Introduction

Editorial change: correct typo and credit secretary properly.

Firmware is the read-only-memory (ROM)-based software that controls a computer between the time it is turned on and the time the primary operating system takes control of the machine. The responsibilities of firmware include testing and initializing the hardware, determining the hardware configuration, loading (or booting) the operating system, and providing interactive debugging facilities in case of faulty hardware or software.

Historically, firmware design has been proprietary and often specific to a particular bus or instruction set architecture (ISA). This need not be the case. Firmware can be designed to be machine-independent and easily portable to different hardware. There is a strong analogy with operating systems in this respect. Prior to the advent of the portable UNIX® operating system in the mid-seventies, the prevailing wisdom was that operating systems must be heavily tuned to a particular computer system design and thus effectively proprietary to the vendor of that system.

IEEE Std 1275-1994 (Open Firmware) is based on Sun Microsystems® OpenBoot™ firmware. The OpenBoot design effort began in 1988, when Sun was building computers based on three different processor families. Thus, OpenBoot was designed from the outset to be ISA-independent. The first version of OpenBoot was introduced on Sun's SPARCstation™ 1 computers. Based on experience with those machines, OpenBoot version 2 was developed and was first shipped on SPARCstation 2 computers. This standard is based on OpenBoot version 2.

Open Firmware has the following features:

— A mechanism for loading and executing programs (such as operating systems) from disks, tapes, network interfaces, and other devices.
— An ISA-independent method for identifying devices “plugged-in” to expansion buses and for providing firmware and diagnostics drivers for these devices.
An extensible and programmable command language based on the Forth programming language.

Methods for managing user-configurable options stored in non-volatile memory.

A “call back” interface allowing other programs to make use of Open Firmware services.

Debugging tools for hardware, firmware, firmware drivers, and system software.

The following individuals were members of the P1275 Working Group at the time IEEE Std 1275-1994 was produced:

William M. (Mitch) Bradley, Chair
David M. Kahn, Vice Chair
John Rible, Draft Editor
Luan D. Nguyen, Mike Williams, Secretary

Paul Fischer Thanos Mentzelopoulos Paul Thomas
Ron Hochsprung David Paktor Mike Tuciarone
Michael Saari Mike Williams
The following definitions give the meanings of the technical terms as they are used in this standard. Terms defined herein are italicized upon their first occurrence in each subclause throughout the rest of the document. Terms related to the Forth programming language are defined in ANSI X3.215-1994.¹

2.3 Definitions of terms

Editorial change: correct typos and references. (Only the changed terms are included.)

The following definitions give the meanings of the technical terms as they are used in this standard. Terms defined herein are italicized upon their first occurrence in each subclause throughout the rest of the document. Terms related to the Forth programming language are defined in ANSI X3.215-1994.¹

2.3.1 active package: The package, if any, whose methods are accessible by name to the command interpreter, and to which newly created methods and properties are added.

2.3.7 cell: The primary unit of information in the architecture of a Forth System. See 2.3.2, ANSI X3.215-1994.

2.3.75 printable character: A character in the range 0x21 through 0x7E or the range 0xA1 through 0xFE (see 2.3.3). See ISO 8859-1: 1987.

3.6.4 Expansion bus device class template

Editorial change: improve a possibly misleading short description.

Editorial change: add a section to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics and explain why the user interface versions of those words are being removed.

A memory-mapped bus logically extends the processor’s memory address space to include the devices on that bus, allowing the use of processor load and store cycles to directly address those devices. The details vary from bus to bus. This standard does not specify the adaptation of Open Firmware to any particular bus, but other related standards do so specify (see 2.1).

3.6.4.1 Bus-specific methods and properties

This subclause lists a set of methods that deal with requirements common to most memory-mapped buses. This subclause is intended as a suggested starting point for the development of complete sets of methods for particular buses; also see related standards, such as IEEE Std 1275.2-1994. The methods provide mapping services for establishing the correspondence between processor virtual and device physical addresses, allocation of DMA memory, and probing to locate plug-in devices.

map-in ( phys.lo ... phys.hi size -- virt ) Map the specified region; return a virtual address.
map-out ( virt size -- ) Destroy mapping from previous map-in.
dma-alloc (... size -- virt ) Allocate a memory region for later use.
dma-free ( virt size -- ) Free memory allocated with dma-alloc.
dma-map-in (... virt size cacheable? -- devaddr ) Convert virtual address to device bus DMA address.
dma-map-out ( virt devaddr size -- ) Free DMA mapping set up with dma-map-in.
dma-sync ( virt devaddr size -- ) Synchronize (flush) DMA memory caches.
probe-self ( arg-str arg-len reg-str reg-len fcode-str fcode-len -- ) Evaluate FCode as a child of this node.

The following properties are specific to this class of device node:

¹Information on references can be found in 2.1.
“ranges” Standard property name to define a bus-specific address translation device’s physical address.

“#address-cells” Standard property name to define the package’s address format.

“#size-cells” Standard property name to define the package’s address size format.

3.6.4.2 Bus-specific register access words

Some expansion bus adaptors have characteristics that interfere with the semantics of the register-access words. This standard does not specify the adaptation of Open Firmware to any particular MMU. This standard does not require the presence of an MMU.

