

Xwedge: Technical Documentation

Germano Caronni
TIK/ETHZ

1. Interface to the Transport Infrastructure

The Xwedge is not only separated into different modules to localize treatment of the X protocol elements, but also contains a lower layer dealing exclusively with communication issues. This layer is called GenIO, a generic I/O interface to the transport infrastructure. In this chapter we give a ratio for GenIO, and show what benefits resulted from using this interface.

1.1 Motivation

After having finished a first version of the Xwedge in October '93, the decision was taken to introduce an independent communication submodule into the Xwedge. This module was designed to fulfill the following requirements:

1.1.1 Simplify the core

Connection-negotiation, buffering, handling of failures and all other directly communication-related parts had to be removed from the Xwedge core, this would substantially reduce the complexity of the agents. A 'generic' I/O-interface was to be introduced and had to allow a standardized handling of exceptions which would occur during communication. Thus error handling in the actual core could be somewhat unified.

1.1.2 Transparently handle different mechanisms, allow additions

At the same time the liberty had to be created to handle different existing communication mechanisms as they are provided by a typical UNIX kernel (e.g. internet- and unix-domain sockets, shared memory) in a way that would be transparent for the application using them. The addition of future transport protocols like XTPX, and the possible employment of multicast-connections to distribute data which has to reach all clients, were supposed to be realizable with changes that were to be localized in a small part of the agent.

1.1.3 Enhance portability

Now, at the end of the whole project, the Xwedge is been implemented on different platforms. The portability of the Xwedge itself was enhanced by the concentration of certain system-dependent parts in the communication submodule. Thus an adaption of the submodule to another architecture like IBM-compatible or MacIntosh machines was mostly sufficient to successfully port the Xwedge itself to a different environment.

1.2 Result: A new abstraction level, the channels

The creation of the communication submodule with its interface to the Xwedge allowed the introduction of a new abstraction level. Links between communicating partners can simply be interpreted as ‘virtual’ channels, whose exact realisation is absolutely irrelevant to the Xwedge. A channel may be created to accept incoming connections using multiple communication mechanisms simultaneously, or simply to connect to such an endpoint, called listener, without knowing how this is done. So called group-channels exist, which allow output to be directed to different partners at the same time, without having to worry about using multicast packets or employing other mechanisms.

1.3 Communicating with the rest of the world

The connectivity to other services is granted by allowing the association of a channel with a UNIX file-descriptor, assuming that read and write on the object the descriptor refers to are meaningful. Another provision to reach ‘outside’ services is the ability to provide specifications for the wanted I/O-mechanisms and addresses while associating a connection or a listener with a channel.

1.4 Performance enhancements

By automatically choosing the most efficient available communication mechanism upon connection setup, and the aforementioned conceptual changes which were introduced into the Xwedge, a significant increase of throughput for X packets is achieved. The measurements for different mechanisms are presented in the paper `XXREFERENZ AUF DAS TON_PAPER`

1.5 Envisioned features: QoS

The communication submodule was envisioned with the ability to provide different types of services depending on implemented communication mechanisms and settable options. At the moment, only reliable channels are supported, but for certain applications like shared picturephones or video/audio presentations, it would be acceptable to transmit live data over lossy or even distorting channels, as long as transmission is more efficient and achieves the speed needed for such applications.

1.6 Conclusions:

The introduction of the communication submodule has led to the following significant changes to the Xwedge:

- The communication issues were isolated from the agents, the so introduced abstraction level helped to simplify the Xwedge core.
- Portability of the Xwedge was enhanced by localizing communication- and therefore system dependent parts in its own module. At the same time its future maintenance and extendability have been simplified.
- By implementing additional communication mechanisms (shared memory and direct) in the generic IO module, considerable performance improvements have been achieved without modifying the Xwedge core.

2. System Independent Properties of GenIO

GenIO is a library (`libgenio.a`) which can be linked to applications, and offers a well defined interface (see `genio.h`) to an abstract entity named 'channel'.

