Zalt File System

## Data Structures

In general there are a couple of basic data structures used in programs:

* Record  
  Structure, Type (class), Tuple etc.  
  Recursive/hierarchical.
* Table  
  Lists, lookup tables etc.  
  A collection of one or more associated Records into one item/row. There is an optional key.
* Hierarchy  
  Directories, Xml, (binary) trees.  
  A hierarchy of tables and/or records
* Graph or Model  
  ERD, Object Model etc.  
  An associated/related net/web of Records.

Access to a data structure:

* Pointer  
  A pointer to the complete data structure in memory. A remote data structure has to be marshaled completely before it can be accessed.
* Handle / Reference  
  An identification of a data structure that has to be locked down or dereferenced in some way to gain a pointer to it. A handle can represent a remote object but the dereferenced handle always points to a data structure in memory (see Pointer).
* Stream  
  The data structure is accessed in sequence. The unit (size in bytes) of this access can differ. Per Byte access if that makes sense (for files for instance), but the unit can also be per record or table or even larger. Remote data structures can be brought over in chunks and worked on directly by the program, not necessarily storing (copying) all the data.
* Enumeration/Iteration  
  Enumerating a collection or multi-set of streams has to maintain a current position within the collection (similar to a current position in a stream). For file systems we may want to iterate meta-streams, but for other stream sources it may be required to iterate the actual streams.  
  (comparable to a multi-result-set in ADO).

Identification of a data structure: Universal Resource Identifier: URI

Identification of ‘nodes’ inside a data structure: Universal Resource Locator: URL

Querying/Searching/Selecting nodes: OData url notation.

REST?

Example: identification/location of a file on a hierarchical file system (Open).

“file://drive/root/folder/file.ext”

Example: Finding text files in the folder directory (Open or List?).

“file://drive/root/folder/\*.txt”

Example: Getting the directory listing for folder – ending slash is important (Open or List?).

“file://drive/root/folder/”

Example: calling remote API (Invoke)

“protocol://host/object/somemethod(withparameter=value)”

Example: subscribe to a file changes (Subscribe)

“file://drive/root/folder/\*.txt”

## Storage

The idea of the Zalt file system is to abstract away the details of working with Storage Media and specific File System implementations as much as possible.

Also, as much processing as possible should be offloaded of the Z80 CPU onto the driver board (possible the System Controller).

Streams are the basic means of accessing file data. For its meta data (file name, attributes, times etc) a different approach may be required (meta stream?).

File navigation is done based on an url mechanism: “file://[named storage]/root/folder/file.txt”. File names and directories are specific to that file system. We only intercept the named storage. One storage is default and targeted when [named storage] is omitted. When using an incomplete uri or wildcard at any position, a directory listing stream is returned.

This stream mechanism can be further extended to allow accessing the internet (http://) or other IO resources (net:// for network shares or audio:// for streaming audio to or from a IO resource).

This uri/url can also be used to subscribe to system event. (sys://\*/topic).

Specific support is implemented for chunked-files (wav, bmp, midi etc). These chunks are an excellent granularity to use for working with partial files.

Uri\* ParseUri(str)

Parses the first file:// or sys:// and defers to a specialized parser for that type.

Stream\* OpenStream(uri, flags)

Returns a Stream\* for the uri and based on the flags the stream can be read-only, writable or a meta (or header) stream. (how to progress from a meta stream to the content stream?)

CloseStream(Stream\*)

Cleans up stream resources. Invalidates the Stream\*.

Stream\* Stream\_New(void\*, size) // inits preallocated memory

Stream\_getLength(Stream\*)

Stream\_setLength(Stream\*, size) // not all providers will support this

Stream\_getPosition(Stream\*)

Stream\_setPosition(Stream\*, pos)

Stream\_getFlags(Stream\*) (read-only/writable/seekable/meta)

Stream\_Read(Stream\*, buffer, size, length)

Stream\_Write(Stream\*, buffer, size)

Stream\_Copy(Stream\*, Stream\*, buffer, length)

Stream\_Notify(Stream\*, fn) // stream subscriptions

StreamReader / StreamWriter specializations:

FileStream-Reader/Writer (TextFileStream-Reader/Writer) (FileMetaStream-Reader/Writer)

Txt: ReadLine / WriteLine

ChunkFileStream-Reader/Writer (ChunkMetaFileStreamReader)

SeekChunk(chunkId)

SkipChunk(number of chunks)

(Sub)Stream\* CurrentChunk()

CurrentChunkId()

## Stream Providers

Registered driver objects that can provide a Stream\* for a specific target/uri protocol. For instance a file:// Stream Provider that interfaces with the HD or SD card to produce file streams and directory listing streams.

* Storage
  + File
  + Directory
  + ChunkFile (IFF)
* Memory
  + Resource/Font ROMs (input)
* IPC: Pipes  
  Synchronization/locks
* Network (sockets)
* Web (service)
* Audio
* Video
* Printer
* Console
  + Keyboard (input)
  + Character Screen (output)
* EEPROM (settings)
* Serial to Parallel IO

## Stream IO Transport

There are several options for communicating the data from the hardware interface of the stream resource to the Z80 program wanting to access it. Any of these are implemented by the specific Stream Provider via a private contract between software Stream Provider and hardware Stream Resource.

* DMA  
  The hardware interface (or third party) request the bus from the CPU and transfers data in blocks (probably 4k memory pages) directly into RAM.
* Serial IO  
  Using I2C or SPI or UART to send a string of bits.
* Parallel IO  
  Using successive Z80 input/output instructions on a specific IO address to communicate data.
* Dual-Ported RAM  
  The Smart IO Device implements its own RAM some of which is mapped into system memory. The Z80 uses one port and the Smart IO Device uses the other port. This saves the Z80 from being interrupted by a BUSREQ and DMA burst.

No transport is needed for Memory Streams.

## Buffered Streams

A lot of StreamProviders will employ a block-based data transfer using a buffer. Buffers are typically multiple of the memory-page size of 4k bytes. The Stream API has to be able to work with these memory blocks in a generic way in order to reduce StreamProvider implementation complexity and volume. This does mean that ALL StreamProviders have to do it in the same way. A specific flag will be available to indicate this. The Stream API can the fulfil a read or write call on its own without calling into the StreamProvider. The client of the Stream API does have to call flush/close in order to write out any buffered data to the StreamProvider, which writes it out to the Smart IO Device.

An alternate way of implementing this is using a default StreamProvider implementation for these calls…