table entry whose zero-relative index is given. If there are **x** entries in the table, **x UNIT** returns that total number of blocks described in the table as does any larger argument.

<ATA (a n nb - a s) Reads **nb** blocks from mass storage starting with absolute block **n** to memory at **a** . The operation may not span a 64k boundary in memory.

>ATA (a n nb - a s) Write operation symmetric with **<ATA** :

2.8.1.2 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.

2.8.1.2.1 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) is a **CONSTANT** whose absolute block number is the origin of the local machine's first floppy.

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** .

A: etc to be documented

2.8.1.2.2 Managing Mapping in PCI-ATA and AHCI-SATA Capable Systems

With the **DISKS** table design has come a vocabulary for displaying and changing the mapping of local **BLOCK** address space. This mechanism is loaded by block 3 early during the **9 LOAD** and may be used in block 6, which is loaded immediately after the following words are defined, to effect any desired mapping. These words are safe, in that the buffer pool is hard-flushed before making the changes. Nevertheless these words should not be used on an active system in which other tasks may be reading or writing parts of the address space whose meaning is being changed, and it is inadvisable to use them to change the meaning of the address space underlying the **9 LOAD** itself. The remapping functions each maintain **FLEX** and **B/box** appropriately, keeping them at their standard positions unless the aggregate space described in **DISKS** exceeds 5,400,000 blocks (5.4 GB).

.SYS (n) Identifies the system we are running based on line 0 of block 7 on its boot media, and shows the **BLOCK** address space mapping currently in effect. It's advisable to use **.SYS** after each of these functions to maintain sanity and avoid mistakes when remapping.

!SIZE (n i) Changes the size in blocks of unit **i** to **n** .

!PORT (n) Changes the logical ATA port number of unit **i** to **n**.

!ORIGIN (n) Sets the origin block of unit **i** on its media (normally zero) to **n** .

2.8.1.3 Network Disk Address Space

When the TCP/IP package is loaded, net disk access is supported. In this environment, the 32-bit **BLOCK** address space is subdivided into a chunk of **B/box** blocks (by default, 6,000,000) for each computer, with the first **B/box** blocks at absolute zero being the local disk(s) mapped for the local machine. The default subdivision allocates a total of 6,000,000 blocks to each accessible computer including the local machine, which still begins at zero. Disk starting at each boundary **B/box*i** is mapped

onto the local mass storage of box **i** in the local machine's host table. Disk access over the net is transparent, behaving like any other **BLOCK** oriented mass storage device, and device status is retrieved from the remote system on each operation and reported in the same way as is status from local controllers, making net disk operations transparent even in the event of device errors or protection violations. Writing a block to a remote box cause it to be written to the physical media on that box immediately; as a consequence there is no need for a network **FLUSH** protocol.

When all systems on a network share a common value for **B/box** each can see the entirety of all others' **BLOCK** address space, and even floppies lie at the same relative position on the other boxes. It's for this reason that **FLEX** is forced up to 5400000 and **B/box** to 6000000 on machines whose disk subsystems are not that large. However, when the **DISKS** table is enlarged to map more than 5400000 blocks, it is necessary to move **FLEX** and **B/box** upward. When this is done, one must be aware that the number of addressable boxes decreases (2^{32} / **B/box**); the floppies on other boxes are not at the same place as one's own; and it is possible to attempt addressing of blocks on other machines that from their perspective would be on the network. Of course, if all boxes on the network have been adjusted to use the same **B/box** value, all but the first of these effects go away. In future we may reconsider how this is done, limiting access to remote boxes to 6000000 blocks each, but for the present time caution is indicated.

When net block access is configured in a system, the following words are added or redefined:

+BOX (n i - n) Adjusts block number **n** to be relative to box **i** with the same **OFFSET** on that box as is currently in effect on the local box.

+ABOX (n i - n) Adjusts block number **n** so that it refers to absolute block **n** on box **i** when used with the current **OFFSET** on the local box.

SCENE (n) Interprets **OFFSET** refer to the same relative position on box **i** as it does on whichever box it would currently lie within. If you are at **4 DRIVE** in some box and say **3 SCENE** you will now be looking at **4 DRIVE** in box 3.

PART (n) is augmented to select the given partition within the box that **OFFSET** currently addresses, effectively remaining within the current **SCENE** .

DRIVE (n) is augmented as is **PART** .

HUNK (n) is also augmented as is **PART** .

2.8.1.4 ATHENA Standard CF Medium

With the above all implemented, ATHENA now has a standard Compact Flash organization used to inseminate new boxes, vastly preferable to the URAM bootable floppies and USB memories of the past. The medium begins with a bootable development system, complete with server capabilities and file system. Following that are other major systems and archives. Typical minimum size is 32 GB.

2.8.1.5 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.

Systems: Although the needs vary by application, a full working system is a complete disk image of an application's code, data base and any backups. Historically 5,400,000 blocks have worked well for this so that with room for floppies and such there are 6 million blocks per machine and >350 machines addressable using positive block numbers. With the advent of PCI-ATA and the AVALUE box it is now practical to have three storage devices in a system, each with the capacity for a complete system, so this is a handy size. When each **UNIT** is set at 1800000 addressable blocks, exactly three fit in the classical 5400-block local mass storage for one machine, with each **UNIT** containing 1500 **PART**s. This is the default for the **DISKS** table at this time; when an application requires larger address space it will simply enlarge each element of its **DISKS** table to the required size and accept a corresponding "hit" on number of network addressable boxes. ***Note that no box should ever dynamically reduce the address space allotted to its working system to something less than it's designed to use! Doing so with data base running can crash, destroy space in the next units or on the network, et cetera.***

Large Media: Because current media, even removable ones such as Compact Flash, commonly have room for substantially more space than our systems require, we have a convention for placing multiple systems or other large hunks on the media. Experience has shown that chunk sizes for doing so ought to be designed to be easy to work with in one's head, and to be numerically distinct from other chunk sizes; violating this rule to make things fit well has proven to be a great way to make tragic mistakes. Accordingly, it is now our convention to organize large media in 3,000,000 block (25,000 **PART**) chunks. This gives us a good ten system-sized chunks on a 32MB Compact Flash, a patently useful number. What is placed in one of these chunks is not necessarily that long, but this is a good working bin size. For handy organizing within a local machine such as an AVALUE box, one can size all three **DISKS** at 30,000,000 blocks for 90 GB on the local machine and still have addressability on the network of 90 GB each in hosts 0..22 using positive block numbers and 0..46 using full unsigned block numbers.

Humungous Things: If it's ever necessary to store more than 3 GB then we simply allocate contiguous 3 GB chunks on the medium.

The large scale organization of a medium or system is presently documented in block 7 of the first system on that medium.

2.8.2 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

2.8.2.1 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.

2.8.2.1.1 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.

2.8.2.1.2 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.

FLUSH!! In addition to **FLUSH!** secures cache and tables in PCI-ATA and AHCI SATA devices for power-down which must immediately follow.

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.

2.8.2.1.3 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.

2.8.3 Mass Storage Status Codes

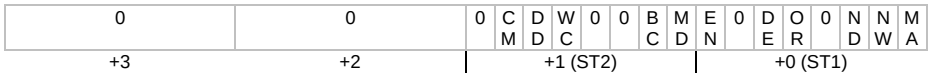
The values returned as status by mass storage primitives, or available in `DISK 4+` after an operation, are encoded in an interface specific manner to represent the maximum feasible detail in a single cell value. In addition, the system generates several specific values representing device independent problems. The device independent values are as follow:

-1 Soft write protect violation, or invalid block number.

Encoding for each supported interface is as follows:

2.8.3.1 Floppy Status Codes

The standard FDC chips return three octets of status information identified as 8-bit values ST0, ST1, and ST2. For a combination of reasons (legacy from 16 bit systems, and less interesting information in ST0), only ST1 and ST2 are returned in the low order 16 bits of the status value, encoded as follows:



CM Control Mark. Deleted address mark on normal read, normal address on read deleted.

DD Data Error. CRC error in data field.

WC Wrong Cylinder. Track address from sector ID field different from believed head position.

BC Bad Cylinder. Track address from sector ID field differs from believed head position and is FF.

MD Missing Data Address Mark. No data or deleted data address mark on the track.

EN End of Cylinder. Tried to access a sector beyond the final sector of the track. Set if TC is not issued after read or write data command.

DE Data Error. CRC error in either ID or data field of a sector.

OR Over/underrun. DMA not keeping up.

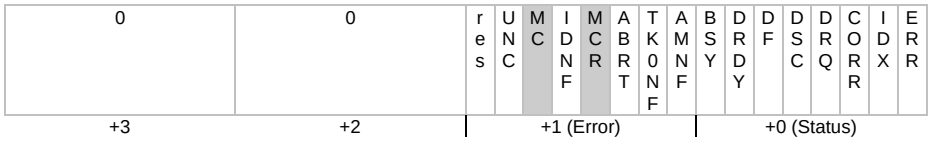
ND No Data. (a) Did not find specified sector on read or read deleted. (b) Cannot read ID without error on a read ID command. (c) Cannot find track sequence on read track command.

NW Not Writable.. WP pin became 1 while executing write data, write deleted data, or format track.

MA Missing Address Mark. (a) No ID address mark at the specified track after two index pulses. (b) no data or deleted data address mark on the track.

2.8.3.2 IDE and ATA Status Codes

The PC/AT disk controllers, such as the Western Digital WD1003-WA2 board, have over the years been transformed and refactored into the IDE and more recently ATA structures. The industry and NCITS committee T13 are actively developing ATA. In general the current hardware is still upward compatible from the original PC/AT disk controller, and the basic status information is still expressed in sixteen bits via two 8-bit registers (Status and Error), as follows (older interfaces may not produce the shaded bits):



UNC Uncorrectable data error.

MC Media Changed.

IDNF ID Not Found (sector ID field)

MCR Media Change Requested (usually at the device, usually by human action)

ABRT Aborted Command, for example invalid command code or parameter.

TK0NF Track 0 not found during recalibrate.

AMNF Data address mark not found after finding correct ID field.

BSY Device busy (not a completion status)

DRDY Device Ready.

DF Device fault.

DSC Device Seek Complete.

DRQ Data Request (not a completion status)

CORR Correctable data error.

IDX Index (vendor specific meaning)

ERR Error. See error register.

2.8.3.3 SCSI Status Codes

2.8.3.4 PCI-ATA Status Codes

See the later section on PCI-ATA support for additional information.

+3	0								
+2									Cmd Block Features register
+1	Sim-plex	Dev 1 DMA capable	Dev 0 DMA capable	0	0	INTRQ write 1 to clear!	Error	Active	Bus Master Status register
+0	BSY busy	DRDY ready	DF dev flt or strm error	def'd write error	DRQ data rqst	*align error*	*SDA sense data avail*	ERR any error bit	Cmd Block Status register
	7	6	5	4	3	2	1	0	

2.8.4 Utilities

2.8.4.1 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.

2.8.4.2 BULK Utility

This simple utility supports efficient massive transfers and comparisons between mass storage areas, devices, and classes. When an area lies on a local mass storage device (IDE/PATA, or PCI-ATA, or SCSI as the system is configured), it is fetched and stored in multi-block sequential chunks, bypassing the buffer pool, while areas on other devices such as floppies or the network are handled a block at a time. There are caveats:

- Because it can bypass the buffer pool, this utility must be used on a quiet system or, at least, on areas of mass storage that are ***absolutely*** not being actively used by other tasks.
- The initial implementation does not check or respect write protection.
- Overlapping areas will only work correctly if the destination lies at a lower block number than the source.

Otherwise these functions are provided:

BLOCKS (s d n) Copies **n** blocks from **s** to **d** .

MATCHES (s d n) Compares **n** blocks between **s** and **d** .

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

On the AVALUE box using good SSD, mSATA or Compact Flash devices, **BLOCKS** and **MATCHES** generally process a gigabyte in 10 sec; **OBLITERATE** takes less than 6 sec for the same.

2.8.4.3 TAPING Utility

This is still supported as a backup facility using SCSI tape but should only be used when compelled by available hardware or site media management practices because it is inefficient, complex, expensive and slow when compared with bootable flash media. In 5d, SCSI may be added to IDE or PCI-ATA systems to access tape drives (but not disks). Like BULK, the disk operations in TAPING have been extended to take advantage of faster multi-block transfers on all local mass storage classes.

2.9 Configuration Files

Up until release 5b, it was commonly necessary to edit source changes in various disjoint places to configure a system for a particular hardware and application environment. The good thing about this practice was that, with the exception of certain parameters that had been gathered in places such as block 2 for convenience, the parameters were declared in context of that part of the program that required them, thus providing superior documentation in all respects. There were, however, several disadvantages. First was the matter of having to maintain a mental or written checklist of places to check when setting up a system. Second was the fact that even if two systems were virtually identical, running only base software with formal configuration choices, it was basically necessary to back up the whole system source to document that configuration. Third was the inconvenience and opportunity for error involved in running an end customer application on a development or support box and environment, in which case both hardware parameters and things like network addressing had to be overridden; this in turn required editing the application source, with subsequent reconciliation problems in returning that image to customer configuration before returning it ... with careful auditing required to ensure that no mistakes were made.

Because Matrix Systems had a compelling requirement to consolidate all formal configuration choices (including those that could not be specified by a simple numeric `CONSTANT`) in one place that could be altered without providing visibility and access to the full system source, we created a mechanism for doing so while addressing as many of the problems above as practicable at the same time. The result was satisfactory and achieved all goals without unduly complicating the source, without creating any major new auditing burdens, and with a small enough penalty in compilation performance to be tolerable in light of the benefits gained. After some consideration, we decided that it was worthwhile to adopt the basic mechanism within the base system at release 5b; the parametric consolidation should benefit anyone else with an installed application base, and would simplify reconciliation of new releases with anyone using this mechanism since the system source has necessarily been altered in those places where the configuration choices take effect.

The following sections describe the architecture as implemented in the sF base. Matrix' implementation is a bit more elaborate owing to the nature of their requirements; differences between the two are noted where relevant.

2.9.1 Architectural components

The central capability of this architecture is the ability to reference a configuration parameter by name within the program source in such a manner that its value, resolved to a string of characters, may be substituted into the source as though it had been written there verbatim.

The identifiers which name these parameters occupy a name space outside the scope of the FORTH dictionary. Although it would be legal to use an identifier identical to, for example, the `CONSTANT` whose compiled value would be obtained from that identifier, we do not recommend this practice. The naming convention used within the system is that ATHENA controls the identifier name space beginning with the characters `a.` (a dot). We suggest that applications select some other similar prefix for their identifiers which will help discriminate between parameter identifiers and FORTH words.

2.9.1.1 Substitution Syntax and Usage

There are three basic supported ways to access the configuration files at compilation time: One query, and two substitution forms. These have proven sufficient so far, but others can be added if needed. They are:

`?C$ ( _ - t)` returns true if the following identifier can be resolved from the configuration files.

`C$ ( _)` is the most commonly used form. The following identifier is resolved from the configuration files and its value, as a string, is `EVALUATE`d immediately. BASE is saved and restored around the EVALUATE and BASE is guaranteed DECIMAL at the start of the EVALUATE. Aborts if the identifier cannot be resolved, or if its value is a zero length string. It may be used in a wide variety of ways; here are two examples:

```
C$ a.MAXRAM  CONSTANT MAXRAM
C$ a.STATEMENT
```

In the first example, the value of `a.MAXRAM` is interpreted and is expected to leave a number on the stack, which is used to define the value of MAXRAM. The string value of `a.MAXRAM` may be as simple as a number, or a phrase such as `HEX 400000`, or anything else including a calculation of the value. In the second example, the string value of `a.STATEMENT` is simply interpreted as though it were a line of FORTH code. It could do anything

that's possible to perform in FORTH, including building of structures, invocation of constructors, and so on. We have used it in cases such as Ethernet interface declarations to replace the entire source line previously used for that purpose, and the source line is simply moved from the program source to the configuration files.

When an identifier is used with `C$` trailing spaces in the value, if any, are eliminated with `-TRAILING` and the resulting string must be of nonzero length. This serves as a check against erroneously specifying this type of value as empty when the desire is for a default value such as zero. To genuinely specify a null value for an identifier to be substituted with `C$` use the value `null` (defined in the same block as `C$` and `S$`). `null` is a high level no op.

`S$ ( _)` is used for simple text string substitutions. It compiles its value into memory as a counted string. Unlike `C$` this function treats trailing white space (possible only in text files) as significant; it also considers null values to be valid.

2.9.1.2 Configuration File Syntax

2.9.1.3 Identifier Resolution Algorithm

2.9.1.4 System Source Changes

2.9.1.5 Default Configuration File Conventions

2.9.1.6 Defined Identifiers and Default Values

3. Basic Entitlements

3.1 Clock and Calendar

Current platforms are normally equipped with a reasonable realtime clock as well as a battery backed clock/calendar chip. We use the latter to establish date and time on boot, and the former to maintain date and time during operation.

3.1.1 Time of Day

Each time the system is booted, the clock chip is interrogated and is used to set the internal clock. Subsequently, this clock is maintained by interrupts from the 8254 timer at approximately 18.206 Hz. Our current convention is that local time is stored in the clock chip, although daylight savings management is vastly simpler and more reliable if UTC is stored there. 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`

3.1.2 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` .

3.1.3 Elapsed and Free Running Time

The system supports two methods of measuring time intervals, and a third in the case of Pentium processors and others that support the RDTSC instruction.

3.1.3.1 Elapsed Time

The 8254 counter in AT architectures is clocked at roughly 1.193166 Mhz (.838106 μS). We set the chip up to interrupt on each rollover of a full 65k count, so that the resulting interrupt frequency, as used to maintain the day clock above, is approximately 18.206 Hz (54.926 mS). For interval timing purposes, we concatenate the low order 16 bits of a free running count of interrupts with the 16 bit value in the counter chip to yield a 32 bit free running elapsed timer at full counter chip resolution of less than a microsecond. Unfortunately, because of race conditions this reading process takes on the order of 10 μS, which should be taken into consideration when making measurements. The 32-bit value rolls over once per hour.

COUNTER (– n) returns a high resolution free running time in ***counter units***.

t/US (– n n) **t/MS (– n n)** return ratios *defining counter units* per microsecond, and per millisecond, respectively.

TIMER (n) computes and displays the time elapsed since the COUNTER value given, in microseconds. The arithmetic used overflows at approximately 35 minutes.

MS (n) delays for at least the given number of milliseconds. This value is rounded down, so for guaranteed minimum values add 1.

3.1.3.2 Long Period Free Running Time

Applications frequently need to measure time intervals with moderate resolution covering periods on the order of many days, and this problem is ill suited to use of day clocks which are subject to resetting, adjustment, daylight savings, and inconvenient structure or moduli. Therefore as a compromise the system provides a 32-bit free running moderate resolution long period timer. On these platforms this timer is incremented at the 8254 rollover frequency of 18.206 Hz, so its resolution is about 55 mS and its rollover period is 2730 days.

tCLK (- a) returns the address of a cell containing a freerunning timer in ***freerunning units.***

f/MS (- n n) **f/SEC (- n n)** return ratios *defining freerunning units* per millisecond, and per second, respectively.

(dur) (n - a n) formats a time interval, in seconds, as [[x]xx**d**][xx**h**][xx**m**][xx**s**] with zero valued fields omitted. With the exception of days, which may be three digits in length, fields are fixed in length. In most cases MIN may be used to truncate cleanly. The maximum interval expressable in 32 bits of seconds is nearly 50,000 days, but this routine only formats the low order four digits of the number of days so it is limited to intervals of 27 years.

3.1.3.3 Pentium High Resolution Timing

Preliminary support is available for the high resolution free running Pentium timer, accessed by the RDTSC instruction. This timer is 64 bits wide, and is incremented at the CPU clock rate. The result is more resolution than anyone is likely to need with a rollover interval not quite geological but certainly on the same order of magnitude as recorded human history. See block 176 for the preliminary code.

3.1.4 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.

NOW (mm/dd/yy | mm/dd/yyyy) given a double number on the stack, sets the system date and updates the clock chip. Dates are interpreted as shown above for `M/D/Y`.

3.1.4.1 Clock/calendar chip management

The MC146818, archtype for the PC/AT model, stores only two digits of year and assumes that year 00 is a leap year. The chip actually rolls over past BCD zero but this is a poor feature to assume, and empirically DOS does not set it to values beyond 99 when given years exceeding 1999. To avoid compatibility and interoperability problems, the following conventions are observed:

- When setting the clock, we store only the low order two digits of the year number.
- When reading the clock, any year in the range [90..99] is taken to mean [1990..1999] while any year in the range [00..89] is taken to mean [2000..2089].

3.2 Capsules

3.3 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.

3.3.1 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.

3.3.2 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 BMC 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.

3.3.3 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.

3.3.3.1 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 2.3.2.3
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.

3.3.3.1.1 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.

3.3.3.1.2 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

3.3.3.2 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.

3.3.3.3 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.

3.3.3.4 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.

3.3.4 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.

3.3.5 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 _name _pers _idle - dv') makes a named P2LDV task given the args for d_term . Leaves its device selected with own task device on stack! If making a net printer, the next word interpreted after d_TERMINAL will normally be a printer configuration function such as PR-PDFMIME.

3.3.6 Phase 3 Logical Devices

These will only be implemented after ATHENA internal and BMC bases are in order, 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.
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. Consider layering logical devices as implied by PDF over MIME over RFC822 mail over TCP.

3.4 Disk Directory

3.5 Fixed Point Arithmetic

3.6 Floating Point Arithmetic

3.7 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.

3.7.1 Flat Files

3.7.2 Ordered Indices

3.7.3 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.

3.7.3.1 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 file 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 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 the structure is *not* literally that of the "B-tree" we do not call it that.

3.7.3.2 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.

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. See also the Usage Recommendations below.

3.7.3.2.1 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.

3.7.3.2.2 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.*

3.7.3.2.3 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.

3.7.3.2.4 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.)

3.7.3.2.5 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.

3.7.3.2.6 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.

3.7.3.2.7 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.

3.7.3.2.8 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.

3.7.3.2.9 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.

3.7.3.2.10 Utilities

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

3.7.3.2.10.1 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.

3.7.3.2.10.2 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.

3.7.3.2.10.3 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.

3.7.3.3 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.

3.7.3.4 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.

4. Extensions and Utilities

4.1 Resident Programmer Interface

4.1.1 Editor

The resident editor is, basically, the venerable character editor which has been standard on polyFORTH systems as well as all ATHENA systems for the past fifteen years or so. 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:

4.1.1.1 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 is still being evaluated since it does not mesh perfectly with other existing mechanism. In particular, there are cases in which it is awkward to arrange for the desired block to be the "other" block. To get around these cases, a temporary word is provided:

`SY (b n)` explicitly makes ***b*** the "other" block and ***n*** be the current character position [0..1023] in block ***b***.

4.1.2 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.

`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.

4.1.3 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.

4.2 Resident Functions

4.2.1 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.

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.

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.