Such bus-specific register-access words can be written in terms of the “generic” register-access words, in the expectation that the parent bus will have substituted implementations of those words that handle any peculiar characteristics imposed by the parent bus. For example, suppose a bus adapter device supports a bus that reverses the order of doublets within quadlets. It cannot predict the characteristics of its parent bus, for example, whether its parent bus reverses the order of bytes within a doublet. The FCode program for such a bus adapter would include a definition of a bus-specific quadlet register-read word that might look something like this:

```plaintext
: my-r1@ ( qaddr -- quad ) r1@ lwflip ;
```

When the FCode for this word’s definition is evaluated, the definition of r1@ that will get compiled-in will be the one that was substituted by the node of the parent bus. The node of the present bus adapter would then, before evaluating its children’s FCode programs, have to perform a substitution that might look something like this:

```plaintext
[ ' ] my-r1@ h# 234 set-token
```

Because such a substitution takes place at the time of evaluation of an FCode program, it becomes problematical to ensure that the semantics of the intended register-access words will be visible at the user interface level. For this reason, it is recommended that device nodes for child-devices supply definitions for the register-access words that will bind the semantics that were substituted for the device’s bus to the user interface name. Such a definition can be very simple, and might look something like this:

```plaintext
: r1@ ( qaddr -- quad ) r1@ ;
```

The net effect of such a definition would be that when the device is selected, and the user enters the name r1@, the user interface interpreter will find the name that occurs in the device’s node; that name would be bound to the behavior that was installed in the FCode interpretation token-table at the time the device node’s FCode was evaluated, causing the correct behavior to be executed.

3.6.5 Memory management device class template

An MMU is a device that performs address translation between a CPU’s virtual addresses and the physical addresses of some bus, typically the bus represented by the root node. In general, the details are both processor-specific and bus-specific. This standard does not specify the adaptation of Open Firmware to any particular MMU, but other related standards may so specify (see 2.1). This standard does not require the presence of an MMU.

This subclause lists a set of methods that deal with requirements common to most MMUs. This subclause is intended as a suggested starting point for the development of complete sets of methods for particular MMUs. The methods provide services for fine-grained control of the allocation and mapping of virtual addresses, particularly intended for use by client programs through the call-method client interface service (see also the “mmu” property of the /chosen node). In general, the use of these methods makes a client program system-specific;
nevertheless, they are useful in some circumstances. The arguments and results shown are intended as guidelines; particular MMUs might require additional arguments or changes to the arguments shown.

The presence of an MMU node does not imply that the Open Firmware is necessarily using virtual-to-physical address translation hardware.

The following methods, defined in the glossary, are recommended for MMU packages:

```plaintext
claim  [ virt ] len size-align -- base ) Allocate (claim) addressable resource
release ( virt len size-- ) Free (release) addressable resource.
map ( phys.lo ... phys.hi virt len size-mode ___ ) Create address translation
unmap ( virt len size-- ) Invalidate existing address translation.
modify ( virt len size-mode ___ ) Modify existing address translation.
translate ( virt -- false | phys.lo ... phys.hi mode ... true ) Translate virtual address to physical address.
```

Additional requirements for the `claim` and `release` methods:
— The address format, `virt`, is a single-cell virtual address.
— The allocation length, `len size`, is a single cell.
— The allocated resource is a region of virtual address space.

The following properties are recommended for MMU packages:

<table>
<thead>
<tr>
<th>Property</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>available</code></td>
<td>The property values are as defined for the standard “reg” format, with single-cell virtual addresses. The regions of virtual address space denote the virtual address space that is currently unallocated by the Open Firmware and is available for use by client programs.</td>
</tr>
<tr>
<td><code>existing</code></td>
<td>The value of this property defines the regions of virtual address space managed by the MMU in whose package this property is defined without regard to whether or not these regions are currently in use. The encodings of <code>virt</code> and <code>len</code> are MMU-specific.</td>
</tr>
<tr>
<td><code>page-size</code></td>
<td>The value of this property is the number of bytes in the smallest mappable region of virtual address space.</td>
</tr>
</tbody>
</table>

**NOTE**—Freeing virtual address space does not necessarily free any associated physical resource. The correct sequence of operations for freeing mapped memory is to first use `unmap`, thus destroying the translation. Then the physical memory and virtual address space can be freed with the `release` methods of the respective nodes.

### 3.8.3 “deblocker” support package

Normative change: add two methods to satisfy the requirements of existing deblocker support package implementations. IEEE Std 1275-1994 neglected to document this requirement.

Normative change: add an explicit clarification of a previously unstated requirement.

The “deblocker” package assists in the implementation of byte-oriented `read` and `write` methods for block-oriented or record-oriented devices such as disks and tapes. It provides a layer of buffering to implement a high-level byte-oriented interface “on top of” a low-level block-oriented interface. The “deblocker” support package defines the following methods:

```plaintext
open ( -- okay? ) Prepare this device for subsequent use.
close ( -- ) Close this previously opened device.
read ( addr len -- actual ) Read device into memory buffer; return actual byte count.
write ( addr len -- actual ) Write memory buffer to device; return actual byte count.
seek ( pos.lo pos.hi -- status ) Set device position for next `read` or `write`.
```

Any package that uses the “deblocker” support package must define the following methods, which the deblocker uses as its low-level interface to the device.
block-size ( -- block-len )  Return “granularity” for accesses to this device.
max-transfer ( -- max-len )  Return size of largest possible transfer.
read-blocks  ( addr block# #blocks -- #read )  Read #blocks, starting at block#, from device into memory.
write-blocks  ( addr block# #blocks -- #written )  Write #blocks from memory into device, starting at block#.
dma-alloc ( size -- virt )  Allocate a memory region for later use.
dma-free ( virt size -- )  Free memory allocated with dma-alloc.

NOTE—Although, in general, methods for some busses have optional parameters, the method for physical addresses that
use the deblocker package shall not require optional parameters.

4.3 Path resolution

Normative change: correct an error in the path resolution procedure that occurs when there is a device alias “a:
whose expansion is “/b:” IEEE Std 1275-1994 translated “a:0” to “/b:0” instead of the correct result “/b:?”

This section defines the process of resolving a device path given by a device-specifier. There are three contexts in
which this can occur:

— find-device. In this context, the intention is to locate the named device node and select it as the active
package without any other side effects.

— open-dev. In this context, the intention is to open every node named in the path by executing its open
method, thereby creating an instance chain, and to return the ihandle of the node at the tail end of the chain
(the node farthest from the root node).

— execute-device-method. In this context, the intention is to open every node named in the path except
for the last node. An instance chain is created, including an instance for the last node, but instead of executing
that last node’s open method, a different method, given as an argument, is executed. Then the open instances
are closed and the instance chain is destroyed.

