System and Interface Changes to Inferno

C H Forsyth
Vita Nuova
forsyth@vitanuova.com
9 June 2003

Overview

This paper describes some of the changes made to Inferno interfaces as they stood in the published Third Edition manuals, to form the current Fourth Edition of the system, and the broad effects on internal and external interfaces. Changes include: extensions to the Limbo language; new instructions in Dis and the virtual machine; extra content in Dis object files; structure of the source tree; configuration of `emu`; replacement of the window system with changes to the client interface; commands renamed, replaced, and removed; revised support for network booting; 9P2000 becomes the basis for Styx; a graphics model offering alpha-blended compositing and general pixel structure; and improvements to Tk.

1. Limbo

Exceptions and fixed point have been added to the Limbo language. They are described in more detail in separate notes by John Firth, shortly to be available on the Vita Nuova web site `www.vitanuova.com`. Channels can now be buffered. A form of polymorphism is now available in Limbo.

1.1. Exceptions

Discussion of exceptions will be restricted here to implications for existing source code. The most obvious changes are that `Sys->rescue`, `Sys->rescued`, `Sys->unrescue` and `Sys->raise` have vanished. Instead the exception handling is expressed using constructions in the Limbo language. Named exceptions can be declared and used (these are described in the note by Firth), and they are declared as part of the type of functions that raise them. There is also a general 'failure' exception that effectively subsumes the old `Sys->rescue` scheme, including run-time errors such as 'out of memory' that can happen in almost any function. Unlike named exceptions a 'failure' exception can be raised or caught by any function, and its value is a string. The `raise` statement raises an exception. This is most obvious in commands that wish to produce an 'exit status'. Instead of

```
sys->raise("fail:usage");
```

one must now write

```
raise "fail:usage";
```

(That is one of the more common source changes required to Third Edition Limbo commands, since that was the most common use of exceptions before.) A block can have an `exception` handler:

```
{
    a := array[128] of byte;
    dosomething(a);
} exception e {
"out of memory:*" =>
    sys->print("i need more space: %s\n", e);
"fail:*" =>
    sys->print("exit status: %s\n", e);
"*" =>
    sys->print("unexpected error: %s\n", e);
    raise;  # propagate it
}
```

If an exception is raised during the execution of the block (including functions it calls), execution of the

block is abandoned, and control transfers to the appropriate exception handler (which is outside the block). Because the compiler and run-time system know the scope of the exception, values such as `a` above are correctly reclaimed on exit from the faulty block. Unhandled failures are propagated to callers; unhandled named exceptions (currently) become failures.

A process group can cause unhandled exceptions in any process in the group either to propagate to all members of the group, or to be propagated to the process group leader after destroying the other processes in the group. This makes it easier to program recovery from exceptions within a group of concurrent processes. For instance, if a process is expected to send to another on a channel, but fails unexpectedly instead (eg, because memory was exhausted), instead of leaving the intended recipient blocked on a receive operation, it can be sent an exception to notify it of the failure of the other process, allowing it to take appropriate recovery action. (This could sometimes be programmed using the `wait` file of *prog*(3), but not always.)

Exception handling is intended for recovering from disaster. We still think it is better Limbo style to use tuples, channels and processes to make ordinary error handling explicit. The few attempts to use failure exceptions to achieve 'pretty' but peculiar control flow have had exactly the usual effect of making the code hard to follow and error-prone.

1.2. Channels

Buffered channels have been added:

```
c := chan [N] of int;
```

where *N* is an integer value, creates a channel that will allow up to *N* integer values to be sent to it without an intervening receive without blocking the sender. If *N* is zero, the channel is unbuffered, equivalent to plain `chan of int`, and synchronises sender and receiver as before.

The restriction that a given channel value could not be sent to or received from in two `alt` statements simultaneously has been removed.

1.3. Polymorphism

John Firth has implemented a form of parametric polymorphism in Limbo. It too will be described in a separate note. Currently we are still fussing over aspects of the constraint syntax and some other implications of the most general form, and since some aspects are therefore subject to change, including syntax, we have not yet published the details. We think it is possible to use the following subset without having to change the code later:

1. Function declarations can be parametrised by one or more type variables: For example:

```
reverse[T](l: list of T): list of T
{
    rl: list of T;
    for(; l != nil; l = tl l)
        rl = hd l :: rl;
    return rl;
}
```

   Such a function can then be invoked on any compatible set of values. The function invocation does not specify the type (the compiler does type unification on the parameters). Thus the above can be used as:

```
l1: list of string;
l2: list of ref Item;
l3: list of list of string;
l1 = reverse(l1);
l2 = reverse(l2);
l3 = reverse(l3);
```

2. ADTs can also be parametrised:

```
Tree: adt[T] {
    v: T;
    l, r: cyclic ref Tree[T];
};
```

   allowing declaration of `Tree[ref Item]` and `Tree[string]` for instance.

3. Values of the parametrised type can only be declared, assigned, passed as parameters, returned, or sent down channels. The only types that can be used as actual parameter types are reference types (ie, `ref` ADT, `array`, `chan`, `list` and `module`), and `string` (which is a value type but is implemented using a reference). At some point we shall allow a function such as `reverse` above to be invoked with any compatible type (not just reference types) but that requires changes to Dis and the virtual machine not yet made.

The formal type parameters can be further constrained by listing a set of operations that they must have (which currently implies the actual parameters must be ADT types with compatible operations). We are not completely happy with the current constraint syntax, and some other aspects of the scheme, and so that be described here later once we have settled it.

2. Dis and virtual machine

To make the Limbo changes and extensions some new operators were added to the virtual machine. (We also added a `casel` operator to allow `case` statements to work on `big` values.) Modules that have exception handlers also have a (new) exception table, added to the Dis object format. Furthermore, we moved the import table used by the `load` operator out of the Dis data space into the object format (which also makes it available for inspection by `wm/rt` amongst others).

