NICHEFILE

VIRTUAL FILE SYSTEM (VFS)
TECHNICAL REFERENCE

51 E Campbell Ave
Suite 160
Campbell, CA. 95008

email: support@iniche.com
http://www.iniche.com
support: 408.540.1160
fax: 408.540.1161

InterNiche Technologies Inc. has made every effort to assure the accuracy of the information contained in this documentation. We appreciate any feedback you may have for improvements. Please send your comments to **support@iniche.com**.

Rev-10.2005

Table of Contents

1. OVERVIEW OF API PROVIDED BY THE VFS

The VFS exports an API that is approximately in conformance to a subset of the ISO 9899: 1990 ("ISO C") buffered file I/O API that is characterized by the functions **fopen()**, **fclose()**, **fread()**, **fwrite()**, etc. The functions which constitute the VFS API are listed below:

vfopen
vfclose
vfread
vfwrite
vfseek
vftell
vgetc
vferror
vunlink
vclearerr

The reader that is familiar with the standard C library buffered I/O functions will note that the VFS API differs from the standard in that in the VFS the standard function names are all prefixed with a '**v**' and the type used to identify a stream handle in the standard API, **FILE**, is instead named **VFILE**. This is done so that the VFS can coexist with a standard C library that provides implementation of the standard API without creating naming conflicts. The calling syntax and semantics of a given VFS function is approximately the same as those of the correspondingly named standard C library function. There are small differences between the VFS API and the standard which are described later in this document.

1.1 VFS Implementation

The VFS is implemented as a flat (non-hierarchical) file system in which the set of files that exist in the file system and those files' contents are stored in target system memory. The set of files is implemented as a singly linked list of structures in which each structure has associated with it a buffer that is used to contain the associated file's contents. Part of this list can be contained as part of the target system executable. The InterNiche Web Server and HTML Compiler use this feature to link the files that contain Web server content with the target system executable. The set of files contained in the list and their contents can be modified at run time to allow this set of files to be updated dynamically.

The VFS includes the concept of a target system dependent backing store that can be used to allow these dynamically created files to be stored to whatever non-volatile storage (typically solid state devices like FLASH EEPROM) that is provided by the target system. The VFS reads the backing store during system initialization in order to reconstruct the file system in memory. Applications can then open, read, write, and close files in the VFS using the VFS API. Data that is read from a **VFILE** is read from normal read/write system memory, like SRAM or DRAM. Data that is written to a **VFILE** is written to normal read/write memory.

Typically (though not necessarily, this behavior is configurable by the porting engineer), the entire subset of the VFS that has been marked as non-volatile is written from the volatile system memory (e. g. SRAM or DRAM) to the non-volatile backing store whenever any file to

which modifications have been made is closed. This behavior allows for simple implementations of backing store drivers for devices like FLASH which by their nature can make it complicated to implement random access writes.

1.2 Source Files that Constitute the VFS

1.2.1 vfsfiles.h

vfsfiles.h should be included by source files that intend to use the VFS API. It contains the definitions of data structures used by the VFS, prototypes of the functions that constitute the VFS API and various defined constants.

1.2.2 vfsport.h

vfsport.h is intended to be a placeholder into which the porting engineer can add structure definitions, function prototypes and defined constants that are particular to a given target system.

1.2.3 vfsfiles.c

vfsfiles.c contains the bulk of the implementation of the VFS API.

1.2.4 vfsutil.c

vfsutil.c contains functions which implement a user interface that allows access to and control of the VFS.

1.2.5 vfssync.c

vfssync.c contains the implementations of functions which write to and read from the VFS backing store .

1.2.6 makefile

makefile contains make utility rules for compiling the source files that constitute the VFS and producing a resultant **vfs.lib** library file that can be linked with the other object modules to produce a target system executable.

1.3 VFS Configuration Options

The VFS provides several configuration options to allow the porting engineer to customize the VFS behavior for a particular target system. The options are described below.

1.3.1 VFS_FILES

The presence of the defined constant **VFS_FILES** enables the VFS described in this document. If **VFS_FILES** is not defined, the inclusion of **vfsfiles.h** causes the VFS API entry points to be defined to be equal to their standard C library equivalents, as in:

```
#define vfopen(n,m)     fopen(n,m)
#define vfclose(fd)     close(fd)
```

InterNiche applications which need access to a file system, like the Web and FTP Servers, perform file system access via the VFS API. By undefining **VFS_FILES**, the porting engineer can cause these applications to access the standard C library buffered I/O API that is provided by their target system's compiler package.

Example usage:

```
#define VFS_FILES 1
```

1.3.2 HT_RWVFS

HT_RWVFS defines whether the VFS is write enabled. The presence of this defined constant causes code that enables write access to the VFS to be included in the target system executable. If **HT_RWVFS** is not defined, files can be opened, read from, and closed via the VFS, but calls which would cause the VFS to be modified, such as **vfwrite()** will not be operational.

