xisoftware

Xi-Text Release 23

API Reference Manual

Table of Contents

1 Introduction to Xi-Text API **4**

2 Installation and access to API **5**

3 The API file descriptor **6**

3.1 Error return codes 6

4 Slot numbers **8**

5 Sequence numbers **9**

6 API Functions **10**

6.1 Sign-on and off 10

6.1.1 xt_open 10

6.1.1.1 Return values 12

6.1.1.2 Example 12

6.1.2 xt_close 12

6.2 Job operations 12

6.2.1 xt_joblist 12

6.2.2 xt_jobread 13

6.2.3 xt_jobfind 17

6.2.4 xt_jobdata(Unix and Linux) 17

6.2.5 xt_jobdata (Windows version) 18

6.2.6 xt_jobpbrk(Unix and Linux versions) 19

6.2.7 xt_jobpbrk(Windows version) 20

6.2.8 xt_jobadd (Unix and Linux versions) 20

6.2.9 xt_jobadd (Windows version) 22

6.2.10 xt_jobdel 23

6.2.11 xt_jobupd 23

6.2.12 xt_jobmon (Unix and Linux versions) 24

6.2.13 xt_jobmon (Windows version) 25

6.3 Printer operations 25

6.3.1 xt_ptrlist 25

6.3.2 xt_ptrread 26

6.3.3 xt_ptrfind 28

6.3.4 xt_ptradd 29

6.3.5 xt_ptrdel 30

6.3.6 xt_ptrupd 30
6.3.7 xt_ptrop 31
6.3.8 xt_ptrmon 32
6.4 User permissions 33
6.4.1 xt_getspu 33
6.4.2 xt_getspd 35
6.4.3 xt_putspu 36
6.4.4 xt_putspd 37

7 Example API program **39**

Chapter 1

Introduction to Xi-Text API

The **Xi-Text** API enables a C or C++ programmer to access **Xi-Text** facilities directly from within an application. The application may be on a Unix host or on a Windows workstation.

Communication takes place using a TCP connection between the API running on a Windows or Unix machine and the server process xtnetserv running on the Unix host in question. The same application may safely make several simultaneous conversations with the same or different host.

The user may submit, change, delete and alter the state of jobs or printers to which he or she has access, and may receive notification about changes which may require attention. In addition, the user access control parameters may be viewed and if permitted, changed.

Chapter 2

Installation and access to API

The API is provided as two files, a header file `xtapi.h` and a library file.

The header file should be copied to a suitable location for ready access. On Unix systems we suggest that the header file is copied to the directory `/usr/include/xi` so that it may be included in C programs via the directive:

```
#include <xi/xtapi.h>
```

The library file is supplied in the form `libxtapi.a` or as a shared library `libxtapi.so` on Unix systems. This should be copied to `/lib` or `/usr/lib` so that it may be linked with the option `-lxtapi` when the program is compiled. On some systems you may have to include a socket handling library as well. The shared library is usually placed in `/usr/lib/xi`.

On Windows systems the library is supplied as `xtapi.dll`. Again we suggest that it be placed in the default search path.

Chapter 3

The API file descriptor

Each routine in the API uses a file descriptor to identify the instance in progress. This is an integer value, and is returned by a successful call to the xt_open or one of the equivalent routines.

All other routines, apart from job string manipulation routines, take this value as a first parameter. As mentioned before, more than one session may be in progress at once with different xt_open parameters.

Each session with the API should be commenced with a call to xt_open or one of the variant routines and terminated with a call to xt_close.

3.1 Error return codes

Nearly all the routines return an integer response code. This is usually zero to indicate success (except for xt_open which returns a positive or zero file descriptor). The error codes are described below.

Those routines which return a pointer to a FILE structure return NULL on error and put the error code in `xtapi_dataerror`.

Code	Name	Meaning
0	XTAPI_OK	No error, successful completion
-1	XTAPI_INVALID_FD	The file descriptor argument is invalid.
-2	XTAPI_NOMEM	Run out of memory allocation within API library.
-3	XTAPI_INVALID_HOSTNAME	Invalid host name in xt_open
-4	XTAPI_INVALID_SERVICE	Invalid service name in xt_open
-5	XTAPI_NODEFAULT_SERVICE	Default service relied upon no default API service set up
-6	XTAPI_NOSOCKET	Cannot create socket
-7	XTAPI_NOBIND	Cannot bind address to socket
-8	XTAPI_NOCONNECT	Connection refused by server
-9	XTAPI_BADREAD	Read error on socket
-10	XTAPI_BADWRITE	Write error on socket
-11	XTAPI_CHILDPROC	Cannot fork to make child process
-23	XTAPI_UNKNOWN_USER	User invoking API is unknown on server
-24	XTAPI_ZERO_CLASS	Class code is effectively zero
-25	XTAPI_BAD_PRIORITY	Invalid priority (outside permitted range)
-26	XTAPI_BAD_COPIES	Invalid number of copies (above limit)
-27	XTAPI_BAD_FORM	Invalid form type (user is restricted)
-28	XTAPI_NOMEM_QF	No memory for queue file on server
-29	XTAPI_BAD_PF	Cannot open page file
-30	XTAPI_NOMEM_PF	No memory for page file on server
-31	XTAPI_CC_PAGEFILE	Cannot create page file
-32	XTAPI_FILE_FULL	Server file system is full
-33	XTAPI_QFULL	Server message queue full
-34	XTAPI_EMPTYFILE	Job file is empty
-35	XTAPI_BAD_PTR	Invalid printer name (user restricted)
-36	XTAPI_WARN_LIMIT	Job exceeds limit, truncated
-37	XTAPI_PAST_LIMIT	Job exceeds limit, not queued
-38	XTAPI_NO_PASSWD	Password required and not given
-39	XTAPI_PASSWD_INVALID	Invalid password
-40	XTAPI_UNKNOWN_COMMAND	Unknown API operation (error in library)
-41	XTAPI_SEQUENCE	Sequence error, operation(s) since last read
-42	XTAPI_UNKNOWN_JOB	Job not found
-43	XTAPI_UNKNOWN_PTR	Printer not found
-44	XTAPI_NOPERM	No privilege for operation
-45	XTAPI_NOTPRINTED	Job has not been printed
-46	XTAPI_PTR_NOTRUNNING	Printer not running
-47	XTAPI_PTR_RUNNING	Printer is running
-48	XTAPI_PTR_NULL	Null printer name
-49	XTAPI_PTR_CDEV	No permission to change device
-50	XTAPI_INVALIDSLOT	Invalid slot number

Chapter 4

Slot numbers

Each job or printer is identified to **Xi-Text** by means of two numbers:

1. The host or network identifier. This is a long corresponding to the internet address in network byte order. The host identifier is given the type `netid_t`.
2. The shared memory offset, or slot number. This is the offset in shared memory on the relevant host of the job or printer and stays constant during the lifetime of the job or printer. The type for this is `slotno_t`.

