Smart*Firmware*™

http://www.codegen.com
mailto: info@codegen.com

1. Introduction

Welcome

SmartFirmware is an implementation of the OpenFirmware standard (IEEE Std 1275-1994 plus errata) designed for embedded systems. A more than passing familiarity with the standard and ANSI Forth is assumed in this document. We make no attempt to teach OpenFirmware, Forth, or C here.

Features

- Written entirely in ANSI C to be simple, maintainable, portable, and very easy to customize.
- Additional code may be written in a mixture of ANSI C, Forth, or Fcode as desired.
- Modular design allows selecting specific functions and packages to be burned into a ROM.
- Includes an OpenFirmware-compliant interpreter, user interface, application interface, client interface, and tokenizer
- Detokenizer included to aid debugging embedded Fcode
- Unix hosted configuration for debugging and testing as well as embedded configuration for burning into a ROM.
- BOOTP/DHCP/RARP/TFTP support module for booting from networks.
- Display package for simple frame buffers and a terminal emulator.
- Drivers are available for several common PCI and ISA devices including Digital ethernet chips, Symbios SCSI chips, various PCI-ISA bridge chips, and ISA serial chips, among others.
- Custom ANSI C-to-Fcode/Forth compiler is available

Please contact us for pricing and availability or generation of additional drivers and modules, or about our C compiler.

2. Basics

This describes the design of SmartFirmware, to set a base for the customization descriptions in following chapters.

Files

The source files are separated into several directories. The upper level contains files common to most platforms. Subdirectories are used for bus-specific code, such as "pci", "isa", and "scsi", or for specific platform build directories, such as "unix", "i386", and "bebox".

Main files

```
Makefile       # Unix makefile for various builds
cour8x16.font  # default font for machdep.c below
cour16x23.font # optional larger font
cour32x44.font # optional really huge font
8x16.font      # another 8x16 font

defs.h         # essential definitions and types
errs.h         # error codes go here
logo.h         # default 64x64x8 logo for CodeGen, Inc.

ctype.h        # simple versions of these files for
stdlib.h       #    embedded use when building ROM
string.h       #    images of SmartFirmware

admin.c        # OpenFirmware Administration commands
client.c       # OpenFirmware Client Interface
cmdio.c        # command-line I/O & editing routines
control.c      # Forth and Fcode control flow words
cpu.c          # template for building /cpu node
deblock.c      # disk deblocking package
debug.c        # Forth debugging command group
device.c       # device-path manipulation Forth words
disklbl.c      # disk label package
display.c      # display package and terminal emulator
```

```
errs.c          # strings for error numbers go here
exec.c          # Forth and fcode execution & parsing
failsafe.c      # failsafe I/O driver for debugging
fb.c            # frame-buffer package for display
forth.c         # Forth words that are not fcodes
funcs.c         # basic words that are also fcodes
funcs64.c       # words to support 64-bit extensions
main.c          # main() - perform OpenFirmware bootup
memory.c        # template for building /memory node
nvedit.c        # on-screen script editor
obptftp.c       # BOOTP/TFTP package
other.c         # other misc. OpenFirmware words
packages.c      # basic package manipulation words
root.c          # template for building / node
stdlib.c        # simple versions of standard C routines
stlb.c          # software TLB to aid 32<->64-bit translations
sun.c           # Forth words compatible with Sun's OBP
table.c         # low-level data-structure manipulation
token.c         # OpenFirmware tokenizer & detokenizer
test-*          # various regression test command files
```

Executable loader files

The contents of subdirectory "exe" are as follows. It supports plug-in binary image loaders.

```
coff.c          # COFF/ECOFF binary image loader
dumpcoff.c      # test program to display COFF files
dumpelf.c       # test program to display ELF files
elf.c           # ELF binary image loader
elf64.c         # ELF (64-bit) binary image loader
exe.c           # plug-in binary image loader manager
exe.h           # header for plug-in binary image loaders
gzip.c          # gunzip "loader" - needs freeware zlib
loadfc.c        # Forth and Fcode loader
```

Filesystem files

The contents of subdirectory "fs" are as follows. It supports plug-in filesystem readers.

```
bsdpart.c       # BSD partition map loader
bsdufs.c        # BSD FFS/UFS filesystem loader
dos.h           # DOS FAT headers
dosfat.c        # DOS FAT filesystem loader
```

```
dospart.c       # DOS partition loader
fs.c            # plug-in filesystem loader manager
fs.h            # header for plug-in filesystem loaders
iso9660.c       # ISO-9660 CD-ROM filesystem loader
```

ISA files

The contents of subdirectory "isa" are as follows. This adds basic support for the PC-compatible ISA bus.

```
isa.c           # basic ISA bus driver package
isa.h           # headers for ISA devices
kbd.c           # PC keyboard driver
ne2000.c        # NE2000-compatible Ethernet card driver
ns16550.c       # serial port driver
vga.c           # VGA-compatible display driver
```

PCI files

The contents of subdirectory "pci" are as follows. This adds support for the PCI bus.

```
dc21140.c       # dummy DEChip 21140 driver for testing
decether.c      # DEChip 21x4x ethernet driver
fakepci.c       # dummy PCI driver for unix host
fakepci.h       # externs for dummy PCI routines
ncrscsi.c       # Symbios 53C8xx SCSI driver
pci.c           # basic PCI bus driver package
pci.h           # definitions specific to the PCI bus
pcialloc.c      # internal PCI allocation routines
pcicode.c       # current list of PCI vendor/device IDs
pcidisp.c       # generic PCI display driver
pciisa.c        # driver for various PCI-ISA bridges
```

SCSI files

The contents of subdirectory "scsi" are as follows, adding support for SCSI buses.

```
scsi.c          # common routines useful for SCSI drivers
scsi.h          # headers of common SCSI routines
scsidisk.c      # generic SCSI disk driver
```

Unix files

The contents of subdirectory "unix" are as follows. These files allow running SmartFirmware under a standard Unix environment for testing and debugging.

```
be-predefs.h   # pre-include definitions for BeOS
beapp.cc       # C++ file to replace main.c for BeOS
BeOS.proj      # MetroWerks project to build on BeOS
BeOS.rsrc      # resources for building under BeOS
fakedisk.c     # fake disk driver - maps to Unix files
mac.c          # Macintosh-specific code
machdep.h      # machine-dependent definitions
machdep.c      # machine-dependent code goes here
Makefile       # build on Unix
mw-predefs.h   # Metrowerks prefix header file
mw-project.hqx # Metrowerks project + other Mac files
```

i386 files

The contents of subdirectory "i386" are as follows. This supports SmartFirmware on standard i386-compatible PCs using the FreeBSD boot-loader. The boot-loader provides a standard 32-bit environment for SmartFirmware to run under. SmartFirmware takes over the hardware in place of the FreeBSD kernel and can be booted directly off of a FreeBSD filesystem or floppy.

```
divdi3.c       # math runtime function from BSD
isabase.c      # low-level ISA support routines
machdep.c      # i386-specific code
machdep.h      # i386-specific definitions
Makefile       # i386 build on Unix
moddi3.c       # math runtime function from BSD
pcibase.c      # low-level PCI support routines
qdivrem.c      # math runtime function from BSD
quad.h         # math runtime function from BSD
start.S        # i386 startup assembly for launching
udivdi3.c      # math runtime function from BSD
umoddi3.c      # math runtime function from BSD
```

BeBox files

The contents of subdirectory "bebox" are as follows. This supports SmartFirmware running on the dual-PowerPC 603e BeBox platform. It is booted in place of the BeOS kernel off of a floppy and takes over the hardware.

```
be-predefs.h   # pre-include definitions for BeBox
bebox.h        # BeBox-specific definitions
BeBox.proj     # MetroWerks project for BeBox
machdep.c      # BeBox-specific code
machdep.h      # more BeBox-specific definitions
pcibase.c      # low-level PCI support routines
```

SmartFirmware is designed so that it is easy to add or remove packages from a particular build, simply by modifying the lists of desired packages and word-sets in *machdep.c* and then *Makefile* to compile and link the desired files. It is a good idea to create custom *machdep.h* and *machdep.c* files by copying another project's versions for a new project, then editing them for the new platform.

Each OpenFirmware package may be implemented as Forth code or as C code. SmartFirmware implements almost everything in C, with the exception of the default boot script and a few Forth words for ease of tokenizing.

Headers

All the major data structures and types are defined in *defs.h* , with some machine dependent definitions going in *machdep.h* . The comments in the files should be reasonably self-explanatory. Additional bus or subsystem-specific code is under various subdirectories, such as *fs/fs.h* or *isa/isa.h*.

Word sizes

The sizes of various type names must be specified in *machdep.h* . They are typically 8, 16, and 32 bits signed and unsigned values. However SmartFirmware can support 64-bit integer operations and Forth words only if the C compiler supports them, such as some versions of the Gnu C compiler gcc.

A *Cell* is typically the standard word size of a platform. It must be large enough to hold a pointer such as "*ch*ar* ". However it may be larger than a pointer for those compilers that support, say, 32-bit pointers but 64-bit integers.

It is possible to build SmartFirmware with 32-bit *Cells* but with some 64-bit integer support.some 64-bit support is turned on if the macro *__LONGLONG* is defined to be the type of the 64-bit integer, such as "*long long*" for gcc. 64-bit *Cells* are turned on if the macro *SF_64BIT* is defined.

Commands

Commands are simply C functions linked into the Forth environment. They are designed to allow using them in both ways with fairly little effort.

Declaring

A C function that implements a Forth word is defined as follows;

```
Retcode cfunc(Environ *e) { ... }
```

A set of macros in *defs.h* make declaring this somewhat easier for large numbers of *Commands*:

```
C(name) {..}  # define a local static Command
EC(name);     # declare an extern for a Command
CC(name){..}  # define a globally visible Command
```

These macros declare and define routines such as *cfunc* taking one argument, and *Environ** which is named *e*. A typedef for *Command* is also declared to handle pointers to these functions.

Calling

An *Environ* is passed by pointer to all C functions to keep from cluttering up the code with a lot global variables. It also makes it much easier to support a thread-safe implementation in the future. Static global variables and data should only be used for initialization.

All C functions must return either *NO_ERROR* or a specific error listed in the enum *Retcode* in *defs.h* . This mechanism is used to handle Forth exceptions without using*setjmp/longjmp*. A C function that calls other C routines must check the return codes for any error to either handle the error itself or to propagate that error to its caller.

The Retcode errors are translated to strings by the routine *err2str* in *exec.c* when any error must be reported to the user. It may be easier to directly use the routine *cprintf* to display error messages, but this is only useful if the console has been probed and installed. The error codes themselves are defined in *errs.h*, and *errs.c* must be kept synchronized with it.