Example usage:

```
#define HT_RWVFS 1
```

1.3.3 VFS_AUTO_SYNC

The presence of the defined constant **VFS_AUTO_SYNC** causes the VFS to call the function **vfs_sync()** to cause the memory resident VFS to be written to the backing store anytime a modified VFS file is closed or a VFS file is deleted. If **VFS_AUTO_SYNC** is not defined, these automatic calls to **vfs_sync()** are not made. In cases like this, **vfs_sync()** can be called via a user interface command or by some other porting engineer provided method.

Example usage:

```
#define VFS_AUTO_SYNC 1
```

1.3.4 HT_EXTDEV

The presence of the defined constant **HT_EXTDEV** causes the VFS to make calls to "external file systems". External file systems are described in more detail in the section "External File Systems" starting on page 25.

1.3.5 HT_LOCALFS

The presence of the defined constant **HT_LOCALFS** causes the VFS API functions to make calls to their analogous standard C library buffered I/O function under certain circumstances. This is described in the section "Local File Systems" starting on page 24.

1.3.6 FILENAMEMAX

The defined constant **FILENAMEMAX** defines the maximum length of a VFS file name. The default value of **FILENAMEMAX** is 16, but the porting engineer can modify this value if 16 characters is not an appropriate length for file names on the target system.

Example usage:

```
#define FILENAMEMAX 50
```

1.3.7 VFS_MAX_TOTAL_RW_SPACE

The defined constant **VFS_MAX_TOTAL_RW_SPACE** defines an upper limit on the amount of memory that the VFS will allocate for use in buffers for the containment of VFS file contents. This allows the porting engineer to limit the amount target system memory that the VFS will consume.

Example usage:

```
#define VFS_MAX_TOTAL_RW_SPACE          100000
```

With the above definition, attempts to write to a **VFILE** which requires more than 100 kilobytes of system memory to be allocated to contain the file contents will fail.

1.3.8 VFS_MAX_DYNA_FILES

The defined constant **VFS_MAX_DYNA_FILES** defines an upper limit on the number of files that the VFS will create dynamically. As with **VFS_MAX_TOTAL_RW_SPACE**, it is a tool that the porting engineer can use to limit the amount of memory that is consumed by the VFS.

Example usage:

```
#define VFS_MAX_DYNA_FILES 100
```

With the above definition, attempts to create more than 100 files on the target system will fail.

1.3.9 VFS_MAX_OPEN_FILES

The defined constant **VFS_MAX_OPEN_FILES** defines an upper limit on the number of files that the VFS will allow to be simultaneously open. It is another tool to limit VFS memory usage.

Example usage:

```
#define VFS_MAX_OPEN_FILES 5
```

With the above definition if five files have been opened and not closed, the next attempt to open a **VFILE** will fail.

2. DETAILED DESCRIPTION OF VFS API

In the following API description, the term "current file pointer" or **CFP** means the relative byte offset from the beginning of the file from which reads will be made and to which writes will be made.

2.1 vfopen

```
VFILE* vfopen(char * name, char * mode);
```

The calling semantics of **vfopen()** are similar to that of the standard C library **fopen()**. The **name** parameter points to a null terminated string that defines the name of the file to be opened. The first character of the string addressed by the **mode** parameter defines what actions are to be taken when opening the file, as shown below:

```
mode[0] == 'r'
```

If the named file does not exist, fail the open. If the file does exist, open the file and position the **CFP** to the beginning of the file.

```
mode[0] == 'w'
```

If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, truncate it to a length of 0 and open it. In both cases position the **CFP** to the beginning of the file.

```
mode[0] == 'a'
```

If the named file does not exist, create a file of 0 length with the given name and open it. If the named file does exist, open it without modifying its existing contents. In both cases position the **CFP** to the end of the file.

Returns

When **vfopen()** is successful, it returns a handle which is a pointer to the type **VFILE**. This handle should be passed to subsequent VFS functions which require a **VFILE** parameter to access the file's contents. When **vfopen()** is not successful it returns NULL. The reason for the error can be retrieved by calling the function **get_vfopen_error()**. The values returned by **get_vfopen_error()** come from the set of **ENP_** errors that are described in the InterNiche TCP/IP Technical Reference.

2.1.1 Differences from the Standard fopen() Call

Only the first character of the **mode** parameter is significant. The '**b**' and '**+**' suffixes that have special meaning in some **fopen()** implementations have no meaning to **vfopen()**. This means that the "open for read access only" semantic of the '**r**' parameter that is present in **fopen()** does not apply. Writes to a file that is **vfopen()**'ed with **mode** '**r**' will not automatically fail like they do on some systems. In that sense '**r**' with **vfopen()** is more like

'**r+**' on most system's **fopen()**. It also means that the '**ascii**' mode of file opening in which newline conversion is done in the API is not performed with the VFS. All reads and writes are strictly binary.