The overall structure of the path resolution process is the same in all three contexts. This description shows it as
one process with conditional tests at places where the details are context-dependent. However, it need not be
implemented that way; for example, each context could be implemented separately.

The process is described in English as a set of procedures, each consisting of steps that are generally executed in
order, with the scope of conditional tests shown by indentation and looping structures shown by labels and “go to”
lines. It makes liberal use of variable names to identify intermediate data items. The scope of such variables is
“global” with respect to the procedures. The use of these variable names does not imply that an implementation
must or should use such variables; they are used solely for descriptive purposes. Similarly, the description of the
process in terms of procedures does not imply that the implementation should be so structured; the separate
procedures were used in the description so that the top-level description would not be unwieldy.

The following notation describes the parsing of pathnames into component parts:

left-split(string, “x”) -> initial, remainder

String, initial, and remainder are the names of string variables, and “x” is a character.

Left-split divides string into two disjoint substrings, setting initial to the portion of string before the first
occurrence of the character “x”, and remainder to the portion of string following the first occurrence of the
character “x”. Neither initial nor remainder contains that first occurrence of “x”, although remainder may contain
other later occurrences of that character. If string does not contain the character “x”, initial is set to string in its
entirety, and remainder is set to the empty string.

right-split(string, “x”) -> initial, remainder
Right split is similar to left split, except that the division of the string occurs around the last occurrence of the
character “x”, rather than the first.

split-before(string, “x”) -> initial, remainder

Split-before is similar to left-split, except that if “x” occurs in string, remainder begins with the first occurrence of
“x” (left-split removes the first occurrence of “x” from remainder)

split-after(string, “x”) -> initial, remainder

Split-after is similar to split-before, except that the division of the string occurs after the last occurrence of the
character “x”. If “x” occurs in string, initial ends with “x”, and remainder begins with the first character, if any,
following that last occurrence of “x”. If “x” does not occur in string, initial is set to string and remainder is set to
the empty string.

The use of the preceding notation does not necessarily imply the existence of functions named left-split,
split-before, and split-after, and right-split, it is simply a notational convention. (This standard does define a
function left-parse-string whose semantics are very similar to left-split, but the details of returning the
results are somewhat different.)

In searching for a matching node, the order in which the various child nodes are considered is unspecified. At the
implementation’s discretion, if no match is found among the children, the search may be widened to include the
children’s children, recursively to any depth.

In the following algorithmic description (4.3.1 through 4.3.5), the text enclosed in boxes is commentary describing
the intention of the algorithm. The text outside of the boxes is definitive.

4.3.1 Path resolution procedure (top level procedure)

If the pathname does not begin with “/”, and its first node name component is an alias, replace the alias with its
expansion.

a) If PATH_NAME does not begin with the “/” character,

1) Left-split Split-before(PATH_NAME, “/”) -> HEAD, TAIL.

2) Left-split Split-before(HEAD, “.”) -> ALIAS_NAME, ALIAS_ARGS.

3) If ALIAS_NAME matches a defined alias,

i) Replace ALIAS_NAME with its alias value.

ii) If ALIAS_ARGS is not empty:

   a) Right-split Split-after(ALIAS_NAME, “/”) -> ALIAS_HEAD, ALIAS_TAIL.

   b) Right-split Split-before(ALIAS_TAIL, “.”) -> ALIAS_TAIL, DEAD_ARGS.

   c) If ALIAS_HEAD is not empty,

      Concatenate(ALIAS_HEAD, “/”, ALIAS_TAIL, ALIAS_ARGS) -> ALIAS_TAIL_NAME.

   d) Concatenate(ALIAS_TAIL, “/”, ALIAS_ARGS) -> ALIAS_NAME.
iii) If TAIL is empty, replace PATH_NAME with ALIAS_NAME.

iv) Otherwise (when TAIL is not empty),
    Concatenate(ALIAS_NAME, $^{<}$T$^{>}$TAIL) -> PATH_NAME.

If the pathname, after possible alias expansion, begins with “/”, begin the search at the root node. Otherwise, begin at the active package.

b) Otherwise (when the (possibly expanded) PATH_NAME begins with the “/” character),
   1) Remove the “/” from PATH_NAME.
   2) Set the active package to the root node.

c) If there is no active package, exit this procedure, returning false.

If the pathname, after possible alias expansion, begins with “/”, begin the search at the root node. Otherwise, begin at the active package.

<<<The remainder of clause 4.3 is unchanged.>>>
The markers \( B \wedge \) and \( T \wedge \) show the “branch” and “target” locations used for the calculation of the value of \( FCode-offset \). The value is the signed number of FCode bytes between \( B \) and \( T \) (positive if \( B \) is before \( T \)), \( Bn/Tn/H/T1 \) are for refer to the corresponding \( FCode-offset1 \) and \( B2/T2 \) are for \( FCode-offset2 \).

NOTE—On some devices, FCode programs are stored with “gaps” between successive FCode bytes. For example, each FCode byte might be stored in the least significant byte of a separate quadlet, in which case it might be necessary to add four to the address to advance to the next FCode byte. This does not affect the calculation of an \( FCode-offset \)—the offset is in terms of the number of FCode bytes, independent of how those bytes are addressed.

The offset size (whether of 8 bits or 16 bits) is established at the beginning of the FCode program by the particular start code that begins the FCode program. \texttt{version1} sets the offset size to 8 bits, and the other start codes (\texttt{start0}, \texttt{start1}, \texttt{start2}, and \texttt{start4}) set the offset size to 16 bits. The offset size may be changed from 8 bits to 16 bits by executing \texttt{offset16}.

In most cases (the exceptions are \texttt{bbranch} and \texttt{b?branch} in interpretation state), the FCode evaluator needs only the sign of the offset, not its numerical value. In these cases, the value of the offset is essentially redundant because control transfers are represented by pairs of FCode functions (a branching function and its target). The offset indicates the distance between the branch and its target, but that information can be derived during the FCode evaluation process without needing the offset value. However, standard FCode programs are required to have numerically correct offsets (as described in the above paragraph) for compatibility with existing practice.