Stacks

The Forth data and return stacks are manipulated using macros defined in *defs.h* . The stack elements are of the *Cell* type declared in *machdep.h* , which is usually 32 or 64 bits wide. These stacks are maintained as simple arrays within the *Environ* struct with pointers to the current top of each stack.

The data stack is used to pass arguments and perform all the standard Forth functions. The return stack is used to maintain a call-chain of Forth words as they are executed at runtime. It is also used to maintain loop variables for the looping control words. This can get messy to manage the return stack for certain words, but this sort of code is well-commented.

Range checks

The following four macros are used to check the stack ranges. The first two are intended for use in your own *if* and *while* statements. The last two are more generally used throughout the code to check the ranges and return an appropriate *Retcode.*

```
CKSP(e, min, max)       # check data stack
CKRETSP(e, min, max)    # check return stack
IFCKSP(e, min, max)     # check & return error
IFCKRETSP(e, min, max)  # check & return error
```

These are used to increment and decrement the respective stack pointers. Ranges are not checked. They are used by the other macros and are generally not used directly.

```
BUMPSP(e)      # increment data stack ptr by one
DROPSP(e)      # decrement data stack ptr by one
BUMP_RETSP(e)  # increment return stack ptr
DROP_RETSP(e)  # decrement return stack ptr
```

Push & Pop

These are used to access the top element of the stacks (a *Cell*). Ranges are not checked.

```
TOP(e)   # get Cell currently on top of data stack
RTOP(e)  # get Cell on top of return stack
```

These are used to get to a specific (nth) element on the stack without checking to see if it is in range. The topmost element on the stack is numbered zero (0), the element below it is one (1), and so on.

```
STACK(e, n)   # get nth element on data stack
```

These push a value on top of the data stack without checking for a range error. The second form is used to cast a pointer value into a pointer-sized integer before pushing it on the stack. It is generally used to eliminate warnings from some compilers.

```
PUSH(e, val)   # push val on top of data stack
PUSHP(e, val)  # push val after casting it from a pointer
```

These simply drop the stack, either by one element, or by a count (n). Ranges are not checked.

```
DROP(e)       # drop the top element on the data stack
DROPN(e, n)   # drop the top n elements on the stack
```

These pop the top of the stack into the specified variable. The second form allows inserting a typecast to cast the *Cell* to a pointer such as a *Byte** . The first assumes the variable is of type *Cell* . The last allows casting to any integer type. The first two may be overridden by custom macros to handle 64-bit to 32-bit (and 32 to 64) pointer translations.

```
POP(e, var)               # var = TOP(e); DROP(e);
POPT(e, var, ptrtype)     # var = (ptrtype)TOP(e); DROP(e);
POPTYPE(e, var, inttype)  # var = (inttype)TOP(e); DROP(e);
```

Return stack

These do everything the above five macros do, only they operate on the return stack.

```
RSTACK(e, n)
RPUSH(e, val)
RDROP(e)
RDROPN(e, n)
RPOP(e, var)
```

Memory

Memory is handled in as much a C fashion as possible. That is, except to mimic certain Forth behaviors, SmartFirmware uses *malloc* and *free* to manage memory. Simple versions of these routines are provided in *stdlib.c* that allocate and free from a large fixed-size chunk of memory. An additional routine *init_malloc* is used to initialize the memory pool used by *malloc* and *free* . This version of *malloc* may be used if there is no suitable *malloc* available in the runtime environment of a target or compiler.

Initializing

At start-up, the routine *machine_initialize* is called to do any machine-dependent setup. This routine, once it has setup and initialized any appropriate hardware, calls *init_malloc* with a chunk of memory to be used for the malloc pool, the size of which was defined by the macro *MALLOC_POOL* in <*defs.h* >.

Once *malloc* is ready, then an *Environ* is created for the Forth world. *init_environ* is used to initialize an *Environ* , which in turn calls *malloc* to allocate a pool of *MEM_SIZE* bytes for the Forth world. This macro is defined in <*defs.h*> , and the call to *init_environ* is in *main.c*.

Allocating

All the basic data structures (*Entry, Table, Package, Instance, and Environ*) are allocated and freed by appropriately named routines in *table.c* . The externs for these routines are also in *defs.h* . Objects are always allocated from the heap and not the stack to keep things simple, must always be created with *new_object* , and freed with *delete_object* .

Temporary string buffers are frequently placed on the stack for convenience. Other objects are simply *malloc*-ed and *free*-d as needed.

Debugging

If the macro DEBUG_MALLOC is defined, the malloc code will switch on additional memory debugging code which does simple checks to catch overruning the head or tail of *malloc*-ed objects, and verifying that *free*-d blocks are not used and that *malloc*-ed objects are initialized before use.

Strings

The OpenFirmware specification requires handling at least the Pascal and Forth style strings. Implementing SmartFirmware in C also requires handling C strings in a reasonably seamless fashion.

String types

There are three sorts of strings managed within the SmartFirmware software. There are C strings, Pascal strings, and Forth strings. C strings (*Byte**) are null-terminated arrays of characters. Pascal strings (*uByte**) use the first byte of the string array to store the length and are thus limited to 255 bytes in most implementations. Forth strings are two separate array and length values (*Byte*, Int*) and are managed as such.

Pascal strings

SmartFirmware uses the Pascal string as its internal canonical form when storing a string in a data structure. When using a string in a function, it is converted to a Forth string using the routine *setstrlen* defined in *table.c* , which takes a string in any of the three forms and returns a string adjusted to be in Forth style, with the actual length returned in another parameter.

Forth strings

Forth strings are then passed to most other C routines. If it takes both a *Byte** and an *Int* parameter, it usually expects a Forth string. Routines taking both arguments always call *setstrlen* in case the strings are in C or Pascal form.

This makes it very easy to pass in C strings or Pascal strings by using the length field set to one of two magic negative values, as defined by the macros *CSTR* and *PSTR* in *defs.h* . Anytime a C string is passed to a routine, simply pass in a length of*CSTR* and *setstrlen* takes it from there.

Copying strings

The routine *lstrdup* , (also defined in *table.c*) with its associated macros *cstrdup* and *pstrdup* , allow creating and copying any form of string using *malloc*. *lstrdup* is used when the type of string is unknown but a length parameter is available. Otherwise, the type of

string determines the right macro to use: *cstrdup* to copy C strings, and *pstrdup* to copy Pascal strings. *lstrdup* always returns a null-terminated Pascal string allocated using *malloc* , so it may be freed by passing the pointer to *free* . The returned pointer plus one is always a null-terminated C string.

Fcodes

The predefined fcode values from the OpenFirmware specification are also used as the internal "compiled" form within SmartFirmware. This eliminates any machine-dependent compilers, debuggers, and so forth, and also uses less memory but at some cost in performance. However, since this code is intended only to boot machines, and most commonly RISC machines, it is extremely unlikely that any performance loss will be a serious problem.

Execution tokens

The internal execution token of a Forth word is usually the same as the fcodes generated by an OpenFirmware tokenizer. Unfortunately, there is no provision made in the OpenFirmware specification to generate Fcodes on-the-fly for any newly created words. The vendor-specific fcode range is too small and has to be reset whenever a new Fcode program is loaded from a device.

SmartFirmware allocates Fcodes that are well outside the fcode range specified by OpenFirmware, specifically values 0x1000 and greater. These are then encoded into the compiled form in a manner which allows easy decoding while remaining compliant to the OpenFirmware specification by using the vendor-specific 0x60 range as an "escape" value.

The array *xtoks* in the *Environ* struct maintains the list of currently defined execution tokens. Each element points to an *Entry* defining a Forth word or method. The index of that element in *xtoks* is also its execution-token value minus 0x1000. Each *Entry* also stores its own execution token value.

Initial values

None of this is generally visible to the programmer except when initializing Forth words using arrays of the *Initentry* struct. If an fcode is explicitly specified in an *Initentry* for a word, that value is used as the execution token of that word.

Otherwise*INVALID_FCODE* must be specified and a new execution token greater than or equal to 0x1000 is automatically generated (and never reused).

Tokenizing

The vendor-specific 0x60 and 0x70 fcode ranges are used internally to mark certain control words for tokenizing. These are generally no-ops at runtime, but are needed as place-holders to calculate offsets and ranges when tokenizing.

All vendor-specific ranges are only used internally in memory and are never written out by the tokenizer. Fcode programs generated by the SmartFirmware tokenizer are completely compatible with other vendors' implementations of OpenFirmware.

3. Machine dependencies

A new port should be managed simply by copying and modifying *machdep.h* and *machdep.c* from some existing port. Some samples for these machine-dependent files may be found under the *unix*, *i386*, and *bebox* subdirectories.

machdep.h

This file contains all the types and definitions required to properly describe the target host. They must all be specified, and the correct types for the required word-widths must be selected. Most will probably not need to be changed. The file is well-commented and more up-to-date than this document, so only the more important portions are covered here.

Macros

Macros that control the switching on of certain features may be defined within *machdep.h* or may be added to the *Makefile* --whichever is more convenient. *LITTLE_ENDIAN* must be defined if your system does not do it for you. The default is big-endian, such as for the Motorola 680x0 processors.

The amount of memory to be used by the SmartFirmware image is also defined here with *MALLOC_POOL* and *MEM_SIZE*. All memory including that for the Forth environment is allocated out of the *MALLOC_POOL* , so it must be large enough to allocate *MEM_SIZE* as well as all other macros. The defaults are defined as fractions of the total *MALLOC_POOL*.

The default values for the various stack sizes are safe to change to larger values but be careful when making them smaller. Some of the other macros are already at their minimum value as specified in the OpenFirmware standard, so they should only be increased if necessary.

Integral types

A set of typedefs define the word sizes for the target host and the sizes used by the Forth environment. Typically, only the former needs to be ported to a host, with some caveats as follow.

A *Cell* and a *Ptr* must both be large enough to hold a pointer of any type.

DoubleNums require 64-bit support from the target compiler. If the compiler supports 64-bit integers, the macro *__LONGLONG* must be defined to that type (usually *long long*). This macro may be defined in CFLAGS in the *Makefile* or at the top of *machdep.h.*

The various **_SIZE* and **_MASK* macros must be defined correctly. Unfortunately, there is no easy way to reliably extract this information from a C compiler or standard header files, so this must be done by hand.

Fonts

Finally there are some macros that describe the built-in default system font. These should only be changed if either of the shipped fonts are not going to be used. If so, at least all the printable ASCII characters must be included in the font, from ' ' (space) to '~ ' (tilde). 8-bit characters may also be included if desired.