2.1 Channel types

A channel hides the differences of possibly employed underlying communication mechanisms from the application, and is one of four kinds:

1. **Listener:** The channel accepts connections from the outside. One listener may accept connections from different underlying communication mechanisms simultaneously. If a connection is established from the outside, a new channel is spawned.
2. **Incoming:** This is the kind of channel spawned from a listener, usually hiding one underlying communication mechanism. A channel offers both the possibility to read and to write data, it is bidirectional.
3. **Outgoing:** No difference exists between an incoming and an outgoing channel after a connection has been established, this distinction is currently unneeded. Any outgoing channel is created by a successful connect to a listener, or a 'connect' to a UNIX file-descriptor.
4. **Group:** This type of channel is write-only. Incoming and outgoing channels may be added to a group channel, causing a write to the group channel to be forwarded to all regular channels associated with it. The concept of group channels was realized to support future multicasting communication mechanisms, and to facilitate the management of communication groups.

2.2 Event driven approach

GenIO assumes the program employing it is running in an event driven fashion. After an initialization phase, control is passed to the main loop of GenIO `io_main()` and stays there for a certain amount of time. Although a more 'polling' behaviour, as to serve a X event loop simultaneously, is possible and has already been used, it is certainly not encouraged. In the main loop, new connections are accepted, incoming data is delivered, and dead connections are signalled. Additionally, timed events may be passed from the application to GenIO, which are then executed when the time specified is reached.

All operations in GenIO which were caused by events on the communication side are passed to the application via back calls. The application has to specify procedures upon creating connections or listeners, which are then called if data arrives, new connections are established on a listener or a connection dies. Thus it is not possible to call a function `'io_read()'`, but a reader procedure of the application is called when data arrives.

2.3 Communication mechanisms

This is the list of communication mechanisms currently employed by GenIO:

- TCP-connection via Internet Domain Sockets.
- TCP-connection via Unix Domain Sockets.

- XTPX-connection via XTP Domain Sockets. Only usable if XTPX is compiled into the kernel.
- Communication via shared memory segments. The (BSD) shared memory segments are only used to pass the actual data between the communicating peers. As the mainloop tries to be non-polling, an additional unix domain socket is created to pass synchronizing information about the memory segments. This results in communication via shared memory segments only being more efficient if large amounts of data are to be transmitted.
- Direct communication (if sender and receiver are in the same program). This speciality was implemented to allow the incorporation of communicating peers in one single process. The ‘communication’ happens by `io_write()` putting the data directly into the buffer of the reader and marking this reader as being ready. The reader will then automatically be called when `io_main()` becomes active.
- Generic file-descriptor (as link to a communication peer). The connectivity to other services is granted by allowing the association of a channel with a UNIX file-descriptor, assuming that read and write on the object the descriptor refers to are meaningful. This is how previously established communication links may be hidden under GenIO channels, e.g. to create a ‘stdin/stdout’ channel.

Another provision to reach ‘outside’ services is the ability to provide specifications for the wanted IO-mechanisms and addresses while associating a connection or a listener with a channel.

The currently used mechanisms may easily be extended by (un)reliable UDP links, or links via serial lines, requiring error correction and flow control. Automatic encryption of traffic may be easily added, an automatic key-negotiation (based on D-H or RSA) at the beginning of the communication is possible.

2.4 Priorities for selection of communication mechanisms

While a listener may accept connections on multiple mechanisms simultaneously, and a connection attempt may be performed on more than one communication mechanism, the resulting communication channel employs only one of these mechanisms. A simple algorithm is used to decide which mechanism takes precedence:

If sender and receiver are in the same process, direct is used. If they are on the same machine, and shared memory is available, this is used. If not, Unix Domain Sockets are employed. If these do not work, or the connection endpoints are on different machines, XTPX (if available) or TCP via internet domain sockets is used.

2.5 two samples using GenIO

This chapter contains `iotest.c`, as an example on how GenIO may be used, and `iodirect.c` to visualize the usage of direct channels. `Iotest` realizes a (simple) server-client chat system, allowing the distribution of typed in messages from one client to all of them.

Standard includes:

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
```

```

#include <signal.h>
#include <stdlib.h>
#include "genio.h" /* Genio interface declarations */

```

```

void *l[100]; /* this are all channels (max. 100) */

```

These are the channels the server users to remember the clients. The clients themselves only use `l[0]` and `l[1]` as channel to the server and tty-channel respectively.

