GreenArrays®

Product Data Book DB013 *Revised 3/31/18*

arrayForth® 3 User's Manual

Rev 03a for G144A12 chips

Running on saneFORTH™/Win32 *and* polyFORTH® for GA144

This manual is designed to prepare you for using arrayForth 3 (aF-3) in designing, implementing and testing applications of our chips.

aF-3 is a complete, interactive software development , debugging and installation environment for GreenArrays Chips. It includes an F18 Assembler, example source code including all ROM on each chip, a full software-level simulator for each chip, an Interactive Development Environment for use with real chips, and utilities for creating boot streams and burning them into flash memory. As of aF-3, colorForth is no longer used in this system.

aF-3 is written to run on polyFORTH in G144A12 environments with sufficient resources, and on saneForth for Win32 environments. These versions complement each other and each has a different emphasis of tools, reflecting their differing purposes. The principal purpose of the Win32 environment is cross-compiling for new chips, commission new boards, and simulate at high speeds. The G144A12 environment is intended for interactive development and testing, with F18 and polyFORTH source code in a single base that can be maintained by either system. In most cases, we intend that you will be using an EVB instead of a PC as the principal host for software development.

Although it is configured to support the GreenArrays EVB002 Evaluation Board, it may easily be used to program and debug our chips on the EVB001 in your own designs.

Along with the above tools, including complete source code for the Virtual Machine environments, this release incorporates the source code for our Automated Testing systems as well as that which has been used in taking the characterization measurements reflected in the G144A12 Data Book.

Your satisfaction is very important to us! Please familiarize yourself with our Customer Support web page at http://www.greenarraychips.com/home/support. This will lead you to the latest software and documentation as well as resources for solving problems and contact information for obtaining help or information in real time.

Contents

1. Introduction to this Manual

This is the primary reference manual for the arrayForth programming environment. It should be read and understood in its entirety. In the interest of avoiding needless and often confusing redundancy, it is designed to be used in combination with other documents.

1.1 Related Documents

The general characteristics and programming details for the F18A computers and I/O used in the GA144 are described in a separate document; please refer to *F18A Technology Reference*. The boot protocols supported by the chip are detailed in *Boot Protocols for GreenArrays Chips*. The configuration and electrical characteristics of the chip are documented in *G144A12 Chip Reference.* The evaluation board has its own Data Book along with numerous Application Notes. polyFORTH is likewise documented separately. The current editions of these, along with many other relevant documents and application notes as well as the current edition of this document, may be found on our website at http://www.greenarraychips.com . It is always advisable to ensure that you are using the latest documents before starting work.

1.2 Status of Data Given

The data given herein are *released* and *supported*. The subject applications are under continual development; thus the software and its documentation may be revised at any time.

Supplemental information is available on our website at http://www.greenarraychips.com/home/support. This page is updated frequently and we recommend that you visit it regularly.

1.3 Documentation Conventions

1.3.1 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". This is true in source code as well.

1.3.2 Node coordinates

Each GreenArrays chip is a rectangular array of ***nodes,*** each of which is an F18 computer. By convention these arrays are represented as seen from the top of the silicon die, which is normally the top of the chip package, oriented such that pin 1 is in the upper left corner. Within the array, each node is identified by a three or four digit number denoting its Cartesian coordinates within the array as *yxx* or *yyxx* with the lower left corner node always being designated as node 000. This convention is expanded to *cyyxx* on multi-chip boards such as the EVB001 and 2, in which case *c* is chip number (0 for host, 1 for target, etc). All functions herein accepting *yxx* notation also recognize *cyyxx* appropriately.

1.3.3 Cardinal directions

When it's necessary to refer to directions in the chip geometry without reference to node-specific port directions such as left or up with, we use cardinal compass directions such as North for positive Y axis and East for positive X.

1.3.4 Register names

Register names in prose may be used with or without the word "register" and are usually shown in a bold font and capitalized where necessary to avoid ambiguity, such as for example the registers **T S R I A B** and **IO** or **io** .

1.3.5 Bit Numbering

Binary numbers are represented 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. The notation P9 means bit 9 of register **P**, whose binary value is x200, and T17 means the sign (high order) bit of 18-bit register **T**.

2. Introduction to arrayForth 3

aF-3 is a complete, interactive software development , debugging and installation environment for GreenArrays Chips. It includes an F18 Assembler, example source code including all ROM on each chip, a full software-level simulator for each chip, an Interactive Development Environment for use with real chips, and utilities for creating boot streams and burning them into flash memory. As of aF-3, colorForth is no longer used in this system.

aF-3 is written to run on polyFORTH in G144A12 environments (polyFORTH or pF/144 herein) with sufficient resources, and on saneFORTH for Win32 environments (saneFORTH or sF herein). These versions complement each other and each has a different emphasis of tools, reflecting their differing purposes. The principal purpose of the Win32 environment is cross-compiling for new chips, commission new boards, and simulate at high speeds. The G144A12 environment is intended for interactive development and testing of nodes on its own chip, or on other chips by internal or external means, with F18 and polyFORTH source code in a single base that can be maintained by either system. In most cases, we intend that you will be using an EVB instead of a PC as the principal host for software development. Here is the checklist of features supported by each system:

Capability Supported	sF	pF/144
F18 native code Assembler	YES	YES
Source and object code auditing	YES	YES
Concordance and fast FIND in source code	YES	---
Support for 2-chip targets	YES	YES
External IDE	YES	tbd
Internal IDE	---	tbd
Boot stream generation	YES	YES
Boot stream auditing	YES	YES
Internal boot stream delivery	---	YES
Serial boot stream delivery	YES	tbd
Flash boot stream burning in target	tbd	tbd
Flash boot stream burning internal to own system	---	YES
Target Compiler for pF/144 Nucleus	---	YES
Chip test via External IDE	tbd	tbd
ATS Chip Testing	---	tbd
Software Simulator	tbd	---

Wherever practical we have kept the human interface to the programming tools consistent between the sF and pF/144 environments. In some cases this is inconvenient; for example, even though sF is a 32-bit system, values that must be double precision on pF/144 must also be double in sF. Experience has taught that this is a case in which a little inconvenience buys portability between the environments and helps avoid nasty programmer traps.

Where there are differences between the two systems, we note them with the markers **sF ONLY** or **pF/144 ONLY** .

Although it is configured to run on, and support, the GreenArrays EVB002 Evaluation Board, aF-3 may easily be used to program and debug our chips on the EVB001 or in your own designs.

2.1 Environment in saneFORTH

On the saneFORTH/Win32 platform, aF-3 is organized in a CAPSULE. Some of the environment is resident in the capsule, such as the Assembler and state variables for the External IDE. Other parts are overlays loaded as needed. The capsule may be loaded by block 9 or it may be loaded when needed. One word does it all:

> **AFORTH** This word, defined in GOLD by block 9, loads the aF-3 environment when it's first invoked, defining a resident capsule AFORTH whose name is also defined in GOLD thus overloading the prior word which created the environment. Thus the first and any subsequent uses of the word AFORTH will yield the same environment, except that state variables are persistent until the system is next booted.

The resident AFORTH environment includes all the words and functionality documented in 5.2 Assembler Syntax and Semantics, below, as well as those in 2.3 Common arrayForth 3 User Vocabulary.

Within the AFORTH environment the following high level things may be done:

> **HELP** displays reminders for these functions.
>
> **CONFIG LIST** shows the main configuration block for arrayForth.
>
> **CONC** refreshes the Concordance data base by re-parsing all of the source code.
>
> **HOST TARGET** or **BRIDGE LOAD** loads the External IDE for indicated mode. See 6.1. External IDE below.
>
> **EXAMINE LOAD** loads tools for examining object bins and comparing them.
>
> **STREAMER LOAD** loads utility for building port executable streams.
>
> **SERIAL LOAD** loads terminal emulator for talking to polyFORTH/144 (or to eForth).

2.1.1 Installation on Win32 Platforms

Procedures and tips for installing and managing aF-3 on several common Win32 platforms may be found in the Appendices of this document, supplemented by FAQ items on our website.

2.1.2 Suggested Usage

The sF environment is useful for all modes of host-target development and debugging with a suitable umbilical connection. It is essential for initial commissioning of any GA144 hardware, such as an EVB, when a commissioned pF/144 system is not available. You may prefer it when the higher speed of host operations is more important than the higher bandwidth and lower latency umbilical connections achievable with pF/144 hosts, and when you do not intend to use pF/144 at all.

2.2 Environment in polyFORTH/144

Using the G144A12 itself as a development platform, resources are more constrained than they are on the x86 so the packaging of the aF-3 environment differs. Some parts, specifically the Assembler and aF state variables, are resident, so that arbitrary application code can assemble F18A instructions. All the rest is overlaid. After booting pF and saying HI the aF-3 environment is established by a single word:

AFORTH This word, defined by block 9, loads the aF-3 environment when it's first invoked, defining a resident set of tools in FORTH and ASM vocabularies. AFORTH is redefined as equivalent to EMPTY . Thus the first and any subsequent uses of the word AFORTH will yield the same environment, except that state variables are persistent until the system is next booted.

The resident AFORTH environment includes all the words and functionality documented in 5.2 Assembler Syntax and Semantics, below, as well as those in 2.3 Common arrayForth 3 User Vocabulary.

Within the AFORTH environment the following high level things may be done:

HELP displays reminders for these functions.

CONFIG LIST shows the main configuration block for arrayForth.

HOST TARGET or **BRIDGE LOAD** loads the External IDE for indicated mode. See 6.1. External IDE below.

EXAMINE LOAD loads tools for examining object bins and comparing them.

STREAMER LOAD loads utility for building port executable streams.

SERIAL LOAD loads terminal emulator for talking to polyFORTH/144 (or to eForth).

2.2.1 Installation on Evaluation Boards

Please see DB006, *G144A12 polyFORTH Supplement*, for installation procedures.

2.2.2 Suggested Usage

The pF/144 environment is useful for all modes of host-target development and debugging with a suitable umbilical connection. In addition, only the pF/144 system permits intimate F18 code development and testing on the development system itself; for example, one can add micro-instructions to extend the polyFORTH Virtual Machine in vivo and test them interactively without rebooting.

This platform, like sF, may be used for initial commissioning of any GA144 hardware, such as another EVB. You may prefer it when the higher bandwidth, lower latency umbilical connections achievable with pF/144 hosts are more important than is the higher speed of host operations in sF.

2.3 Common arrayForth 3 User Vocabulary

This section lists resident words (those accessible after **EMPTY** or after naming AFORTH on the sF system) that are published for general use in high level programming. These are all high level FORTH functions, not F18 functions; for F18 Assembler syntax see section 5.2 below.

Chip Configuration Words

nnx (-n) node columns/chip.

nny (-n) node rows/chip.

nns (-n) number of nodes/chip.

nnc (-n) number of object bins.

nn-b (nn-n) convert ccyyxx to linear node (or bin) number.

b-nn (n-nn) convert linear node number to ccyyxx form.

AFORTH Config Words

0bin (-n) starting absolute block of the array of object bins. Size of this region is nnc blocks.

CFORG (-n) starting absolute block of a reference array of object bins.

Bin Manipulation Words

)BIN (nn-a) return start address of given bin in block buffer.

@BIN (nn) retrieve given bin to bMEM array.

Named Blocks

STOCK (-n) assembles all the standard object bins that are supporetd by GreenArrays.

SRAM (-n) assembles the bins for the SRAM cluster. SRAM 1+ is BDL for the cluster. SRAM 2+ defines residual paths for boot nodes after SRAM is installed.

PFVM (-n) assembles the bins for the polyFORTH virtual machine. PFVM 1+ is BDL for the full virtual machine environment including ganglia. PFVM 3 + is an example of 1-chip pF boot stream for flash.

ENIC (-n) assembles the bins for the Ethernet cluster. ENIC 1+ is BDL for the Ethernet cluster.

SRAMIF (-n) is loaded at, normally, an org of x39 to provide the interface routines needed by an SRAM client.

Persistent Parameters

A-COM A-BPS are variables for the "A" (host port). **sF ONLY**.

C-COM C-BPS are variables for the "C" (target port). **sF ONLY**.

U-COM U-BPS are variables for the port currently in use by the external IDE. **sF ONLY**.

DH (-dh) returns currently open IDE file handle or -1. **sF ONLY**.

Unnamed Load Blocks

2202 Serial boot of 2-chip target system for polyFORTH, with or without Ether NIC.

2205 Serial boot of 2-chip target system for polyFORTH, with or without Ether NIC.

2208 makes 2-chip Flash boot stream for polyFORTH, with or without Ether NIC.

2211 makes 1-chip Flash boot stream for polyFORTH, with or without Ether NIC.

2121 generates example stream to locate one node internally using the snorkel through node 207 .

2122 generates a similar example for the second chip assuming bridge has been built.

2123 generates another example of using the snorkel to add the Ethernet NIC to a running polyFORTH system. Requires editing block 9 and rebooting before ETHER LOAD..

2124 generates an example serial boot to wiggle a pin.

xxx Words

3. Tools for Managing Disk

Source code is maintained using the standard polyFORTH character and line `EDITOR`. Large scale examination is supported by the word set `ax nx bx qx`. Reconciliation and management of blocks is done with the standard polyFORTH `DISKING` utility.

3.1 Concordance

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.

`CONC` Runs utility to re-parse source and rebuild the data base.

`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`** move forward and backward within the blocks containing the result set. The current block number must be within `[0..38400[` relative to `OFFSET`.

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.

3.2 Printing Listings

4. Mass Storage

aF-3 mass storage is organized in standard fixed length Forth blocks of 1024 bytes.

4.1 Disk Organization

Both systems use a primary disk organization of 4800 blocks, subdivided into 2400 of source followed by 2400 of associated shadows, and further subdivided into 60-block index pages with the following organization as shipped (the green highlighted pages are redacted in systems configured for public release and available for application use):

Block	sF/Win32	pF/144
0	9 LOAD options, tools, utilities	<--- Same
60	x86 Utilities	pF/144 Nucleus Source
120	x86 Extensions, tools, tests	Extensions, tests, documentation
180	x86 Nucleus Source	Novix arithmetic for conversion
240	x86 Target Compiler & build components	Benchmarks, memory tests, utilities
300		
360	x86 Arithmetic	
420	x86 Data base support	
480	x86 utilities, benchmarks, term emulator	
540	x86 PE (32-bit) EXE generation	Ethernet and TCP/IP Networking Support Partially converted from x86 to pF/144 Eventually, much of this space will be freed for other use.
600	x86 Win32 APIs	
660	x86 Win32 Window Management	
720		
780	F18 Nucleus Source for x86 Targeting	
840	F18 Target Compiler for x86	
900	GLOW	
960	GLOW	
1020	GLOW	
1080	GLOW 180nm IC	
1140	GLOW 28nm HPP	
1200	*reserved for GLOW*	
1260	*reserved for GLOW*	
1320	*reserved for GLOW*	
1380	*reserved for GLOW*	
1440	x86 Cryptography	
1500	x86 Trace, dumps, concordance	
1560	EXATRON	
1620	x86 PKI Development	
1680	x86 SHA256 Studies	
1740	x86 Litecoin Studies	
1800	x86 Win32 imageFORTH	
1860	x86 Win32 imageFORTH	
1920	x86 JTAG studies	
1980	pF VM (Temporary)	pF VM and Ethernet NIC (Temporary)
2040		
2100	F18 asm & tools	F18 asm & tools
2160	More tools	More tools
2220	G144A12 ROM and F18 system pieces	G144A12 ROM and F18 system pieces
2280	Stock F18 code	Stock F18 code
2340	Target output (x86 and F18)	Copy of some 35-block boot stream.

4.2 Configuration Blocks

We will now discuss several configuration blocks of particular interest.

`Blocks [0..12[` The first twelve blocks are reserved for the native system's colorForth kernel and boot.

`Blocks [12..18[` These six blocks hold the binaries for the fonts used on the colorForth display.

`Block 18` This load block defines the configuration of the colorForth system and application. It is loaded on cold or `warm` boot, compiling various extensions to colorForth, giving names to utilities, and loading blocks 144, 202 and 204 to complete the definition of the arrayForth environment. *The three magenta variables at the start of this block (`ns nblk nc`) must never be moved. Use great caution if altering block 18 or anything it loads since if this sequence is aborted early enough the editor is not available to fix the problem.* The yellow word `qwerty` in this block configures the system for standard keyboard layout and semantics; the yellow word `seeb` immediately following **`qwerty`** configures the system to display blue words. These are the recommended settings.

`Block 144` This load block extends colorForth to support the arrayForth tools with global variables, extra functions, names of further utilities, resident capability to generate png files of screen graphics, and the principal chip configuration parameters from the three blocks 190, 192, and 194.

`Block 202` This block defines application tools and is maintained by GreenArrays. When you have identified the COM port numbers corresponding to the three USB ports on the Eval Board, you will need to edit the values for **`a-com`** and **`c-com`** to be the COM port numbers for USB ports A and C, respectively. The baud rates for these ports are specified here as well.

`Block 204` This block is yours, for any definitions you wish to make global so that they survive empty . Before adding a word to this block, make sure it does not redefine anything else that lies under the empty! Use **`def`** and **`' (tick)`** to verify no redefinitions.

`Block 200` This is also yours, for compiling F18 code as described in section **Error! Reference source not found.**.

`Block 148` is the load block for softsim. At present it's necessary to alter this block to control what code is loaded and executed in each node.

By reading the system load blocks starting with 18 in load order you will be able to follow the process of booting arrayForth and become familiar with its components.

To find the ROM code for SPI node 705 for example, type the phrase `705 @rom list` . **`@rom`** takes a node's cyyxx coordinates and returns the number of its ROM load block.

4.3 The Bonus Materials

Some of the "study material" supplied on this disk image may be executed to do useful things, as follow:

`selftest (n)` If the ATS materials supplied in 480 to 720 are still intact, this phrase will run all relevant ATS tests on the chip whose IDE COM port number is given. Note that in order to test the Host chip in this way, its no-boot jumper J26 must be installed before running **`selftest`** or the IDE will hang.

`autotest (n)` Again if the ATS materials are intact, this will cause the Host chip to run ATS tests on the Target chip using the same general procedure as does the factory chip tester, by running tests through the synchronous connection between each chip's node 300. In addition, it runs SERDES test between Host node 701 and Target node 001, transferring a quarter million words of known unique values each way and verifying correct receipt with line turn-arounds.

`450 load` Host polyFORTH IDE boot procedure. Install no boot jumper before loading this.

`460 load` Burns polyFORTH into flash.

`1140 load` Host eForth IDE boot procedure. Install no-boot jumper before loading this.

`1190 load` Burns eForth into flash, for example when updating eForth.

5. Programming the F18

If you have not read DB001, the F18A Technology Reference, please do so to familiarize yourself with the computers and their instruction sets before reading this section.

Both environments include a resident Assembler for the F18 instruction set. As usual, the F18 assembler is a "vocabulary engine" meaning that when the appropriate vocabulary is selected the F18 opcodes, directives, and user defined symbols are available. On the Win32 saneFORTH system this is vocabulary number 9 while on pF/144 it is number 5. The full Assembler environment also uses a special Interpreter to access user defined labels which are stored with the object code rather than in the local dictionary.

The assembler may be used in two distinct ways. The most common is to generate code to reside in and execute from a node's RAM or ROM. In this case code is laid down at consecutive addresses based on a location counter, and is assumed to execute at the addressing for which it was compiled, and in the node stated for things like the "warm" multiport execution address and for ROM content.

The second way is to generate an instruction word as a data item. The usual reason for doing this is to make an instruction word that will be used in port execution. In the past, it was often necessary to hand code such instructions and write them in hex; on the other hand, when it *was* possible to generate the instructions symbolically, the destination address checking for jumps and calls was based on memory location counter for the instruction stream in which the literal instruction was being stored, not on the port in which it would actually be executed. This could generate instructions that would not work as expected when executed in the port. This new method solves both problems. By default, no location counter is assumed and the assembler will generate jumps and calls that don't depend on the execution address of the instruction word. Directives may be used to specify an execution address (normally a port) for greater range of possible destinations at the expense of tailoring the instruction word for a particular port.

Other than the common essentials of Forth compilation technology, this environment includes the mechanism to compile code for hundreds of nodes, each of which has ROM that may be used by the application.

5.1 Object Code

The assembly process was monolithic in earlier versions of arrayForth. As of aF-3 we return to incremental assembly, with object code stored in a binary file area from which it may be accessed to load nodes or chips. This file area survives reboots so it is no longer necessary to reassemble all of the source code every time one needs any part of it.

A set of binary images, called ***bins***, reside on mass storage starting at block ***0bin*** (4800*3 or 14400 by default). Each bin is one BLOCK (1024 bytes long) and contains 256 bytes of RAM image (stored as 32-bit numbers), 256 of ROM, and a table of up to 50 labels for RAM and ROM code. One bin exists for each physical node, and there are additional virtual bins for code that may be used by utilities or placed in different nodes for different applications. The constant **nnc** gives the number of bins for which space is allocated. By typing **nnc** you will see on the stack the value 432 which is 144*3; the area is sized for programs using up to two chips, with an extra 144 nodes' worth of virtual bins into which object (for library code distributed by GreenArrays) may be stashed as noted later.

Because the label tables are persistent in object bins, a programmer with a twisted mind can create much confusion by defining primary Assembler directives or FORTH words as labels, giving the appearance of a broken system. If this has happened to you, either reassemble ROM for this node or fix it with the tools in the `EXAMINE` utility.

5.1.1 EXAMINE - Object Code Auditing

This minor utility for working with object code is useful during development. It can examine bin label tables and bin content in binary or dis-assembled, including comparison with reference binaries. Utilities are provided for initializing a set of bins and for moving quantities of bins.

Two sets of nnc bins are available to this utility. The first, starting at block 0bin , is the object code produced by this machine's Assembler. The second, starting at CFORG , is a reference set produced by this or another system (such as colorForth). Facilities are provided for comparing these and for displaying differences.

-ALL erases all RAM and ROM object code for all nnc bins. Additional words -CH0 -CH1 -XTR erase the three chip-size sections of the bin array.

?ROMS compares ROM for chips 1 and 0 with reference and displays an array of status indicators for all nodes with = indicating identical ROM content and a highlighted ? indicating disagreement.

?RAMS does the same for RAM object for pseudo-nodes 1600 thru 2317, chip 1 and chip 0.

USING (nn) selects a particular bin for examination.

.RAM dumps current RAM for selected bin, highlighting each word that differs from reference.

.CRAM dumps reference RAM for selected bin, highlighting each word that differs from current.

.ROM dumps current ROM for selected bin, highlighting each word that differs from reference.