4.2.2 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
```

4.3 Elective Functions

TCP/IP Package

Cryptography

4.4 Utilities

4.4.1 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.

4.4.2 CPU Identification

Tools for identifying Intel compatible processors are presently provided in blocks 174 and 175. The code must be updated as new information is released by Intel, and is subject to being moved. This is not considered a problem since application code should in general not depend upon this information; the only dependency within the system occurs within the floating point support, which must take actions required of 486 and later processors but not supported on 386.

4.4.3 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 windows

4.5 Debugging Tools

4.5.1 386 Debug Registers

Blocks 489 and 500 contain optional code for managing the onchip "debug registers" of the 386 and up architecture. Up to four independent locations in memory may be monitored for accesses in selected widths (byte, half, or cell) and type of access (read/write, write, or execute). Detection is achieved by trapping. See the code and its shadows for syntax and usage. You will probably want to adjust trap behavior to suit the conditions under which you are debugging at the time.

If you envision ever using this feature, we recommend that you load it in block 9. The rationale for this recommendation is that you are most likely to want it when an unknown party is storing inappropriately; since store errors are frequently address sensitive, adding this code to the load sequence while debugging can change memory allocation and defeat efforts to reproduce the problem.

4.5.2 Panic Dump

You may optionally configure any selection of traps to halt the system and attempt a panic dump of all memory to disk. This capability is only supported for SCSI disk with “smart” host adaptors, or for IDE/ATA disks that support Logical Block Addressing (LBA). It is specifically not supported for the NCR/Symbios/LSI Logic host adaptors because they are software intensive and depend on a healthy host operating system. To arm the panic dump do the following:

1. Ensure that `.INDEX` shows a valid allocation named `DumpImage` whose **size must be at least `MAXRAM 1024 /` blocks.** By default this is defined in block 485 but your system may be different. Double check that the size is adequate; if it is less than required, the dump may overwrite the following area.

2. Examine block 1515. Comment any traps that you ***do not*** want to produce a panic dump. By default, all traps cause dumps; if you actually want to dump on a zero divide, un-comment `00 TRP`.

3. Carefully load block 1515. Quite often, problems whose diagnosis requires this extreme measure are address sensitive, for example. The best practice is to load the entire system and application, and when it is all resident and initialized, load block 1515 into the dictionary of a task that is not involved in the failing scenario. If `OPERATOR` is not involved, this might be conveniently done at the end of block 9 or 534 or equivalent.

4. Now that the trap has been laid, wait for the problem to manifest itself. If the failure mode is to trap, simply wait for it to occur. If instead it’s a hung system with interrupts still working, such as can happen when a task is in an infinite loop that does not include `PAUSE`, you can go to the `OPERATOR` keyboard after the hang has occurred and use the key sequence `ALT+ESC` to force a trap from within the keyboard interrupt. If you are unable to force a dump by any of these means, the last resort is to find some way to produce an NMI.

5. When the dump starts, the top half of the `OPERATOR` screen is changed to be highlighted characters; the character is ASCII `@` plus the trap number, so `@` would mean zero divide. The attribute is changed after a SCSI command has been started, or after an ATA dump has been completed. In either case, due to the possibility of caching, wait for the disk light to go out before rebooting the system.

6. It is not unusual in these extreme cases that the underlying problem is a wild store onto code. If you are able to determine that this is what’s happening, the next step might be to set up the debug registers to monitor the location in question and then add the panic dump code to capture the erring store in full context.

Subsequently an accompanying utility may be used to interactively inspect this dump image with minimal dependency on the integrity of the saved memory image. The code is in 1518ff. Dumps must be examined on the same systems as generated them, since address resolution and searching are based on the dictionary and memory content of the examining system.

To read a panic dump, load block 1518. As long as you are careful with your vocabulary and capsule context, things like `WHAT` and `.SLEEPER` work to a degree even if dumped memory is severely clobbered; likewise if you are following links that may

have been clobbered you can get some mileage by following them in live memory using `@` and only examining the contents of dumped memory with `P@`. Naturally, if the dumped environment includes overlay loading your task will be more difficult.

`CSTACK` summarizes what caused the dump. `P@` and `PC@` operate on the dumped memory, as does `PDUMP`. To see actual stacks for any task as of the dump you will need to examine their user areas and employ `.stk` manually because `.SLEEPER` is neither smart nor brave enough to obtain active stack boundaries from the dump.

In February, 2017, we learned that the reason some machines were hanging during a panic dump, particularly when using the ATA version of dump writing in a SCSI system, is that some combination of chipset and BIOS were preventing the interrupt bits in PIC register from being seen when polled. The new version of this routine works on such boxes by polling the BSY bit in x1F7.

5. Construction

5.1 Target Compilation

Compile the nucleus conventionally: COMPILER LOAD ISA 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.

5.2 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.

5.3 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.

5.4 Making a boot floppy

The following procedure does this:

```
0 DRIVE  DISKING LOAD
0 OFFSET !  0 FLEX 1200 BLOCKS  SHADOWS FLEX 1200 + 240 BLOCKS
      0 FLEX 1200 MATCHES  SHADOWS FLEX 1200 + 240 MATCHES
-1 DRIVE  (now you are looking at the floppy as seen at boot time)
```

As the system has grown to encompass more hardware, it is no longer practical to house a full system with all networking capability on a single 1200x240 floppy. Until 5c it was necessary to selectively comment things in order to make a full capability bootable floppy. As of 5c this is much simpler. Having written the basic floppy as above and climbed up onto it with -1 DRIVE you should attend to the following:

- Ensure appropriate nucleus in boot area. Unless the nucleus you just copied is correct for the system on which the floppy is to boot, you may need to re-target it. From -1 DRIVE you may configure 182, retarget and INSTALL as described above. At -1 DRIVE the INSTALL will write into the floppy's boot area.
- Configure network. If the boot floppy is to come up and be on the network without assistance from someone on site who can edit source, this is the time to work on block 559 to configure it properly for the target system.
- Configure block 2. You might want to set a reasonable or identifiable Zname. The constant AVALUE should be zero always because, at 5c, we can't do a 9 LOAD from floppy on an AVALUE box anyway. MAXRAM should be small enough to avoid the diagnostic on the target system. Finally, set ?FLOP to 1. This flag is used during the 9 LOAD (and also

ETHER LOAD) to leave out activities that will interfere with doing a 9 LOAD from floppy (such as changing the origin of floppy, see blocks 3 and 6) as well as loading things like PDF printing that don't fit onto the floppy in the first place.

That's it. You might want to test the boot floppy before sending it to a site, especially if it's to be used by someone who can't edit the sourde.

6. Miscellaneous

6.1 New Facilities Under Evaluation

The following facilities are provided to solve specific problems. Their use is for evaluation. Applications that need these functions are welcome to use them, but if you establish any operational dependencies you should monitor their source carefully for changes in coming releases.

6.1.1 SCSI Media Size Interrogation

We have encountered optical disk drives whose sector size varies with the installed media. Although we cannot at present adapt to sectors larger than 1k, one such drive gives us either 512 or 1024 depending on how the medium was formatted. Thus, every time there is a media change, the geometry and size of the media may also change.

Since it's necessary to clear the Unit Attention status before using such a drive in any case, the following function has been added. It is temporarily placed in block 509. Later, if it proves to be as useful as it might be, we may refactor matters so that its functionality is more deeply embedded. For right now, before beginning use of a disk drive that *may* have a pending Unit Attention, or whose media *may* have changed, the following definition may be used:

`!MEDIA ( unit - n sts)` interrogates the device associated with the given `DRIVES` table index for its sector size and drive size. Returns the actual drive size, in blocks, under status (nonzero if error). If there were no errors, the `DRIVES` table is updated to reflect sector size. Drive size is *not* updated because this would lead to functionality which would not be compatible with the next generation of `DRIVES` table. Sector size must be a power of two and in [128..1024]. `PAD` is used for scratch. Since this is intended to solve problems with media changes on removable optical drives, unit attention is cleared too.

`!SIZE ( n mod unit)` sets unit size to given number of blocks, rounded up to given modulus. In the next generation of the `DRIVES` table, the semantics of this function will need to be changed since there will be separate values indicating the address space allotted for each drive and accessible size of the drive. The two values will be identical only in cases where the drives are intentionally to be concatenated as a single large drive. This will in turn create more options than may be discussed now.

7. ISA Hardware Support

Much to be written here.

7.1 ATA Disk Support

The classical PC WD1003-WA2 register set has as of ATA-8 morphed to the following, with origin at 1F0 for primary and 170 for secondary PATA interfaces:

		7	6	5	4	3	2	1	0	
Data	+0	Function Code								
Features	+1 wr	Coded value								Ancient write precomp
Error	+1 rd	ICRC	UNC	*0*	IDNF	*0*	*ABO*	EOM ILRER	CCTO ILI MED APrrr INCS	Ancient different bits
Sec Count	+2									
LBA Low	+3									Old Sector#
LBA Mid	+4									Old Cyl# Low
LBA High	+5									Old Cyl# High
Device	+6		LBA		DRV	Lba 27:24				Old head#
Command	+7 wr	Command Code (written last)								
Status	+7 rd	BSY busy	DRDY ready	DF SE fault, stream	DWE SERV multi use	DRQ data rqst	*CRD* *correct* *ed* *data*	*IDX* *index*	ERR any error bit	Ancient different bits
Device Ctl	3F6 wr					HS3	RST	-IEN	0	376 for secondary
Alt Status	3F6 rd									... not used.

Basic support in our system has since 1987 used CHS mode for addressing and has used PIO for transfers, with 16-bit INS and OUTS instructions.

8. Microchannel Hardware Support

There are significant architectural differences in the basic PC model between ISA and Microchannel, requiring a different version of the system. sFxxx/MC denotes Microchannel support. The most significant differences result from two factors.

The first factor is that Microchannel interrupts are defined to be level rather than edge triggered and are in fact inherently sharable. This requires at minimum attention to all interrupt code. Further, basic facilities such as the interrupting clock require manual reset of the interrupt stimulus since the 8254 has no direct reset function on chip and the onchip stimulus is not suitable for level interrupt support without an external resettable latch between the chip and the PIC. Other minor differences pertain to such things as the disk activity LED which by IBM conventions is completely software controlled. Other differences have emerged since some vendors, such as NCR, have cloned subsets of for example the PIC's such that auto EOI is not supported on *either* the master or slave.

The second factor is that peripheral vendors have not enthusiastically provided MC boards that are architecturally equivalent to their ISA offerings. Indeed, vendors do not seem to have been very enthusiastic about supporting MC at all. As a result, basic facilities such as SCSI Host Adaptors become problematic and have required drivers specially written for the MC platforms. The relative paucity of MC platforms and interfaces has precluded much of the extensive interoperability testing we've been able to do with ISA derivatives, and as a result our MC support is much more a custom thing than is the ISA system.

Barring these considerations, however, from an application perspective the system model is equivalent to the ISA platform and in general non hardware dependent applications will port between these platforms with little or no difficulty.

9. ISA/VLB Hardware Support

Thus far we have needed no special support for VLB equipment. Once configured, all hardware resources function as they do in ISA bus systems. The sole significant exception is that VLB can address more than 16Mb and on machines with more than 16Mb of memory some will appear above the 16Mb point, skipping over BIOS PROM area in some size region immediately below the 16Mb point. The system does not currently provide any particular support for skipping the PROM or for sizing the extended memory.

10. ISA/EISA Hardware Support

Thus far we have needed no special support for EISA equipment. Once configured, all hardware resources function as they do in ISA bus systems. The sole significant exception is that EISA can address more than 16Mb and on machines with more than 16Mb of memory some will appear above the 16Mb point, skipping over BIOS PROM area in some size region immediately below the 16Mb point. The system does not currently provide any particular support for skipping the PROM or for sizing the extended memory.

11. ISA/PCMCIA Hardware Support

Unlike MC, VLB, and EISA, our system is required to take an active role in powering and configuring PCMCIA cards. To do this we have to know something about the platform and about the cards in question.

For ISA compatible platforms using i82365 compatible PCMCIA host interfaces, we support card and socket control sufficient to initialize, disable, and power down all cards; and to power up and initialize selected cards. The mechanism is currently in block PCM-CFG, intended to be resident; loading the code initializes the PCMCIA subsystem for a maximum of eight card slots (two 82365's). It is used in the TCP/IP package which supports selected PCMCIA Ethernet interfaces.

PCMCIA documentation forthcoming. For the present, all socket configuration is handled by the code which requires it. The only manual step required is to load **`PCM-CFG`** in block 9.

12. ISA/PCI Hardware Support

As has proven to be the case with MC, VLB, and EISA, it's in general practical to use the slot configuration services provided by the BIOS for addressing and configuring of the more standard PC interfaces such as VGA, keyboard, floppy, and IDE disk. As a result, for basic applications using only ISA compatible peripherals and ISA boards the system requires no adjustment.

However, the BIOS places some things in unfortunate positions and so we provide some configuration mechanism for PCI cards. The support is based on the PCI Local Bus Specification, Revision 2.0, published by the PCI SIG. It recommends a well documented and straightforward "Configuration Mechanism 1" for PC-AT compatible systems, and this is what we support.

The code currently located in block PCI-CFG is intended for resident use and contains mechanism for displaying all PCI devices and for configuring individual cards. The following functions are generically provided:

`>pBUS (n8)` Selects bus ***n8*** [0..255] for subsequent operations, not altering other addressing fields. Enables access to configuration space.

`>pDEV (n5)` Selects device ***n5*** [0..31] for subsequent operations, not altering other addressing fields.

`>pFUN (n3)` Selects function ***n3*** [0..7] for subsequent operations, not altering other addressing fields.

`@pci (ra - n)` and **`!pci (n ra)`** Fetch and store of full cell values at the configuration space address ***ra*** [0..252]. Configuration space is addressed by byte but must be accessed with full cell operations on full cell boundaries. The standard prescribes the following assignments for the first 64 bytes (16 registers) of configuration space:

	+3	+2	+1	+0	
+0	Device ID (Vendor Assigned)		Vendor ID (FFFF if not present)		
+4	Status ***(note)***		Command ***(note)***		
+8	Class	Subclass	Register Interface	Rev (vendor asg)	
+0C	BIST Selftest ctl/status	Hdr Type msb=1 if multifunc	Latency Timer for bus masters	Cache line size for bus masters	
+10	Base register 0 ***(note)***				
+14	Base 1				
+18	Base 2				
+1C	Base 3				
+20	Base 4				
+24	Base 5				
+28	Reserved				
+2C	Subsystem ID		Subsystem Vendor ID		PCI 2.1
+30	Expansion ROM Base Address ***(note)***				
+34	Reserved				
+38	Reserved				
+3C	Max latency r/o	Min Grant r/o	Int pin (signif???)	IRQ #; 255 disables	

Several of the above fields warrant special attention:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Status	par err	sig syser	mas abort	tgt abort	sig tgtab	devsel 0fast 1med 2slow		data par	fast b-b							
Command							fast b-b	enab serr#	enab step	enab par	vga snoo	enab winv	enab spcy	enab mstr	enab mem	enab i/o

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Base, memory	Addr																Prefetch, Low 1Mb
Base, mem64	Addr																Prefetch
	High order address																
Base, I/O	Addr														0	1	
Exp ROM base	Addr												Res		Reserved	E	Enable
	+3				+2				+1						+0		

Status bits indicate events or capabilities. Events are cleared by writing one into same position.

Base addresses occur in any order. Size (and boundary) inferrable by writing all ones to address field; bits to right of address modulus will read back as zero.

.PCIS scans all 65k possible PCI card addresses (256 buses, 32 devices, 8 functions) and produces a one line display for each agent found to be present. An example display from an AST Premmia MX with ATI Graphics Pro Turbo graphics board and SMC 8432BT Ethernet board, in that order, is as follows:

```
.PCIS
Bus=0 Dev=5 Fun=0 Class=0 0 0 Rev=2 Vendor=8086 Device=486
Bus=0 Dev=9 Fun=0 Class=3 0 0 Rev=1 Vendor=1002 Device=4758
Bus=0 Dev=A Fun=0 Class=2 0 0 Rev=23 Vendor=1011 Device=2
```

The above machine was current during Winter of 1994/5. Three years later in the deployment of PCI, we find considerably more as is illustrated by the following, from a Micron Millenia XKU Pentium II/300 machine sporting 128 Mb RAM, an "AGP" graphic board, an SMC Ethernet interface, and three NCR/Symbios SCSI Host Adaptors:

```
.PCIS
Bus=0 Dev=0 Fun=0 Class=6 0 0 Rev=3 Vendor=8086 Device=7180
Bus=0 Dev=1 Fun=0 Class=6 4 0 Rev=3 Vendor=8086 Device=7181
Bus=0 Dev=7 Fun=0 Class=6 1 0 Rev=1 Vendor=8086 Device=7110
Bus=0 Dev=7 Fun=1 Class=1 1 80 Rev=1 Vendor=8086 Device=7111
Bus=0 Dev=7 Fun=2 Class=C 3 0 Rev=1 Vendor=8086 Device=7112
Bus=0 Dev=7 Fun=3 Class=6 80 0 Rev=1 Vendor=8086 Device=7113
Bus=0 Dev=D Fun=0 Class=2 0 0 Rev=11 Vendor=1011 Device=14
Bus=0 Dev=E Fun=0 Class=1 0 0 Rev=4 Vendor=1000 Device=4
Bus=0 Dev=F Fun=0 Class=1 0 0 Rev=12 Vendor=1000 Device=1
Bus=0 Dev=10 Fun=0 Class=1 0 0 Rev=1 Vendor=1000 Device=C
Bus=1 Dev=0 Fun=0 Class=3 0 0 Rev=10 Vendor=12D2 Device=18 ok
```

.PCI (b d f) dumps the full 256 bytes (64 cells) of configuration space on the given bus/device/function. The standard prescribes that nonexistent devices will return all 1's in at least the vendor ID field. Only the first 64 bytes (16 cells) are defined by the standard. Examples of such dumps using the same example as above are as follow:

The first device, which is an Intel PCI chipset i82425/426, shows configuration space below. Status shows parity, master, and target aborts, and medium DEVSEL assertion time. Command indicates SERR# enabled, along with master, memory, and I/O. The configuration registers starting at 40, plus a few I/O registers, control all motherboard "glue" functions. See 82420EX PCIset manual..

```
0 5 0 .PCI
   0   4868086  B2000107         2         0
  10         0         0         0         0
  20         0         0         0         0
  30         0         0         0         0
  40        40         0      1615    330100
  50   D1B8003  AA000000  88991000  88888888
  60  10100808   9800010      1000         0
  70         6         0         0         0
  80         0         0         0         0
  90         0         0         0         0
  A0    C00009   C001002         F         0
  B0         0         0         0         0
  C0         0         0         0         0
  D0         0         0         0         0
  E0         0         0         0         0
  F0         0         0         0         0
```

The second device, the ATI Graphics Pro Turbo, is shown below. Its status indicates medium DEVSEL assertion. The command indicates address/data stepping, memory and I/O enabled. A memory window is defined at 78000000 which is consistent with that board's linear frame buffer as claimed by DOS. No I/O window is explicitly defined, a somewhat ominous observation. ROM is shown as disabled at C0000, not likely true.

```
0 9 0 .PCI
   0  47581002   2000083   3000001         0
  10  78000000         0         0         0
  20         0         0         0         0
```

30	C0000	0	0	0
40	0	0	0	0
50	0	0	0	0
60	0	0	0	0
70	0	0	0	0
80	0	0	0	0
90	0	0	0	0
A0	0	0	0	0
B0	0	0	0	0
C0	0	0	0	0
D0	0	0	0	0
E0	0	0	0	0
F0	0	0	0	0

An old SMC 8432 Ethernet interface yields the following. This is much more interesting since that device is capable of bus mastering. Status shows medium DEVSEL and also fast back to back transfer capability. Command shows SERR# enabled along with mastering, memory and I/O.The latency timer is shown as 16 bus clocks; I/O at FE80, memory at FEFFFC00, and no PROM. The interrupt pin register shows INTA# in use, and the interrupt line register shows IRQ9.

```
0 0A 0 .PCI
   0     21011   2800107   2000023      1000
  10      FE81  FEFFFC00         0         0
  20         0         0         0         0
  30         0         0         0       109
  40      FC00         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70         0         0         0         0
  80         0         0         0         0
  90         0         0         0         0
  A0         0         0         0         0
  B0         0         0         0         0
  C0         0         0         0         0
  D0         0         0         0         0
  E0         0         0         0         0
  F0         0         0         0         0
```

PCI uses an interesting method of indicating the amount of address space required in each base register. To find this out, store all 1's into the address portion of the base register, then read it back. According to the standard, the bits which come back as 1's are those which may be altered, and the lowest order 1 will indicate both addressing granularity and size of allocation. The card may or may not actually decode everything within this space.

12.1 "Legacy" ISA Hardware Compatibility

As consumer hardware has migrated, PC hardware platforms have generally maintained upward compatibility ... sometimes at great cost. This section documents specific things we have had to do in those relatively few cases when compatibility is not the default.

12.1.1 USB Keyboards

So far, all the PC platforms we've worked with have emulated the AT keyboard controller and keyboard protocols by running system management mode (invisible) code that traps the register accesses and manages USB host controllers. This has been transparent except that some visible high memory is used for the USB host data structures, so watch out with MAXRAM.

The only way we can use the USB hubs ourselves disables all of this SMM code; so before we can talk with USB devices in general we have to support USB keyboard. This isn't done yet.

12.1.2 SATA Disks

So far most PC platforms have had BIOS options to enable legacy or IDE mode; when this is done the chipsets have been configured, usually, to actually enable PATA emulation with no effort on our part.

Recently we've encountered BIOS that set up the ICH7 chipsets to emulate IDE host interface on the SATA host controller, but have not enabled the emulation. Matrix found two boxes like this in 2013 and 2014. Consequently a new defnition **!PATA** has been added to the nucleus and has been invoked in **HOME** . This definition lies in block 263, and its strategy is to check for the first instance of PCI class/subclass 0101 to see if it is of a type requiring this treatment, and to apply it if so. As new exceptional devices are discovered they will be added to this definition.

In at least one case, the IDE emulation seems to require that #hd be set to 16 not 15 for correct operation; the problem has been reported but not the failure mode.

Because we now support PCI ATA host controllers very well, we recommend using PCI-ATA instead of IDE except in those cases where the chipset or the BIOS make it impractical. An example of the latter is the BIOS in Dell SC430 boxes which only assign I/O space to the first of the five mapping registers in their PATA controllers.

12.1.3 Deprecation of Floppy Disk Controllers

This has required extraordinary measures to bring new systems up. See the section on Bootstrapping much earlier to learn about the URAM boot procedure and nucleus support for RAM disk. The method has been used successfully with both USB floppy disk drives and USB flash "disk" devices. It will potentially work with any BIOS bootable device other than optical disk. With the support of PCI-ATA we've also made it possible to commission a new box using a bootable, and fully read/write accessible, Compact Flash card.

12.1.4 Future Problems

Eventually these legacy emulation features may wither away. Likewise the UEFI boot protocol may eventually supplant BIOS boot. Any of these things will require significant work.

12.2 ATI Graphics Pro Turbo (Mach64, PCI)

Support is available, but not in the base, for adding the above ATI board to an existing system that already has a stock VGA, preferably onboard or ISA type, as a completely independent display device. Notes obtained during this exercise follow.

Using Telnet with only the ATI board in a freshly booted system, we see the PCI bus configuration space as shown in the above examples. We find VGA BIOS at [C0000..C8000[, VGA registers in [3C0..3E0[, VGA text aperture [B8000..C0000[, and the ATI specific registers at 2EC with high I/O address lines selecting registers. The register values as found are:

```
.ATI
          x2EC      x6EC      xAEC      xEEC
 2EC    4F0069    280454   1DF01F3    2301E0
12EC         8  20000000        21     20200
22EC         0         0         0  20800824
32EC  202222A0     80088        AA     A0002
42EC  80008000  21000000       108  990E20F1
52EC       1F3     10000     10000  40FF3F40
62EC     52000         8      1E00   10000D7
72EC  3FFF8BEF         1         0    4F0069
```

Have so far been unable to read the extended DAC control registers using 62EC:0-1. This may prove to be a serious problem. Found in their listing that the 68860 is type 5 DAC and that by setting extended DAC address lines in a register indexed through 1CE/1CF (undocumented) the extended registers *are* visible through accelerator register space:

```
A0 1CE OUTPUT 8 1CF OUTPUT  .DAC  // 40 3F FF 40 ok
A0 1CE OUTPUT 28 1CF OUTPUT .DAC  //  2  0  0  2 ok
A0 1CE OUTPUT 48 1CF OUTPUT .DAC  //  2 80 1D 80 ok
A0 1CE OUTPUT 68 1CF OUTPUT .DAC  // ED  F  0 E0 ok
```

It seems that the accelerator mode CRTC is set for something like 640x480 noninterlaced at a reasonable refresh rate with 8-bit pixels.

Setting 80000 bit in 6AEC (81E00), the monitor goes to power save mode (blanked). ATI registers are still visible. The BIOS is still visible at C0000 (although it may be cached), the VGA memory aperture has vanished, the VGA registers have vanished, and the undocumented registers at 1CE and 1CF have vanished. Manipulation of config register 30 on the card can make garbage appear elsewhere than C0000, but seems unable to disable the PROM at C0000.

Enabling the memory aperture in 6AEC for 8Mb (81E02) now shows data at 78000000 and the registers in fact appear at 787FFC00 as published. Further, when addressing these registers through the aperture with VGA disabled, the DAC_EXT_SEL field (low two bits) of DAC_CNTL register do indeed control selection of DAC registers. It would seem that only one source for these bits is enabled at a time.

Setting extended mode with bit 24 of CRTC_GEN_CNTL (1020200) we have no video but upon setting the CRTC enable with 3020200 in the same register we are now up and running, with stable crud on the screen.

The above is intended to give you some insight into the way things appear on a PCI bus.

12.3 Digi ClassicBoard 4/7 PCI

These boards sport one or two quad UARTs with the same octopus style cable as the old Digi PC/8 used, although the insertion force seems to have been improved somewhat. UARTs are mapped into PCI I/O space, consecutively, with status register following the UARTs. The status register is bit mapped rather than encoded. The UART clock has been quadrupled requiring us to decode it. Otherwise the standard I/O strategy works fine.

12.3.1 PCI Configuration Space

On a typical machine we find the following presented by the PLX PCI-9052 chip:

```
Bus=0 Dev=A Fun=0 Class=7 80 0 Rev=2 Vendor=114F Device=28

HEX 0 0A 0 .PCI
   0    28114F   2800003   7800002         8
  10  E4001000      6301      6401         0
  20         0         0         0         0
  30         0         0         0       10A
```

The first two registers map the PLX glue chip's 84 bytes of Local Configuration Registers into 128 bytes of memory and I/O space respectively. The third maps quad UARTs 0 and 1, if present, into 32 bytes of consecutive I/O space each; followed by one byte of Interrupt Status Register at offset 64. The next two, uninitialized above, would define memory mapped expansion ROM and memory mapped UART and ISR registers. Interrupt 10 has been assigned to this board. Note that the device ID code indicates the number of UARTs present (28 means 4 ports, 29 means 8 ports) and that Digi uses a different subclass than GTEK.

12.3.2 PLX Chip Internal Registers

These registers are fully initialized from onboard serial EEPROM and in the unit first evaluated the values are as follow:

```
HEX E4001000 60 DUMP
          0          0    FFFF800    FFFFF01  < E4001000 >
     400000     100000          1          0  < E4001010 >
       38C0   540100A8     200000     800000  < E4001020 >
         21          0          0          0  < E4001030 >
         24          0         43     100201  < E4001040 >
          0          0          0   101A0140  < E4001050 > ok
```

It isn't immediately obvious why the Expansion ROM address space is not allocated by PCI BIOS since this appears to enable that address space for 2k. The above seems OK for operational use except the register at offset `4C` which controls the polarity and enabling of interrupts. It must be set to `53` if there is one UART and `5B` if there are two.

12.3.3 UART and Interrupt Status Register

Here is the I/O area on a 4 port board:

```
HEX 6400 44 .IO
     6400     0  0  1  0  0 60  0 FF    0  0  1  0  0 60  0 FF
     6410     0  0  1  0  0 60  0 FF    0  0  1  0  0 60  0 FF
     6420    20 24 24 24 24 28 28 28   28 2C 2C 2C 2C 30 30 30
     6430    30 34 34 34 34 38 38 38   38 3C 3C 3C 3C 40 40 40
     6440     0  0  0  0 ok
```

12.3.4 ST16C654 Quad UARTs

This UART is upward compatible with the 16450 and it is unknown whether the 16450 race is faithfully reproduced. If really needed, the chip can go to 1.5 million baud, has 64 byte transmit and receive FIFOs that may be enabled, and can be instructed

to handle both hardware and Xon/Xoff flow control internally. In addition, each port is configurable for IrDA optical interface encoding. Relevant to this board, however, is that Digi elected to present the UARTs with a 4x clock unconditionally. The result is that while we can go to 460800 baud, we have to scale the baud rate divisor differently than we do on all other boards.

12.4 GTEK PCI Cyclone 16/32

This asynchronous interface connects 16 or 32 RS-232 devices using a single PCI slot and external box with standard PC DB9 male DTE connectors for ports labeled 1-relatively. The PCI card contains a PLX PCI-9050 glue chip, miscellaneous logic, and a -12v supply to meet the needs of the RS-232 drivers without overtaxing the typical inadequate -12v supply in PC's. This card is connected via a cable using DB25 connectors to an external box which supports the first 16 channels. A second external box is connected with headers and brackets to the first when a 32 ports are desired. The external box contains its own +12 to +5v regulator, four Exar ST16C654 quad UARTs, and line drivers / receivers. All parts are socketed, and according to GTEK engineer Dave Higgdon the transceivers can be replaced with very robust MAXIM 1488/89 ECPD chips for use in environments likely to blow them out.

12.4.1 PCI Configuration Space

On a typical machine we find the following presented by the slave only PCI-9050 chip:

```
.PCIS
Bus=0 Dev=A Fun=0 Class=7 0 2 Rev=1 Vendor=10B5 Device=C001 ok
HEX 0 0A 0 .PCI
   0  C00110B5   2800003   7000201         8
  10  E4001000      6301      6401      6501
  20         0         0         0  C00110B5
  30         0         0         0       10A
rest zero.
```

The first two address registers map the PLX glue chip's 84 bytes of Local Configuration Registers into 128 bytes of memory and I/O space, respectively. The third and fourth, as the chip is initialized below, map the control register and UARTs respectively.

12.4.2 PLX Chip Internal Registers

These registers are fully initialized from the onboard serial EEPROM and in the unit first evaluated the values are as follow:

```
E4001000 60 DUMP
          0          0    FFFFF01    FFFFFF9  < E4001000 >
          0    3000001    3000101          0  < E4001010 >
   FC3FFFC0   54387C00          0          0  < E4001020 >
    3000109          0          0     800000  < E4001030 >
         5F          0          0    3000001  < E4001040 >
          0          0          0   18780B46  < E4001050 > ok
```

The third address register defines 256 bytes of I/O space occupied by 32 consecutive instances of the 8 register space used by each UART element, and the fourth register defines the 8-bit interrupt status register. The internal registers above define the functions of the glue chip, whose highlights are as follow, with [+n] referring to a relative register address in hex:

[+0] I/O space for third window (ISR), decode 8 bytes. Reason unknown, could have been 4 bytes.

[+4] I/O space for fourth window (UART's), decode 256 bytes (is 32x8).

[+14] Remap third window (ISR) to 300010x.

[+18] Remap fourth window (UART's) to 30000xx.

[+28] ISR Wait states NRAD 16, NRDD 3, NXDA 3, NWAD 16, NWDD 3; bus 8, read & write strobe delays 1, write cycle hold .

[+30] UART Wait states NRAD 31, NRDD 3, NXDA 3, NWAD 31, NWDD 3, bus 8, strobe delays & write cyc hold 3.

[+30] Fifth window bus 32 for unknown reasons.

[+3C] Chip select 0 (ISR) on local 3000100..7

[+40] Chip select 1 (UART) on local 3000000..3FFFFFF which basically makes that chip select meaningless and no doubt forces him to regenerate it externally if he indeed needs it at all.

[+50] USER0 out high. USER1 in currently low. USER2, 3 are CS2_ and CS3_. PCI reads interlock with TRDY#. No flush of pending reads because no read FIFO operations. Writes interlock with TRDY# when FIFO is full. Retry interval is 15 but that part of the manual is not self consistent so perhaps it is not interlocking after all.

UART register access timing was measured by doing a large number of reads in a coded BEGIN v IN LOOP sequence on a 6x86/200 PCI box, and were compared with the same operation on the COM1 port in that box. The GTEK register read time was 1.50 uSec while COM1 measured 1.36 uSec. Considering that the glue chip is configured for its maximum values in all time delays and strobe widths this is not too surprising.

12.4.3 Interrupt Status Register

According to Dave Higgdon at GTEK, the ISR is encoded the same as is the corresponding Digiboard register: Namely, it is either hex FF or is the zero relative number of the interrupting UART. Retriggering is not a problem because the PCI interrupt lines are by convention level triggered. We are promised there are no race conditions for its value...

12.4.4 ST16C654 Quad UARTs

This UART is upward compatible with the 16450 and it is unknown whether the 16450 race is faithfully reproduced. If really needed, the chip can go to 1.5 million baud, has 64 byte transmit and receive FIFOs that may be enabled, and can be instructed to handle both hardware and Xon/Xoff flow control internally. In addition, each port is configurable for IrDA optical interface encoding.

12.5 Digi NEO Universal PCI (4 or 8 Ports)

These boards use the EXAR XR17D158 PCI Octal UART. The octopus cable has been refined with a compact and more robust connector similar to that used for LVDS SCSI. UARTs are mapped into PCI Memory space only, 512 bytes of address space per UART, with device configuration registers at hex 80 relative to the start of the board. Interrupt and clock management are unique. The memory mapping requires, unfortunately, a parallel rewrite of all serial interrupt code to support this chip since it refuses to I/O map the UARTs.

12.5.1 Level of Support

To accommodate the EXAR chip we have had to enlarge the port indexed characteristic tables to full cell width with a new convention (8 bit means memory mapped) in uTYPE , to add new routines !muarts and basic terminal interrupt code <MTUBE> , and to create extended definitions of baud BAUD uart UART and +SERIAL so that these will work with either I/O or memory mapped UARTs. V>ADR has been extended to return clean zero in low half for memory mapped UARTs. The following new functions are also added:

~uA (2-2) is a subroutine that replaces a `V>ADR` form in register 2 with full UART base address in either memory or I/O space as appropriate.

uA (n:0 – a) does the same thing for high level code.

?uM (- 2 i) is a subroutine that returns, based on `DEVICE` , the UART base address (memory or I/O) in register 2 and indicators `0= NOT` if the address is in memory.

Unfortunately, there are many other places in our system where UARTs are directly manipulated, and every one of them must be converted. Here is the status as of this writing of each that we know of:

- Normal terminal I/O. **Converted and tested.**
- SERIAL utility (terminal emulator). *Not converted.*
- SLIP transport in TCP/IP. *Not converted.*

- Serial clulsterFORTH. *Not converted.*
- MATRIX Loop I/O. *Not converted.*
- MATRIX SERIAL.EXCHANGE protocol. *Not converted.*
- MATRIX serial mouse for old RMS workstations. *Not converted.*
- MATRIX RMS workstation protocol. *Not converted.*
- MATRIX Seismic Loop I/O. *Not converted.*

12.5.2 PCI Configuration Space

On a typical machine we find the following presented by the EXAR XR17D158 chip:

```
Bus=1 Dev=4 Fun=0 Class=7 0 2 Rev=9 Vendor=114F Device=B0 IRQ=C

HEX 1 4 0 .PCI
   0    B0114F    800002   7000209         0
  10  EC000000         0         0         0
  20         0         0         0    B0114F
  30         0         0         0       10C
```

Note that the class codes are the same (7,0,2) as for the GTEK board; use caution in setting `th` .

The first register maps the 4k memory address space used by the chip. Each of the 8 UARTs has 512 bytes, and there is a set of device configuration registers at hex +80 in the first UART's allocation. Interrupt 12 has been assigned to this board. Note that the device ID code indicates the number of UARTs supported (B0 means 4 ports, B1 means 8 ports; there are at least 9 more device ID's that are or have been used by Digi Neo devices but they seem to apply to different connectors on backplane like DB9, RJ45, RS422/485, PCI-E 4 port, and IBM version) and that the Neo uses generic serial interface subclass unlike the older Digi ClassicBoards. ***The present code recognizes only device ID's B0 and B1.***

12.5.3 EXAR Chip Device Configuration Registers

These registers are fully initialized from onboard serial EEPROM and in the unit first evaluated the values are as follow:

```
HEX EC000000 80 + 14 XD
    000   00 00 00 00 00 00 00 00 00 00 00 00 09 28 00 00
    010   FC 00 00 FF ok
```

Most of these are unimportant to us. These are relevant: Interrupt status is used in the interrupt code to identify a UART to service, and the rest are used if indicated to reset the chip in **`/neo`** :

`+80` Interrupt Status. Bit *n* means UART *n* is requesting an interrupt.
`+84` TIMERCNTL. Our reset writes all zero to disable, and reads to reset any interrupt.
`+88` 8xMODE. Bits *n* doubles speed of UART *n*.
`+8A` RESET. Write 1 to bit *n* to reset UART *n*. Our reset writes all 1.
`+8B` SLEEP. Must be zero to disable sleep mode.
`+8E` REGB. Write to enable gang configuration or manually access SPI device. Our reset zeroes it.
`+8F` MPIOINT. Must be zero to prevent any MPIO interrupts. Our reset zeroes it.
`+90` MPIOLVL. Our reset reads it to reset any interrupt.

We do not use the general purpose timer, sleep mode, or MPIO capabilities of this chip. Interestingly, the MPIO lines are by default all set for input, no inversion, and all interrupts masked. The value `FC` above is read from the pins and indicates that two of the pins are at least connected to something different than are the others; beware!

12.5.4 UART and Interrupt Status Register

Here is the I/O area on a 4 port board:

```
: Z  8 0 DO  I 200 * EC000000 + 10 XD LOOP ;  Z
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 60 00 FF 00 00 00 00 00 00 00 00
    000   00 00 01 00 00 7A 11 FF 00 00 00 00 00 00 00 00
```

It is rather interesting that UART 7 seems to power up with garbage (break, framing, overrun errors; CTS, delta CTS) on its receiver while the others do not. ***This makes me wonder if the board actually does reset cleanly.***

12.5.5 Reset and Initialization

The EXAR manual talks as though there is a chip-wide soft reset function, but nowhere is such a function described. The only thing that sets the Device Configuration Registers to their defaults is a PCI bus reset, which occurs only when the computer powers up or the equivalent of a `COLD-BOOT` is performed, so we must set these to desired states ourselves. As for the UARTs, we can use the RESET register however nowhere does the data sheet indicate how long this RESET operation actually takes. The manual reset is done by **`/neo`** callable from code, and **`/NEO`** from high level. The manual reset procedure is included in the `RELOAD` chain.

12.5.6 XR17D158 Octal UARTs

There are some departures from the 16550 PC model.

1. The MSR output bits are not implemented and therefore do not control enabling of the interrupt for each UART. This is problematic because as we have seen unused UARTs such as 7 may be connected to garbage on the board, and any UART *we do not wish to use* may still be listening to all manner of events from an existing or nonexistent device. For this reason, it's imperative that we initialize IER to zero for all UARTs.

This UART is upward compatible with the 16450 and it is unknown whether the 16450 race is faithfully reproduced. Many extended capabilities are supported, but we have not made use of any of them.

12.6 Axxon MAP/950 PCI (8 Ports) Model LF571 ONLY

This board uses Oxford OX16PCI954 for PCI & local bus interface plus an octal UART, and an OX16C954 on the local bus for a second 8 ports. ***A similar seeming board, the LF716KB, is not supported*** because unlike the LF571 it does not have mechanism for determining which UART(s) are requesting interrupts without disturbing the registers of the UARTs themselves.

12.6.1 Level of Support

This board is fully supported as a multiport serial adaptor using our standard softvectoring mechanism. Application oriented interrupt routines written to 16450 standards may be used without change.

12.6.2 PCI Configuration Space

This board has two functions, corresponding with the two quad UARTs. The I/O space for the first four UARTs is not necessarily contiguous with that assigned for the second.four. Here are the two functions as seen on a Matrix MS430 box:

```
Bus=3 Dev=1 Fun=0 Class=7 0 6 Rev=0 Vendor=1415 Device=9501 IRQ=B
Bus=3 Dev=1 Fun=1 Class=6 80 0 Rev=0 Vendor=1415 Device=9511 IRQ=B

HEX 3 1 0 .PCI
```

```
   0  95011415   2900003   7000600    800000
  10      E881  FEB9F000      E801  FEB9E000
  20         0         0         0     11415
  30         0        40         0       10B
  40  6C010001

HEX 3 1 1 .PCI
   0  95111415   2900003   6800000    800000
  10      E481  FEB9D000      E401  FEB9C000
  20         0         0         0     11415
  30         0        40         0       10B
  40  6C010001
```

The first register for each function I/O maps the quad UART. The register at 18 maps the configuration register space. The board is wired with interrupts for ports 4 thru 7 connected to MIO bits 7 thru 10, so that the interrupt commutator must read two registers, then mask, align, and combine them to have the equivalent of an 8-bit interrupt request register.

12.6.3 Reset and Initialization

There doesn't seem to be a clean board-reset function, so on RELOAD we must rely upon the PIC mask until the UARTs have been configured. It is possible that this may lead to trouble on RELOAD with shared interrupts. The driver has been written to support interrupt sharing but that capability has not been tested.

12.6.4 OX16C954 Quad UARTs

There are some departures from the 16550 PC model.

1. The MSR output bits are not implemented and therefore do not control enabling of the interrupt for each UART. This is problematic because as we have seen unused UARTs such as 7 may be connected to garbage on the board, and any UART *we do not wish to use* may still be listening to all manner of events from an existing or nonexistent device. For this reason, it's imperative that we initialize IER to zero for all UARTs.

This UART is upward compatible with the 16450 and it is unknown whether the 16450 race is faithfully reproduced. Many extended capabilities are supported, but we have not made use of any of them.

12.7 NCR (Symbios) 53C8xx PCI SCSI Host Adaptor Chips

NCR's Microelectronics Division made the 53C7xx (Microchannel) and 53C8xx (PCI) SCSI chips. These parts will be referred to by the name NCR since as a result of ownership changes (AT&T, Symbios/Hyundai, LSI Logic, and after that who knows, Dell?) the name is simply too much trouble to chase continually.

12.7.1 Chips

The Microchannel version of sF, no longer supported, understood the 53C700. These PCI chips bear a close family resemblance to that chip. The following sections introduce each chip in developmental order, identifying features which were introduced at that model. This is feasible since technical progress through this line is in general upward compatible. Along with the discussion of features is included an example of configuration space for the chip taken after sF has booted in machines that *do* have BIOS code available for initializing Symbios chips; hence the values do not necessarily reflect power-up states. Note that in each case the last half of configuration space (80 through FF) is an image of the internal register space.

12.7.1.1 53C810

The original, basic single ended SCSI-2 chip. Superseded by 810A, but still encountered on Teknor industrial processor boards. Its microcode is upward compatible from the 700 and the chip supports additional features:

- 3 timers.
- Direct single ended drivers.
- 5/10 Mb/sec SCSI transfer rates.

- 64 byte DMA FIFO.
- Relative branch addresses in microcode.
- "Table Indirect" addressing in microcode for buffer descriptors & negotiated transfer properties.
- I/O instructions add set/clear carry function, encode SCSI ID instead of including bitmask.
- Jump instructions add test for carry, add Interrupt On The Fly.
- New instruction: Read/Write.
- New instruction: Move.

```
0 9 0 .PCI  (Teknor 810)
   0     11000  82000007   1000002      2000  No cache support.
  10      6101  E2000000         0         0
  20         0         0         0         0
  30         0         0         0       10B  No cache support.
  40         0         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70         0         0         0         0
  80        D8  1F060067     40000   20F0080
  90  795C3154  FFFFFF00  2035F000         0
  A0         0  820B0000     9FA14     9FA14
  B0     9FA1C   6006140   9001580    13F430
  C0       68F   F0000EB    80000C  80000007
  D0  FFFF0000  FFFFFF00  FFFF0000  91920002
  E0  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
  F0  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

12.7.1.2 53C815

This is essentially the same core as the original 810 but with:

- Supports local memory bus for PROM/FLASH. Has GPIO2,3,4.
- Op code burst fetch added from here on out.

```
0 0E 0 .PCI  (Cimbel 815)
   0     41000   2000007   1000004      4200
  10      F801  FEDFF800         0         0
  20         0         0         0         0
  30         0         0         0  40080109
  40         0         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70         0         0         0         0
  80  330000D8   F060047     F0000   2090080
  90     97D30  FFFFFF00  4031F000  49412874
  A0   F000800  86830000     9E9F0     9E9F0
  B0       1C8  25F7685A  8100008A     9EBB8
  C0         0   E300200    80000C  80000007
  D0  FFFF8200  FFFF0000  FFFF0000  395038A3
  E0  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
  F0  FFFFFFFF  FFFFFFFF  FFFFFFFF  FFFFFFFF
```

12.7.1.3 53C810A

This is a considerably updated version of the 810 with 815 enhancements (but no local memory bus) and the following:

- 3.3v capable.
- 7/10 Mb/sec SCSI transfer rates.
- PCI cache support.
- 80 byte DMA FIFO.
- Microcode prefetch.
- Selectable IRQ disable (DCNTL:1)
- New instruction: Load & Store.
- New instruction: No Flush Mem-Mem Move.

```
0 0F 0 .PCI  (Cimbel 810A)
   0     11000   2000017   1000012      4208
  10      F401  FEDFF400         0         0
  20         0         0         0         0
  30         0         0         0  4008010A
  40         0         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70         0         0         0         0
  80   30000D8  1E060067     40000   2000080
  90     97D30  FFFFFF00  2131F000         0
  A0   1000000  820B0000     9F634     9F634
  B0     9F63C         0  A100158E    13EC70
  C0       68F   E400600   E80000C  80000047
  D0  80000000  80000000  80000000         0
  E0         0         0         0         0
  F0         0         0         0         0
```

12.7.1.4 53C895

This chip supports high performance wide, Ultra-2 SCSI transfers with the following enhancements:

- Supports local memory bus for PROM/FLASH. Has GPIO2,3,4.
- Direct S/E and LVD drivers. Has DIFFSENS pin.
- 80 Mb SCSI transfer rates.
- 112/816 byte DMA FIFO.
- 4k internal microcode storage.
- New instruction: Chained Move
- Subsystem & Subsystem Vendor ID's.
- Microcode is *compiler* compatible with 7xx/8xx according to manual. The object is upward compatible except for the ID field encoding in all 8xx chips. We find no other barriers to compatibility.

```
0 10 0 .PCI  (Cimbel 895)
   0     C1000   2000017   1000001      F708
  10      F001  FEDFF000  FEDFE000         0
  20         0         0         0  10001000
  30         0         0         0  401E010B
  40         0         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70         0         0         0         0
  80  770000D8   B0F0047         0   2000080
  90     97D30  FFFFFF00  1131F000     80008
  A0    240800  86830000     9E9F0     9E9F0
  B0       1C8         0  A100004E     9EBB8
  C0         0   ED01C00    80000C  80000C07
  D0  80E00000  80E00000  80E00000         0
  E0  2E04594F  AC5F7EF7  5D880008  BF9BB7CF
  F0  84800208  7DF3F544  A831B63A  DB6156B9
```

12.7.2 Registers

The internal registers of these chips are accessible in the second half of PCI configuration space (80 through FF), and via I/O and/or memory mappings as may be enabled. Most are 8 bits long, although some *aligned* pairs and quartets are grouped into 16 and 32 bit registers. To maximize annotation space, the table below is only 8 bits wide. Dark shaded bits are reserved; lightly shaded bits and text indicate items that differ within the 8xx line, and in each register's case the point of introduction or change is identified. Register use appears upward compatible within the 8xx but certainly not from the 7xx.

Bit and/or register names are emphasized in bold if they are particularly relevant to our coding.

The registers have defined states on power up or reset. These are documented in the individual manuals and are shown in the diagrams below as non bold values in the bottom of each bit cell. When our normal operating values differ, they are shown in bold italic parenthesized form. If there are significant differences between chips in initial values, these are shown with asterisks.

All registers are read/write unless otherwise noted. ***As with the 53C700, when a SCRIPT is executing none of these registers may be accessed except ISTAT, under threat of chip malfunction.***

SCNTL0	+0	ARB 0=simple 3=full 3		START 0	WATN 0	**EPC** enable parity ***(1)***		AAP atn on parity 0	TRG tgt mode 0	START, WATN, TRG norm set by script. Avoid write to reg after init.
SCNTL1	+1	EXC 1clk xtra tx 0	ADB assert data 0	DHP tgt only 0	CON con'd status 0	**RST** bus reset 0	AESP force parerr 0	IARB immed arb 0	SST start xfer 0	Bus reset 25u minimum. Avoid write to reg aft init.

SCNTL2		+2	SDU disconn unxpctd 0	CHM chained mode 0	SLP MD slpar 0	SLP HBEN slpar 0	WSS wide send 0	VUE0 tgt only 0	VUE1 tgt only 0	WSR wide rcv 0	Shaded at 895 for wide, chain. Chaining should be irrelevant. However WSS/WSR may be pertinent for odd transfers.
SCNTL3		+3	**ULTRA** ena fast 20 & 40 0	**SCF** sync sclk conv fac 0..7 = 3,1,1.5,2,3, 4,6,8 This /4 rcv rate. 0			**EWS** enable wide 0	**CCF** async sclk conv fac]50-75, 16.6-25,]25-37.5,]37.5-50,]50-75,]75-80, 120, 160 ***(3)***			Shaded at 895. Truly looks as though we must measure the damn clock!
SCID		+4		**RRE** reselect enable 0	SRE sel ena tgt only 0		**ID3** only for wide 0	**ID** ID for this chip. 7 is max priority in all modes. ***(7)***			Source ID RRE not set in first version of the code.
SXFER		+5	**TP** Sync xfer period is 4 + this value in SCLK/SCF units 0			**MO4** added at 895 0	**MO** Max sync offset. 0=async; up to 8 for fast/10, to 31 for fast/40. 0				Register is loaded from I/O data structure on Table Indirect I/O commands. Negotiated values.
SDID		+6					ID3 only for wide 0	ID ID of (init) tgt being (re) sel Set by script instruction 0			Destination ID
GPREG		+7				GPIO4 flash wrt arm x	GPIO3 diff_ input x	GPIO2 no use x	GPIO1 serial nvram x	**GPIO0** 0 = led active x	All default as inputs. Shaded only in chips with local mem buses (815, 895.) Symbios boards use as shown.
SFBR		+8	First byte in each block move receive such as msg, status, data in. Believe this ended up handy in the 53C700, must check that								First Byte Received
SOCL		+9	REQ	ACK	BSY	SEL	ATN	MSG	C/D_	I/O_	SCSI Output Control Latch Line assertions by scripts.
SSID	ro	+0A	VAL ID is valid 0				ID3 only for wide 0	ID of entity selecting or reselecting us 0			Selector ID VALid only if multi init protocol (two lines on select) is in use.
SBCL	ro	+0B	REQ	ACK	BSY	SEL	ATN	MSG	C/D_	I/O_	Bus Control Lines Instantaneous current state.
DSTAT *access timing restrictions*	ro	+0C	DFE fifo empty 1	**MDPE** parity if ctest4:3 0	**BF** PCI bus fault 0	ABRT kill via istat:7 0	SSI single step 0	**SIR** intrpt instr 0		**IID** illegal instr 0	DMA Status. Read clears DIP in ISTAT and current set in this. May reveal stacked bits.
SSTAT0	ro	+0D	ILF sidl full 0	ORF sodr full 0	OLF sodl full 0	AIP arb in progres 0	LOA lost arb 0	WOA won arb 0	RST bus rst line 0	SDP(0) parity line 0	SDP is SDP0 for wide.
SSTAT1	ro	+0E	FF bytes/halfcells in SCSI fifo. 0				SDP(0) x	MSGL x	C/D_L x	I/O_L x	FF4 is part of FF on 895. Lines latched by leading REQ
SSTAT2	ro	+0F	ILF1 sidl msb full 0	ORF1 sodr msb full 0	OLF1 sodl msb full 0	FF4 0	SPL1 sdp(1) latched x	DM difsns ≠ stest2:dif x	LDSC disconn occured 1	SDP1 sdp(1) line x	LDSC indicates discon/recon has happened, poss. new context. Shaded ff4 895, dm LVD, rest wide.
DSA		+10/4	Base address for Table Indirect operations. x								Data Structure Address
ISTAT		+14	**ABRT** set to abort 0	**SRST** reset chip 0	SIGP signal hst/scrp 0	SEM either way 0	CON conn'd to bus 0	INTF int on the fly 0	**SIP** scsi int sist0,1 0	**DIP** dma int dstat 0	Interrupt Status. See manual, use caution.
reserved		+15/3									
CTEST0		+18									Old 700 reg, scratch till 895 res.
CTEST1	ro	+19	FMT bits indicate empty bytes in bottom of DMA fifo, all 1 fifo empty.				FFL bits indicate full bits in top of DMA fifo, all 1 fifo full.				
CTEST2	ro	+1A	DDIR inbnd to host 0	SIGP copy of istat:sigp 0	CIO i/o win enabled x	CM mem enabled x	SRTCH adrs in scra&b rw 0	TEOP internal endpro c 0	DREQ internal datarq 0	DACK internal datack 0	Shaded in 895. When this reg is read, istat:sigp cleared.
CTEST3		+1B	Chip Rev ro x				**FLF** flush dma fifo 0	**CLF** clear dma fifo 0	FM uses gpio0 0	WRIE wr&inv enable 0	FLF must be cleared when done. Shaded at 895.
TEMP		+1C/4	Return address for CALL instruction								
DFIFO		+20	DBO byte offset in dma fifo for xfer to/from SCSI. Use with DBC reg (and ctest5 if big fifo enabled) to calc amount left in the fifo.								Shaded at 895.
CTEST4		+21	**BDIS** disable burst 0	ZMOD set chip high Z 0	ZSD scsi dat high Z 0	SRTM shad reg tst 0	**MPEE** master par ena 0	FBL byte lane select for fifo testing with ctest6 0			MPEE enables PCI and target reported parity errors.