The VFS supports only one current file pointer per **VFILE**. Some buffered I/O systems will do reads from the "current file pointer" which is settable with **fseek()** but will only allow writes to the end of the file (as weird a "standard" behavior as one can imagine). With the VFS, **read**s and **write**s are always initiated from the **CFP**.

The VFS imposes no requirements on file names other than that they are not to exceed **FILENAMEMAX** characters in length. Embedded spaces and punctuation characters are legal, as are ASCII characters with the most significant bit set. A file name of 0 length is legal. Slash (forward slash), '**/**', and backslash, '****', have no special meaning. The one exception to this is that if a file name begins with a slash, '**/**', it will be removed from the file name before the file is created. Thus the file names **/foo** and **foo** refer to the same file.

2.2 vfclose

```
void vfclose(VFILE * vfd);
```

Files that are opened with **vfopen()** should eventually be closed with **vfclose()**. Depending on how the VFS has been configured and whether any changes to the file have been made since it was opened, a call to **vfclose()** can cause the function **vfs_sync()** to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

2.3 vfread

```
int vfread(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

The calling semantics of **vfread()** are similar to that of the standard **fread()**. An attempt to read the product of **items** times **size** bytes from the **CFP** of the **VFILE** addressed by the **vfd** parameter into the caller supplied buffer addressed by the **buf** parameter is made. If at least that many bytes are available in the file starting at the **CFP**, the call succeeds and returns **items** to the caller. If less than that many bytes are available, as much as is available is copied to the caller's buffer and the number of bytes copied divided by size is returned to the caller. This is an integer division, which implies that if it is important to know how many bytes were actually read, **size** should be **1**. In all cases the **CFP** is incremented by the number of bytes successfully read.

Returns

The number of items successfully read into the caller's buffer.

2.4 vfwrite

```
int vfwrite(char * buf, unsigned size, unsigned items, VFILE * vfd);
```

The calling semantics of **vfwrite()** are similar to that of the standard **fwrite()**. An attempt to write the product of **items** times **size** bytes from the caller's buffer addressed by the **buf** parameter to the **CFP** of the **VFILE** addressed by the **vfd** parameter is made. When successful, the **CFP** is incremented by the number of bytes written.

Returns

Because of its implementation, calls to **vfwrite()** either succeed completely and return **items,** or fail completely and return **0** to indicate that the file's contents were not modified. There is a possible exception to this when an external or local file system is used. The reason for the failure can be determined via a call to the **vferror()** function. The set of errors includes:

ENP_LOGIC - An attempt was made to do a write to a VFS in which write access is not enabled (**HT_RWVFS** is not defined).

ENP_FILEIO - An attempt was made to do a write to a VFS file that is write protected. Write protection of individual files is described later.

ENP_NOMEM - There was insufficient memory available to store the added file contents.

2.5 vfseek

```
long vfseek(VFILE * vfd, long offset, int mode);
```

The calling syntax of **vfseek()** is similar to that of the standard C library **fseek()**, however the semantics are quite restricted. **vfseek()** allows the caller to change the **CFP** of a **VFILE**. The **offset** parameter must be 0. Two values are accepted for the **mode** parameter: **SEEK_SET** and **SEEK_END**. Thus **vfseek()** allows the caller to position the **CFP** to either the beginning (**SEEK_SET**) or the end (**SEEK_END**) of the file.

Returns

vfseek() returns the value of the modified **CFP** when successful. It returns -1 when unsuccessful. The reasons for failure usually have to do with invalid parameter values.

2.6 vftell

```
long vftell(VFILE * vfd);
```

For uncompressed files, **vftell()** functions much as the standard **ftell()**. It returns the **CFP** of the specified **VFILE**. For compressed files, **vftell()** returns the uncompressed size of the file if the **CFP** is at the end of the file, else is returns the byte offset into the compressed file image of the current **CFP**. File compression is described in the section, "Internal Data Structures" beginning on page 14.

2.7 vgetc

```
int vgetc(VFILE * vfd);
```

vgetc() returns the value of the byte at the current **CFP** and increments the **CFP**. It returns **EOF (-1)** when the end of the file is reached.

2.8 vferror

```
int vferror(VFILE * vfd);
```

vferror() returns an error code describing what went wrong on the last attempt to write to the file.

2.9 vclearerr

```
void vclearerr(VFILE *vfd);
```

vclearerr() clears the error condition returned by **vferror()**.

2.10 vunlink

```
int vunlink(char *name);
```

vunlink() deletes the named file from the set of files maintained by the VFS. Depending on how the VFS has been configured, a call to **vunlink()** can cause the function **vfs_sync()** to be called which allows for the RAM resident VFS to be stored to the target system's backing store.

Returns

0 if the file was successfully deleted, **-1** otherwise.

The reasons for failure are:

- The named file does not exist in the VFS.
- The named file exists but was not marked as writable.

vunlink() does the same modifications to the passed in file name as does **vfopen()**.

3. INTERNAL DATA STRUCTURES

This section describes important data structures that are used by the VFS.

3.1 vfs_file Structure