```

int cnt=0; /* number of highest valid channel */
void *g=NULL; /* the group-channel */

```

Server routines:

```

int g_write(void *inchannel, char *buf, int len)
{
    printf("from: %p buf: %p len %d\n", inchannel, buf, len);
    if (g) io_write(g, buf, len);
    return len;
}

```

This procedure is installed as a reader for all channels in the server. Anything reaching the server will therefore be distributed to all clients using the group channel `g`.

`g_write()` is called by `io_main()` as soon as data arrives on one of the channels it is installed for. The length of the written data has to be returned to `io_main` to allow emptying its buffer. It is assumed that `io_write()` succeeds in writing everything to everybody.

```

int l_except(void *chan, int action, void *newchan)
{
    int i;
    printf("l_except %p %d %p\n", chan, action, newchan);
    switch (action) {
        case IO_EXCEPT_CONN_NEW:
            /* upointer could be set to cnt, and you could look for the
             * first free entry in the list instead of always increasing
             * cnt
             */
            l[cnt++]=newchan;
            g=io_add_to_group(g, newchan);
            return 0;
        case IO_EXCEPT_CONN_DEAD:
            /* this could use the upointer to know which 'i' is right*/
            for(i=0; i<100; i++) if (l[i]==chan) l[i]=NULL;
            return 0;
        case IO_EXCEPT_ERR_READ:
        case IO_EXCEPT_ERR_WRITE:
            return 0; /* These error conditions are simply ignored */
    }
    return -1;
}

```

`l_except()` is the exception routine which is installed for all channels in the server. Remember that `CONN_NEW` will only occur upon a connect from a new client, and `CONN_DEAD` only for a leaving client.

Client routines:

```

int toserver(void *inchannel, char *buf, int len)
{
    return io_write(l[0], buf, len);
}

```

This is installed as the reader for the tty-channel, so anything reaching the client via keyboard is directly forwarded to the server.

```

int myreader(void *inchannel, char *buf, int len)
{
    io_write(l[1], "-> ", 3);
    io_write(l[1], buf, len);
    return len;
}

```

This procedure is installed as reader on the channel to the server. Thus data coming from the server is directly passed to the channel `l[1]`, the `tty`. The `io_write()`s in this procedure might as well be replaced by `puts()`. The assumption is taken that a `io_write()` to the `tty` does not fail.

```
int c_except(void *chan, int action, void *newchan)
{
    printf("c_except %p %d %p\n",chan,action,newchan);
    switch (action) {
        case IO_EXCEPT_CONN_DEAD:
            printf("Connection closed. exiting...\n");
            io_close_all();
            exit(0);
        case IO_EXCEPT_CONN_NEW:
        case IO_EXCEPT_ERR_READ:
        case IO_EXCEPT_ERR_WRITE:
            return 0;
    }
    return -1;
}
```

The exception-handler for the client does never have to handle new connections.

Routines common to both server and client:

```
void finish(void)
{
    printf("finish\n");
    io_close_all();
    exit(0);
}
```

Installed for some signals, most important on the server side to allow a cleanup of certain sockets in `/usr/tmp`.

```
void quak(void *a,struct timeval *b)
{
    struct timeval etim;
    printf(">>THIS IS QUAK\n");
    gettimeofday(&etim,NULL);
    etim.tv_sec+=15;
    event_store(&etim,quak,NULL);
}
```

This function is calling itself periodically by using the event-mechanism of GenIO.

```
void main(int argc, char *argv[])
{
    int i;
    io_opt *opt=io_get_empty_opt();

    /*opt->wbuf_flush = -1;*/
    /*opt->wbuf_maxdelay=10000000;*/
}
```

Enabling these two lines causes server and client not to flush all data immediately, but to keep it up to ten seconds in the buffer. This would be time-flush, internally using GenIO-events. If these two lines are not enabled, an empty `io_opt`, which should be `free()`d afterwards, is used, consisting of default values, in this case: no options, immediate flush.

```
signal(SIGTERM,finish);
signal(SIGINT,finish);

io_debug(/*IO_DBG_TXT|*/IO_DBG_ERR|IO_DBG_PID,NULL);
```

Enables debugging output for all channels. Although you can specify debugging output to happen only to a few channels, this is not used here. `DBG_TXT` is quite chatty, usually you will not want it.

