Interactive Development Environment

in arrayForth for Generic GreenArrays Chips
As released in aF-42e3 (GA4) and 34k2 (GA144)

Current as of 5 April 2010

1. Introduction

This is an interactive development and debugging environment, part of arrayForth, that we have used with each GreenArrays chip. Its salient features are as follow:

- The node(s) under test are invaded as gently as is practicable given the limiting factor, namely RAM space in the root node for adjacent node support.
- The present code still respects usage rules for very old chips. These are:
  1. Driving of address bus by P is assured at the end of each instruction word executed in a port.
  2. Boot stream segments are still padded to be of even length, as was required by boot ROM in old chips, even though that is not required by our boot ROM any longer.

  As a result of the need to ensure P is enabled onto A for each port instruction fetch, many things that could be accomplished with a single word require two or three. Optimal code will be introduced at a later time.
- Three Umbilical interfaces are supported: On Windows, the USB interface which emulates SCSI disk may be used with a synchronous boot node, at roughly 1 Mbit/second rates, and/or RS232 async interfaces at up to 460800 baud to an async boot node may be used. On Native systems, RS232 async at up to 115200 baud is supported (more if faster UARTs are available.)
- A means to access F18 code by name is being considered but will not likely be implemented any time soon. Use the grey words to display absolute addresses in F18 code.

The following sections describe the usage and implementation of this version as it applies to all chips. Section 1.2 lists information specific to each chip.

1.1 Change History

- Version 1a (5 April 2010): Changed nomenclature for computers from g18 to F18.

1.2 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.

Presently the colorForth code starts in block 100 for USB, 114 for Windows serial, 96 for native serial. Path lists for USB Synchronous operations start in block 112; for async, in 120.

1.3 Chip specific details

Node numbers, default paths, and the supported boot methods vary from chip to chip. These, and other possible variations, are discussed in the following:

1.3.1 GA32-1.0 (March 2009) and 5v variant

Root nodes 19 (USB sync), 33 (async).

1.3.2 GA4-1.2

Root node 00 (async). No synchronous boot on this chip. One wire boot is not supported.

To support the reset protocol on the GA4, state management is somewhat different for the async IDE. First connect the serial interface and power up the GA4. Then load `serial .` After the program is loaded, the following steps may be used for initial boot or for, in effect, resetting the GA4:

1. Type **`-pwr`**
2. Cycle power on the GA4, making sure the power bus is pulled to ground so that the chip is actually reset.
3. Type **`talk`** to take the chip out of reset and load the root node program.

1.3.3 GA144-1.10

Root nodes 300 (USB Sync), 708 (async). One wire boot is not supported. GA144 paths are too long to list in one block, so the path list blocks extend into others.

2. Usage

It is, fortunately, far simpler to describe how to *use* this tool than it is to explain how it *works*.