.CROM dumps reference ROM for selected bin, highlighting each word that differs from current.

-ASM (a n) displays disassembly of the given address range (covering RAM and ROM) in the selected bin and annotating instruction words with labels. Differences from the reference are highlighted and the corresponding reference disassembly is shown in a second column.

.SYM displays the label table for the current bin.

In a normal development cycle where accumulated source changes are reconciled in the process of updating the backup, object can be audited in the same way. To set the current object as reference, use DISKING to copy nnc blocks from 0bin to CFORG .

5.1.2 Getting Object Bins from colorForth

When converting an existing body of colorForth based code to aF-3, it is prudent to use the colorForth object as a reference. The following steps accomplish this:

1. In colorForth, **compile** all of the code of interest.
2. Say **bnamed obj-cf** (or other name of your choice) to output file name.
3. Say **32768 4800 wback** to write all the bins to a 4800-block file of the above name. Only the first 864 blocks are relevant, but this simplifies file mapping into the sF system.
4. Move this file into the pf directory
5. On the sF system, in block 149 temporarily replace **OBJ-AF3** with your file's name, **FLUSH** and **RELOAD** .
6. See block 98. This is a utility for converting your incoming file, mapped as 11 UNIT , into the aF-3 bin format as one of the sections of **OBJ-REF** which is mapped as 10 UNIT . To see a text directory of these sections, list block **10 UNIT 60 -** Note that loading block 98 write enables 10 UNIT
7. Select one of the 600-block reference slots for your object and maintain the directory accordingly.
8. Load block 98 and say **n WAM** where **n** is the starting relative block (multiple of 600) for your data in the **OBJ-REF** file at 10 UNIT .
9. In block 149, restore the **OBJ-AF3** file name, **FLUSH** and **RELOAD** .
10. in aF-3, find the definition of CFORG and aim it at your new reference image. **FLUSH** , **RELOAD** and proceed.

5.1.3 Assembling Stock Code

While incremental assembly is generally a good thing, it does make object code persistent and as a result a lapse in attention or a simple mistake in numbering can deposit garbage in a bin with lasting effects.

Any time the integrity of the object bin array is in doubt you may re-populate it with known good content as follows:

11. Using the **`DISKING`** utility, verify the integrity of the aF-3 source code for tools and for all stock F18 code.
1. Erase all bins by saying **`EXAMINE LOAD  -ALL`** .
2. Reassemble all GreenArrays code by saying **`STOCK LOAD`**
3. Assemble any application code you have added.
4. Verify object integrity with **`EXAMINE LOAD  ?RAMS ?ROMS`**

5.2 Assembler Syntax and Semantics

F18 coding may be interspersed with high level polyFORTH coding; indeed, it may even be used to generate instruction words as literals within high level FORTH definitions. This requires clear and explicit bounding between F18 coding and host system compilation / interpretation regimes. In this section, words shown in red are deprecated from colorForth while those shown in green are new in aF-3.

To assemble F18 code simply interpret Forth that includes one of the code bounding words ASM[or A[when necessary. The F18 assembler is resident in pF/144 and loaded as needed in saneFORTH.

5.2.1 Assembler State Variables

The Assembler's state is represented by five VARIABLEs:

`IWD` 18-bit instruction word being built.

`'SLOT` The next unused slot [0..3] of the instruction in `IWD`. If 4, further ops are forbidden (inline).

`'IW` holds the F18 address at which the instruction in `IWD` will be stored. If negative, bits 10 and up are set, the low order bits [0..9] are the same as they are in `'IP`. In this case the instruction is being generated inline and will not be stored into a bin directly by the Assembler.

`'IP` holds the address of the next word to store into target memory, corresponding with **p** register during execution. If negative, bits 10 and up are set, and the value of **p** is unknown for inline assembly. When an address is specified for inline assembly, the address is stored into the low order 10 bits of both `'IP` and `'IW`, and bits 10 and up of `'IP` are zero.

`'CL` holds the slot number of call opcode in preceding word. Negative if there isn't one. Used in tail optimization. Second volatile cell is slot of last opcode stored.

`'##` Volatile flag set zero by # and true after parsing a number, a named literal, or a named call. When true these things perform their normal behaviors of generating literals or calls. When false, each of them leaves a number on the Assembler's stack.

5.2.2 Assembler Directives for Code Bounding

Source code compatibility between 32-bit (Win32) and 16-bit (pF/144) host platforms when generating code for an 18-bit computer is, fortunately, simple to assure. On both platforms anything being prepared to store into memory must be double precision. All other values are single precision on both platforms.

A special Forth interpreter provides the dictionary searching mechanism to allow access to ***labels*** defined in the current or in other nodes' ***bins*** in both Win32 and pF/144 implementations. In addition, this interpreter generates F18 literal references when naked numbers are encountered.

`ASM` Selects the Assembler's vocabulary without invoking the full environment, used when necessary in tool building.

`# (_ - n)` When running in the Assembler environment, as described below, most numbers occurring in the source code are intended to generate literals for the F18. When it is necessary to provide a number as an argument to one of the Assembler directives, preceding it by # will leave the number on the stack rather than generating a literal. # also conditions named values, such as the port names, to push values on the stack, and the same is true for named calls. This usage is shown where appropriate in the following examples.

Directives are used to control the Assembler's assumptions and behavior in generating inline instructions or bins for code. This section shows appropriate usage for each type of assembly. In the examples, curly brackets { } surround optional words or phrases, and the vertical bar | should be read as "or but not both".

5.2.2.1 *Inline Instructions*

To build an F18 instruction and leave it on the stack, the following pattern is used:

```
A[  {# <port> =P}  <opcodes>  ]]
```

By default, the instruction will be generated with no foreknowledge of the value in **P** during execution. A port (or other) address may be specified using the =P directive. The stack value produced will be double precision. When used within F18 code the value may be laid down in the memory image being built using **,** while when used in high level Forth code the value may be laid down in host memory using I, . Each of these words is defined appropriately in each version of the Assembler to compensate for host cell size. The formal vocabulary is as follows:

A[Enters the Assembler environment, saving its state and setting up for generation of a single instruction word. Both interpretable and IMMEDIATE, usable in both FORTH and Assembler environments to generate instruction words in-line for use as literals or in building tables.

]] (-d) Exits that environment, returning a single instruction word (double precision) and restoring the Assembler's state as it was before A[was encountered.

=P (a) Sets the assumed address in register **P** when this instruction is executed. May only be used within an inline instruction definition.

, (d) lays the double instruction or other value down at the next available address in F18 memory, used when the phrase A[...]] is nested within F18 code (see below).

I, (d) lays the double instruction or other value down in host memory as a four-byte number, used when the phrase A[...]] occurs in high level FORTH code or data construction.

5.2.2.2 *Building a bin*

The bin mechanism allows ROM and RAM source code to be maintained and assembled separately. Unlike the colorForth environment, aF-3 does not reassemble everything whenever a changes is made; instead we ideally assemble the ROM code only once, and incrementally assemble application RAM code as needed. To build a bin of F18 code, the following general pattern is used:

```
ASM[  # ccyyxx NODE  {# ccyyxx BIN}  { {{-SYM}ERS}|{-ROM} }
   # <addr> org                   always necessary
   <coding>
   FORTH ... ASM                  simple vocabulary switch
   ]ASM   <host code>   ASM[      complete environment switch
   A[ ... ]] {,|lit}              Generate instruction for table/lit
   {>ROM}  >BIN  ]ASM
```

ASM[Enters the full Assembler environment for F18A *without initializing any of its state*. Visible in FORTH.

]ASM Exits that environment.

Within ASM[ ]ASM the ASM vocabulary is selected and a special interpreter runs. As noted above in 5.2.2, this special interpreter recognizes the words in this section and by default generates opcodes in instruction words, literals and so on. To put numbers on the Assembler's stack it is necessary to use # because if you simply write numbers down they will be assembled as literals, and if you write a label it will be called. All numbers used with Assembler directives are of the width documented herein. Numbers to be assembled as literals ***must*** *be written in double precision form.*

`NODE` `(nn)` Specifies which node's position, ROM and RAM to assume, and by default set the target bin to the same. Reads that node's bin into the local working image. Required.

`BIN` `(nn)` Overrides the target bin. Optional. *Bin assignments are currently controlled by GreenArrays; at this time we recommend you compile directly for the intended nodes and not use `bin` yourself.*

`-SYM` Wipes ROM label table, used before `ERS` in nodes with many unnecessary ROM labels to recover space for application labels.

`ERS` Wipes the object memory and label table for the RAM portion of the working image. Does not touch ROM.

`-ROM` Wipes the entire object memory and label tables in the working image.

`>ROM` Secures the current label table as ROM labels in the working image.

`>BIN` Writes the working image to the current target bin on mass storage. To write identical content to multiple bins, use `BIN >BIN` phrases repeatedly before `]ASM` .

`reclaim` This deprecated word was used in most colorForth code to prevent crashes due to filling the colorForth dictionary. Dictionary management in aF-3 uses conventional polyFORTH methods, applicable to any host code or data structures you might build. The host dictionary is not used for F18 labels in this system.

Later on, when loading code into nodes with the IDE "by hand" or when specifying boot conditions for tools such as the automated IDE loader or the stream generator, object code is identified by its *bin number*. By default that is simply the node number unless you have used **`BIN`** to stash the code elsewhere.

5.2.2.3 Special Interpreter Considerations

There is no problem with encapsulating multiple bins in a single source block, and in fact the source code distributed with aF-3 includes examples of doing this. However, the special interpreter activated by `ASM[` requires slight changes in common sF/pF practices.

5.2.2.3.1 Use of EXIT

If `ASM[` interprets an `EXIT` the block continues being processed by the normal FORTH interpreter. To achieve the usual effect of `EXIT` you must write it twice, as in `EXIT EXIT` .

5.2.2.3.2 Bins Needing more than 1 Block

This can be done in two ways, and the code distributed with aF-3 includes examples.

For the first method, see the code for Async Boot ROM. The first block loads the second within `ASM[ ]ASM` and the second block must begin with `ASM[` to activate the special interpreter. Note that the second block does not require any closing bracketing other than an explicit or implicit pair of `EXIT` .

For the second, see the Master DMA Nexus (bin 110) of the Ethernet NIC. The first block begins with all of the normal boilerplate to start a capsule's source, but has no closing bracketing. The second block begins with `ASM[` to resume using the special interpreter, and ends with the normal closing bracketing.

5.2.3 Location Counter

This is an incremental Assembler. Opcodes and literals are written into an image of target node memory as they are encountered; the only retroactive action it performs is to store into the destination fields of words containing forward referencing jumps (such as `if`) or calls (`leap`) when those forward references are resolved, or to change a call opcode to a jump when a call is followed by semicolon (tail optimization). The Assembler keeps track of the current position in target F18 memory with three variables documented earlier: `'IW 'SLOT and 'IP` .

As opcodes are assembled, they are added to the instruction word being built at the address in `'IW` and `'SLOT` is maintained until the word is full or until the next opcode will not fit into the word. At that time the word being built is padded with `.` (nop) opcodes if necessary, and a new instruction word is started at the address in `'IP`. This leaves `'IW` pointing at the new instruction word and `'IP` pointing at the following location in memory. The duality of `'IW` and 'SLOT is necessary to support multiple instructions per word; the duality of `'IW` and `'IP` is necessary to support literals.

When literals are assembled, a `@p` opcode is generated and then the literal value is stored at `'IP`, advancing `'IP`. Alternatively you may write `@p` yourself and lay the following words down using `,` (comma), such as values calculated interpretively or instruction words generated by the assembler's inline nesting feature.

Jumping, calling, and memory operations address a word, not an opcode. When encountering a colon label, or any other assembler directive defining a place that may be addressed in memory, any code under construction is padded with `.` (nop) opcodes if necessary to align the location counter on a word boundary. This means that in absence of explicit control transfer opcodes, execution continues across alignment boundaries including the start of a colon labeled definition, a technique we have learned to use often.

The location counter `'IP` is incremented in the same way the hardware increments **P**: The low order seven bits increment without changing the remaining bits of **P**. If you are generating code in RAM you will stay in RAM, wrapping its address space at x80 in terms of the value of **P** and also at x40 for actually addressing the memory image in the sense that the hardware ignores bit 6 of the address. The compiler will also generate code for ROM, in which case the wrapping points are at x100 and xC0. The P9 bit (x200) may be set as you wish to specify addressable destinations which will run in Extended Arithmetic Mode.

The location counter is managed using these words:

.. forces word alignment; if an instruction word has been started, fills the rest of the word with nops. Equivalent to `1 <sl`.

org `(a)` forces word alignment then sets the compiler's location counter to a given address at which following code will be compiled into the current node's bin. `'IW` and `'IP` are initially equal but will separate after the first opcode has been compiled.

Instructs the Assembler that the following number (or label, named literal, named call, or use if `its` [reference to a label in another bin]) should leave a number on the stack instead of assembling a literal or call as appropriate. For all but named literals the number will be single precision. If you wish, for example, to org to a given address you need to write a phrase like `# x20 org` or `# joe org`. See also 5.2.2 above.

here `(-a)` forces word alignment and returns the current aligned location.

, `(d)` forces word alignment and lays the double instruction or other value down at the next available address in F18 memory, advancing location counter.

+cy forces word alignment then turns P9 on in the location counter. Places in memory subsequently defined will be run in Extended Arithmetic Mode if reached by jump, call, execute or return to those places.

-cy forces word alignment then turns P9 off in the location counter.

<sl `(n)` Ensures that the next slot to be assembled will be ***less than*** the given number, forcing alignment in a new word if that is not the case. Used to solve problems with forward jumps or calls.

Slot assignment is a microscopic aspect of location counter management but is important to an F18 programmer for reasons such as optimizing forward references, aligning code on word boundaries for generation of literal instruction words to send another node through a com port, and the like. The words `{ . .. <sl }` are your main tools for managing slot allocation.

5.2.4 Control Structure Directives

These are used like those in classical Forth. The stack effects shown in square brackets reflect the host Assembler's stack; those shown in regular parens reflect the F18 stack at execution time.

5.2.4.1 Simple Forward Transfers

Forward transfer opcodes are assembled into the next available slot and may, unless you are controlling slot allocation yourself, be placed into slots 0, 1 or 2 with 10, 8 or 3-bit destination fields. The stack "handle" shown as **sa** for the unresolved forward transfer identifies both the location and the slot of the transfer opcode as well as the adjustment for **P** due to any preceding `@p` or `!p` opcodes in the same word. When `then` resolves a forward transfer, it will abort with error message "Range!" if the transfer is unable to reach the position at which `then` occurs without a larger destination field; when this occurs you must alter the code to resolve the problem.

if `[-sa]` If **T** is nonzero, program flow continues; otherwise jumps to matching `then` .

-if `[-sa]` If **T** is negative, program flow continues; otherwise jumps to matching `then` .

zif `[-sa]` If **R** is zero, pops the return stack and program flow continues; otherwise decrements **R** and jumps to matching `then` .

ahead `[-sa]` jumps to matching `then` .

leap `[-sa]` assembles a call to matching `then` .

then `[sa]` forces word alignment and resolves a forward transfer.

5.2.4.2 Count-controlled Looping

The F18 hardware supports looping under control of a count in **R** . The number in **R** is zero-based so the number of iterations such a loop makes is one greater than the initial value in **R** and that value will be zero during the last iteration of a loop. No forward transfers are used by these words and there are no issues with slots; all directives that generate backward transfers will pad the code if needed so that an opcode with the necessary size destination field may be assembled. The directives are as follow:

for `[-a] (n)` pushes n onto the return stack, forces word alignment and saves `here` to be used as a transfer destination by the directive that ends the loop. There are times when it is useful to decompose this directive's actions so that the pushing of the loop count and the start of the loop itself may be separated by such things as initialization code or a label. In this case you may write a phrase like **>r** `<other things>` **begin** .

next `[a]` ends a loop with conditional transfer to the address a . If **R** is zero when next is executed, the return stack is popped and program flow continues. Otherwise **R** is decremented by one and control is transferred to a .

unext `[a]` ends a micronext loop. Since the loop occurs entirely within a single instruction word, the address is superfluous; it is present only so that the form **<n> for … unext** may be written. The micronext opcode may be compiled into any of the four slots.

5.2.4.3 Arbitrary Control Structures

As with ANS Forth, any desired control structure may be generated based on a few simple directives and flexible semantics; see the ANS Forth standard, or more to the point see the F18 code supplied with arrayForth, for many examples of composite control structures. The following directives are provided; the same stack notation (a for destinations and sa for handles to forward references) is employed here. If necessary you may code SWAP to affect the host Assembler's stack. New words introduced in aF-3 are shown in green.

begin `[-a]` forces word alignment and saves `here` to be used as a transfer destination.

while `[x - sa x]` equivalent to **if SWAP** . Typically used as a conditional exit from within a loop.

-while `[x - sa x]` equivalent to **-if SWAP** . Typically used as a conditional exit from within a loop.

until `[a]` If **T** is nonzero, program flow continues; otherwise jumps to `a` . Typically used as a conditional exit at the end of a loop.

-until `[a]` If **T** is negative, program flow continues; otherwise jumps to `a` . Used like `until` .

again `[a]` unconditionally jumps to `a` . The old colorForth spelling **end** may also be used.

repeat `[sa a]` unconditionally jumps to `a` and resolves the forward jump at `sa` . equivalent to **again then** .

else `[sa - sa]` jumps to matching then and resolves preceding forward transfer.

***next** `[sa x - x]` equivalent to **SWAP next** .

5.2.5 F18 Opcodes

The preferred opcode names, as shown in the *F18A Technology Reference,* each compile an opcode into a slot:

```
ex  @p @+ @b @  !p !+ !b !
+* 2* 2/ -  + and or drop  dup pop over a  . push b! a!
```

The `call` opcode is compiled when an F18 label is referenced.

lit `[d]` generates a literal of the given value by inserting a @p opcode and laying the value down in memory.

alit `[u]` generates a literal of the given *unsigned* single precision value by inserting a @p opcode and laying the value down in memory.

<a valid number> encountered during assembly also generates a literal ***(in which case it must be written as double precision!)*** except when preceded by `#` in which case it leaves single or double precision on the Assembler stack as written.

S>D `[n - d]` converts a signed number to double precision, sometimes necessary with other directives.

Tail optimization is performed by **;** if immediately preceded by a `call`. In this case, the `call` is converted into a `jump`, conserving return stack space and leading to other useful techniques. Other jumps are generated by control structure directives, described later.

Four of the original opcode names assigned in colorForth, have proven to be poor human factors decisions because their names conflict with standard Forth usage and therefore create "programmer traps." In aF-3 we have deprecated the four opcodes shown in red above to eliminate these "traps" and renamed them as follows:

inv	replaces **-** for a bitwise ones complement of T.
xor	replaces **or** for a bitwise exclusive OR of S and T.
r>	Replaces **pop** for moving a word from the return to the data stack.
>r	replaces **push** for moving a word from the data to the return stack.

A block of code is provided to add the old colorForth names to the Assembler if you truly wish to use them, however it is not resident by default to save memory.

By default, unused slots are set to return (**;**) opcodes for object compatibility with colorForth (and with the code in ROM on the chip.) When the Assembler concludes that it must start a new instruction word within a code stream such that execution may continue from the preceding word into the new one, it must fill any unused slots in the preceding word with nops (**.**). You may do this yourself, for example to lay down an inline machine code literal not encapsulated by `A[` and `]]` with the **..** directive that's described earlier.

5.2.6 Dictionary Labels

During assembly, a table of labels is built inside each bin image. Each label is stored as a counted string with maximum of the first seven characters actually saved for uniqueness. Each label has a 16-bit value. The following words manage this dictionary:

equ `(n _)` creates a new label whose name follows, assigning it the given value.

: `(_)` forces word alignment and defines a label at `here` as did red words in colorForth.

its `(nn _ - n)` normally assembles a `call` to the following label as defined in the given bin. However when preceded by `#` the ***single precision*** value is placed on the Assembler's stack.

<valid label> when encountered during assembly, a valid label defined in the working image assembles a `call` to that label. However when preceded by `#` the ***single precision*** value of the label is left on the stack.

Writing a valid <label> is equivalent to writing `# <label> call` .

5.2.7 Other Useful Words

Several additional resident definitions facilitate writing source code for F18s:

5.2.7.1 Named Literals

The following words, naming registers, normally assemble literals in the F18 instruction stream, however they simply leave their values (double precision) on the stack when preceded by `#` .

io right down left up data ldata

5.2.7.2 Named Calls

Each of the 15 valid multiport addresses has a named word that normally assembles a `call` to that address. However when preceded by `#` they simply leave their ***single precision*** addresses on the stack:

---u --l- --lu -d-- -d-u -dl- -dlu

r--- r--u r-l- r-lu rd-- rd-u rdl- rdlu

await generates a call to the default multiport execute for a node based on its position in the array, also may be conditioned by `#` to return address on the stack.

5.3 Module Organization

A preliminary convention has been adopted for organizing and packaging code in a modular way, from a single node to a cluster. This convention allows use of a single block number or name as a "handle" for the module. The first two or more blocks of the module have fixed functions at fixed offsets. We recommend that you apply these principles in packaging your own code. The SRAM Control Cluster Mark 1 may be studied as an example while reading the following sections; the constant **sram** is the number of the first block of that module.

5.3.1 Load Block

The first block in a module is a single block which when loaded will cause all of the source code needed by a module to be compiled. There may be arguments to this block. When completed the necessary object code will be stored in appropriate bins (which may simply be those belonging to the nodes programmed.) If by its design a module needs to export addresses or other identifiers, these will be available in the dictionary after it has been compiled for use in code compiled later. *In a degenerate case this block may actually contain all of the module's source code.* See block **sram** for an example.

5.3.1.1 Identifier Scope Control

F18 compilation is done incrementally, which means that identifiers may not be "forward referenced" symbolically. Any identifier of any sort must be defined earlier in the compilation sequence than its references.

The scope of identifiers, including intentional or inadvertent overloading of system or compiler vocabulary, is controlled using the **remember** word **reclaim** . In the normal case, identifiers are not exported at all, and if they are it is usually not far in the compilation sequence. As a general practice, we recommend starting each node's code with **reclaim** unless you are forced to do otherwise. Elaborate scope management structures may be implemented by defining and using your own **remember** words within the scope of **reclaim** .

5.3.2 Boot Descriptors

The second block in a module (load block number plus two) defines the loading requirements for the module, using the high-level language described in section 7.1.1 below. This language lists the nodes that have to be loaded, indicates what code to load into their RAM, provides for initialization of registers and/or stacks, and provides a starting execution address for each node. The data are recorded in tables so that the order of declaration does not need to match the actual order in which the nodes are loaded. See block **sram 2 +** for an example.