There is now an internal interface to set conditions under which modules must be signed to be loaded, and to check a signature on a module. Appropriate stubs are defined when module signing is not configured; if *sign*(3) is configured, however, it replaces them by ones that enforce its signing policy.

3. Window manager

The window manager *wm*(1) has been reimplemented by Roger Peppe. It now multiplexes pointer and keyboard input to applications, and manages windows on the display. *Tk*(2) no longer manages windows from inside the kernel. In some ways the structure is closer to that of *mux*(1) and more specifically the design described in Rob Pike's paper ''A Concurrent Window System''. It is possible to import and export window system environments between hosts.

This is one of the bigger causes of source file changes, although many of them can be done by global substitutions (eg, using *acme*(1)). Appendix A gives details. `Wmlib` is no longer the application's interface to the window system. Instead that is done through a new `Tkclient` module; see *tkclient*(2). (It uses a different `Wmlib` as an auxiliary module, and also uses a new `Titlebar` module to allow the look of the window decoration to be changed more easily). An application acquires a window by a call to `Tkclient->toplevel`; starts pointer or keyboard input if desired by calling `Tkclient->startinput`; and puts the window on screen (after sending it Tk configuration commands) using `Tkclient->onscreen`. Nothing appears on screen until that is called (which amongst other things avoids the resizing on start up that afflicted the original scheme). `Onscreen` gives it a connection to the window manager for pointer, keyboard and control input, with a separate channel for each. When it receives data from any of the channels (typically using `alt`) it must pass it to Tk using calls to appropriate `Tkclient` functions.

The toolbar used by the old *wm* is now provided by a separate program `wm/toolbar` (see *toolbar*(1)), and it is `toolbar` that interprets the `/lib/wmsetup` file. `Wm` invokes `wm/toolbar` by default so most users will see no difference, but it does make it easier to develop alternative interfaces. More visible is that `wm/logon` is now a *client* of the window manager, and must be invoked as follows:

```
wm/wm wm/logon
```

Applications need not even use *tk*(2). There is an interface for `draw`-only clients, *wmclient*(2).

4. Inferno source tree

The structure of the Inferno source tree has changed in the following ways.

4.1. Library source

The `image` and `memimage` directories have gone, replaced by `libdraw` and `libmemdraw`. The directories in the Inferno root that contain the source for libraries now always have names starting '`lib`': `libcrypt`, `libinterp`, `libkeyring`, `libmath`, etc.

4.2. Emu source

The `emu` directory now contains a subdirectory structure similar to the `os` kernels, and uses a similar configuration file (parts list) to say what goes in a given instance of `emu`. This allows platform-dependent selection of drivers, libraries and even `#/` (ie, *root*(3)) contents to be done easily.

The top directory, `/emu`, contains: `mkfile` that simply moves to the platform configured by `/mkconfig`, allowing builds in the Inferno root as before; a subdirectory `port` containing portable code (including some code shared by several platforms, such as `devfs-posix.c`); and a subdirectory for each hosting platform, distinguished by an upper-case initial letter. Current platforms include `FreeBSD`, `Irix`, `Linux`, `Nt` (for all Windows platforms after 95), `Plan9`, `Solaris`, and several others.

4.3. Emu configuration

Each platform-specific directory contains a configuration file with the same structure and indeed similar contents to the ones used for the native kernel. The default configuration file is called `emu`. Another can be chosen, again in a similar way to the native kernel, by using

```
mk 'CONF=cfile'
```

where *cfile* is the name of the configuration file. The name of the resulting executable file contains the configuration file name but depends on the platform: it is *cfile*`.exe` on Windows, `o.`*cfile* on Unix systems, and `8.`*cfile* on 386 Plan 9 systems. The configuration file format and contents is documented for all types of kernels by *conf*(10.6).

4.4. Tk source

The Tk implementation in `libtk` has been made more modular. It allows a significantly different 'style' to be implemented, and although that is by no means trivial to do, there is at least an interface to do it. We hope to change various aspects of the standard style further, but that has not yet been done.

5. Commands and modules

There are new commands and library modules, others have become obsolete and been removed, and a few existing ones have been given new names (typically when ones with similar function have been collected together). The biggest change has been to *wm*(1), which retains the same name but slightly different invocation and completely different implementation, as discussed above. Here I shall simply note the bigger changes, rather than discuss new functionality.

5.1. Renamed commands

As part of a mild reorganisation of the `/appl` and `/dis` trees, we have moved commands out of `/dis/lib` so that it now contains only library modules except for a few commands left there temporarily for compatibility. Commands themselves have sometimes been shuffled to subdirectories, often copying seemingly better structure from Plan 9, so that authentication commands are `auth/`..., naming service commands are `ndb/`..., and IP-specific commands are `ip/`... .

One noticeable change is that `lib/cs` is now `ndb/cs`. More dramatically, the command `lib/srv` (ie, *srv*(8)) has been replaced by *sh*(1) scripts, all described by *svc*(8), that contain appropriate calls to *listen*(1) after setting up any locally-desired environment.

Other commands have also moved:

- `lib/plumber` is now simply `plumber`
- `lib/bootp` and `lib/tftpd` have become `ip/bootpd` and `ip/tftpd`, documented in *bootpd*(8)
- `lib/virgild` has become `ip/virgild` (see *virgild*(8))
- `lib/chatsrv`, `lib/rdbgsrv` and `cpuslave` have moved to `auxi` (ie, `/dis/auxi` and `/appl/cmd/auxi`)
- `csquery` has become `ndb/csquery`

5.2. New or newly-documented commands

- an authentication server (signer) can use *keyfs*(4) to store its keys securely in the encrypted file `/keydb/keys` (instead of the unencrypted `/keydb/password`), and run *keysrv*(4) to offer secure change of password remotely. They are typically started, with other signing services, by `svc/auth` described in *svc*(8).