These two quantities uniquely identify any job or printer.

It might be worth noting that there are two slot numbers relating to a remote job or printer.

1. The slot number of the record of the job or printer held in local shared memory. This is the slot number which will in all cases be manipulated directly by the API.
2. The slot number of the job on the owning host. This is in fact available in the job structures as the field `apispq_rslot` and in the printer structure as the field `apispp_rslot`.

These fields usually have the same value as the slot number in local memory for local jobs or printers, but this should not be relied upon.

Chapter 5

Sequence numbers

These quantities are not available directly, but are held to determine how out-of-date the user's record of jobs or printers may be.

Every time you read a job or printer record, the sequence number of the job or printer list is checked, and if out-of-date, you will receive the error `XTAPI_SEQUENCE`. This is not so much of an error as a warning. If you re-read the job or printer required, then you will not receive this error.

If you want to bypass this, you can access the job or printer without worrying about the sequence using the flag `XTAPI_FLAG_IGNORESEQ`, however you might receive an error about unknown job or printer if the job or printer has disappeared.

Chapter 6

API Functions

The following sub-sections describe the **Xi-Text** API C routines including each function's purpose, syntax, parameters and possible return values.

The function descriptions also contain additional information that illustrate how the function can be used to carry out tasks.

In some cases there are slight differences between the Unix and Windows variants, these are noted where appropriate.

6.1 Sign-on and off

6.1.1 xt_open

```
int xt_open(const char *hostname,
          const char *servname,
          const classcode_t classcode)

int xt_open(const char *hostname,
          const char *servname,
          const char *username,
          const classcode_t classcode) /* Windows */

int xt_login(const char *hostname,
          const char *servname,
          const char *username,
          char *passwd,
          const classcode_t classcode)

int xt_wlogin(const char *hostname,
          const char *servname,
          const char *username,
          char *passwd,
          const classcode_t classcode)

int xt_locallogin(const char *servname,
          const char *username,
          const classcode_t classcode)

int xt_locallogin_byid(const char *servname,
          const int ugid_t uid,
          const classcode_t classcode)
```

The function xt_open is used to open a connection to the **Xi-Text** API. There are some variations in the semantics depending upon whether the caller is known to be a Unix host or a Windows or other client. This can be controlled by settings in the servers host file, typically `/etc/Xitext-hosts` and the user map file `/etc/xi-user.map`.

The server will know that the caller is a Unix host if it appears in the hosts file as a potential server, maybe with a `manual` keyword to denote that it shouldn't be connected unless requested (with spconn). In such cases user names will be taken as Unix user names.

In other cases the user names will be taken as Windows Client user names to be mapped appropriately.

Windows user names are mapped on the server to Unix user names using the user map file and constructs in the host file, with the latter taking priority.

Note that it is possible to use a different set of passwords on the server from the users' login passwords, setting them up with xipasswd. This is desirable in preference to people's login passwords appearing in various interface programs.

All of these functions return non-negative (possibly zero) on success, this should be quoted in all other calls.

In the event of an error, then a negative error code is returned as described on page **??**.

xt_open may be used to open a connection with the current effective user id on Unix systems, or (using the extra `username` parameter a predefined connection for the given user on Windows systems.

No check takes place of passwords for Unix connections, but the call will only succeed on Windows systems if the client has a fixed user name assigned to it.

This happens if the client matches entries in `/etc/Xitext-hosts` of the forms:

```
mypc - client(unixuser)
unixuser winuser clienthost(mypc)
```

The call will succeed in the first instance if the user is mapped to `unixuser` and running on `mypc`.

In the second case it will succeed if it is running on `mypc` and `winuser` is given in the call, whereupon it will be mapped to `unixuser`.

This is over-complicated, potentially insecure, and preserved for compatibility only, and xt_open should only really be used on Unix hosts to log in with the effective user id.

xt_login should normally be used to open a connection to the API with a username and password. If the client is not registered as a Unix client, then the user name is mapped to a user name on the server as specified in the user map file or the hosts file. The password should be that for the user mapped to (possibly as set by xipasswd rather than the login password).

xt_wlogin is similar to xt_login, but guarantees that the user name will be looked up as if the caller were not registered as Unix client so that there are no surprises if this is changed.

xt_locallogin and xt_locallogin_byid may be used to set up an API connection on the same machine as the server without a password. The username, if not null, may be used to specify a user other than that of the effective user id. To use a user other than the effective user id, *Write Admin* permission is required.

In all cases, hostname is the name of the host being connected to or null to use the loopback interface. servname may be NULL to use a standard service name, otherwise an alternative service may be specified. Note that more than one connection can be open at any time with various combinations of user names and hosts.

All functions take a *classcode* which is "anded" with the calling user's classcode unless the user has *override class* permission. The resulting classcode must not be all zeroes, however an argument of zero will be replaced by the user's default classcode. This may be used to limit the list of jobs or printers reported by xt_joblist and xt_ptrlist.

When finished, close the conection with a call to xt_close.

6.1.1.1 Return values

The function returns a positive value if successful, which is the file descriptor used in various other calls, otherwise one of the error codes listed on page **??** onwards, all of which are negative.

6.1.1.2 Example

```
int fd;
fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0)  { /* handle error */
    ...
}
        ...
xt_close(fd)
```

6.1.2 xt_close

```
int xt_close(const int fd)
```

The xt_close function is used to close a connection to **Xi-Text**. Fd is a file descriptor previously returned by a successful call to xt_open or variant routines.

xt_close returns 0 if successful or XTAPI_INVALID_FD (Invalid File descriptor, a constant defined in xtapi.h) if the passed file descriptor was not valid, perhaps because it was never opened successfully.

6.2 Job operations

6.2.1 xt_joblist

```
int xt_joblist(const int fd,
               const unsigned flags,
               int *numjobs,
               slotno_t **slots)
```

The xt_joblist function is used to obtain a list of jobs.

Fd is a file descriptor previously returned by a successful call to xt_open or the parallel routines to open a connection.

Flags is zero, or a bitwise OR of one or more of the following values

XTAPI_FLAG_LOCALONLY	Ignore remote printers/hosts, i.e. not local to the server, not the client.
XTAPI_FLAG_USERONLY	Ignore other users jobs

Numjobs is a pointer to an integer value which, on successful completion, will contain the number of job slots returned.