### 6.3.2.4 Memory

Editorial change: correct misspelled name.

\[ \text{IN: } [\text{address}] \text{ virt, size, align} \]
\[ \text{OUT: } [\text{address}] \text{ baseaddr} \]

Allocates size bytes of memory. If \( \text{align} \) is zero, the allocated range begins at the virtual address \( \text{virt} \). Otherwise, an aligned address is automatically chosen and the input argument \( \text{virt} \) is ignored. The \( \text{alignment} \) boundary is the smallest power of two greater than or equal to the value of \( \text{align} \); an \( \text{align} \) value of 1 signifies 1-byte alignment. \( \text{Baseaddr} \) is the beginning address of the allocated memory (equal to \( \text{virt} \) if \( \text{align} \) was 0) or \(-1\) if the operation fails (for example, if the requested virtual address is unavailable).

The range of physical memory and virtual addresses affected by this operation will be unavailable for subsequent mapping or allocation operations until freed by \( \text{release} \).
release
IN: [address] virt, size
OUT: none
Frees size bytes of physical memory starting at virtual address virt, making that physical memory and the corresponding range of virtual address space available for later use. That memory must have been previously allocated by claim.

6.3.2.5 Control transfer
Normative change: explicitly describe the behavior of the “enter” and “exit” client services for the case where the Open Firmware user interface does not exist.

boot
IN: [string] bootspec
OUT: none
Exits the client program, resets the system (as with the command reset-all), and reboots the system with the device and arguments given by the null-terminated string bootspec. The string bootspec is interpreted in the same manner as the arguments of the command boot.

enter
IN: none
OUT: none
If an IEEE std 1275 user interface exists, exit the client program and enter the Open Firmware command interpreter (e.g., called by the operating system after a console input device abort). The client program may be resumed if the user continues execution with the go command. Otherwise, if another user interface exists, transfer control to that interface. When no user interface exists, return control to the client program.

exit
IN: none
OUT: none
If an IEEE std 1275 user interface exists, exit from the client program and enter the Open Firmware command interpreter. The execution of the client program may not be resumed. Otherwise, if another user interface exists, transfer control to that interface. When no user interface exists, idle.

chain
IN: [address] virt, size, [address] entry, [address] args, len
OUT: none
Frees size bytes of memory starting at virtual address virt, then executes another client program beginning at address entry. The argument buffer args, len is copied into the Open Firmware memory and passed to the other program. The address of the arguments in the Open Firmware memory is the client program’s second argument, and their length is its third argument. chain is used to free any remaining memory for a secondary boot program and begin executing the booted program.

NOTE—The behavior of the chain client interface service includes the functions of init-program and go on behalf of the new client program, but does not include the functions of reading the client program into memory, parsing its header, or allocating its memory.
6.3.2.6 User interface

Normative change: define the format of arguments whose existence was implied in IEEE Std 1275-1994 but which were unintentionally not specified.

interpret

IN:  [string] cmd, stack-arg1, ..., stack-argP
OUT: catch-result, stack-result1, ..., stack-resultQ

Pushes one less than N-args items, stack-arg1, ..., stack-argP, onto the Forth data stack, with stack-arg1 on top of the stack; executes the null-terminated string cmd as a Forth command line guarded by catch.

Pops the result returned by catch into catch-result. If that result is nonzero, restore the depth of the Forth data stack to its depth prior to the execution of interpret. If that result is zero, pops up to one less than N-returns items, stack-result1, ..., stack-resultQ, from the Forth data stack into the returned values portion of the argument array, with stack-result1 corresponding to the top of the stack.

N-args and N-returns are stored in the argument array and may be different for different calls to interpret. If the number of items X left on the Forth data stack as a result of the execution of cmd is less than N-returns, only stack-result1, ..., stack-resultX are modified; other elements of the returned values portion of the argument array are unaffected. If X is more than N-returns, additional items are popped from the Forth data stack after setting stack-result1, ..., stack-resultQ so that, in all cases, the execution of interpret results in no net change to the depth of the Forth data stack.

An implementation shall allow at least six stack-arg and six stack-result items. interpret is optional; it need be present only if the Open Firmware user interface is present.

set-callback

IN:  [address] newfunc
OUT: [address] oldfunc

Client programs may define a routine for handling the Open Firmware routines $callback$ and $sync$. $Newfunc$ is the address of the entry point of the callback routine. This service sets the callback handler to $newfunc$ and returns as $oldfunc$ the address of the entry point of the previously installed callback handler.

The Open Firmware shall use the same calling conventions specified in 6.3.1 for client interface services when calling the callback handler function. See $callback$ and $callback$ glossary entries for details.

A client program callback handler shall return either a nonzero error code in the ret1 cell of the argument array if the service indicated by the service argument is unavailable, or zero otherwise. The client program callback handler shall return any additional results in the ret2 ... retN cells, setting N-returns to the total number of return values including the error code (or zero) that is in the ret1 cell. The handler shall not store more than M results, where M is the value that was in the N-returns cell when the handler was called, nor shall the returned value of N-returns exceed M.

set-symbol-lookup

IN:  [address] sym-to-value, [address] value-to-sym
OUT: none

Sets the symbol table resolution defer words sym>value and value>sym so that they execute the client program callbacks whose addresses are given by the arguments sym-to-value and value-to-sym, respectively. If either argument is zero, the corresponding defer word is set to the action of false.

sym-to-value is called as follows:

IN:  [string] symname
OUT: error, symvalue

The service name string in the argument array is a pointer to a null-terminated string containing “sym-to-value”.

Searches for a symbol whose name is symname. If such a symbol is found, returns zero in error and the symbol’s value in symvalue. If no such symbol is found, returns −1 in error and zero in symvalue.
value-to-sym is called as follows:

IN:  symvalue
OUT:  offset, [string] symname

The service name string in the argument array is a pointer to a null-terminated string containing “value-to-sym”.