1. To prepare for debugging, prepare the desired source code to be compiled for RAM under control of the main load block at 1300. Avoid clobbering the IDE code that is compiled as directed there since this code is required. For more information about RAM source placement, see the documentation for `softsim`. Note that each node's RAM code may use words by name in that node's ROM, so long as you correctly use the `node` and `bin` words. Remember what ***bins*** you use since these numbers will be used later on.
2. Examine and, if necessary, change the path lists as appropriate to make sure you can reach all the places you need to.
3. If your intentions include any host side exercisers, change block 100 or `serial` to load them before the canonical Forth words such as `@ ! dup drop +` and so on are defined for target operation.
4. If you are planning to use `serial` on Windows, be sure that you have set the COM port number correctly in both the `okad.bat` file and the variable `sport`, and that you have loaded OKAD using `okad.bat`.
5. Say **`compile`** to compile the F18 source for the first time.
6. Say **`100 load`** (for USB) or **`serial load`** (for async) to load the appropriate interactive environment.
7. Plug the USB or serially interfaced GreenArrays chip into your machine if you have not done so as yet and, if USB, wait for it to be recognized by Windows. Close any dialogs and any file explorer windows that may appear so that nothing in the Windows environment will have an open handle for the device. This last is also important for async operation; only one instance of colorForth may have an open handle to a COM port.
8. Say **`talk`** to find, open, and download root node talker code into the chip. For the USB interface, this cycles power which produces an automatic reset. For the async interface, power must be forced on (or turned on via USB) and talk will only reset the chip. ***On some very old test boards, you will need to press a reset button before this step when operating async***.
9. Display the indicator panel (Say **`panel`**) if you wish to view it (recommended!)
10. As the indicator panel will show, you should at this point have three defined paths, one to each of the nodes immediately adjacent to the root node.
11. 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 ensure that all other bootable nodes are completely idle, begin your activities by saying `2 <root> hook` and then `2 -hook` to tear that path down.*** As arrayForth is posted, path 2 is capable of reaching as many nodes as it is possible with a single path; on chips without test circuits, this is always the entire chip.
12. If you wish to change the code you are downloading into any node(s), change the F18 source and say **`compile`** which recompiles that source to binary and then re-enters the interactive environment, displaying the block you have most recently displayed. 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. *Recompilation does not communicate with the chip under test.*
13. When you are finished, you can say **`mute`** to close the active handle for the USB device, or simply exit arrayForth with **`bye`** which will do the same. The need to say `mute` is rare; if you need to do this on the async interface, use the word **`-sea`** instead. Usage on these words will be changed/refined in the future to facilitate leaving the chip running indefinitely and reestablishing contact from host system without disturbing the chip.

2.1 Vocabulary

Words come in three classes: Those which are used to wire, rip, and select paths; those used to operate on the target node; and those which are specifically remote, which may be employed by user code to exercise a node.

2.1.1 Path routing control

The simplest way to visualize and check the current routing situation is to simply display the **panel** .

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 (n) 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 n) 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 left in its default warm state (multiport execute); the target node *is not affected*.

2.1.2 Target operations

Each of these operations applies to the target node of the currently selected path. To help in orientation, it is good to display the panel unless there's something better for you to be looking at while working.

compile Recompiles both the F18 source and the arrayForth IDE source, allowing changes in the binary images that may be loaded into nodes, or aspects of the IDE such as automated test functions (see below). Connection to the chip is maintained, and any wiring context presently in place is preserved.

panel displays the indicator panel block.

upd retrieves all ten words of the data stack into the local array **stak** 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). panel displays the contents of stak in two rows. 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. This is isomorphic with the stack array and makes its behavior clearer.

?ram retrieves all of RAM from the selected node, updating the panel display.

?rom retrieves all of ROM from the selected node, updating the panel display.

lit (n) removes a number from the colorForth stack, pushing it onto the data stack of the target node and updating the stack display on the panel .

@ (a-n) reads RAM, ROM, register or port *a* in the target node, pushing the data read, *n* onto the colorForth stack. *Not to be confused with the F18 @/@a function.*

! (n a) writes a number from colorForth data stack into target memory (RAM, register, port) address *a* . *Not to be confused with the F18 !/!a function.*

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 bin) loads code into the target node, starting at address *a* for *n* words, from the current binary output area identified by *bin* . 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 left right place chip port addresses on the colorForth stack.

The remaining functions are stack and arithmetic operators that run in port execution on the target node. These affect only the target node's stack, memory, registers; they have no interaction with the colorForth host machine's stack. *Others may be added in the future.* All of those listed below update the `panel` stack.

`!b (n)` F18 store T into memory/register addressed by B.

`+*` F18 multiply step operating on S, T, and A.

`2* (n-n)` F18 left shift.

`2/ (n-n)` F18 arithmetic right shift.

`- (n-n)` F18 ones complement (invert) of T

`+ (n n – n)` F18 add

`and (n n – n)` F18 logical AND

`or (n n – n)` F18 logical *exclusive* OR

`drop (n)` F18 discard top of stack

`dup (n – n n)` F18 push a copy of T onto stack

`over(p q – p q p)` F18 push a copy of S onto stack

`a! (n)` F18 pop T into A

`a (-n)` F18 push A onto stack.