Slots is a pointer to to an array of slot numbers. These slot numbers can be used to access individual jobs. The memory used by this vector is owned by the API, therefore no attempt should be made by the user to free it. This contrasts, for example, with X library routines. Also note that certain other calls to the API, notably xt_ptrlist, with the same value of fd, may reuse the space, so the contents should be copied if required before other API calls are made.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to list all jobs:

```
int fd, ret, nj, i;
slotno_t *slots;
fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
      ...
}
ret = xt_joblist(fd, 0, &nj, &slots);
if (ret < 0) { /* error handling */
     ...
}
for (i = 0; i < nj; i++) {
     slotno_t this_slot = slots[i];
     /* process this_slot */
     ...
}
xt_close(fd);
```

6.2.2 xt_jobread

```
int xt_jobread(const int fd,
               const unsigned flags,
               const slotno_t slot,
               struct apispq *jobd)
```

The xt_jobread function is used to retrieve the details of a job from a given slot number.

`fd` is a file descriptor previously returned by xt_open or the equivalent routines.

`flags` is zero, or a bitwise OR of one or more of the following values

`XTAPI_FLAG_LOCALONLY`	Ignore remote printers/hosts, (from the point of view of the server, not the client).
`XTAPI_FLAG_USERONLY`	Ignore other users jobs
`XTAPI_FLAG_IGNORESEQ`	Ignore changes since the list was last read

`slot` is the slot number corresponding to the job as returned by xt_joblist or xt_jobfindslot.

`Jobd` is a descriptor, which on return will contain the details of the job in a `struct apispq` as defined in `xtapi.h` and containing the following elements:

Type	Field	Description
`jobno_t`	`apispq_job`	Job number
`netid_t`	`apispq_netid`	Host address (network byte order)
`netid_t`	`apispq_orighost`	Originating host address
`slotno_t`	`apispq_rslot`	Slot number on owning machine
`time_t`	`apispq_time`	Time job was submitted
`time_t`	`apispq_starttime`	Time job was started (if applicable)
`time_t`	`apispq_hold`	Time job held to, 0 if not held
`unsigned short`	`apispq_nptimeout`	Time after to delete job if not printed (hours)
`unsigned short`	`apispq_ptimeout`	Time after to delete job if printed (hours)
`unsigned short`	`apispq_extrn`	External job type index
`unsigned short`	`apispq_pglim`	Job size limit applies
`long`	`apispq_size`	Size of job in bytes
`long`	`apispq_posn`	Offset reached if currently being printed
`long`	`apispq_pagec`	Currently-reached page if being printed
`char[]`	`apispq_uname`	User name of job owner
`char[]`	`apispq_puname`	User name of posting user
`unsigned char`	`apispq_cps`	Copies
`unsigned char`	`apispq_pri`	Priority
`classcode_t`	`apispq_class`	Class code bits 1=A 2=B 4=C etc
`unsigned short`	`apispq_jflags`	Job flags
`unsigned char`	`apispq_dflags`	Despooler flags
`slotno_t`	`apispq_pslot`	Printer slot assigned to if printing
`unsigned long`	`apispq_start`	Start page 0=first page
`unsigned long`	`apispq_end`	End page
`unsigned long`	`apispq_npages`	Number of pages
`unsigned long`	`apispq_haltat`	“Halted at” page
`char []`	`apispq_file`	Job title
`char []`	`apispq_form`	Job form type
`char []`	`apispq_ptr`	Printer pattern assigned to job
`char []`	`apispq_flags`	Post-processing flags

The following bits are set in the `apispq_jflags` field to indicate job parameters:

Bit (#define)	Meaning
APISPQ_NOH	Suppress header
APISPQ_WRT	Write result
APISPQ_MAIL	Mail result
APISPQ_RETN	Retain on queue after printing
APISPQ_ODDP	Suppress odd pages
APISPQ_EVENP	Suppress even pages
APISPQ_REVOE	Invert APISPQ_ODDP and API_EVENP after printing
APISPQ_MATTN	Mail attention
APISPQ_WATTN	Write attention
APISPQ_LOCALONLY	Handle job on local machine only
APISPQ_CLIENTJOB	Job originated with windows client
APISPQ_ROAMUSER	Job originated with DHCP windows client

The apispq_dflags field contains the following bits:

Bit (#define)	Description
APISPQ_PQ	Job being printed
APISPQ_PRINTED	Job has been printed
APISPQ_STARTED	Job has been started
APISPQ_PAGEFILE	Job has a page file
APISPQ_ERRLIMIT	Error if size limit exceeded
APISPQ_PGLIMIT	Size limit in pages not KB

Note that the field apispq_pglim and the field bits APISPQ_ERRLIMIT and APISPQ_PGLIMIT will always be zero when read, but the description is included for completeness. The fields are only used when creating jobs.

The function returns 0 if successful otherwise one of the error codes as error codes as listed in chapter 3.

An example to read the names of all jobs

```
int fd, ret, nj, i;
struct apispq job;
slotno_t *slots;

fd = xt_open("myhost", (char *)0, 0);
if (fd < 0) { /* error handling */
    ...
}

ret = xt_joblist(fd, 0, &nj, &slots);
if (ret < 0) { /* error handling */
    ...
}

for (i = 0; i < nj, i++) {
    ret = xt_jobread(fd, 0, slots[i], &job);
    if (ret < 0) { /* error handling */
        ...
    }
    printf("%s\n", job.apispq_file);
}

xt_close(fd);
```

6.2.3 xt_jobfind

```
int xt_jobfind(const int fd,
               const unsigned flags,
               const jobno_t jobnum,
               const netid_t nid,
               slotno_t *slot,
               struct apispq *jobd)

int xt_jobfindslot(const int fd,
                   const unsigned flags,
                   const jobno_t jobnum,
                   const netid_t nid,
                   slotno_t *slot)
```

The xt_jobfind and xt_jobfindslot functions may be used to find a job from a given job number rather than by the slot number. xt_jobfind retrieves the job descriptor, xt_jobfindslot just retrieves the slot number.

fd is a file descriptor previously returned by xt_open or the equivalent routines.

flags is zero, or a bitwise OR of one or more of the following values

XTAPI_FLAG_LOCALONLY Ignore remote printers/hosts, (from the point of view of the server, not the client).

XTAPI_FLAG_USERONLY Ignore other users jobs

jobnum is the job number to be searched for.

nid is the network-byte order IP address of the host of the machine whose job is to be searched for. This should be correct even if XTAPI_FLAG_LOCALONLY is specified.

slot is a pointer to a location in which the slot number of the job is placed if the search is successful. It may be NULL if this is not required (but this would be almost pointless for xt_jobfindslot).

jobd is a descriptor containing the job descriptor as defined in xtapi.h.

The fields in struct apispq are defined in the xt_jobread documentation on page 14.

The functions return 0 if successful otherwise one of the error codes as listed in chapter 3.

6.2.4 xt_jobdata(Unix and Linux)

```
FILE *xt_jobdata(const int fd,
                 const unsigned flags,
                 const slotno_t slotno)
```

The function xt_jobdata is used to retrieve the job file of a job.