Locates the symbol whose value is closest to but not greater than symvalue and returns offset, the non-negative offset from the value of that symbol to symvalue, and symname, the symbol name.

If symvalue is less than the value of any known symbol, or is insufficiently close to any symbol value according to an implementation-dependent criterion, returns −1 in offset and the empty string in symname.

set-symbol-lookup is optional; it need be present only if the Open Firmware user interface is present and the Client Program Debugging command group (see 7.6) is implemented.

Annex A. Open Firmware glossary

Changes to glossary definitions are described for each definition.

Editorial change: use more precise language to describe the bit significance and intended requirements.

bljoin  ( b1.lo b2 b3 b4.hi -- quad ) F 0x7F

Join four bytes to form a quadlet.

The high bits of each of the four bytes must be zero.

Combine the eight least significant bits of each operand to form a quadlet. Other operand bits shall be zero.

b<mark>  (F: -- ) F 0xB1

Target of backward branch implemented with bbranch or b?branch.

FCode evaluation:  (F: -- )

Perform the interpretation or compilation semantics of begin.

FCode ONLY (Tokenized by begin)

Editorial change: clarify the language of the definition.

Normative change: correct an error for the case where the else clause of the Fcode equivalent of an if ... else
then construct is executed in Fcode interpretation state.

b>resolve  ( -- ) F 0xB2

Target of forward branch implemented with bbranch or b?branch.

FCode evaluation:  (F: -- )

If in interpretation state:

Do nothing

If in compilation state:

Perform the compilation semantics of then. Then, if the current definition is temporary and the depth of the control flow stack is the same as its depth when the temporary current definition was initiated, perform the FCode evaluation semantics of b(; ) and execute the temporary current definition.

FCode ONLY (Tokenized by else, then, and repeat)
bwjoin ( b.lo b.hi -- w ) F 0xB0

Join two bytes to form a doublet.

The high bits of each of the two bytes must be zero.

Combine the eight least significant bits of each operand to form a doublet. Other operand bits shall be zero.

--

claim ( [ addr ... ] len size ... align -- baseaddr ... ) M

Allocate (claim) addressable resource.

Allocates len size ... (whose format depends on the package) bytes of the addressable resource managed by the package containing this method. If align is zero, the allocated range begins at the address addr ... (whose format depends on the package). Otherwise, addr ... is not present, and an aligned address is automatically chosen. The alignment boundary is the smallest power of two greater than or equal to the value of align; an align value of 1 signifies one-byte alignment.

Baseaddr ... (whose format is the same as addr ...) is the address that was allocated (equal to addr ... if align was 0).

If the operation fails, uses throw to signal the error.

Claim does not automatically create an address translation for the allocated resource. See 3.6.5.

NOTE—This method provides fine-grained control over the allocation of addressable resources. In general, such control is needed only by system-specific programs. General-purpose memory allocation can be accomplished in a portable fashion by alloc-mem.

See also: alloc-mem, "available", free-mem, release.

encode-int ( quad a -- prop-addr 4 prop-len ) F 0x111

Encode a number into a prop-encoded-array.

The property encoding of a (quadlet) number is a sequence of 4 bytes, with the most significant byte first (i.e., at the smallest address).

No alignment is implied; the sequence of 4 bytes begins at the first available location.

Used as: 5000 encode-int ( prop-addr 4 prop-len )

find-package ( name-str name-len -- false | phandle true ) F 0x204

Locate the support package named by name string.

If the package can be located, return its phandle and true; otherwise, return false.

Interpret the name in name string relative to the “/packages” device node. If there are multiple packages with the same name (within the “/packages” node), return the phandle for the most recently created one.
Editorial change: clarify the description of xt.

Normative change: add explicit error-handling requirement to the display driver’s open routines and add a required return value. (Historical implementations had such a return value, but IEEE Std 1275-1994 neglected to mention it.)

```
is-install ( xt -- ) F 0x11C
```

Create open, other methods for this display device.

Create methods for accessing the display device driver in the active package. xt is the execution token of a routine to initialize the display device, whose stack diagram is (--).

```
Used as: ['] my-open-routine is-install
```

Create the following methods:

```
open
   When later called, execute the display driver’s “my-open-routine” (whose execution token is xt) guarded by catch.
   If the execution of xt results in a throw, return false. Otherwise, and initialize the terminal emulator and return true.
```

```
write
   When later called, pass its argument string to the terminal emulator for interpretation.
```

```
draw-logo
   When later called, execute the display driver’s “my-draw-logo” procedure which was installed into the defer word draw-logo by the driver’s “my-open-routine”.
```

```
restore
   When later called, execute the display driver’s “my-reset-screen” procedure which was installed into the defer word reset-screen by the driver’s “my-open-routine”.
```

```
lbsplit ( quad -- b1.o b2 b3 b4.hi ) F 0x7E
```

Split a quadlet into 4 bytes.

The high bits of the 4 bytes are zero.

All but the least significant eight bits of each result value shall be zero.

```
lwsplit ( quad -- w1.o w2.hi ) F 0x7C
```

Split a quadlet into two doublets.

The high bits of the two doublets are zero.

All but the least significant sixteen bits of each result value shall be zero.
NOTE—In general, -2 indicates no data was available at the time read was done and -1 indicates that an error occurred.

Zero is generally used only for devices where data arrives in records, packets, or other such container, and indicates that a valid but empty container was received.

Additional requirements for the write method:

- Transmit the network packet of length len bytes from the memory buffer beginning at addr. Return the number of bytes actually transmitted.
- The packet to be transmitted begins with an IEEE 802 Media Access Control (MAC) header.
- Usage restriction: The caller must supply the complete header; the source hardware address will not necessarily be “automatically inserted” into the outgoing packet.

Additional requirements for the load method:

- Read the default client program into memory, starting at addr, using the default network booting protocol.

A standard package with this “device_type” property value may implement additional device-specific methods.