- `/dis/auth` and `/appl/cmd/auth` contain commands related to authentication; they rely on *keyfs*(4) in most cases. The older ones that use `/keydb/passwd` are still in `/dis/lib` and `/appl/lib` during the transition
- *dns*(8) has replaced the `lib/ipsrv` implementation of *srv*(2); when used, it must be started before `ndb/cs`. *Srv*(2) has reverted to being a hosted-only interface to the hosting system's native DNS resolver. It is automatically used by *cs*(8) if it cannot find *dns*(8), and *dns*(8) will also use it if available before consulting the DNS network.
- *chgrp*(1), *cpuview*(1), *grid*(1), *9660srv*(4), *cpuslave*(4), *dossrv*(4), *keyfs*(4), *keysrv*(4), *nsslave*(4), *palmsrv*(4), *registry*(4), *rioimport*, *styxchat*(1), *styxlisten*, *wmexport*, *wmimport*, and *uniq*(1) are new
- the multiplayer games software previously in `/appl/games` has been replaced by a related but significantly different system in `/appl/spree`. (Also see *spree*(2) for supporting modules.)
- *Registry*(4) provides dynamic registration and location of services using sets of attributes/value pairs, through a name space. *Registries*(2) provides a convenient Limbo interface for registration and query.

5.3. Commands removed

- `lib/csget` (see *cs*(8) for its replacement `csquery`)
- the undocumented and obsolete commands `lib/isrv` and `lib/istyxd` have been removed, since either the `none` authentication protocol, or the `-A` option to `mount` can be used if no authentication is needed
- `lib/srv` has been replaced by *svc*(8) as mentioned above.
- `getenv` and `setenv` have been removed since the Shell provides alternatives
- `wm/license` is no longer needed

5.4. New modules

There are library modules to support: registries and configuration files of attribute/value pairs; Internet address parsing and manipulation; management of windows and subwindows (used by *wm*(1) itself); timers; Styx; Styx servers; exception handling; memory and performance profiling; Freetype interface; parsing Palm databases; and navigating XML files (without reading them all into memory) and interpreting style sheets.

6. Styx

Styx was derived from the 9P protocol used by Plan 9 in 1995, with changes that reflected the requirements of the Inferno project of the time, mainly by removing features that were thought too closely tied to the Plan 9 environment. Some 9P messages were removed, particularly those that incorporated details of the Plan 9 authentication methods; Styx moved authentication outside the file service protocol. Other changes eliminated file locking and append-only files. Some restrictions that 9P imposed were retained, however, such as limiting file names to 27 bytes. This last restriction is fine for synthetic network services, but has been troublesome when trying to access Unix and Windows systems, amongst others.

A recent revision of 9P adds support for much longer file names and takes the opportunity to improve other aspects of the protocol. It also removes details of authentication algorithms from the protocol. The Styx implementation now uses the new version of 9P as the default file service protocol. (It is possible that for interoperation with older Inferno systems the system will be able to interact with both old and new versions of Styx.)

6.1. Protocol changes

The messages `Tauth` and `Tversion` are new to Styx. `Tversion` includes negotiation (at connection start) of the message size and protocol version; it also introduces a new session. `Tauth` obtains access to a special authentication file if the server requires authentication within a Styx session. `Tclone` has been replaced by a more elaborate form of `Twalk` that allows zero to MAXWELEM (16) elements to be walked, perhaps to a new fid, in a single message, returning a sequence of qid values in `Rwalk`. (A clone is simply a walk of a fid to a new fid with zero elements.) A walk of several elements can return partial results if the walk of the first element succeeds but subsequent ones fail. A partial walk leaves the state of the fids unchanged. `Ropen` and `Rcreate` return a suggested size for atomic I/O on the fid (0 means 'not given'). All strings are variable length, and consequently `Twstat` and `Rstat` data is variable length and formatted differently. Data returned from `Tread` of a directory is similarly changed, because directory entries are not fixed length.

`Tnop` has gone.

Tags remain 16-bit integers, but fids and counts become 32-bit integers (mainly of interest to large systems), and qids have a different structure. Previously a qid was a pair of 32-bit integers, path and vers, where path had the top bit set for a directory. Now a qid is a triple: a 64-bit path, 32-bit vers, and 8-bit type. The type is defined to be the top 8 bits of the file's mode. The path does not have the top bit set for a directory, and indeed the path value is not interpreted by the protocol. There are now bits in the file mode for append-only and exclusive-use files (new for Inferno), and for authentication files (new for both Plan 9 and Inferno). The stat information includes the user name that last caused the file's mtime to be changed. All strings in the protocol are variable length: file names, attach names, user names, and error text.

The message format on the wire is significantly different. The message size is negotiated for a connection by `Tversion`, and messages can be large, allowing much more data to be sent in single `Twrite` and `Rread` messages. The header includes a 32-bit message size, making it easy to find message boundaries without parsing the contents. Strings are represented as a 16-bit size followed by the string's UTF-8 encoding (without zero byte). R-messages do not carry a copy of the fid from the T-message. Padding bytes have gone. The order of some fields has changed of course to match message parameter changes.

Authentication of the connection itself, and optionally establishing the keys for digesting and encryption, is done before the protocol starts, in both Inferno and Plan 9. Details will follow on the protocol for that, and Limbo interfaces. For now, it can be assumed that the old authentication messages can still be used, even after a more flexible protocol has been implemented. `Tauth` can be used to authenticate particular accesses within such a session, but implies trust by the server that the client system will not cheat its users. (That trust is typically established by the connection level authentication which is needed anyway for link encryption, and thus for single-user clients further authentication seems extraneous in most cases.) Most Inferno services that run as file servers within a system (eg, `9660srv`) will, like Plan 9's, reply to `Tauth` with an `Rerror` stating ''authentication not required''. Access to them when exported is typically controlled as now by verifying the incoming connection.