5.3.3 Residual Paths

In rare cases there may be modules that need to be loaded, activated, and used during a boot sequence; SRAM and SDRAM control clusters are examples of these. Once started it may be difficult or impossible to stop them without resetting the chip and, in the case of SDRAM, perhaps losing the data in the device. If an application's boot process has this sort of complication then the module in question should provide a new path for use by stream or IDE loaders that can reach and boot all nodes remaining in the chip that are not part of the module which is running and hence now "in the way." See block **sram 4 +** for an example, showing suitable paths for boot nodes 708 (async serial) and 705 (SPI flash.)

5.3.4 Organization of Larger Projects

The appropriate structure for a larger program depends on its size and intended use. The load block should still be first. Boot descriptors should start in the second block but may require several. If IDE operation is appropriate after the program has been loaded, an IDE personalization with path(s) adjusted to access the remainder of the chip may be useful. Scripts may be necessary to specify boot stream generation, IDE loading, and/or **softsim** setup as appropriate. You might wish to include a script to produce HTML listings. Not all of these elements are appropriate for every application, but if they are small you might wish to place them in this area before the actual source code. These are merely suggestions and the recommendations may change as the system evolves.

5.4 Methods of Loading Code

After all the desired code has been compiled into object bins, it may be loaded for execution or simulation. There are presently several basic ways in which this may be done:

Manual, interactive loading into the real chip with External or Internal IDE: Insert code into RAM, do any necessary initialization manually, and call routines to observe behavior, test boundary conditions, look for side effects and so on. It is impossible to over-emphasize the importance of this practice. Simple, straightforward unit testing is a fundamental entitlement of a Forth programmer and its use dramatically simplifies debugging later on. Don't deny yourself this advantage by combining a cluster of nodes full of untested code unless you enjoy coping with serious problems either immediately or some day in the future.

Manual interactive loading into softsim: Use **NODE** to select a node and **BOOT** to insert code.

Automated loading into softsim: Initialize nodes as directed by Boot Descriptors.

Automated loading into the real chip using boot streams: Use the **STREAMER** utility to construct a boot stream as directed by Boot Descriptors, then present the resulting stream to a boot node or inject it into the chip using the Snorkel.

Fully automated loading into the real chip upon RESET- signal: Use the **STREAMER** utility to construct a boot stream as directed by Boot Descriptors, then burn it into the front of SPI Flash.

Each of the automated methods uses the common Boot Descriptor Language to define what code, if any, shall be loaded and what other initialization shall be performed before starting each node.

Which methods are appropriate depends on which host system you are running on and what connection you have to the target system, if any. sF may use one or two serial COM ports connected to one or two ("host" and "target") chips. pF/144 may use its 1.8V async interface from node 708 (requires jumpers and cables), or its 1.8V sync interface from node 300 either to the target chip on the same eval board (default jumpers), or to another chip (requires jumpers and cables). pF/144 uses node 500 to drive the reset pin of the chip and this also requires attention to jumper and cable depending on the target chip. pF/144 may also inject streams into the chip it's running on via the Snorkel in node 207, and by building a bridge these streams may continue into another chip.

6. Interactive Testing

Interactivity is a critical, cardinal virtue of Forth as a programming system. Turning Forth into a batch mode programming system would be a travesty, sacrificing one of its most potent properties and setting the clock back nearly half a century. When using Forth, a programmer becomes accustomed to having her fingers literally on the fabric of the computer, directly manipulating its memory, registers, and other resources; exercising hardware and software without necessarily having to write (and debug) software simply to exercise those things.

The IDE was implemented as a natural and obvious extension of the umbilical methods used with embedded Forth systems for many decades. The method used is as non-invasive of hardware and software as is practical in this architecture, requiring no RAM or ROM in a node being tested and touching its registers as lightly as feasible.

With aF-3 there are two different environments. External IDE allows a host system to manipulates a target chip using synchronous or asynchronous serial interfaces as appropriate; this environment has been in use with arrayForth since the dawn of GreenArrays. Internal IDE is new with aF-3 and employs the Snorkel/Ganglia mechanism to permit a pF/144 system running on a G144A12 chip to interact in analogous ways with other nodes on the same chip as pF/144 is running on *(even some nodes that are part of the pF Virtual Machine!)* and, if the chip is bridged to another, on a second chip as well. There is a technical App Note AN019 on the IDE internals. This section covers the use of both environments.

6.1 External IDE

This utility is implemented for both sF and pF/144.

6.1.1 Terminology

Debugging is done on one or more *target nodes*, using an umbilical connection which will involve one or more intermediate nodes of one to three kinds (*root*, *wire*, and *end*), depending on the physical location of the target node within the chip and on the *path* taken to reach it. For IDE into the asynchronous interfaces of a GA144 including either of the chips on an EVB001 board, the root node for either chip will always be 708 because that is the one and only asynch boot node on a G144A12. The asynch IDE may thus be used to debug code in any node *except* 708.

The inner load block for the asynchronous IDE is **`SER`** but this is intended to be used as a factor of other utilities. To work with the host chip, say **`HOST LOAD`** and to work with the target chip say **`TARGET LOAD`** after remembering to edit **`A-COM A-BPS C-COM`** and **`C-BPS`** to define the correct COM ports and line speeds.

6.1.2 Using the External IDE

1. Assemble the necessary code into bins and make any necessary connections to the target system. Set serial port numbers as needed. **sF ONLY:** Connect at least one COM port (usually high speed FTDI) to the target system. If this is a new port, configure it correctly as described in the Appendix applicable to your platform. Edit the main configuration block, `AF-3 2 +`, to reflect this port number as host or target chip, and select a baud rate that works for the electrical interface in use. Repeat this process if you are using two connections.
2. Examine and, if necessary, change the path lists to make sure you can reach all the places you need to. See the definitions of `0PA 1PA 2PA` after the block named `SER` for examples.
3. Make sure nothing else in the windows environment (like dialog boxes) has any open handles for these device(s).
4. To work with the host chip, say **`HOST LOAD`** on either platform. **sF ONLY:** To work with the target chip say **`TARGET LOAD`**.
5. Say **`TALK`** to reset the target chip (may be prevented) and download root node talker code into the chip. ***On some target boards, you may need to cycle power or press a reset button before this step***. If the serial port is already set up and you do not wish to reset the chip, use **`Talk`** instead.

6. Say **.PTH** for a concise display of IDE state. You should at this point have three defined paths, one to each of the nodes immediately adjacent to the root node, for example:

```
.PTH
  2  709    1 709
  1  608    1 608
  0  707    1 707  <--- SELECTED
```

   The first column is path number; the second shows which direction it goes (first node away from the root). Third column shows the number of COM port boundaries between the root and the target node, and the fourth column is the target node to which the path is currently connected.

7. To see more about the target node use **SEE** which displays the path state as well as the stack and RAM in the target node.
8. Define, tear down, and select paths as you wish, and operate on the nodes in question using the words described below. *Note: If you wish to prove that all other bootable nodes are completely idle, begin your activities by saying* **2 <root> HOOK** *(for example, 2 708 HOOK when using async IDE) and then* **2 -HOOK** *to tear that path down. The default path 2 can reach all 143 nodes accessible to the IDE.*
9. Operate on the selected target node as you wish using the basic or enhanced vocabulary.
10. If you wish to change the code you are downloading into any node(s), change the F18 source and load it to update the object bin(s). IDE state is not affected by normal assembly. Any paths you had wired up, and whichever you had most recently selected, will still be in effect as you can see by displaying **PANEL** again. *Assembly does not communicate with the chip under test.*
11. When you are finished you may depart the IDE by ....

sF ONLY As long as you do not say bye to your arrayForth session, you may later load the IDE again and resume communicating with whichever paths you had left connected.

6.1.3 External IDE Vocabulary

Words come in two classes: Those which wire, rip and select paths, and those that operate on the target node.

6.1.3.1 Path Routing Control

To see how the path routing is currently set up, say **.PTH** . Node numbers used with these words are always in the cyyxx form.

PATH (i) Selects path **i** (0, 1, or 2) so that all subsequent target operations will apply to the target node for that path. *This is necessary when more than one path leads to the same target node.*

NODE (nn) Selects whichever path presently has node **n** as its target, if any. Note that it is possible to set up more than one path to different ports of the same node; if you have done this, **PATH** will permit you to select the desired port. Leaves a default path selected if none of them targets the node given.

HOOK (i nn) Wires path **i** to target node **n** after ripping out any existing wiring for that path, leaving path **i** selected. If the node in question is not accessible in the route list for that path, leaves the path set for the appropriate adjacent node as target.

-HOOK (i) Forces ripping out of any wiring for path **i** . Each node that had previously been part of this path is normally left in its default **warm** state (multiport execute); the target node *is not affected*.

Path Node Hook -Hook These words do the same things, but silently.

PTB (i-a) an array of starting addresses indexed by path number. To adopt a new path, store its address into this table at a time when that path index has been unhooked.

6.1.3.2 Target Operations

Each of these operations applies to the target node of the currently selected path.

UPD retrieves all ten words of the data stack into the local array **STACK** which is indexed (T, S, and the eight elements in the F18 stack, first element being the one that would next be popped into S). Displays the stack conditionally (using **?STK** .)

.STK displays the contents of **STACK** in two rows, in the current **BASE** . First row is the deepest four elements of the F18 stack, followed by the shallower four elements, then S and T. The shallowest element is immediately left of S; "deeper" moves to the left on that row, then to the right on the row above, with the deepest element on the right side of the top row. Thus, the eight words of the stack array seem to move circularly. This is isomorphic with the stack array and makes its behavior clearer.

```
.STK                          vv--Next Push from S
  2AAAA  2AAAA     15     15     (S)    (T)
  2AAAA  2AAAA  2AAAA  2ABAA     15     15
                              ^^--Next Pop to S
```

?STK displays the stack as does **.STK** if **?MUTE** is zero.

?RAM displays all of RAM from the selected node (in the current **BASE**).

?ROM displays all of ROM from the selected node (in the current **BASE**).

-ASM (a n) disassembles **n** words of code starting at **a** in ROM or RAM. Labels are integrated from whatever bin is current.

LIT (d) removes a double number from the host computer's stack, pushing it onto the data stack of the target node and displaying the updated stack if not muted.

R@ (a-d) reads RAM, ROM, register or port word at **a** in the target node, pushing the data read, **d** onto the host computer's stack. *In this release, uses and does not restore register **a** in the target node.*

R! (d a) writes a double number from the host computer's stack into target memory (RAM, register, port) word at address **a** . *In this release, uses and does not restore register **a** in the target node.*

RINS (d) Executes a given instruction word in the port of the target node. Usually built with `A[ .. ; ]]`

CALL (a) calls, from the port, the code at address **a** in the target. If that code returns to the port or re-establishes the normal rest state of the target node, interactive use may continue once the code completes. No interlocking is done so this method may also be used to start application processing which will not admit to further port execution access. Interaction with a node actively running an application may also be arranged but at the cost of periodically polling, as described below.

BOOT (a n nn) loads code into the target node, starting at address **a** in both target and bin for **n** words, from the current binary output area identified by **nn** . This need not be the same node as the target.

FOCUS forces the target to call only the port on which the current IDE path is talking to it. This remains in effect until the target is directed to execute elsewhere.

VIRGIN forces the target to call its default multiport execution address to un-do the effects of FOCUS .

IO DATA UP DOWN LDATA LEFT RIGHT place chip port addresses on the host's stack.

RB! @B !B RA@ RA! @A !A @+ !+ R+ R+* R2* R2/ RINV RAND ROR RDROP RDUP ROVER Execute single F18 instructions in target, using target's stack, and update stack display in panel. Note: These functions may be omitted in favor of `A[ opcodes ; ]] RINS`

6.1.4 Advanced External IDE Uses

When ripping a path out, all wire and end nodes (if any) are left in their default multiport executes. This is necessary so that, for example, paths may be crossed so long as they are not used concurrently to cause conflicts. There are occasions, such as when starting nodes that send unsolicited code or data to other nodes, when this is not appropriate. In those cases the following two functions are useful:

UNFOC Conditions -hook to work as stated above..

FOC Conditions -hook to leave every wire and end node focused back toward the root. This leaves the path completely impenetrable to anyone other than the root node, and further hooking and starting of nodes will be necessary if that path is ever to be permeable again.

The IDE is factored such that it may become a tool of other utilities. Examples are the **SELFTEST** and **AUTOTEST** routines, the IDE boot and flash burning tools used for eForth and polyFORTH, and even the host and target load blocks. Until these conventions are documented here, please read the code just identified to see how it is done.

6.1.5 Working with Two Chips using External IDE

The IDE may be used to interactively debug software on both chips of the Evaluation Board as though they were a single chip with 288 nodes (two of which are invisible), using just the serial interface for the host chip.

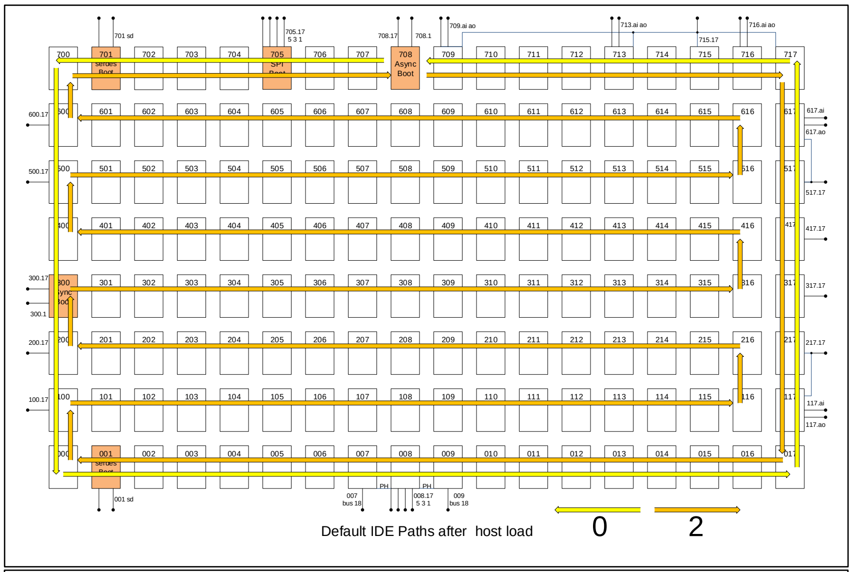
`BRIDGE LOAD` compiles a host IDE version capable of operating on both chips. Initially its environment and function is identical with `HOST` and you should establish connection with the host chip using `TALK` if it isn't already established.

`SPAN` extends the bridge IDE to encompass both chips. In order to successfully invoke `SPAN` the edge nodes of the host chip starting with 707, proceeding to the left to 700, and downward to and including 300 must be accessible for programming; it is typically used immediately after loading `BRIDGE` and establishing host connection unless something such as polyFORTH needs to be set up on the host chip first. `SPAN` resets the Target chip and programs node 300 on each chip as a transparent bridge for carrying port communications between them, using 2-wire synchronous communications. New default paths 0 and 2 are set up to cover both chips. Thereafter, nodes 300 are dedicated to this purpose until the chips are reset.

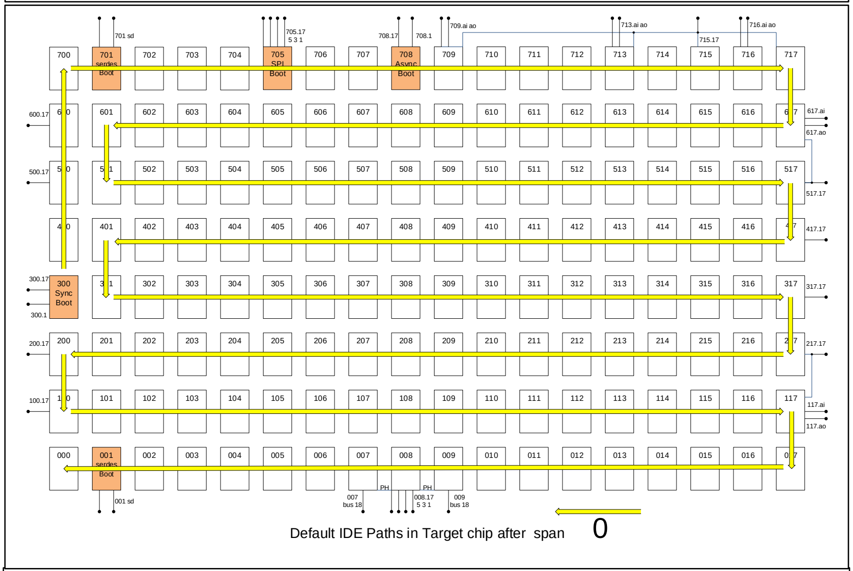
With the port bridge built, the **up** ports of node 400 on each chip are logically connected as though they were a simple COM port. Port read/write communications, such as IDE, are basically transparent across this connection except that data transfers take 100 or more times longer. The current version of the bridge supports flow control, so unlike the old version node 400 may determine whether the bridge has data for our chip or is ready to send a word to the other chip by examining **io**: If the bridge is sending us a word, node 400 can see **up** writing, and after we write to up from node 400 we will not see **up** reading again until the word was accepted by 400 in the other chip. *However, you may not infer by seeing* ***up*** *reading that node 400 on the other chip is at this time reading its* ***up*** *port. Any programming method that depends upon such awareness will have to be implemented in some other way.*

The default paths after `SPAN` , shown below, facilitate full access to the Target chip with or without the polyFORTH virtual machine present on the Host chip. Extended node numbering is supported in the form cyyxx where c is zero-relative chip number; thus nodes 000 through 717 are on the Host chip, while nodes 10000 through 10717 are on the Target chip. The IDE is aware that a connection from 400 to 10400 exists through their up ports. ***You may only have one path hooked through the bridge at any time! Attempting to violate this rule will simply hang.***

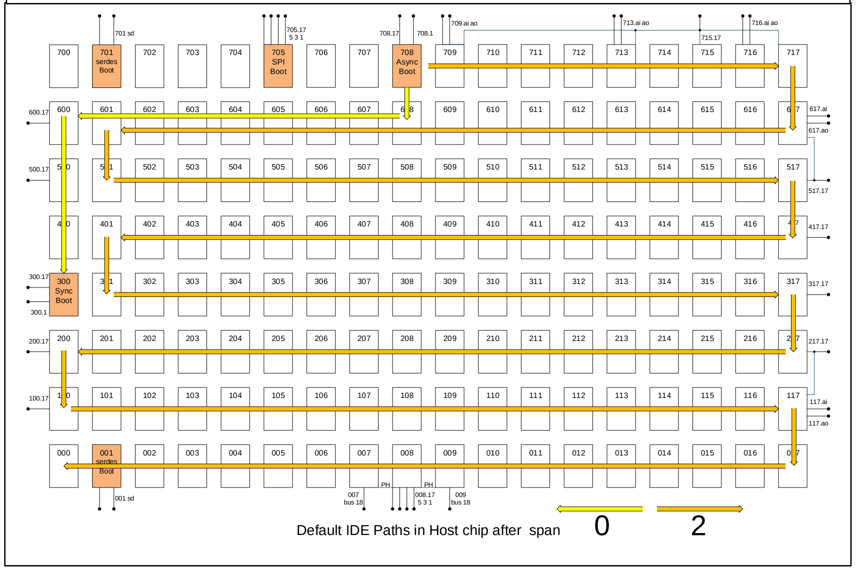
6.1.6 Default External IDE Paths


Default IDE Paths after host load

After loading HOST, BRIDGE or TARGET these paths exist in the chip to which the serial connection has been made. Path 1 is available for general use at that time.


Default IDE Paths in Target chip after span

After SPAN path 0 is generally used for target chip, starting on host and passing thru the bridge to cover the entire target as shown here


Default IDE Paths in Host chip after span

... and path 2 is generally used for the host chip. Path 1 is available for general use. Path 0 by default avoids passing through the top row of nodes on the host because in polyFORTH environment node 705 is programmed for SPI flash operations.

Because paths 0 and 2 share nodes one should be unhooked before the other is hooked through those nodes.

The port bridge only requires special code in nodes 300 and 10300.

6.2 Internal IDE

6.2.1 Terminology

6.2.2 Using the Internal IDE

Internal IDE Vocabulary

7. Preparing Boot Streams

While the external IDE may be used to load and interact with F18 code in any chip for hardware or software testing without requiring any other hardware or initialization on the part of the chip, higher level modules such as the polyFORTH Virtual Machine may require additional hardware and the loading and initialization of many nodes as well as perhaps external SRAM in a practically simultaneous manner. This is best done by generating a ***boot stream*** to do all of that.

Unless equipped with special ROM or booted using SERDES, our chips must be booted after reset by having a suitable *boot stream* made available to one of its *boot nodes* (such as async serial, 2-wire synchronous serial, or SPI flash nodes that are enabled for boot after reset.) A boot stream consists of one or more *boot frames*, which are data structures defined and processed by the boot nodes of our chips. Each boot frame contains zero or more words of data, a starting memory or port address at which the data are consecutively written, and a jump address to which control is transferred by the boot node after the frame has been processed. Every boot node defines a *concatenation address* to which a boot frame may jump to process another frame. After reset, a chip may boot itself from a slave device such as SPI flash memory, or it may wait to receive boot stream(s) on one or more of its enabled interfaces. For example, a daisy chain of subsidiary chips may be booted by a master chip using frames stored in the master's mass storage or read from its boot device (such as an SPI flash). As another example, the external IDE works by transmitting boot frames into the chip under test. arrayForth hosts can build and transmit streams of boot frames into a target chip for efficient booting (the old IDE based boot mechanism has been deprecated in arrayForth-3.)

7.1 The Streamer Utility

This utility is a set of tools for generating and packaging ***boot streams*** that consist of one or more ***frames***. In a boot stream, each frame has a ***header*** to be interpreted by a ***boot node***, and will end either by returning to a simple function in the boot node such as its **warm cold** or its appropriate ***concatenation address***; in some circumstances the boot node will be given code to execute, which may read additional data from the boot stream and dispose of it in some way (for example, copying the polyFORTH nucleus into external SRAM) before completing its job and returning to such a place in the boot node, perhaps with a new ***starting address for the next frame*** in the case of SPI flash. Streams may also be generated for insertion into a node attached to the Snorkel, in which case no boot headers are necessary and special methods must be used if that first node is itself to be programmed.

The Streamer operates in two phases. In the first, it composes the entire stream as instructed, one 18-bit word per 32-bit cell in memory (in pF/144 we do this using unallocated space in the nearly 1 Megaword of extended memory.) In the second phase, the boot stream is converted into a form suitable for whatever medium is being used (for SPI it is a stream of 18-bit values expressed as a succession of bytes; for async serial each word is inverted and shifted into a 3-byte form equipped for auto-baud of each word. For delivery through the Snorkel, it is left unchanged.)