The OpenFirmware specification requires that the first scan-line of a font glyphs' bitmap be all zeros, and that the last scan-line must not be included in the bitmap, but is instead assumed to be all zeros. All this affects FONT_HEIGHT, which SmartFirmware uses as the real height of the font including the virtual bottom row of zeros.

A struct "eself" may be created if the port needs additional data stored within the *Environ* struct defined in *defs.h* . This field may be initialized and filled in when needed by the *machine_init_args* routine described below.

machdep.c

This file contains code that must be ported to the target host, such as code to access registers, memory tests, non-volatile RAM, timers, and so on. The key routines that must be ported are named *machine_** . There are some other Forth words in this file, but they are unlikely to need to be modified. It is also well commented, so the following notes serve only as an addendum rather than a replacement for *machdep.c* .

Memory initialization

The first routine to port is *machine_initialize* since it must first initialize *malloc* so that the rest of SmartFirmware can be initialized. The default version simply creates an enormous static block of memory, which should work if the target C compiler's runtime environment supports large static blocks of memory. (This block does not have to be initialized--*init_malloc* will take care of that.)

The globals *g_machine_memory* and *g_machine_memory_size* must also be initialized to the start of RAM and the amount of real RAM in the machine. *init_malloc* is called with only a portion of this memory (*MALLOC_POOL* defined in <*defs.h*>), to leave the rest available for the client interface.

Debug output

The *dprintf* routine is useful to bring up a port of SmartFirmware. It is typically defined to use the failsafe I/O routines defined later in the file. The failsafe I/O routines typically use a serial port for I/O, such as for the i386 and BeBox ports.

failsafe_read and *failsafe_write* routines should also be provided, so that if a console device cannot be accessed for some reason, at least the error messages won't get dropped in the bit-bucket. These simply use a hard-wired serial port for fall-back I/O in the i386 and BeBox ports, and stdio under Unix.

Memory and I/O access

The memory and I/O access routines are required to get/set unaligned words from memory. Similar routines are used to access device registers perhaps through I/O space. The generic versions shipped with the default *machdep.c* may ben sufficient for most needs, but they should be checked to be sure. The versions for the i386 and BeBox ports may be more instructive.

Timers

The timing routines in SmartFirmware use *machine_gettime* to do the real work. If the target compiler has a version of the standard Unix function*gettimeoday* in its runtime library, this routine will most likely not need to be modified. Otherwise, some other method must be used.

It is possible to bring up a new port without having to manage timers and interrupts at all. Simply defined the routine *u_sleep* to be a simple timed loop, and enough of Smart-Firmware will run to bring up the port. Then the proper timer code can be turned on once *dprintf* is available to debug it.

machine_gettime is used when pausing for a specified time and for handling alarms by using polling loops. Real timer interrupts could be used, but SmartFirmware is not interrupt-safe, so some method of synchronizing interrupt tasks would still need to be used. If real interrupts are necessary, some of the routines that use *machine_gettime* in *machdep.c* may also need to be modified.

Device probing

machine_probe_all is used to probe the hardware at start-up before a prompt would be displayed. This will almost certainly need to be customized. One of the things it may do is identify a console device. The i386 and BeBox ports probe for both a PCI and an ISA bus.

Binary images

machine_init_program and *machine_go* are used to setup and launch a machine-dependent binary image. The defaults assume that the image is simply a pointer to the start of a C function and can simply be called using a C pointer-to-function. This code may in turn call the plug-in binary image loader defined under the *fs* subdirectory by using the *exec_is_exec* and *exec_load* routines. The actual binary images supported are listed here in the global variable *g_exec_list.*

Client-interface callback

machine_callback is used for client-interface callbacks. The default assumes that the call-back is simply a pointer to a C function and calls it. This does not need to be ported if the client-interface will not be used.

Built-in words and packages

This file also contains several global lists which are used to configure the modules and packages to be linked into the final image. These lists are called *init_list* and *install_list* , for a list of pointers to *Initentry** tables and a list of pointers to initialization *Commands* .

Each module that wishes to install a set of words into the global Forth dictionary creates an initialization list at the bottom of the file (see *funcs.c* or *devies.c* for an example). Each package that needs more complex initialization creates a function that does the work. *machdep.c* determines which of these will actually be linked into the final image. This process is described in more detail in Creating new words.

Built-in drivers

Drivers for built-in PCI devices may also be specified using the *pci_drivers* global list. This is used by pci.c when probing the PCI bus. Built-in ISA drivers are handled similarly by defining the devices in the global *isa_devices.* Built-in filesystem support may be specified by the global *g_filesys.*

Non-volatile RAM

Non-volatile memory (NVRAM) is managed here by routines that read a static array to initialize or use to set default values. The sample code has code that ignores all writes. This should be sufficient to bootstrap a preliminary port of SmartFirmware then adding full NVRAM support later. A static array should still be used as a fall-back in case NVRAM is inaccessible or corrupt.

The initial definitions for "input-device" and "output-device" in NVRAM are important when bringing up a new port of SmartFirmware. They should first be defined as "/failsafe" to bring up a console using the fall-back failsafe I/O routines. (This driver is defined in the file *failsafe.c.*) Once devices are properly probed and visible to the system in the device tree, then these definitions may be changed to the standard "keyboard" and "screen" aliases, with the actual devices for the aliases possibly being selected by *machine_probe_all* .

The routine *machine_init_args* is called from *main()* once the Forth environment has been created and initialized. At this point Forth is ready to run but no devices have been probed. *machine_init_args* is passed the arguments that were passed to *main* if they are useful for a particular port. It may initialize the "self" field of its *Environ* *parameter if desired.

4. Creating new words

This chapter describes how to create new Forth words and packages and install them into SmartFirmware.

Words

A word is a Forth executable object, usually a C function or a Forth/FCode routine. First we will implement a sample word in C, and show how to put it into SmartFirmware.

Declaring

A file named *hello.c* is created and contains the following:

```
#include "defs.h"

C(hello)
{
    cprintf(e, "Hello, world!\n");
    return NO_ERROR;
}
```

This declares and defines a local static Forth *Command* . A *Command* must always return an appropriate *Retcode* , in this case *NO_ERROR* . The macro "**C**" is used to declare the function, which will be named *f_hello* . All these types and macros are defined in *defs.h* .

Initentry

To enter this word in SmartFirmware's global dictionary, it must be first put into an initialization list (composed of *Initentry* objects).

```
const Initentry init_hello[] =
{
    { "hello", f_hello, INVALID_FCODE },
    { NULL, NULL }
};
```

This list must always end with the "NULL, NULL" combination which properly terminates the list. The Forth world will know this function as the string "hello". It will call the routine *f_hello* whenever the word "hello" is invoked.

The *INVALID_FCODE* value tells the SmartFirmware initialization code that this routine does not have a pre-defined FCode value in the OpenFirmware specification, so automatically generate an arbitrary value at runtime that is safely out of the pre-defined ranges in the specification.

init_list

The last thing to do is add init_hello to the global list-of-init-lists in *machdep.c.* Look for the code that begins with *extern Initentry* in that file. Our *init_hello* will be added to the end of that list, and also inside the *init_list* global.

```
...
/* bunch of other extern Initentrys here */
extern const Initentry init_display[];
extern const Initentry init_hello[];    /* add this line */

const Initentry* init_list[] =
{
    ...
    init_display,
    init_hello,   /* and add this line */
    NULL          /* must be NULL terminated! */
};
```

Now re-compile *machdep.c* and relink SmartFirmware, start it up, and type "hello" at the prompt. It should respond with "Hello, world!".

More Initentry

An *Initentry* can contain other types of words. Here is a way to create a word as a string of Forth commands:

```
const Initentry init_hello[] =
{
    { "hello", f_hello, INVALID_FCODE },
    { "hello2", (Command)".\" Hello, world!\" cr",
            INVALID_FCODE, F_NONE, T_FORTH
            HELP("(--) display hello message")
            },
```

```
    { NULL, NULL }
};
```

Recompile *hello.c*, relink, and now invoking "hello2" will do the same thing as "hello". Notice that machdep.c does not need to be recompiled.

There are more involved forms of the various arguments to an *Initentry*, used for special compilation words, pointers to fields of C structures, and constants. Browse some of the SmartFirmware sources to see the different ways that a word can be created, should you need anything more complex than the above two forms.

The *HELP* macro inserts a message to be displayed by the *help* command for the *hello2* word. This help may be turned off and on using the macro *DETAILED_HELP* in *machdep.h*. Detailed help messages take up more data space so if memory is tight, it may be turned off. Note that there must be no preceding comma before invoking the *HELP* macro. It automatically inserts a comma if *DETAILED_HELP* is on.

Packages

A package can be a special-function subroutine library such as *deblock.c* , or it can be a bus or device driver, such as *pci.c* . The simplest packages to study are *disklbl.c* (for finding and managing disk labels and partitions, if any) and *failsafe.c* (for faking a console and keyboard using fail-safe I/O routines in *machdep.c*).

Packages may be written in Forth or may be embedded as FCode on a plug-in card. The User Manual describes more about how OpenFirmware devices trees, paths, and packages interact. The OpenFirmware specification should be consulted for the final word in these matters.

struct self

One important C idiom that the C packages use is defining a *struct self* at the top of the file. The field *struct self *self* is declared within *Instance* in *defs.h* but is never explicitly defined. Each package can create and manage its own *struct self* without worrying about some other package's definition. This allows C code to easily access instance-specific variables without having to go through the Forth environment. This does mean that a package cannot directly access another package's private C variables, but that's probably a good thing anyway.

Methods

Methods are declared and used for packages pretty much as other Forth words. The only difference is that these words are package-specific and are not installed in the global dictionary but rather in the specific package's own private dictionary. These are then called methods of the package.

A package should always have an *open* and *close* method. Empty stubs are fine, but they should always return *NO_ERROR* . *open* must also push *FTRUE* on the top of the data stack.

Initialization

A package is also initialized differently from plain-vanilla Forth words. Because a package is always created relative to a parent at runtime, a C function must be used to initialize the package. This C routine must create the new *Package* object, and initialize its method table. Typically it will also allocate and initialize its *Self* struct.

For instance, in disklbl.c, first the C methods are placed into an Initentry list to be used later.

```
static const Initentry disklabel_methods[] =
{
    { "open", f_open, INVALID_FCODE },
    ...
    { NULL, NULL }
};

CC(install_disklabel)
{
    Package *pkg = new_pkg_name(e->packages,
            "disk-label");

    return init_entries(e, pkg->dict, disklabel_methods);
}
```

The routine *install_disklabel* is then entered into a global list in *machdep.c*.

```
EC(install_disklabel);    /* declare it */
...
const Command install_list[] =
{
    ...
    install_disklabel,    /* and insert it at the end */
```

```
    NULL
};
```

The *install_disklabel* routine must call *new_pkg_name* to create the new package with the specified parent (in this case the node "/packages") and then call *init_entries* to initialize the new package's dictionary. Additional package-specific initialization could be done here, but note that the *open* method is responsible for all instance-specific initialization.