CTEST5	+22	ADCK step dnad 0	BBCK step dbc 0	**DFS** enable big fifo 0	MASR internal twiddle 0	DDIR internal twiddle 0	**BL2** dmode extens x	BO9,8 dfifo:dbo extension x		Shaded at 895.
CTEST6	+23	DMA FIFO writes to byte lane selected by ctest4:fbl, test only.								
DBC	+24/3	24 bit count for block move; zero illegal. Also part of instruction reg, and holds offset for table indirect addressing.								DMA Byte Counter
DCMD	+27	Part of instruction register.								DMA Command
DNAD	+28/4	General purpose address pointer used during instruction fetch/execute.								DMA Next Data Address
DSP	+2C/4	Instruction pointer. Write (high byte) starts processing.								DMA Scripts Pointer
DSPS	+30/4	2nd half of instruction register.								DMA Scripts Pointer Save
SCRATCHA	+34/4	General purpose. Address of onchip RAM if ctest2:srtch set.								Shaded at 895.
DMODE	+38	**BL** (see ctest5) log_2 (max burst length, cells) - 1 0		SIOM move src I/O 0	DIOM move dst I/O 0	**ER** enable rd line 0	**ERMP** enable rd multi 0	**BOF** ena brst op fetch 0	MAN no start dsp wrt 0	DMA Mode. BOF at 815, ERMP at 810A.
DIEN	+39		MDPE 0	BF 0	ABRT 0	SSI 0	SIR 0		IID 0	DMA Interrupt Enable. Set 1 to enable interrupt. Masking does not prevent halt or istat:dip tho.
DWT/SBR	+3A	Scratch register. Was watchdog timer in 7xx chips.								Name change at 810A
DCNTL	+3B	**CLSE** cachlin siz ena 0	**PFF** flush prefetc h 0	**PFEN** enable prefetc h 0	SSM single step 0	IRQM totem driver 0	STD start or step 0	IRQD disable irq pin 0	COM compat w/700 0	DMA Control. Shaded at 810A.
ADDER	+3C/4	Internal register, test only								Adder Output
SIEN0	+40	M/A 0	CMP 0	SEL 0	RSL 0	SGE 0	UDC 0	RST 0	PAR 0	SCSI Interrupt Enable (0)
SIEN1	+41				SBMC 0		STO 0	GEN 0	HTH 0	SCSI Interrupt Enable (1). SBMC at 895.
SIST0 *access timing restrictions*	+42	M/A phase mis-match	CMP arb function compl	SEL selectd (target only)	RSL re-selectd	SGE gross scsi error	UDC unexp disconn (sel to?)	RST scsi bus reset	PAR scsi parity error	SCSI Interrupt Status (0). Read clears current set in this. May reveal stacked bits. Only initiator mode shown.
SIST1 *access timing restrictions*	+43				SBMC busmod change difsens		STO select time out	GEN general purpos e timeout	HTH inter-hanshk timeout	SCSI Interrupt Status (1) Read clears current set in this. May reveal stacked bits. Only initiator mode shown.
SLPAR	+44	Any write zeroes. For wide, 16 bit register presentation controlled by scntl2:slpmd and :slphben								Longitudinal parity. Shaded at 895.
res/SWIDE	+45	Wide Residue. Various meanings.								SWIDE at 895
MACNTL	+46	Chip type, encoded & undefined				**DWR** data writes 0	**DRD** data reads 0	**PSCPT** indirect tbl indir 0	**SCPTS** instruc fetch 0	Memory Access Control Set bits route these operations locally. Not clear if PCI asg adr decodes also done.
GPCNTL	+47	ME master gpio1 0	FE fetch gpio0 0		IO4 0	IO3 1	IO2 1	IO1 1	**IO0** active low led ***(0)***	The IOn bits are zero for output, 1 for input on GPIOn. Shaded in 815, 895.
STIME0 *zero before chg*	+48	**HTH** Handshake to Handshake max interval from sack to sack 0				**SEL** Select Timeout 0				SCSI Timer (0) Values are no. of bits to shift 125/2uSec, but zero disables the timer.
STIME1	+49		**HTHBA** start hth on sbsy 0	**GENSF** stretch gen scl 0	**HTHSF** stretch hth scl 0	**GEN** General Purpose 0				SCSI Timer (1). Shaded in 895. Scale factors increase basis by factor of 16 to 1 mS.
RESPID0/1	+4A/2	**Bitmask of IDs** recognized on select or reselect; lsb is ID 0, 8 or 16 bit register for narrow or wide chips. Presumably 1 means recognized.								High byte reserved <895
STEST0	ro +4C	ID3 valid if wide 0	ID SCSI ID as which we have been (re)selected. 0			SLT internal select 0	ART internal arb max 0	SOZ syn ofs at zero 0	SOM syn ofs at max 0	ID at 810A. ID3 in 895.
STEST1	+4D	**SCLK** use pci for sclk 0	SISO isolate scsi bus 0			**QEN** power up 4*clk 0	**QSEL** use 4* for sclk 0			SISO at 810A. QEN/QSEL 895. Sequence rules for turning on the clock quadrupler.

		7	6	5	4	3	2	1	0	
STEST2	+4E	SCE force sodl to scsi bus 0	**ROF** rst ofs after a gros err 0	**DIF** cfg for external hvd bus 0	SLB loop back mode 0	SZM hiZ bus for loop back tst 0	AWS allow wide nondata 0	EXT filtering -usable fast10 0	LOW enable bitbang usage 0	Shaded at 895.
STEST3	+4F	**TE** use active negate ***(1)***	STR enable fifo test reads 0	**HSC** halt the scsi clock 0	DSI require scsi2 select 0	S16 assume all devs wide 0	TTM timer test mode 0	**CSF** clear scsi fifo 0	STW enable fifo test writes 0	Shaded at 895.
SIDL	ro+50/2	Input data latch. Test only.								High byte reserved <895
STEST4	ro +52	**SMODE** diffsens sees: 0=never 1=hvd 2=se 3=lvd x		**LOCK** 4*clk locked on OK 0						reserved <895 Chip tracks SMODE however stest2:dif must be set iff hvd.
reserved	+53									
SODL	+54/2	Output data latch. Test only.								High byte reserved <895
reserved	+56/2									
SBDL	+58/2	Bus data lines (unlatched). Test only.								High byte reserved <895
reserved	+5A/2									
SCRATCHB	+5C									
SCRATCHC	+60									reserved <895
	...									reserved <895
SCRATCHJ	+7C									reserved <895

12.7.3 Microcode (Script) Assembler Syntax

In the 53c700, microcode was assembled to object in a separate step and this object image was absorbed into the nucleus during target compilation. The assembler syntax is upward compatible from that used with the 53c700.

The commands are as follow:

a$_{32}$ *n*$_{24}$ *ph* **MOV** [**|*TBL]	Initiator Block Move
dest id **SELECT** [&ATN]	Select; jump if we were selected/reselected.
WDISC	Wait for Disconnect
dest **WRSEL**	Wait for Reselect
{ATN|ACK} [ACK|ATN +] {**ASSERT**|**CLEAR**}	Assert or clear signals.
dest **JUMP** *cond*	Jump to dest if condition
dest **CALL** *cond*	Call dest if condition
RET *cond*	Return if condition
n$_{32}$ **INT** *cond*	Halt & Interrupt if condition
FWD **JUMP** ... **THEN**	Forward jump forms

ph is phase name {Dout|Din|Cout|Sin|Mout|Min}
cond default is unconditional. Syntax is:
[WAIT] [*ph* PH] [mmdd 1ST] [FALSE|NEITHER]
WAIT waits for valid phase (REQ).
PH tests for phase; 1ST tests First Byte Rcvd = dd with bits 1 in mm ignored.
Both must match; if FALSE|NEITHER, neither must match.

Features and instructions not yet implemented:

- *TBL in MOV
- *TBL in I/O instructions
- Relative *dest* addressing
- ASSERT/CLEAR carry
- Encoded ID in SELECT
- *cond* to test carry
- Interrupt on the fly in jumps
- Read/write instructions (includes ALU)
- Memory move instructions
- Load and Store instructions

12.7.4 Implementation

12.7.4.1 Generalizations

There are several ambiguities that remain in the generalizations. These are:

Termination is not handled identically across even the Symbios boards, let alone other implementations.

SCSI Clock is apparently 40 MHz on all the Symbios boards, and is apparently necessary to obtain the optimal specified SCSI timings, especially those for Ultra-2 (Fast-40). However, the chips apparently work with clocks at least as fast as 50 MHz. In addition, if there is no clock signal on the SCLK pin, the chips may be configured to use the PCI bus clock for SCSI timings. It is not clear that there is any way to determine the actual clock frequency in use without directly measuring it. Further, after examining the registers it is not at all obvious how one easily measures it. Maybe we could try selecting ourselves and see how long it took for the timeout to occur.

Clock Configuration see 2-8, 2-12, 2-11, 2-18 in the four manuals.

Latency Timer values calculable (see 895/3-11). Hmmm.

Our initial support for these chips covers their common synthesis, which may be characterized by the following general statements:

SCRIPTS microcode resides in main memory. This is not optimal for the 895 but is the only model which is applicable to all of the chips.

12.7.5 Boards and OEM Implementations Supported

As far as board level products are concerned, we support several Symbios boards as well as other implementations identified below. The good thing about PCI chips is that they make it difficult for OEM's to introduce many implementation specific details that confound common I/O support. In the case of the Symbios chips, the central differences between implementations, other than choice of chip, are essentially confined to connectors, cabling, and termination control.

12.7.5.1 Symbios Boards

The boards made by Symbios have onboard, transparent adaptive termination logic and with the exception of the latest model boards there is no provision at all for manual or programmatic override of the decisions made by that circuitry. This leads to a definite requirement for care in system configuration since it is possible to subvert the Symbios logic.

According to tech support, and with no schematics to back up the information, a typical Symbios board has two connectors (internal and external), and contains logic to examine each of these connectors independently to infer the presence of devices

on that connector. It seems that the onboard terminators are then *enabled if either connector seems empty* and are therefore *disabled if both connectors seem to have devices.* This obviously leads to pathological behaviors in the following case:

1. One of the two connectors has no cable, bus, or devices attached. Therefore, the onboard terminators are ***enabled***.

2. The other connector attaches to the interior of a SCSI bus, rather than at one end of the bus. While the board sees the bus on this connector, it forces itself to terminate the bus ***erroneously*** simply because its other connector is open.

In the following descriptions of Symbios boards, the above behavior is assumed and only exceptions are discussed where applicable.

12.7.5.1.1 SYM20810 Board (53C810A)

This is the simplest, least expensive board available from Symbios (less than $30 in 4/98). It supports narrow, single ended SCSI up to synchronous speeds of 10 MB/sec. The chip is actually more advanced than the 815 on the next board in line, but without a local memory bus this board cannot be used for BIOS booting. There is however no problem with using the host adaptor after having booted from any other medium we support, and at this price there is little excuse for failure to have stock of usable PCI boards. *The board may be connected to internal and/or external buses, but due to its termination logic the board must be at one end of each bus segment to which it is connected, no exceptions.*

Single ended bus length limits are: 6m for asynchronous operation; 3m for 10 Mb synch with 4 or less devices, 1.5m for 5 or more devices.

12.7.5.1.2 SYM8150S Board (53C815)

This board is also inexpensive (less than $90 in 4/98) and is capable of booting from a SCSI device. It is otherwise functionally equivalent to the 20810, although the advanced features of the 810A chip are not available. Termination rules are the same. Single ended bus lengths are the same.

12.7.5.1.3 SYM8951U Board (53C895)

This board is more expensive (on the order of $200 in 4/98) but supports the most up to date SCSI standards of that date: Wide, Ultra-2 with Low Voltage Differential (LVD) or Single Ended bus signals. The connectors are also fully up to date: Internally, a 68 pin High Density connector, and externally a 68 pin VHDCI connector. Boot capability is supported and the board has nonvolatile RAM so its boot BIOS supports option selection. This board is capable of full Ultra-2 performance, meaning 7/14 Mb/sec asynchronous (narrow/wide), and 40/80 Mb/sec synchronous (narrow/wide).

The board makes no provision for supporting narrow devices in terms of cabling or termination. Appropriate cables and terminating adaptors must be used in such cases.

If all devices on the bus are LVD capable, then the bus runs at 40/80 Mb and the bus length limit is 12m. If any device on the bus is not LVD capable, then the bus must be run single ended which degrades performance to Ultra speeds of 20/40 Mb and bus lengths of 3/1.5m according to one reference, 1.5m according to another. Note also that since it is highly improbable that any LVD device will ever be made with narrow width, it is almost certain that the presence of any narrow device on a bus will force single ended operation and hence the lower performance capabilities. Such situations cry out for multiple SCSI buses.

Termination on this board is significantly more flexible than with the other Symbios boards. Each connector has a two pin jumper block which may be shorted. If a jumper is shorted then the board will assume that devices exist on that connector. This allows the board to be placed in the middle of a SCSI bus on one of its connectors (and by definition in that case the other connector will be open) by shorting both jumpers, or by shorting only the jumper on the open connector. Either of those activities should prevent the board from terminating the bus.

12.7.5.2 OEM Implementations

This section documents non-Symbios OEM products such as SCSI Host Adaptors, mother boards, and embedded CPU boards that use the 53C8xx chips and which we have tested: While the base system may work with some of these, that will be the case

only if (a) bus termination is handled automatically or by manual signal switching or jumpering; and (b) GPIO0 is connected in such a way that it may be used as an activity LED driver.

12.7.5.2.1 Teknor VIPer 820 halfsize Pentium iSBC CPU board

This device uses the old, original 53C810. The board has a nonstandard (adapted) 50 pin header for a single ended bus.

Active termination is provided by Unitrode UC5601DWP. Termination is controlled by jumper W1, 3 pins, which is physically adjacent to the IDE and SCSI connectors. There are three defined settings:

1-2 shorted gives "software" termination control.

2-3 shorted gives "hardware" termination (terminators enabled).

All open disables terminators.

13. PCI ATA Hardware Support

We finally investigated SATA in Winter 2015 because support for PATA emulation (Legacy or Compatibility mode) was becoming spotty when BIOSes such as Dell's combine with current chipsets. Support for PCI PATA/SATA is thus becoming necessary for continued viability of our native system. There turns out to be no real performance advantage unless DMA can be used. Note: The VIA chipset in our P3/1GHz machines seems to present an unusable subset of the standard Host Controller.

13.1 PCI-Native vs AHCI

Leon implemented native mode for FORTH, Inc's system because AHCI was not universally supported. Upon investigation we see the same thing; AHCI generally requires BIOS enablement and that is certainly not universal, nor is AHCI support present in all versions of even Intel's chipsets. Further, it appears we would have to allocate non-cacheable memory for the AHCI area if we have to allocate it ourselves, meaning much carnal knowledge of MTRRs and upper memory allocation by the BIOS. Finally, it is a good bet that whatever other chipsets do it is unlikely identical to Intel, and it is certainly our experience that those chipsets are not documented at all. For these reaso/ns we have chosen, like FORTH, Inc., to support the native PCI mode.

13.2 PCI-Native ATA Host Interface

Native PCI mode is defined in ANSI INCITS 370-2004 with its associated working draft T13/1510D, Revision 1. This specification covers both PATA and SATA host controllers, along with the "compatibility mode" that has been used to emulate the PC "IDE" host interface through 8-bit register space. Essentially the IDE register interface is superimposed upon 16-bit PCI I/O space in multiple instances for each host controller, and is enhanced by a reasonably orderly arrangement for bus mastering DMA. For the devices themselves, we used T13/2015-D Rev 7 as primary reference.

Each Host Controller connects to a single (shared) interrupt and supports two "Channels" with distinct register sets, each in turn supporting two "Ports" or devices selected by the **DEV** bit in the Device register.

13.2.1 Command and Control Registers

The actual command and status register blocks are getting harder to find in the ATA documentation (ATA-6 at least identifies them; the PATA transport standard at ATA-8 shows a table of the registers as well in Annex C.). This table serves to document them as they are at the start of 2016, *using only LBA conventions*; the register structure found at the PCI base addresses is the same as that which was found at 1F0/170 for 8 and 3F6/376 for 2 on the ISA bus.

CMD BLOCK	+0	16-bit data access for PIO mode								Data register
Features	+1 wr	Coded value command dependent								Ancient write precomp
Error	+1 rd	ICRC intfc CRC	UNC uncorr error	obs	IDNF ID not found	obs	ABRT	EOM	multi use error	Ancient different bits
Sec Count	+2									
LBA Low	+3									Old Sector#
LBA Mid	+4									Old Cyl# Low
LBA High	+5									Old Cyl# High
Device	+6		LBA		DEV	LBA 27:24 (28bit mode only)				Old head#
Command	+7 wr	Command Code (written last)								
Status	+7 rd	BSY busy	DRDY ready	DF dev flt or strm error	def'd write error	DRQ data rqst	*align error*	*SDA sense data avail*	ERR any error bit	Ancient different bits
CONTROL	+2 wr	HOB				HS3	RST	-IEN	0	HOB for 48 bit addr
Alt Status	+2 rd									... not used.
		7	6	5	4	3	2	1	0	

The adopted standard for 48-bit mode (48 bit sector addresses, 16 bit sector counts) is simple: Commands that use 48 bit mode have different command codes, and the LBA and sector count registers are all implemented as FIFOs where the high order part is written first and the low order second. The alt status per spec does not clear pending interrupt.

13.2.2 PCI Configuration Space

The following usage is documented in the ANSI Standard for each ATA Host Controller (HC):

+0	Device ID (Vendor Assigned)		Vendor ID (FFFF if not present)		
+4	Status ***(note)***		Command ***(note)***		
+8	Class **01**	Subclass **01**	Register Interface	Rev (vendor asg)	
+0C	BIST Selftest ctl/status	Hdr Type msb=1 if multifunc	Latency Timer for bus masters	Cache line size for bus masters	
+10	Channel 0 Command Block (8 IO)				
+14	Channel 0 Control Block (4 IO, only +2 to be used)				
+18	Channel 1 Command Block (8 IO)				
+1C	Channel 1 Control Block (4 IO, only +2 to be used)				
+20	Bus Master Regs (x10 IO, 8 for 0 and 8 for 1)				
+24	Base 5				
+28	Reserved				
+2C	Subsystem ID		Subsystem Vendor ID		PCI 2.1
+30	Expansion ROM Base Address ***(note)***				
+34	Reserved				
+38	Reserved				
+3C	Max latency r/o	Min Grant r/o	Int pin (signif???)	IRQ #; 255 disables	
	+3	+2	+1	+0	

Relevant fields:

Status	par err	sig syser	mas abort	tgt abort	sig tgtab	devsel 0fast 1med 2slow	data par	fast b-b								
Command							fast b-b	enab serr#	enab step	enab par	vga snoo	enab winv	enab spcy	enab mstr	enab mem	enab i/o

Base, I/O	Addr	Addr	Addr	Addr	0	1
Exp ROM base	Addr	Addr	Addr	Res	Reserved	E
	+3	+2	+1		+0	Enable

Reg Interface	Bus master capable	0	0	0	2 is writable	1 native 0 compat	0 is writable	1 native 0 compat
	7	6	5	4	3	2	1	0

13.2.3 Bus Mastering Registers and Usage

Apparently, any ATA command may move its data using this facility. In general, everything (tables and transfers) must be cell aligned and may not cross 64k boundaries in memory. Each channel has eight consecutive registers as follow:

ds	DMA Status	ds	DMA Command
PRD address (cell aligned, may not cross 64k boundary)			
+3	+2	+1	+0

The command and status registers are as follow, where INTRQ is the actual interrupt for this channel:

Command	+0	0	0	0	0	Write to memory	0	0	Start DMA
Status	+2	Sim-plex	Dev 1 DMA capable	Dev 0 DMA capable	0	0	INTRQ write 1 to clear!	Error	Active
		7	6	5	4	3	2	1	0

The PRD table consists of one or more two-cell entries; no crossing 64k boundaries, last marked by E, and the sum of the counts must be ≥ the total transfer length covered by the command given:

Start Address, must be even			
E	0	Byte count, even, 0=64k	
+3	+2	+1	+0

13.3 sF Support for PCI Native ATA

A new driver replaces the standard IDE support in the nucleus. This driver exclusively uses LBA addressing (28-bit mode by default) and prefers drives that support **READ DMA** and **WRITE DMA** operations. Enumeration is done by Host Controller, Channel and Port as noted above, where in general a port is a single device ("master" or "slave" per **DEV** selection). We enumerate the ATA drives by a *logical port number*; the low order bit of the port number is the **DEV** bit to use, and bit 1 selects one of the two *channels* in a *Host Controller*. Thus four logical port numbers are allocated per PCI Host Controller, and this port number is the fundamental addressing identifier for a physical storage unit.

13.3.1 Enumeration and Discovery

This is mostly done in a couple of phases during **HOME** , however full identification of device capabilities is left for later, after the entitlements of the **9 LOAD** can be assumed.

13.3.1.1 Host Controller Identification

In the first phase, PCI configuration space is examined to find PCI Native ATA host controllers and enumerate the logical ports found there (normally four per interface.) This mechanism is controlled by the following parameters:

m#hc is a CONSTANT giving the max number of PCI-ATA Host Controllers to be checked; default is 2.

a#th is a table listing zero relative instance numbers of PCI Host Controllers to be checked; default is 0, 1.

HOME uses **?atas** to check the prescribed instance(s) of PCI device class/subclass x01/x01 for usability by this code. Usability means that the register interface byte is either set to native mode or is able to be so set (x05 bits set or settable) and also that its bus master capability bit is set. Some HCs would like to assert an interrupt at this time, so we read the main status registers and reset the interrupt bits in bus master status registers. During this enumeration a display is created, showing the ordinal number of the interface checked, and if found its Vendor and Device IDs, followed by qualifying register interface value and interrupt line (if this ordinal is not found, "None found" is shown instead; if the interface is determined to be unusable, then "Not usable" is shown instead of an interrupt line. For example, the below resulted on an AVALUE box with **a#th** set to 1, 0. The first interface is an IDE Compact Flash interface and the second is an NM10/ICH7 SATA interface:

```
0 PCI-ATA 1th: 197B2368  85 7
1 PCI-ATA 0th: 808627C0  8F 7
```

?atas leaves results behind in the following (capabilities & sizes are set later, during **9 LOAD**):

#ap A VARIABLE giving the number of PCI Native ATA logical ports found (four times **#aic**). The following tables hold **#ap** entries each describing a logical port:

achp Chip register interface value used to qualify for usability.

airq Interrupt line number alleged by BIOS in PCI register x3C.

apic Base IO address of PIC for this interrupt.

aimk Mask bit for this interrupt on that PIC.

acmd Base IO address for 8-byte ATA Command Block.

actl Actual IO address for control/alt status register within its 4-byte control block.

amas Base IO address for 8-byte bus mastering register area, accessible with byte/half/fullword operations.

adev Device register value for this port (x40 or x60). Capabilities encoded in **adev 1+** as follow:

drive present	LBA		SMART feature			48-bit addr	DMA
7	6	5	4	3	2	1	0

asz2 Size of device in sectors when 28-bit addressing used.

asz4 Size of device in sectors when 48-bit addressing used.

13.3.1.2 Port and Device Checking

After **?atas** has enumerated ports, we really don't know whether or not any of them actually have devices. The second phase of **HOME** checks for existence, enables interrupts, and so on:

The storage device on a given port may have various capabilities, such as DMA transfers or 48-bit addressing. The enumeration above does not query the device and so, by default, port capabilities are set for 28-bit addressing and no DMA. Incidentally, bits 5 and 6 "Device n DMA Capable" in the Bus Master ATA Status Register have been found set when the storage device does not support DMA. Likewise, valid looking status has been seen on a Compact Flash controller for both **DEV** bit settings.