A minimal use of the Streamer consists of the following elements:

```
STREAMER LOAD            Compile the utility
  nn STREAM[             Specify root node and clear
  {a COURSE}             Override default path for that node
  FRAME[                 Begins a frame suitably for root node
     boot description --- see 7.1.1, Boot Descriptor Language (BDL)
  ]FRAME                 Ends a frame suitably for root node and,
                         if a forward reference was made, resolves
                         it at the current aligned flash location.
  ]STREAM                Completes a stream and packs it as needed
```

Here is the full STREAMER vocabulary, followed by the Boot Descriptor Language:

STREAM[(nn) If usage from root node nn has been defined, selects the header formulation and final packaging suitable for that node as well as a default path whose address is returned by **ENTIRE** after STREAM[has been used. Clears the entire stream buffer.

The node numbers defined for this use are at the time of this writing **207, 708, 300** and **705** . For node 708 we use asynch serial header and 3-byte asynch word packing. For node 300 we use synchronous serial header and don't pack the stream. For node 207 we generate no headers and do no packing for Snorkel injection into the chip. And for node 705 we use SPI flash headers, compress each 8 18-bit words into 18 bytes (9 16-bit words), and set special end-frame behavior for forward references because at the end of loading each stream node 705 must be instructed to read the next stream starting at a byte boundary.

COURSE (a) Overrides the default path selected by root to be used as ENTIRE at any time. In a multi-frame stream it will often be necessary to change the path, for example because an earlier frame some of the chip's nodes have been programmed. It is the nature of boot streams that the first node loaded will be the last one in the path. The first node in a path *must* be the number of the node from which it originates. The address given must be the tick of an ARRAY holding a sequence of nn numbers ending with -1. Default paths for the standard root nodes are named df708 thru df207 .

FRAME[begins generation of a frame. All descriptor tables are initialized with their default settings. The root node is either not programmable at all (node 207) or partly programmable (memory settings and /P are the only valid operations on a boot node itself)

]FRAME Ends a frame. Calculates the frame length and generates the appropriate frame header if any, followed by all port and memory pumps to load the nodes in the current COURSE .

/ROOT (jmp a n bin) is used outside of FRAME[ ]FRAME encapsulation to generate a boot frame for the root node, loading that node with n cells from the given bin and jumping to the address jmp .

FORWARD is used immediately after /ROOT when generating a stream for SPI flash to indicate that the program just loaded will access flash before resuming stream processing. When this is done the address register in the flash loses synchronization with bit stream reading. In order to start up correctly again, the stream is padded to the next zero modulo 8 18-bit word index in the overall flash image, and the code loaded in the root node is patched to provide the absolute byte address for continuation in a standard manner. See definition of FORWARD and the source code it is used with.

]STREAM ends a stream. Converts the stream in place to the appropriate form for the intended node of origin.

.STREAM dumps the current stream, which must be in unpacked numeric form.

?STREAM audits an unpacked stream against a reference stream, see below.

?FLASH audits a packed flash stream against a reference stream, see below.

STREAM (- a n) sF ONLY returns origin and length in octets of the converted stream. Normally used within other tools.

STREAM (- da n) pF/144 ONLY returns origin and length in octets of the converted stream. Normally used within other tools.

7.1.1 Boot Descriptor Language (BDL)

Automated loading *touches* all nodes in a defined path starting at the relevant root node. Automated softsim loading touches all nodes in a chip. By default, a stream does as little as possible to each node touched. One item is by necessity pushed onto each stack, and register A is altered. Register P is set to the appropriate IDLE multiport address for that node, and B is set to the address of IO for all nodes not otherwise loaded. BDL is used to specify non-default treatments using statements that start with the word **+NODE** and continue with phrases describing the needed initialization of memory and/or registers in the node. Nodes may appear in any order since these statements are simply filling tables for later use. For the same reason, **+NODE** statements are cumulative in that one **100 +NODE** phrase may set the memory loading and starting address for node 100 while another **`100 +NODE`** phrase interpreted later may override the starting address specified earlier. The following words constitute the BDL. Their use in context is described here and in the chapter on Softsim; examples may be found in the distributed source code.

`+NODE (nn)` Selects table entries for node number nn, in cyyxx notation. The values in the table entries are not changed by **`+NODE`** and so their values will be default unless a previous **`+NODE`** phrase has been interpreted for the same node.

`/RAM (bin)` Loads all of RAM from the given bin in cyyxx notation. *By default nothing is loaded*. Partial loads may be described using the following words. Up to three RAM load descriptors may be specified for each node. Such loads are applied to the node in the order encountered in the boot description, so that if there are overlaps the last descriptor will overwrite earlier ones. When a RAM load of all 64 words is specified, any previously declared RAM loads are deleted and will not appear in the stream. Thus it is possible to preset a default background, such as of Ganglia, and override that for nodes that won't be able to serve as Ganglia, without extending stream length.

> **`/SOME (s d n bin)`** Loads part of RAM, **`n`** words from the given **bin**, starting at address **`s`** in the bin, are loaded into the node's RAM at address **`d`**.
>
> **`/PART (a n bin)`** Loads part of RAM, equivalent to `a a n /SOME` .

`/B (d)` Specifies an initial value for register **`B`**. *By default B is set to IO.*

`/A (d)` Specifies an initial value for register **`A`**. *By default a port call instruction is left in A.*

`/IO (d)` Specifies a value to be loaded into the **IO** register. *By default IO is not altered*.

`/STACK (<n double values> n)` Specifies up to ten values to be pushed onto the data stack, with the rightmost value on top. For example `30 20 10 3 /stack` produces the same effect as though a program had executed code **`30 20 10`**.

`/RSTACK (<n double values> n)` Specifies up to nine values to be pushed onto the return stack, with the rightmost value on top. For example `30 20 10 3 /stack` produces the same effect as though a program had executed code **`30 20 10`**.

`/P (a)` Specifies an initial value for register P . Default value is xA9 which is the routine warm in every node's ROM. When the node is started, warm jumps to the appropriate multiport execute for the node's position in the array. If you wish to leave the boot routine enabled in a boot node that is touched by an automated loading procedure, specify its P value as xAA as is done by reset for such nodes

`!ND ( nn)` Macro to load node nn with code from same bin.

`+ND ( a. b. p nn)` Macro to load node nn with code from same bin and initialize registers.

`ITS (nn _ - a)` Returns the value of the label whose name follows in bin **`nn`**.

`RIGHT LEFT UP DOWN IO LDATA RDATA` Register names returning double addresses in I/O space.

7.1.2 Stream Structure

The components used for building frame content differ from those used in the External IDE. This section includes information for calculating worst case frame length. For calculations based on node quantity, we assume a bridged pair of chips containing a total of 288 nodes, two of which (300 and 10300) are inaccessible because they are the bridge, and the boot node itself which is loaded in a separate frame; so the frame will visit a new of 285 nodes in the full, 2-chip path.

A frame begins with a 3-word header unless it isn't needed. The first node receiving data via the frame body is given a focusing call that must be present in the stream; subsequent nodes get their focusing calls from the pump in the preceding node. So we begin with 1 or 4 words and add to it modules of the following kinds.

7.1.2.1 Port Pumps

A frame begins with one port pump for each node except the last in the path. The port pump is still five words but uses **A** rather than **B** to point to the port at which the pump is directed; thus the port pump clobbers A, and it also has to push one word on each stack. For n nodes in path after the root, n-1 pumps are required; for two chips n-1 is 284 so 1420 words of pumps add to the 4 above for 1424. Structure within the stream:

```
   04DAF     @p dup a!  @p  (preceded by focusing call)
   12xxx     Literal, the focusing call given to next node
   lng-1     Literal, number of words following pump -1
   2FAB2     >r !
   05A72     begin  @p !  unext
```

Following the sequence of port pumps, there is a Memory Load module and a Post-Load Initialization module for each node beginning with the last node in the path and proceeding backward along the path to and including the first node after the root. When a node given a port pump is finished with pumping, the next thing in the stream will be its memory load(s) (if any) and initialization (at least of P and IO.)

7.1.2.2 Memory Loads

Each memory pump is five words long followed by the text to be loaded. In the typical case there is one memory pump and 64 words of text for a total of 69 words. Complicating this, we permit combining up to three memory loads for a node, in which case one might expect still a max of 64 words of text but with two additional pumps, bringing the total to 79 words. 285 of these would be 22515 words, for a total thus far of 22939 although a smaller number is more common because normally there is only one memory load in effect for each node. Structure:

```
   04A12     @p a!  @p
   <adr>     Literal, start address of this RAM load
   lng-1     Literal, number of words following -1
   2E9B2     >r
   05872    begin  @p +!  unext
```

It is possible to describe an even worse case in which there would be three fully overlapping 64-word memory loads for each node, but this is not worth considering. We do make provision for all nodes to be initially loaded as a background default by such things as the Ganglia, but to avoid letting this run up the size of the frame we check in the RAM load specifiers for full 64-words descriptors and if one is encountered all previous RAM load descriptors for that node are deleted so that they will not be present in the frame.

7.1.2.3 Post-Load Initializations

Initializing IO costs 4 words; A and B cost 2 words each and P costs 1. Initialization of each stack takes 2 words per value. We could fight for a reduction on the stack initializations but the benefit would appear to be marginal. The initialization components are structured as follow:

```
Set IO   04BB2     @p b!
         0015D     Literal, address of IO
         05BB2     @p !b
         value     Literal
Set A    04AB2     @p a!
         value     Literal
Set B    04BB2     @p b!
         value     Literal
Push R   048B2     @p >r
         value     Literal
```

```
Push S   049B2     @p
         value     Literal
Set P    10xxx     Jump to start address
```

P is (and must be) always set, so the minimum module is 1 word; for consistency with colorForth, we also always initialize B, to IO if nothing else is specified in the BDL. To initialize everything possible we have 9 words for the discrete registers and 38 words for all of the stacks, for a total of 47 words worst case and 13395 for two chips. This brings us to 36334 words so far for a practical worst case 2-chip frame.

That number pushed us across a 64k byte boundary and so we have chosen to reserve the first 128 kB of flash media for boot stream text.

7.1.2.4 Root Node Programming

The root node may simply be programmed as the last step in a complete boot stream. Earlier in the stream it may be necessary to program the root node to do other things such as for example speeding up the flash clock timing, or copying data from flash to an external SRAM as is done in booting eForth or polyFORTH.

Such intermediate programs for the root may be generated using `/ROOT` which does not employ BDL, or included in the BDL for a stream in which case the streamer generates two frames for the boot path (one for all nodes but the root, and one for the root itself.) When such a program will be accessing other flash during its operation and then resuming stream processing, the word FORWARD should be used after]FRAME in a flash boot stream to pad the stream to the next even byte boundary in flash, and to patch the root node program for a forward reference to the padded restart address.

Only one memory load may be specified for a root node, and it must have the same source and destination offsets if the load is specified by /SOME . Although it is possible to initialize more than RAM and P in a root node, this process would be complex and we have not built any particular such mechanism into the BDL. None of this is relevant for node 207, but it is true for any real boot node.

7.1.2.5 Special Considerations for SPI Flash

The SPI node processes standard boot frame headers which may be concatenated so long as the flash is not being accessed in any other way during booting. This stipulation is necessary because the end of a boot frame may occur on any even bit boundary within the flash, hence 0, 2, 4 or 6 bits into an addressable byte. A straight concatenation may begin on the next bit boundary to be read. A standard BDL frame will actually generate two concatenated boot frames: One to load all nodes but the root, and the next to load the root node and begin execution. The second frame is actually generated using /ROOT but is derived from the limited BDL allowed for the root node. Initialize P to the concatenation address `[ 705 ITS spi-exec ]` if the root is not going to be otherwise accessing flash and there is to be another concatenated frame.

In complex flash boots there may be several logical steps, and some of them may involve programming the root node to operate on the flash. Examples include reading the polyFORTH nucleus from flash and writing it into external SRAM, and accessing a table of parameter values to be written at specified places in various nodes before booting the rest of their memory. In these cases the bit alignment of the end of the previous boot stream is lost and a new starting address must be written into the root node's memory so that after that step is complete the normal boot code may be started at a byte-aligned place following the end of the previous boot stream. This address, within the flash, is only known after the boot stream image has been padded, and FORWARD is used to patch the root node program with this address information.

To build an SPI boot stream that is unpacked for debugging purposes, use the phrase `0 S-END !` within `STREAM[ ... ]STREAM`.

7.1.2.6 Special Considerations for Serial boot

When preparing a 2-chip boot stream for serial delivery through node 708, it's infeasible to visit node 000 on the first chip as part of the main stream. So, if you need to program this node in any way it must be done as part of the bridge construction. The 2-chip serial pF stream generator we provide loads a ganglion into node 000. Copy the BDL for your node 000 here if you need it.

7.2 Transmitting Boot Streams

You must generate a boot stream for the particular root node you plan to present it through. While the BDL may be the same for chips to be booted through various nodes, the text of the boot streams differs because the paths necessarily differ for each root node.

The vocabulary for injecting these streams depends on the system you're running on and on the method to be used. These methods are as follows:

7.2.1 From saneFORTH

7.2.1.1 Asynchronous Serial

sF uses its external IDE to inject asynchronous boot streams into node 708 of the HOST chip

`S-ORG S-LNG  HOST LOAD  !STREAM` resets the chip and pumps the stream just built into it through boot node 708. Returns when all words have been transmitted. There is no flow control so completion does not prove success.

`!stream` is used when the serial port is already set up and the chip has already been reset. Used for second and subsequent streams when multiple streams are being injected.

`!NUCLEUS` transmits the polyFORTH nucleus from block zero of the pF serial disk when initializing RAM over serial port.

The asynchronous boot stream can also span both host and target. Upon completion the external IDE remains loaded and, if the stream in question left IDE code in node 708 with proper P setting, the external IDE may still be used with care and due considerations for what paths are feasible after the stream has been loaded. Use `Talk` instead of `TALK` to use existing serial port and avoid resetting the chip.

7.2.2 From pF/144

7.2.2.1 Internal Streams rooted at node 207

Streams may be injected, at very high speed, directly into the host chip after being built:

`!SNORK` pumps the stream just built out through the Snorkel into node 208 or 307 as indicated by the path in effect at the start of the stream. Returns when all words have been absorbed by at least the first node in the path.

Here is an example that instructs node 715 to emit an approximately 75 MHz signal on pin 715.17, assuming that nothing but ganglia have been loaded into any of the nodes along its path:

```
2121
( Descriptor test)
ASM[ # 715 NODE ERS # 0 org
   begin begin  !b unext unext   >BIN ]ASM

0 ARRAY MYP  207 ORGN 210 TO 710 TO 715 TO -1 ,

207 STREAM[   ' MYP COURSE
   FRAME[   715 +NODE  0 1 715 /PART
   -1. -1. -1. -1.  -1. -1. -1. -1.  -1.  9 /RSTACK
   x20000. x30000.  2OVER 2OVER  2OVER 2OVER  2OVER 2OVER
                    2OVER 2OVER  10 /STACK  IO /B  0 /P
   ]FRAME  ]STREAM
!SNORK
```

This example instructs node 500 of the target chip to emit a similar signal on its pin 500.17, assuming that the bridge has already been installed and activated, and that there are no obstructions along the path. Note the use of ORGN to make the hop between nodes 400 and 10400:

```
2122
   ( 2-chip test)
ASM[ # 10500 NODE ERS # 0 org
   begin begin  !b unext unext   >BIN ]ASM

0 ARRAY MYP  207 ORGN 407 TO 400 TO  10400 ORGN 10500 TO  -1 ,

207 STREAM[   ' MYP COURSE
   FRAME[ 10500 +NODE  0 1 10500 /PART
   -1. -1. -1. -1.  -1. -1. -1. -1.  -1.  9 /RSTACK
   x20000. x30000.  2OVER 2OVER  2OVER 2OVER  2OVER 2OVER
                    2OVER 2OVER  10 /STACK  IO /B  0 /P
   ]FRAME  ]STREAM
!SNORK
```

For the next example, we begin with a flash boot that does NOT load anything but polyFORTH and, if desired, the bridge. This demonstrates that polyFORTH can load the ethernet cluster from within a live chip. It is a torturous procedure because we must initially do a 9 LOAD for serial clock (HOME 9 LIST and edit is the best way to go) :

```
2123
( Add Ether to pF)
STREAMER LOAD  1 CONSTANT ?CLK   1 CONSTANT ?ETH

0 ARRAY MYP  207 ORGN 208 TO 108 TO 110 TO 10 TO 17 TO
   617 TO 616 TO 116 TO 115 TO 415 TO 414 TO 114 TO 111 TO -1 ,

207 STREAM[   ' MYP COURSE
   FRAME[  ( Clock/Ether)  ENIC 1+ LOAD  ]FRAME  ]STREAM
!SNORK


4523
Assumes flash boot only has polyFORTH and, if desired, bridge.
   This loads and activates the ethernet NIC Mk1 and the 10MHz
   clock.  Because of the latter several steps are involved:

   1.  Reset and hit space.
   2.  HOME 9 LIST and edit to use serial clock.  HI
   3.  AFORTH  2123 LOAD  ... and link should become active.
   4.  RELOAD  hit space
   5.  HOME 9 LIST and edit to use Ethernet clock.  HI
   6.  ETHER LOAD

ETHER may in fact be loaded on top of AFORTH, or presumably
   vice versa, but conflicts on double constants may exist.
```

7.2.2.2 Asynchronous Serial to Another Chip

7.2.2.3 Synchronous Serial to Another Chip

7.3 Burning Flash

By default, when generating a stream for node 705, both saneFORTH and pF/144 pack the stream into bytes that may be written to a flash memory starting at its absolute location zero. This section describes the procedures for writing that stream image into a local or remote flash.

7.3.1 Burning Flash from saneFORTH

7.3.2 Burning flash from pF/144

Code is provided for burning one's own flash (for example, the SPI flash that boots the Host chip on an Evaluation Board) or for burning the flash on another chip with which pF is communicating using external IDE.

7.3.2.1 Burning pF's Own Flash

Burning one's own flash is straightforward; all we need do is to copy the stream image to the front of the flash starting at absolute block zero (normally mapped at 40 DRIVE). After generating the stream, simply execute the following word to rewrite flash boot on your own system:

`WRITE-FLASH` copies the packed boot stream just generated to the boot area of the current system's flash, at absolute block 48000 and regardless of the current OFFSET .

Warning! This procedure assumes the flash boot area begins at absolute block 48000 in the `UNITS` mapping that is currently active on the pF/144 system. If you are using some other mapping you will need to attend to the definition named **)FLASH** .

7.3.2.2 Burning Flash on Another Chip

7.3.3 Erasing Flash

Flash needs to be erased before being written. This is done with the word `ers` which expects two parameters: A starting address and a number of *bytes* to be erased.

Example: **`0 8092 ers`**

The smallest unit of flash that can be erased on the Evaluation Board is 4K bytes, so the starting address should be on a 4K byte boundary and that's where the erasing will start. The number of bytes erased will be rounded up to the nearest 4K as well. *Note: Currently the whole flash is always erased. However, `ers` should be called with correct arguments for upward compatibility.*

7.3.4 Writing Flash

There are two streamer commands for writing flash: `burn` and `18burn` .

`burn (s d n)` writes flash in 16 bit units. `s` is the *word* address of a buffer of 16-bit words in host memory, `d` is byte address of destination in flash, and `n` is number of 16-bit words to write.

`18burn (s d n)` writes flash in 18-bit units. `s` is the word address of a buffer in host memory with a sequence of 18-bit values stored in consecutive 32-bit words, `d` is byte address of destination in flash, and `n` is the number of 18-bit words to write.

`burn` is used for 16 bit data such as the polyFORTH nucleus or the eForth kernel. Both are virtual code for virtual machines that run out of 16-bit SRAM.

Example: **`<filebuf> x8000 <filelen/2> burn`**

This example burns <filelen/2> words to flash at byte address x8000 from the given buffer.

`18burn` is used to write 18-bit data such as boot streams. The word `stream` returns the beginning address in host memory and the length in 18-bit words describing the output of the `framer` utility. Having built a boot stream, burn it into flash at address 0 as follows:

Example: **`stream 0 swap 18burn`**

7.4 Auditing Streams

When working with new stream structures it's useful to compare the stream just made with a reference. Mechanism for doing this is included in STREAMER . Standard areas involved in these operations are 4740, where a packed flash stream may be saved, and 2340, where an unpacked stream of up to 60 blocks (14760 words) may be saved.

`UNPACK (n)` unpacks the given number of blocks of flash stream from 4740 to 2340.

`STASH` saves the streamer's current stream buffer to 2340, limited to 60 blocks.

`?STREAM` dumps the current ***unpacked*** stream buffer in hex, highlighting each word that differs from the reference stream in the area at 2340. In saneFORTH the reference stream is at 2340 on the serial disk used by pF/144.

`?FLASH` dumps the current stream buffer in hex ***packed for flash***, highlighting each word that differs from the current boot flash. In saneFORTH the reference stream is at 4740 on the serial disk used by pF/144.

7.4.1 Getting Streams from Flash

The simplest way to obtain a stream made by colorForth is to burn the stream into flash from colorForth, boot to it, and then copy it to the comparison area. With standard mapping and when BLOCK takes a signed number, this can be done as follows (two steps are required because the boot area is more than 32k blocks from the working flash:

1. 0 DRIVE DISKING LOAD
2. 20 DRIVE 24000 4740 nn BLOCKS
3. 0 DRIVE
4. 24000 4740 + 4740 nn BLOCKS

7.4.2 Getting Streams from colorForth

When converting an existing body of colorForth based code to aF-3, it is prudent to use the colorForth stream for loading that code as a reference. The following steps accomplish this:

1. In colorForth, **`compile`** all of the code of interest.
2. Say **`bnamed mystream`** (or other name of your choice) to output file name.
3. Edit the stream generation block to abort with a bad word after the word stream
4. Generate stream, let it abort and note the address and count found there.
5. Say **`32768 nnc + nnc + 4800 wback`** to write a 4800-block file starting with the stream. Only the first few blocks are relevant, but this simplifies file mapping into the sF system.