A standard package with this “device_type” property value shall implement the following property if the associated device has a preassigned IEEE 802.3-style MAC (network) address:

"local-mac-address"

NOTE—Such packages often use the “obp-tftp” support package to implement the “load” method.

See also: “address-bits”, “max-frame-size”
Editorial change: clarify the behavior of \texttt{next-property} when various unusual circumstances are encountered.

\texttt{next-property} \hspace{1em} ( previous-str \ previous-len \ phandle \ -- \ false \ | \ name-str \ name-len \ true ) \ F \ 0x23D

Return the \textit{name} of the property following \texttt{previous} of \texttt{phandle}.

\textit{Name} is a null-terminated string that is the \textit{name} of the property following \textit{previous} in the property list for device \texttt{phandle}. If \textit{previous} is zero or points to a zero-length string, \textit{name} is the \textit{name} of the \texttt{phandle}’s first property. If there are no more properties after \texttt{previous} or if \texttt{previous} is invalid (i.e., \textit{name} of a property that does not exist in \texttt{phandle}), \textit{name} is a pointer to a zero-length string.

Locate, within the property list of the \texttt{package} identified by \texttt{phandle}, the first property if \textit{previous} is zero, or the property following the property named by the text string \textit{previous} \ \textit{previous-len} otherwise. Return \textit{name-str} \ \textit{name-len} and \texttt{true} if such a property exists, or \texttt{false} otherwise (i.e., if there are no more properties, or if there is no property in that package with the \textit{name} given by \texttt{property-str} \ \texttt{property-len}).

A sequence of invocations of \texttt{next-property} with the same \texttt{phandle} value, beginning with \textit{previous-len} equal to zero, and passing the \textit{name-str} \ \textit{name-len} result of the previous invocation as the \textit{previous} \ \textit{previous-len} argument to the next invocation, continuing until \texttt{false} is returned, shall enumerate the \textit{name}s of all properties of that \texttt{package}. The order in which those individual \textit{properties} are returned is undefined (e.g. the first property returned by \texttt{next-property} is not necessarily the one that was created first). If a new \textit{property} is created within that \texttt{package} between invocations of \texttt{next-property} in such a sequence, the new \textit{property} name may, but need not, be returned as a result of one of the following invocations of \texttt{next-property} within that same sequence.

Normative change: add a method that additional experience with IEEE Std 1275-1994 has shown to be required.

\texttt{page-size} \ S

\texttt{prop-encoded-array}:

Integer encoded as with \texttt{encode-int}.

The value of this \textit{property} is the number of bytes in the smallest mappable region of virtual address space.

Editorial change: improve a possibly misleading short description.

\texttt{ranges} \ S

Standard \textit{property} \textit{name} to define a bus-specific address translation \texttt{device}’s physical \texttt{address}.

\texttt{rb}! (FCode function) \hspace{1em} ( byte \ \textit{addr} -- ) \ F \ 0x231

Store a byte to device register at \texttt{addr}.

Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.

Register is stored with identical bit ordering as the input stack item.

\textbf{NOTE}—A bus \texttt{device} can substitute (as with \texttt{set-token}) a bus-specific implementation of \texttt{rb}! for use by its \texttt{children}. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.
Normative change: remove a user interface word that in some situations cannot be implemented properly. Implementations may continue to support it, but its presence is no longer required.

```
rb1 (user interface) (byte addr --)
```

Store a byte to device register at addr.

```
Compilation: __________________________ ( )
Interpretation: ________________________ (byte addr --)
```

Perform the equivalent of the phrase:

```
h# 231 get-token if execute else compile, then
```

```
rh! 231 get-token drop execute
```

**NOTE**—A bus device can substitute (see **set-token** ) a bus-specific implementation of `rb1` for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of `rb1` ensure that such substitutions are visible at the user interface level.

Editorial change: add a note to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics.

```
rb@ (FCode function) (addr -- byte) F 0x230
```

Fetch a byte from device register at addr.

Data is read with a single access operation.

Result has identical bit ordering as the original register data.

**NOTE**—A bus device can substitute (as with **set-token**) a bus-specific implementation of `rb@` for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.

Normative change: remove a user interface word that in some situations cannot be implemented properly. Implementations may continue to support it, but its presence is no longer required.

```
rh0 (user interface) (addr -- byte)
```

Fetch a byte from device register at addr.

```
Compilation: __________________________ ( )
Interpretation: ________________________ (addr -- byte)
```

Perform the equivalent of the phrase:

```
h# 230 get-token if execute else compile, then
```

```
rh0 230 get-token drop execute
```

**NOTE**—A bus device can substitute (see **set-token**) a bus-specific implementation of `rh0` for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of `rh0` ensure that such substitutions are visible at the user interface level.

Editorial change: add a note to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics.

```
rl! (FCode function) (quad qaddr --) F 0x235
```

Store a quadlet to device register at qaddr.

Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.

Register is stored with identical bit ordering as the input stack item.

**NOTE**—A bus device can substitute (as with **set-token**) a bus-specific implementation of `rl!` for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.
A bus device can substitute (see for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of r11 ensure that such substitutions are visible at the user interface level.

Editorial change: add a note to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics.

r1@ (FCode function) ( qaddr -- quad )

Fetch a quadlet from device register at qaddr.

Data is read with a single access operation.

Result has identical bit ordering as the original register data.

NOTE—A bus device can substitute (as with set-token) a bus-specific implementation of r1@ for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.

r1@ (user interface) ( qaddr -- quad )

Perform the equivalent of the phrase:

h# 235 get-token if execute else compile, then

Perform the equivalent of the phrase:

h# 235 get-token drop execute

NOTE—A bus device can substitute (see set-token) a bus-specific implementation of r1@ for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of r1@ ensure that such substitutions are visible at the user interface level.

Editorial change: add a note to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics.

rw! (FCode function) ( w waddr -- )

Store a doublet w to device register at waddr.

Data is stored with a single access operation and flushes any intervening write buffers, so that the data reaches its final destination before the next FCode function is executed.

Register is stored with identical bit ordering as the input stack item.