6.2. Limbo interface changes

Because Limbo's interface to file service via `Sys` and other modules uses Limbo `string` for names, and that is inherently variable length, there are no interface changes required for that aspect of the protocol change, and consequently no source changes (in contrast to the introduction of 9P2000 in C implementations). Similarly the Inferno directory reading interfaces remain unchanged.

The 'directory mode' bit previously called `CHDIR` is now called `DMDIR`. It is used *only* in `Dir.mode`. `CHDIR` is no longer defined, partly because it was used both in `Dir.mode` and `Qid.path`, and the latter instances must change (discussed below). There are bits (new to Inferno) for `DMAPPEND` (append-only file), `DMEXCL` (exclusive-use file), and `DMAUTH` (authentication file). The protocol can return the user name of the user that caused `mtime` to be changed on a file; that is now available as `Dir.muid`.

The structure of `Qid` has changed. Previously a Qid had a 32-bit `path` and a 32-bit version number, `vers`. The top bit (`CHDIR`) of `path` was set iff the Qid was that of a directory. The `path` is now 64 bits (which is `big` in Limbo and `vlong` in the kernel), and there is no longer the convention that the top bit of `path` must be 1 for a directory. Instead, there is a new, separate `type` field (called `qtype` in Limbo) that has the value of the top 8 bits of the file's mode. Each bit `DM`*x* in `Dir.mode`, has got a corresponding bit `QT`*x* in `Qid.qtype`: `QTDIR`, `QTAPPEND`, `QTEXCL` and `QTAUTH`. The bit `QTDIR` *must* be set in the `Qid.qtype` for a directory, and only then. There is an extra constant `QTFILE` that is defined to be zero, and is used for clarity when neither `QTDIR` nor `QTAUTH` is set.

In Styx file servers, changes are required to reflect the slightly different set of message types and a few new parameters, but the main changes are: handling zero or more name elements at once in `Twalk` and `Rwalk`; changing `CHDIR` to `DMDIR` in `Dir.mode` (easy); the use of the new `Qid.qtype` field and `QTDIR` instead of `CHDIR` in `Qid.path` (a little more effort); and (typically) the insertion of casts to force `Qid.path` to `int` and thus ensure the use of 32-bit operations except where 64-bit paths really are needed (hardly ever in synthetic file servers). The new modules for use by file servers are discussed in the next section.

The revised definition of `Twstat` in *stat*(5), and thus `sys->wstat`, provides for ''don't care'' values in `Dir` that are tedious to provide directly; a new adt value `Sys->nulldir` provides the right initial value for a `Dir` which is then changed as needed for `wstat`.

Examples

Create a directory:

old:
```
fd := sys->create(name, Sys->OREAD, Sys->CHDIR | 8r777);
```

new:
```
fd := sys->create(name, Sys->OREAD, Sys->DMDIR | 8r777); # not CHDIR
```

Make Qids for a file and a directory:

old:
```
Qdir, Qdata: con iota;
qd := Sys->Qid(Sys->CHDIR | Qdir, 0);
qf := Sys->Qid(Qdata, 0);
```

new:
```
Qdir, Qdata: con iota;
qd := Sys->Qid(big Qdir, 0, Sys->QTDIR);
qf := Sys->Qid(big Qdata, 0, Sys->QTFILE);
```

Test if a file is a directory:

old:
```
isdir(d: Sys->Dir): int
{
    return (d.mode & Sys->CHDIR) != 0;
OR:
    return (d.qid.path & Sys->CHDIR) != 0;
}
```

new:
```
isdir(d: Sys->Dir): int
{
    return (d.mode & Sys->DMDIR) != 0;
OR:
    return (d.qid.qtype & Sys->QTDIR) != 0;
}
```

If one wishes to have values `big` only when required, one can write:

```
case int dir.qid.path {
Qdir =>
    ...
Qdata =>
    ...
Qctl =>
    ...
}
```

Of course with the Dis change mentioned above, `case` can now be applied to `big` values, so it is no longer necessary to add the cast (as it once was). Even so, 32-bit operations are faster when they suffice.

6.3. Styx protocol in Limbo: Styx and Styxservers

A new module `Styx`, defined by `styx.m`, provides access to the Styx protocol messages, as variants of pick adts `Tmsg` and `Rmsg`. (There was an old, undocumented `Styx` module but this new interface is completely different.) It is used by several file servers, such as `dossrv`, `cdfs`, and the new `logfs`. See the attached manual page. There are several implementations with the same signature, covering different combinations of old and new Inferno and old and new protocols, through the same interface. There are slight differences in the application code for old and new systems because of the changed type and structure of `Qid`. The versions that talk the old protocol need to store some internal state, and are intended only to meet compatibility requirements during the transition.

Many file service applications, however, serve a simple name space, requiring more than can be done with `file2chan`, but wishing some help in handling the protocol details. Two new modules `Styxservers` and

`Nametree` are provided to make such applications easier to write. They are closely related and thus both modules are defined by `styxservers.m`.

`Styxservers` provides help in handling fids and interpreting the Styx requests for navigating a name space, and provides a reasonable set of default actions, allowing the application to focus on implementing read and write access to the files in the name space. It uses `Styx` to talk to the Styx client on a connection. It interacts with the application through a channel interface and the `Navigator` adt to navigate an abstract representation of the application's name space. The module can be used on its own, with the application doing the work of replying to those queries itself, or it can get extra help in the common cases from `Nametree`. `Nametree` provides a `Tree` adt and operations for the application to build an abstract representation of a name space and maintain it dynamically quite simply, and it exports the channel interface used by `Styxservers` for navigation, thus connecting the two, but leaving the application in complete control of the name space contents viewed by Styx. See the manual pages *styxservers*(2) and *styxservers-nametree*(2), attached. The latter includes a short working example of combining the two modules.