5. Compiling, Running, Etc.

This chapter describes how to compile and run SmartFirmware.

Building

The usual way to build SmartFirmware is using the included Unix*Makefile* . It builds SmartFirmware on a variety of Unix systems, including FreeBSD, NetBSD on Alpha, Solaris on the Sparc, and under BeOS on either Macintosh or BeBox. Most configurations will require either modifying the Makefile to select the appropriate macro definitions for CFLAGS and the like.

New targets

Each target port or platform is placed in a subdirectory. Each has its own *Makefile* that refers to common files or machine-specific files as needed. Some implementations of *make* are unable to handle the relative paths to common source files. For these systems, the common files may simply be linked into the platform's subdirectory before building. Or the BSD make may simply be ported to the target platform.

Macintosh

For the Macintosh, a CodeWarrior project file is included named *mw-project.hqx* . Simply decode it on a Mac to create the SmartFirmware project file and the Mac resource file, open the project file, then simply select the Make menu to build a Mac version of SmartFirmware.

The Mac version uses an off-screen PixMap and the SmartFirmware 8-bit display package in *fb.c* rather than using the *stdio.c* interface. This allows the display code to be exercised and tested before being burned into a ROM. However, since the off-screen PixMap is blitted to the Mac window every time a character is displayed, performance is lousy since we try to remain reasonably Mac-friendly by calling *WaitNextEvent* .

BeOS

Building under the BeOS requires the unlimited Metrowerks linker. The file *BeOS.proj* is a Metrowerks project files and *BeOS.rsrc* is the resources for the project. The *Makefile* may also be used if the BSD *make* has been ported or uploaded to the BeOS.

This version of SmartFirmware uses an off-screen bitmap and uses the display package rather than more Unix-like stdio interface. SmartFirmware proper runs as a separate thread to closely mimic the behavior of an embedded system. A separate thread is responsible for keyboard and screen I/O.

Running

Running the Unix SmartFirmware on a Unix-like system is simply typing *of* at the prompt. The unix SmartFirmware initializes its fake devices, creates a dummy console and keyboard device attached to stdout and stdin (respectively), and enters a standard Forth command eval loop.

On the Mac, double-click the application "OpenFirmware" to launch it. After that, it's pretty much like the Unix implementation, except there is no way to pass in any command-line arguments so everything must be run from inside the Forth environment. The BeOS port is similar to the Mac version.

Running other versions of SmartFirmware depends upon each port. The FreeBSD i386 port runs by using the FreeBSD boot-loader to setup 32-bit protected mode, and thus may be launched instead of the kernel. Other ports may be placed into ROM and may be launched simply by powering up a VME graphics card, or booted off of a floppy.

Tokenizing

Command line

Under Unix, "of" takes several command-line arguments. The first allows running an arbitrary Forth or Fcode file on the local host:

```
$ of -file forth-or-fcode-file
```

The second form runs just the tokenizer:

```
$ of -tokenize tokenizer-source-file
```

Forth prompt

Both of these commands may be run from within the Forth environment (which is the only way to run them on the Mac or BeOS). To load a file:

```
ok load-file forth-or-fcode-file
```

To tokenize a file:

```
ok tokenize tokenizer-source-file
```

The tokenizer source file should begin and end with the tokenizer words *fcode-version2* and *fcode-end*, as the OpenFirmware errata now requires.

Detokenizing

SmartFirmware also supports detokenizing an Fcode file. This is essentially an Fcode disassembler and is useful for debugging. To run it at the Unix prompt:

```
$ of -detokenize fcode-file
```

From within SmartFirmware do this:

```
ok detok fcode-file
```

The output is rather verbose and probably somewhat cryptic. We apologize for any inconvenience. It's easiest to look through the sources to see what the various values mean.

6. Plug-in executable images

The plug-in layer is a non-standard interface to more easily support a variety of executable image formats instead of writing custom loaders for each target platform. The source files of the plug-in executable layer are located under the *exe* subdirectory.

The plug-in executable layer sits above the low-lever device loader code and below the user-level *load* command. After an image is loaded from a disk or over the network, it is then checked to see if a plug-in supports the image format, and if so, it may be executed.

Interface

The interface to the plug-in executable layer is through the routines listed in *exe.h*:

```
extern Bool exec_is_exec(Environ *e);
extern Retcode exec_load(Environ *e);
extern Bool exec_length(Environ *e, uInt *len);
extern Sym_table *exec_load_symbols(Environ *e);
void exec_free_symbols(Environ *e, Sym_table *tab);
extern Sym_ent *exec_sym2addr(Environ *e, Sym_table *tab,
        Byte *sym, Int slen);
extern Sym_ent *exec_addr2sym(Environ *e, Sym_table *tab,
        uLong addr);
```

Calling plug-in routines

The callers of these routines are in the platform-dependent *machdep.c* file:

```
CC(machine_init_program)   /* (--) */
{
  /* see if we have some supported executable image */
  if (exec_is_exec(e))
  {
       exec_free_symbols(e, e->loadsyms);
       e->loadsyms = exec_load_symbols(e);
       return NO_ERROR;
  }
```

```
  return E_BAD_IMAGE;
}
```

The exec_is_exec routine simply iterates through the list of built-in supported formats to identify the image at e->load.

```
CC(machine_go)                  /* (--) */
{
  Retcode ret;

  if (exec_is_exec(e))          /* sanity check */
  {
       ret = exec_load(e);

       if (ret == NO_ERROR)
       {
            exec_free_symbols(e, e->loadsyms);
            e->loadsyms = exec_load_symbols(e);
       }

       /* machine-dependent launch through e->entrypoint */
  }
  else
       ret = E_BAD_IMAGE;

  return ret;
}
```

This code may additionally include custom loaders if desired, but it will generally be easier to create a custom plug-in executable module as described below.

(The *machine_init_load* routine must correctly point *e->load* to a large area of free memory where it is safe to load an image.)

Describing built-in formats

The *machdep.c* file must also list the image formats that are supported:

```
extern Exec_entry exec_fcode;
extern Exec_entry exec_forth;
extern Exec_entry exec_coff;
extern Exec_entry exec_elf;
extern Exec_entry exec_elf64;
extern Exec_entry exec_gzip;

Exec_entry *g_exec_list[] =
```

```
{
  &exec_fcode,
  &exec_forth,
  &exec_coff,
  &exec_elf,
  &exec_elf64,
  &exec_gzip,
  NULL
};
```

Each *Exec_entry* is defined in a separate file that supports that image format. In the example above, this platform will support Fcode, Forth, COFF and ELF format images. The plug-in layer iterates through all supported images to first identify a supported image and then load it.

Supplied image formats

Fcode and Forth images may be supported by any platform.

The *coff.c* plug-in requires that the macros *COFF_MAGIC_0* and *COFF_MAGIC_1* be defined in the platform's *machdep.h* file. These determine the correct magic number values of the image as generated by your platform's linker. Please see the file *exe/coff.c* for more details.

elf.c also requires identifying the correct machine for the target by defining *ELF_OUR_MACHINE* to the correct value in *machdep.h*. Please see the file *exe/elf.c* for more details.

The files *dumpcoff.c* and *dumpelf.c* are included to display the contents of their respective binary images on stdout, and also display their respective magic numbers. Compile these files using your native host's compiler to build the programs *dumpcoff* and *dumpelf* respectively, then run them on a sample image.

Sample Forth image format

The file *loadfc.c* includes code for loading Forth and Fcode "images" following an Open-Firmware Recommended Practice document. The Forth image loader is a good outline for creating custom image loaders. A loader begins with these include files:

```
#include "defs.h"
#include "exe.h"
```

is_exec function

Each loader has two entry points. One routine determines if the image pointed to is an executable of the correct type or not:

```
static Bool
forth_is_exec(Environ *e, uByte *load, uInt loadlen)
{
  if (loadlen > 2 && load[0] == '\\' && load[1] == ' ')
       return TRUE;

  return FALSE;
}
```

The entry point is passed a pointer to the image loaded into memory and its length. This pointer is typically e->load but may not necessarily be so. An image loader should look for some magic bytes and possible verify a checksum to insure the image is of the supported type.

The Forth *is_exec* routine looks for the first characters of the image to be the beginning of a Forth text comment and if so, assumes it is a Forth image. It could also check that the rest of the image is all only ASCII characters if desired.

Return *TRUE* if the image is the correct format and loadable and *FALSE* if not.

load function

The second entry actually loads the image and returns the entrypoint into the image:

```
/* run the Forth image - loading makes no sense for Forth
 */
static Retcode
forth_load(Environ *e, uByte *load, uInt loadlen, uInt *entry-
point)
{
  *entrypoint = -1;
  return interp_text(e, (Byte*)load, (Int)loadlen);
}
```

This routine interprets the Forth text at the load address directly rather than return an entrypoint into the image. The entrypoint **-1** tells the caller that there is nothing more to do here.

An image loader may copy portions of the image into some other piece of allocated memory, perform relocation and other patching operations, then return a pointer to the

entrypoint in the new image. It may simply assume that the target memory is safely outside of the SmartFirmware's address space and directly copy portions of the image straight to where it needs to be. Please see the *coff.c* and *elf.c* files for details.

Exec_entry

Last the entry points are listed in an *Exec_entry* structure along with a name to identify this loader:

```
const Exec_entry exec_forth =
{
  "Forth",
  forth_is_exec,
  forth_load
};
```

Add a reference to this entry point in machdep.c's g_exec_list, and SmartFirmware now understands the new image format.

7. Plug-in filesystem

The plug-in filesystem code is a non-standard extension to OpenFirmware that permits listing and loading files within filesystems on random-access media. The source files of this layer are located under the *fs* subdirectory.

The plug-in filesystem works above the device-driver layer of code and below the SmartFirmware *disk-label* package.

How it works

When the SmartFirmware disk-label package (implemented in file disklbl.c) is opened by a disk device, it automatically inserts a new method into the disk device named *list-files*. This hook allows the filesystem layer to be used to display the contents of filesystems rather than simply loading them.

Otherwise, the disk-label package simply calls the file-system hooks defined in fs.c to do the work. These hooks iterate through the built-in supported filesystems and try to identify various partition and filesystems on a drive.

A user-level command *list-files* is also added which behaves much as the user-level *load* Forth command but which also takes an optional directory string argument. The *list-files* command then calls the *list-files* method on the device just as the *load* command calls the *load* method.

list-files device-path:(partition,...) (filesys-path)

List files under a specified device possibly under an optional directory name.