`fd` is a file descriptor previously returned by xt_open or the equivalent routines.

`flags` is zero, or `XTAPI_FLAG_IGNORESEQ` to ignore changes since the job list was last read.

`slotno` is the slot number corresponding to the job previously returned by functions such as xt_joblist or xt_jobfindslot.

The result is a `FILE` pointer which can be used with all standard I/O input functions such as fgets(3), getc(3) etc. At the end of the data fclose(3) must be called. For reasons of synchronisation the file should be read to the end before other operations are attempted.

If an error is detected, xt_jobdata returns NULL and an error code is placed in the external variable `xtapi_dataerror`. This will be one of the error codes as listed in chapter 3.

An example to retrieve the data for a job:

```
int      fd, ret, ch;
slotno_t slot, *list;
FILE *inf;
fd = xt_open("myhost", (char *) 0, 0);
if   (fd < 0) { /* error handling */
    ...
}
/* Select a job slot and assign this to "slot" */
    .......
inf = xt_jobdata(fd, 0, slot);
if (!inf)  { /* handle errors */
    ...
}
while ((ch = getc(inf)) != EOF)
     putchar(ch);
fclose(inf);
xt_close(fd);
```

6.2.5 xt_jobdata (Windows version)

```
int xt_jobdata(const int fd,
               const int outfile,
               int (*func)(int, void*, unsigned),
               const unsigned flags,
               const slotno_t slotno)
```

This format of the xt_jobdata function is for use by Windows programs, as there is no acceptable equivalent of the pipe(2) construct.

The second argument `outfile` is (possibly) a file handle to the file to which the job data is passed as the first argument to `func`.

The third argument `func` is a function with the same specifications as write, indeed it may very well be write. The main reason for doing it this way is that some versions of Windows do strange things if write is invoked from within a DLL.

Other aspects of the interface are similar to the Unix routine, apart from the routine returning zero for success and an error code for failure rather than a `FILE*` or NULL. For consistency with the Unix version, the external variable `xtapi_dataerror` is also assigned any error code returned. This will be one of the error codes as listed in chapter 3.

6.2.6 xt_jobpbrk(Unix and Linux versions)

```
FILE *xt_jobpbrk(const int fd,
                 const unsigned flags,
                 const slotno_t slotno)
```

The function xt_jobpbrk is used to retrieve the page break offset file of a job.

`fd` is a file descriptor previously returned by xt_open or the equivalent routines. `Flags` is zero, or `XTAPI_FLAG_IGNORESEQ` to changes since the job list was last read.

`slotno` is the slot number corresponding to the job previously returned by functions such as xt_joblist or xt_jobfindslot.

The result is a `FILE` pointer which can be used with all standard I/O input functions such as fread(3), fgets(3), getc(3) etc. At the end of the data fclose(3) must be called. For reasons of synchronisation the file should be read to the end before other operations are attempted.

If an error is detected, xt_jobpbrk returns NULL and an error code is placed in the external variable `xtapi_dataerror`. This will be one of the error codes as listed in chapter 3.

If there is no page offset file, probably because the delimiter is set to formfeed, then this isn't really an error, but an error report of `XTAPI_BAD_PF` will be returned. You can tell whether there is a page file from the `struct apispq` job structure returned by xt_jobread or xt_jobfind. The field `apispq_dflags` has the bit designated by `APISPQ_PAGEFILE` set if there is a page file.

The data is returned in three parts.

1. `struct apipages` This is an instance of the following structure, defined in `xtapi.h`, and described below.
2. `delimiter string` This is the delimiter string itself,
3. A vector of longs giving the offsets of the start of each page, including the first page, which is always zero, within the job data (as read by xt_jobdata).

The `struct apipages` structure is as follows:

```
struct apipages {
    long delimnum;   /* Number of delimiters */
    long deliml;     /* Length of delimiter string */
    long lastpage;   /* Number of delimiters remaining on last page */
};
```

6.2.7 xt_jobpbrk(Windows version)

```
int xt_jobpbrk(const int fd,
               const int outfile,
               int (*func)(int, void*, unsigned),
               const unsigned flags,
               const slotno_t slotno)
```

This second format of the xt_jobpbrk function is for use by Windows programs, as there is no acceptable equivalent of the pipe(2) construct.

The second argument outfile is (possibly) a file handle to the file from to which the job data is passed as the first argument to func.

The third argument func is a function with the same specifications as write, indeed it may very well be write. The main reason for doing it this way is that some versions of Windows do strange things if write is invoked from within a DLL.

Other aspects of the interface are similar to the Unix routine, apart from the routine returning zero for success and an error code for failure rather than a FILE* or NULL. For consistency with the Unix version, the external variable xtapi_dataerror is also assigned any error code returned.

6.2.8 xt_jobadd (Unix and Linux versions)

```
FILE *xt_jobadd(const int fd,
                struct apispq *jobd,
                const char *delim,
                const unsigned deliml,
                const unsigned delimnum)

int xt_jobres(const int fd,
              jobno_t *jobno)
```

The functions xt_jobadd and xt_jobres are used to add a job under Unix and Linux.

fd is a file descriptor previously returned byxt_open or the equivalent routines.

jobd is a pointer to a struct apispq, as defined in xtapi.h and on page 14 containing all the details of the job. The fields in struct apispq are defined in there.

Note that we recommend that the whole structure be cleared to zeroes initially and then required fields added; this approach will cover any future extensions with additional fields which will behave as at present if zero.

Also note that from release 23 an additional field is provided in the structure. If this is non-zero, then the size of the job is limited. If the bit APISPQ_PGLIMIT in apispq_dflags is zero, then the size is limited to the given number of kilobytes. If this bit is set, then the size is limited to the given number of pages. If a job exceeds the given limit, then its treatment depends upon the setting of the bit APISPQ_ERRLIMIT in apispq_dflags. If this is zero, then the job is truncated to the given number of kilobytes or pages and still proceeds (although a warning code is returned by xt_jobres. If it is set, then it is rejected altogether.

delim is a pointer to a string containing the page delimiter string, or NULL if the user is content with the single formfeed character. deliml is the length of the delimiter string delim. This is necessary because delim is not necessarily null-terminated.

delimnum in the number of instances of the delimiter string/character to be counted to make up a page.

The result is either a standard I/O stream, which can be used as output for putc(3), fprintf(3), fwrite(3) etc, or NULL to indicate an error has been detected. The I/O stream connection should be closed, when complete, with fclose(3). Finally a call should be made to xt_jobres.

For reasons of synchronisation you must call xt_jobres immediately after fclose(3) even if you are not interested in the answer. Apart from that several calls to xt_jobadd may be in progress at once to submit several jobs simultaneously.

xt_jobres returns zero on successful completion (or XTAPI_WARN_LIMIT if the job was truncated but still submitted). The parameter jobno is assigned the job number of the job created. This value is also assigned to the field apispq_job in the passed structure jobd to xt_jobadd.