NOTE—A bus device can substitute (as with set-token) a bus-specific implementation of rw! for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.
A bus device can substitute (see set-token) a bus specific implementation of \texttt{rw} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of \texttt{rw} ensure that such substitutions are visible at the user interface level.

Editorial change: add a note to describe the recommended technique for using existing standard words to handle bus-dependent register access semantics.

**rw@** (FCode function) \( \texttt{rw@ \ (waddr -- w)} \) F 0x232

Fetch a doublet \( w \) from device register at \( waddr \).

Data is read with a single access operation.

Result has identical bit ordering as the original register data.

**NOTE**—A bus device can substitute (as with set-token) a bus specific implementation of \texttt{rw@} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. See clause 3.6.4.2 for further details and for user interface semantics.

Normative change: remove a user interface word that in some situations cannot be implemented properly. Implementations may continue to support it, but its presence is no longer required.

**rw@** (user interface) \( \texttt{rw@ \ (waddr -- w)} \)

Fetch a doublet \( w \) from device register at \( waddr \).

Perform the equivalent of the phrase:

\[ \texttt{h} \ 232 \ 	exttt{get-token} \ 	exttt{if} \ 	exttt{execute} \ 	exttt{else} \ 	exttt{compile}, \texttt{then} \ 	exttt{get-token} \ 	exttt{drop} \ 	exttt{execute} \]

**NOTE**—A bus device can substitute (see set-token) a bus specific implementation of \texttt{rw@} for use by its children. This is sometimes necessary to correctly implement its semantics with respect to bit-order and write-buffer flushing. The given user interface semantics of \texttt{rw@} ensure that such substitutions are visible at the user interface level.

**translate** (virt -- false | phys.lo ... phys.hi mode ... true) M

Translate virtual address to physical address.

If a valid virtual to physical translation exists for the virtual address \( \texttt{virt} \), return the physical address \( \texttt{phys.lo} \ ... \ \texttt{phys.hi} \), the translation mode \( \texttt{mode} \), and \( \texttt{true} \). Otherwise return \( \texttt{false} \). The physical address format is the same as that of the "memory" node (the node whose \texttt{handle} is given by the value of \texttt{chosen’s memory} property). The interpretation of \( \texttt{mode} \) is MMU implementation dependent.
**Annex C (informative). The tokenizer**

Normative change to informative annex: add tokenizer commands to standardize the beginning and end of tokenizer source code.

**C.3.3 Fcode start and end**

These commands create an FCode header for the Open Firmware source between them.

<table>
<thead>
<tr>
<th>Command</th>
<th>Arguments</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fcode-version2</code></td>
<td>( -- )</td>
<td>T</td>
<td>Begin tokenizing an FCode program.</td>
</tr>
<tr>
<td><code>fcode-end</code></td>
<td>( -- )</td>
<td>T</td>
<td>Finish tokenizing an FCode program.</td>
</tr>
</tbody>
</table>

**Annex E (informative). SCSI host adapter package class**

**E.6.1 overall.fth**

Normative changes to informative annex: fix bugs in the tokenizer source code.

FCode driver for hypothetical SCSI host adapter

`fcode-version2`
`hex`
CORE REQUIREMENTS AND PRACTICES

"XYZI,scsi" name "Name of device node
"XYZI,12346-01" model "Manufacturer's model number
"scsi-2" device-type "Device implements SCSI-2 method set
3 0 intr "Device interrupts on level 3, no vector

external

\ These routines may be called by the children of this device.
\ This card has no local buffer memory for the SCSI device, so it
\ depends on its parent to supply DMA memory. For a device with
\ local buffer memory, these routines would probably allocate from
\ that local memory.

: dma-alloc ( n -- vaddr ) " dma-alloc" $call-parent ;
: dma-free ( vaddr n -- ) " dma-free" $call-parent ;
: dma-sync ( vaddr devaddr n -- ) " dma-sync" $call-parent ;
: dma-map-in ( vaddr n cache? -- devaddr ) " dma-map-in" $call-parent ;
: dma-map-out ( vaddr devaddr n -- ) " dma-map-out" $call-parent ;
: max-transfer ( -- n )
  " max-transfer" ['] $call-parent catch if 2drop h# 7fff.ffff then
\ The device imposes no size limitations of its own; if it did, those
\ limitations could be described here, perhaps by executing:
\ my-max-transfer min
;

fload scsiha.fth
fload hacom.fth

new-device
  fload scsidisk.fth \ scsidisk.fth also loads scsicom.fth
finish-device

new-device
  fload scsitape.fth \ scsitape.fth also loads scsicom.fth
finish-device

fcode-end

E.6.2 scsiha.fth

Normative changes to informative annex: fix bugs in the tokenizer source code.

<<<the code in E.6.2 prior to open-hardware is unchanged>>>>

: open-hardware ( -- okay? )
  map
  7 to my-id
  \ Should perform a quick "sanity check" selftest here,
  \ returning true if the test succeeds.
  true
; : reopen-hardware ( -- okay? ) true ;
: close-hardware ( -- ) unmap ;
: reclose-hardware ( -- ) ;
: selftest ( -- 0 | error-code )
  \ Perform reasonably extensive selftest here, displaying
  \ a message and returning an error code if the
  \ test fails and returning 0 if the test succeeds.
IEEE
Std 1275-1994
IEEE STANDARD FOR BOOT (INITIALIZATION CONFIGURATION) FIRMWARE:

0
;
: set-address ( unit target -- )
  to his-id to his-lun
;

E.6.3  hacom.fth

Normative changes to informative annex: fix bugs in the tokenizer source code.

<<<the code in E.6.3 prior to diagnose is unchanged>>>

external

: diagnose ( -- error? )
  0 0 true test-unit-rdy-cmd 6 -1 ( dma$ dir cmd$ #retries )
  retry-command if
    ( [ sensebuf ] hardware-error? )
    if
      " Test unit ready failed - "
    else
      " hardware error (no such device?)" cr
    endif
  else
    " extended status = " cr
  endif
  base @ r> hex
  8 bounds ?d o 1 3 u.r loop cr
  r> base !
  then
  true
  else
    send-diagnostic ( fail? )
  then

headers

E.6.4  scsicom.fth

Normative changes to informative annex: remove unnecessary code that refers to a draft version of IEEE Std 1275-1994.