Examples:

```
ok list-files /pci/scsi/disk@3
DOS partition 0: fat12 (0x1)
DOS partition 1: bsd (0xA5)
DOS partition 2: unused (0x0)
DOS partition 3: unused (0x0)
ok list-files /pci/scsi/disk@3:0
```

```
 Volume is DOS
IO.SYS          33430
MSDOS.SYS       37394
COMMAND.COM     47845
[DOS]
CONFIG.SYS      71
AUTOEXEC.BAT    81
...
ok list-files /pci/scsi/disk@3:1
BSD partition a: ffs (7)
BSD partition b: swap (1)
BSD partition c: unused (0)
BSD partition d: unused (0)
...
ok list-files /pci/scsi/disk@3:1,a
[.]
[..]
[dev]
[usr]
kernel
[root]
[var]
...
ok list-files /pci/scsi/disk@3:1,a /root
[.]
[..]
.profile
.exrc
mbox
.Xdefaults
...
ok boot /pci/scsi/disk@3:1,a /kernel -s
...
```

Partitions

Disk partitions are simply viewed as directories in this scheme, only with short names and no subdirectories. This allows easy nesting of partitions and filesystems mostly to support x86 systems. For instance, the example above denotes a DOS partition **1** that contains a BSD root partition **a** that in turn contains a BSD filesystem under which we are interested in the contents of **/dir**.

Typically partitions should be noted in a comma-separated list after the device name and a colon. File path names are usually indicated after a space as part of the option

arguments to a device. Beware of back-slashes for DOS path names followed by a space as Forth may see it as a comment.

Interface

The *g_filesys* list must be defined in the target *machdep.c* file to list all supported built-in filesystems:

```
extern Filesys g_dos_partition;
extern Filesys g_dos_fat;
extern Filesys g_bsd_partition;
extern Filesys g_bsd_ufs;
extern Filesys g_iso9660_fs;

Filesys *g_filesys[] =
{
  &g_dos_partition,
  &g_dos_fat,
#if defined(__FreeBSD__) || defined(__NetBSD__) ||
defined(__OpenBSD__)
  &g_bsd_partition,
  &g_bsd_ufs,
#endif
     &g_iso9660_fs,
  NULL
};
```

Additional filesystem and partition schemes may be created and added to the list.

filesys entry

There is only one entry point to each particular filesystem. This is through the single *action* function defined as follows:

```
Retcode (*action)(Environ *e, Filesys_action what, Instance
*disk,
        Byte *path, uLong loc, uByte *buf, uInt size,
        uByte *retbuf, uLong *val);
```

The simplest filesystem code to explore is in the files *dospart.c* and*bsdpart.c* which handles the DOS and BSD partition schemes respectively. The actual filesystem code is in *dosfat.c* and *bsdufs.c* and are considerably more complex, especially since much code has been copied and modified from freely available software.

The *action* routine is called with either one of *FS_PROBE*, *FS_LIST*, or *FS_LOAD* Filesys_action values to probe for a filesystem, list files within a filesystem, or load a file from a filesystem respectively.

The rest of the arguments are an opened instance to the *disk* device, a comma-separated list of *path* names, the location *loc* at which this filesystem must begin its operation on the disk, a DMA-safe buffer *buf* to use for reading data from the disk, the *size* of the buffer (at least as large as the disk's reported block size), and a return buffer *retbuf* and its optional length *val* to return any loaded data.

The *retbuf* is also used to return a comma-separated list of probed partitions that is subsequently stored as a property *partition-types* in the disk package. This list is generated in the order the filesystems are identified.

As a filesystem probes, lists, or loads from a device, it must use the *path* argument to determine if it needs to forward those actions to sub-partitions. It must first identify a partition or directory name to the first comma in the *path*. If there is still something left to do, it must forward the request to the generic filesystem layer and let it probe, list, or load from any sub-partitions, after suitably adjusting the *loc* parameter.

8. PCI-bus interface

The PCI bus code supports probing and initializing PCI devices. The source files of this layer are located under the *pci* subdirectory.

While most of the code is machine-independent, a certain amount must be customized to communicate with the PCI configuration registers, memory space, and I/O spaces on the target system. The PCI drivers themselves are machine-independent being written to communicate with the PCI bus layer. Finally, the target *machdep.c* file must specify the built-in PCI devices and beginning the probe of the PCI bus.

This chapter will use the *i386* subdirectory as an example for how to write a PCI bus layer. This target runs on Intel x86 hardware using a FreeBSD boot-loader that turns on 32-bit addressing/protected mode.

PCI-ISA bridges are detected as PCI devices and are automatically probed as devices. If one is found, the ISA bus methods are installed and accessed through the appropriate PCI calls. Otherwise an ISA bus interface may be explicitly added and probed as described in the ISA chapter.

PCI-PCI bridges are also detected as PCI devices and automatically probed, as are all devices behind them.

Machine-dependent bus interface

The bulk of the work to probe PCI buses and devices is handled in the file *pci.c*, with interfaces to useful routines and structures in *pci.h*. This creates the /pci device node, installs the methods for the bus layer as per the OpenFirmware specification, then probes the bus for all devices connected to it and creates device nodes in the tree for all of them.

The file *i386/pcibase.c* defines the machine-dependent interface for the PCI bus. This is where PCI configuration space, memory space, and I/O space are accessed, mapped, allocated, and freed. All these functions must be defined for a target system.

There are useful macros in pci.h to transform a PCI bus, device, function, etc address into its constituent parts and back again to a 32-bit number.

pci_num_host_bridges

This must return the number of host PCI bridges in the system. There is no way to determine how many PCI buses are built into a system, so this must be specified explicitly. This does not affect PCI-PCI bridges, which are detected and probed as a normal PCI device.

pci_config_read pci_config_write

These routines read and write a value from a specified address in PCI configuration space. For an x86, this is accessed through special registers in I/O space using I/O instructions (defined below). Other systems will simply memory-map configuration space at some known address.

pci_intr_ack pci_special_cycle

These are special PCI words to implement the PCI Forth words "intr-ack" and "special-!". They are otherwise unused by pci.c.

pci_mem_read pci_mem_write

These routines read and write a 1, 2, or 4-byte value from PCI memory space. This is usually the same as the system's memory space but may not necessarily be so for 64-bit systems. Alignment issues must be handled for systems incapable of byte or short access.

pci_mem_read64 pci_mem_write64

These routines read and write a 1, 2, or 4-byte value from 64-bit PCI memory space. They should behave the same as pci_mem_read and pci_mem_write for 32-bit addresses. The x86 does not support 64-bit addressing.

pci_io_read pci_io_write

These perform 1, 2, and 4-byte reads and writes to and from PCI I/O space. For an x86, this is the same as its normal I/O space using its special

I/O instructions. Most other systems will either map I/O space into memory or provide special registers to peform I/O bus cycles.

pci_map_in pci_map_out

These routines map a specified PCI address into and out of the system's memory space to allow subsequent memory-mapped access to the PCI address. For an x86, system memory space is the same as PCI memory space, so the pointer returned is always the same.

pci_dma_alloc pci_dma_free

These routines allocate and free a piece of memory suitable for DMA from a PCI device. All memory is accessible from the PCI world on the x86, so malloc is a good way to allocate memory. The memory block must be aligned for the DMA_PAGE_SIZE, and so to free the block, a list of globally allocated blocks must be maintained to remember the original pointer allocated by malloc keyed by the virtual aligned address. Only a system memory address is returned by these routines - it must still be mapped to a PCI device address below.

pci_dma_map_in pci_dma_map_out

These routines convert a system memory address for a DMA block to an address suitable for a PCI device to access. The memory must have been allocated by pci_dma_alloc above.

pci_dma_sync

This routine must flush the specified DMA memory address out of cache to synchronize a subsequent DMA operation. Otherwise the cached contents will not match the actual data copied in using DMA. For the x86, a special instruction simply flushes the whole cache.

pci_init_addresses

This is called by the code in pci.c to initialize a structure with the memory requirements and specifications for each PCI bus. The contents of the

struct are defined in pci.h. All fields must be initialized to appropriate values to handle legacy ISA I/O spaces, memory ranges, and so on. Unsupported fields should be simply set to zero, as for the 64-bit entries on an x86.

pci_bus_package

This is called by pci.c to return a pointer to a Package for a given host bridge number. For a system with only one host bridge, e->currpkg is sufficient. Other systems will need to keep a global table indexed by the host bridge number.

Driver interface

Since PCI defines how to probe for devices, a built-in C PCI device driver doesn't have to worry about it. It has to specify how to identify itself and an install routine that creates the device node and all associated properties.

Expansion ROMs on PCI devices are automatically probed for OpenFirmware Fcode and executed if detected. Drivers in expansion ROMs take precedence over any built-in C drivers. If there are no drivers detected for a device, it is still inserted into the device tree with a generic configuration.

Some example drivers are *pci/pcidisp.c* for generic frame-buffer display cards, *pci/decether.c* for Digital 21X4X* Ethernet chips, *pci/ncrscsi.c* for NCR/Symbios 53c8xx SCSI chips, and *pci/pciisa.c* for various PCI-ISA bridges.

Pci_driver structure

The key code is at the bottom of the various PCI drivers to initialize a structure *Pci_driver* defined in *pci/pci.h*. Here's an entry for an Intel PIXX3 PCI-ISA bridge from *pci/pciisa.c*:

```
/* Intel 82371SB (PIXX3) PCI-ISA bridge */
Pci_driver intel_piix3_driver =
{
  { 0, 0, 0, 0, 0,
       0x060100, 0x8086, 0x7000, 0, 0, 0 },
  { 0, 0, 0, 0, 0,
       0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0, 0, 0 },
```

```
  install_pciisa_driver
};
```

This struct is used to search for a built-in C driver when a new PCI device is detected by *pci/pci.c*. The first entry is for a *pci_device_info* called "match", the second for a "mask", and the last is a pointer to a C function to install the device driver.

The *match* and *mask* fields are complementary.*match* tells pci/pci.c which values are expected, and *mask* tells it which portions of the values are important. A new device's ID fields are or-ed with *mask* and then compared to *match* to identify the device. For the PCI-ISA bridge above, the classcode, vendor ID, and device ID must match exactly to identify this chip.

Install PCI driver

The install function for a driver must be declared using the *PCI* macro defined in *pci/pci.h* as follows:

```
PCI(install_pciisa_driver) { ... }
```

This will pass in a pointer to the current *Environ*, a pointer to the newly created and uninitialized device *Package*, and a pointer to the *Pci_device_info* describing the chip that was just probed and successfully matched with a *Pci_driver* described above.