Note that you should not call xt_jobres if xt_jobadd returns NULL for error. Most errors are detected at the xt_jobadd stage and before any data is passed across, but this should not in general be relied upon.

An example to add a job called readme from standard input:

```
int fd, ret, ch;
struct apispq outj;
jobno_t jn;
FILE *f;

fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}

/* It is safest to clear the structure first */
memset((void *) &outj, '\0', sizeof(outj));

/* set defaults */

outj.apispq_nptimeout = 24 * 7;
outj.apispq_ptimeout = 24;
outj.apispq_cps = 1;
outj.apispq_pri = 150;

/* The class code specified in xt_open is not used here.  However the
   user's class code will be &ed with this unless the user has
   override class privilege.  */

outj.apispq_class = 0xffffffff;

/* set a large page range to to ensure all pages are printed */

outj.apispq_end = 4000;

/* Only the form type is compulsory here.  The others may
   be set to NULL */
```

```
strcpy(outj.apispq_file, "readme");
strcpy(outj.apispq_form, "a4");
strcpy(outj.apispq_ptr, "laser");

/* add the job with the default page delimiter */

f = xt_jobadd(fd, &outj, (char *) 0, 1, 1);
if (!f)  { /* error handling error in xtapi_dataerror */
    ...
}

/* now send the data */

while ((ch = getchar()) != EOF)
    putc(ch, f);
fclose(f);

ret = xt_jobres(fd, &jn);
if (ret < 0) { /* error handling */
    ...
} else
    printf("success the job number is %ld\n", jn);
xt_close(fd);
```

6.2.9 xt_jobadd (Windows version)

```
int xt_jobadd(const int fd,
              const int infile,
              int (*func)(int, void*, unsigned).
              struct apispq *jobd,
              const char *delim,
              const unsigned deliml.
              const unsigned delimnum)
```

This second format of the xt_jobadd function is for use by Windows programs, as there is no acceptable equivalent of the pipe(2) construct.

The second argument infile is (possibly) a file handle to the file from which the job is created and is passed as the first argument to func.

The third argument func is a function with the same specifications as read, indeed it may very well be read. The main reason for doing it this way is that some versions of Windows do strange things if read is invoked from within a DLL.

Other aspects of the interface are similar to the Unix routine, apart from the routine returning zero for success and an error code for failure rather than a FILE* or NULL.

There is no xt_jobres in the windows version, the job number is placed in the field apispq_job in the passed structure jobd to xt_jobadd. For consistency with the Unix version, the external variable xtapi_dataerror is also assigned any error code returned.

6.2.10 xt_jobdel

```
int xt_jobdel(const int fd,
              const unsigned flags,
              const slotno_t slot)
```

The xt_jobdel function is used to delete a job, aborting it if it is currently printing.

fd is a file descriptor previously returned by xt_open or the equivalent routines.

flags is zero, or the bitwise OR of one or both of the following:

XTAPI_FLAG_IGNORESEQ	Ignore changes since the list was last read
XTAPI_FLAG_FORCE	Ignore "not printed" flag

Slot is the slot number corresponding to the job as previously returned by xt_joblist or xt_jobfindslot.

If the job has not been printed, and flags does not contain XTAPI_FLAG_FORCE, then the job will not be deleted, but the error XTAPI_NOT_PRINTED will be reported. You can tell whether the job has been printed from the struct apispq job structure returned by xt_jobread or xt_jobfind. The field apispq_dflags has the bit designated by APISPQ_PRINTED set if it has been printed.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to delete all jobs:

```
int fd, ret, nj, i;
slotno_t *slots;
fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}
ret = xt_joblist(fd, 0, &nj, &slots);
if (ret < 0) { /* error handling */
    ...
}
for (i = 0; i < nj; i++) {
    ret = xt_jobdel(fd, XTAPI_FLAG_FORCE, slots[i]);
    if (ret < 0) { /* error handling */
       ...
    }
}
xt_close(fd);
```

6.2.11 xt_jobupd

```
int xt_jobupd(const int fd,
              const unsigned flags,
              const slotno_t slot,
              struct apispq * jobd)
```

The xt_jobupd function is used to update the details of a job.

fd is a file descriptor previously returned by xt_open or equivalent routines.

flags is zero, or XTAPI_FLAG_IGNORESEQ to ignore changes since the list was last read.

slot is the slot number corresponding to the job as previously returned by xt_joblist or xt_jobfindslot.

jobd is a descriptor containing the job descriptor as defined in xtapi.h.

The fields in struct apispq are defined in the xt_jobread documentation (see page 14).

Note that we recommend that the whole structure be first read in with xt_jobread or xt_jobfind and then required fields updated; this approach will cover any future extensions with additional fields.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to change the name of job readme.txt to myfile.

```
int fd, ret, nj, i;
struct apispq job;
slotno_t *slots;

fd = xt_open("myhost", (char *) 0, 0);
if  (fd < 0) { /* error handling */
     ...
}

/* make a list of jobs */

ret = xt_joblist(fd, 0, &nj, &slots);
if (ret < 0) { /* error handling */
     ...
}

for (i = 0; i < nj; i++) {
     ret = xt_jobread(fd, 0, list[i], &job);
     if  (ret < 0)
         continue;
     if (strcmp(job.apispq_file, "readme.txt"))
         continue;
     strcpy(job.apispq_file, "myfile");
     ret = xt_jobupd(fd, 0, list[i], &job);
     if (ret < 0) { /* error handling */
        ...
     }
     break;
}
xt_close(fd);
```

6.2.12 xt_jobmon (Unix and Linux versions)

```
int xt_jobmon(const int fd,
              void (*fn)(const int))
```

The xt_jobmon function is used to set the function fn to be called upon notification of any changes to the jobs list.

fd is a file descriptor previously returned by xt_open or equivalent routines.

fn is a function which must be declared as returning void and taking one const int argument. Alternatively, this may be NULL to cancel monitoring.

The function fn will be called upon each change to the job list. The argument passed will be fd. Note that any changes to the job queue are reported (including changes on other hosts whose details are passed through) as the API does not record which jobs the user is interested in.

The function xt_jobmon returns 0 if successful otherwise the error code XTAPI_INVALID_FD if the file descriptor is invalid. Invalid fn parameters will not be detected and the application program will probably crash.

6.2.13 xt_jobmon (Windows version)

```
int xt_setmon(const int fd,
              HWND hWnd,
              UINT wMsg)

int xt_procmon(const int fd)

void xt_unsetmon(const int fd)
```

The xt_setmon routine may be used to monitor changes to the job queue or printer list. Its parameters are as follows.

fd is a file descriptor previously returned by xt_open or equivalent routines.

hWnd is a windows handle to which messages should be sent.

wMsg is the message id to be passed to the window (WM_USER or a constant based on this is suggested).