The previous release of the system had a module `Styxlib` that combined the functions of `Styx` and `Styxservers`. It remains for a time for transition, but newer applications should use either `Styx` or `Styxservers`.

A new command *styxchat*(8) exchanges Styx messages with a server, reading a textual representation of T-messages on standard input. It can be helpful when testing a Styx server implementation. (It was originally developed to test the `Styx` module implementations in several configurations.) See the attached manual page for details. It also supports an option that allows it to act as a server, printing T-messages as they are received from clients, and reading R-messages in a textual form from standard input for replies.

6.4. Device driver changes

Most of the differences for most drivers are relatively minor (in `diff` terms).

Throughout the hosted and emulated kernels:

- `Qid` now is the structure:

```
struct Qid {
    vlong   path;
    ulong   vers;
    uchar   type;
};
```

  The `type` field has values `QTDIR`, `QTFILE`, `QTAPPEND`, etc. The test previously written

```
if(qid.path & CHDIR)
```

  is now written

```
if(qid.type & QTDIR)
```

  Because of that change, the various `switch` statements in the drivers that previously read

```
switch(c->qid.path){
```

  or

```
switch(c->qid.path & ~Sys->CHDIR){
```

  now read

```
switch((ulong)c->qid.path){
```

  to keep operations to 32 bits (except where otherwise required).

- The first entry of a driver's `Dirtab` *must* be an entry for `"."`, if the driver uses `devgen` to help implement *walk*, *stat*, *devdirread* or *open* operations.
- Offsets passed to the driver's *read* and *write* entry points are 64-bit `vlong`, not 32-bit `ulong`.
- The *stat* entry point has an extra buffer size parameter:

```
int xyzstat(Chan *c, uchar *dp, int n)
```

  It also returns an integer: the size of the result. `Devstat` accepts the extra parameter and returns an appropriate result:

```
static int
xyzstat(Chan *c, uchar *dp, int n)
{
    return devstat(c, dp, n, rtcdir, nelem(xyzdir), devgen);
}
```

- The biggest change is to *walk*. It has the signature:

  ```
  Walkqid *xyzwalk(Chan *c, Chan *nc, char **names, int nname);
  ```

  and it allows zero or more elements to be walked in a single call, returning its result in a newly-allocated `Walkqid` structure:

  ```
  struct Walkqid {
      Chan*   clone;
      int nqid;
      Qid qid[1];
  };
  ```

  Note that the array `Walkqid.qid` must actually hold up to *nname* Qids, and thus is allocated as follows:

  ```
  wq = smalloc(sizeof(Walkqid)+(nname-1)*sizeof(Qid));
  ```

  The driver must take care that the space is reclaimed if `error` is called before its *walk* function returns, by using `waserror` as required. Fortunately, `devwalk` looks after the details of *walk* and `walkqid` for most drivers:

  ```
  static Walkqid*
  xyzwalk(Chan* c, Chan *nc, char** name, int nname)
  {
      return devwalk(c, nc, name, nname, xyzdir,
          nelem(xyzdir), devgen);
  }
  ```

- The *clone* entry point has gone, since cloning is seen by a driver as a particular form of call to its *walk* entry, where the parameter values satisfy:

  ```
  c != nc && nwname == 0
  ```

  One difference is that a node can be cloned and walked in a single operation, in other words `nwname` can be non-zero, and the incoming `nc` is often nil and a new `Chan` must be allocated. Note that if the driver found it adequate to call `devclone` previously, then the new `devwalk` will generally look after it as well. `Devclone` remains for use as a utility function for the few drivers that need to clone a channel themselves, in their *walk* operations or elsewhere.
- The *detach* entry has been renamed *shutdown* (it was never the opposite of *attach*). The stub `devshutdown` can be used by devices that do not need it.

For drivers that serve a simple name space using the functions of `dev.c` (described in *devattach*(10.2)), only a handful of simple changes are required. Most are pointed out by the compilers as type clashes. The main exception is the need for a `Dirtab` to have its first entry be an entry for `"."` if the `Dirtab` will be passed to `devgen` via `devwalk`, `devstat` and `devdirread`.

7. Sys module changes

7.1. Sys: name change(s)

The name `ERRLEN` has become `ERRMAX` (since it is the limit to any error string, not its necessary length). `NAMELEN` has been removed, to allow each instance to be found (by compilation) and either removed (where it was simply limiting the length of a file name), or replaced by `NAMEMAX` where it was used as a buffer size to read in names such as `/dev/sysname` or `/dev/user`.

7.2. Sys: file sizes

The Styx protocol has always supported 64-bit file sizes and file offsets. The Inferno interface has not. `Sys` has changed so that length and offset values become `big`, specifically: file size `Dir.length`, the offset parameter to `seek`, and `seek`'s result.

These and the Qid changes account for quite a few changes in our own source tree. Typically, applications did things like this:

```
    old:
buf := array[d.length] of byte;

sys->seek(fd, 0, Sys->SEEKSTART);
off := sys->seek(fd, 0, Sys->SEEKRELA); rec := off + HDRLEN;
for(offset := 0; offset < d.length; offset += RECSIZE){
    sys->seek(fd, offset, Sys->SEEKSTART);
    ...
}
```

The compiler now objects in each case because `big` values are now appearing where `int` is required, or conversely. In some cases it is obvious that adding a cast is correct; in others it is worth considering whether the calculation should indeed be `big` because file sizes for instance can in practice exceed the range of a signed integer without too much trouble today, especially when the 'file' is a storage device. The case that some people like and some dislike is:

```
if(sys->seek(fd, big offset, Sys->SEEKSTART) < big 0) ...
```

where the `big 0` is needed because `sys->seek` is `big`, and there are no 'usual arithmetic conversions' as in C. (Given the tangle that several languages have made of such conversions, perhaps being strict is correct.)