`b! (n)` F18 pop T into B

2.1.3 Words for user exercisers

Before the canonical names such as `@` and `dup` overload the host colorForth definitions, user code for exercisers may refer to any of the operators above in their remote specific spellings.

`path node hook -hook upd lit call` and **`boot`** use the same names since there are no conflicts.

The remaining words are spelled with a leading `r` as in **`r@`** and **`rover`**.

2.1.4 Current tester words

As of this release, the following functions are supplied in the "tester" block: This block may be changed during a session; `compile` will make those changes available. Some of these functions differ slightly between chips.

`one (n)` Runs a particular test on node *`n`* using path 2. One or more tests will appear at the end of the definition; the one that is not commented is the one that will be run.

`n6tst` Loads the most recently compiled binary for bin 6 into the node under test, executing it at the start address in `ent` and determining if the node *survived* the test by nondestructively changing location zero in RAM and verifying that the store operation actually worked.

`ramtst` Exercises RAM of the node under test with various patterns. All operations are done over the umbilical a word at a time, so at low umbilical bit rates this can take a while.

`all (n)` Runs the test in `one` against node *`n`* and, if it passes, against *`n-1`* and so on until node 0 has been tested. Automatically skips the root node. If a test fails, we abort and the node with problems will be visible by the wiring context for path 2 as shown in panel .

`watch` periodically displays the stack of the node under test via the current wiring context; strike any key to stop. Doing this on a node that is actively running a program requires insertion of a special polling function in its outer loop. Here is an example of the polling function in a node being debugged through its `up` port to permit debugging access while running:

```
: poll   @b 200 and if  up b!  @b push ;'  io b!  then drop ;
```

3. Implementation

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. This section reflects one particular implementation and may differ from the one you are looking at. Consider it a walk-through as you study the code.

3.1 Root Node

The root of our umbilical connection is typically a ***boot node***, since these nodes already know how to accept and execute memory loads, and to pass port streams to and from other nodes. *The root node may not be debugged;* to interact with the boot node you are currently using, it is necessary to establish an umbilical connection using some other boot node as root. The root node may have one active path to a target node for each of its three com ports (two if a corner node).

The root node supports two sorts of debugging operations: Direct, in which the target node shares a port with the root, and indirect, in which an end node and zero or more wire nodes lie between the root and the target. In the former case, the root functions as an end node, while in the latter it functions as a wire node. The requirement to combine host communications with end node functions makes root node RAM capacity the limiting factor in target test functions.

The following sections discuss the implementation of each root node:

3.1.1 USB to hybrid synchronous serial

The USB chip produces a hybrid interface, in that it controls the clock for both transmit and receive. As a result, it is not practical for it to wait on completion of variable time activities in the chip without polling; hence, most operations are open loop and things like executing code in the target node produce no direct, end to end, feedback regarding completion. Otherwise, the general pattern of function and code organization is analogous with other interfaces. ***This interface is only supported by the Windows version of colorForth, and it may not be available on some boards. Also, some boards will not power up until you have successfully established communication with the USB chip.***

The colorForth code starts in block 100 with machine code for byte oriented operations, structure definition, and such. Block 102 contains the code to search for a compatible device and open a handle for the raw device, checking `//./D:` through `//./W:` in alphabetical order. The first compatible device encountered is the one that will be used; *no effort is made to deal with multiple devices on a single computer.*

Block 104 supports SCSI passthrough operations. Block 106 provides the mechanism to lock and unlock the device, and to perform vendor unique SCSI commands that cycle power and reset the chip, transmit a stream of words to the chip, transmit and tristate the data line, and another that receives a stream of words. The final block of truly USB specific code, block 108, transforms 18 bit words to and from the bitstream format required by the USB chip in the SCSI data buffers; thus, there are five blocks of colorForth that exist almost entirely to support the this USB chip.