```
quak(NULL,NULL); /* initialize quak and let it install itself */
```

Client side of `main()`:

```
if (argc>1) {
```

argv[1] contains an ID-string, to which we try to connect.

```
io_addr *ad; /* for the address of the terminal */
l[0]=io_connect_id(argv[1],opt,myreader,c_except);
if (!l[0]) {
    printf("could not connect_id %d %d %s\n",io_errno,
          io_aux_errno,io_err_msg);
    exit(1);
}
cnt++;

ad=io_get_empty_addr();
ad->filedes_in=0; /* stdin */
ad->filedes_out=1; /* stdout */
```

A new (empty) io_addr is allocated, and has to be freed afterwards. It is set to contain the addresses for the tty.

```
l[1]=io_connect_direct(
    IO_MECH_FILEDESC,ad,opt,0,toserver,c_except);
if (!l[1]) {
    printf("could not connect_direct %d %d %s\n",io_errno,
          io_aux_errno,io_err_msg);
    exit(1);
}
free(ad);
cnt++;
```

Now, the two client channels l[0] and l[1] to server and tty respectively have been established.

```
i=io_write(l[1],"Welcome!\n",9);
if (i!=9) {
    printf("io_write %d %d %s\n",
          io_errno,io_aux_errno,io_err_msg);
    exit(1);
}
} else {
```

Server side of main():

```
l[0]=io_create_listener(IO_MECH_ALL,opt,io_write,l_except);
```

If you use this line of code, the input from a client to the server will just be returned to this client. The reason for this is the usage of io_write() as reader. But let's use the group channel g instead:

```
l[0]=io_create_listener(IO_MECH_ALL,opt,g_write,l_except);
if (!l[0]) {
    printf("could not create listener %d %d %s\n",io_errno,
          io_aux_errno,io_err_msg);
    exit(1);
}
cnt++;
```

After creating the listener, the ID-string that is needed for the clients to connect to this listener is output. This is the complete initialization of the server specific part. Now any client from any machine may connect to this server, as by using IO_MECH_ALL, we allow internet-connections to occur.

```
printf("Connect to the server by typing:\n");
printf("%s %s\n",argv[0],io_get_id(l[0]));
}
```

Common to both sides of main():

```
while(1) io_main(cnt,l,NULL);

/* never reached */
}
```

Using `NULL` as timeout value causes `io_main()` to be blocking. Take care that `io_main()` may return quite important error messages, but this simple application may ignore them, as the exception routines handle the important exceptions. Remark that `io_main()` returns after any reader has been executed or an exception has taken place. This allows the new composition of the array `l`, maybe containing new channels, to take effect.

Now `iodirect.c`. This program does nothing really useful, it just passes its `stdin` to its `stdout`:

Standard includes:

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

#include "genio.h"
```

```
void *lis, *io, *lo, *co; /* The direct and filedesc channels */
```

The input passes through the following path: `stdin` -> `io` -> `do_loop()` -> `co` -> `[direct]` -> `lo` -> `io_write()` -> `co` -> `receive()` -> `io` -> `stdout`. This interconnection is done in the respective readers, which are set up in `io_create_listener()` and `io_connect_*`.

```
void finish(int dummy) { printf("finish\n"); io_close_all(); exit(0); }

int exc(void *a, int b, void *c)
{
    switch (b) {
        case IO_EXCEPT_CONN_DEAD:
            finish(0);
        case IO_EXCEPT_CONN_NEW:
            lo=c;
        case IO_EXCEPT_ERR_READ:
        case IO_EXCEPT_ERR_WRITE:
            return 0;
    }
    return -1;
}
```

`IO_EXCEPT_CONN_NEW` will occur only once, to handle `io_connect_id()`.