7.3. Sys: export

`Sys->export` now has the signature:

```
export: fn(c: ref Sys->FD, dir: string, flag: int): int;
```

allowing a directory *dir* other than `"/"` to be exported. It replaces the `exportdir` function of (later) Third Edition.

7.4. Sys: Styx support

The revision of Styx has caused three calls to be added:

```
fauth:    fn(fd: ref Sys->FD, aname: string): ref Sys->FD;
fversion: fn(fd: ref Sys->FD, msize: int, version: string): (int, string);
iounit:   fn(fd: ref Sys->FD): int;
```

`Fversion` initialises a Styx session on connection *fd*, sending the message size *msize* and protocol version string *version*; it returns a tuple giving the message size and version returned by the Styx server. It is rarely called directly; the `mount` operation does it automatically on an uninitialised connection.

`Fauth` sends a Styx `Tauth` message on connection *fd*, and if successful, returns a file descriptor that refers to an authentication file provided by the file server, which may be read and written by `Sys->read` and `Sys->write` to implement the authentication protocol(s) supported by the server. `Fauth` is needed only when the server requires authentication.

`Iounit` returns the 'atomic IO unit' suggested for the file *fd* by its file server when it was opened.

7.5. Sys: mount

The `mount` system call has acquired a second file descriptor parameter:

```
mount: fn(fd: ref Sys->FD, afd: ref Sys->FD, on: string,
          flags: int, spec: string): int;
```

Afd is nil if the file server is known not to require authentication within a Styx session. (The connection might itself have been authenticated previously, for instance, and most file servers such as `dossrv`, `ftpfs` and `dbfs` are invoked to provide services to an already-authenticated user, and therefore do not require authentication within a session.) If the server does require authentication, *afd* refers to a file descriptor

returned by a previous `fauth` on connection *fd*, on which an authentication protocol has subsequently been executed as required by the file server connected to *fd*.

7.6. Sys: other new system calls

There are two more new system calls:

```
fd2path: fn(fd: ref Sys->FD): string;
werrstr: fn(s: string): int;
```

`Fd2path` returns the path name under which the file descriptor *fd* was originally opened (if known). One result is that *workdir*(2) produces reasonable results for the name of the current directory in the presence of mounts and binds.

`Werrstr` sets the per-process system error string to *s*, to allow a Limbo function to save and restore an error string over other system calls, to present a similar interface as the system calls on errors, or to annotate the error from a system call for its own caller.

7.7. Sys: directory reading

The *sys-dirread*(2) system call's signature has changed:

```
dirread: fn(fd: ref Sys->FD): (int, array of Sys->Dir);
```

Previously it accepted an array of `Dir` to fill and returned a count; now it returns a tuple containing the count and the array of values read. The change was needed because the representation of directory entries is now variable length, and it is difficult to limit the number returned (it is possible, but all the methods have disadvantages). `Dirread` still reads a directory incrementally, requesting a block of directory entries of reasonable size from the file server, and unpacking them into the returned array. Use *readdir*(2) to read whole directories at once.

8. Bufio

There are several changes to `Bufio`:

```
Iobuf: adt {
    ...
    seek:   fn(b: self ref Iobuf, n: big, where: int): big;
    offset: fn(b: self ref Iobuf): big;
};
# flush: fn();  # deleted
```

The module-level function `Bufio->flush` has been removed (*not* `Iobuf.flush`), to allow concurrent use of a single `Bufio` instance; applications must `close` or `flush` each output file explicitly.

As a result of the change to 64-bit offsets for `Sys->seek`, `Iobuf.seek` also accepts and returns `big` offsets. `Iobuf.offset` is new, and returns the current file offset in bytes, taking account of any buffering.

`Iobuf.flush` has been extended to flush any data buffered on input files.

9. Draw

The graphics model represented by the *draw*(3) device and the `Draw` module is significantly different, including support for a range of pixel formats, and compositing in the drawing operations. Most source code that uses Images directly will require some changes, but the scope of them is limited: needing only extra or different parameter values to individual operations, not radical restructuring. The following changes affect most non-Tk graphics application code:

- Pixels in an `Image` can now be more than 8 bits and have a more flexible structure (eg, several colour channels, and an optional alpha channel, of up to 8 bits each). To support that, the old `ldepth` field has gone, replaced by a channel descriptor `chans` of type `Chans`, which describes the pixel structure, and an integer `depth` field, which gives the total pixel size (depth) in bits.
- The colour parameters are now 32-bit RGBA values (red, green, blue and alpha components, 8-bit each, and big-endian only when an `int`).
- The graphics subsystem supports Porter-Duff compositing, combining a destination image with a source image (within an optional matte) according to a compositing operator. The interpretation of the old 'mask' Image parameter to `draw` and `gendraw` has changed. Previously it provided a simple binary mask; it now provides a 'matte', and its alpha channel shapes the source image and adds

partial transparencies. If the matte parameter is nil, the source image is used unmodified. If it lacks an alpha channel, one is computed from the matte image colour channels. The drawing operations `draw`, `gendraw`, `line`, `text`, and so on, have all got variants `drawop`, `gendrawop`, `lineop`, `textop`, and so on, each taking an extra final parameter that specifies a Porter-Duff compositing operator from a set predefined by `Draw`: `SoverD`, `SinD`, `DatopS`, and so on. In each case, `S` refers to the source image (within a matte, if provided), and `D` refers to the destination image. Most of them are useful only when either or both source or destination images have got alpha channels (or a matte is used to shape the source). The old function names without the `op` suffix use the most common compositing operation `Draw->SoverD`, drawing the source image over the destination, taking account of the shaping of the source and destination images by their alpha channels, with the source further shaped by the optional matte. Thus `Image.draw` continues to do the 'obvious' thing.