Block 110 composes sync node boot frames, and is aware of its USB synchronous interface. **`/frame`** **`+frame`** and **`!frame`** begin a standard boot frame, append words to it, and close/transmit it to the root node respectively. **`+ram`** appends binary code for a given address and length from the F18 output for a given node. **`talk`** finds and opens the device, loads the support code into the root node, and leaves us ready to begin interactive operations. **`exch`** performs a send/receive transaction. Constants are defined for definition entry points in root node RAM.

The code for the root node is identified in block 1300 of the arrayForth base. In order for this node to function as either an end node or a wire node, there are some things ... particularly those involving feedback to the host ... which are not practical or efficient to do solely as streamed operations. These things include **`word`** which transmits 18 bits in a USB chip specific way; **`+out`** which waits for clock low to signal that we may begin sending words; and **`-out`** which waits for clock high to signal that the USB chip has data for us. Higher level functions generate or pump input to the host: **`ok`** simply sends a word to acknowledge a command; **`fet`** pumps one word from the target to the host; **`stat`** pumps ten words from target to host. Stack dump for an adjacent node requires an extensive routine **`bstk`**.

Block 112 declares default route lists rooted at the sync boot node. Each is a sequence of nodes starting with one of the three com ports on the root. The lists end with -1. The default lists vary between chips, but 2 is used by default for automated tests and by default can reach all nodes. These can be redefined and/or customized as needed for any particular situation.

3.1.1.1 Generic code for all interfaces and root nodes

The remainder of USB support is common for all umbilicals:

Block 122 implements the debugging functions when the target is adjacent to the root. In addition there are some functions that are aware of chip geometry: **`wall`** takes two node numbers, which must be adjacent, returning a code { 0 1 2 3 } indicating the port connecting them { r d l u }. **`port`** is a resident function that takes this code, returning the address of that port.

Blocks 124 and 126 generate multinode stream loaders to create and destroy paths from the root node, through one of its three com ports, through zero or more wire nodes, to an end node which knows which of its ports leads to the target for that path. In an active path, each of the involved nodes (wire or end) is running a dedicated program, listening for a request on the port leading from the root node, prepared to pass that request along if a wire node, or to transform it into port execution sequences for the target if an end node, and to relay responses if any back to the host.. Wire and end nodes are loaded with the programs that can be found from block 1300. When a path is destroyed, all nodes involved (except the target) are returned to their natural default multiport executes from **`avail`** which, given a node number, returns the natural address for that node to multiport execute when idle and only listening for stimuli from other nodes.. Thus, to force every node of a typical chip to multiport execution rather than leaving any chance that they are in pause loops or running boot code, simply create and destroy a path as though the root node was the target, using path 2 which can normally access all of them.

Block 128 defines needed abstractions and the primary words **`path node -hook`** and **`hook`** which constitute the human interface for wiring up paths and selecting the one that is active; see usage description later.

Block 130 defines **`r@ r! rins call upd lit`** and **`boot`** which work on the current path to a node that may be adjacent or remote. Block 132 defines single instruction words suitable for port execution on the target, with remote specific names.

Block 134 uses blue words to display a live data view, showing the most recently retrieved stack and the state of all defined paths along with the most recent memory dump. It may be displayed by typing **`panel`** .

Block 136 defines some automated test functions, subject to change, and user extensible. At this point in the load sequence, generic functions such as `@` and `+` still refer to the host system's colorForth vocabulary.

Block 138, loaded last (and after any user exercisers) redefines canonical Forth words to apply to the currently selected target node.

In the initial GA144 releases, block 140 is an extension of path 2 definition, and 142 adds exerciser for loading and activating long period (about 5.5 minutes) CPU activities in up to 143 nodes.

3.1.1.2 USB to Synchronous Node Operations

Operationally, the bulk of the work of the root node is done by simply interpreting and acting on standard boot frames from the host. For most functions on either adjacent or nonadjacent nodes, a suitable stream from the host is simply passed along through the appropriate port, whether it is a “message” to be relayed to an end node, or a port execution stream for an adjacent node. After passing the stream along, the jump address in the boot frame will generally go to one of the functions in the root node to produce the appropriate feedback. The only major exception to this rule is the stack dump operation, which requires considerable code to help an adjacent node accomplish this nondestructively. In fact, with that function in the root node, very little room remains for additional operations. This will be ameliorated when restrictions on port execution are relaxed.