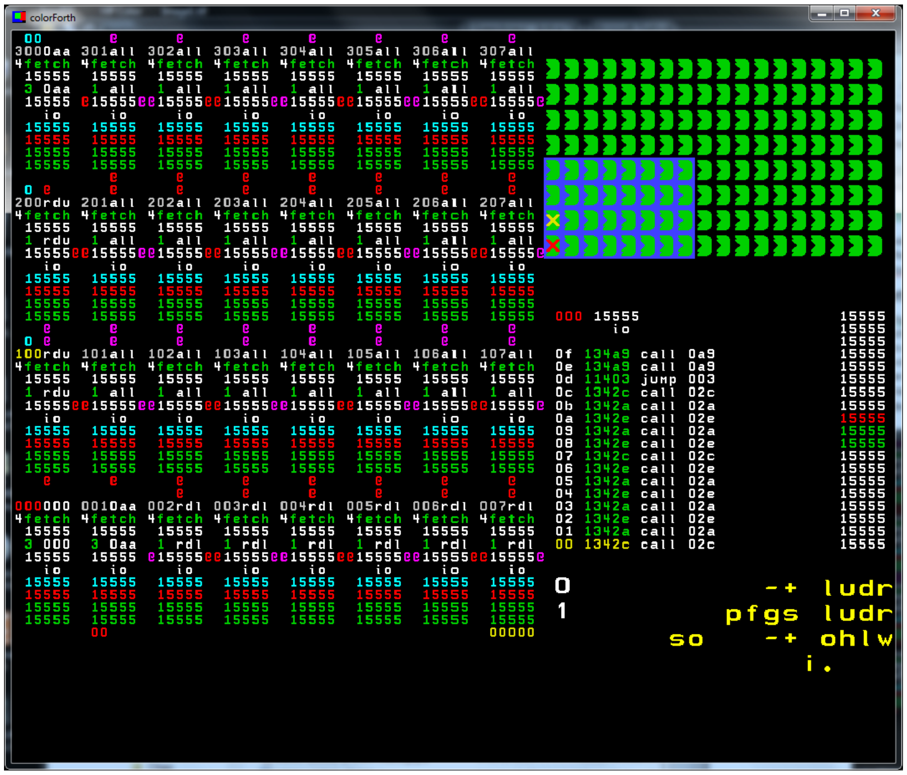
6. Move this file into the pf directory
7. On the sF system, in block 149 temporarily replace **`OBJ-AF3`** with your file's name, **`FLUSH`** and **`RELOAD`** .
8. Use utility not written yet and conventions not made yet to capture and save the stream.
9. In block 149, restore the **`OBJ-AF3`** file name, **`FLUSH`** and **`RELOAD`** .
10. Additional steps to be determined.
11. Remove the change you made to the stream generation block in colorForth.

8. Simulation Testing with Softsim

Softsim (the Software Simulator) is a program that simulates the actions of Green Arrays computers, ***in a single GA144 chip***, at a pretty high level. It takes short cuts that allow it to run much faster than a full hardware simulation. The following is a quick tutorial in how to use softsim.

8.1 Getting Started

After starting arrayForth, type **so** at the command line. You should see a colorful display. In the default layout shown below, there are three sections which help you "drill down" into the chip. In the upper right corner is a *full chip view* showing its 144 computers as an 8x18 array. On the left is an *overview* showing the major status and registers of a rectangular section of the chip's nodes. This can be up to a 4x8 array as shown. The overview section is highlighted in the full chip display. In the lower right part of the display you'll see two white numbers in the larger font which represent "time" (the current step number) and "gap" (the number of steps between display updates). Further to the right you'll see yellow keyboard hints.



```
 00        @      @      @      @      @      @      @
3000aa 301all 302all 303all 304all 305all 306all 307all
4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch
 15555  15555  15555  15555  15555  15555  15555  15555
 3 0aa  1 all  1 all  1 all  1 all  1 all  1 all  1 all
 15555 @15555@@15555@@15555@@15555@@15555@@15555@@15555@
   io     io     io     io     io     io     io     io
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
        @      @      @      @      @      @      @
 0 @    @      @      @      @      @      @      @
200rdu 201all 202all 203all 204all 205all 206all 207all
4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch
 15555  15555  15555  15555  15555  15555  15555  15555
 1 rdu  1 all  1 all  1 all  1 all  1 all  1 all  1 all
 15555@@15555@@15555@@15555@@15555@@15555@@15555@@15555@
   io     io     io     io     io     io     io     io
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
   @      @      @      @      @      @      @      @
 0 @      @      @      @      @      @      @      @
100rdu 101all 102all 103all 104all 105all 106all 107all
4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch
 15555  15555  15555  15555  15555  15555  15555  15555
 1 rdu  1 all  1 all  1 all  1 all  1 all  1 all  1 all
 15555@@15555@@15555@@15555@@15555@@15555@@15555@@15555@
   io     io     io     io     io     io     io     io
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
   @      @      @      @      @      @      @      @
                 @      @      @      @      @      @
000000 0010aa 002rdl 003rdl 004rdl 005rdl 006rdl 007rdl
4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch 4fetch
 15555  15555  15555  15555  15555  15555  15555  15555
 3 000  3 0aa  1 rdl  1 rdl  1 rdl  1 rdl  1 rdl  1 rdl
 15555  15555 @15555@@15555@@15555@@15555@@15555@@15555@
   io     io     io     io     io     io     io     io
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
 15555  15555  15555  15555  15555  15555  15555  15555
        00                                        00000
```

```
000  15555                          15555
       io                           15555
                                    15555
0f  134a9  call  0a9                15555
0e  134a9  call  0a9                15555
0d  11403  jump  003                15555
0c  1342c  call  02c                15555
0b  1342a  call  02a                15555
0a  1342e  call  02e                15555
09  1342a  call  02a                15555
08  1342e  call  02e                15555
07  1342c  call  02c                15555
06  1342e  call  02e                15555
05  1342a  call  02a                15555
04  1342e  call  02e                15555
03  1342a  call  02a                15555
02  1342e  call  02e                15555
01  1342a  call  02a                15555
00  1342c  call  02c                15555

0                           -+  ludr
1                         pfgs  ludr
                   so       -+  ohlw
                                 i.
```

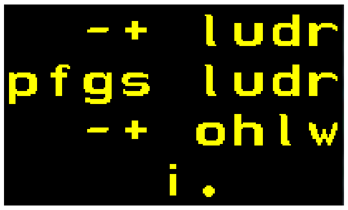
Immediately below the full chip view there is a *detailed view* of a single node, called the *focus node*. Its node number, 000, is shown in red at the top left of this view. This node's position is shown in the full chip view with a red "X". In the overview, the focus node's three-digit number, normally shown in grey at the upper left of each node's rectangle, is changed to red for the *focus* node.

In addition to the focus node, a second node, called the *other node*, may be selected as well. Its detailed view may replace the full chip view using the **w** key on the control panel, and the *focus* and *other* node numbers may be exchanged with the **o** key on the control panel. The other node, if visible on the chip, is marked with a yellow "X" in the full chip display, and with yellow node numbers in the overview and detailed view.

All sections of the display are animated during simulation. In the full chip view, each node is green when active and grey when suspended.

8.2 Navigating

Your right hand is the main control for what you are seeing on the display.



```
  -+  ludr
pfgs  ludr
  -+  ohlw
    i.
```

Now look at the keyboard hints in the lower right corner. The **ludr** hints mean left, up, down, and right. The middle row's **ludr** keys change the current focus node, changing the red node number in the lower right detailed view, and moving the red markers in overview and full chip view. It's possible to move the red marker off the screen, but you will still know where it has gone by looking at the detailed view on the lower right. Try it and see.

Now try pressing the **ludr** keys in the upper row of the hints. This will move your overview window around inside the chip. You'll see the grey node numbers change as you press these keys, and the highlighted rectangle in the full chip view moves correspondingly. The focus and other nodes do not change when the overview is moved.

Suppose you want to watch two nodes interacting in more detail. That's why the *other* node exists. The middle row **ludr** keys only affect the focus node in the lower right, but you can swap the focus and other nodes by pressing the **o** key on the control panel. Try it. You'll see the red and yellow nodes changing roles. Now you can choose two nodes to watch closely (use the **w** key to replace the full chip view with the *other* node's detailed view.)

In addition to the full stack display (deepest item of return stack on top, , each detailed view shows a sixteen-word memory dump. You can navigate through the memory dump of the focus node by pressing the **h** (higher) and **l** (lower) keys on the control panel. These move through RAM and ROM in 8-word steps.

8.3 Making it Go

Let's call the two large, white numbers in the larger font in the lower right *time* and *gap*. *time* is the number of steps that have occurred since the start of the simulation. *gap* is the number of steps that will occur between display updates. *gap* starts as 1, so the display is updated after each single step. There is another variable named *fast* which can be swapped with *gap* if you want the display to run faster and show fewer updates. This variable can be increased or decreased by 1 using the **+** or **-** control panel keys in the bottom row, or by 100 using the **+** or **-** keys in the top row. The **f** key toggles the value of *gap* between *fast* and 1. The simulation runs very much faster when it's not updating the display.

The **g** and **s** keys in the left hand control the progress of the simulation. Press and release the **s** (step) key. You should see time change from 0 to 1 and some changes will occur in the overview as well.

When you press the **g** (go) key, the simulation will go on running and updating the display until you press another key. Be careful because whatever key you press will also be take its own effect. If you just want to stop running the simulation press the **s** key. The simulation will stop after one more step and nothing else will happen after that until you press another key.

The **p** key is for power. It simulates a power on reset, starting the simulation over from the beginning and setting time back to 1.

The **i** key runs the current instruction in the focus node to completion rather than just a single time step. It also initiates *instruction word tracking* in the memory display. When tracking is active, the currently executing instruction word is always visible in the lower 8 words of memory display. Pressing **g** for go will not change the state of tracking. Pressing **s** for step turns tracking off, as does navigating the memory display.

When you're done with softsim or need to use the interpreter, press the **.** key (space bar). That returns you to the colorforth interpreter.

8.4 The Memory Dump and Decompilation

Each detailed view shows a combination memory dump and decompilation/disassembly of the code for a node, along with a dump of both stacks. The A and B registers appear at the top next to the node number. The stacks are shown as a single array with the return stack growing upward from R (in red) and the data stack growing downward from T and S (in green). The memory dump shows the addresses of memory words in the left column. The instruction word or 18 bit data word follows in green. The instructions for each slot appear next, sometimes followed by the value of an address field. While a node with detailed view is executing, the line where the current instruction word was read appears in red. The instruction in the current slot is also in red. The other instructions remain white. The line that the P register points to is shown in yellow. If the instruction word has been fetched from a com port the disassembly and instruction word are shown in red above the memory dump, below the B register.

8.5 The left side display

In the array of nodes on the left each rectangle contains a list of the values of registers and opcode names representing the current state of that node. Most of the numbers have identifying colors which turn to grey when the node is suspended. You'd probably like to know which registers the numbers represent. Each node has twelve lines.

From top to bottom, using node 100 from the above display as an example, they are:

Row	Example	Name	Description
1	`0 @`	Pins and/or ports	States of the pins are represented by 0 or 1 for low or high, yellow for output and cyan for input. Blank when the node has no external pins. Analog and Serdes pins are red. Pins are on the top for top nodes, bottom for bottom nodes, and on the outside for side nodes. If the node has a port on this side, see row 12 for its representation
2	`100rdu`	NODE/Abus	The grey node number followed by the white address bus. Com port addresses appear as three characters including r, d, l, u, or 'all' if it's a four port address.
3	`4fetch`	SLOT/OPCODE	The white slot number is followed by the green opcode name.
4	`15555`	I	White, the instruction register
5	`1 rdu`	M/P	The memory timer counts down in green. '-' indicates that the node is suspended. The value of P, the program counter, follows in white.
6	`15555`	A	White, 18 bit pointer register A.
7	`io`	B	White, 10 bit pointer register B.
8	`15555`	IO	Cyan, the write-only part of the IO register.
9	`15555`	R	Red, top of return stack.
10	`15555`	T	Green, top of data stack. Carry latch shown as high order **.**
11	`15555`	S	Green, second on data stack.
12	`@`	Pins and/or ports	When a com port is being read from or written to it is represented on the appropriate side of the node with a "@" for reading and a "!" for writing. The Right and Down ports are in red while the Up and Left ports are in Magenta. Pins appear here for top and bottom edge nodes.

8.6 Getting your Code to Run

By default when softsim is loaded each node's memory (RAM and ROM) is initialized with the code most recently compiled into the bin whose number is the same as that of the node. Again by default, register **B** is initialized with the address of **io** and register **P** is initialized the same as the hardware does after reset (see chip data book.) Thus, without any further arrangements on your part (and without the SMTM demonstration program that's started in block 1234 as delivered), the simulated chip will only run boot nodes' initialization code and soon all nodes will be suspended. There are several general ways in which to get the chip started.

The first method is closest to the actual operation of the chip. This is done by building a testbed that directly simulates one of the four possible boot sources for the chip (SPI flash, async serial, sync serial, or SERDES) and allows the simulator to boot a complete application just as the chip does. This can take a very long time and you will probably prefer to use a more direct, if slightly less realistic, method.

The second method is to force one or more nodes to execute code from RAM. As indicated above, if you simply compile code for node 105 that code will appear in node 105's simulated RAM unless you have indicated otherwise. To run this code simply specify any desired starting address for the node. By using the boot descriptor syntax you may fully control what code is loaded into each node, what if any register and/or stack initialization is done, and what starting address is loaded into **`P`** . As of rev 1g, softsim will process standard Boot Descriptors (see 7.1.1 above) and may therefore be used to simulate an application of any size using the same initial conditions as the IDE or stream loaders create in the real chip, using the same specifications.

A third method may be used by compiling your own ROM code (simply compile code with location counter in ROM). This method is useful for testing proposed new ROM code, including application specific ROM.

8.6.1 Softsim Example Program

For demonstration purposes block 200 compiles a short program into node 0 that will copy itself into neighbor nodes along a predetermined path. As delivered, block 216 has a line reading **`0 +node 0 /ram 0 /p`** . This tells softsim that when it starts node 0 should begin executing at address 0. Softsim already has the code for bin 0 in node 0, so **`0 /ram`** serves only as documentation. Type **`so`** to start softsim. When softsim starts for the first time node 0 will be the focus node. It should be marked with a red X in the lower left corner of the full chip view. Notice that address 0 in the memory dump is yellow. That means the **`P`** register contains a 0 and the program will fetch its first instruction from there. Press the **`i`** key repeatedly and watch the program as it makes a call to address x02c. Note that pressing the **`i`** causes the memory display to track the instruction word, so it automatically navigates to address x02c. Pressing **`s`** instead would leave the memory display at address 0. Press the **`f`** (fast) key. The "gap" should show as 100. Press the **`g`** (go) key and watch the module migrate from node to node.

8.6.2 Breakpoints

Each node may have one and only one breakpoint set using the word `<slot> <addr> <node>` **`break`** . When that node is about to execute the instruction from addr/slot softsim will stop and paint a colored box around that node to identify the reason for stopping. A cyan colored X will appear on the same node in the full chip view as well. Those markers will disappear as soon as another step is taken by pressing **`s g`** or **`i`** .

The breakpoint may be removed via the command `<node>` **`-break`** . Breakpoints may be set or cleared at any time interactively.

8.7 Testbeds

In order to test your simulated program's interaction with the outside world you will need to make a testbed to simulate inputs and outputs. There are two example testbeds in softsim as released, namely the SPI and Sync boot testbeds on blocks 1230 and 1244-1246. We'll start with a simpler example to build up to what you'll need to know in order to understand these testbeds and to create your own.

Unfortunately, testbeds are written and execute in x86 colorForth. Everywhere else in this manual we have been able to avoid discussing the peculiarities of the x86 system's vocabulary, but here you will be using it. Follow examples and contact customer support for assistance if needed.

The softsim engine uses vectored execution to run testbed code. An execution vector is just a variable that contains the address of some code to be executed. If you've stored the address of a code snippet into a variable, say `this` for example, then you execute the snippet with the phrase **`this xqt`**. The softsim engine has a couple of node variables, local to a particular node, which are meant to contain the addresses of testbed code. The first is called **`softbed`** and holds the behavior of the testbed after reset. The second is **`'bed`** which holds the behavior to be executed in the next time epoch. The address in `softbed` is copied to `'bed` when softsim is initialized or restarted. If your testbed is best defined as a state machine, you may assign the code for each major state to `'bed` in each step (a vastly superior way to mechanize state machines than case statements).

It's simple to do this. Use the arrayForth word **`assign`** to store the address of the code following `assign` into the vector. Here's an example of its use. Suppose you want to simulate a fast square wave on pin 17 of node 600:

```
/wave softbed assign  time @ 10 / 1 and ?v p17v ! ;
```

```
600 !node /wave
```

Add this code to block 216 where testbeds are compiled. Run softsim and watch the pin in node 600 toggle every 10 steps. The first line defines the testbed. The second line attaches that testbed to node 600 interpretively. **`600 !node`** makes node 600 the current node with regard to node variables. When **`/wave`** is executed in that interpretive phrase, the address of the code following `assign` is stored into softbed. When softsim is initialized (by **`/softsim`**) that address is copied from `softbed` to `'bed` for node 600. Then for each step of softsim that code is executed for node 600.

The assigned code begins with a yellow **`time`**. `time` is a global variable which starts at 0 and is incremented for every step of softsim. *`time` is literally the very rough timebase for softsim (note that it does not correspond directly to any particular unit of time but is a necessary evil in order to simulate the chip's operations in an orderly way. Each step is on the order of T_{SLOT} (1300 to 1650 nS at 1.8v and 22°C, see G144A12 Chip Reference "Typical Instruction Timings.") The softsim engine allocates more than one step for longer instructions but only in integer multiples. Coupled with the fact that each node may have a different T_{SLOT} than its neighbors due to variations in silicon process, temperature due to its duty cycle of operation, and accumulated aging of each node which varies by lifespan duty cycles, timing in steps must be taken with a grain of salt. In future versions of softsim we might make a wall-clock time base for testbeds to use and allow the step time to be specified for a given run, allowing for boundary condition testing; if you wish to do this yourself now, do arithmetic on `time` before using it in the testbed.* For node 600 at each step the value of `time` is fetched and divided by 10. If the result is odd the pin goes high; if even the pin goes low. The word **`?v`** is part of softsim and turns a truth value into a voltage. If the input is 0 it remains 0; If it's non-zero (true), the value of the global variable **`vdd`** is returned. This is 1800 by default, in units of millivolts. **`p17v`** is a node variable. It contains the voltage on pin 17 for the current node. Since `/wave` is only executed for node 600 this code will set the voltage on 600.17 as a function of time; nominally a square wave whose period is roughly 28 nanoseconds. Node variables **`p1v`** **`p3v`** and **`p5v`** also exist for nodes with two or four pins.

In the Sync boot testbed on block 1230 `time` is not used as the timebase. Instead a global variable named **`dly`** was created to keep time. Also note that instead of using `?v` to get a voltage value `vdd @` is used in yellow. Global variables in x86 colorForth need to be yellow but are usually followed by a green `@` or `!`. In this case we use a yellow `@` to fetch the value from `vdd` at compile time. The constant value is then compiled as a literal by the yellow to green color transition. This is a feature of the x86 colorForth which doesn't apply to F18A code.

Another example might simulate a wire connecting two pins, 600.17 and 500.17. We will assume 600.17 is an output and 500.17 is an input.

```
/follower softbed assign  600 !node p17v @ 500 !node p17v ! ;
```

```
500 !node /follower
```

This testbed will cause 500.17 to have the same state as 600.17. Your program can set or clear 600.17 and code in node 500 can read 500.17 from the io register and act accordingly, as if a real wire connected the two pins. A fine point from this example is that for consistent timing this sort of testbed should usually be attached to the receiving node.

The SPI testbed in blocks 1244 and 1246 doesn't need a timebase. The testbed can simply react to changes in pin voltage set by the F18A program. It does this by reading the voltage on a pin via, for example, `p17v @` and interpreting the state of the pin as high when V_{IH} or V_{IL} thresholds are crossed. In fact it will be either 1800 mv or 0 mv for a GPIO pin in the present simulator. The situation is a bit more complicated because the testbed must pretend to be a device which responds to the SPI protocol. For example, the SPI device must wait until the enable pin goes low before paying attention to the clock pin. You can make such testbeds as simple or as elaborate as you require.

Here's a simple state machine that waits on a pin before taking some action.

Suppose you have a circuit that reads the signal on pin 708.17 and puts the inverted value on 708.1. Softsim already has the word **`low?`** which sets the x86 minus flag if its input is less that 900 mv. It's used as in `p17v @ low? -if ....` Note that `-if` is used, rather than `if` . The word **`high?`** below can be used as `p17v @ high? -if ...` similarly.

```
high? v negate vdd @ + low? ;
```

```
/waiter softbed assign begin
```

```
begin 'bed assign 0 p1v ! p17v @ low? until
```

```
begin 'bed assign vdd @ p1v ! p17v @ high? until end
```

```
708 !node /waiter
```

In this case we change the execution vector `'bed` to reflect changing conditions. The third line is a `begin until` loop. It first assigns the following code to `'bed` and exits. Next time this node's `'bed` executes pin 1 is set to 0 millivolts and pin 17 in read. If pin 17 is not low the `until` returns to `begin` and reassigns `'bed` to the same address, ending execution for this step. If pin 17 is low the code falls through to the next line where the values are reversed. Finally when pin 17 goes high again `end` jumps back to the first `begin` and the cycle starts over. Note that `assign` not only makes the assignment but also terminates execution with a return.

With these simple building blocks more complicated devices can be simulated by custom testbeds.

8.8 Interactive Testing with Softsim

Interactive softsim is intended to work very much like the interactive IDE. Skills learned in one should apply to the other, except that this mechanism can interact with a node to which no path could be built in IDE. The *remote opcodes* listed below have the same names as the words in the IDE and have the same results once you realize that they always work on the softsim's focus node. The command **`node`** will make any node become the focus node without having to navigate to it. Similarly the command **`other`** will make any node directly become the *other* node without having to navigate to it. These words may of course be used in scripting to automate test set-up.

Note that most of the remote opcodes names start with "r" for remote, to distiguish them from the PC colorForth words with the same names. Some of the opcodes have no PC equivalents and dispense with the "r" prefix.

The commonly used ports have names to make them easy to use. For example you can say **`io r@`** to read the **io** register of the focus node. You can also say (hex) **`30000 io r!`** for example in a node with an io pin and see the pin change state to a yellow 1. **`0 right r!`** writes a 0 to the node sharing its right port with the focus node. **`0 call`** starts the focus node executing code at address 0 in its RAM. It runs for three steps in order to fetch the instruction into the instruction register then leaves you in the softsim keyboard handler so you can single step or go to finish executing that code. The word **`lit`** takes the number on top of the host stack and pushes it onto the top of the focus node stack, just as it would in the IDE. To manually resume control panel operation, say **`ok h`** .