+atas Scans all of the existing ports and sets the **drive present** bit in **adev** for each that appears to exist.

13.3.1.3 Interrupts

Interrupt management is relatively simple because the ATA devices only interrupt when an operation completes with or without error, or a PIO sector transfer (other than the first sector output) is requested. Therefore we will never see a valid interrupt from any port other than the one we are currently talking to. Our key problem then is working in a heavily shared interrupt environment. To facilitate this, the following data base is used:

aix# IRQ number on which an interrupt is actually expected, -1 if none expected.

'ivs All 16 interrupt vector addresses before we define and enable ATA interrupts.

It seems that bit 2 of the Bus Master ATA status register does in fact indicate any interrupt from the port, without any masking other than the PIC mask, so we need not play guessing games with the state of the status register. That is great!

13.3.2 Native PCI ATA Operations

The code for running commands is factored to accommodate the various capabilities of the storage devices.

13.3.2.1 Primitives

The command issuance procedure is factored into sections that facilitate choice of addressing and DMA capabilities.

!aport (p) Selects the given logical port by setting variables to save indexing hassle during the rest of an operation, and to facilitate selection of appropriate command types based on device capabilities. These variables are all protected by **PCIATA** :

ap# Holds the logical port number for active operation.

advr Holds device register and capability flags.

'acb Holds I/O address of command block.

'adm Holds I/O address of Bus Master DMA register block.

'apc Holds I/O address of PIC for this port's interrupt.

amk Holds PIC mask bit for this port's interrupt.

ai# Holds IRQ number used by active port.

!aop (lba n c) Used after **!aport** to set the Command Block for an operation given by **c** but does not initiate the operation; the value **c** is saved in **afnc** and is encoded as shown below. Sets the Feature Register based on the value in **c** , the Sector Count Register to **n** , the LBA and Device Registers appropriately for the given addressing mode. In 28-bit addressing, the low order 24 bits of **lba** are stored into three registers and the low nibble of the high order byte of this value are merged into the Device Register value from **advr** . When 48-bit addressing is used, the value **lba** may be either single or double precision. When single precision, 16 bits of high order zeroes are generated; when double precision, **lba** must be double on the stack. In either case, 24 high order bits of an effective 48-bit **lba** value are stored into the three LBA registers and the high order 8 bits of the 16-bit **n** are stored into the Sector Count field; and then the low order 24 bits of **lba** are stored along with the low order byte of **n** . In 48-bit addressing the Device Register does not hold any part of the LBA. This allows us to support a maximum of 2 TB with single precision BLOCK numbers.

aptr Address of next sector to move, only used for PIO operations.

actr Number of sectors left to move in interrupt code for PIO operations, or zero if DMA or nondata.

atsk Status address of task to awaken, or adr of next cell if not expected.

afnc Holds the function code, encoded as follows (non-data operations should not have the input bit set):

	Flags	Features	Command
+3	+2	+1	+0

Flags Byte +2

7	6	5	4	3	2	1	0
				M DMA op	D double LBA	X 48-bit	I input

!acmd Starts the operation prepared by **!aop**, storing **afnc** into the Command Register and informing the interrupt code about what is expected of it.

<ats (- s) Waits for an operation to complete then retrieves and returns status (zero unless an error is indicated). The status value is encoded as follows:

	7	6	5	4	3	2	1	0	
+3	0								
+2									Cmd Block Features register
+1	Sim-plex	Dev 1 DMA capable	Dev 0 DMA capable	0	0	INTRQ write 1 to clear!	Error	Active	Bus Master Status register
+0	BSY busy	DRDY ready	DF dev flt or strm error	def'd write error	DRQ data rqst	*align error*	*SDA sense data avail*	ERR any error bit	Cmd Block Status register

!acmd Starts the operation prepared by **!aop**, storing **afnc** into the Command Register and informing the interrupt code about what is expected of it.

13.3.2.2 Generic Operations

Disk reading and writing must be done with canonical operations, described below, to select between PIO and DMA transfers and between 28- and 48-bit addressing. Most other operations require neither of these things and so can be performed using generic non-data or PIO functions as appropriate. These do not require protection via **DISK** but do their own with **PCIATA**:

ataND (lba n c p - s) Performs a non-data ATA command **c** on port **p** using parameters **lba n** and the feature byte encoded in **c**. 48-bit and double precision **lba** modes may be selected using flags in **c**.

ataPIO (a lba n c p - a s) Performs a data-moving ATA command and direction encoded in **c** using the pararmeters given. **n** is the number of 512-byte "sectors" to move, to or from the address **a**. 48-bit and double precision **lba** modes may be selected using flags in **c**. If a command uses the sector count field for something else, a different generic function will have to be created.

13.3.2.3 Canonical Operations

For use across devices that may or may not support DMA, and may or may not support 48-bit addressing, canonical operations examine the capabilities indicated in **adev** and perform the appropriate operation to make best use of those capabilities. Different functions must be used to operate on devices with >2TB capacities.

<ATAC (a n ns p - a s) Reads **ns** sectors from port **p** starting with sector **n** at memory address **a**. The sector address is a 28-bit value unless the port supports 48-bit addressing, in which case it's a 32-bit value. DMA is used if available, PIO if not.

>ATAC (a n ns p - a s) Writes to port **p** in the same way **<ATAC** reads.

13.3.3 BLOCK Operations

To accommodate gaps in the logical port number space and to allow for mapping of block numbers onto the desired ports, the **DRIVES** table previously used with SCSI has been adapted for use with this subsystem but has been renamed **UNITS** in accordance with the Cardinal Rule. It was not feasible to do this in an upward compatible manner, so the structure has been altered; the new structure and conventions are described in the *Mass Storage Management* section earlier in this manual.

13.3.4 Capability Discovery

Initially all discovered ports are assumed to support only 28-bit PIO operations, and are presumed but not required to support LBA addressing. Later on near the end of the **9 LOAD** we may interpret block 520 which interrogates all discovered ports to determine whether LBA, 48-bit mode, DMA or SMART are supported, setting the appropriate flags in **atac+1** for each.

As soon as this has been done, DMA and/or 48-bit addressing will be used for subsequent canonical operations such as **BLOCK** will be used to get the best possible performance from each device.

13.3.5 Open Questions

The first implementation for PCI-ATA, done in January 2016, leaves some open questions to be resolved later:

- On the AVALUE box, we occasionally (1/10000) do a bad read using PIO. In the bad read, we've seen the second sector of a block read into the first 512 bytes of a buffer and the second 512 are unchanged. Speculation is that this results from the BIOS assigning IRQ7 to the PCI-ATA chips; there are many nonsense interrupts received by IRQ7. Perhaps the mask does not work for this interrupt so that the second sector's interrupt uses the same start pointer that has not yet been updated at the end of the first interrupt. Not a problem with DMA but should be run down, explained and fixed.
- We are currently not resetting the port using its control register. If we do, does this affect settings for DMA, and does it affect disposition of unflushed volatile write cache?
- Should we do anything else on **RELOAD** such as disabling the interrupts?
- Should we flush cache before **RELOAD**, noting that this is not practical to do in the coded sections of **RELOAD** ?

14. AHCI SATA Hardware Support

Having decided almost six years ago against implementing AHCI, here we are in Fall 2021 being forced to do so anyway inasmuch as the currently available Intel PCHs support neither native PCI ATA nor the vastly worse IDE emulation. So it goes. The initial work is being done with 6th gen mobile i3-6100U and matching PCH (SKU PCH-U Premium). Referencing as standard SATA AHCI Rev 1.3.1 and ATAPI Command Set 3 (ACS-3).

14.1 AHCI SATA Host Interface

The Host Interface is defined in Serial ATA Advanced Host Controller Interface (AHCI) 1.3.1 Specification. Essentially AHCI is a seemingly complicated way to deliver commands to a SATA device much like the ancient ATAPI command (and get replies of the same sort) as described in T13/2161-D, ATA/ATAPI Command Set-3 (ACS-3). The interface is entirely through "registers" and data structures mapped into memory address space, with special requirements such as non-cacheability. With more study, it appears we can save ourselves more trouble, and two extra SATA messages, by writing the commands in native FIS form. We will do this. There is, fortunately, a well defined protocol for getting ownership of the HBA (Host Bus Adapter) away from the SMM code; unfortunately, this hardware does not implement that method; so, unless the SMM code from the BIOS emulates that function, we are guessing. For the devices themselves, we used Serial ATA Revision 3.2 as primary reference.

Each HBA connects to a single (shared) interrupt and supports up to 32 "Ports" each of which may connect to a single device or to several via a Port Multiplier (which this hardware does not support). Each Port has its own set of registers. The **DEV** bit in the old Device register has no actual meaning any longer.

14.1.1 Classical Command Block (aka Task File)

While we think and write in terms of some of the registers in the Command Block found at 1F0/170 for 8 and 3F6/376 for 2 on the ISA bus, that structure appears nowhere in our code, because we work directly with the FIS messages that are the lowest level of protocol in both the SATA communications and in the AHCI. When "Task File" is mentioned here, it is a reference to these registers, but they are transmitted much more simply in our code. For reference, here is the Task File:

		7	6	5	4	3	2	1	0	
CMD BLOCK	+0	16-bit data access for PIO mode								Data register
Features	+1 wr	Coded value command dependent								Ancient write precomp
Error	+1 rd	ICRC intfc CRC	UNC uncorr error	obs	IDNF ID not found	obs	ABRT	EOM	multi use error	Ancient different bits
Sec Count	+2									
LBA Low	+3									Old Sector#
LBA Mid	+4									Old Cyl# Low
LBA High	+5									Old Cyl# High
Device	+6		LBA		DEV	LBA 27:24 (28bit mode only)				Old head#
Command	+7 wr	Command Code (written last)								
Status	+7 rd	BSY busy	DRDY ready	DF dev flt or strm error	def'd write error	DRQ data rqst	*align error*	*SDA sense data avail*	ERR any error bit	Ancient different bits
CONTROL	+2 wr	HOB				HS3	RST	-IEN	0	HOB for 48 bit addr
Alt Status	+2 rd									... not used.

14.1.2 PCI Configuration Space

The following usage is documented in the AHCI spec 1.3.1, see PCH vol 2 section 16 for details on vendor specifics.

	+3	+2	+1	+0	
+0	Device ID (Vendor Assigned)		Vendor ID (FFFF if not present)		
+4	Status ***(9D03)***		Command ***(8086)***		
+8	Class **01**	Subclass **06**	Reg Interface **01**	Rev (vendor asg)	
+0C	BIST Selftest ctl/status	Hdr Type msb=1 if multifunc	Latency Timer for bus masters	Cache line size for bus masters	
+10	MSI-X Table Base Address				These five registers
+14	MXP Base Address				are not mentioned in
+18	Undefined (IO)				the AHCI spec so are
+1C	Undefined (IO)				specific to this PCH
+20	AHCI Index Data Pair Base Address (IO)				implementation
+24	AHCI Base Address: Low 4 bits 0 => Not cacheable, not prefetchable				
+28	Reserved				
+2C	Subsystem ID 7270		Subsystem Vendor ID 8086		PCI 2.1
+30	Expansion ROM Base Address				
+34				Capabilities Pointer	
+38	Reserved				Vendor specifics:
+3C	Max latency r/o	Min Grant r/o	Int pin (signif???)	IRQ #; 255 disables	
+90	SATA Genl Config		SATA clk gating	Port CS / Port Map	
+A0	SATA Capability 1	SATA Capability 0	SATA Init reg data	SATA Init reg index	
+D0		PBA offset / BIR	Table offset / BIR	Msg ctl / Identifier	MSI-X
+E0		Transmit Data 2	Transmit Data 1	Ctl/Status	BIST FIS

Relevant fields:

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Status	par err	sig syser	mas abort	tgt abort	sig tgtab	devsel 0fast 1med 2slow		data par	fast b-b		HBA 66 MHz cap	capab list exists	INT IS ass-erted				Supersets of PCI Native ATA regs
Command						int disab	fast b-b	enab serr#	enab wcc	enab par	vga snoo	enab winv	enab spcy	enab mstr	enab mem	enab i/o	

	+3	+2	+1	+0	
Base, I/O	Addr	Addr	Addr	Addr 0 1	
Exp ROM base	Addr	Addr	Addr Res	Reserved E	Enable

Here's what we find on our box after successfully booting (from USB URAM):

```
HEX 0 17 0 .PCI
   0  9D038086   2B00007   1060121         0  Cmd enables mstr, mem, i/o;
                                               Sts dvsl med, fast b-b,
                                               66 MHz, capability list
  10  DF148000  DF14D000      F091      F081
  20      F061  DF14C000         0  72708086
  30         0        80         0       10B
  40         0         0         0         0
  50         0         0         0         0
  60         0         0         0         0
  70  4003A801         8         0         0  Pwr mgmt capability
  80      7005         0         0         0  MSI capability (NOT enabled)
  90  82020500       183  20DC0224  80000030  AHCI, ports 0&2 disabled
                                               1 enabled&present, OOB retry
                                               ABAR/MSIX ??, Reglock
  A0        A4         0    100012        48
  B0         0         0         0         0
  C0         0         0         0         0
  D0        11         0         1         0  MSI (looks unused).
  E0         0         0         0         0
  F0         0         0   8400FB3         0  Unknown meaning.
```

14.1.3 HBA Memory Registers

The entire AHCI exists within a single contiguous address space. Here are layouts with the values we found on boot, by section. Max access 64 bits, and accesses may not cross 8-byte alignment boundaries. See PCH Vol 2 §16 for default / RO values.

This memory is supposed to be noncacheable. Verifying that this is so, as we are handed the machine by the BIOS, and indeed it has been. The variable MTRRs start as follows:

```
.VARS A Default=0000.0000.0000.0C06. 6
200  0000.0000.C000.0000. 0000.007F.C000.0800.
```

By this, default memory not listed is write-back cache (6) and the entire range C0000000 thru FFFFFFFF is ***not*** cacheable (0), so the BIOS has done its job properly in setting up memory we can use with the AHCI. (It also seems "special" memory starts at 8D800000 on this box. with 4GB memory installed.)

14.1.3.1 Generic Host Control [0..2B]

Immediately after boot from URAM flash, here is what we find at the start of memory pointed to by the HBA PCI space:

```
HEX DF14C000 2C DUMP
          2          0   80000000   C334FF00  < DF14C000 >  __4C_____________
          0          0          0      10301  < DF14C010 >  ____________________
          0          0         3C          0  < DF14C020 >  ____<_____________
```

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
GHC-CAP	+0	64 bit	ncq ue	sno tify	pre sw	sss	alp m	act led	clo vrd	Max i/f speed 1,2,3 1.5,3,6				res	aho nl	prt mul	fis sw	pio md	slu mb	par tst	# cmd slots/port-1				ccc	em s	ext sat	# ports -1					
seen after boot		1	1	0	0	0	0	1	1	3 (6G)				0	1	0	0	1	1	1	1F (32 slots)				0	0	0	0 (1 ports???)					C334FF00
default on rst		1	1	1	1	1	1	1	1	3 (6G)				0	1	0	0	1	1	1	1F (32)				0	0	0	7 (8)					Why ports diff? RO
GHC	+4	AE	res																											msi rev		IE	HR hba rst	
		1	0																											0		0	0	80000000
IS	+8	Interrupt Pending by port. Other ports if they exist (this controller only has three possible)																												p2		p1	p0	0
GHC-PI	+0C	Ports Implemented (exposed). Other ports if they exist (this controller only has three possible)																												i2		i1	i0	2 => port 1
	+10																																	
VS		Major AHCI version (1)																Minor (3)												Subminor (1)				10301
CCC-CTL	+14																																	0 Not Impl
CCC-PORTS	+18																																	0 Not Impl
EM-LOC	+1C																																	0 Not Impl
EM-CTL	+20																																	0 Not Impl
GHC_CAP2	+24	res																												des onl slu		agr dev slp	sup dev slp	aut opa rtsl um	res	os han doff
																														1		1	1	1	0	0	3C No BIOS h/off
BOHC	+28	res																												BB		OC	SC	OS	BI	Not Supported
		+3								+2								+1												+0						

Notes:

- GHC-IS cannot be cleared until AFTER its sources in the port(s) have been cleared.
- There is a LOT of stuff in section 16 not detailed here, deemed irrelevant once BIOS hands off to us.
- The port control in PCI space is fixed; AHCI stuff can only talk to what's enabled.

14.1.3.2 Gaining Access and Control

The BIOS handoff mechanism is not required by SATA and is not implemented in this device. Frightening. With variation in BIOS behavior that can change with updates, how do we know how to safely take control of the HBA as platforms and BIOSes vary?

Presently we're just assuming it is ours. We may learn that the BIOS is not actually done with it when we get control of the CPU.

14.1.3.3 Port Control Regs [100..1100[by 80

These registers are shown with addresses relative to origin of the Port's sets starting at 100, 180, 200 usw. for ports 0,1,2, ...
After boot, on port 1 we see:

```
DF14C180 80 DUMP
          0      9E000          0      9DC00  < DF14C180 >  _\_______`_______
          0    4000006          0          0  < DF14C190 >  ___________________
        300        133        101         50  < DF14C1A0 >  P_______3_______
<all the rest zero to DF14C200>
```

		+3	+2	+1	+0	
PxCLB	+0	Base address of command list, 0 mod 1024			res	9DC00
PxCLBU	+4	Upper 32 bits				0
PxFB	+8	Base addr of table for received FISes, 0 mod 256 (4k long, 4k aligned if FIS based switching)			res	9E0001
PxFBU	+0C	Upper 32 bits				
PxIS	+10	col dpr es, tfile err or, hbf atal err, hbd ata er, IF fata lerr, IFn onft lerr, res, rx oflo w, por mul sts, phy rdy chg, res, dev me cpr, pcs chg, des pro w/l, unk fis Rx, set dbi nt, dm ase tRx, pio set Rx, d2h fis Rx				0 read/wrt to clr except phyrdy, pcschg, unk
PxIE	+14	^ ^ ^ ^ ^ ^ ^ res ^ ^ ^ <-- enables for corresponding bits above to int --> ^ ^ ^ ^ ^ ^ ^ ^				0
PxCMD(&sts)	+18	ICC cmd to chg power state, 1=active; agg slu mb er; agg lpm low er; drv led; dev is ata pi; aut par t to slu m; fis bas ed sw cap; ext sat a por t; col d pre s det; me c pre s sw; hot plu g cap; prt mu att; col d pre s det; cm dlst run nin g; fis Rx run nin g; me chp res sw sta; CCS Command slot being processed. Reset to 0 on ST transition.			res; phy slp pre sen t; res; FR E fis Rx ena; cm dlst ove rrid e; pod noc old pre sdt; sud spi nup dev ice; ST	04000006 r/o are orange
res	+1C					reserved
PxTFD	+20	res		ERR from latest FIS of task file	STS from same	50
PxSIG	+24	LBA Hi from 1st D2H after reset	LBA Med	LBA Low	Sec count	101
PxSSTS	+28	res; IPM 0=not pres 1=active 2=partial PM 6=Slumber 8=DeepSleep			Current Speed 0=not pres 1=Gen1 (1.5G) 2=Gen2 (3) 3=Gen3 (6); Detection 0=not pres 1=pres no phy 3=pres & phy 4=offline	133
PxSCTL	+2C	res; PMP not used		SPM not used; IPM 3 disables partial&slumber	SPD 0 allows all; DET allows offline detect/reset	300
PxSERR	+30	res; X exc hgd; F bad fist ype; T tra nsp stat; S link seq; H han dsh ake; C crc err Rx; D not use d; B 10- 8de cod; W co mw ake; I phy inte rnal; N phy rdy chg		res; E intr nm em; P pro toc ol; C per sda tint; T tra nda tint; res; M rec co mm; I rec dat aint		0 read/wrt to clr grn must be written here to clr IS greens
PxSACT	+34	Bits set before send command, cleared when a tagged (queued) command is completed.				0
PxCI	+38	Bits set to indicate cmd ready for slot. Cleared when FIS rcvd clearing BSY, ERR and DRQ.				0
PxSNTF	+3C	Bits set by PM Notify messages from port multiplier.				not implemented
PxFBS	+40	Control & status for port multiplier FIS based switching				not implemented
PxDEVSLP	+44	res; DITO multiplier; DITO: DEVSLP Idle Timeout		DEVSLP min assertion time in ms	DETO max time for device to become ready after end of DEVSLP; DS Pe nab; AD SE agg	0 (disabled)

14.1.3.4 Vendor-Specific Registers [A0..FF]

We will ignore these inasmuch as they are what they're called. For the record, after boot we see:

```
HEX DF14C0A0 30 DUMP
          0          0      F0000         78  < DF14C0A0 >  x_________________
          0          0          0          0  < DF14C0B0 >  ___________________
          0         C0         1F     319D03  < DF14C0C0 >  __1_____@_______
```

		+3	+2	+1	+0	
VSP	+A0	res; sfm sup; pf sup; res				0
		res				78
VS-CAP	+A4	res; NVM remapped reg offset		res; Memory space limit, units of 128 bytes; nm rm		For NAND memory
		0; **xF (x780)**		**0**; **0**; **0**		F0000
RPID	+C0	OFST offset to Prem Feat Blk in dwords from begin ABAR.		RAID Platform ID		
		x31 (means C4 below)		**9D03 (default 1C02)**		319D03
PFB	+C4	res; em ail; oe mio ctl				
		res				1F
SFM	+C8	res; orom delay; sm artr esp; rrto nes ata; led loc ate; hdd unl ock; oro mui ban; ena rrt; ena r5; ena r10; ena r1; ena r0				
		0; **0**; **0**; **0**; **1**; **1**; **0**; **0**; **0**; **0**; **0**; **0**				C0

14.2 sF Support for AHCI SATA

A new driver replaces the standard IDE, SCSI or Native PCI-ATA support in the nucleus. Naming convention has been to rename Native PCI model names that start with "a" to start with "ah" so that, if we ever encounter a system that has both, it will be feasible to adapt for it. This driver exclusively uses LBA addressing (48-bit mode by default) and prefers drives that support **`READ DMA`** and **`WRITE DMA`** operations.

14.2.1 HBA Resource Usage

Each port must be given its own FIS structure (256 bytes) in main memory, which holds the text of the most recently received DMA Setup, PIO Setup, D2H, Set Device Bits and Unknown FIS messages. Each port must be given its own Command List Structure (up to 32 entries of 32 bytes each). Each command list entry points to a Command Table, and has a good number of bits for qualifying the mode in which the command is to be executed. The Command Table has the FIS to send the device, the text of an ATAPI command (12 or 16 bytes) which may or may not be used at the sole discretion of the driver, and a Physical Region Descriptor Table of 16-byte entries controlling DMA.

All of this is designed to support an independent queue of commands for each port, and scatter/gather DMA operations, neither of which we employ. Unlike all prior ATA interfaces, we issue all commands by writing FISs to send the device in lieu of the classical ATAPI 8-bit register-oriented command structure, a great idea saving much hassle in messing with the bit fields. When the HBA is running, we need only build a single Command List slot, attach it to a single command list entry, set up only one DMA PRD, and turn on the CI bit for that command list slot to get it started; then wait for an interrupt when it completes. All data transfers, *including "PIO"*, are handled transparently as DMA by the AHCI HBA when used in this mode.

14.2.2 Enumeration and Discovery

Enumeration is done by Host Bus Adapter (HBA) and Port as noted above, where in general a port is a single device (**DEV** is no longer used). We enumerate the AHCI drives by a *logical port number* which is implemented on one of the HBAs we have examined during initialization. The logical port number is in order of discovery; the HBA will almost certainly have disabled many (or most) of its ports, so the first usable port found on that HBA may very well not be "port 0" of that HBA, but will be our logical port zero if that HBA is the first one we examine and is the fundamental addressing identifier for a physical storage unit. This is mostly done in a couple of phases during **`HOME`** , however full identification of device capabilities is left for later, after the entitlements of the **`9 LOAD`** can be assumed.

14.2.2.1 Host Controller Identification

In the first phase, PCI configuration space is examined to find AHCI SATA host bus adapters and enumerate the logical ports found there (maximum of 32 per interface.) This mechanism is controlled by the following parameters:

`m#ahc` is a CONSTANT giving the max number of AHCI SATA HBAs to be checked; default is 2.

`ah#th` is a table listing zero ordinal instance numbers of AHCI SATA HBAs to be checked; default is 0, 1.

`m#ahp` is a CONSTANT giving the max number of logical ports to be enumerated; default is 8.