When the root node is simply passing things to a wire node, the message traffic and code involvement is as follows (to make the old boot frame rules happy, each of these sequences is preceded by a focusing `call (12xxx)` and, for even lengths, a `jump (10000)` to port. These are not strictly necessary and could be omitted; wait for adjustment till the old usage rule fixes are not needed so nops can be used freely):

```
fet   12000   2 word stream
        adr
        <-n   fet in boot node passes one word.

sto   12005   3 word stream
        adr
          n   ok in boot node sends one word 31416.

inst  1200A   2 word stream
      instr   ok in boot node sends one word 31416.

push  1200D   2 word stream
          n   ok in boot node sends one word 31416.

stk   12010   1 word stream
   <-10 wds   stat in boot node passes ten words.
```

When the root node is acting as an end node, the message traffic and code involvement differ. The streams below include the focusing call and optionally jump inserted by `exch` but unlike the above, these are *necessary*:

```
fet   12ppp   Focusing call
      2C9B7   . . . @p+
        adr
      2BE35   a! @ !p+ ;
        <-n   fet in boot node passes one word.

sto   12ppp   Focusing call
      10ppp   Padding jump
       4AB7   @p+ a! . @p+
        adr
          n
       B555   ! ; ; ;  ok in boot node acknowledges.

inst  12ppp   Focusing call
      instr   arbitrary instruction  ok in boot node acknowledges.

push  12ppp   Focusing call
      10ppp   Padding jump
       5555   @p+ ; ; ;  ok in boot node acknowledges.
          n

stk   12ppp   Focusing call            12ppp
       CDBE   !p+ dup push !p+          CDBE   !p+ dup push !p+

              Control passes to bstk
              in boot node to complete
              the operation:

              +out call
              @ dup push .               <-T
        <-S   @ word                     <-S
        <-T   word
              @p+ ! @p+ .
                @p+ . push .            49BA   @p+ . push .
                7
              dup push ! .                 7   7
```

```
              @p+ ! . .
                !p+ . . unext          C9B4   !p+ . . unext
      <-stk   begin @ word            <-stk   8 words
              next
              @p+ ! pop
                pop @p+ ;             27D55   pop @p+ ;
              ! -out ;                    T   Restore last
```

3.1.2 RS232 asynchronous serial

The async boot node is supported for both Native and Windows versions of colorForth. The appropriate code on either version is loaded via `async load`. ***This may require installation of a daughter board for RS232 to 1.8v logic interface. It also may require soldering or installation of a switch so that the board may be powered up without use of the USB chip if present. It may further require soldering so that the serial interface can conrol the reset line.***

The async interface may, if you wish, be used along with the USB interface, if both are installed. In this case the computer, or the windows colorForth instance, which is controlling the USB interface also has control over board power. In a dual configuration such as this, six connection paths may be open concurrently.

On the windows version, it is necessary to start `okad2.exe` via a batch file called **`okad.bat`**. This file sets serial interface parameters such as baud rate, character size, parity setting, handshake and so on. Baud rates higher than 115200 do not have standard Windows setting nomenclature. Before you use a serial port, including USB adaptors, you will have to ascertain the COM port number Windows has assigned to that port. This number must be applied inside the `okad.bat` file, and must also be stored into the variable called **`sport`** in block `serial+1` . To change this in a lasting fashion, load `async` then store the correct port number into `sport` and say `save`. To make it effective immediately, say `compile`. The port number in this variable is used at compile time to form the path name used in opening a handle for the port.

Data are transmitted to the root node as three 8 bit characters per 18 bit word, in the format required by the boot ROM. In this format the 18 bit word is XORed with 3ffff then shifted left 6 bits. Hex 12 is inserted into the low order six bits of the resulting 24 bit number. The three octets of this number are sent in ascending order starting with its least significant octet. Data received from the root node are also three 8 bit characters per 18 bit word, but without shifting or inversion. The first octet received is the low order 8 bits of a word; the second, the next higher 8 bits; the third has the two highest bits of the 18 bit word and the six high order bits of that octet are to be ignored.