To decode the message, the xt_procmon is provided. This returns XTWINAPI_JOBPROD to indicate a change or changes to the job queue and XTWINAPI_PTRPROD to indicate a change or changes to the printer list. If there are changes to both, two or more messages will be sent, each of which should be decoded via separate xt_procmon calls.

To cancel monitoring, invoke the routine

```
xt_unsetmon(const int fd)
```

If no monitoring is in progress, or the descriptor is invalid, this call is just ignored.

6.3 Printer operations

6.3.1 xt_ptrlist

```
int xt_ptrlist(const int fd,
               const unsigned flags,
               int *numptrs,
               slotno_t **slots)
```

The xt_ptrlist function is used to obtain a list of printers.

`fd` is a file descriptor previously returned by xt_open or the equivalent routines.

`flags` is either zero, or `XTAPI_FLAG_LOCALONLY` to request that only printers local to the server be listed.

`numptrs` is a pointer to an integer value which, on successful completion, will contain the number of printer slots returned.

`slots` is a pointer to to an array of slot numbers. These slot numbers can be used to access individual printers. The memory used by this vector is owned by the API, therefore no attempt should be made by the user to free it. This contrasts, for example, with X library routines.

Also note that certain other calls to the API, notably xt_joblist, with the same `fd`, may reuse the space, so the contents should be copied if required before other API calls are made.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to list all printers

```
int fd, ret, np, i;
slotno_t *slots;
fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}
ret = xt_ptrlist(fd, 0, &np, &slots);
if (ret < 0) { /* error handling */
    ...
}
for (i = 0; i < np; i++) {
    slotno_t this_slot = slots[i];
    /* process this_slot */
    ...
}
xt_close(fd);
```

6.3.2 xt_ptrread

```
int xt_ptrread(const int fd,
               const unsigned flags,
               const slotno_t slot,
               struct apispptr *ptrd)
```

The xt_ptrread function is used to retrieve the details of a printer from a given slot number.

`Fd` is a file descriptor previously returned by xt_open or the equivalent routines.

`Flags` is zero, or a bitwise OR of one of the following values

XTAPI_FLAG_LOCALONLY	Ignore remote printers/hosts (from the point of view of the server, not the client).
XTAPI_FLAG_USERONLY	Ignore other users jobs
XTAPI_FLAG_IGNORESEQ	Ignore changes since the list was last read

slot is the slot number corresponding to the printer as previously returned by a call to xt_ptrlist or xt_ptrfindslot.

ptrd is a descriptor, which on return will contain the details of the printer in a struct apispptr as defined in xtapi.h and containing the following elements:

Type	Field	Description
jobno_t	apispp_job	Job number being printed
slotno_t	apispp_jslot	Slot number of job being printed
char	apispp_state	State of printer
char	apispp_sflags	Scheduler flags
unsigned char	apispp_dflags	Despooler flags
unsigned char	apispp_netflags	Network flags
unsigned short	apispp_extrn	External printer type 0=standard
classcode_t	apispp_class	Class code bits 1=A 2=B 4=C etc
int_pid_t	apispp_pid	Process id of despooler process
netid_t	apispp_netid	Host id of printer network byte order
slotno_t	apispp_rslot	Slot number on remote machine
unsigned long	apispp_minsize	Minimum size of acceptable job
unsigned long	apispp_maxsize	Maximum size of acceptable job
char []	apispp_dev	Device name
char []	apispp_form	Form type
char []	apispp_ptr	Printer name
char []	apispp_feedback	Feedback message
char []	apispp_comment	Printer description

The following bits are set in the apispp_sflags field to indicate printer flags:

Bit (#define)	Meaning
APISPP_INTER	Had interrupt message, not yet acted on it.
APISPP_HEOJ	Had halt at end of job

The following bits are set in the apispp_dflags field to indicate printer flags:

Bit (#define)	Meaning
APISPP_HADAB	Had “Abort” message
APISPP_REQALIGN	Alignment required

The apispp_netflags field contains the following bits:

Bit (#define)	Meaning
APISPP_LOCALONLY	Printer is local only to host.
APISPP_LOCALHOST	Printer uses network filter

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to read the names of all printers

```
int fd, ret, np, i;
struct apispptr ptr;
slotno_t *slots;
fd = xt_open("myhost", (char *)0, 0);
if (fd < 0) {  /* error handling */
    ...
}
ret = xt_ptrlist(fd, 0, &np, &slots);
if (ret < 0) { /* error handling */
    ...
}
for (i = 0; i < np, i++) {
     ret = xt_ptrread(fd, XTAPI_FLAG_IGNORESEQ, slots[i], &ptr);
     if (ret < 0)       { /* error handling */
         ...
     }
     printf("%s\n", ptr.apispp_ptr);
}
xt_close(fd);
```

6.3.3 xt_ptrfind

```
int xt_ptrfind(const int fd,
               const unsigned flags,
               const char *name,
               const netid_t nid,
               slotno_t *slot,
               struct apispptr *ptrd)
```

```
int xt_ptrfindslot(const int fd,
                   const unsigned flags,
                   const char *name,
                   const netid_t nid,
                   slotno_t *slot)
```

The xt_ptrfind and xt_ptrfindslot functions may be used to find a printer from a given printer name rather than by the slot number. xt_ptrfind retrieves the printer description, xt_ptrfindslot just retrieves the slot number.

fd is a file descriptor previously returned by xt_open or equivalent routines.

flags is zero, or XTAPI_FLAG_LOCALONLY to ignore remote printers/hosts, (from the point of view of the server, not the client).

name is the printer name to be searched for.

nid is the network-byte order IP address of the host of the machine whose printer is to be searched for. This should be correct even if XTAPI_FLAG_LOCALONLY is specified.

slot is a pointer to a location in which the slot number of the printer is placed if the search is successful. It may be NULL if this is not required (but this would be almost pointless for xt_ptrfindslot).

ptrd is a pointer to a field to contain the printer name as defined in xtapi.h.

The fields in struct apispptr are defined in the xt_ptrread documentation on page 27.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

NB If two or more printers on the same host have the same name, then it is not defined which is returned by xt_ptrfind and xt_ptrfindslot. In such cases, the whole printer list should be read and the correct one selected.

6.3.4 xt_ptradd

```
int xt_ptradd(const int fd,
              struct apispptr *ptrd)
```

The function xt_ptradd is used to install a printer.

fd is a file descriptor previously returned by xt_open.

ptrd is a struct apispptr describing the details of the printer. It is defined in the file xtapi.h and as described in the xt_ptrread documentation on page 27.