`IO_EXCEPT_CONN_DEAD` will only be called when `finish` is executed. `io_close_all()` takes care to protect against needless recursion.

```
int do_loop(void *c, char *b, int l){return io_write(co,b,l);}
int receive(void *c, char *b, int l){return io_write(io,b,l);}
```

These are the readers for the incoming end of `io` and the receiving end of the direct link `co` -> `lo` respectively. They imply the interconnection of the channels.

```
void main(void)
{
    io_addr *ad;
    char *id;
    void *z[4];

    signal(SIGTERM,finish);
    signal(SIGINT,finish);

    io_debug(IO_DBG_ERR|IO_DBG_TXT,NULL);

    ad=io_get_empty_addr();
    ad->filedes_in=0; /* stdin */
    ad->filedes_out=1; /* stdout */
    io=io_connect_direct(IO_MECH_FILEDESC,ad,NULL,0,do_loop,exc);

    lis=io_create_listener(IO_MECH_DIRECT,NULL,io_write,exc);
    id=io_get_id(lis);
}
```

```
co=io_connect_id(id,NULL,receive,exc);
```

This causes the execution of `exc()` even without `io_main()` being executed. After `io_connect_id()` both `co` and `lo` exist and are ready. This may be tricky, if the initialisation of your program is not then completed. Thus, take care to create direct channels as last thing before calling `io_main()`.

Data written to `co` is received by `lo`, where it is automatically passed to the ‘reader’ `io_write()` and thus goes back to `co`. `lo <-> co` is the direct channel, employing no transport mechanisms.

```
z[0]=lis;z[1]=io;z[2]=lo;z[3]=co;
while(1) io_main(4,z,NULL);
}
```

2.6 Abstract connection endpoint identifier (Channel-ID)

As the examples in section 2.5 show, `io_connect_id()` receives the address it has to connect to as a character string from the caller. This string describes the mechanisms under which a listener is reachable, and is created by `io_get_id(listener)`. This function may be used only on channels of type `listener`. A typical example would be:

```
GENIO-0.01a:i81844208,32968:d357F0,964:u/tmp/aaaa000F4:s/tmp/baaa000F4:Z6146F18A
```

The ID-string is composed of a header, entries separated by a double colon and pertaining to the individual mechanisms, and a checksum, ending the ID-string. The individual fields consist of:

- `i`: Internet, followed by 32-bit hexadecimal IP number, decimal port number.
- `d`: Direct, followed by memory address of the direct listener, pid.
- `u`: Unix, followed by unix domain socket in the file system.
- `s`: Shared Memory, followed by synchronizing unix domain socket.
- `x`: XTPX, followed by 32-bit hexadecimal IP number, decimal XTPX port number.
- `m`: `want_multicast`, an unused option.
- `Z`: Checksum. For calculation traverse ID-string up to trailing ‘:’ and do:

```
while(*t) a=(a*5)^(*t++);
```

While a nonmatching or missing checksum causes refusal of the connection attempt, a wrong version string just causes a warning to be generated.

When establishing a connection with `io_connect_id()` and (optionally) with `io_connect_direct()` the connecting peer initially sends its channel-ID as string to the incoming channel just spawned from a listener. The peer GenIO would be able to reply, but does not use this possibility currently. This allows the future negotiation of connection options, like passing additional multicast ports etc. If the first input received by any channel does not have the correct header (disregarding version numbers) or an incorrect checksum, it is passed to the reader, and no further interpretation of communication data will be attempted. This allows you to connect to tty’s, X servers or whatever.

2.7 Buffering

GenIO currently allows four different kinds of output data buffering:

- No buffering: After each `io_write()` `io_flush()` is called automatically.
- Unlimited buffering: All data is buffered locally until `io_flush()` is called by the application. Unlimited buffering is in reality limited by the available memory and swap space. If they are exhausted, an error is returned for future `io_write()`s. Warning: Currently GenIO will just reference a `NULL`-pointer.
- Buffering limited to a certain amount of data: After each `io_write()` the current buffer size is checked. If it exceeds a certain limit, `io_flush()` is called.
- Any of the above combined with delay-triggered flushing: Additionally, a time interval may be specified (in μsec) after which all currently unflushed data is flushed. This interval is reset each time the first write after a flush occurs.

Input data is buffered by GenIO only until the reader has been called, unless the reader does not consume (all) the data passed to it. In this case, it will be buffered and prepended to any new data that might arrive.