8.8.1 Vocabulary Reference

Host Commands

`node (nn)` sets node nn (yyxx form) as the *focus* node.

`other (nn)` sets node nn as *other*.

`call (a)` makes a call to address a in focus node and starts softsim control panel.

`boot (acn)` moves c words from address a of bin n into focus node RAM.

`lit (n)` pushes n from host stack onto focus node's data stack.

`r@ (a-n)` fetches a word from focus node address a onto host stack..

`r! (na)` stores n from host stack to focus node address a .

`break (slot addr node)` sets breakpoint for given node to stop before executing the opcode in the given 0-relative slot of the instruction word at the given address.

`-break (node)` clears breakpoint for the given node.

F18 Opcodes

`rdup (n-nn)` *all of these*

`rdrop (n)` *use node's own*

`rover (ab-aba)` *stack.*

`rpop (-n)` from return stack

`rpush (n)` onto return stack

`r- (n-n)` bitwise invert

`rand (nn-n)` bitwise AND

`ror (nn-n)` bitwise exclusive OR

`r+ (nn-n)` 18-bit add, *not in EAM*

`r+* (st-st)` multiply step uses **A**

`r2* (n-m)` left shift

`r2/ (n-n)` arith right shift

`ra! (n)` to register **A**

`ra@ (-n)` from register **A**

`rb! (n)` to register **B**

`!b (n)`

`!a (n)`

`!+ (n)`

`@b (-n)`

`@a (-n)`

`@+ (-n)`

Convenient Constants

`io (-a)` pushes address of io register onto host stack.

`data (-a)` up data register.

`ldata (-a)` left data register.

`right (-a)` right port.

`down (-a)` down port.

`left (-a)` left port.

`up (-a)` up port.

9. Practical Example

We have chosen a simple application to act as a practical example of how to develop and test an arrayForth program. The only parts we will need are those included in your EVB001 evaluation kit so you should be able to reproduce our results exactly. Our application is a simple PWM algorithm generating output to an LED.

Although simple, the PWM we will demonstrate uses a nontrivial approach. Many PWMs divide a fixed interval into a low period and a high period such that their sum is a constant period. The output value is the ratio of high to low time. The fixed maximum update rate derives from the period chosen. The resolution derives from the number time units the period is divided up into, usually a power of two.

PWMs have the benefit of generating an analog value from a digital output which is linear, assuming you can feed a perfect integrator. All PWMs force a trade-off between resolution and update rate. Usually this trade-off is fixed by the designer for any given application by choosing an inner loop timing interval and a number of intervals in the major period.

The algorithm we will demonstrate has several benefits over the classical design. The value presented for output is represented as a binary fraction between 0 and 1. The precision of the output is not affected by the inner loop update frequency which should always run as rapidly as attainable by the selected hardware. The higher the inner loop frequency the faster any given output will reach its desired average value. The maximum output rate is determined by period of the inner loop times the power of two represented by the least significant bit you have decided is important.

9.1 Selecting resources

For our example we will be using node 600 from the host chip because its output is easily accessible. We must move the jumper on J39 from 1-2 to 2-3 to expose host 600.17 output. This would affect automatic MMC access but will not interfere with access to the boot flash.

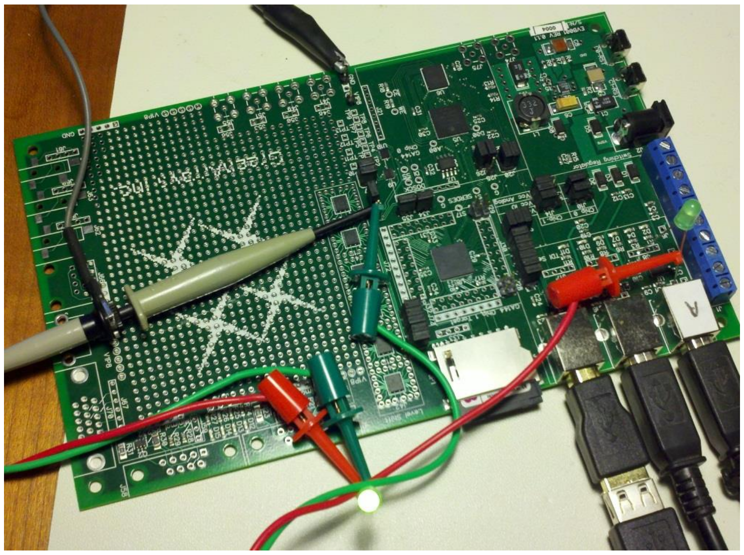
For brightness we will use 3.3v provided by one of the FTDI chips. Pin J7 is from the USB chip we will be using so it makes a good choice. We have chosen to solder one of our LEDs between J7 and J8-3 as a com input activity light. By connecting one of our clip leads to the anode of this LED we can pick up the 3.3v safely.

To minimize measurement interference we placed our scope between ground and J39-1. We have soldered stake pins to one of the ground areas near the prototyping region on our EVB001 and connected scope ground there.

We will use the default IDE hook path 0 which runs the perimeter counter clockwise. To make sure that node 705 is available right after reset be sure to keep J26 1-2 jumpered (for SPI no-boot select) whenever testing.

9.2 Wiring

See the adjacent image for a wiring example. We twisted the clip leads together loosely and connected one lead from the J7 USB LED anode to the anode of our free standing test LED. The other color clip lead runs from J39-1 to the test LED cathode. You may choose to place one of the resistors supplied between 3.3v and the LED to limit current when the LED is powered. We have chosen to leave it out in this demo as the measured voltage is just below spec and it simplifies the setup.



We can verify our wiring and check our assumptions by simple interactive use of the IDE.

1. Make sure the No-Boot jumper J26 is installed to avoid conflicts which might cause hangs.
2. Simply type **`host load panel`** to load the IDE.
3. Then type **`talk 0 600 hook upd`** to gain access to node 600 and display its stack.
4. Switch to hex input (see **Error! Reference source not found.**) and type **`20000 io r!`** to place the pin in s trong pull down and the LED should illuminate. A glance at the scope should show the pin to be near 0.5v due to maximum current draw through the LED.

Try typing the following in succession and observe the voltage on your scope:

- **`10000 io r!`** (weak pull-down)
- **`30000 io r!`** (drive pin high)
- **`0 io r!`** (high impedance)

9.3 Writing the code

The sample code we will show you has been placed in block 842. This is part of an open range of blocks that you may use freely. To make sure that our work is automatically run through the F18 compiler we have also placed a load for it into block 200. (You should follow the same pattern when you begin your own projects. If your project spans many blocks it's a good practice to use the first block to load all the others and load only this first block from 200.) Blocks 200 and 842 in the arrayForth distribution contain these things to facilitate your walking through this exercise.

When picking a sequence for loading your blocks the safest one is to load each server node sometime before their respective clients. As a matter of definition a pair of nodes have a server/client relationship if the server trusts jumping or calling the shared port and the client feeds instructions to the same port. In practice this relationship does not change dynamically. By loading the servers first then names defined in them are available for making instruction words that the client can feed back to them.

Our sample is only one block and one node long. It begins with an identifying comment and then shows that it is F18 code for node 600 beginning from location zero. If this code were node-independent, you might want to leave out the **`600 node`** phrase and specify that from the block loading this one. The code is divided into three sections, each one in turn more aware of the others. We will describe the code in the order of its writing, as if wrapping the onion.

```
                                                  842 list
pwm demo for host node 600                        pwm demo 600 node 0 org

pol checks for ide inputs and calls down          pol 000 @b 2000 dw and if
when noticed.                                     ... 003 ... down b! @b push ex
rtn is the return point from a down call          rtn 006 ... io b! then 008 drop
and is used by upd as an re-entry point.
cyc begins the actual pwm code.                   cyc ie- 1FFFF and over . + -if
upd is the ide entry point for initial            ... 00C ... 20000 !b pol ;
start or output update.                           ... 00F then 10000 0 !b pol ;

                                                  upd 012 xex- drop push drop 100
                                                  ... 014 pop pop iex- rtn ; 016
```

The core function is the three lines beginning with the comment "cyc". This code expects two stack items: an increment value and an error accumulator. It implements the inner loop of the PWM algorithm by calculating and sending a new value to the output pin. The hex number 20000 sets the output to strong pull down which will turn on the LED. All other output values will not cause significant current flow. A hex value of 30000 would select strong pull up and is not useful for this application. The hex value 10000 sets weak pull down and turns the LED off. The commented 0 would select tristate output which also turns off the LED but permits the pad to float higher. By toggling which of these values is commented one can rapidly compare the consequences. We will discuss how this is done in the next section.

The algorithm used is essentially an adaptation of the classical Bresenham line interpolation algorithm (or at least one understanding of it). PWMs are good at adjusting average power by controlling the duty cycle of a current source or sink. Power sinks such as LEDs or motors are examples of good candidates for PWM control. Because the switch to the energy source is either full on or off they are more efficient than typical analog control methods. The following analogy should help us to visualize this use of the algorithm.

Think of a pixel as being the smallest unit of energy that we can control; full power times the shortest time period we can cycle the algorithm. We can map the set of all positive slopes onto the set of binary fractions between 0 and 1 by thinking of the slope as the ratio of power on to power off time. A vertical slope is full power and a flat slope is zero power. An infinitely thin line leaving the origin with rational ratio will pass between many pixels without striking them dead center. There is an error term that maintains the amount by which each ideal point is missed as the line goes by. The slope, as a binary fraction, is added to this error term. Each time there is an overflow a one is output. Each time there is no overflow a zero is output. The remainder less than one left in the error term is always carried forward to the next interval.

At 0.5 duty cycle there is a perfect square wave at maximum frequency. Above 0.5 the high pulses begin to concatenate, separated by low pulses of the minimum width. Below 0.5 the low pulses concatenate between lone high pulses. At 0.25 there is a pulse train at half the maximum frequency. At 0.125 the frequency is a quarter of maximum. Above 0.5 for complementary slopes (where complementary is defined as 1-x and x is 0.5 to a positive power), the signal frequency changes just as it does for those x below 0.5, except that the output signal is inverted. For slopes below 0.5 represented by more than one 1 bit in their binary fractional forms, the output is made up from all contributing frequencies interspersed. Because only a single overflow is possible in each cycle, each frequency is magically merged at its own unique phase. At slopes of either 1.0 or 0.0 the error term never changes and the output either saturates or stops.

In our implementation the hex number 20000 represents 1. The number 10000 is a half and so on. At the beginning of "cyc" the **`1ffff and`** removes any present overflow bits from the error term so that the next new one can be detected. The phrase **`over . + -if`** adds the slope in S to the error term in T and tests the overflow bit. In the case of overflow **`20000 !b`** maximizes the output current. For the non-overflow case **`10000 !b`** minimizes the current flow. If our algorithm never had to represent but a single slope then at this point each of the two output phrases would simply jump back to the **`cyc`** point. We want our code to entertain new inputs as well as to accept debugging illumination requests so instead we jump to the command monitoring function called **`pol`** above.

When the code is running in diagnostic mode it will be loaded from an IDE "wire" coming from node 700. Examination of the G144 quick reference poster shows that 600 and 700 are connected by their **`down`** ports. When an IDE command is issued across the wire the first instruction is a focusing call that limits the target's program counter to only a single port decode (in case the target had been executing a multiport fetch before). At the completion of an IDE command, other than the call command, a return instruction completes the command and returns the target to its original task. For an idle target node this will be a multiport execute. In our case we will be executing the PWM rather than a port fetch. The three lines of code starting with **`pol`** serve to poll for IDE commands and if one is detected then we turn control over to the port completely.

The first line of code fetches the IO port value, masks out the down write bit and if the result is zero (no write pending) it jumps to the **`drop`** which restores the stack to the values expected by "cyc". If a write is pending we need to give up control to the port but need to leave a return path to the PWM in case the intentions of the IDE are temporary. We assume there is a focusing call to **`down`** sitting in the port and this instruction needs to be removed by reading. We also must emulate executing this call but with a return address back to ourselves. The phrase **`@b push ex`** pulls the call from the port, pushes it to the return stack where **`ex`** (execute) performs a co-routine jump to the address pushed as part of the call. The new return address replaces the call address on the return stack. In this way when the IDE completes it will return us to the **`io b!`** phrase on the last of these three lines, restoring B.

The final requirement that must be met for this demonstration code is to help the IDE in setting up and changing the operating conditions. The IDE provides for pushing a literal item onto the target data stack but it does not support insertion of an item to replace S in a single operation. The two lines of code called **`upd`** provide this function. The hex literal 100 at the end of the first line is known to occupy location 13 in ram. We will code an IDE script word in the next block called **seed** which will store a value from the arrayForth stack at location 13 and then call **upd** to inject that value into the PWM as a new slope.

As soon as the code is written we add the phrase **842 load** into block 200 so that the next time we type **compile** this block will be included in the compilation. Once we do so we will observe that all our grey numbers which were initially 001 have all been modified by the F18 compile to reflect the program counter at that point in the program. If you mistyped any names or used a name before defining it the compiler will abort with question mark appended to the word you typed (such as **compile**) to perform the compilation. If you respond with **e** the editor will take you to the point in the block where the error was first detected. If all compiles well the next step is to generate some IDE script to help us install and test the code.

9.4 The IDE script

We believe that interactive development is not merely the responsibility of some esoteric, third-party, software development platform. We believe it is primarily a mindset. The choices and tools presented by the development platform must simply not conflict with the proper mindset. The mindset cannot be enforced by the tools. The following precepts can be considered to be part of the mindset.

- Make small changes.
- Save often.
- Test every change.
- Make sure all steps are repeatable (such as rebuilding everything before each test).
- Choose development paths that do not preclude testing for significant intervals.
- Shun tools which delay your feedback because they serve to distract you.

The reason colorForth keeps all source code memory-resident and compiles all code from pre-parsed tokens is to encourage recompiling often. The use of scripts encourages repeatability and in colorForth you cannot even make a definition without first committing it into a block. The arrayForth word **compile** supports these precepts by quickly rebuilding all F18 object code and also by reloading whatever test environment you have configured for the current stage in your development. To support the latter function, whenever you load testing tools they should begin with the phrase **0 fh orgn !** which directs compile to reload that test environment as part of its job.

In block 844 you will find the test script we have built for exercising the PWM demo. The template for this script was copied from the **host** block that you loaded earlier when we typed **host load** as part of assumption checking for node 600 and our wiring. We encourage you to read the shadow for the **host** block. In our present case we have deleted the sample definition and the **canon load** and added two script definitions on top the standard serial IDE functions. We could also, if we had wanted, defined here a custom wiring path to replace the standard one but the defaults will be adequate for such a simple program.

```
                                            844 list
configure ide for demo testing.             demo ide boot empty compile serial load

***no canonical opcodes***                  customize -canon 0 fh orgn !
use the 'remote' ones                       a-com sport ! !nam

seed loads pwm 'rate' and re-/runs cycle.   seed n 13 r! 12 call upd ;
run selects node 600 target, loads the pwm
into it and starts it with a default value. run talk 0 600 hook 0 64 600 boot
                                            upd ?ram panel 0 lit 18000 seed ;
```

The word **seed** uses IDE remote commands to place its argument on top of the template literal in node 600 **upd** and to restart the PWM at the **upd** entry point. The panel stack display is also updated so that, if you are viewing the panel, you will see your new argument there. It is helpful when outputting a sequence to see what you put out last. On the other hand outputting a new seed does not force panel display as this would be presumptuous and could easily be considered a distraction.

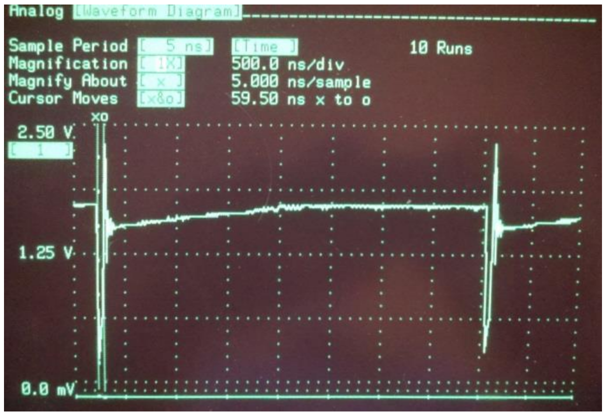
The word **`run`** forces a reset to the host chip and reloads node 600 via path 0. It displays the panel and starts the PWM with an initial seed value of 0.75 duty cycle. Note that you would not need to use this word after a **`compile`** if you had not made changes to the F18 code. If you had only added or changed some script function you would be good to go.

To compile this script code for the first time and get it hooked in you type **`844 load`** this time only. In the future simply use **`compile`** and you will also load these definitions.

So now that your PWM and test code is loaded, it is time to power up the scope, type **`run`** and use **`seed`** to observe the effects of different values upon the waveform and the LED. Note that although the waveform energy is a linear function of duty cycle, your eyes do not perceive the LED intensity as linear. In a dark room you can observe the smallest value of 1 as well as increments of 1 but as soon as it becomes significantly bright you can no longer resolve such small differences. Your eye has some kind of a logarithmic response and responds better to powers of two or less. Perhaps a Fibonacci ramp would be more pleasing.

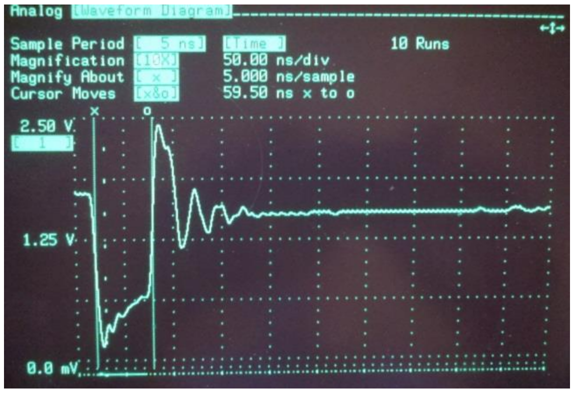
9.5 Output Observations

The output waveforms shown in this section demonstrate the behavior of the PWM. We will use the term *bit cell* when referring to the smallest unit of output and the word *frame* to refer to the period determined by the frequency contribution of the least significant one bit in the slope value. For consistency all the waveforms shown have a frame size of 64 bit cells. The size of our bit cell is determined by the both the algorithm and the particular speed of the node and chip under test. For our case the bit cell time is approximately 59.5ns and the frame time comes out to approximately 3.81us. The output values of 1/64 and 63/64 have been chosen because they are easy to sync a scope to, because they are complementary, and because they highlight the effect and shape of a single bit cell. The last value of 33/64 demonstrates how two frequency components merge.



```
Analog [Waveform Diagram]
Sample Period  [ 5 ns]  [Time]            10 Runs
Magnification  [ 1X ]   500.0 ns/div
Magnify About  [ x ]    5.000 ns/sample
Cursor Moves   [x&o]    59.50 ns x to o
```

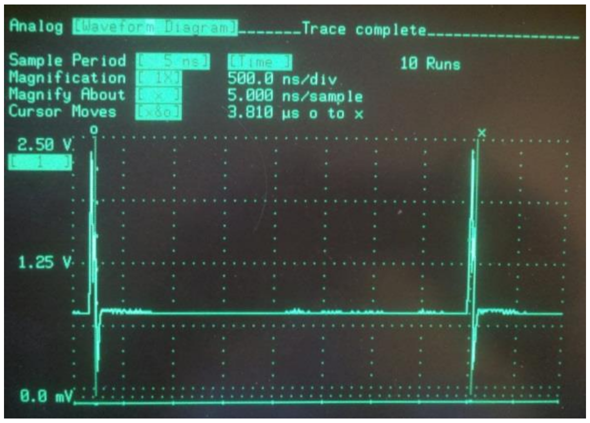
The first image shows a frame of mostly zero (higher voltage). You can see how the output rings and then floats up gradually.



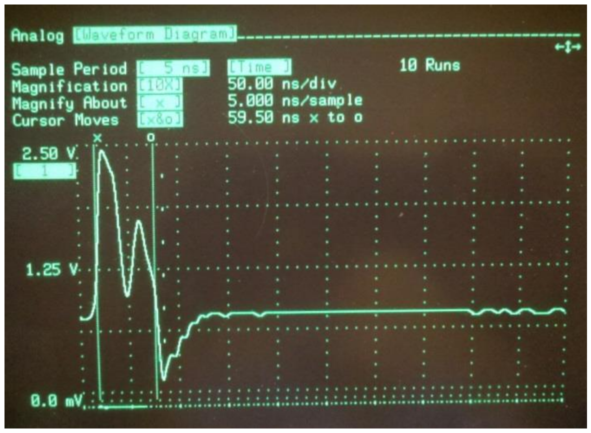
```
Analog [Waveform Diagram]
Sample Period  [ 5 ns]  [Time]            10 Runs
Magnification  [10X]    50.00 ns/div
Magnify About  [ x ]    5.000 ns/sample
Cursor Moves   [x&o]    59.50 ns x to o
```

The second image zooms in on the one bit at the perimeter of the frame. We see a strong negative going pulse that arcs back up to a stable voltage just before releasing.

The third image is of a frame of 63 one bit cells and a single zero bit cell. You can see how the frame begins with a strong low going pulse that quickly stabilizes to a firm value and ends with a short ring as it is released for one bit cell.



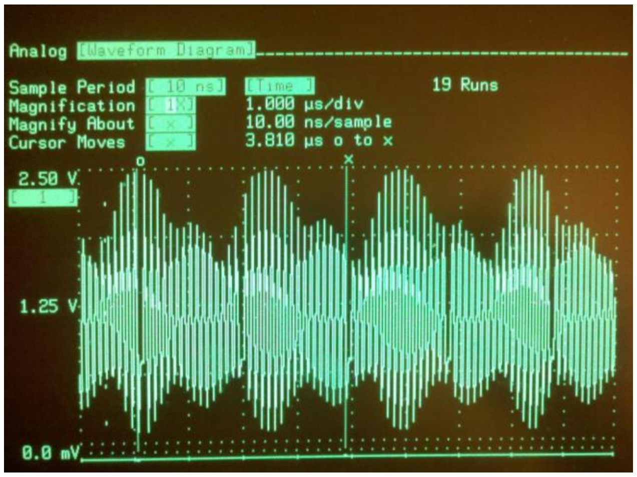
```
Analog [Waveform Diagram]_______Trace complete________________
Sample Period [ 5 ns]   [Time ]          10 Runs
Magnification [ 1X]     500.0 ns/div
Magnify About [ x ]     5.000 ns/sample
Cursor Moves  [x&o]     3.810 µs o to x
```



```
Analog [Waveform Diagram]_____________________________________
Sample Period [ 5 ns]   [Time ]          10 Runs
Magnification [10X]     50.00 ns/div
Magnify About [ x ]     5.000 ns/sample
Cursor Moves  [x&o]     59.50 ns x to o
```

The fourth image is a zoom into the frame transition of the previous capture and shows the single zero bit cell being replaced by the long string of ones.