`HOME` uses **`?ahcis`** to check the prescribed instance(s) of PCI device class/subclass x01/x06 for usability by this code. Usability means that the register interface byte is set to the value for AHCI (1). 01/06 can also be RAIDs, for example. During this enumeration a display is created, showing the ordinal number of the interface checked, and if found its Vendor and Device IDs. If this ordinal does not exist, "**None found**" follows, or if it exists but is not usable (wrong register interface) "**Not usable**" follows. If the device appears to be a usable AHCI HBA, the qualifying register interface value and interrupt line are displayed; ports are checked and the system displays "**LP<l>=<p>**" where <l> is the logical port number assigned and <p> is the port number on this HBA. For example, the below resulted on an AVBOX2 with **`ah#th`** set to 0, 1. The first interface is a 6th Gen PCH with AHCI SATA interface and one device; the second doesn't exist. The first HBA has one port enabled (BIOS can selectively disable ports), physical port 1 on the HBA, which has been assigned as logical port 0 by ?ahcis (the phrase "**Too many!**" below is what would appear after the first and any subsequently discovered ports that could not be used due to exceeding the m#ahp limit on number of ports):

```
0 AHCI-SATA 0th: 80869D03   1 B LP0=1 Too many!
1 AHCI-SATA 1th: None found!
```

?ahcis leaves results behind in the following tables, indexed by port (some set by interrupts and later, during **9 LOAD**):

#ap A VARIABLE giving the number of PCI Native ATA logical ports found (four times **#aic**). The following tables (static after discovery) hold **#ap** entries each describing a logical port: *Named the same as PCI-ATA because it is used in several pieces of code after boot.*

ahirq Interrupt line number alleged by BIOS in PCI register x3C.

ahpic Base IO address of PIC for this interrupt.

ahimk Mask bit for this interrupt on that PIC.

ahGHC Start of HBA noncacheable memory made by the BIOS... Global and per-port registers.

ahPORT Start of HBA noncacheable memory made by the BIOS... Global and per-port registers.

ahFIS Addr of 256-byte Received FIS structure, one part per FIS type with most recently Rx FIS of that type.

ahCLST Addr of 32-entry Command List (of which we use only one).

ahCTBL Addr of 144-byte Command table with outbound FIS, ATAPI command, PRDT (DMA) descriptor.

adev Device register value for this port (x40). *Named the same as PCI-ATA because it is used in several pieces of code after boot.* Capabilities encoded in **adev 1+** as follow:

drive present	LBA		SMART feature			48-bit addr	DMA
7	6	5	4	3	2	1	0

14.2.2.2 Port and Device Checking

After **?atas** has enumerated ports, we really don't know whether or not any of them actually have devices. The second phase of **HOME** checks for existence, enables interrupts, and so on:

The storage device on a given port may have various capabilities, such as DMA transfers or 48-bit addressing. The enumeration above does not query the device and so, by default, port capabilities are set for drive present and LBA, and will by default issue PIO 28-bit commands for read and write until we've queried the device later. Although we'd like to learn more at this point, we really can't trust anything left behind by the BIOS because its tables are in regular memory that we may have clobbered by booting, so are assumptions are minimal. This part of the process is done by two routines:

+ahcis Scans all of the existing ports. It stops their FIS and Command List engines, and sets the **drive present** bit in **adev** for each that appears to exist, as well as setting up all the tables, pointers and interrupts.

+PORTS Turns on the FIS and Command List engines for each port.

14.2.2.3 Interrupts

Interrupt management is relatively simple because the ATA devices only interrupt when an operation completes with or without error. Therefore we will never see a valid interrupt from any port other than the one we are currently talking to. Our key problem then is working in a heavily shared interrupt environment. To facilitate this, the following data base is used:

ahix# IRQ number on which an interrupt is actually expected, -1 if none expected.

'ahivs All 16 interrupt vector addresses before we define and enable ATA interrupts.

The interrupt routine saves status in **ahIS ahSER ahst** from registers **IS** , **SERR** and **TFD** respectively, resets **ahix#** and awakens the task performing the operation for its status analysis.

We may receive more than one interrupt for an operation. The only practical way we've found to discriminate this case and to ignore the first interrupt is to check Task File Status for BSY or DRQ. Presently, this defeats the COMRESET part of error recovery described below and could hang. This warrants further thought!

14.2.3 AHCI Command Operations

The code for running commands is factored to accommodate the various capabilities of the storage devices.

14.2.3.1 Operation Data Base

Because each port is an independent engine for managing its SATA connection and performing DMA, we are concerned here with the Port Control Registers above, and with a minimal set of tables for performing operations. These are:

14.2.3.1.1 Command List

We use only entry zero in the command list, whose address is in **'CLST** after port selection with **!aport** .

	+3	+2	+1	+0	
+0	Length of PRD tbl (0 or 1)		PMP | r | C | B | R	P | W | A | CFL	Written for each op
+4	Report from DMA of bytes transferred				
+8	Command Table Address (0 mod 128)			0 | 0 | 0 | 0 | 0 | 0 | 0	
+0C	Upper 32 bits				

When setting up an operation, length of PRD table is zero for non-data. CFL is always 5 because we only send the Register H2D FIS. C is 1 to clear PI after FIS transmitted and acknowledged. B (BIST) and R (Reset) are zero. P is always 1 to indicate prefetch of PRDs when length>0. W indicates write to device. A (ATAPI) always zero. PMP is zero because we do not use port multipliers. So, for the low order half of cell zero, we use only two values: **x04C5** when writing to device, **x0485** when reading.

14.2.3.1.2 Received FIS Structure

When not using the four decade old IDE/ATAPI command block kludge, this is where we get feedback from the device.

		+3	+2	+1	+0	
DSFIS	+0	res	res	A | I | D | r | PMP	Type (x41)	DMA Setup FIS
	+4	DMA Buffer ID low				A = auto activate
	+8	DMA Buffer ID high				I = Int
	+0C	res				D = Dir (input)
	+10	DMA Buffer Offset				
	+14	DMA Transfer Count (even, nonzero)				
	+18	res				
PSFIS	+20	Error	Status @start xfer	r | I | D | r | PMP	Type (x5F)	PIO Setup FIS
	+24	Device	LBA (low)			I int, D dir (input)
	+28	res	LBA (high)			
	+2C	Status @end xfer	res	Sector count		
	+30					
RFIS	+40	Error	Status	r | I | r | r | PMP	Type (x34)	D2H Register FIS
	+44	Device	LBA (low)			I-int
	+48	res	LBA (high)			
	+4C	res		Sector count		
	+50	res				
SDBFIS	+58	Error	r | stat6..4 | r | stat2..0	N | I | r | r | PMP	Type (xA1)	Set Device Bits FIS
	+5C	protocol-specifid				N=notify I=int

14.2.3.1.3 Command Table

The single command list entry points to our single command table. It begins with an outbound FIS which is always the Register H2D FIS that replaces, and obsoletes, the old ATAPI Command Table. We use a single PRD table entry to provide DMA parameters when relevant.

CFIS	+0	Features 7..0	Command	C	0	0	0	PM(0)	FIS Type (x27)	All of these
	+4	Device	LBA 23..0							are set by each
	+8	Features 15..8	LBA 48..24							command
	+0C	Control	ICC	Count						we send.
	+10	Auxiliary								Not used by our cmds.
ATAPI	+40	12- or 16-byte ATAPI Command, not used.								
PRD Tbl	+80	Start Address (must be even)								Three cells set for each
	+84	Upper 32 bits								data operation.
	+88	res								
	+8C	I	res	22-bit byte count -1						4 MB max transfer lng.
			+3	+2		+1		+0		

The description of "C" in CFIS leaves a little bit to be desired. We found it 1 after BIOS read of sector zero, so will set it 1 too. Semantics of the other registers are as prescribed in ACS-3 command set. The I b it in the PRD table should always be zero.

14.2.3.2 Primitives

The command issuance procedure is factored into sections that facilitate choice of addressing and DMA capabilities.

!aport (p) Selects the given logical port by setting variables to save indexing hassle during the rest of an operation, and to facilitate selection of appropriate command types based on device capabilities. These variables are all protected by **AHCISATA** :

ahp# Holds the logical port number for active operation.

ahi# Holds IRQ number used by active port.

'ahpc Holds I/O address of PIC for this port's interrupt.

ahmk Holds PIC mask bit for this port's interrupt.

'GHC Points to pre-allocated, uncacheable General Host Controller registers.

'PORT Points to this port's pre-allocated, uncacheable Port registers.

'FIS Points to this port's incoming FIS buffer/table.

'CLST Points to this port's command list (we only use 0th entry).

'CTBL Points to this port's command table.

ahdvr Holds device register and capability flags.

Several other variables are meaningful only in the context of the currently selected port:

ahfnc Holds our extended SATA function code, see below.

ahtsk Holds STATUS address for task to awaken. See below.

ahix# Holds the IRQ number currently armed. Set -1 after interrupt received.

#ahi Counts AHCI interrupts received.

ahIS Holds port's interrupt status register as of last interrupt.

ahSER Holds port's SERR register as of last interrupt.

ahst Holds two bytes of task file error|status as of last interrupt.

These four routines are used to compose data and non-data transfer primitives:

!ahprd (a lba n c:) Used after **!ahport** to set address and length in PRD for use by DMA if this is a transfer operation.

!ahop (lba n c) Used after **!ahport** to **!ahop (lba n c)** Used after **!ahport** to set the Register H2D FIS for an operation given by **c** but does not initiate the operation; the value **c** is saved in **ahfnc** and is encoded as shown below. Sets the Feature Register based on the value in **c** , the Sector Count Register to **n** , the LBA and Device Registers appropriately for the given addressing mode. 28- and 48- bit addressing are handled the same, i.e. 48 bits are stored into the FIS; if D is set, the value of **lba** must be double precision, otherwise the high order half is zeroed. This allows us to address a maximum of 2 TB with single precision BLOCK numbers. It seems no special setup is required for PIO operations; these are apparently handled by the DMA engine with no intervention on our part.

ahtsk Status address of task to awaken, or adr of next cell if not expected.

ahfnc Holds the function code, encoded as follows (non-data operations should not have the input bit set):

ahfnc	Flags	Features	Command
	+3	+2	+1	+0

Flags Byte	+2				N Non-Data	M DMA op	D double LBA	X 48-bit	I input
		7	6	5	4	3	2	1	0

!ahcmd Starts the operation prepared by **!ahop** , setting **ahix#** to legitimize the interrupt, the task to awaken in **ahtsk** and turning on the **CI** bit for command slot zero.

<ahts (- s) Waits for an operation to complete then retrieves and returns status (zero unless an error is indicated). The status value is encoded as follows:

+3	CPDS	TFES fatal	HBFS fatal	HBDS fatal	IFS fatal	INFS	res	OFS	High ord octet of IS reg
+2	DMPS	PCS	DPS	UFS	SDBS	DSS	PSS	DHRS	Low ord octet of IS reg
+1	ICRC intfc cRC	UNC uncorr error	obs	IDNF ID not found	obs	ABRT abort	EOM end of media	multi use error	Task File Error register
+0	BSY busy	DRDY ready	DF dev flt or strm error	def'd write error	DRQ data rqst	*align error*	*SDA sense data avail*	ERR any error bit	Task File Status register
	7	6	5	4	3	2	1	0	

Before returning status, **<ahts** performs error recovery as prescribed by AHCI manual section 6.2.2, which may include a port (COMRESET) if the operation ended with BSY or DRQ bits set. *Necessarily, only some of this could be tested*: We've exercised device detected errors such as bad LBA values, *but could not simulate any of the hardware failures at the device or interface or host bus levels*. This error recovery procedure is essential because the command processing engine is halted when any of the "fatal" errors (above) occurs.

14.2.3.3 Generic Operations

From here on, spelling and semantics are the same as they are for PCI-ATA host bus adapters. Disk reading and writing must be done with canonical operations, described below, to select between PIO and DMA transfers and between 28- and 48-bit addressing. Most other operations require neither of these things and so can be performed using generic non-data or PIO functions as appropriate. These do not require protection via **DISK** but do their own with **AHCISATA** :

ataND (lba n c p - s) Performs a non-data ATA command **c** on port **p** using parameters **lba n** and the feature byte encoded in **c** . 48-bit and double precision **lba** modes may be selected using flags in **c** .

ataPIO (a lba n c p - a s) Performs a data-moving ATA command and direction encoded in **c** using the pararmeters given. **n** is the number of 512-byte "sectors" to move, to or from the address **a** . 48-bit and double precision **lba** modes may be selected using flags in **c** . If a command uses the sector count field for something else, a different generic function will have to be created.

14.2.3.4 Canonical Operations

For use across devices that may or may not support DMA, and may or may not support 48-bit addressing, canonical operations examine the capabilities indicated in **adev** and perform the appropriate operation to make best use of those capabilities. Different functions must be used to operate on devices with >2TB capacities.

<ATAC (a n ns p - a s) Reads **ns** sectors from port **p** starting with sector **n** at memory address **a** . The sector address is a 28-bit value unless the port supports 48-bit addressing, in which case it's a 32-bit value. DMA is used if available, PIO if not.

>ATAC (a n ns p - a s) Writes to port **p** in the same way **<ATAC** reads.

14.2.4 BLOCK Operations

To accommodate gaps in the logical port number space and to allow for mapping of block numbers onto the desired ports, the **DRIVES** table previously used with SCSI has been adapted for use with this subsystem but has been renamed **DISKS** in accordance with the Cardinal Rule. It was not feasible to do this in an upward compatible manner, so the structure has been altered; the new structure and conventions are described in the *Mass Storage Management* section earlier in this manual. The structure used is compatible with, and could be merged with, the same table in Native PCI ATA support.

One advantage in the AHCI structure is that by disabling a port in the BIOS one can shift the logical addressing of the Mass Storage universe. If, for example, the main drive for housing the sF system has a lower physical port number than does a possible backup drive, then it would be feasible in the BIOS to disable the main drive, thus shifting the backup down, and to name the backup as the boot medium. In this way, a backup system could be activated without requiring any skill beyond BIOS usage on the part of a field service person.

14.2.5 Capability Discovery

Initially all discovered ports are assumed to support only 28-bit PIO operations, and are presumed (and required) to support LBA addressing. Later on near the end of the **9 LOAD** we may interpret block 520 which interrogates all discovered ports to determine whether LBA, 48-bit mode, DMA or SMART are supported, setting the appropriate flags in **atac+1** for each.

As soon as this has been done, DMA and/or 48-bit addressing will be used for subsequent canonical operations such as **BLOCK** to get the best possible performance from each device.

14.2.6 Considerations Decided Against

We decided to take no action on these:

- Should we do anything else on **RELOAD** such as disabling the interrupts? NO.
- Should we **FLUSH!!** before **RELOAD ?** NO; this is not practical to do in the coded sections of **RELOAD** .

14.3 Intel Chipsets with SATA (reference material)

At the end of 2015 Intel's ICH-7 is still in common use. For example it's used in the MS430 and AVALUE boxes, the Dell SC430 and possibly others (Intel has made the spec updates, which list the device IDs, NDA items.)

Intel supports three modes of operation in the SATA section of the ICH: Native, AHCI and RAID. Not all versions of the chipset can support AHCI (limited to the mandatory parts of AHCI spec rev 1.0) and the device ID does not suffice uniformly for discriminating between these sorts of ICHes.

14.3.1 ICH6 (Jun 2004)

ICH-6M and R introduced AHCI support for rev 1.0. When subclass is 6 (1=IDE, 4=RAID, 6=SATA), 0.1F.2:9 is 1 if AHCI supported; the LPC (0.1F.0) does not have an FDVCT register. AHCI base adr in 0.1F.2:24. Max of x400 bytes of registers there, and manual has large section documenting them.

GA has an ICH6R (0.1F.2 DevID 2652) whose 0:1F:2 has subclass set to 1. This means IDE and in that case AHCI is not available. We'd have to force subclass to 6 in order to find out if AHCI is supported.

14.3.2 ICH7 (Apr 2005)

ICH-7R, DH, M and M DH support AHCI rev 1.0; ICH-7 and 7U do not. FDVCT (0.1F.0:E4) bit 3=1 indicates AHCI support disabled, 0 says available.

Dell SC430 has ICH-7 whose FDVCT is 200000 indicating AHCI (and RAID) capability.

The MS430 has an ICH-7 whose FDVCT is 2400A8 meaning AHCI disabled.

The AVALUE box, Atom with NM10 Express chipset, has ICH-7 whose FDVCT is 906402A1 meaning AHCI capable.

14.3.2.1 Enabling AHCI on an ICH7

Following true from ICH7 thru PCH9:

1. Store x60 into MAP (0.1F.2:90). The 20 bit specifies assignment of all SATA ports to controller 1 in later chipsets. The 40 bit specifies AHCI mode. This will change Subclass to x06 and Device ID to a specific value ... 27C1 for ICH7 desktop, 27C5 for ICH7 mobile; no change for ICH6.

2. Store an appropriate 1K aligned table address into ABAR (0.1F.2:24). Note that this is a request for IO space in IDE mode.

14.3.3 ICH8 (Jun 2006)

Spec update is privileged, don't know how to identify it.

Introduces GbE NIC as part of package.

ICH-8R, DH, DO, E support AHCI rev 1.0; others do not. Has FDVCT as ICH7 with bit 3 indicating AHCI disabled. AHCI base in 0.1F.2:24

14.3.4 ICH9 (Jun 2007)

No direct PATA support from here on. ICH9, R, DEH, DO support AHCI. 9M, EM, M-SFF might not.

Spec update is privileged, don't know how to identify it.

14.3.5 ICH10 (Jun 2008)

ICH10, R, D, DO all seem to support AHCI.

Spec update is privileged, don't know how to identify it.

14.3.6 PCH 5 Series / 3400 (Sep 2009)

No FDVCT register.

SATA rev 1.0a. All chipsets except 3400 (which requires unspecified "appropriate system configuration and software drivers") support AHCI rev 1.0 mandatory features.

14.3.7 PCH 6 Series / C200 (Jan 2011)

All chipsets except H61 (which requires unspecified "appropriate system configuration and software drivers") support AHCI with rev 1.2 mandatory features. All but the H61 also support at least one 6GB port. eSATA is also supported in AHCI mode.

snake (Acer) has an H67,

14.3.8 PCH 7 Series / C216 (Apr 2012)

SATA rev 3.0. All chipsets support AHCI with rev 1.3 mandatory features.

14.3.9 PCH 8 Series / C220 (Jun 2013)

14.3.10 PCH 9 Series (May 2014)

15. ATA Device Operations

ATA command sets provide ways to manage device status and operating modes. Some of these have proven important and are supported in sF.

15.1 SMART

"Self-Monitoring, Analysis and Reporting Technology" was created by manufacturers for prediction of device degradation and/or faults, and later standardized by T13 as part of ATA. Most useful for our purposes is a set of Attributes maintained by a drive and made available to us. Unfortunately, while the Standard defines the means for retrieving the Attribute list from a device, and the format of the first four bytes of the 12-byte Attribute structure, there is no guaranteed common ground in the set of Attributes a given device supports, nor in the meaning of a given Attribute nor of the meaning and format of the remaining 8 bytes of the structure. All of this is vendor-specific, and while some vendors (Intel) document the attribute list for each device model scrupulously, others (Western Digital) simply list names of attributes without much (or necessarily any) discussion such as defining the format, meaning, value or units of the attributes, and yet others (SanDisk) do not bother to provide documentation anywhere we've been able to find. Nevertheless the information is vital and worth accessing.

The SMART Attribute table consists of 2 bytes of structure version followed by a sequence of up to 30 12-byte Attribute entries. The basic format of an Attribute is as follows:

		7	6	5	4	3	2	1	0	
	+0	Attribute ID								Valid if nonzero
Status Flags	+1	res	res	SP	EC	ER	PE	OC	PW	
More Flags	+2	res	res	res	res	res	res	res	res	
	+3	Normalized Value								
	+4	*Threshold or other reference for Value (normally)*								Not guaranteed.
	+5									Sometimes n-bit little endian
	+6									"Raw Value"; other times
	+7									multiple fields.
	+8									
	+9									
	+10									
	+11									

Even the rough layout shown above is not prescribed by ATA; according to the Standard, the entire 362-byte area is shown merely as "Vendor specific".

The PW bit, as used by Intel, identifies attributes that indicate imminent drive failure when Value reaches a threshold in one direction or the other. The value of byte +4, from Intel, is always 100 and the actual threshold, if any, is specified in the device data sheet.

The OC bit, as used by Intel, indicates that the Attribute is maintained during normal operation and not by some offline process. Typically all Intel Attributes have this bit set.

The remaining status flags, tinted grey above, are Vendor specific but their definitions by Intel are as follow:

SP: Self-preserving Attribute.
EC: Event count.
ER: Error Rate.
PE: Performance Attribute.

Even this much, and even on an Intel drive, must be taken with a grain of salt because its own use of SP and EC for example is not self-consistent across drives.

15.1.1 Attributes of Interest

The following table identifies SMART attributes we should monitor for early warning purposes. It includes only data from devices and manufacturers we are presently supporting in the field. Mfgr codes are I Intel, S Samsung, U Universal so far. This

table only shows Attributes we have been able to see nonzero, or exercise, or otherwise verify. There are several more that are potentially interesting and they will be added after they have been seen and verified.

ID	Mfgr	Name	Value Decode
x05	U	Retired Blocks	Raw value is count (unverified, have seen none so far.)
x09	U	Power-on Hours	Raw value is number of hours. Assume 32 bits.
x0C	U	Power Cycles	Raw value is count. Assume 32 bits.
xAA xE8	I	Available Reserved Space	Normalized value in % goes down from 100. Threshold is 10.
xAE	I	Unclean Shutdowns	Raw value is number of shutdowns without FLUSH!! Assume 32 bits.
xBB	IS	Uncorrectable Errors	Raw value is count
xBE	IS	Temperature °C	i3700: BE 22 00 43 3C 21 00 20 29 00 00 00 bosk i2500: BE 32 00 22 64 22 00 30 00 15 00 00 small i2500: BE 32 00 21 64 21 00 2D 00 1D 00 00 box-a I2500: BE 32 00 22 2A 22 00 2A 00 12 00 00 boxey s850: BE 32 00 38 30 2C 00 00 00 00 00 00 box-a The Normalized Value is either the actual temp or 100-actual. Fortunately the raw data (16 bits) is actual temp in all cases. For the i2500 only, the next two halfcells are lifetime high and low.
xE9	I	Wearout Indicator	Normalized Value goes down from 100 as avg cycles accumulate.

16. PCI/USB Hardware Support

In 2006, roughly 10 years after USB made its debut, it had matured to the point that USB interfaces were beginning to replace standard PC interfaces with a vengeance. Floppy drives were disappearing from desktop boxes; the very FDC (Floppy Disk Controller) chip was disappearing from notebooks, along with the ATAPI over IDE interface for CD and DVD drives. Keyboard controllers are disappearing; PS/2 connectors for keyboards and mice are disappearing. Standard 16450 compatible UARTs and serial ports are disappearing. All of these things are being replaced by USB devices, while the IDE disks are being replaced by SATA devices. Floppies are disappearing as boot media, more aptly replaced by USB Flash devices. Although these trends did not develop rapidly, by 2020 they were maturing. So we initially supported USB 2 (EHCI+UHCIs) in 2022.

The Universal Boot for saneFORTH has been tested and works fine for all USB mass storage devices (USB floppies, USB flash, USB hard disk) other than CD/DVD (due to format issues.), and that will be the case until UEFI boot is rammed down our throats at some time in the future. However, once the system has finished the BIOS boot sequence, we must be prepared to talk with USB devices directly starting with the keyboard so we don't have SMM code contending for our Host Controllers, and mass storage devices so we can run on USB devices while bringing new boxes up. ***In order to use USB host controllers at all, we must disable BIOS/SMM code that emulates AT keyboard.***

We have chosen to support the USB 2.0 standard. This means that we support one EHCI (Enhanced Host Controller Interface) with up to four UHCIs (Universal Host Controller Interfaces) for a total of 8 ports. Our nucleus only uses the UHCIs, leaving the EHCI reset and fallow.

We minimally support USB Hubs at present. There are three reasons for this. Firstly, a basic PC chipset manages plenty of ports without any hubs at all. Secondly, we do not have a representative set of hubs to check interoperability with. Thirdly, after becoming better acquainted with the USB standards and hardware, we've learned that USB itself has instability designed into its standards, which is only exacerbated by adding hubs. Again, the cost of extensive support is not truly justified.

Nevertheless, it turns out that even our USB KVMs look to us like USB 1.1 hubs. So, we have no choice but to minimally support them. Our initial rules will be described later.

Gotchas: 64-bit addressing capability means we have to use the Appendix B EHCI structures rather than those in Section 3.

Device class definitions may be found at [Document Library | USB-IF]

16.1 AVBOX1 (USB 2) Resources

The initial USB support is designed only to serve the AVALUE box, equipped with NM10 (ICH8) chipset. This allows us to limit our exposure to the EHCI controller and, if we must, the UHCIs. The AVALUE box has four UHCI(1.1) and one EHCI:

```
Bus=0 Dev=1D Fun=0 Class=C 3 0 Rev=2 Vendor=8086 Device=27C8 IRQ=B
Bus=0 Dev=1D Fun=1 Class=C 3 0 Rev=2 Vendor=8086 Device=27C9 IRQ=7
Bus=0 Dev=1D Fun=2 Class=C 3 0 Rev=2 Vendor=8086 Device=27CA IRQ=A
Bus=0 Dev=1D Fun=3 Class=C 3 0 Rev=2 Vendor=8086 Device=27CB IRQ=B
Bus=0 Dev=1D Fun=7 Class=C 3 20 Rev=2 Vendor=8086 Device=27CC IRQ=B
```

Behavioral overview of the four UHCI Host Controllers appears in the *NM10 Chipset Data Sheet* section 5.18 and in UHCI Design Guide rev 1.1. Each UHCI Host Controller (UHC) includes a root hub with two separate USB ports each, for a total of eight USB ports. Overcurrent detection on all eight USB ports is supported. The UHCI controllers use the Analog Front End (AFE) embedded cell that allows support for USB full-speed signaling rates, instead of USB I/O buffers. Section 3.1 - 3.3 of the *Universal Host Controller Interface Specification, Revision 1.1* details the data structures used to communicate control, status, and data between software and Chipset. Section 3.4 of that Specification describes the details on how HCD and Chipset communicate via the schedule data structures. The UHCIs are accessible in I/O space and its memory data structure is a single linked list.