<<<the code in E.6.4 prior to self-test is unchanged>>>

headerless

: selftest ( -- error? )
  "code-revision h# 3.0000 >= if
  " my-unit " set-address" $call-parent
  " diagnose" $call-parent
  else
  0
  then

headers

E.6.5  scsidisk.fth

Normative changes to informative annex: fix bugs in the tokenizer source code.

<<<the code in E.6.5 prior to r/w-blocks is unchanged>>>

: r/w-blocks ( addr block# #blks input? command -- actual# )
  cmdbuf d# 10 erase
  2 pick h# 100000 u> 2over h# 100 u>
  swap h# 200000 u> or if \ Use 10-byte form ( addr block# #blks dir cmd )

224444
h# 20 or 0 cb! \ 28 (read) or 2a (write) ( addr block# #blks dir )
-rot swap ( addr dir #blks block# )
cmdbuf 2 + 4c! ( addr dir #blks )
dup cmdbuf 7 + 2c! ( addr dir #blks cmd-len )
d# 10 ( addr dir #blks cmd-len )
else \ Use 6-byte form ( addr block# #blks dir cmd )
  0 cb! ( addr block# #blks dir )
-rot swap ( addr dir #blks block# )
cmdbuf l+ 3c! ( addr dir #blks )
dup 4 cb! ( addr dir #blks )
  6 ( addr dir #blks cmd-len )
then
  tuck >r >r ( addr input? #blks ) ( R: #blks cmd-len )
  /block * swap cmdbuf r> -1 ( addr #bytes input? cmd cmd-len #retries )
  retry-command if ( [ sensebuf ] hw? )
  0= if drop then r> drop 0 else ( )
  r>
  then ( actual# )
;
the remainder of the code in E.6.5 is unchanged
Annex G (informative). Summary lists

G.2 Assigned FCode numbers

Editorial change: correct typo in the specified table entry.

Annex H (informative). Historical notes

H.4 New FCodes and methods

Editorial change: correct typo.

Most pre-Open Firmware systems do not implement the following FCodes and methods:

<table>
<thead>
<tr>
<th>FCode#</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x023E</td>
<td>byte-load</td>
<td>On pre-Open Firmware, &quot;byte-load&quot; $find could be used.</td>
</tr>
<tr>
<td>0x023F</td>
<td>set-args</td>
<td>On pre-Open Firmware, &quot;set-args&quot; $find could be used.</td>
</tr>
<tr>
<td>0x025D</td>
<td>next-property</td>
<td></td>
</tr>
</tbody>
</table>

4.6 New user interface commands

Editorial change: remove commands that existed prior to Open Firmware from the list.

Most pre-Open Firmware systems do not implement the following user interface commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>Execute named method in the specified package.</td>
</tr>
<tr>
<td>char</td>
<td>Generate numeric code for next character from input buffer.</td>
</tr>
<tr>
<td>close-dev</td>
<td>Close device and all of its parents.</td>
</tr>
<tr>
<td>$create</td>
<td>Call create; new name specified by name string.</td>
</tr>
<tr>
<td>environment?</td>
<td>Return system information based on input keyword.</td>
</tr>
<tr>
<td>fm/mod</td>
<td>Divide d by n.</td>
</tr>
<tr>
<td>noshowstack</td>
<td>Turn off showstack (automatic stack display).</td>
</tr>
<tr>
<td>parse</td>
<td>Parse text from the input buffer, delimited by delim.</td>
</tr>
<tr>
<td>parse-word</td>
<td>Parse text from the input buffer, delimited by white space.</td>
</tr>
</tbody>
</table>
### FCode name changes


The following FCodes names have changed from their pre-Open Firmware versions for clarity and consistency. While this can affect the tokenizer and/or user interface behavior, the actual behavior of the function associated with that FCode number has not changed. Existing (already-tokenized) FCode programs that use these FCodes will be unaffected.

Items marked with a * have retained the old name, as a synonym.

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>u#</td>
<td>#</td>
<td>u#</td>
</tr>
<tr>
<td>#&gt;</td>
<td>u#&gt;</td>
<td>#</td>
<td>u#</td>
</tr>
<tr>
<td>s</td>
<td>lshift *</td>
<td>s</td>
<td>rshift *</td>
</tr>
<tr>
<td>attribute</td>
<td>property</td>
<td>/c*</td>
<td>chars *</td>
</tr>
<tr>
<td>b(is)</td>
<td>b(to)</td>
<td>decode-2int</td>
<td>parse-2int</td>
</tr>
<tr>
<td>delete-attribute</td>
<td>delete-property</td>
<td>eval</td>
<td>evaluate *</td>
</tr>
<tr>
<td>flip</td>
<td>wflip</td>
<td>version</td>
<td>fcode-revision</td>
</tr>
<tr>
<td>get-inherited-attribute</td>
<td>get-inherited-property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get-my-attribute</td>
<td>get-my-property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get-package-attribute</td>
<td>get-package-property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/n*</td>
<td>cells *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>invert *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u*x</td>
<td>um*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>version</td>
<td>fcode-revision</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### User interface name changes

Editorial change: remove names from the list which existed only in some implementations and whose behavior is different than the indicated new name.

The following user interface command names have changed from their pre-Open Firmware versions, with no change in behavior.

<table>
<thead>
<tr>
<th>Old name</th>
<th>New name</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>properties</td>
</tr>
<tr>
<td>cd</td>
<td>dev</td>
</tr>
<tr>
<td>reset</td>
<td>reset-all</td>
</tr>
<tr>
<td>select-dev</td>
<td>open-dev</td>
</tr>
<tr>
<td>unsselect-dev</td>
<td>device-end</td>
</tr>
</tbody>
</table>