The last image is of the value 33/64. It contains two complete frames. In each frame you may count 33 one bit cells and 31 zero bit cells. Observe that this packing is accomplished by joining two one bit cells in each of two locations. There is also a change of phase in the second half of each frame.



```
Analog [Waveform Diagram]_____________________________________
Sample Period [ 10 ns]  [Time ]          19 Runs
Magnification [ 1X]     1.000 µs/div
Magnify About [ x ]     10.00 ns/sample
Cursor Moves  [ x ]     3.810 µs o to x
```

9.6 Further Study

Please see the hybrid DAC function **`-dac`** described in the *G144A12 Chip Reference*. That algorithm, present in ROM of the Analog nodes, uses duty cycle variation to enhance the resolution of the 9-bit current sourcing DACs.

10. Appendix: Commissioning a New G144A12 Board

This section describes the simplest procedures for "bringing up" a new PCB with one or more G144A12 chips. If your hardware differs, use these as examples to develop your own procedures.

10.1 G144A12 Standalone, No Flash or SRAM

10.1.1 sF as Host

Arrange and configure an FTDI serial interface. Use EIDE interactively to explore and debug. Use serial boot streams to program.

10.1.2 pF/144 as Host

Not supported yet.

10.2 G144A12 Embedded, Flash Only

10.2.1 sF as Host

Take the steps as for Standalone above. External flash burning is not supported yet. Ensure board set for no-boot.

10.2.2 pF/144 as Host

Not supported yet.

10.3 G144A12 polyFORTH Capable

10.3.1 sF as Host

One method is presently supported:

1. Take the steps as for Standalone above and verify the chip is alive with EIDE. Ensure board set for no-boot.
2. Bring up polyFORTH (no Ether or clock) using serial boot stream, 1- or 2-chip.
3. SERIAL LOAD PLUG <space> to autobaud
4. HI (default UNITS table maps serial disk first). If you have recompiled nucleus on that serial disk or have changed clock selection in block 9 you may need to think about what you are doing.
5. Make flash boot stream using 2208 or 2211.
6. WRITE-FLASH if not present in the load block.
7. ?FLASH to verify that it was written.
8. Copy at least the first 9 blocks of serial disk to the flash disk area so there is a nucleus to boot.
9. Set board to flash boot and Reset board.
10. Repeat steps 3 and 4 above, and you are running pF/144 booted from flash.
11. Your board is now self-maintaining using either serial or flash disk as you wish.

10.3.2 pF/144 as Host

Not supported yet.

11. Appendix: Reference Material

11.1 F18A Code Library

GreenArrays has written, tested and used a considerable Library of generally useful F18A code; this varies from single utility nodes, such as the Perfect Wire, to multi-node clusters such as the Ethernet NIC. Some of the code is useful anywhere on the chip, while other code such as, for example, the SRAM cluster, depends on the I/O and geometric properties of the nodes for which it is designed. Every bit of this code is distributed as source. Where practical, such code is pre-assembled and is distributed with the system as object code in virtual bins (1600 to 2400) so that it may be used readily by tools such as the IDE or included in applications via boot descriptors. In other cases, such as the Ethernet Cluster or the polyFORTH Virtual Machine, enough nodes are involved that it makes more sense to assemble it for the target nodes when needed by an application being built.

This section documents the code, and its usage, for each such element in the Library.

11.1.1 Perfect Wire (any node)

It's often necessary to use a node as a simple wire to connect ports that are not physically adjacent. In the simplest case, when the use is permanent (until chip reset) and the data move in only one direction, there is a demonstrably perfect solution: A single instruction word program that, once fetched from memory, runs entirely in the Instruction Register I and is an infinite loop with minimal latency and minimal jitter (one word of every 262144 transferred takes an extra `unext` time.)

The instruction word may be written as `A[ begin @ !b unext unext ]]` . This single word is provided as location zero of Virtual Bin 1802 and may be placed in an application by the following descriptor macro:

`+wire` `(s d nn)` Places perfect wire in RAM location zero of the given node. Initializes **a** and **b** to the source and destination addresses **s** and **d**. Initializes the return stack with nine -1s.

11.1.2 64-word Delay Line (any node)

Check Mark 2 stuff in 236..8

A node may be programmed to serve as a 64-word delay line with throughput on the order of 5 memory cycles (~25 ns) per data word, and this performance remains the same no matter how many such nodes are placed in series (although latency from start to end when a word is pushed into a quiescent line does depend on the number of nodes).

Each data word is pushed into the first delay line node surrounded by two instruction words, and each data word between nodes and emitted at the far end of a delay line is also surrounded by the same two instruction words. No Virtual Bins are used because no code in RAM or ROM is involved at all. Delay line nodes are only committed to this use in the sense that they require special initialization of the data stack (with instructions) and registers **a** (as any valid RAM address) and **b** (as the destination port). So, at any time, a delay line node may be recommissioned via port execution (by default, we set it up with the source port address in **P**, but any multiport address may be used if desired.)

To place a delay line node in an application, use one of the following descriptor macros:

`+dly` `(s d nn)` Initializes **P** and **b** of the given node to the source and destination addresses **s** and **d**. Sets a to zero. Initializes the data stack with 5 copies of this instruction pair:

`A[ !b @p @ ]]` in T and `A[ !b !+ !b ]]` in S.

`+dx` `(nn n)` Places **n** delay line nodes starting with node **nn** and moving horizontally (to the East if **n** positive, to the West if **n** negative). Source and destination ports of all nodes including the first and last proceed in the direction indicated.

+dy `(nn n)` Places **n** delay line nodes starting with node **nn** and moving vertically (to the North if **n** positive, to the South if **n** negative). Source and destination ports of all nodes including the first and last proceed in the direction indicated.

The following code shows one way to insert a datum into the delay line:

```
: shove ( n)   <port> b!  @p !b A[ !b @p @ ]] ,
   !b @p !b ; A[ !b !+ !b ]] ,
```

At the output end, here's one way to receive a datum:

```
: suck ( - n)   @ drop  @  @ drop ;
```

As you can understand by studying the above, this is an excellent example of the remarkable capabilities of the F18A computer. The "program" for a delay line node is the pair of instructions pushed into its source port and is executed in that port; these two instructions surround the incoming datum. What this program does is to send, from its own data stack, exactly the same two instructions to the next node; however the datum between those two instructions is the oldest word in this node's RAM, which is replaced by the incoming data word.

11.1.3 Synchronous Port Bridge (300 and others)

An almost-transparent port bridge in bin 1904 uses nodes 300 on two chips to simulate a COM port between the UP ports of nodes 400 on each chip. The exception about transparency is that a node 400 cannot tell by inspecting its IO register whether the other chip's node 400 is actually reading its UP port. This bridge code is used by External and Internal IDE, by boot streams, and afterward by applications in 2-chip environments. For discussion, see 6.1.5 above.

11.1.4 Ganglia Mark 1 (any node)

11.1.5 Ganglia Mark 2 (any node)

By default we like to fill all nodes with this versatile messaging fabric, which is capable of exchanging up to 262k word payloads and replies between any two nodes on one or more chips that can be reached via any path through contiguous nodes running the ganglia and connected by COM ports or equivalent, such as the synchronous port bridge or any other bridging mechanism that supports flow control. These Ganglia are documented fully in App Note AN017, *Ganglia Mark 2*.

Object code is stored in Virtual Bins 1714, 1715, 1716 and 1717, one for each of the four physical orientations of F18A nodes. To fill both chips with ganglia, use the descriptor macro **`ganglia`** as *an early step* (before any memory load descriptors are processed) in a new frame definition.

11.1.6 SRAM Mark 1 Cluster (7, 8, 9, 107, more)

Code for providing services to, and coordination among, up to three Masters for a 1 MWord external SRAM. See AN003, *SRAM Control Cluster Mark 1*, for detailed documentation of usage and of internals.

11.1.7 Snorkel Mark 1 (any SRAM client node)

Code for a programmable DMA channel that may act as one of the Masters for the SRAM cluster. See AN010, The Snorkel Mark 1, for complete details.

11.1.8 IDE Components (708, 300 + any nodes)

The External IDE loads these components into the nodes it uses to reach into a chip and touch any nodes to which a path of available nodes exists.

11.1.8.1 IDE Asynchronous Interface (708)

Code for node 708 to provide services for external IDE. See AN019, *Interactive Development Environment*, for discussion of internals.

11.1.8.2 IDE Synch Interface (300)

Code for node 300 to provide services for external IDE. See AN019, *Interactive Development Environment*, for discussion of internals.

11.1.8.3 IDE Special Wire (any node)

Code for the nodes, if any, between the root node and the end node of an external IDE path. See AN019, *Interactive Development Environment*, for discussion of internals.

11.1.8.4 IDE End Node (any node)

Code for the final node, if any, between the root and target nodes of an external IDE path. See AN019, *Interactive Development Environment*, for discussion of internals.

11.1.9 Boot Stream Components

Boot stream generation is done entirely by distributed utilities. The components should be changed only if absolutely necessary and then only with great care.

11.1.9.1 SPI Flash Speed-up (705)

The ROM code for SPI flash booting uses, initially, a very slow timing parameter so that it can interoperate with slow devices. The first boot frame included in a boot stream for use from flash should begin by dropping a short program into node 705 and executing it to adjust this parameter for a higher speed. This program is provided in Virtual Bin 1608 for use by GreenArrays utilities.

11.1.9.2 Synch Boot Master

This code for node 300, in bin 1901, can transmit boot frames to a a GreenArrays synchronous boot node. It's used as a step in the set-up of a Port Bridge and for other purposes.

11.1.9.3 Port Bridge Set-up

This code for node 400, in bin 1907, is used ephemerally along with the Synch Boot Master in node 300 and a copy of the code for the Synchronous Port Bridge in node 500. Once all three of these nodes have been loaded it initializes node 300's I/O, resets the other chip via node 500 on pin 500.17, loads the port bridge code into node 300 of the other chip and starts it up, then loads the same code into our chip's node 300 and starts it as well. After this is completed nodes 400 and 500 may be re-loaded with application code, filled with Ganglia, or whatever else you desire.

11.1.9.4 SRAM Initialization (specific nodes)

11.1.10 polyFORTH Cluster (specific nodes)

This body of F18 code implements a virtual machine that runs a pseudo-instruction set from external SRAM. It includes a 2-node basic VM with four nodes of extension coprocessors, serial I/O through nodes 100 and 200 with associated wiring, and SPI flash mass storage. Dependencies include the SRAM Control Cluster Mk1, Snorkel Mk1 and Ganglia Mk2. Support is provided for the Ethernet NIC and for the various 10MHz frequency references it supports. For details about this body of code, please see DB006, *G144A12 polyFORTH Supplement*.

11.1.11 Ethernet NIC Cluster (specific nodes)

11.1.12 ATS Mark 1 (Serial IDE based)

11.1.13 ATS Mark 2 (On-chip based)

11.1.14 eForth (specific nodes)

11.1.15 Documented Examples

11.1.15.1 Practical Example used above

11.1.16 Additional Test Code

11.2 Bin Assignments for Rev 03a

"Bins" are receptacles for object code and are numbered like nodes in the chips. Bins 000 through 717 contain code that is by default destined for the nodes on the host chip. Bins 800 through 1517 map onto the target chip. Bins 1600 through 2317 are reserved for utility and tool code, and are managed by GreenArrays. Their purpose is to allow this code to be compiled and available for use at any time without interfering with application code. This table documents the assignments in effect as of the release identified above; yellow highlight indicates conversion to af-3 source, green shows that the binary has been validated.

1600	IDE async root node	1900	ats/ide analog	2200	Creeper test modules 3 (half)
1601	IDE sync root node	1901	ats/ide sync boot master	2201	
1602	IDE wire node	1902	ats/ide test frame	2202	
1603	IDE end node	1903	ats/ide test frame	2203	
1604	ide all nodes template	1904	Sync Bridge (with flow)	2204	
1605	Snorkel Mk1	1905	ats/ide uut bridge debug	2205	
1606	SST25WFxxx Flash node 705 pF	1906	ats/ide tester bridge debug	2206	
1607	New flash helper 706	1907	framer bridge builder	2207	
1608	spi speedup function	1908	Sync Bridge (no flow)	2208	
1609	18-bit flash r/w for 705	1909		2209	
1610	18-bit flash helper for 706	1910		2210	
1611	8-bit flash r/w	1911		2211	
1612	Flash ops for new devices	1912		2212	
1613	flash erase	1913		2213	
1614	SRAM Node 107 interface (Mk1)	1914		2214	
1615	- Node 007 SRAM Data Mk1	1915		2215	
1616	- Node 008 SRAM Ctls Mk1	1916		2216	
1617	- Node 009 SRAM Adr Mk1	1917		2217	
1700	polyFORTH Stack (106)	2000	Creeper test modules 1 (full)	2300	<reserved creeper tests 4>
1701	- Stack down (006)	2001		2301	
1702	- Stack up (206)	2002		2302	
1703	Bitsy (105)	2003		2303	
1704	- Bitsy down (005)	2004		2304	
1705	- Bitsy up (205)	2005		2305	
1706	Serial tx (100)	2006		2306	
1707	- Rx (200)	2007		2307	
1708	- Interface (104)	2008		2308	
1709	- Wire (102 and others)	2009		2309	
1710	Flash to sram for 705	2010		2310	
1711	- Wire for 605 etc	2011		2311	
1712	- <unused>	2012		2312	
1713	- Temp SRAM code for 108	2013		2313	
1714	Ganglion nodes eee (Mk2)	2014		2314	
1715	- nodes eoo	2015		2315	
1716	- nodes oee	2016		2316	
1717	- nodes ooo	2017		2317	
1800	<reserved for eForth/pF>	2100	Creeper test modules 2 (full)		
1801	Ethernet DMA	2101			
1802	Small tools (wire, etc.)	2102			
1803		2103			
1804		2104			
1805		2105			
1806		2106			
1807		2107			
1808		2108			
1809		2109			
1810		2110			
1811		2111			
1812	Flash to SRAM 8-bit	2112			
1813	Node 600 code (temporary)	2113			
1814	Flash to sram for 705	2114			
1815	- Wire for 605 etc	2115			
1816	- SRAM interface for 208	2116			
1817	- Temp SRAM code for 108	2117			

11.3 Examples to hang onto

A simple stream:

```
2121
( Descriptor test)
ASM[ # 715 NODE ERS # 0 org
   begin begin  !b unext unext   >BIN ]ASM
0 ARRAY MYP  207 ORGN 210 TO 710 TO 715 TO -1 ,
207 STREAM[   ' MYP COURSE
   FRAME[   715 +NODE  0 1 715 /PART
   -1. -1. -1. -1.  -1. -1. -1. -1.  -1.  9 /RSTACK
   x20001. x30002. x20003. x30004. x20005. x30006.
   x20007. x30008. x20009. x30010.  10 /STACK  IO /B  0 /P
   714 +NODE  111. 222. 333. 444. 4 /STACK
   ]FRAME  ]STREAM
!SNORK
```

```
.STREAM 512
       0  10175  4DAF 121D5    78 2FAB2  5A72  4DAF 12175   208,9
       8     72 2FAB2  5A72  4DAF 12115    6C 2FAB2  5A72   210
      10   4DAF 12145    66 2FAB2  5A72  4DAF 12115    60   310,410
      18  2FAB2  5A72  4DAF 12145    5A 2FAB2  5A72  4DAF   510,610
      20  12115    54 2FAB2  5A72  4DAF 121D5    4E 2FAB2   710
      28   5A72  4DAF 12175    48 2FAB2  5A72  4DAF 121D5   711,12
      30     42 2FAB2  5A72  4DAF 12175    3C 2FAB2  5A72   713
      38   4DAF 121D5    2E 2FAB2  5A72  4A12     0     0   714,15
      40  2E9B2  5872  9175  4BB2   15D  48B2 3FFFF  48B2   Regs
      48  3FFFF  48B2 3FFFF  48B2 3FFFF  48B2 3FFFF  48B2
      50  3FFFF  48B2 3FFFF  48B2 3FFFF  48B2 3FFFF  49B2
      58  20001  49B2 30002  49B2 20003  49B2 30004  49B2
      60  20005  49B2 30006  49B2 20007  49B2 30008  49B2
      68  20009  49B2 30010 101B5  49B2    6F  49B2    DE   714
      70   49B2   14D  49B2   1BC 101B5 101B5 101B5 101B5   713-11
      78 101B5 101A5 101A5 101A5 101A5 101A5 101A5 101A5   710-208
```

12. Appendix: Microsoft Windows® Platform

12.1 Windows arrayForth Requirements

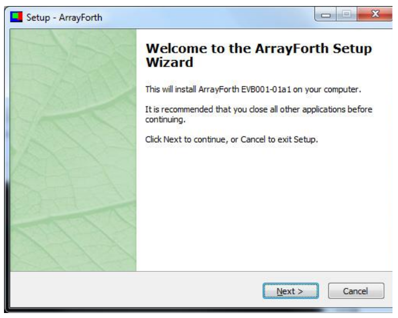
The host system requirements are as follow:

- Microsoft Windows with Win32 environment. We have tested this system on Windows 2000 Professional, Windows XP Professional, Windows Vista and Windows 7 and 8. See below for Windows 8 issues.
- Sufficient physical memory and swap file to run at least one instance of a program that is capable of requiring commitment of 768 megabytes of virtual memory. Actual requirements depend on how much memory you actually access, which should be considerably less than this.
- Display capable of at least 1024x768x24 bit graphics. The display may be resized after the program is started.
- Standard PC keyboard.
- At least one RS232 interface with COM port drivers if you wish to use the Interactive Development Environment to communicate with GreenArrays chips. For the EVB001 Evaluation Board, this is done with direct USB cables to the FTDI chips on the board, allowing much faster communications than are reliable with RS232 electrical interfaces.

12.2 Installation

Starting with release 01b, arrayForth is distributed as a conventional, executable Windows installer. Use it as follows:

1. Download and run the installer, or run it directly from your browser if the browser supports that operation. You will see a greeting diaglog that looks like this. Click the "next" button.



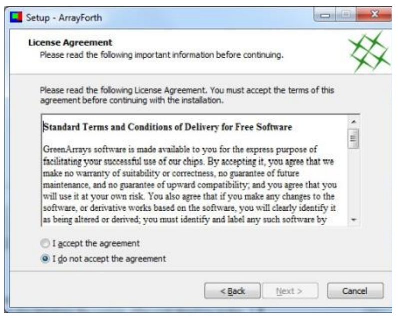
Welcome to the ArrayForth Setup Wizard

This will install ArrayForth EVB001-01a1 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

2. You will now be asked to read and accept our Standard Terms and Conditions for Delilvery of Free Software. If you click the "I accept the agreement" item, the "next" button will be enabled and you may proceed.



License Agreement
Please read the following important information before continuing.

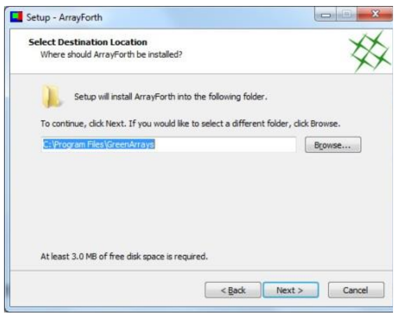
Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

Standard Terms and Conditions of Delivery for Free Software

GreenArrays software is made available to you for the express purpose of facilitating your successful use of our chips. By accepting it, you agree that we make no warranty of suitability or correctness, no guarantee of future maintenance, and no guarantee of upward compatibility; and you agree that you will use it at your own risk. You also agree that if you make any changes to the software, or derivative works based on the software, you will clearly identify it as being altered or derived; you must identify and label any such software by

I accept the agreement [✓ I do not accept the agreement]

3. The installer asks you to select a location at which arrayForth should be installed. By default this is a directory ("folder") that will be called c:\ Program Files\GreenArrays. If you don't want it to be there, please change the destination in this dialog; the shortcuts made by the installer will be configured for this directory and it will be simpler for you to place it where you like initially rather than to move it after installation. You may install multiple instances as you wish. Hit next when done.



Select Destination Location
Where should ArrayForth be installed?

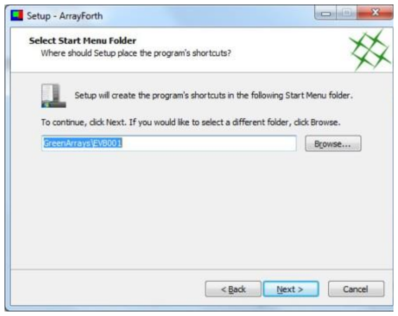
Setup will install ArrayForth into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files\GreenArrays

At least 3.0 MB of free disk space is required.

4. Next, the installer asks where it should place the program's shortcuts. By default there will be a new Start menu group called GreenArrays with a subgroup EVB001 in which shortcuts will be generated for this instance of arrayForth and for a file explorer view of the entire GreenArrays directory structure, facilitating your finding of your files. You will need to get at them later. Specify where you want them and hit next.



Setup - ArrayForth

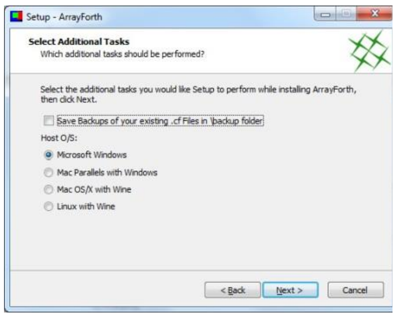
Select Start Menu Folder
Where should Setup place the program's shortcuts?

Setup will create the program's shortcuts in the following Start Menu folder.

To continue, click Next. If you would like to select a different folder, click Browse.

GreenArrays\EVB001

5. The next dialog is important. It begins with a check box which, if checked, will instruct the installer to save backups of your existing source files (.cf for arrayForth and .blk for polyFORTH) in a \backup folder. This will be a subdirectory inside the arrayForth directory structure. It is advisable that you check this box if you are installing an update on top of an existing installation; you may then, later on, use a copy of your previous working .cf file as the backup source file in order to merge your work with the updated system. The second area asks you to select one of a set of buttons to indicate which supported platform you are using. This will affect mainly what shortcuts the Installer will generate. For Windows and Mac Parallels platforms, the shortcuts will be in the Start menu and will assume a Windows compatible environment. For Wine environments, the shortcuts will be on the desktop and will use different scripting files to start the arrayForth program. Select the appropriate items and hit next.