The async interface does not control power as the USB interface does. It may, however, be wired to assert chip reset, and the async IDE assumes it is talking to a board that supports RTS and that this signal is tied to chip reset with inverted polarity such that when RTS is asserted (high), the reset line is grounded, and when RTS is deasserted (low), the reset line is driven to Vdd.

Receive operations in the native system may be aborted by striking any key. In Windows, the receives are blocking so there is no practical way to punch out if the chip is not replying. This can be ameliorated by specifying a receive timeout in the `okad.bat` file, ***if and when we work that out.***.

The colorForth code is much shorter than that required for USB. For Windows the `async` load block is at 114, followed by 116 which is about the serial port and 118 which is about the root node. For Native colorForth the `async` load block is at 96, followed by 98 which is about the root node.

The code for the root node is compiled in the load block at 1300. In order for the root node to function as either an end node or a wire node, there are some things ... particularly those involving feedback to the host ... which are not practical or efficient to do solely as streamed operations. These things include **`word`** which transmits 18 bits back to the colorForth host, and **`-out`** which goes back for another boot frame but which may be edited to a 3 port read `rdl-` for debugging of node 33 code from another node. Higher level functions generate or pump input to the host: **`ok`** simply sends a word to acknowledge a command; **`fet`** pumps one word from the target to the host; **`stat`** pumps ten words from target to host. Stack dump for an adjacent node requires an extensive routine **`bstk`**.

Block 120 declares default route lists rooted at the async boot node. Each is a sequence of nodes starting with one of the three com ports on the root node. The lists end with -1. Like the USB interface, the default list for path 2 can reach all nodes so that it may be used for automated testing. These can be redefined and/or customized as needed for any particular situation.

3.1.2.1 Generic code for all interfaces and root nodes

Async umbilicals use the same generic code starting at 122 as do all the others.

3.1.2.2 Asynchronous Node Operations

Operationally, the bulk of the work of the root node is done by simply interpreting and acting on standard boot frames from the host. For most functions on either adjacent or nonadjacent nodes, a suitable stream from the host is simply passed along through the appropriate port, whether it is a "message" to be relayed to an end node, or a port execution stream for an adjacent node. After passing the stream along, the jump address in the boot frame will generally go to one of the functions in the root node to produce the appropriate feedback. The only major exception to this rule is the stack dump operation, which requires considerable code to help an adjacent node accomplish this nondestructively. In fact, with that function in root node RAM, very little room remains for additional operations. This will be ameliorated when restrictions on port execution are relaxed.

When the root node is simply passing things to a wire node, the message traffic and code involvement is as follows (to make the old boot frame rules happy, each of these sequences is preceded by a focusing `call (12xxx)` and, for even lengths, a `jump (10000)` to port. These are not strictly necessary and could be omitted; wait for adjustment till the old usage rule fixes are not needed so nops can be used freely):

```
fet   12000   2 word stream
        adr
        <-n   fet in boot node passes one word.

sto   12005   3 word stream
        adr
          n   ok in boot node sends one word 31416.

inst  1200A   2 word stream
      instr   ok in boot node sends one word 31416.

push  1200D   2 word stream
          n   ok in boot node sends one word 31416.

stk   12010   1 word stream
   <-10 wds   stat in boot node passes ten words.
```