2.8 Exception handler and reader

As the exception handlers and readers for listeners and data channels have to strongly interact with GenIO, it is important to understand the interconnection of these application procedures with the library. The handlers may call GenIO procedures, e.g. to `io_close()` the actual channel, but they may neither call `io_main()`, nor cause the same exception or reader to be called again. In such a case the offending action is cancelled and an error code returned.

Exception handlers (`int except(void *oldchannel, int event, void *newchannel)`) are called when anything special occurs, they receive the active channel, an opcode and eventually other data when invoked. The following opcodes exist:

- `CONN_NEW`: The third argument passed to the exception handler contains the channel descriptor of the newly spawned channel, while the first argument contains the listener. The new channel is ready to be used, the reader has been inherited from the listener, but has not yet been called.

Returning 0 indicates a successful completion of the exception handler, a value <0 indicates an error, which will be passed up to the caller of `io_main`. This does not cause the channel to be closed, you have to close it yourself. The channel may be closed in the exception handler.

This opcode is only passed for listeners accepting a connection, never for a newly created outgoing channel.

- `CONN_DEAD`: Occurs when the communicating peer uses `io_close()`, `io_abort()` or the communication link is somehow interrupted. The third argument is always `NULL`, the channel in question is passed as first argument. You may close the channel in the exception handler. If you do not do it, it will be done by the library upon returning from your exception handler.

Take care to reset any upointer defined for this channel, as GenIO checks this before closing the channel.

Returning 0 indicates a successful completion of the exception handler, a value <0 indicates an error, which will be passed up to the caller of `io_main`.

- `CONN_EXCEPT`: Is called if `select()` receives an exception (out-of-band data) on any socket associated with a channel. Out-of-band data is not used by GenIO, you would have to retrieve the file descriptor for this socket by looking into GenIO-internal data structures and handle it yourself.
- `ERR_READ`: If a `read()` on a file descriptor returns an error, this is invoked for the associated channel. A return value of 0 causes the error to be ignored, otherwise the negative error number is passed up to the caller of `io_main()`. Positive return values cause the UNIX `errno` to be passed back as `io_aux_errno` of GenIO.
- `ERR_WRITE`: If a `write()` on a file descriptor fails, this is invoked for the associated channel. A return value of 0 causes the error to be ignored, all unwritten data is discarded. Positive return values cause the UNIX `errno` to be passed back as `io_aux_errno` of GenIO.

Readers (`int reader(void *channel, char *buf, int len)`) are called if there is data to be read for a channel. They receive the channel, a pointer to a buffer, and the number of characters waiting in this buffer. The 0-termination of the buffer is guaranteed.

The reader has to return the number of characters that are to be removed from the buffer. If 0 is returned, the reader will not be called until new characters arrive or `io_restart_read()` is invoked. Unread data is kept and prepended to new data. If a value <0 is returned, this is an 'Internal reader error' which goes back up to the caller of `io_main()`. At the same time, the buffer is flushed. The reader may call `io_close()` on any channel.

2.9 User defined events

As addition to the IO-related topics above, the GenIO-library offers another service. Assuming that the application is written in event-driven fashion, the 'main-loop' of the application is within GenIO. To allow execution of other parts of the application depending on timer events, appropriate queues exist and functions registered in there are automatically called.

The events are guaranteed not to be processed earlier than requested, and will be treated the first time `io_main()` or `event_ripe_handle_time()` is called and no earlier events are pending. Execution time is specified in absolute time, with the same resolution as it is offered by the operating system to process the `select()` system call.

3. GenIO for the SUN platform

This section mentions the internals and peculiarities of the GenIO implementation for UNIX. The actual implementation is known to work under SunOS 4.1.4 and Solaris 2.3.

All IO mechanisms have somehow been mapped to file descriptors, on which a `select()` is performed, to refrain from a polling behaviour. This is also true for shared

memory segments (BSD semantics and `mmap()` is employed), where an additional unix domain socket exists.

Principally, GenIO should perform asynchronous IO. In reality this is not true. Connection establishment of TCP sockets is blocking. This is due to a strange behaviour of Suns when connecting to a MacIntosh, causing EPIPE to occur. Writes are also blocking, as GenIO only returns from a `io_flush()` or a flushing `io_write()` if all data has been written to the corresponding descriptor, memory segment. This is mainly visible when writing to group channels, as the data is written sequentially to each channel in the group.