The first thing this routine should do is call *pci_load_reg_and_name_props* to initialize the "reg" and "name" properties to reasonable values. This may be handled explicitly if desired - *pci_load_reg_and_name_props* is only provided for convenience. It sets "name" to the generic name, if any, and otherwise to the default "pciN,N" form, disables the device, probes all the BARs to initialize the "reg" property, adding in the expansion ROM if any is detected.

Next, the driver should initialize the "device_type" property if appropriate. Then any methods should be created and a copy of the *Pci_device_info* saved if needed by the methods. It is generally easier to save and access this C structure rather than decode the PCI physaddr properties, but it is not required.

The driver methods simply need to perform the appropriate actions for that type of device, such as open/close or read/write. They must call the various pci_dma_alloc/free, pci_dma_map_in/out, pci_dma_sync, and pci_map_in/out as appropriate to correctly translate PCI address and host addresses on any platform. The *pci/decether.c* driver is a good example of how this should work. It also has to be careful with device endianness as PCI devices are little-endian, but the host may not be.

machdep.c interface

Finally, the PCI interface has to be hooked in through the system's *machdep.c*.

All desired built-in PCI drivers must be explicitly specified and linked in by creating a global *pci_drivers* list:

```
extern Pci_driver pci_display_driver;
extern Pci_driver digital_21x4x_driver;
extern Pci_driver ncr_53C8xx_driver;
extern Pci_driver intel_piix3_driver;
...

const Pci_driver *pci_drivers[] =
{
  &pci_display_driver,
  &digital_21x4x_driver,
  &ncr_53C8xx_driver,
  &intel_piix3_driver,
  ...
  NULL
};
```

There may be more than one type of driver if needed, for instance to handle different brands of PCI-ISA bridges or Ethernet devices. These drivers will only be used if no on-board Fcode option ROM is detected on the device.

The *pci_install* function has to be added to the global *install_list*::

```
...
EC(install_pci);
...

const Command install_list[] =
{
  install_root,  /* should be first */
  install_memory,     /* should be second */
  ...
  install_pci,
  ...
  NULL
};
```

This does not probe the PCI bus if it is detected but simply installs the PCI subsystem into the resulting SmartFirmware image. Probing has to be done explicitly in the routine *machine_probe_all*:

```
CC(machine_probe_all)      /* (--) */
{
  Retcode ret;

  ...

  /* point e->currpkg to the PCI bus node */
  PUSH(e, "/pci");
  PUSH(e, 4);
  ret = execute_word(e, "find-device");

  if (ret == NO_ERROR)
       ret = execute_word(e, "probe-pci");

  ...

  return ret;
}
```

If all goes well, there should be a device tree under /pci when SmartFirmware boots.

9. ISA-bus interface

The ISA bus code supports probing and initializing legacy ISA devices. The source files of this layer are located under the *isa* subdirectory.

While most of the code is machine-independent, a certain amount must be customized to communicate with the ISA configuration registers, memory space, and I/O spaces on the target system. The ISA drivers themselves are machine-independent being written to communicate with the ISA bus layer. Finally, the target *machdep.c* file must specify the built-in ISA devices and beginning the probe of the ISA bus.

This chapter will use the *i386* subdirectory as an example for how to write a ISA bus layer. This target runs on Intel x86 hardware using a FreeBSD boot-loader that turns on 32-bit addressing/protected mode.

PCI-ISA bridges are detected as PCI devices and are automatically probed as devices. If one is found, the ISA bus methods are installed and accessed through the appropriate PCI calls as described in the PCI chapter. These PCI methods will override the pre-defined ISA interface and probe for ISA devices.

The biggest assumption in this code is that there is only a single ISA bus in the entire system even though there may be multiple PCI host bridges.

Machine-dependent bus interface

The ISA bus interface is managed through a set of global function pointers defined in *isa/isa.c* and declared in *isa/isa.h*. These function pointers must be initialized to the appropriate routines to access the ISA bus. A PCI-ISA bridge will override these function pointers. They may be explicitly intiialized for a system with only an ISA bus as in the file *i386/isabase.c*.

The various routines in the *isabase.c* file must behave pretty much like the routines of similar names in *pcibase.c*, although being much simpler.

isa_mem_read isa_mem_write

Read and write 1, 2, and 4 byte quantities from/to ISA memory space.

isa_io_read isa_io_write

Read and write 1, 2, and 4 byte quantities from/to ISA I/O space.

isa_map_in isa_map_out

Map an ISA address into and out of system memory space.

isa_dma_alloc isa_dma_free

Allocate and free memory suiltably aligned for DMA from system memory.

isa_dma_map_in isa_dma_map_out

Map a system DMA address into and out of ISA space.

isa_dma_sync

Flush caches so that a DMA may be safely performed.

The function *install_isabase* must explicitly set the global ISA function pointers to the routines defined to perform these actions. *install_isabase* is called from *machdep.c*'s *machine_probe_all*.

Driver interface

Declaring a built-in ISA C driver is more difficult than a PCI driver since every thing about the driver must be explicitly specified. The first step is to fill in an *Isa_device* struct with the appropriate info. This struct is declared in *isa/isa.h*. The *isa/kbd.c* driver looks like this:

```
Isa_device isa_keyboard =
{
  "keyboard",         /* device name */
  "keyboard",         /* device type */
```

```
  ISA_IO_ADDRESS,     /* I/O or memory address? */
  0x60,               /* address - physlo */
  5,              /* number of bytes at address */
  { 1, 0 },       /* IRQ number and type */
  { -1, },        /* DMA info, if any */
  0x0,            /* BIOS ROM address */
  0, NULL,        /* no extra reg props */
  kbd_probe,
  kbd_install,
  kbd_methods
};
```

The isa/isa.c probe code calls all probe function pointer for all built-in ISA devices. If the probe function returns successful, then the install function pointer is called. The methods entry points to the method table for the device.

Both probe and install routines must be declared using the *ISA* macro defined in *isa/isa.h*.

The probe function should take care when probing for itself as it is fairly easy to lock up the ISA bus. Some probe functions may simply assume that a device is present.

The install function should allocate any additional memory it needs, set its dev->self parameter to this data, then call the *new_isa_device* function to create the device. *new_isa_device* uses the *Isa_device* info to create the new device node under the /isa node with the correct name, type, and reg properties, and to create any methods for the device. *new_isa_device* is simply for convenience and is not required.

Multiple devices may the probe, install, and methods tables but simply have different IRQs and ISA addresses. Please see *isa/ns16550.c* for an example of a serial driver for the four tradition ISA ports COM1−COM4.

machdep.c interface

An example for how to link in an ISA bus is in *i386/machdep.c*.

First all built-in ISA drivers must be entered into the global *isa_devices* list:

```
extern Isa_device ns16550_com1;
extern Isa_device ns16550_com2;
extern Isa_device isa_keyboard;
extern Isa_device vga_display;
...

Isa_device *isa_devices[] =
```

```
{
  &ns16550_com1,
  &ns16550_com2,
  &isa_keyboard,
  &vga_display,
  ...
  NULL
};
```

Then the machine_probe_all routine must first install the ISA base routines to initialize the ISA subsystem, then probe for ISA devices. Here is a simplified example for a system with only a single ISA bus:

```
CC(machine_probe_all)      /* (--) */
{
  Package *isa = new_package(e->root);
  install_isabase(e); /* initialize ISA subsystem */
  return install_isa(e, isa);   /* probe for devices */
}
```

The *i386/machdep.c* file has a somewhat more complex machine_probe_all that will first probe for a PCI bus, and if none is found, only then initialize an ISA bus. If a PCI bus is found, it assumes that a PCI-ISA bridge will also be found thus initializing the ISA subsystem and probing for ISA devices.

10. Client interface

The client-interface is defined by IEEE-1275 to allow a client program (presumably loaded by the firmware) to use the firmware as a simple I/O library. This is typically used to access the console device before an operating system's kernel is fully up and running, search for devices, and to aid debugging.

The actual API callback from the client program to the firmware is unfortunately machine-dependent assembler. Different processor bindings define the interface in different ways to handle the vagaries of registers and other such things.

SmartFirmware also provides a more machine-independent C interface designed to be used from a C (or C-compatible) client program. The only difference is that the compiler is used for the low-level machine-dependent assembly interface. The calling conventions and data passed to both APIs are identical.

The SmartFirmware APIs are implemented in the source-files `misc/sfclient.h` and `misc/sfclient.c`, and a demo program is provided in `misc/chello.c`.

IEEE-1275 client API

The SmartFirmware API and the low-level IEEE-1275 assembly API look essentially like this, where g_client_interface is a pointer whose value is determined in some machine-specific manner:

```
extern void (*g_client_interface)(Cell array[]);
```

The argument array of Cells is defined by IEEE-1275 as follows:

`service`	`array[0]`	address of null-terminated string name of service
`N-args`	`array[1]`	number of input arguments to the service
`M-returns`	`array[2]`	number of return values from the service
`arg1..N`	`array[3..N+2]`	input arguments to the service (if N > 0)
`ret1..M`	`array[N+3..N+M+2]`	return values from the service (if M > 0)

The standard client services defined by IEEE-1275 and some extensions below are merely summaries. Detailed arguments and return values are described in the IEEE-1275 standard. It will generally be far easier to use the SmartFirmware C API described below.

`test`	test if the client-service ''name'' exists
`test-method`	test if a method ''name'' exists in a package
`peer`	return next sibling of a package or root if NULL
`child`	return first child of a package
`parent`	return the parent of a package
`instance-to-package`	return the package of the instance
`getproplen`	get the length of a property
`getprop`	get a property's value
`nextprop`	return the name of the next property
`setprop`	set a property's value
`canon`	make an ambiguous device path fully-qualified
`finddevice`	return package of a device if it exists
`instance-to-path`	return a device path from an instance
`instance-to-interposed-path`	return an interposed device path from an instance
`package-to-path`	return the fully-qualified path of a package
`call-method`	call a method of an open instance
`open`	open a device and return an instance
`close`	close a previously opened device instance
`read`	read from an opened device instance
`write`	write to a device instance
`seek`	seek a device instance, if appropriate
`claim`	allocate requested size of memory
`release`	free previously allocated memory
`boot`	exit the client program and reboot

`enter`	enter the command-interpreter (''ok'' prompt)
`exit`	exit the command-interpreter
`chain`	load another client program and launch it
`interpret`	execute arbitrary Forth commands
`set-callback`	set entry points for Forth ''callback'' and ''sync''
`set-symbol-lookup`	set defer words for ''sym>value'' & ''value>sym''
`milliseconds`	return number of milliseconds since bootup

SmartFirmware C API

The file sfclient.c provides a set of convenience routines to make it easier to call client services from a C program. These routines package up their standard C arguments into an array, call the client-interface, then extract the return values into the C return values.