When the root node is acting as an end node, the message traffic and code involvement differ. The streams below include the focusing call and optionally jump inserted by `exch` but unlike the above, these are *necessary*:

```
fet   12ppp   Focusing call
      2C9B7   . . . @p+
        adr
      2BE35   a! @ !p+ ;
        <-n   fet in boot node passes one word.

sto   12ppp   Focusing call
      10ppp   Padding jump
       4AB7   @p+ a! . @p+
        adr
          n
       B555   ! ; ; ;  ok in boot node acknowledges.

inst  12ppp   Focusing call
      instr   arbitrary instruction  ok in boot node acknowledges.

push  12ppp   Focusing call
      10ppp   Padding jump
```

```
       5555   @p+ ; ; ;  ok in boot node acknowledges.
          n

stk   12ppp   Focusing call            12ppp
       CDBE   !p+ dup push !p+          CDBE   !p+ dup push !p+

              Control passes to bstk
              in boot node to complete
              the operation:

              +out call
              @ dup push .                <-T
        <-S   @ word                      <-S
        <-T   word
              @p+ ! @p+ .
                @p+ . push .             49BA   @p+ . push .
                7
              dup push ! .                  7   7
              @p+ ! . .
                !p+ . . unext            C9B4   !p+ . . unext
      <-stk   begin @ word              <-stk   8 words
              next
              @p+ ! pop
                pop @p+ ;               27D55   pop @p+ ;
              ! -out ;                      T   Restore last
```

3.2 Wire Node

The code for wire nodes may be found by checking load block 1300. Wire nodes are initialized with A and P pointing to the port from which messages are received (toward the root node), and B pointing to the port to which they are sent (toward the target node). The message sent to a wire node is encoded as a stream consisting of a call to a fixed RAM location for each function. This RAM location is the same in both wire and end nodes, so that the code in a wire node does not need to know which sort of node follows. If the function requires any arguments, they follow the call. If there are any results, they are passed back after any arguments have been delivered. When the transaction is complete, the node returns to the host side port, waiting for the next message (or for the path destruction stream). Message traffic and code involvement in the wire node are thus:

```
Func   Msgs  Code                      To/from next node
fet   12000
        adr  00 @p+ !b @ .
             01   fet call             12000
             02 !b @b ! ;                 adr
        <-n                               <-n

sto   12005
        adr  05 @p+ !b @ .
             06   sto call             12005
          n  07 !b @ !b ;                 adr
                                            n

inst  1200A
      instr  0A @p+ !b @ .
             0B   inst call           1200A
             0C !b ;                  instr
```

```
push  1200D
          n  0D @p+ !b @ .
             0E   push call           1200D
             0F !b ;                     n

stk   12010
             10 @p+ !b. .
             11   stk call           12010
             12 @p+ push
             13   9
   <- 10 wds 14 @b ! unext ;       <- 10 wds (S, T, rest of stack)
```

3.3 End Node

The code for wire nodes may be found by checking load block 1300. End nodes are initialized with A and P pointing to the port from which messages are received (toward the root node), and B pointing to the port shared with the target node. The end node receives the same message as is passed by wire nodes, but the code at the destination of the message's call is quite different. The end node must interact with the target node by sending it an interactive port stream, and both arguments sent to the target and results received from it will be, by necessity, intermixed within the port stream sequence of instructions. For this reason the end node program is somewhat larger than is that of the wire nodes. The end node program needs to know the port address across which we talk with the target, for focusing purposes. This address is written as the last word of the program, to be made available as a literal to the `focus` routine. The message traffic and code involvement in the end node are as follow:

```
Func   Msgs  Code                        To/from Target
fet   12000
             00 @p+ focus                12ppp   port call
             01   @p+ a! @ !p+            4A0E   @p+ a! @ !p+
        adr  02 @ !b @b @p+               adr
                                          <-n
             03   ;
        <-n  04 !b ! ;                      0   ; ; ; ;

sto   12005
             05 @p+ focus                12ppp   port call
             06   @p+ a! . @p+            4AB7   @p+ a! . @p+
        adr  07 @ !b @ .                  adr
          n
             08 !b @p+ !b ;                 n
             09   ! ;                     B555   ! ; ; ;
inst  1200A
      instr  0A @ focus                  12ppp   port call
                                         instr   arb instruction
             0B <fill>
             0C <fill>

push  1200D  0D @p+ focus                12ppp   port call
             0E   @p+ ;                   5555   @p+ ; ; ;
          n  0F @ !b ;                       n

stk   12010  10 @p+ focus                12ppp   port call
             11   !p+ dup push !p+        CDBE   !p+ dup push !p+
             12 @b @b ! dup                <-T
        <-S                                <-S
        <-T  13 ! @p+ !b @p+               ...   Next fast, no fix
             14   @p+ . push .            49BA   @p+ . push .
             15   7
```

```
             16 dup push !b .              7   7
             17 @p+ !b . .
             18   !p+ . . unext          C9B4   !p+ . . unext
     <-stk   19 @b ! unext .           <-stk   8 words
             1A @p+ !b !b ;
             1B   pop @p+ ;            27D55   pop @p+ ;
                                           T   Restore last

             1C <fill>
             1D <fill>

focus        1E @p+ !b !b ;             12ppp   Generate focus
             1F   12ppp               <inst>   on stack on entry
```