Upon using `IO_MECH_FILEDESC` you should take care that the referenced descriptors are not `O_NDELAY`. Upon closing a channel, the corresponding file descriptors are not closed. This has to be done by the application. `robleme` mit `fd`. GenIO tries to remember the old `fcntl()` state on filedescriptors, which is reset after `io_close()`, but this may fail on Solaris 2.3.

Beside the possibility to switch on debugging messages for each channel separately by using `io_debug()`, debugging messages and the behaviour of GenIO may be influenced by the setting of environment variables. The following variables are currently known: `GENIO_DEBUG`, `GENIO_VERSION`, `GENIO_HELP` and `GENIO_NO{XTPX, DIRECT, SHMEM, UNIX}`. `GENIO_DEBUG` takes the numeric `IO_DEBUG_*` flag values as arguments.

When an error occurs during a write on a group channel, the state of the recipients might be unequal, Some might have received the data, some not.

The reader receives an buffer which was aligned by `malloc()` at the first call. If not all data is read, the reader has to take care about boundaries of subsequent packets.

The following compiler switches are defined in the `Makefile`: `DEBUG`, `USE_WARN`, `SOLARIS` and `HAVE_GOOD_PROTOTYPES`. `NDEBUG` may be defined to disengage the `assert()`s. Warnings (if enabled but the compiler switch) are written to `stderr` upon significant events like recursion in a reader, incompatible version strings, etc. The Solaris define mainly switches of `fcntl()`s. The prototype define is to be used on not-braindamaged SunOS installations, where prototypes are available.

Warning: In `genio.h` the size of the `io_opt` and `io_state` structures vary with the definition of `DEBUG`. Take care to use the same case in both your application and GenIO.

Before each `write()` a `signal(SIGPIPE, SIG_IGN)` is issued (the signal is restored after the `write()`). If you do not otherwise need this signal, you may set it to `SIG_IGN` yourself before calling the first `io_write()`, and call `io_control(*NOSIGPIPE)`, to switch this behaviour of. This increases the performance of GenIO. No other signals are ever modified by GenIO.

All interface declarations are in `genio.h`. The implementation of the interface is in `main.c` and `event.c` for the event routines. Thus all functions except event handling are called through `main.c`. Each communication mechanism has its own source module, in which 'lowlevel' functions are implemented. They use their own small interface, declared in the respective `.h` files.

All internal data structures are defined in `defs.h`. You may include this, if you want e.g. to access the raw file descriptors, or twiddle with the buffering mechanisms. This is NOT recommended. All channels use the `typedef chan`. `chan_anchor` allows you to traverse all existing channels, `event_anchor` all events. The auxiliary error numbers are also declared in this file.

For instructions on the functions see `<genio.h>`

4. GenIO for the MAC platform

This section mentions the internals and peculiarities of the GenIO implementation for the Mac. The actual implementation is known to work under System 7.1, and was written by Hasan.

There have been a few changes to the GenIO interface, due to the double mainloop problem described below, and the implementation of GenIO was mostly rewritten.

The communication mechanisms `BYPASS` and `PPC` have been added. The first one is a speciality needed to serve the internet port 6000 to the pseudoserver, the second one is used to support the Mac-specific process-to-process communication, used internally between pseudo server, pseudo client and QuiX.

`io_init()` and `io_exit()` have been added to allow the correct initialization and the releasing of allocated resources upon program completion. The parameter `complt` has been added to `io_flush()`, allowing a completion routine to be called when memory is to be freed. `io_get_buff_state()` returns the number of unread and unwritten bytes of a channel. The event routines are missing.

On the Mac, the crucial problem with GenIO lies in the fact, that it is locked into interrupt routines, coming from MacTCP. While these routines are running, no `malloc()` is possible. This caused changes to the buffering strategy of GenIO. On the other side, MacX, which is used by the pseudoclient, tries to keep as much as possible of the control flow to itself. So care had to be taken, when to announce to MacX that data is ready. The solution was mostly to uncouple the readers from the upcall of MacTCP, so memory allocation would only be done when being called from above, not when upcalls are active.