Only values for the name, device, formtype, description, local flag, the minimum and maximum job sizes, the network filter flag and the class code are accepted. All other parameters are ignored. We suggest that you clear all fields to zero before starting. Future releases with additional fields will be guaranteed to default to the existing behaviour if the additional fields are set to zero.

xt_ptradd returns zero if successful, otherwise error codes as listed in chapter 3.

An example to add a printer called hplj1 on device /dev/tty12 with form type a4.

```
int fd, ret;
struct apispptr ptr;
fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}
memset((void *) &ptr, '\0', sizeof(ptr));
ptr.apispp_class = 0xffffffff;
ptr.apispp_minsize = ptr.apispp_maxsize = 0;
strcpy(ptr.apispp_ptr, "hplj1");
strcpy(ptr.apispp_form, "a4");
strcpy(ptr.apispp_dev, "tty12");
strcpy(ptr.apispp_comment, "My new printer");
ret = xt_ptradd(fd, &ptr);
if (ret < 0) { /* error handling */
```

```
        ...
    }
    xt_close(fd);
```

6.3.5 xt_ptrdel

```
    int xt_ptrdel(const int fd,
                  const unsigned flags,
                  const slotno_t slot)
```

The function xt_ptrdel is used to delete or “uninstall” a printer. Note that the definition of the printer, setup files and similar in the printers directory on the server, by default `/usr/spool/printers`, is not altered, the printer is only removed from the online list.

`fd` is a file descriptor previously returned by xt_open or similar routines.

`Flags` is either zero, or `XTAPI_FLAG_IGNORESEQ` to ignore changes since the list was last read.

`slot` is the slot number corresponding to the printer as previously returned by xt_ptrlist or xt_ptrfindslot.

xt_ptradd returns zero if successful, otherwise error codes as listed in chapter 3.

An example to delete all printers:

```
    int fd, ret, np, i;
    slotno_t *slots;

    fd = xt_open("myhost", (char *)0, 0);
    if (fd < 0) { /* error handling */
        ...
    }
    ret = xt_ptrlist(fd, XTAPI_LOCALONLY, &np, &slots);
    if (ret < 0) { /* error handling */
        ...
    }

    for (i = 0; i < np; i++) {
          ret = xt_ptrdel(fd, 0, slots[i]);
          if (ret < 0) { /* error handling */
              ...
          }
    }

    xt_close(fd);
```

6.3.6 xt_ptrupd

```
    int xt_ptrupd(const int fd,
                  const unsigned flags,
                  const slotno_t slot,
                  struct apispp *ptrd)
```

The xt_ptrupd function is used to update the details of a printer.

fd is a file descriptor previously returned by xt_open.

flags is zero, or XTAPI_FLAG_IGNORESEQ to ignore changes since the list was last read.

slot is the slot number corresponding to the printer as previously returned by xt_ptrlist or xt_ptrfindslot.

ptrd is a descriptor containing the printer descriptor as defined in xtapi.h.

The fields in struct apispptr are defined in the xt_ptrread documentation on page 27.

Note that we recommend that the whole structure be first read in with xt_ptrread or xt_ptrfind and then required fields updaated; this approach will cover any future extensions with additional fields.

Only changes to the name device, description, form type, local flag, the minimun and maximum job sizes, the network filter flag and the class code are accepted, and none at all if the printer is running.

The function returns 0 if successful otherwise one of the error codes as listed in chapter 3.

An example to change the form type on printer hplj1.

```
int fd, ret;
struct apispptr ptr;
slotno_t pslot;

fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}

/* Find printer */
ret = xt_ptrfind(fd, 0, "hplj1", servip, &pslot, &ptr);
if (ret < 0)  { /* error handling */
    ...
}

strcpy(ptr.apispp_form, "a4.p10");
ret = xt_ptrupd(fd, 0, pslot, &ptr);

if (ret < 0) { /* error handling */
    ...
}

xt_close(fd);
```

6.3.7 xt_ptrop

```
int xt_ptrop(const int fd,
             const unsigned flags,
             const slotno_t slot,
             const unsigned op)
```

The xt_ptrop function is used to perform an operation on a printer.

fd is a file descriptor previously returned by xt_open or the equivalent routines.

flags is zero, or XTAPI_FLAG_IGNORESEQ to ignore changes since the list was last read.

slot is the slot number corresponding to the printer as previously returned by a call to xt_ptrlist or xt_ptrfindslot.

op is one of the following values:

Operation code	Description
PRINOP_RSP	Restart printer
PRINOP_PHLT	Halt printer at the end of the current job
PRINOP_PSTP	Halt printer at once
PRINOP_PGO	Start printer
PRINOP_OYES	Approve alignment page
PRINOP_ONO	Disapprove alignment page
PRINOP_INTER	Interrupt printer
PRINOP_PJAB	Abort current job on printer

The function returns zero if successful, otherwise error codes as listed in chapter 3.

An example to halt all printers:

```
int     fd, ret, np, i;
struct apispptr ptr;
slotno_t *slots;

fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}

/* make a list of all the printers */
ret = xt_ptrlist(fd, 0, &np, &slots);
if (ret < 0) { /* error handling */
      ...
}

for (i = 0; i < np; i++) {
     ret = xt_ptrop(fd, XTAPI_FLAG_IGNORESEQ, slots[i], PRINOP_PHLT);
     if (ret < 0  &&  ret != XTAPI_PTR_NOTRUNNING) {
         /* error handling ignoring ones already stopped*/
         ...
     }
}
xt_close(fd);
```

6.3.8 xt_ptrmon

```
int xt_ptrmon(const int fd,
                    void (*fn)(const int))
```

NB that this routine is not available in the Windows version, please see the section on xt_setmon in section 25 which covers both jobs and printers.

The xt_ptrmon function is used to set the function `fn` to be called upon notification of any changes to the printers list.

`fd` is a file descriptor previously returned by xt_open or the equivalent routines.

`fn` is a function which must be declared as returning void and taking one `const int` argument. Alternatively, this may be NULL to cancel monitoring.

The function `fn` will be called with `fd` as an argument upon each change to the printer list.

Please note that any changes to the printer list is reported as the API does not record which printers the user is interested in.

The function xt_ptrmon returns 0 if successful otherwise the error code `XTAPI_INVALID_FD` if the file descriptor is invalid. Invalid `fn` parameters will not be detected and the application program will probably crash.

6.4 User permissions

The following routines access user permissions (in most cases the user will need to have write administration file privilege).

6.4.1 xt_getspu

```
int xt_getspu(const int fd,
           const char *user,
           struct apispdet *res)
```

The function xt_getspu is used to retrieve the defaults for a particular user. Unless the calling user has *Write Administration File* privilege, the user name must be the calling user.

`fd` is a file descriptor previously returned by xt_open or an equivalent routine.

`user` is a pointer to the username of the user details being retrieved.