An overview of the EHCI Host Controller appears in *NM10 Chipset Data Sheet* section 5.19 and in EHCI Specification rev 1.0. This EHCI supports up to eight USB 2.0 high-speed root ports. The same pins are used for this 480Mb/s communication as for the eight full- and low-speed ports supported by the UHCIs, with selection on each pin made by the chipset's port-routing logic; EHCI ports 0,1 are shared with UHCI0, ..., 6,7 are shared with UHCI3. The EHCI is accessible in memory space and its data structure is separated into Periodic and Asynchronous lists. Because the EHCI registers tell us how many UHCIs and how many ports per are supported, we can handle other geometries but do not support the arbitrary mapping list.

Apparently, if the Configured Flag is set to indicate no EHCI drivers present, the UHCI owns all lines; in this case, connection of either full/low or high speed devices is handled by the UHCI and in that case the high speed device will operate in ***compatibility mode***. Otherwise each port by default belongs to the EHCI and may be handed off to the UHCI only when the EHCI writes 1 to the port owner bit after resetting the port. The port routing logic is what notices real, physical connect/disconnect information; it then informs the port status logic within each host controller. Overcurrent is provided to both host controllers. The port routing logic is powered during suspend so that re-enumeration and re-mapping is not required on exit from sleep state. Legacy keyboard support is provided for both low/full and high speed devices but is mutually exclusive with native USB operation.

In order to detect USB mass storage in the BIOS, "Legacy" USB support and EHCI handoff must both be enabled.

16.2 USB 1 Hardware and Code

16.2.1 PCI Configuration Space Registers

USB Host Controllers are class 0C subclass 03. Programming model identifies HCI type ***(we do NOT support OHCI)***.

It is typical to find one or more USB 1 (UHCI) controllers and a single USB 2 (EHCI) controller on a PC motherboard. This is because for a USB 2.0 controller to talk with full or low speed USB 1 devices, it must have companion USB 1 controller(s) and port routing logic, as specified at considerable length in the EHCI standard.

Class	+8	Serial bus (0C)	USB (03)	00 = UHCI USB 1 10 = OHCI USB 1 20 = EHCI USB 2 30 = XHCI USB 3	(rev – don't care)
		+3	+2	+1	+0

The EHCI Spec says we will find the UHCIs in **th** sequence followed by the EHCI, unless there are no UHCIs in which case the EHCI must be function 0. The UHCIs are mapped to EHCI ports in one of two ways prescribed by the EHCI spec, algorithmically or by a table. It is possible to have more than one EHCI in which it occurs after the next batch of UHCIs, again in **th** order. The code in the sF nucleus only talks to the EHCI sufficiently to disable it; the nucleus uses USB1 until later when we add EHCI support during the 9 LOAD.

16.2.2 UHCI Registers

PCI Base Address Register x20 points to 32 bytes of I/O space. A 16-bit register at xC0 has flags for legacy keyboard mode (the enable bits of this register are ORed and the status bits shared by all four UHCIs). The I/O space registers are as follow (note that while they are documented in bytes, they are writable as WORDs only):

		7	6	5	4	3	2	1	0	
USBCMD	+0	max pkt 64	is config swonly	debug single step	force global resum	enter global susp	global reset	HC reset	RUN not stop	r/w
	+1	res							loop back	
USBSTS	+2			HC halted	HC proc error	Host sys error	resum detect	USB error	USB intrpt	Bits indicate interrupt reasons. r/wc - clear by writing 1s
	+3									
USBINTR	+4				scratch pad	short packet	int on com-plete	resume int	timeout or CRC	r/w interrupt enables
	+5									
FRNUM	+6	Frame number lo								r/w word only, current frame number. Low 10 bits addr 4-byte entry in the frame list during scheduled execution.
	+7						Frame number hi			
FRBASEADD	+8	0								r/w Addr of frame list, on 4k boundary. Low 12 bits are supplied by frame number. 1k entries, 4 bytes each, 4 byte aligned.
	+9	15..12				0				
	+10	23..16								.
	+11	31..24								
SOFMOD	+12		SOF Timing Value (coded)							r/w Modifies frame timing to adjust for clock error, is reset so BIOS value must be retained across reset.
	+13									
	+14									
	+15									
PORTSC0	+16	1	resum detect	line status D-	line status D+	delta port enable r/wc	port is now enabled	connect status change r/wc	device is now present & conn	r/wc, ro, r/w word only
	+17				non global susp	over current ever r/wc	over current now	port reset	low speed device	
PORTSC1	+18	1	resum detect	line status D-	line status D+	delta port enable r/wc	port is now enabled	connect status change r/wc	device is now present & conn	r/wc, ro, r/w word only
	+19				non global susp	over current ever r/wc	over current now	port reset	low speed device	

Here are the values from the I/O registers for the four controllers in an AVALUE box with an AT to USB keyboard plugged into USB port 1 (the USB connector underneath LAN1):

```
F080 20 .IO  (USB ports 0,1)
     F080    C1  0  0  0  4  0 F7  2   DC EB  6 7F 40  0  0  0
     F090    80  0 A5  1  0  0  0  0    0  0  0  0  0  0  0  0 ok
F060 20 .IO  (USB ports 2,3)
     F060    C1  0  0  0  4  0  D  5   34 F4  6 7F 40  0  0  0
```

```
     F070    80  0 80  0  0  0  0  0    0  0  0  0  0  0  0  0 ok
F040 20 .IO  (USB ports 4,5)
     F040    C1  0  0  0  4  0 21  4   84  0  7 7F 40  0  0  0
     F050    80  0 80  0  0  0  0  0    0  0  0  0  0  0  0  0 ok
F020 20 .IO  (USB ports 6,7)
     F020    C1  0  0  0  4  0 D5  0   54 13  7 7F 40  0  0  0
     F030    80  0 80  0  0  0  0  0    0  0  0  0  0  0  0  0 ok
```

Structure of frame list and UHCI behavior in processing it is documented in UHCI spec.

16.2.3 USB 1 Data Base & Operations

The bottom level data base is built during port (but not device) enumeration. We then maintain (dynamic) device enumeration. Finally there is the operational data base, which is USB version independent. This section describes the first two of these levels. All data that exist for each UHCI or port inhabit arrays auto-indexed (in high level use) by the following USER variable:

`UB2P (-a)` contains zero relative port number that is **current** (term used below) for this task.

We set UB2P zero when loading code; it ***must*** be zero when assembling machine code.

16.2.3.1 Port Enumeration Data

Because we only support UHCI for USB 1, there are exactly two ports per controller. We identify these by scanning PCI configuration space. There is a single, contiguous space of 0-relative USB port numbers mapping 1:1 onto the physical ports. We enumerate these two at a time when each UHCI controller is found. Each port has 16 bits of control and status registers devoted to port connectivity, power management and so on. However, the UHCI operates a 2-port Root Hub, so all outbound data is sent in parallel to both ports while incoming data selects the port on which data are seen. *This implies obvious problems in device enumeration and control; not an impressive design, boys.*

The key operational data base for each UHCI is a 4kb descriptor list which is stepped through at 1 kHz. Our four lists are allocated at UB1FL and the appropriate list is addressed when a port is selected by UB2P (see below). Each descriptor in the list points to a queue belonging to the UHCI owning the descriptor list. We support three queues, 0 for transactions like keyboard, 2 for bulk transactions like mass storage, and 3 for control transactions. The descriptor list processes queues 0 and 1 alternating at 500 Hz, queue 2 also at 500 Hz after handling queue 0 transactions.

The following variables and arrays have to do with enumeration and port selection.

`m#ubp (-n)` returns max number of USB ports.
`#ubp (-a)` holds the number of UHCI ports discovered.

The following arrays, auto-indexed by UB2P, hold USB1 (UHCI) information:

`UBIRQ (-a)` contains IRQ number for current port's interrupt.

`UBPIC (-a)` contains PIC base register address for current port.

`UBIMK (-a)` contains PIC mask bit for current port.

`UB1TH (-a)` contains th value for indexing this port's UHCI with nPCI .

`UB1CR (-a)` contains I/O address for this UHCI's registers.

`UB1PR (-a)` contains I/O address for 16-bit port status & control.

`UB1Q0 (-a)` returns address of 500 Hz queue for transactions.

`UB1Q1 (-a)` returns address of 500 Hz queue for bulk.

`UB1Q2 (-a)` returns address of 500 Hz queue for control.

`UB1FL (-a)` returns address of 4k descriptor list for UHCI zero.

The following variables are set while discovering the USB2 EHCI controller:

`#ube (-a)` zero if no EHCI; 1 if EHCI found but not being used; -1 if found and being used..

`ub2th (-a)` th value for indexing the EHCI.

`'UB2CAP (-a)` Contains memory address of EHCI controller capabilities area.

`'UB2OP (-a)` Contains memory address of EHCI operational area.

16.2.3.2 TD Queue Configuration

We've set this up as simply as feasible. Each 2-port UHCI has its own **UB1Q0** ("interrupt" protocol), **UB1Q1** ("bulk" protocol), and **UB1Q2** ("Control" protocol). The UHCI does not have any explicit way to know which protocol a particular transaction uses. Each of the 1024 1ms descriptor list entries in an UHCI's **UB1FL** list points to its **UB1Q0** which links to **UB1Q2** and then **UB1Q1** in order. **UB1Q1** links to itself to facilitate "bandwidth reclamation".

16.2.3.3 Port Enumeration Operations

USB2 enumeration and system initialization in the nucleus fully sets the UHCI controllers up for low and full speed operations, but only treats with the EHCI controller as necessary to make it inert and leave all ports under UHCI control. This gets us to the point of operating a USB keyboard and mass storage device but none at high (USB2) speed. The component functions are as follow, and their data base is protected by facility **USBCTL** :

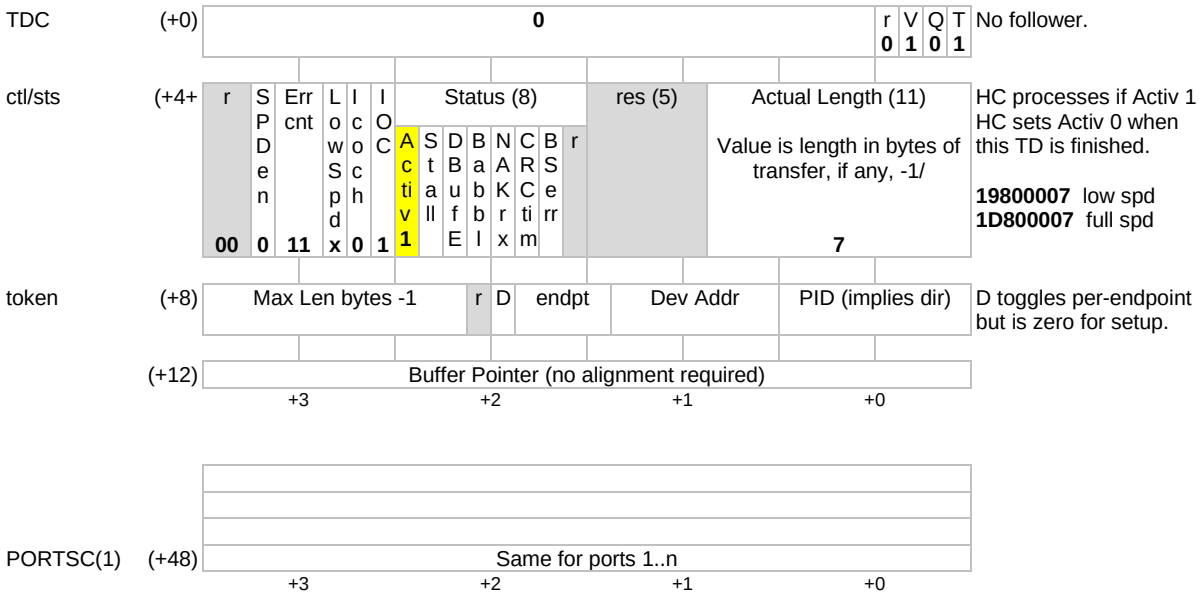
`?ubhcis` finds, and builds enumeration tables for, UHCIs. Finds EHCI if any and builds its data too.

+1EHCI does the first phase of EHCI initialization to make the interface inert and give control of all ports to the UHCIs. Steps are: Claim from BIOS; Halt EHCI operations; Reset the EHCI.

+UHCI initializes the UHCIs and activates them with nothing to do. Disables/clears AT emulation interrupt control and status in PCI registers. Builds all primary queues (3 per UHCI) with nil pointers. Sets and populates Frame List with pointers to queue headers according to the schedule described above.

16.2.3.3.1 Control Transactions

These consist of writes, reads and ACK/NAK. The Transaction Descriptor is as follows:



16.2.3.4 Handling of Hubs

It seems that hubs carrying USB1.1 are a necessary evil because our USB KVM switches look like hubs. Therefore we have minimal support for them in the nucleus.

Hubs are sent bus-reset, brought to connected, initial descriptor read done, address assigned, device and config descriptors read, and config 0 selected by **UBENUM** as are all other devices on our root ports. Here are device and config descriptors for two example hubs:

```
ucDevd 18 XD  (A  Linksys USB1.1 only)
    000   12 01 10 01 09 00 00 08 51 04 46 20 25 01 00 00
    010   00 01
ucDevc DUP 2+ U@ XD
    000   09 02 19 00 01 01 00 E0 00 09 04 00 00 01 09 00
    010   00 00 07 05 81 03 01 00 FF

ucDevd 18 XD  (B  USB KVM switch)
    000   12 01 00 02 09 00 00 40 40 1A 01 01 11 01 00 01
    010   00 01
ucDevc DUP 2+ U@ XD
    000   09 02 19 00 01 01 00 E0 32 09 04 00 00 01 09 00
    010   00 00 07 05 81 03 01 00 FF

Both return SIMILAR Hub Descriptors:
    000   09 29 04 00 00 32 64 00 1E  (A has FF for last)
```

Hub A max packet 8, B=64. Hub A=USB 1.1, B=2.0. B uses 100 mA, A assumes that's the default. Both say four ports, 100 ms power on - power good time, and 100 uA controller current.

16.2.3.5 Device Enumeration Data

We begin identifying a device by reading the *device descriptor* into **ucDevd** . Although this descriptor contains fields for class, subclass and protocol, in practice we find even for simple devices like keyboards and USB flashes that these fields are zero meaning they will be found in an *interface descriptor*. To get one of those we must read and parse a *configuration descriptor,* into **ucDevc** , which includes *interface* and *endpoint* descriptors. In practice it seems that all of this generality is not used, at least for devices like keyboards and mass storage which must be understood by a BIOS. Nevertheless, we see differences even in these cases.

When a task owns **USBCTL** it may read the current configuration data by calling **ubid1** with the desired port number. This definition is chatty, so you must make a version of your own if it's to be silent.

On completion of USBID which finds all currently known devices, the following variables define results:

ubKPT (-a) Contains port number for keyboard, -1 if none connected.
ubKAD (-a) Contains address assigned to keyboard
ubKEP (-a) Contains endpoint for keyboard input
ubKMX (-a) Contains max packet size for keyboard
ubKI# (-a) Contains keyboard interface number

ubSPT (-a) Contains port number for SCSI device, -1 if none connected.
ubSAD (-a) Contains address assigned to SCSI device
ubSiE (-a) Contains inbound endpoint
ubSoE (-a) Contains outbound endpoint
ubSiM (-a) Contains inbound max packet (assumed 64 in code)
ubSoM (-a) Contains outbound max packet (assumed 64 in code)
ubSI# (-a) Contains SCSI device interface number

The following is an example of the OPERATOR screen display while enumerating USB interfaces and devices; in this system there are four UHCIs, one EHCI, witha flash in port 0 and keyboard in port 4. If keyboard or flash are missing, a line saying so with asterisks surrounding appears before **hi** . This is illustrated under ========= below.

```
sF386/ISA.5e1 04/17/22
USB Scan 0 808627C8 UHCI/B  1 808627C9 UHCI/7  2 808627CA UHCI/A
         3 808627CB UHCI/B  4 808627CC EHCI
    EHCI Claim: -1 01000001 Success Halt: Reset:    0 UHCI
Stop:Reset:Run:
    2 UHCI Stop:Reset:Run:  4 UHCI Stop:Reset:Run:  6 UHCI
Stop:Reset:Run:
0 MaxPkt=40 Addr: A Descr: Cfg 0 Lng= 20 Is drive; use.
4 MaxPkt=8 Addr: E Descr: Cfg 0 Lng= 22 Is kbd; use.

============================================================
*** NO KEYBOARD FOUND! ***
0 PCI-ATA 0th: 808627C0  8F 7
1 PCI-ATA 1th: 197B2368  85 7
Ports: 2
**** NO USB SCSI MEMORY ***
```

16.2.3.5.1 Keyboards

Here are two keyboard examples: An old Dell L100 and a relatively recent Redragon.

```
66 6 UBENUM MaxPkt=8 Addr: 42 Descr: Cfg 0 Lng= 22
    2010200   2003413C    8000000    1100112  < 66058C >  ________<A_ ____
     66057C     22E45B          0        100  < 66059C >  ________[d"_|_f_
    1030100      40923   A0040101     220209  < 6605B8 >  __"____ #_______
    8038105    7004122    1000110   21090501  < 6605C8 >  ___!____"A______
          0          0          0       1800  < 6605D8 >  _________________ok
11 1 UBENUM MaxPkt=40 Addr: B Descr: Cfg 0 Lng= 3B
    2010109   50040C45   40000000    2000112  < 66058C >  _______@E__P____
     66057C     22E45B          0        100  < 66059C >  ________[d"_|_f_
    1030100      409C8   A0000102     3B0209  < 6605B8 >  __;____ H_______
    8038105    7004F22    1000111   21090001  < 6605C8 >  ___!____"O______
      11121    9000201    3010001    4090100  < 6605D8 >  _____________!___
          0      10040    3820507     712201  < 6605E8 >  _"q_____@_______ok
```

Looking at these we see cfg(2), intfc(4), endpt(5) and HID(x21):

Posn	Name	Dell	Red	Remarks
Cfg+4	#interfaces	1	2	
+5	Config Value	1	1	
+8	Max Pwr	35	200	70mA vs 400mA must be all those LEDs
Intfc+2	Interface number	0	0	
+3	Alt Setting	0	0	Meaning none
+4	#endpoints	1	1	
+5	Interface class	3	3	HID
+6	Subclass	x01	x01	Boot Interface
+7	Protocol	x01	x01	Keyboard
HID+2	Spec release	x0110	x0111	1.10, 1.11
+4	Country Code	0	0	
+5	#HID class descr follow	1	1	
+6	Rpt descr type	x22	x22	
+7	Rpt descr length	x41	x4F	
End+2	Endpt Address	x81	x81	IN endpoint 1
+3	Atributes	x03	x03	Interrupt
+4	MaxPkt for endpt	8	8	
+5	Poll interval ms	x18	x01	Who implements this?
Intfc+2	Interface number		1	

+3	Alt settings		0	
+4	#endpoints		1	
+5	Interface class		3	HID
+6	Subclass		x01	Boot Interface
+7	Protocol		x02	Protocol Mouse
HID+2	Spec release		x0111	
+4	Country Code		0	
+5	#HID class descr follow		1	
+6	Rpt descr type		x22	
+7	Rpt descr length		x71	
End+2	Endpt Address		x82	IN endpoint 2
+3	Attributes		x03	Interrupt
+4	MaxPkt for endpt		x40	
+5	Poll interval ms		1	

16.2.3.5.2 USB Mass Storage

Here are two flashes: An old 1 GB Patriot/Kingston and a recent SanDisk 256GB.

```
44 4 UBENUM MaxPkt=40 Addr: 2C Descr: Cfg 0 Lng= 20
    2010110   1A0013FE   40000000    2000112  < 66058C >  _______@~_______
     66057C     22E45B          0        103  < 66059C >  ________[d"_|_f_
    6080200      40964   80000101     200209  < 6605B8 >  __ _____d_______
       4002    2050700     400281    5070050  < 6605C8 >  P_____@_____@__ok
55 5 UBENUM MaxPkt=40 Addr: 37 Descr: Cfg 0 Lng= 20
    2010100   55830781   40000000    2100112  < 66058C >  _______@___U____
     66057C     22E45B          0        103  < 66059C >  ________[d"_|_f_
    6080200      40970   80000101     200209  < 6605B8 >  __ _____p_______
       4002    2050700     400281    5070050  < 6605C8 >  P_____@_____@__ok
```

Looking at these we see cfg(2), intfc(4) and endpt(5):

Posn	Name	Patriot	SanD	Remarks
Cfg+4	#interfaces	1	1	
+5	Config Value	1	1	
+8	Max Pwr	100	112	
Intfc+2	Interface number	0	0	
+3	Alt Setting	0	0	Meaning none
+4	#endpoints	2	2	
+5	Interface class	8	8	Mass Storage
+6	Subclass	6	6	SCSI Transparent Command Set
+7	Protocol	x50	x50	BBB (Bulk-only transport)
End+2	Endpt Address	x81	x81	IN endpoint 1
+3	Atributes	x02	x02	Bulk
+4	MaxPkt for endpt	x40	x40	
End+2	Endpt Address	x02	x02	OUT endpoint 2
+3	Attributes	x02	x03	Bulk
+4	MaxPkt for endpt	x40	x40	

16.2.3.6 Device Recognition in the Nucleus

Because the device manufacturers have kept their configurations simple for the benefit of BIOS in these two cases, simple recognition suffices for us. After enumerating a device, the nucleus code parses the first configuration to find device class. If we find class/sub/proto of 3/1/1, we have identified a keyboard and fire it up by setting the first configuration we looked at. If the class etc are 8/6/x50, we have found a straight SCSI bulk-only flash or hard disk (such as the 2TB WD My Passport).

During recognition, each port is shown as keyboard, drive or nothing. After that, if the class has already been found, we display "EXTRA" and don't look further. If not extra, we check for usable endpoints and, if all is kosher, we say "use."

All of the USB discovery, setup, and device identification generates a lot of verbose crap before "hi" on boot, but it is important until experience proves our code is solid enough to take chances with it. We have also left ?CREATE disabled through the 9 LOAD temporarily so that this discovery stuff will be visible after HI (for autoloading during development).

16.3 USB 1.1 Interrupt and High Level Code

We support low and full-speed keyboards in boot mode only, as noted below. The nucleus does not support hot plugging, so any USB devices (such as keyboard and mass storage) to be used after boot, and before the 9 LOAD has completed, must be plugged into USB sockets before booting. Additionally, we do not support external hubs at all due to system stability concerns.

Interrupt code is trickier than usual. For the UHCIs, the interrupt must be shared with other devices, including other UHCIs; that is not a big or new problem because an interrupting UHCI can be identified by its status register. However, each UHCI has two USB ports, and the UHCI does not indicate which port (or both) is causing the assertion of interrupt. Actually, this is symptomatic of a serious design flaw in the specification of USB root hubs: When sending an address assignment to a USB device using the default USB address (0), it is broadcast on all ports of the hub, so that if two or more USB devices on the ports of that hub are in the unaddressed state, ***both*** will assume the given new address leading to a failure mode that is not unambiguously identifiable. Oops, guys. At any rate, we will have to compensate for the first of these defects with our own state machine, while for the second we don't have a reasonable solution.

The interrupt code for the UHCI is far less efficient than that of most PCI peripherals, and when we come to the USB keyboard it is really ridiculous compared with for example the AT keyboard: In addition to the UHCI bruhaha, we have to implement rollover and repeating keys. Both are sons of bitches.

We apparently do not get an interrupt on port status change, which is a load of crap.

Each action that can cause an interrupt is registered in a list of activity tables linked from **uhHEAD** . When an interrupt is received from a UHCI controller, we scan this table and, for each action table whose **Pending** cell is zero and whose **Port#** is on the interrupting HCI, we store the UHCI status value into **Pending** and call the code provided with the address of the activity tale in register **I** and all registers, including **I** scratch except **S** , with interrupts prevented. It is the duty of the code to verify from its transaction descriptor list that this activity caused the interrupt, because the UHCI does not tell us which port caused the interrupt; in fact, per the specs, there is nothing to prevent activity on the non interrupting port to continue and, itself, interrupt us. If the code is satisfied that the activity in question did not cause the interrupt, it may simply zero the **Pending** cell again before returning.

Activity			
	+0	^next activity	Zero if none.
	+4	Pending	Zero if pending int. NZ last ubif sts, neg if handled..
	+8	Port#	Zero-relative UHCI port number in use.
	+12	^code	Code to determine if int, awaken task, clear pending
	+16	^status	Task to awaken if any
	+20	^TD List	Pointer to first TD in activity's list (EHCI ptr to TD queue hdr)
	+24	^queue	Addr of pointer linking to our TDs (EHCI not used)

16.3.1 Keyboard Input Code

ubkdi is a single TD with opcode **IN** attached "horizontally" to queue 0, initialized by **/ubkdb** . This TD is continually seeking to read the 8 byte keyboard message into the buffer **ubKIN** , polling (by the UHCI in the background) at about 1 kHz. Normally the TD is showing a NAK (status byte x88) and is not producing an interrupt; when the keyboard does respond, we get an interrupt and the data appear in **ubKIN** . To resume polling we must toggle the **D** bit in the token field of the TD and turn on the **ACTIVE** bit in the TD's CSR. Thus no fiddling with the UHCI is required after it's been set up; just catch an interrupt, clear it, check **ubkdi** status byte, if 0 then grab **ubKIN** , toggle D and turn on ACTIVE bit again. This part can easily be handled in interrupt code, as can the key processing itself. The main complications for key processing are roll-over logic and key repeat, both functions removed from the keyboard processor and laid in our laps.