- There are new colour map conversion functions.

The `Chans` adt is the following:

```
Chans: adt
{
    # interpret standard channel string
    mk:    fn(s: string): Chans;
    # standard printable form
    text:  fn(c: self Chans): string;
    # equality
    eq:    fn(c: self Chans, d: Chans): int;
    # bits per pixel
    depth: fn(c: self Chans): int;
};
```

Values are created by `Chans.mk`, which accepts a string that is a sequence of channel descriptors, each being a letter representing a channel type followed by an integer giving the channel's size (depth, width) in bits. The letters include: `r`, `g` and `b` for red, green and blue; `a` for alpha; `k` (!) for greyscale; and `x` for padding (''unspecified'', ''don't care''). Thus `Chans.mk("r8g8b8a8")` produces a descriptor for a 32-bit pixel with 8-bit colour and alpha components. The same descriptor is used in the revised *image*(6) format, although the older image file format with ldepth only is still recognised. Given a Chans value *c*, *c*`.text()` returns such a descriptor for it as a string.

When `newimage` previously was called with a specific value for `ldepth`, an appropriate `Chans` value must replace it. A few common variants are defined as constants of type `Chans` in `Draw`. (We extended the Limbo compiler last year to support the use of `con` with adt and tuple constants with this use in mind.) For example, the value `Draw->CMAP8` is the descriptor for the 8-bit deep *rgbv* colour-mapped Image format previously used by Inferno. The list of predefined channels includes:

Old ldepth	*Name*	*Bit depth*	*Description*
0	`GREY1`	1	single 1-bit deep greyscale channel
1	`GREY2`	2	single 2-bit deep greyscale
2	`GREY4`	4	single 4-bit deep greyscale
–	`GREY8`	8	single 8-bit deep greyscale
3	`CMAP8`	8	single 8-bit deep *rgbv* colour-mapped channel
–	`RGB15`	15	three channels RGB: r5g5b5
–	`RGB16`	16	three channels RGB: r5g6b5
–	`RGB24`	24	three channels RGB: r8g8b8
–	`RGBA32`	32	four channels: RGB and alpha: r8g8b8a8

The use of `Chans` instead of `ldepth` means that calls to `Display.newimage` must be changed. For instance:

(old)
```
buffer := display.newimage(r.inset(3), t.image.ldepth, 0, Draw->White);
```

becomes

(new)
```
buffer := display.newimage(r.inset(3), t.image.chans, 0, Draw->White);
```

There is an obvious difference: the use of `t.image.chans` instead of `t.image.ldepth` to create a buffer Image with the same pixel structure as `t`. There is, however, another difference. The final colour parameter

to `newimage` is also different in structure: in the new graphics model, it is a 32-bit integer value giving RGBA components, not a colour map index, and the name `Draw->White` has the value `16rFFFFFFFF` not `0`. Because a symbolic name was used, however, the source need not change. As another example, `Draw->Palegreyblue` is `int 16r4993DDFF`. Note the final `FF` for the alpha component (creating a fully opaque colour). When the top bit is set, the `int` cast shown here is needed to force the otherwise `big` value to 32 bits.

The values of colour components are now uniformly expressed as intensity, so that a pixel with all zero colour components is black and one with all colour components at maximum (all ones, full intensity) is white. The *rgbv* map has therefore been reversed. Given a map index, `Display.cmap2rgba` returns the 32-bit RGBA format used as a parameter in other calls. All colour components are *linear* values, as required for compositing to work properly; gamma correction is done as required by the display subsystem.

The colour components of a pixel with an alpha component are always *pre-multiplied* by the alpha value, following Porter and Duff, as further justified by Alvy Ray Smith and Jim Blinn. ''Thus a 50% red is `16r7F00007F` not `16rFF00007F`.'' The function `Draw->setalpha` does the computation.

Because of the changes to colours and the replacement of simple masks by mattes, the Images `Display.ones` and `Display.zeros` are no longer defined. Instead, when they were intended to represent colours, the new Images `Display.black` and `Display.white` provide the obvious colours. When `ones` and `zeros` were used as masks, the new predefined Images `Display.opaque` and `Display.transparent` are used instead as constant mattes, with alpha channels (fully opaque and fully transparent, respectively). As noted above, where `Display.ones` was used as a mask parameter in drawing operations, one can simply specify a nil Image as a matte ('no matte') instead. (That has been allowed for quite some time and is in use but might not be widely known.)

For example, Charon allocated a mask using:

```
dpicmask = display.newimage(pic.r, 0, 0, Draw->White);
```

which becomes

```
dpicmask = display.newimage(pic.r, Draw->GREY1, 0, Draw->Opaque);
```

where `GREY1` is a constant value of the `Chans` adt type, predefined by Draw, for Images that have a single 1-bit deep grey channel (ie, a bitmap). (Note that to form a fully-opaque matte, `Draw->Opaque` was used for clarity, not `Draw->White`; `Draw->Transparent` could also be used, as the basis for building a matte with transparency.)

A small if obscure change is that `Display.newwindow` has a new parameter:

```
newwindow:  fn(screen: self ref Screen, r: Rect,
    backing: int, color: int): ref Image;
```

The *backing* parameter should usually be `Draw->Refbackup`, except for windows allocated on an image that already has got backing store assigned, for instance because it is an image on a screen on an existing window image, in which case it should be `Draw->Refnone`, because the parent window already provides the backing.