3.4 Target Node

Our methods tread very lightly on the target node. All operations are done through port execution on the target, so no memory is required or disturbed. It is assumed that the target node is in a resting state which expects instructions from at least the port through which we are talking to it. Thus, we assume the return stack, and R, are volatile. Normal operations do not change A, B, P, carry latch, IO, T, S, or the data stack, except when that is the essence of the operation such as `+`. Memory fetch and store do destroy the bottom word of the stack, and as presently coded alter A. The remote `call` function is a subroutine call to the port, for focus, and a jump to the routine, to simulate a call from the node's resting state. If the routine is able to return, that will leave P unchanged by the operation. If the routine clobbers the whole return stack, it is the routine's responsibility to re establish appropriate node resting state if further debugging is to be allowed.

3.5 Summary of Code Streams

Wire paths are created and destroyed using stream loader techniques. Streams are composed of a subset of the module types available for composition of boot streams, using what we need and omitting what we do not. Wherever practical these modules are the same as those generated for the same purpose by other tools.

The code streams used are all the same from the perspectives of wire and end nodes, regardless of the kind of root node being used. This is true of the active message traffic, the creation, and the destruction of umbilical paths. The active message traffic is as shown above, encapsulated in a single boot frame. The creation and destruction are made up of stock modules designed to be as close as practicable to those used in the standard Stream Loaders. One exception is that we do not care about side effects within our wire and end nodes, such as loss of return stack and so on. In some cases this allows us to use simpler usage rule work arounds. The stock modules are as follow:

```
Boot Frame Header:

  10xxx   Boot node jumps to xxx after processing stream
   dest   RAM or port address at which to store stream
      n   2*(n+1) words must follow

First Node Focus Prefix:

  12ppp   Focusing call to the port stream is being sent over
  10ppp   Jump to port, for benefit of pausing nodes.  This is
          deleted if necessary to make the remainder of stream
          come out to an even number of words.  That is never
          possible on path construction streams (all node code
          we lay down is even length, code wrappers come out even
          too).  Although this is tidy, it is not necessary in order
          to traverse nodes that may be in pause loops since we will
          never return to those loops in a node that's part of an
```

```
          umbilical path.

Port Pump module, 8 words.  Sent into a node to make that node
          pass subsequent stream on to another:

   4B17   @p+ b! @p+ @p+    Set outbound b, pick up focus
    ppp      outport
  10ppp      jump           Jump to outport for pausers.
  12ppp      call           Focusing call to outport (1st instruct)
   9BB7   !b !b . @p+       Send focus and jump, get count.
     nw      count          nw+1 words follow this module.
  24861   dup push if then  Count for loop (s40 fixup)
   5BB4   @p+ !b . unext    Pump following words.

Load Pump module, 5 words.  Sent into a node to make that node
          store subsequent stream into its RAM.

   4AB7   @p+ a! . @p+
    adr      address        Address for first word stored.
     nw      count          nw+1 words follow this module.
  24861   dup push if then  Count for loop (s40 fixup)
   58B4   @p+ !+ . unext    Pump following words.

Initialization Postamble, 5 words.  Follows each load pump seg
          to initialize a, b, and p.

   4BB7   @p+ b! . @p+
     b      b reg (outport)
     a      a reg (inport)
  2BDBD   a! @p+ push ;
     p      p reg
```