We use the keyboard in **boot mode** because dynamically parsing and implementing a report descriptor would be needlessly Byzantine and an obscene amount of code, rivaling the whole FORTH nucleus. In this protocol, the 8 bytes we receive from the keyboard on each non-dry poll (i.e. something changed) are as follow:

modbits	reserved	code/0	code/0	code/0	code/0	code/0	code/0

The modbits 7..0 are: Right (GUI, ALT, Shift, CTL), Left (GUI, ALT, SHIFT, CTL). The codes are from the **Key Codes** usage page and are ***not*** the same as those returned by the AT keyboard, but this is a blessing because it removes some of the AT kludges; all codes are a single byte, and there are no ambiguities. Codes appear in no particular order and are not justified to either end (there may be zeroes embedded in a set.) They merely indicate which non-modifier keys are currently depressed. Note that **Num Lock** is treated as an ordinary key, not as a modifier; it and **Caps Lock** must be handled specially.

This message is handled by rollover logic described below, which produces a key code to process if a new key has been pressed (releases are not events from that logic nor are flagellations of modifier keys). The left and right modifier bits are ORed together, shuffled from **asc** to **sca** order and stored into **SHIFTS** (high byte of is locks **cns** .) We then invoke kbd which as a subroutine does what the AT keyboard interrupt did. (Note LEDS on Dell kbd show **ncs** order).

Our interface with task code is, insofar as practical, identical with that of the AT keyboard.

'key contains zero or the ASCII value of a key event.

'key 4+ contains zero or the address of a task to awaken when a key event occurs.

'KEY normally contains the address of key which simply returns an ASCII value when one is available. It may vector off to enhanced code for replacing function keys with character strings.

?key (-t) returns true (an ASCII character) if one awaits processing.

key (-c) returns the next ASCII character (including zero) from the keyboard.

Key (-c) returns the next ASCII character to be processed, which may be from replacement of function keys by character strings.

(EXPECT) (anf) for OPERATOR console, with full capabilities..

(CR) for OPERATOR console, respects scroll lock.

16.3.1.1 Rollover Logic

16.3.2 LED Output Code

These keyboards do not inform us of a separate output endpoint, so for a universal solution we must control the LEDs using a three-TD **SET_REPORT** control sequence at **ubkdo** on endpoint zero (also initialized by **/ubkdb**) and also linked horizontally to queue 0. The operation is to set new value in **ubKLED** , reset **ACTIVE** bits in the three TDs and let it rip. We actually maintain the pointer from ubkdi to minimize overhead. Output is done only by the keyboard input interrupt code, with the sole exception of keyboard initialization code which uses >leds to establish initial (default) value.

16.3.3 Mass Storage Devices

The principal compelling motive for supporting USB was to enable use of USB flashes to back up and restore disk on systems such as AVBOX1 that don't really offer any other physical media. In addition, we support these devices for system "disk" that can be booted and can run a full system while exploring new hardware, installing systems, studying crashes and so on.

Our support is for specific things, and excludes for example typical USB floppies which use a different protocol. We support devices with these characteristics:

- Maximum Packet Size 64 octets (practically universal).
- Sector/block sizes of 512 or 1024 only.
- Max transfer 1k bytes (one block) (measurements show little or no advantage to larger transfers)
- USB Bulk-Only Transport Specification.
- SCSI Reduced Block Command Set

For our purposes, 32-bit sector numbers (2 Gigablocks, 2 terabytes) of mass storage are more than sufficient, and indeed at low and full USB speeds it would take a very long time to read or write that much; therefore, we can live with the 10-byte command blocks supported by all such devices for reading and, most likely, all for writing (how else does one write them to have something to read?). This is the same as we have done for all other SCSI Host Adaptors; type 0 commands (6 bytes) and type 1 (10 bytes).

The **hh** byte used in prior SCSI implementations does not exist here, and SBC-3 which applies to the USB-SCSI devices does not seem to make any use of the "control byte" **yy** . The "service action" field **xx** is defined often. Expected data length must be accurate because status will land in a data buffer if it's too long.

16.4 USB1 Integration with sF System

On boot, we only use the first USB1.1 compatible keyboard we find, and, if present, the first USB1.1 compatible bulk-only SCSI memory device. The memory device may be used to boot, and, to be used immediately, both must be plugged in before the BIOS begins considering peripherals preparatory to booting us

In the 9 LOAD we add features such as dynamic discovery of device disconnection and reconnection, but the system will always use one and only one device of each class (keyboard, storage).

16.4.1 Keyboard Integration

This keyboard looks to the sF environment like the original AT keyboard, including the low level **'key ?key** and **key** usage. The system survives if no keyboard, but **OPERATOR** is useless. Code translation is the same except that we now decode F11 and F12. The ESC functions are preserved: Ctl-ESC is **RELOAD** , Ctl-Alt-ESC is **COLD-BOOT** , and Alt-ESC does an illegal opcode inside the keyboard interrupt. Here is an example of reading a panic dump's stack to unravel the victim's state at the moment of the ALT-ESC:

```
CSTACK
Dump taken 4/18/22 22:16:16

Stack ptr as of trap 813492C >ha call 8135DEB
Regs  W-I-R-S / 3-2-1-0
       1363F        8954     8134B8C     8134954
     8134BA8        84A8           6           1
Trap 6 Opcd
PC/CS/Flg:  868C 8 2 ok

HEX 8134BA8 1 DUMP
    81349A8    8134998      8FFFE   6452D7FF

81349A8 8134954 .stk
   VALUE  Word.............  Ofs(hex)  Ofs(dec)
    868C  Keys                    29C       668 \
       8  None                      8         8  > Trap frame
       2  None                      2         2 /
    891F  ukREP                    D3       211 \
    6520  uhis                    -8        -8  > Inside int
    8954  ukACT                    14        20 /
    55B8  COUNTER                  -4        -4 \  W  PUSHA
    5650  MS                       28        40  | I  at start
 8134B8C  Mole                  28F8     10488  | R  of int
 813498C  Mole                  26F8      9976  | S  (Victim's
 8134BA8  Mole                  2914     10516  | U   registers)
     117  None                   117       279  | 2
       0  None                     0         0  | 1
      7A  None                    7A       122 /  0
    66CE  uhics                  156       342  Call from portal
    55CE  COUNTER                 12        18 \  <--- THE VICTIM
       8  None                      8         8  > Int frame
     246  None                   246       582 /
 4611F5C  .SLEEPER           3FB6BD0  66808784
 4611F5C  .SLEEPER           3FB6BD0  66808784
       0  None                     0         0 ok
```

16.4.2 Mass Storage Integration

The sF nucleus has been updated, for PCI-ATA disk subsystem only, to accommodate USB SCSI disk devices. This has required extending the definition of the **DISKS** table, and in replacing **SEL** with **USEL** due to changes in the stack effects. We haven't arranged to combine USB-SCSI with AHCI-SATA because we don't have a box of the latter type that has simple EHCI/UHCI USB support.

For USB 1 devices, the kind field (+8) in a **DISKS** entry is 2 for USB 1 and the byte at +4 is logical unit number; presently only zero is supported. **BULK** only does 1-block operations on anything at or above the first USB 1 unit in **DISKS** .

We have tested this code with a number of flash devices, all USB 2 or higher, and hard disks such as a 2TB Western Digital My Passport Ultra.

FLUSH!! does not issue synchronize cache because the USB flashes don't seem to support that operation.

16.4.3 Inseminating Systems from USB Flash

In the past, we used floppies and, more recently, floppies with URAM boots so we could employ USB floppy drives. We have also used flash with URAM boot, made more torturously with intervention from sF-Win32. The only problem with this method is that while we can change the code in the RAM image, it will be overwritten next boot.

Now we can direcly write boot flashes and, on relatively recently made boxes, we can talk to USB after booting via USB and, thus, have a full read/write (USB) system available for exploring new hardware and inseminating it. On more recently made boxes and/or BIOSes, this works fine; boot from flash, **`HOME 6000 DRIVE 9 LOAD`** with our traditional disk layout. *We have yet to make a means to configure a system with USB and IDE or USB and nothing.*

Some older BIOSes don't seem to support handing the USB controllers off to us when USB has been used to boot. Our T1 routers parsit and jard , machines with "legacy" ISA bus support, are examples. We have not yet found out exactly what is going on. For now, on such boxes as these, we have to make a USB URAM boot that uses AT keyboard and IDE disk. This will work well enough to say **`HOME -2 DRIVE 9 LOAD`** and then copy system from -2 DRIVE to the main disk. These old systems don't seem to have a hand-off problem when we boot from the IDE disk in the box, so once there you can target a new system with USB keyboard support and boot it thereafter from IDE disk.

16.5 USB 2 Hardware and Code

During the 9 LOAD, we add USB support (for the EHCI) along with dynamic device enumeration/recognition as applies to mass storage. The purpose for doing this is one thing: Vastly faster disk backup and restore using USB flashes as the removable media, which justifies allowing the SCSI operations to vector on the delivery system.

EHCI is simpler to use than are the UHCI controllers; large transfers (max is between 16 and 20k depending on alignment of the buffer) are much simpler, and interrupt handling is less ambiguous and therefore simpler. To avoid having to duplicate much of the keyboard control, though, we force keyboards onto the UHCIs.

Here's the EHCI hardware synopsis:

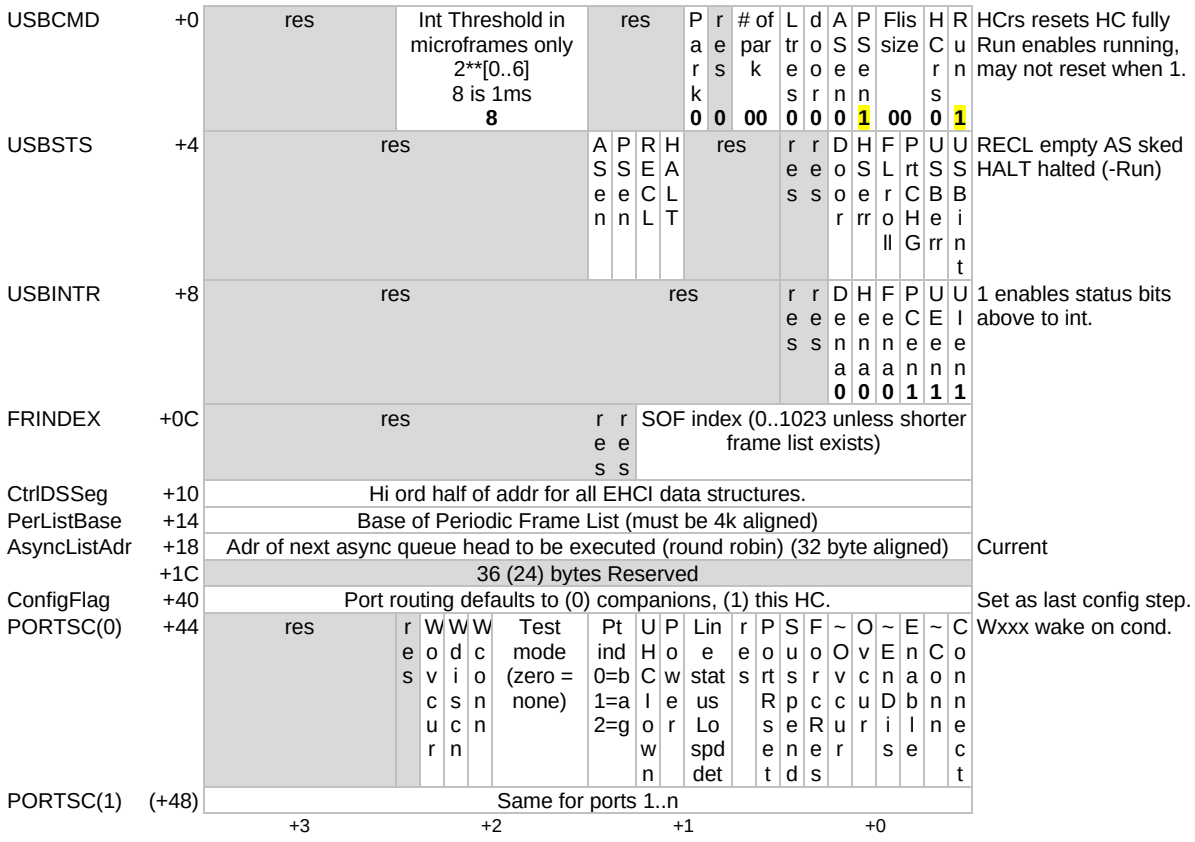
16.5.1 EHCI Registers

PCI registers in AVBOX1 on boot (with no SMM keyboard emulation, USB HCs halted):

```
HEX 0 1D 7 .PCI
   0  27CC8086   2900006   C032002         0
  10  DFF05000         0         0         0
  20         0         0         0  27CC8086
  30         0        50         0       10B
  40         0         0         0         0
  50  C9C25801         0  20A0000A         0
  60   1FF2020         0         1      2000
  70  3FCF0000         0         0         0
  80         0        11         0         0
  90         0         0         0         0
  A0         0         0         0         0
  B0         0         0         0         0
  C0         0         0         0         0
  D0         0    FFAA00    FF00FF  88000020
  E0         0    6DB6DB         0         0
  F0   9008000    408588     20F86  2002170A
```

PCI register x60 (byte) is USB release number, x20. x61 is frame length adjustment analogous to SOF timing value in UHCI. PCI Base Address Register x10 points to 1k bytes of memory mapped ***non-prefetchable*** registers. The first five cells are "Capability" registers; the next set, at relative offset in the 0th byte of capability registers but at x20 in the NM10 chipset, are "Operational" registers. The last item in the Operational Registers is an array of port CSR's whose length is determined by the low order nibble of the Structural Parameters register. ***All accesses to this register array must be in 32-bit cells.***

'UB2CAP @	+0	Version **x0100**				res		Offset to Opregs, bytes **x20**								
hcsparams	+4	res		Debug Port 1-relative **(1)**	res	Pind	# of compan USB1.1 ctrlrs **(4)**	# of ports per compan **(2)**	Rtbl	res	res	PwrC	# of ports implem >0 **(8)**			structural, RO
hccparams	+8	res		res			EECP ptr to extended capabs in PCI cfg space **x68**		Isoc sked thresh **7**			res	Park **0**	PFlis **0**	64b **1**	capability, RO Park – async park Prog Frame List length 64bit addr structures
portroute	+0C	rte7	rte6	rte5	rte4	rte3	rte2	rte1		rte0						Routing table, used iff rtbl bit is set above
	+10	res	rte14	rte13	rte12	rte11	rte10	rte9		rte8						
'UB2OP @:	x20															



It appears we can completely disable the USB2 functions by disabling all the interrupts (storing zero to USBINTR) and storing zero into ConfigFlag. That would be an excellent goal for nucleus support of USB.

Note that port 0 is also the debug port, and there is a bit in its registers that can force ownership by the EHCI regardless of the other ownership controls. Perhaps resetting of the EHCI is the best course of action.

Legacy support can be turned on and controlled using PCI configuration registers, however it is better to simply comply with the standard for taking possession of all the USB hardware. Unfortunately while EHCI has a defined protocol for taking control I have been unable to find a corresponding protocol definition for the UHCIs. Will create mechanism to perform the EHCI protocol and see if there is any visible state change on the UHCIs.

16.5.2 USB 2 Data Base & Operations

During initialization in the nucleus, the EHCI controller was enumerated so it could be disabled, resulting in these data:

1#ube (-a) zero if no EHCI; 1 if EHCI found but not being used; -1 if found and being used..

ub2th (-a) th value for indexing the EHCI.

'UB2CAP (-a) Contains memory address of EHCI controller capabilities area.

'UB2OP (-a) Contains memory address of EHCI operational area.

16.5.3 Port Management and Data

16.5.4 TD Queue Configuration

16.6 USB 2 Interrupt and High Level Code

Because the architects of the EHCI did not create a way to queue up a sequence of bulk operations on different endpoints, as is required for USB-SCSI (it makes one wonder if they read the manuals on any of the USB protocols), we were forced to make three queue headers (command, data, status) with a single TD for each. The interrupt code expects offset x48 from queue header to be either the address of the next queue header to schedule, or zero; and, so long as status is good for an operation, will schedule the next if any.

16.7 USB 2 Integration with sF System

With USB2 support, there are now two distinct environments to be aware of; that supported by the nucleus at boot time, and that supported after 9 LOAD. The former is more restrictive than is the latter.

16.7.1 USB2 at Boot Time

At boot time, the system supports one keyboard and one USB-SCSI storage (in PCI-ATA and any other environments with which the USB-SCSI had been integrated) using USB 1.1 (low or full speed modes) with the UHCIs. The USB-SCSI device must be plugged into a root hub port (i.e. those on the back of the computer), not on a proper hub. The keyboard must be able to work at low or full USB 1.1 speeds and may be plugged into a root hub port or into a single layer of external hub. The hub is only used in USB 1.1 mode.

These devices must be plugged in before booting the system, and the nucleus does not support any form of hot plugging them after boot.

16.7.2 USB2 after 9 LOAD

This code adds support for an EHCI controller and will use an USB 2 compatible USB-SCSI storage device at high speed. All other devices and hubs are "given" to the USB 1.1 (UHCI) controller. Dynamic (hot) plug and unplug of the keyboard and storage device are supported. To see what is going on, use **.USBDYN** as in the following which shows what happened during 9 LOAD. On activation of the EHCI controller, it takes control of all ports so the first thing we see is the removal of two devices from the UHCI controller to EHCI custody. Then the EHCI controller sees on port 0 a USB2 compatible flash device that it sets up for use at high speed (we don't have any flashes old enough to be USB 1.1 only.) Next the EHCI sees a hub on port 6 which it sends to the UHCI, where it's enumerated and we find a keyboard.

```
.USBDYN
21:15:09 UHCI port 0 removed.
21:15:09 UHCI port 6 removed.
21:15:09 EHCI port 0 inserted.
USB2 Port 0 MaxPkt=40 Addr: 10 Descr: Cfg 0 Lng= 20 Is drive; use.
21:15:09 EHCI port 6 inserted.
USB2 Port 6 MaxPkt=40 Addr: 70 Descr: Cfg 0 Lng= 19 Is Hub. To UHCI. To UHCI.
21:15:10 UHCI port 6 inserted.
USB Port 6 MaxPkt=40 Addr: 70 Descr: Cfg 0 Lng= 19 Is Hub; use 4
Hub Port Lo2 MaxPkt=8 Addr: 72 Descr: Cfg 0 Lng= 22
Hub Port 3 MaxPkt=40 Addr: 73 Descr: Cfg 0 Lng= 3B Is kbd; use.
```

The EHCI dynamics send everything but USB-SCSI devices, including hubs, to the UHCIs. The UHCIs, in turn, will enumerate things on a newly plugged in hub, but do not recognize there anything but keyboards; and there is no dynamic support for things plugged into or removed from hubs after the hub itself has been enumerated. This works for USB KVMs that move a single hub from system to system; when the KVM is switched to sF, we discover the hub and enumerate it for keyboards. When the KVM switches away, the hub "unplugs" to be rediscovered when it reappears.

17. Specific Platform Support

This section describes code that we are treating as being platform-specific because it addresses capabilities described in the documentation (or lack thereof) for chips and chipsets in supported platforms, rather than capabilities derived from standards that all such devices can fairly be expected to support. The fact that something works on one platform does not mean it will, or will not, necessarily work on another.

17.1 AVALUE EES-CDV "AVBOX1"

This system, with no moving parts, is strongly supported by ATHENA in that our company owns 20 of them and has several in use 24x7, replacing standard "tower" PCs, in both Windows (7 and 10) and native system use (such as our mail system.) It is equipped with the Intel Atom D2550 Dual Core 1.86GHz CPU. The motherboard includes an Intel NM10 Express chipset, 2 GB (can have 4) of DDR3 memory at 1066 MHz, a Nuvotron W83627DHG-P I/O chip, a watchdog timer, and a Realtek ALC892 audio chip. The Intel complex includes Cedarview integrated graphics capable of 1920x1200 on VGA and (concurrently) 1920x1080 on HDMI at 60 Hz. In addition it comes with two 82574L GbE interfaces. A mini-PCIe socket supports mSATA. A JMicron JMB386 PCIe PATA Host Controller runs a socket for Compact Flash types I or II. Connections are provided for a 5V only SATA mass storage device. We typically configure these boxes with a quality SATA SSD and a good mSATA SSD as an internal backup and spare boot device.

AVALUE This CONSTANT is defined in block 2 and **its value is 1 for AVBOX1 systems**, enabling the loading and interpretation of coding specific to this box, such as interfaces for hardware features, and certain items included in the email reporting facility.

17.1.1 Peculiarities

This box does not support the old kludges dating from the PC-AT in which the numeric processor asserts IRQ13 for exceptions and uses I/O port xF0. When AVALUE is nonzero, we enable the floating exception through control registers, and set to use trap x10 for this purpose.

17.1.2 Nuvoton Chip Capabilities

Code in blocks 78..80 reads temperatures and voltages using this chip; these are monitored and reported in routine status email messages.

17.1.3 Power Management

For BMC use in the new backup paradigm, the backup box needs the ability to power up and boot on a timed schedule, to recognize when it has done so and, in that case, to synchronize with the floor system automatically, powering down when that has been done. We use the ACPI capabilities of the NM10 chipset to achieve this. The low level functions are in block 77. These must be used with some care; if we enable the RTC alarm and it happens while the system is up and running, the system hangs in what appears to be an SMI interrupt loop. On RTC alarm awakening and reboot, we find alarm disabled in both RTC register 11 and ACIP register 2. If we awaken early due to power button, the same occurs and if the time arrives while still up and running we do not die. We cannot definitively tell which of these caused the power-up and reboot (evidence is not left visible in the ACPI registers) so that will have to be resolved otherwise.

)ACPI (n-a) Returns the I/O address of the given byte offset in ACPI register space.

?RTCA (-t) NZ if an RTC alarm has occurred, whether or not it resulted in a wake-up. We have not been able to see this value nonzero after any sequence of actions; it may be impossible.

/RTCA turns off the indication of RTC alarm tested by ?RTCA.

+RTCW enables wake-up on RTC alarm.

-RTCW disables wake-up on RTC alarm.

POWER-DOWN Forces system to state S5, which is power down.

!RTCW (hh:mm) Arms wakeup on RTC alarm at the given time, adjusted later if necessary so that the actual wakeup time will be at least five minutes in the future. Having armed the wakeup, we *must* power down to avoid the hang.

DOWN-TILL (hh:mm) Safely powers the system down, to be fired back up by the power button or by hitting the given wake-up time, which is forced to be at least five minutes in the future.

17.2 AVALUE EPC-SKLU "AVBOX2"

This system, with no moving parts, is supported by ATHENA. It is equipped with a 6th gen i3-6100U Dual Core 2.2GHz CPU. The motherboard includes an Intel 6th Gen PCH, 4 GB (can have more) of DDR3 memory at 1066 MHz, and an EC ITE IT8528E I/O chip for which we haven't been able to find documentation. In addition it comes with two GbE interfaces, one an i211AT and the other an i219LM. Mass storage is only provided by a single AHCI SATA Host Bus Adapter, with three ports (M.2 2242B, 5V SATA SSD, and one mSATA card). Unlike the AVBOX1 with its Compact Flash, this one lacks a read/write mass storage device we support that can be used to boot and run in lieu of the SATA devices. We typically configure these boxes with a quality SATA SSD and a good mSATA SSD as an internal backup and spare boot device. The M.2 may be wonderful, but is not exactly easy to access physically.

AVALUE This CONSTANT is defined in block 2 and **its value is 2 for AVBOX2 systems**, enabling the loading and interpretation of coding specific to this box, such as interfaces for hardware features, and certain items included in the email reporting facility.

17.2.1 Peculiarities

This box does not support the old kludges dating from the PC-AT in which the numeric processor asserts IRQ13 for exceptions and uses I/O port xF0. When AVALUE is nonzero, we enable the floating exception through control registers, and set to use trap x10 for this purpose.

17.2.2 ITE IT8528E I/O Chip Capabilities

Documentation is lacking on this chip.

17.2.3 Power Management

For BMC use in the new backup paradigm, the backup box needs the ability to power up and boot on a timed schedule, to recognize when it has done so and, in that case, to synchronize with the floor system automatically, powering down when that has been done. In the AVBOX1, we use the ACPI capabilities of the NM10 chipset to achieve this. The AVBOX2 seems to do this with the I/O chip for which we don't have documentation. One function does work however.

)ACPI (n-a) Returns the I/O address of the given byte offset in ACPI register space.

POWER-DOWN Forces system to state S5, which is power down.

18. Revision History

REVISION	DESCRIPTION
210918	Update to current format.
210926	Add support for AHCI SATA and AVBOX2
220220	Finalize support for AHCI-SATA and AVBOX2, prepare for full native USB.
220511	Release 5f with initial native USB 1.1 support.

ATHENA Programming ™

saneFORTH PRM *Revised 5/24/22*

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@athenaprog.com