As a small but helpful change, the adt `Draw->Pointer` has a new element `msec` that reports a relative time stamp in milliseconds.

The `Draw->Context` content is significantly different, for the benefit of the new window system implementation.

10. Tk module

There is a new function in `Tk`:

```
quote:   fn(s: string): string;
```

`Quote` returns string *s* quoted according to Tk's '`{}`' quoting conventions. It replaces `Wmlib->tkquote`.

There is a new widget type: *panel*(9). A panel instance can be packed and otherwise manipulated in the same way as any other Tk widget. An image is associated with it by calling `Tk->putimage` defined in *tk*(2). The associated images can be drawn on directly by the application, using all the operations provided by `Draw`. The coordinates of the changed rectangle must be given to Tk using the `panel` widget command `dirty`; that part of the image will be redrawn if necessary at the next Tk `update`. A panel has no default bindings. See *panel*(9) for details.

For example, `wm/coffee` now uses the following:

```
r := Rect((0, 0), (400, 300));
buffer := display.newimage(r, t.image.chans, 0, Draw->Black);
tk->cmd(t, "panel .f.p -bd 3 -relief flat");
tk->cmd(t, "pack .f.p -fill both -expand 1");
tk->cmd(t, "update");
org := buffer.r.min;
tk->putimage(t, ".f.p", buffer, nil);
```

When it has updated the `buffer`, it tells Tk:

```
tk->cmd(t, ".f.p dirty; update");
```

In this case the whole image is marked dirty, but `dirty` can be given an optional rectangle parameter to restrict redrawing.

`Tk->putimage` and `Tk->getimage` replace `imageput` and `imageget`.

11. Selectfile, Tabs and Dialog

The functions `filename`, `mktabs` (and `tabsctl`), `dialog` and `getstring` have been moved to separate new modules, to allow those aspects of the user interface to be changed by replacing the implementations, and to allow standard modules to be provided for picking colours (for instance). `Selectfile` acquires `filename`, `Tabs` acquires the 'tabs' Tk pseudo-widget, and `Dialog` acquires `dialog`, which is renamed `prompt`, and `getstring`. In cases where the functions took a `Tk->Toplevel` as a parameter to specify a `parent` window, they now take a `Draw->Context` and (parent) `Image` parameter; given a Toplevel `t`, use `t.image`. See *dialog*(2), *selectfile*(2) and *tabs*(2).

Appendix A: Tk client conversion

Wm(1) applications now have to feed their own pointer and keyboard input to Tk. The window manager is now kept informed about the placement of windows.

A Tk toplevel now holds a window manager context:

```
Wmcontext: adt
{
    kbd:        chan of int;        # incoming characters from keyboard
    ptr:        chan of ref Pointer;    # incoming stream of mouse positions
    ctl:        chan of string;     # commands from wm to application
    wctl:       chan of string;     # commands from application to wm
    images: chan of ref Image;  # exchange of images
    connfd: ref Sys->FD;        # connection control
    ctxt:       ref Context;
};
```

It contains some channels on which the window manager sends information to the application, and a file descriptor that can be used to write requests to the window manager. The channels used directly by the application are:

`kbd` characters typed by the user (pass them to `Tk->pointer`)

`ptr` pointer events (pass them to `Tk->keyboard`)

`ctl` application control requests. Passing these to `Tkclient->wmctl` will do the default action. Requests starting with an exclamation mark (`!`) can cause the application's image to change.

The toplevel also holds a channel `wreq` on which it sends application control requests; these have the same form as those sent on `Wmcontext.ctl`, and can be forwarded to `Tkclient->wmctl` in the same way.

Control requests currently understood by *wm*(1) are:

`!reshape` *tag reqid minx miny maxx maxy* [*how*]
Reshape the window referenced by *tag*, creating a new image if *tag* did not previously exist. *Reqid* is ignored. *How* can be one of:

`place` *Wm* attempts to find a suitable patch of screen real estate on which to place the window; the size of the given rectangle is taken to be the minimum size for that window.

`exact` Reshape to the exact rectangle requested. This is the default if *how* is not

given.

`onscreen` The given rectangle is adjusted so that it is no bigger than the available screen space, and is entirely on screen.

`delete` *tag*
Delete the image associated with *tag*.

`raise`
Raise the window

`lower`
Lower the window

`!move` *tag reqid startx starty*
Request the user to move the window to a new place. *Startx* and *starty* are the coordinates of the pointer when the request was initiated.

`!size` *tag*
Request the user to resize the window.

To convert a typical Tk application, do the following.

1. Use an editor to make the following changes:

Old	*New*
`Wmlib`	`Tkclient`
`wmlib`	`tkclient`
`tkclient->titlebar`	`tkclient->toplevel`
`tkclient->titlectl`	`tkclient->wmctl`
`tkclient->taskbar`	`tkclient->settitle`
`tk->imageput`	`tk->putimage`
`tk->imageget`	`tk->getimage`

2. Insert the following code at the top of the central `alt` statement. The names '`wmctl`' and '`top`' will need changing to the appropriate variables in the program:

```
s := <-top.ctxt.kbd =>
    tk->keyboard(top, s);
s := <-top.ctxt.ptr =>
    tk->pointer(top, *s);
s := <-top.ctxt.ctl or
s = <-top.wreq or
s = <-wmctl =>
    tkclient->wmctl(top, s);
```

3. Add the following just after the Tk configuration code and before the main processing starts:

```
tkclient->onscreen(top, nil);
tkclient->startinput(top, "kbd"::"ptr"::nil);
```

This is possibly the easiest part to forget.

Be careful of cases where a blocking function is called from the main loop that relies on keyboard/mouse input. The easiest solution can be to spawn a thread to handle the keyboard and mouse independently.