Each standard client-interface call has a C wrapper function that is also documented in sfclient.h, along with the IEEE-1275 standard's section number (also noted below in square-brackets).

The type ''Retcode'' may be one of R_OK, R_ERR, or R_NO_DATA (also defined in sfclient.h) for most typical return values. It may also be some other negative value for a more specific Forth error code. The types ''Instance'' and ''Package'' are opaque handles to IEEE-1275 instances (from opening device nodes) and packages (device nodes). A ''Cell'' is a machine-dependent size of a Forth word, usually at least 32-bits.

C functions

```
Retcode sf_init( Retcode (*client_interface)(Cell array[]) );
```

Initialize the C interface and the console I/O. This must be the first routine called from main().

```
Retcode sf_call_firmware(const char *name, Int nargs, Int nre-
turns,
          Cell args[], Cell returns[]);
```

Low-level routine to call the firmware's client-interface entry-point. This should not be needed for most purposes.

Client interface (6.3.2.1)

These routines are used to test for the existence of client services and package methods:

```
Retcode sf_test(const char *name);
```

Return 0 if service "name" exists, and -1 if it does not.

```
Retcode sf_test_method(Package *pkg, const char *name);
```

Return 0 if method "name" exists in "pkg", and -1 if it does not.

Device tree (6.3.2.2)

These routines are used to walk the device tree:

```
Package *sf_peer(Package *p);
```

Return the next sibling of the package ‘‘p’’, or the root if ‘‘p’’ is NULL.

```
Package *sf_getroot(void);
```

Return the root node of the device tree. Not-standard but more readable than ‘‘sf_peer(NULL)’’.

```
Package *sf_child(Package *p);
```

Return the first child of the Package.

```
Package *sf_parent(Package *p);
```

Return the parent of the package.

```
Package *sf_instance_to_package(Instance *inst);
```

Return the package of the instance.

These are used to read or set the properties of a given device node (Package):

```
int sf_getproplen(Package *pkg, const char *prop);
```

Get the length of a property, 0 if no value, or -1 on any error.

```
int sf_getprop(Package *pkg, const char *propname,
          char *propbuf, int propbuflen);
```

Get a property’s value, returning -1 on error and the actual size of the property loaded into propbuf.

```
int sf_nextprop(Package *pkg, const char *prevprop, char *prop-
buf);
```

Return the property name of the property following "prevprop" of the Package. propbuf must be at least 32 chars big. Returns -1 on error, 0 if no more properties exist after "prevprop", or 1 otherwise.

```
int sf_setprop(Package *pkg, const char *prop, char *value, int
len);
```

Set the property's value, creating the property if it doesn't exist. Return -1 on error or the actual size of the new property.

These are miscellaneous routines to find, access, and extract device paths:

```
int sf_canon(const char *devspec, char *buf, int buflen);
```

Convert a possibly ambiguous device-specifier to a fully-qualified path - return -1 on error or the length of the fully-qualified name.

```
Package *sf_finddevice(const char *devpath);
```

See if a device-path exists. Return the Package of the device if it exists and NULL on any error.

```
int sf_instance_to_path(Instance *inst, char *buf, int buflen);
```

Return the fully qualified device path of the instance or -1 on error.

```
int sf_instance_to_interposed_path(Instance *inst, char *buf, int
buflen);
```

Return the actual interposed device path of the instance or -1 on error. An interposed path includes hidden device nodes that are not normally seen or accessed other than for debugging.

```
int sf_package_to_path(Package *pkg, char *buf, int buflen);
```

Return the fully qualified device path of the package or -1 on error.

```
Retcode sf_call_method(const char *method, Instance *inst,
          Int nargs, Int nreturns, Cell args[], Cell returns[]);
```

Similar to sf_call_firmware() described above, but uses the "call-method" service to invoke a specific method of a device.

Device I/O (6.3.2.3)

These routines open, close, and access device nodes for I/O:

```
Instance *sf_open(const char *devpath);
```

Open a device and return its instance, or NULL on error.

```
void sf_close(Instance *inst);
```

Close a previously opened device instance.

```
int sf_read(Instance *inst, char *buf, int len);
```

Read data (non-blocking) from an opened device. Returns the actual length read, -1 on error, or -2 for no data pending for the non-blocking read.

```
int sf_write(Instance *inst, const char *buf, int len);
```

Write data to an opened device - return the actual length written or -1 on error.

```
Retcode sf_seek(Instance *inst, int poshi, int poslo);
```

Seek device to the location specified by poshi..poslo. Returns -1 on error or if the device cannot seek.

Memory (6.3.2.4)

These words may not function in all implementations but are provided for completeness. They are used to allocate and free virtual memory, if the firmware is running in and supports such a mode:

```
void *sf_claim(void *virt, int size, int align);
```

Allocate requested size of memory, beginning at "virtaddr" if "align" is 0, or at any addr if "align" is any other value ("virt" is ignored). Returns the baseaddr of the memory, or -1 on any error.

```
void sf_release(void *virt, int size);
```

Release memory allocated by "claim".

Control transfer (6.3.2.5)

These words may not function in all implementations. They are intended primarily for debugging:

```
void sf_boot(const char *bootspec);
```

Exit the client program, reset the system, and reboot with the given "bootspec" argument passed to the "boot" command.

```
void sf_enter(void);
```

Enter the command-interpreter ("ok" prompt).

```
void sf_exit(void);
```

Exit the command-interpreter.

```
void sf_chain(void *virt, int size, void *entry, char *argstr,
int arglen);
```

Free ‘‘size’’ bytes at virt, then load another client program at entry, pass argstr/arglen to the new program ,and launch it.

User Interface (6.3.2.6)

These allow running arbitrary Forth code and define callbacks for the firmware to use when looking up symbols in a loaded program image for debugging:

```
Retcode sf_interpret(const char *cmd, Int nargs, Int nreturns,
          Cell args[], Cell returns[]);
```

Similar to sf_call_firmware() above, but uses "interpret" service to execute arbitrary Forth commands.

```
Callback sf_set_callback(Callback newfunc);
```

Define routine for handling "callback" and "sync" Forth words. The old entry point is returned.

```
void sf_set_symbol_lookup(Callback sym_to_value, Callback
value_to_sym);
```

Define Forth "defer" words "sym>value" and "value>sym" to look up symbols in images.

Time (6.3.2.7)

```
unsigned long sf_milliseconds(void);
```

Return number of milliseconds, typically since bootup.

Console I/O

These non-standard routines provide basic console I/O, much like their stdio equivalents:

```
int sf_getchar(void);
```

Read a character from the console (non-blocking). Returns R_NO_DATA or a character.

```
int sf_gets(char *str, int len);
```

Read a line from the console (blocking). Waits until a CR or LF is typed from the console before returning.

```
void sf_putchar(int ch);
```

Write a character to the console.

```
void sf_puts(const char *str);
```

Write a string to the console.

Initialization

Getting the actual pointer to the client-interface is also unfortunately machine-dependent. SmartFirmware also provides a C way to get this pointer by passing it to the client-interface's main() routine as a bogus argument at the end of the argv[] list. This pointer should be passed to the sfclient.c interface as follows:

```
int
main(int argc, const char *argv[], const char *envv[])
{
     sf_init( (void*)argv[argc + 1] );
     ...
}
```

sf_init() also initializes its console I/O variables so that the sf_get*() and sf_put*() routines will work.

Other systems may acquire the client-interface's entrypoint using other methods, such as decoding an entry in the ''env[]'' environment table, or hard-wiring a special address.

Example

An example C program that uses the SmartFirmware client-interface is provided in chello.c. This displays the traditional ''hello world'' message and then walks the OpenFirmware device tree displaying the names of all the network devices it finds.

Appendix A: SmartFirmware User Manual

This document describes the SmartFirmware User Interface, and how to use it to manipulate device trees, view and set parameter values, and to control boot-up procedures.

Basics

SmartFirmware is an implementation of the OpenFirmware IEEE standard 1275-1994 plus errata changes.

The basis of SmartFirmware is a Forth engine. Forth is essentially a stack-based language, much like the well-known Hewlett-Packard calculators. Forth "words" are names that are executed when they are typed in at the "ok" prompt.

For instance, to add two numbers, type "3 4 +. cr" at the prompt. This sequence first pushes the number '3' then the number '4' onto the stack. The '+' then pops and adds the top two stack elements and pushes the result on the stack. The "." pops and displays the sum on the top of the stack, namely the value '7'. The "cr" simply displays a carriage-return and doesn't affect the stack.

This document expands upon the built-in "help" Forth word.

Devices

A device is a node in a tree, much as a file in a Unix file-path. A device name is usually of the form "/name@addr1,addr2:options/dev2/...". All the portions of a device path are machine-dependent. An example of a path is "/duart@C0800A00/uart@1:19200,9600,8,1,N". The "@addr" and ":options" portions of the name may be left out, in which case the first matching name is used ("/duart/uart" for the previous example).

Devices may also have aliases for convenience. For instance, the alias "tty" may refer to "/duart@C0800A00/uart@0", and may then be referred to with "tty:19200,9600,8,1,N" with options if desired.

Displaying

Devices can be listed using the "ls" command from the SmartFirmware User Interface. The "dev" or "cd" commands (which are synonyms) moves around the device tree. The "pwd" command shows the current device being accessed.

The command "show-devs" lists all devices in the device tree. The command ".properties" can be used to display all property lists at any node in the device tree, after using "dev" to first open that device.

pwd

Print current (working) device path.

ls

Show all devices at the current point in the device tree.

dev <device path>

cd <device path>

Change the currently open device to the one specified on the command-line. The path may be relative to the current device. The path may also be the string ".." to open the parent of the current device. The device will also parse any optional arguments to configure the device, if any are provided.

Example: cd /duart@C0800A00:9600

show-devs

Show all the devices in the device tree.

.properties

Show all the properties of the current device.

words

Show all the methods of the current device, or all the Forth words if there is no current device.

devalias [name string]

Show all the device aliases currently defined if nothing is on the command line. Otherwise, create a device alias called "name" with the value of "string". This is handy to avoid typing in long path names again and again. Useful in "nvramrc" scripts (below).

Example: devalias tty /duart@C0800A00/uart@0

Parameter settings

Parameter settings are stored in non-volatile memory (NVRAM). They affect a variety of SmartFirmware activities from bootup to running custom scripts to setting the size of the display window.

Parameters are simply string value pairs that usually have a default value (stored in ROM) and an active value (stored in NVRAM). They may be displayed, set, and restored using the following commands.

The supported parameter variables are described in the next section.

The following commands are used to manipulate NVRAM parameter settings:

printenv [var]

Display all parameter variables, their current values, and their default values if no "var" is specified on the command-line. Otherwise just display the values for that variable.