Setup - ArrayForth

Select Additional Tasks
Which additional tasks should be performed?

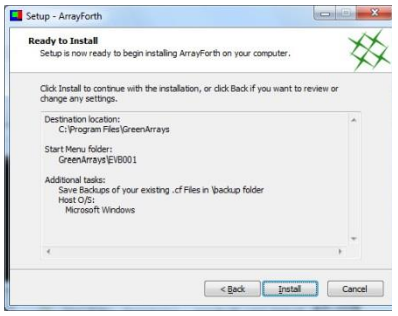
Select the additional tasks you would like Setup to perform while installing ArrayForth, then click Next.

☐ Save Backups of your existing .cf Files in \backup folder

Host O/S:

● Microsoft Windows
○ Mac Parallels with Windows
○ Mac OS/X with Wine
○ Linux with Wine

6. The next dialog is the familiar "Ready to Install" summary of what the Installer proposes to do based on your input in the preceding dialogs. If there are unpleasant surprises, use the back button and correct your input as needed. When you are satisfied with what you see here, hit Install. You will then see a progress meter dialog that should complete very quickly.



Setup - ArrayForth

Ready to Install
Setup is now ready to begin installing ArrayForth on your computer.

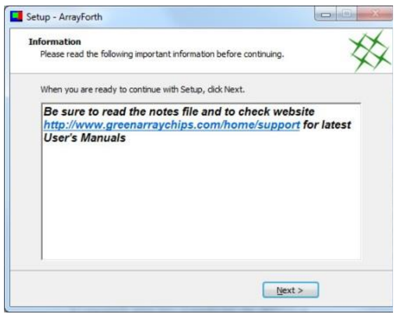
Click Install to continue with the installation, or click Back if you want to review or change any settings.

```
Destination location:
    C:\Program Files\GreenArrays

Start Menu folder:
    GreenArrays\EVB001

Additional tasks:
    Save Backups of your existing .cf Files in \backup folder
    Host O/S:
      Microsoft Windows
```

7. You will then see a dialog which purports to convey Important Information… and indeed it might. Should any given release require that you take any special actions or be aware of any changes in procedure or usage that might be at odds with the current editions of manuals like this one, this may be your only chance to learn of such before encountering it. Please take the time to read what is written here so you will not later regret having failed to do so!



Setup - ArrayForth

Information
Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

Be sure to read the notes file and to check website http://www.greenarraychips.com/home/support for latest User's Manuals

8. Finally you will see a dialog about Completing the Setup Wizard, which will by default offer you the opportunity to read our standard text file that documents changes in this version. Again this is highly recommended reading; if you must forego it at this time, the file is present in the arrayForth directory structure for future reading.

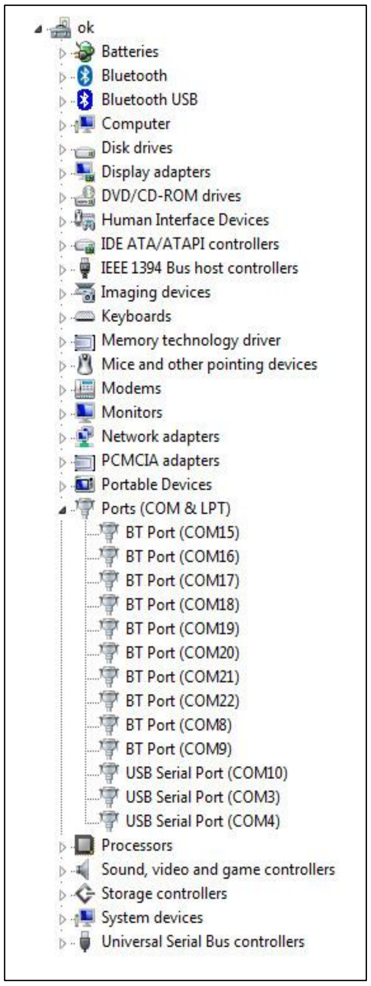
This concludes the initial phase of installing arrayForth. You should be able to run arrayForth using the installed shortcut and will only need to take further steps if you intend to communicate with real GA144 chips, such as those on the Evaluation Board.

12.2.1 Identifying and Configuring COM Ports

Until release 01c, COM port baud rates, framing, and so on were set using **`mode`** commands in the `okad.bat` script file on Windows systems, or **`stty`** commands in the `linuxwine.bat` script for Wine systems. Starting in 01c, arrayForth sets these parameters itself. The script files are still used, in case you encounter a system in which they turn out to be necessary, but the `mode` and `stty` commands are commented in the released files. If you find a situation in which arrayForth is unable to set the COM port up properly, please inform Customer Support right away.

If you plan to use COM port(s) to communicate with a GreenArrays chip or with the Eval Board, you will need to identify the COM port(s) in question. This procedure is designed for use with the Evaluation Board; to use arrayForth to interact with our chips on other boards, you will most likely be using some single USB to RS232 adaptor and so some of the steps may be omitted. *We recommend FTDI based communication devices for all communications with our chips, whether they be RS232 adaptors or embedded chips such as those used in our Eval Boards. This is because the FTDI drivers appear to be free of some common bugs in FIFO management that can create havoc with automated communications.*

1. The EVB001 board has three USB connectors identified as A, B and C reading left to right along the top left corner of the board. Port A is by default used for IDE on the Host chip, port B is used for a serial terminal on eForth or polyFORTH, and port C is used for IDE on the Target chip. The following steps should be taken for each of these three independent USB to serial adaptors. If instead you are configuring your installation for use with one or more actual USB to RS232 adaptors, perhaps for use with other boards than the EVB001, you will need to take these steps for each of those adaptors and should do them completely with one adaptor at a time. If you intend to be using several adaptors it is a good idea to leave the ones you have already configured plugged in while configuring the next, so that the system does not try assigning all of them the same COM port number!



- ok
  - Batteries
  - Bluetooth
  - Bluetooth USB
  - Computer
  - Disk drives
  - Display adapters
  - DVD/CD-ROM drives
  - Human Interface Devices
  - IDE ATA/ATAPI controllers
  - IEEE 1394 Bus host controllers
  - Imaging devices
  - Keyboards
  - Memory technology driver
  - Mice and other pointing devices
  - Modems
  - Monitors
  - Network adapters
  - PCMCIA adapters
  - Portable Devices
  - Ports (COM & LPT)
    - BT Port (COM15)
    - BT Port (COM16)
    - BT Port (COM17)
    - BT Port (COM18)
    - BT Port (COM19)
    - BT Port (COM20)
    - BT Port (COM21)
    - BT Port (COM22)
    - BT Port (COM8)
    - BT Port (COM9)
    - USB Serial Port (COM10)
    - USB Serial Port (COM3)
    - USB Serial Port (COM4)
  - Processors
  - Sound, video and game controllers
  - Storage controllers
  - System devices
  - Universal Serial Bus controllers

   a. Obtain Windows drivers for each USB serial adaptor you intend to use and appropriate to the version of Windows you are running. Although it is generally recommended to install drivers before first mating the device with the computer, Windows 7 seems to have relaxed this requirement. Nevertheless even on Windows 7 you may have to connect to Windows Update or to go to the manufacturer of your adaptor for drivers.

   b. Plug in the USB to serial adaptor (or Eval board port) and go through the procedure for installing a new Windows USB device. This can vary from trivial to a grueling hassle. Eventually your device will be "ready to use."

   c. Find the Windows Device Manager. The exact procedure varies between versions of Windows but in general you can get there by right clicking "My Computer" desktop icon (or corresponding clickable area of Start menu) and selecting "Properties". This should take you to a display on which you may select "Device Manager", or perhaps "Hardware" and then "Device Manager". You may also find it in the "Control Panel." Persist until you are looking at a tree view, like this one, listing various classes of devices. Open the tree view for "Ports" which include COM and LPT port names. Your USB port should be listed here. If there are more than one and you are uncertain which is your new port, unplug the device while watching this display. It should update and the device in question should have disappeared. Now plug it back in and record its COM port number and which of your interfaces (such as Eval Board port A/B/C) it represents.

Regarding this relationship: Some USB devices have internal unique serial numbers and some do not. The default behavior of later Windows systems is to make a permanent association between a given vendor/device/serial number and a given COM port, regardless of which USB connection it is plugged into. This is a *good thing*. For devices lacking a serial number, Windows tends to assign the COM port to that vendor/device on a particular USB socket. Thus the COM port number can be expected to change if you move the same USB to RS232 adaptor from a direct USB socket on the computer to one on, say, a docking station. This is the stuff of madness but you will need to cope with it if you use such interfaces.

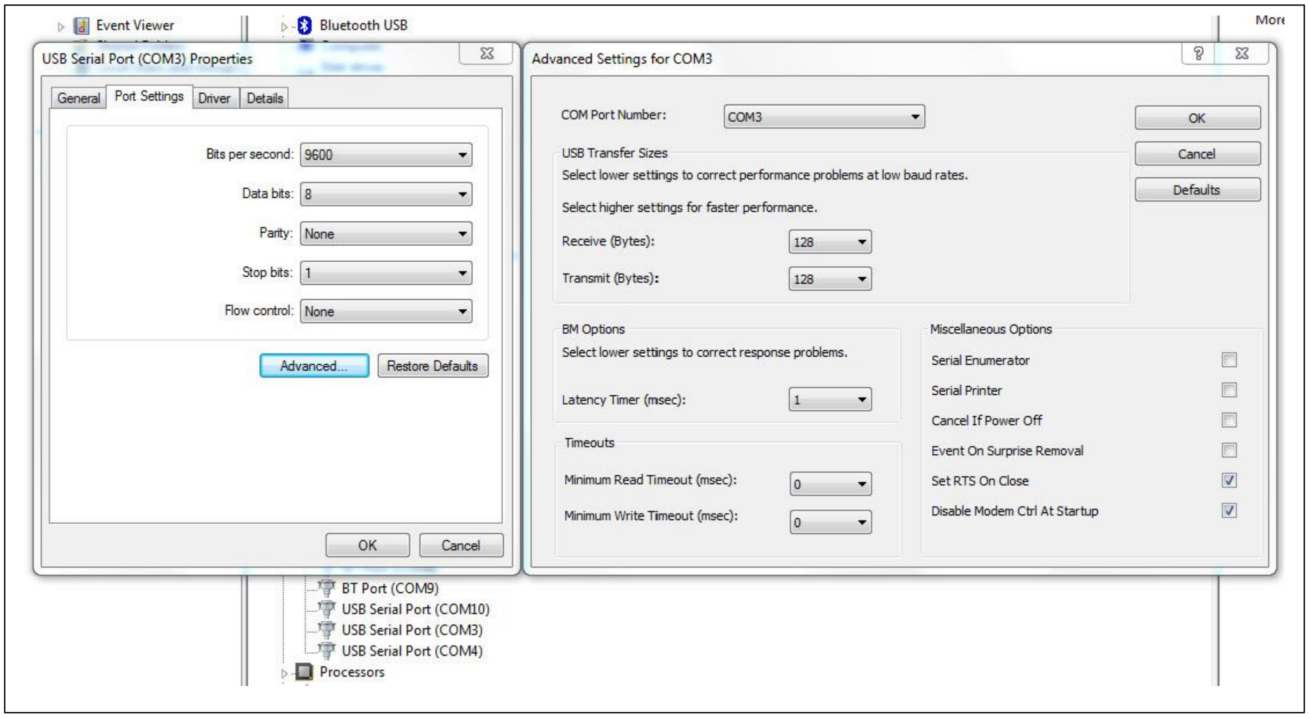
d. Having identified the new port, double click it in the Device Manager and you should get a USB Serial Port Properties dialog box. On the front page / first tab you will see the manufacturer ID.

e. If the manufacturer is FTDI, you have some things to do in order to optimize this port. On the Port Settings tab you should find an "Advanced" button and on pressing it should get a separate dialog box entitled something like "Advanced Settings for COMxx". If this is the case, please refer to the screen image below and do the following:

   i. In the "USB Transfer Sizes" section, the default settings for Receive and Transmit are 4096 bytes. Reducing these to 128 bytes seems to improve performance in our uses.

   ii. The next item called "Latency Timer (msec)" which defaults to 16. We get the best results in IDE and serial terminal performance with this value reduced to 1 ms.

   iii. Please see "Miscellaneous Options." We get the best results from setting these as shown. Instructions in app notes relative to the EVB001 will assume you have made these same settings.



USB Serial Port (COM3) Properties

Bits per second: 9600
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Advanced Settings for COM3

COM Port Number: COM3

USB Transfer Sizes
Select lower settings to correct performance problems at low baud rates.
Select higher settings for faster performance.
Receive (Bytes): 128
Transmit (Bytes): 128

BM Options
Select lower settings to correct response problems.
Latency Timer (msec): 1

Timeouts
Minimum Read Timeout (msec): 0
Minimum Write Timeout (msec): 0

Miscellaneous Options
Serial Enumerator ☐
Serial Printer ☐
Cancel If Power Off ☐
Event On Surprise Removal ☐
Set RTS On Close ☑
Disable Modem Ctrl At Startup ☑

2. Decide what baud rate(s) you will employ (but do not enter it in the above property sheets). What is feasible depends on the electrical characteristics of the PC serial interface in use. Our default baud rate, 921600, is correct for the EVB001 USB ports (all factory testing of each Eval Board is done at this speed.) If you are using an actual RS232 interface then the maximum usable rate will depend on the quality of its

chipset including its line transceivers. We have on rare occasion found RS232 interfaces that we could use at 460,800 baud, but more commonly the limit will be 115,200 or if you are lucky 230,400. Don't go any lower than 19,200 baud because our async nodes only have 18 bit time delay counters…

3. Start arrayForth using the shortcut made by the installer, or by double-clicking Okad.bat. Say `202 edit` and edit the port number for `a-com` (IDE for host or only chip) or `c-com` (target chip on Eval Board) as appropriate to reflect the ports(s) you have identified. Make sure the new port number is in yellow. Similarly, edit desired baud rates for these ports `a-bps` and `c-bps` if necessary; see above regarding appropriate baud rates to be used. Say `save` .
4. Shut down arrayForth by saying `bye` and restart the program.
5. Check the command window (see below) which should give feedback of success in configuring the serial port(s). You are now ready to begin IDE operations.

You may install multiple copies or versions of arrayForth on the same machine; just make a separate directory for each, and name your shortcuts to avoid confusion. This may be useful if you are working with multiple chips, or even multiple projects.

12.2.2 Windows 8 Installation

Our software is designed to run on a desktop computer, not on a telephone. It is assumed that you will be using the desktop interface to Windows 8, and that you will have installed one of the after-market programs such as StartIsBack that restores easy access to the start menu directories that are maintained in Windows 8 but are hidden by the asinine configuration in which that system is shipped. Contact our customer support hotline if you need advice on how to go about this.

12.3 Running arrayForth

Once you have taken care of the above chores, running arrayForth is simple:

1. Click the shortcut for the desired copy of the software.
2. When the arrayForth logo screen appears, the system is ready to use. The graphic window may be resized, minimized, dragged to other display screens, and so forth.
3. In addition to the arrayForth graphic window, you will also see a simple console window. Diagnostic messages may appear here. This console window is actually the primary window for arrayForth; if you wish to kill arrayForth by closing a window, closing the console window will accomplish this in one step. Note: It is normal to see a message `FFFFFFFE Error in system operation` immediately following the line that starts with `Reading file OkadWork.cf`. This simply means that the file was compressed and all 1440 blocks could not be read.
4. To exit arrayForth normally, simply say **bye** to the interpreter.

You may run multiple copies of arrayForth so long as you don't exceed the virtual memory capacity of the operating system you are using. This can be convenient when talking to more than one chip. For example, in one step of our factory testing we run `selftest` on both host and target chips simultaneously using two arrayForth sessions. Be careful to only edit your source base in one instance if they both use the same directory, otherwise `save` operations will overwrite one another.

13. Appendix: Apple Mac® Platform with Windows

13.1 Mac Requirements

arrayForth can be run on Apple computers that are built from Intel x86 processors, by using Parallels, a product of Elements5, hosting a Microsoft Windows® operating system (XP or later). The following procedures have been tested on a MacBook running OS X version 10.6, Parallels 5.09, and Windows XP Professional. We also believe they should work using Windows installed on a Mac system using VirtualBox; if you install our software on this configuration, please let us know how it went.

Note: If you are new to Windows, be aware that you will need to change some of the default settings in Windows before it is practical to do some of these things. For example, by default the Windows file explorer hides file extensions lilke .bat and .exe … and even hides whole files. If you are in this situation and have no idea how to do those things, contact our support hotline and we may be able to supply a "script" describing how to do it on the version of Windows you have installed.

13.2 Installation

Because Parallels (and VirtualBox) are Virtual Machine environments running actual Windows software, the procedures are almost identical with those on a Windows system. The differences we know of are as follow:

1. You should download the installer executable into some path actually rooted in `C:` in the windows environment's file system. Working directories are not correct when Windows programs are executed from places that are not, such as `\\.psf\Home\Documents` as one example.
2. When first plugging an adaptor or Eval Board port, go through the procedure for installing Windows drivers, associating the device with Virtual Machine rather than Mac OS when you are given that choice.

13.3 Running arrayForth

Again, running arrayForth is the same as on the Windows platform with these exceptions:

1. Be prepared for keyboard surprises and check the Parallels documentation to explain anomalies. For example, to use the `F1` key you may have to hold down the `Fn` key as a shift.

14. Appendix: unix Platform including Mac OS X

If you have an Intel x86-based unix system, you may be able to run arrayForth using the Wine subsystem. Wine is an open source implementation of the Windows API that runs on BSD Unix, Linux, Mac OS X and Solaris systems. Its home page is at WineHQ.org. The following has been tested in a freshly installed Ubuntu Linux version 11.04 with a freshly installed Wine obtained from the "Ubuntu software center" as "Microsoft Windows Compatibility Layer (meta package)."

14.1 Installation

Although Wine is not a Virtual Machine environment, most steps are the same as with the Windows platform. The differences are as follow:

1. Download the Windows Installer from our website. Save it in your Downloads directory.
2. Start Nautilus, the GUI file manager program, and navigate to your Downloads directory.
3. Right click on the icon for the setup program, select "Properties", select the "Permissions" tab on the resulting dialog, and check the box to "Allow executing file as program." (equivalent to `chmod u+x`)
4. Right click again and choose the menu item "Open with Wine Windows Program Loader." The installer should run as described above.

5. In the "Select Additional Tasks" window, check the box "Linux with Wine." When you finish the process, there should be two new icons on your desktop. The first is the arrayForth icon named "arrayForth EVB001 <vers>" that starts arrayForth, and the other is a folder icon named "GreenArrays" which opens the install directory using a GUI program called "Wine File".
6. What Wine calls drive C: in this window can be found in the Ubuntu file system as `/home/<yourlogin>/.wine/drive_c` .

14.1.1 Identifying and Configuring COM Ports

1. The procedure for identifying COM ports in unix is quite different than in Windows; contact Customer Support for assistance or suggestions. If you plug USB serial devices in one at a time they will by default be assigned consecutive unix device names of the form `/dev/ttyUSBn` where `n` starts at zero. The capability of Windows to identify a particular USB device by serial number does not seem to exist. Not only does the order in which currently inserted USB devices matter, but also the amount of time a device was unplugged before it is plugged back in seems to matter as well. You will probably be best served by plugging cables into your machine in a fixed sequence and following that sequence, without skipping cables, every time. We recommend that EVB001 users make a habit of plugging the three cables for USB ports A, B, and C into the computer in that same sequence every time; if port C is to be plugged in, make sure A and B have been plugged in or C will be assigned to one of the ports normally used by A or B. Follow this procedure for each of the three USB ports A, B, and C in order:
   a. Plug in the first USB to serial adaptor and run `dmesg` at the `bash` command line. You will see the name of the device listed near the end of the report, most likely `/dev/ttyUSB0`. If you do not see such a line, unplug the adaptor and plug it back in, waiting several seconds between steps; log writing seems to be out of phase some times.
   b. When this has been done, do the same for adaptors or cables B and C in order, one at a time.
   c. In the shell, navigate via `cd ~/.wine/dosdevices` . Typing ls there should show you `c:` and `z:` . If you have a real serial port its device name will be `/dev/ttyS0` . You will need to create a link for each of the USB ports you have identified; we recommend you use the following pattern for the commands that do this:

      ```
      ln -s /dev/ttyUSB0 com1
      ln -s /dev/ttyUSB1 com2
      ln -s /dev/ttyUSB2 com3
      ```

      You can use whatever COM port numbers you like, but 1, 2, 3 are sensible for A, B and C. The script files supplied assume you have done so. After you have done this, in future you may check which devices are currently plugged in using the command `ls -l ~/.wine/dosdevices` where color coding of device names will indicate their status.
2. Now, explore the install directory using the desktop icon. You will see a file called linuxwine.bat which is the script normally used by the installer in making the arrayForth icon. Open this file with a text editor. You will see the following:

   ```
   rem used ONLY for Linux/wine desktop shortcut.

   z:\\bin\\stty -F /dev/ttyUSB0 921600 -parenb cs8 -cstopb -crtscts raw -echo
   z:\\bin\\stty -F /dev/ttyUSB2 921600 -parenb cs8 -cstopb -crtscts raw -echo

   Okad2-41b-pd.exe
   rem howdy
   ```

   The purpose of these lines is to set the baud rate and other parameters for the IDE connection(s) to USB ports A and C on the eval board. If you have had to use some other ports than USB0 and USB2 for these purposes, edit this file accordingly You should now be ready to restart arrayForth and communicate with your chips.

The `linuxwine.bat` file may need editing for other configurations. In addition you may need to employ other scripting options we have worked out; contact customer support for advice if necessary.

14.2 Running arrayForth

Preferably you will use the desktop shortcuts. You may choose to create scripts to run arrayForth from the BASH or other shell:

1. Navigate to the directory containing the `.exe` file, `OkadWork.cf`, and `Okad.sh`.
2. Type `./Okad.sh` after editing it to do the stty commands for your ports.

15. Data Book Revision History

REVISION	DESCRIPTION
161231	Initial revision from arrayForth xxx base.
170219	Version passed out with preliminary aF-03a5+
180331	Documents aF-03a6. STREAMER works in both systems and reproduces flash boot made by colorForth. Chip can burn own boot flash; sF can inject serial boot streams.

GreenArrays®

Product Data Book DB013 *Revised 3/31/18*

IMPORTANT NOTICE

For current information on GreenArrays products and application solutions, see www.GreenArrayChips.com

Mailing Address: GreenArrays, Inc., 821 East 17th Street, Cheyenne, Wyoming 82001
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com
Copyright © 2010-2017, GreenArrays, Incorporated