```
struct vfs_file {
    struct vfs_file *next;
    char name[FILENAMEMAX + 1];       /* name of file under "path" */
    unsigned short flags;
    unsigned char * data;             /* pointer to file data, NULL if none */
    unsigned long real_size;          /* size in bytes of file before compression */
    unsigned long comp_size;          /* size in bytes of file compressed */
    unsigned long buf_size;
    /* size in bytes of memory buffer used to store file */
#ifdef WEBPORT
    /* routine to call on GET, POST, or special SSIs; NULL if none */
    int (*ssi_func)(struct vfs_file *, struct httpd *, char * args);
    /* routine to call if file is treated as CGI executable */
    int (*cgi_func)(struct httpd *, struct httpform *, char ** text);
#endif
#ifdef HT_EXTDEV
    void * method;                    /* pointer depends on flags */
#endif
  };
```

Each file in the VFS is represented by an instance of a **vfs_file** structure. These structures are linked together in a list using the **next** field. The head of the list is stored in the global:

```
struct vfs_file *vfsfiles;
```

The **name** field contains the name of the file. The **flags** field is a field of bits that describes various attributes of the file. The **flags** field is described in more detail in the section "Bits of the flags Field" starting on page 15.

The **data** field points to a buffer that contains the contents of the file. When a file is newly created, the **data** field contains NULL. A buffer is allocated and assigned to the **data** field when the first write is made to the file. As the file grows in size and exceeds the size of the allocated buffer, a new buffer is allocated to replace the old buffer, the file contents in the old buffer are copied to the new buffer and the old buffer is freed. This has an implication for the memory requirements of the VFS. When a large file is written to such that the write exceeds the size of the file, there is a short period between the time when the new buffer is allocated and old buffer is freed when there must be sufficient memory available to store effectively twice the size of the contents of the file. Porting engineers should keep this in mind when determining the memory requirements for a target system's VFS.

The **real_size** field contains the size of a compressed file before it was compressed. This information is used by the InterNiche Web Server. The **comp_size** field contains the size

of actual file image. The **buf_size** field contains the size of the buffer addressed by the data field.

The **ssi_func** and **cgi_func** fields are used by the InterNiche Web server and are not further described in this document. The **method** field is used in conjunction with external file systems, which are described later in this document.

3.2 Bits of the flags Field

The following defined constants identify the bits of significance in the **flags** field of the **vfs_file** structure. For purposes of this document they are divided into two groups.

The following four bits are of significance only to the InterNiche Web Server and are not further described in this document.

VF_AUTHBASIC	0x02
VF_AUTHMD5	0x04
VF_MAPFILE	0x08
VF_CVAR	0x10

The remainder of the bits, shown below, are significant to the VFS itself.

VF_HTMLCOMPRESSED	0x01
VF_WRITE	0x20
VF_DYNAMICINFO	0x40
VF_DYNAMICDATA	0x80
VF_NONVOLATILE	0x100
VF_STALE	0x200

The **VF_HTMLCOMPRESSED** bit indicates that the file image was compressed using the InterNiche HTML compiler. When this bit is set, the functions which read the VFS, **vgetc()** and **vfread()**, apply a decompression algorithm to the image before returning the file's contents to the caller. When a new file is created with **vfopen()** and whenever a file is written to with **vfwrite()**, the **VF_HTMLCOMPRESSED** bit is reset. The bit can be set again after the file is closed using the user interface commands described later. The intent here is to allow a file to be compressed at one location, perhaps a central site or development center and uploaded to a target system using the InterNiche FTP server. Once the FTP transfer has been completed, the target system's user interface can be used to set the bit so that when the Web server reads the file, it will be decompressed. This allows target systems' memory resources to be minimized by taking advantage of the decompression while not incurring the code overhead on the target system that would be required if the compression algorithm was also located on the target system.

The **VF_WRITE** bit indicates that the file can be written to with **vfwrite()** and deleted with **vunlink()**. Files that are created dynamically with **vfopen()** get their **VF_WRITE** bits set. Files that are linked with the target system executable via the HTML compiler may or may not have their **VF_WRITE** bits set depending on the requirements of the target system. The user interface allows the **VF_WRITE** bits to be set or reset.

The **VF_DYNAMICDATA** and **VF_DYNAMICINFO** bits are used internally by the VFS to track whether the data buffer associated with the **vfs_file** structure and the **vfs_file** structure itself were allocated dynamically.

The **VF_NONVOLATILE** bit indicates whether the file should be stored to the backing store or not. The **VF_STALE** bit is used to determine whether the file's contents have changed since it was opened. This enables the **vfclose()** function to determine whether **vfs_sync()** should be called.

3.3 vfs_open Structure

```
struct vfs_open {
    struct vfs_open * next;
    struct vfs_file * file;
    unsigned char * cmploc;      /* current position in data buf */
    unsigned char * tag;         /* current position in compressed tag, if any
    int error;                    /* last error, if any */
};

typedef struct vfs_open VFILE;
```

When a file is opened with **vfopen()**, an instance of a **vfs_open** structure is allocated. The address of the structure is what is returned as the file handle to the caller.

vfs_open structures are stored in a singly linked list using the structures' **next** field. The list is headed by the global:

```
VFILE *vfiles;
```

The **file** field of the structure points to the **vfs_file** structure that is associated with the opened file. The **cmploc** field points into the buffer addressed by the **vfs_file** structure's **data** field. It is the **cmploc** field that effectively implements the file's **CFP**. The **tag** field is used by the decompression algorithm to decompress compressed files. It is unused with regular, uncompressed files. The **error** field is used to store the error that is returned by **vferror()**.

4. PORTING ENGINEER PROVIDED FUNCTIONS

The following constructs should be provided by the porting engineer when porting the VFS to a given target system. InterNiche provides direct support for some target systems. If the target system to which a port is to be made is one of these supported target systems, or is close to it, the code that implements these constructs that is located in the target system dependent directory can be used as a starting point.

4.1 VFS_VFS_FILE_ALLOC()

VFS_VFS_FILE_ALLOC() should return a pointer to a zeroed block of memory into which the VFS will store the contents of a **vfs_file** structure, thus the block of memory returned by **VFS_VFS_FILE_ALLOC()** should be at least as large as a **vfs_file** structure. When memory is unavailable **VFS_VFS_FILE_ALLOC()** should return NULL.

For many target systems, the following defined constant implementation of **VFS_VFS_FILE_ALLOC()** which uses the InterNiche **npalloc()** function will work fine:

```
#define VFS_VFS_FILE_ALLOC() (struct vfs_file *) npalloc(sizeof(struct fs_file))
```

4.2 VFS_VFS_FILE_FREE()

The VFS will call **VFS_VFS_FILE_FREE()** when it no longer needs a buffer that had previously been allocated with **VFS_VFS_FILE_ALLOC()**. **VFS_VFS_FILE_FREE()** takes a single parameter which is the address of the buffer to be freed.

For many target systems, the following defined constant implementation of **VFS_VFS_FILE_FREE()** which uses the InterNiche **npfree()** function will work fine:

```
#define VFS_VFS_FILE_FREE(x) npfree(x)
```

4.3 VFS_VFS_OPEN_ALLOC()

VFS_VFS_OPEN_ALLOC() should return a pointer to a zeroed block of memory into which the VFS will store the contents of a **vfs_open** structure, thus the block of memory returned by **VFS_VFS_OPEN_ALLOC()** should be at least as large as a **vfs_open** structure. When memory is unavailable **VFS_VFS_OPEN_ALLOC()** should return NULL.

For many target systems, the following defined constant implementation of **VFS_VFS_OPEN_ALLOC()** which uses the InterNiche **npalloc()** function will work fine:

```
#define VFS_VFS_OPEN_ALLOC() (struct vfs_open *) npalloc(sizeof(struct
fs_open))
```

4.4 VFS_VFS_OPEN_FREE()

The VFS will call **VFS_VFS_OPEN_FREE()** when it no longer needs a buffer that had previously been allocated with **VFS_VFS_OPEN_ALLOC()**. **VFS_VFS_OPEN_FREE()** takes a single parameter which is the address of the buffer to be freed.

For many target systems, the following defined constant implementation of **VFS_VFS_OPEN_FREE()** which uses the InterNiche **npfree()** function will work fine:

```
#define VFS_VFS_OPEN_FREE(x) npfree(x)
```

4.5 vfs_lock() and vfs_unlock()

The VFS makes use of two singly linked lists to keep track of allocated data structures: the list of **vfs_file** structures headed by the global **vfsfiles** and the list of **vfs_open** structures headed by the global **vfiles**. Because some VFS API functions make additions to and deletions from these lists, while others simply traverse them, it is important to serialize access to these lists in order to prevent data corruption on systems in which it is possible for one process or task that is accessing the VFS to pre-empt another. **vfs_lock()** and **vfs_unlock()** are provided for this purpose.

Each VFS API function makes a call to **vfs_lock()** before it accesses the internal VFS data structures. Each of these functions call **vfs_unlock()** before returning to the caller. On target systems in which it is possible for task preemption to occur, the porting engineer should provide an implementation of **vfs_lock()** that blocks on the acquisition of an RTOS semaphore or mutex before returning to the caller and an implementation of **vfs_unlock()** that releases the semaphore or mutex. On superloop based systems (systems without an operating system in which only one task or process executes) or multitasking systems in which task preemption cannot occur, the implementations of these function can safely be no-ops. The functions take no parameters and return nothing to the caller.

One could implement **vfs_lock()** as a function that disabled interrupts, with **vfs_unlock()** re-enabling them, though the porting engineer should understand that there has been no attempt to make the VFS "real time" and the interrupt latency that would be introduced by such an implementation could be quite long. It has been assumed in its implementation that the VFS API will not be called from interrupt service routines. There is nothing to prevent it from functioning from ISR context, but again the interrupt latency involved would make such an approach unsuitable for most applications.

4.6 vfs_sync() and vfs_restore()

The function **vfs_sync()** is called to cause the contents of the volatile, RAM resident VFS to be written to a target system's non-volatile backing store. The function **vfs_restore()** is called at initialization to read the contents of the VFS from the target system's backing store into RAM. The prototypes of these function are shown below:

```
void vfs_sync(struct vfs_file *vfp);
void vfs_restore(void);
```

vfs_sync() is called from two locations in the VFS. When a file is closed, the **flags** field of the associated **vfs_file** structure is interrogated. If the **VF_WRITE** and **VF_NONVOLATILE** bits of the **flags** field are set, it means that the file is marked as non-volatile and has been modified since it was opened. In this case, **vfs_sync()** is called with the address of the **vfs_file** structure passed to it as a parameter. The other occasion is when a file is unlinked, **vfs_sync()** is called with a parameter of NULL. These calls are not made when the **VFS_AUTO_SYNC** defined constant is not set.

When **vfs_sync()** is called, it should do whatever is necessary to cause the contents of the RAM resident list of **vfs_file** structures headed by **vfsfiles** plus the associated file contents buffers to be copied to whatever non-volatile storage is provided by the target system in such a way that when **vfs_restore()** is called during initialization it will be possible for **vfs_restore()** to reconstruct the list in RAM. How this is best done on a given target system will depend on the requirements and capabilities of the target system. Two strategies come to mind, a complete backup and an incremental backup.

The complete backup strategy is probably the simplest to implement for most small embedded systems. In this strategy, **vfs_sync()** copies all of the files that are marked as non-volatile to the system's backing store whenever it is called. At the time of the writing of this document, the most common non-volatile storage that is found on small embedded systems is FLASH EEPROM. FLASH can be a cost effective means to provide a system with non-volatile storage, however the requirement that FLASH sectors or blocks be erased in their entirety before they can be written to makes it complicated to implement algorithms in which it is possible to perform random writes to the device. The simplest way to write to FLASH is to always erase the sectors to be written to before the write is performed. This makes a complete backup strategy in which the FLASH sectors are erased and the complete VFS is written to those sectors the most straight forward way to implement **vfs_sync()** on FLASH devices.

In an incremental backup strategy, only those files that need to be copied to the backing store are copied. **vfs_sync()** is provided with a parameter in order to allow the porting engineer to implement an incremental backup strategy. When the parameter is not NULL, the **vfs_file** structure addressed by the parameter is what needs to be backed up. When the parameter is NULL, it is because a file is deleted and all that needs to be recorded is that the list of **vfs_file** structures has had a member deleted.

The file **vfssync.c** contains implementations of **vfs_sync()** and **vfs_restore()** that implement a complete backup strategy that can be suitable for many simple embedded systems. These functions effectively implement the saving and restoring of a file directory structure to a non-volatile storage device. These implementations in turn call simpler functions that perform the actual reading and writing of blocks of data to a non-volatile device. The porting engineer can provide implementations of these simpler functions for the target system or if the target system happens to be one of the targets for which InterNiche provides ports, the functions in the target system's specific directory can be used without modification. These simpler functions are described below.

```
void *vfs_sync_open(int clear,int *p_error);
int vfs_sync_write(void *xfd,void *buf,unsigned int len);
int vfs_sync_read(void *xfd,void *buf,unsigned int len);
int vfs_sync_close(void *xfd);
```

vfs_sync_open() is called by **vfs_sync()** and **vfs_restore()** to initiate writing to or reading from the device, respectively. The **clear** parameter is set to 1 when **vfs_sync_open()** is called by **vfs_sync()** and 0 when **vfs_sync_open()** is called by **vfs_restore()**. In the InterNiche implementations of these functions for the supported target systems, **vfs_sync_open()** erases the target system's FLASH devices when **clear** is non-zero. **vfs_sync_open()** returns a non-null pointer which is used as a device handle by the other functions. If **vfs_sync_open()** fails for whatever reason, it stores a failure code in the variable addressed by the **p_error** parameter and returns NULL.

vfs_sync() calls **vfs_sync_write()** to write data to the backing store. The **xfd** parameter is the handle returned by **vfs_sync_open()**. **vfs_sync_write()** writes the **len** bytes of data stored at **buf** to the device. The requirement is that data written to the device with successive calls to **vfs_sync_write()** will be read back in the same order with successive calls to **vfs_sync_read()**. The device can in this sense be thought of as a serial or FIFO device. **vfs_sync_write()** returns 0 when successful and a non-zero failure code otherwise.

vfs_restore() calls **vfs_sync_read()** to read data from the backing store. The **xfd** parameter is the handle returned by **vfs_sync_open()**. **vfs_sync_read()** reads len bytes of data from the device into the buffer addressed by buf, returning 0 when successful and a non-zero failure code otherwise.

vfs_sync_close() is called by **vfs_sync()** and **vfs_restore()** when they are done writing to and reading from the device respectively. In the InterNiche implementations of this function for the supported target systems, **vfs_sync_close()** simply returns 0 to indicate that it was successful without doing any other processing.

The porting engineer can make the following assumptions in his implementations of these function:

The logical device provided by these functions is of a serial nature. There is no need to provide any sort of random access.

Between calls to **vfs_sync_open()** and **vfs_sync_close()** , only calls to **vfs_sync_read()** or **vfs_sync_write()** will be made. It is not necessary to support an open followed by a write, followed by a read, or visa-versa. In other words, **vfs_sync()** only does writes and **vfs_restore()** only does reads, and these operations will be bounded by calls to **vfs_sync_open()** and **vfs_sync_close()**.

The call to **vfs_sync_open()** should reset the device so that the first call to read from or write to the device does so starting at the logical beginning of the data in the device.

As an example, suppose the target system contained a battery backed up RAM device as a backing store and that the base address of the RAM was at memory address 8000h. Given this simple example, simple implementations of these functions are provided below:

```
#define RAM_BASE 0x8000
unsigned int ram_address;
void *vfs_sync_open(int clear,int *p_error);
{
    /* reset read/write pointer to beginning of battery backed up RAM device */
    ram_address = RAM_BASE;
    /* the address of the read/write pointer is our handle in this example */
    return &ram_address;
}
int vfs_sync_close(void *xfd)
{
    return 0; /* can't fail */
}
int vfs_sync_read(void *xfd, void *buf, unsigned int len)
{
    /* copy contents of battery backed up RAM at ram_address to buffer */
    memcpy(buf,*((unsigned int *) xfd),len);
    /* increment ram_offset by size of read */
    *((unsigned int *) xfd) += len;
    return 0;                       /* success */
}
int vfs_sync_write(void *xfd, void *buf, unsigned int len)
{
    /* copy contents of buffer to battery backed up RAM at ram_address */
    memcpy(*((unsigned int *) xfd), buf,len);
    /* increment ram_offset by size of write */
    *((unsigned int *) xfd) += len;
    return 0;                       /* success */
}
```

5. USER INTERFACE

The VFS includes a user interface that presents several commands that are useful for viewing and manipulating the VFS. These commands are described below.

5.1 vfsfilelist

The command **vfsfilelist** will cause the contents of the VFS to be displayed at the target system's user interface console. Each file is represented by one line of display that includes the file's name, the address of the file's contents buffer, the values of the associated **vfs_file** structure's **real_size**, **comp_size** and **buf_size** fields and the values of the bits in the **vfs_file** structure's **flags** field.

Each **flags** field bit is represented by a single character in which the character has a unique value if the bit is set and is the dash character ('**-**') if the bit is not set. The mapping of bits to characters is shown below:

H VF_HTMLCOMPRESSED
B VF_AUTHBASIC
5 VF_AUTHMD5
M VF_MAPFILE
V VF_CVAR
W VF_WRITE
I VF_DYNAMICINFO
D VF_DYNAMICDATA
N VF_NONVOLATILE
S VF_STALE

These characters are concatenated together into a string that is displayed to the right of the file name. This string also contains characters that indicate whether the **cgi_func**, **ssi_func**, and **method** fields of the **vfs_file** structure are NULL or not. When the field is not NULL, a unique character is displayed else a dash is displayed. The mapping of fields to characters is shown below:

s ssi_func
c cgi_func
m method

5.2 vfsopenlist

The command **vfsopenlist** will cause a listing of all open files to be displayed. Only the name of each open file is displayed by this command. This command can be useful during debugging to locate instances where files are being opened but not closed.

5.3 vfssetflag and vfsclearflag

On target systems in which read/write access to the VFS has been enabled (**HT_RWVFS** is defined), the **vfssetflag** and **vfsclearflag** commands allow some of the bits

in the **flags** field of **vfs_file** structures to be set and reset, respectively. The syntax of the commands is shown below:

```
vfssetflag <file name> <bit>
vfsclearflag <file name> <bit>
```

Where **file name** is the name of the existing VFS file to be modified and **bit** is a single character which indicates which of the bits is to be modified. The mapping of bits to characters is the same as that used in the **vfsfilelist** command. Not all bits can be modified in this way. The bits that can be modified and their associated identifying characters are shown below:

H VF_HTMLCOMPRESSED
B VF_AUTHBASIC
5 VF_AUTHMD5
M VF_MAPFILE
W VF_WRITE
N VF_NONVOLATILE

There is special handling for the **VF_HTMLCOMPRESSED** bit that is needed for use with WebPort, the InterNiche Web Server. When this bit is set, the HTML decompression algorithm is executed on the file's contents in order to determine the uncompressed size of the file. This size is stored in the associated **vfs_file** structure's **real_size** field. When this bit is reset, the **real_size** field is set equal to the **comp_size** field. An application of this special handling is outlined below to allow a compressed HTML file to be uploaded to a target system that is already in service.

Create the HTML file at some central location.

Compress it using the InterNiche HTML Compiler.

Use a standard FTP client and the InterNiche FTP Server running on the target system to **PUT** the compressed file to the target system. The file will be created on the target system with the **VF_HTMLCOMPRESSED** bit reset.

Use a standard TELNET client and the InterNiche TELNET Server to TELNET to the target system and use the **vfssetflag** command to set the **VF_HTMLCOMPRESSED** bit of the uploaded file.

The target system's Web server will now decompress the uploaded compressed file when Web clients that access the server cause the file to be referenced.

5.4 vfssync

The **vfssync** command will cause **vfs_sync()** to be called on the target system. This can be useful on target systems in which the **VFS_AUTO_SYNC** flag is not set.

6. LOCAL FILE SYSTEMS

When the constant **HT_LOCALFS** is defined, the functions which make up the VFS API will make their analogous standard C library buffered I/O function calls under certain circumstances. This behavior can be useful on some target systems to allow files to be accessed in the VFS and the local file system provided by the target system's C library.

The circumstances under which a VFS function will call its analogous standard C library function are described below.

vfopen() calls fopen()

When the name of the file passed in begins with a porting engineer provided prefix that is defined by the constant **VFS_NATIVE_PREFIX**. The default value of **VFS_NATIVE_PREFIX** is:

```
#define VFS_NATIVE_PREFIX "\\disk\\"
```

When **vfopen()** is called with file name that does not exist in the VFS and **HT_RWVFS** is not defined. Therefore the VFS is not configured to allow files to be created dynamically.

And when **vfopen()** is called with a file name that does not exist in the VFS but the **mode** parameter begins with '**r**', indicating that the named file must exist for the call to succeed.

vunlink() calls unlink()

When **vunlink()** is called with a file name that does not exist in the VFS.

vfread()calls vfread()
vfwrite() calls fwrite()
vfseek() calls fseek()
vftell() calls ftell()
vgetc() calls getc()
vferror() calls ferror()
vclearerr() calls clearerr()
vfclose() calls fclose()

When the passed in **VFILE** descriptor is not in the list of open files that is maintained by the VFS.

7. EXTERNAL FILE SYSTEMS

When the constant **HT_EXTDEV** is defined, the VFS will make calls to an "external file system" under some circumstances. The porting engineer can define an external file system using the following structure:

```
struct vfroutines {
    struct vfroutines * next;  /* keep these in a list */
    VFILE* (* fopen)(char * name, char * mode);
    void   (* fclose)(VFILE * vfd);
    int    (* fread)(char * buf, unsigned size, unsigned items, VFILE * vfd);
    int    (* fwrite)(char * buf, unsigned size, unsigned items, VFILE * vfd);
    long   (* fseek)(VFILE * vfd, long offset, int mode);
    long   (* ftell)(VFILE * vfd);
    int    (* fgetc)(VFILE * vfd);
    int    (* unlink)(char*);
};

extern struct vfroutines * vfsystems;
```

The **vfroutines** structure is used to define a set of entry points into an external file system. To define an external file system, the porting engineer should allocate an instance of a **vfroutines** structure, initialize its various fields with entry points into the code that implements the external file system and link the structure to linked the list of structures headed by the global **vfsystems**.

When **vfopen()** is called with a function name that does not exist in the list of **vfs_file** structures, **vfopen()** will traverse the list of **vfroutines** structures headed by **vfsystems** and call the function addressed by the **fopen()** field of each structure. This is the opportunity for the external file system to claim ownership of the file. If the file name "belongs" in the external file system, the **fopen()** function of the external file system should allocate a **vfs_file** structure for the file and set its **method** field to address the file system's **vfroutines** structure, link the **vfs_file** structure into the list of **vfs_file** structures headed by **vfsfiles**, allocate a **vfs_open** structure, set its **file** field to point to the allocated **vfs_file** structure, add the **vfs_open** structure to the list of such structures headed by **vfiles** and return a pointer to the allocated **vfs_open** structure. It should also do whatever initialization is necessary to create and maintain a file in the external file system. If the file name does not "belong" to the external file system, the external file system's **fopen()** function should return **NULL**. Note that for external file systems, the **vfs_open_files** variable must be incremented. It will be decremented by **vfclose()**.

Subsequent to this, whenever a **VFILE** handle is passed to one of the VFS entry points, the **method** field of the associated **VFILE file** field is inspected and when this **method** field is found to be non-NULL, the appropriate **vfroutine** field function pointer is called to handle the API request. For example, assuming that a **vfopen()** of a given file caused the external file system to claim the file and allocated its own **VFILE** to represent the open file instance, a subsequent call to **vfread()** with that **VFILE** handle would result in **vfread()** calling the **fread()** entry point of the external file system because the **method** field of the **vfs_file** structure addressed by the **VFILE file** field would point to the file system's **vfroutine** structure.

Index

W

X