Example: printenv diag-switch?

setenv var string value

Set the value of "var" to be "string value". Everything up to the end of the line, including spaces, will be stored in NVRAM.

Example: setenv diag-switch? true

set-default var

Set the value of "var" to be its default value. This is handy to restore some variable whose value may have gotten scrambled for some reason.

set-defaults

Set all parameter values back to their default values. This is handy if NVRAM has become corrupted.

nvedit

Edit the script stored in parameter "nvramrc". This opens up a very simple Emacs-style text editor with the contents of the "nvramrc" variable.

Type ^C (Control-C) to exit. Most of the usual Emacs commands may be used to move the cursor around, including ^F (forward-char), ^B (backward-char), ^N (next-line), and ^P (previous-line).

This command does not alter the contents of "nvramrc" but merely edits a copy in memory. See "nvstore", "nvquit", "nvrecover", and "nvrun" (below).

nvstore

Stores the contents of the current "nvedit" buffer into the parameter variable "nvramrc". See "nvedit" (above).

nvquit

Erase the contents of the nvedit buffer without storing it. Prompts the user to confirm before erasing the buffer. Does not alter the contents of "nvramrc". See "nvedit" (above).

nvrecover

Attempts to recover the contents of nvedit after it has been erased. Normally "nvstore" is supposed to erase the contents of the edit buffer. SmartFirmware does not do this, so nvrecover is effectively non-functional. See "nvedit" and "nvstore" (above).

nvrun

Execute the contents of the nvedit buffer as Forth code. Handy way to make sure that the "nvramrc" script works before committing it to NVRAM. See "nvedit" (above).

nvalias alias device

Add a "devalias" to the "nvramrc" script to create this alias at bootup. If there is already a "devalias" command there, edit it to point to this new device. Finally, run the new "devalias" command so the alias is available for immediate use. This is a short-cut operation for most typical uses of "nvramrc".

Example: nvalias tty /duart@C0800A00/uart@0

nvunalias alias

Remove the specified "devalias" command for creating this alias at bootup from the "nvramrc" script, if there is one there.

Parameter variables

Parameter variables are manipulated by the commands described in the previous section. Most are used during bootup and are guarded by boolean parameters to turn features on or off. All these variables may not be present on all systems. For instance, if the system does not support booting, none of the boot parameter vars would be of much use.

diag-switch?

Boolean: "true" or "false". Switch on diagnostic-mode if true, which turns on extended tests, more verbose output, and alters the boot commands below.

secondary-diag?

Boolean: "true" or "false". Switch secondary-diagnostics on or off at bootup. This is not a standard OpenFirmware parameter and controls additional comprehensive test routines for SmartFirmware.

auto-boot?

Boolean: "true" or "false". If true, execute the word in "boot-command" after the standard bootup process. Otherwise run the Forth interpreter on the console and display an "ok" prompt.

auto-boot-timeout

Integer: number of milliseconds, usually 500. This is the amount of time to wait (in milliseconds) for a key-press to abort auto-boot. This is not a standard OpenFirmware parameter.

boot-command

String: usually "boot". The command to execute to boot the system if the variable "auto-boot?" is true.

boot-device

String: usually "disk". The device to use to load the boot image when "boot" is executed. The string is usually an alias to the actual device.

boot-file

String: usually blank. The file to load from the "boot-device" when "boot" is executed.

diag-device

diag-file

Like "boot-device" and "boot-file", but are used if "diag-switch" is true when "boot" is executed. Typical values are "net" and "diag" respectively, where "net" is typically an alias to a network device.

use-nvramrc?

Boolean: "true" or "false". If true, execute the contents of "nvramrc" at bootup.

nvramrc

The script to execute at bootup. This script is only executed if "use-nvramrc?" is set to true (below).

The normal bootup sequence is: "probe-all install-console banner". If an "nvramrc" script is provided, it should perform the above three in the same order to make sure that the device tree is probed and ready. If these commands are not executed in the "nvramrc" script, then they will be executed by SmartFirmware once the script finishes running.

input-device

String: usually "keyboard". The device path to use for the console input. The device "keyboard" is usually an alias to the actual input device determined in some machine-dependent fashion.

output-device

String: usually "screen". The device to use for console output. Again, this is usually an alias to the actual device.

screen-#rows

screen-#columns

Integer: usually 0. The number of rows and columns desired for the console output. If zero, the largest allowable number will be used depending upon the font used.

inverse-video

Boolean: "true" or "false". Display text on the console as black-on-white if this parameter is true, else as white-on-black. This is not a standard OpenFirmware parameter.

oem-banner?

Boolean: "true" or "false". If true, display the contents of "oem-banner" when the command "banner" is executed in place of the default banner string.

oem-banner

String to display when the "banner" command is executed, if "oem-banner?" is set to true.

oem-logo?

Boolean: "true" or "false". If true, display the bitmap in "oem-logo" in front of the banner when the command "banner" is executed. Otherwise a default logo (or no logo) will be displayed.

oem-logo

Bitmap: 64x64x1 (512 bytes). The bitmap to display if "oem-logo?" is true. The contents may be machine-dependent.

Booting

The bootup process is as follows, as required by the OpenFirmware standard. The process is largely machine-dependent but the steps are the same for all platforms:

1. Power-on self-test
2. System initialization
3. Evaluate the script "nvramrc" if "use-nvramrc?" is true.
4. If the script was not executed or if there was no console after the script finished executing, then:
    1. Execute "probe-all" (evaluates FCode)
    2. Execute "install-console"
    3. Execute "banner"
5. Secondary diagnostics, if "secondary-diag?" is true, and other system-dependent initialization.
6. Default boot if "auto-boot?" is true and no key is held down.
7. Run the Forth command interpreter (if not booted)

Commands for booting include:

boot [device] [args]

Boot the specified device with arguments, if any. If args is not specified, use the contents of the parameter "boot-file" or "diag-file" depending on

the current setting of "diag-switch?". If device is not specified, use "boot-device" or "diag-device" instead. "boot" is the same as a "load" followed by a "go" (below).

Example: boot /flash pterm

load [device] [args]

As boot above, except do not actually boot, but do everything else to prepare an image for booting. Run the "go" command to actually boot.

go

Boot the image that was prepared with "load" above.

probe-all

Probes the system for all devices and builds the device tree. Used in the "nvramrc" script. Should only be called once at bootup.

install-console

Selects and installs a console from the device tree, typically a screen and keyboard or a serial-port. Used in the "nvramrc" script. Should only be called once at bootup.

banner

Display the bootup system banner, optionally with a custom color logo. This is usually called in the "nvramrc" script at bootup, but is safe to execute multiple times.

Net-booting

Booting over a network device is probably the most useful feature of SmartFirmware. SmartFirmware supports the standard UDP/IP protocols of BOOTP, DHCP, or RARP for configuring a network device, and TFTP for loading an images over that network device.

The commands ‘‘load’’ and ‘‘boot’’ (described above) when used with a network device accept the following optional device-specific arguments as follows:

`net:[dopt,][prto,]sia,fname,cia,gia,bret,tret [args]`

‘‘net’’ may be a device alias or a device-path to an ethernet device.

''dopt'' are device-specific options. SmartFirmware's builtin ethernet drivers support the strings ''promiscuous'', ''speed=10'', ''speed=100'', ''duplex=full'', and ''duplex=half'', assuming the device supports the specific capability. The options may be in any order and must be separated by commas. Additional options may also be supported by a specific driver.

''prto'' is optional and specifies which network boot protocol to be used. SmartFirmware supports the strings ''bootp'', ''dhcp'' or ''rarp'' to force BOOTP, DHCP, or RARP protocols to be used. BOOTP is the default on most platforms (although Sparcs may use RARP).

''sia'' is the server internet address in dotted octet or hexadecimal format of the server that will be used for TFTP.

''fname'' is the name of the file to be downloaded over the network using TFTP.

''cia'' is the internet address that SmartFirmware will use for its local (client) interface address.

''giadr' is the internet address of the gateway between the local interface and the server if such a gateway is present.

''bret'' and ''tret'' are decimal numbers and indicate the number of times that retries will be sent before the BOOTP or TFTP operations fail.

''args'' are the optional arguments passed to the program when it is executed.

All device arguments are optional, but the commas separating the arguments are required. Any missing fields are filled in by the information returned using BOOTP/DHCP/RARP protocols. Thus the simplest way to boot over a network device is simply ''boot net''. ''boot net:,file'' boots a specific file over the net.

Testing

It is possible to exercise some of the test methods from the Forth command prompt. If the parameter variable "diag-switch?" is true, then more verbose and more thorough tests are turned on. It may be useful to run the non-verbose commands at bootup, if desired. The commands are as follows.

test [device]

Test the specified device (or the currently open device if none is specified) by executing its "selftest" method. This may also turn on LEDs or other indicators that testing is in progress.

test-all [device]

Run all "selftest" methods recursively at and below the specified device (or "/" if no device is specified). This runs all tests on the requested node, then on all its children, and their children, and so on.Typically used to run all the tests on all devices from "/".

Changing the console

The console device used may be changed at any time, or selected during bootup. To change it during bootup, the typical method is to either change the NVRAM parameters "input-device" and "output-device" to the desired console device, or add "devalias" commands to "nvramrc" to change the aliases for "screen" and "keyboard" to the desired console.

Otherwise, the console may be changed on-the-fly using the following Forth words. Note that they expect their parameters on the Forth stack and not on the command-line as do most of the commands above.

input (device-str str-len --)

Used by first pushing a string containing the desired device onto the Forth stack, then executing the word "input". The console input will be immediately changed unless an error occurs. Example: " /duart/uart@0" input

output (device-str str-len --)

Changes the current console output device.

io (device-str str-len --)

Changes both the input and the output device at one time. Only useful if the device is capable of both input and output.

Forth variables

Forth variables may be viewed and set using standard Forth words. Some additional variables have been added for SmartFirmware that are not part of the OpenFirmware standard.

To view a Forth variable called, say, "lines/page", use "." to display the value on the top of the stack and "cr" to print a carriage-return:

```
ok lines/page . cr
```

To set a Forth variable, the new value must be first pushed onto the Forth stack before using the "to" word:

```
ok -1 to lines/page
```

lines/page

The current number of lines per page specified for automatic pagination of the output. If the value is zero, no pagination is performed. If the value is negative, it is subtracted from the number of lines in the display. Otherwise the value specifies how many lines are to be displayed before a "More" prompt temporarily stops the output.

scroll-step

The number of lines to scrol l the display by when at the bottom of the screen. This can speed up slower frame buffers by scrolling by 4 lines instead of 1, for instance.