`res` is a descriptor, which upon return will contain the details of user. The structure `apispdet` is defined in the file `xtapi.h`, and contains the following fields:

Type	Field	Description
unsigned char	spu_isvalid	Valid user ID
char []	spu_resvd1	Reserved
int_ugid_t	spu_user	User ID
unsigned char	spu_minp	Minimum priority
unsigned char	spu_maxp	Maximum priority
unsigned char	spu_defp	Default priority
char []	spu_form	Default form type
char []	spu_formallow	Allowed form type pattern
char []	spu_ptr	Default printer
char []	spu_ptrallow	Allowed printer pattern
unsigned long	spu_flgs	Privilege flag
classcode_t	spu_class	Class of printers
unsigned char	spu_cps	Maximum copies allowed
unsigned char	spu_version	Release of **Xi-Text**

The spu_flgs field of res will contain a combination of the following:

PV_ADMIN	Administrator (edit admin file)
PV_MASQ	Masquerade as other users
PV_SSTOP	Can run sstop (can stop scheduler)
PV_FORMS	Can use other forms than default
PV_CPRIO	Can change priority on queue
PV_OTHERJ	Can change other users' jobs
PV_PRINQ	Can move to printer queue
PV_HALTGO	Can halt, restart printer
PV_ANYPRIO	Can set any priority on queue
PV_CDEFLT	Can change own default priority
PV_ADDDEL	Can add/delete printers
PV_COVER	Can override class
PV_UNQUEUE	Can unqueue jobs
PV_VOTHERJ	Can view other jobs not neccesarily edit
PV_REMOTEJ	Can access remote jobs
PV_REMOTEP	Can access remote printers
PV_FREEZEOK	Can save default options
PV_ACCESSOK	Can access sub-screens
PV_OTHERP	Can use other printers from default
ALLPRIVS	A combination of all of the above

The function returns zero if successful, otherwise error codes as listed in chapter 3.

An example to view the privileges of user mark:

```
int     fd, ret;
struct apispdet res;
fd = xt_open("myhost", (char *)0, 0);
if (fd < 0) { /* error handling */
    ...
}
ret = xt_getspu(fd, "mark", &res);
if (ret < 0) { /* error handling */
    ...
}
if (res.spu_flags & PV_HALTGO)
    printf("user mark cannot halt printers\n");

printf("marks maximim priority is %d\n", res.spu_maxp);
xt_close(fd);
```

6.4.2 xt_getspd

```
int xt_getspd(const int fd,
              struct apisphdr *res)
```

The xt_getspd function is used to retrieve the defaults privileges, form types etc for new users on the host with which the API is communicating. No particular privilege is required to perform this operation.

fd is a file descriptor previously returned by xt_open or the equivalent routines.

res is a descriptor which upon return will contain the the default user privileges. The structure apisphdr is defined in xtapi.h and contains the following elements:

Type	Field	Description
long	sph_lastp	Time last read password file
unsigned char	sph_minp	Minimum priority
unsigned char	sph_maxp	Maximum priority
unsigned char	sph_defp	Default priority
char []	sph_form	Default form type
char []	sph_formallow	Allowed form type pattern
char []	sph_ptr	Default printer
char []	sph_ptrallow	Allowed printer pattern
unsigned long	sph_flgs	Privilege flag
classcode_t	sph_class	Class of printers
unsigned char	sph_cps	Maximum copies allowed
unsigned char	sph_version	Release of **Xi-Text**

The spu_flgs field will contain a combination of the following:

PV_ADMIN	Administrator (edit admin file)
PV_MASQ	Masquerade as other users
PV_SSTOP	Can run sstop (can stop scheduler)
PV_FORMS	Can use other forms than default
PV_CPRIO	Can change priority on queue
PV_OTHERJ	Can change other users' jobs
PV_PRINQ	Can move to printer queue
PV_HALTGO	Can halt, restart printer
PV_ANYPRIO	Can set any priority on queue
PV_CDEFLT	Can change own default priority
PV_ADDDEL	Can add/delete printers
PV_COVER	Can override class
PV_UNQUEUE	Can unqueue jobs
PV_VOTHERJ	Can view other jobs not necessarily edit
PV_REMOTEJ	Can access remote jobs
PV_REMOTEP	Can access remote printers
PV_FREEZEOK	Can save default options
PV_ACCESSOK	Can access sub-screens
PV_OTHERP	Can use other printers from default
ALLPRIVS	A combination of all of the above

The function returns zero if successful, otherwise error codes as listed in chapter 3.

An example to view the default privileges on the host machine:

```
int fd, ret;
struct apisphdr res;

fd = xt_open("myhost", (char *) 0, 0);
if (fd < 0) { /* error handling */
    ...
}

ret = xt_getspd(fd, &res);
if (ret < 0)  { /* error handling */
    ...
}

if (res.sph_flgs & PV_HALTGO)
    printf("users cannot stop and start printers\n");
printf("the default maximum priority is %s\n", res);
xt_close(fd);
```

6.4.3 xt_putspu

```
int xt_putspu(const int fd,
              const char *user,
              struct apispdet *newp)
```

The xt_putspu function is used to set privileges for a user. The calling user must have *write administration file privilege*, or must be the same as the specified user and be only trying to change the default form type or priorities (with the appropriate privilege for that).

`fd` is a file descriptor previously returned by xt_open.

`user` is a pointer to the user name for which the details are being updated.

`newp` is a pointer to a structure containing the new user privileges.

The `struct apispdet` is defined int the file `xtapi.h`. The fields of the structure are as defined for xt_getspu on page 33.

he function returns zero if successful, otherwise error codes as listed in chapter 3.

An example to give a user permission to add and delete printers

```
int fd, ret;
struct apispdet new_privs;

fd = xt_open("myhost", (char *)0, 0);
if (fd < 0) { /* error handling */
    ...
}

ret = xt_getspu(fd, "helen", &new_privs);
if (ret < 0)   { /* error handling */
    ...
}

new_privs.spu_flgs |= PV_ADDDEL;
```

```
xt_close(fd);
```

6.4.4 xt_putspd

```
int xt_putspd(const int fd,
              struct apisphdr *ret)
```

The xt_putspd function is used to set the default user privileges on the local host.

Its parameters are as follows:

`fd` is a file descriptor previously returned by xt_open.

`res` points to a structure which contains the privileges. The `struct apisphdr` is defined in the file `xtapi.h` as described for xt_getspd on page 35.

The function returns zero if successful, otherwise error codes as listed in chapter 3.

An example to give all users the permission to add and delete printers:

```
int fd, ret;
struct apisphdr new_privs;
fd = xt_open("myhost", (char *)0, 0);
if (fd < 0) {  /* error handling */
     ...
}
/* get the current permissions */
ret = xt_getspd(fd, &new_privs);
if (ret < 0) { /* error handling */
    ...
}
new_privs.sph_flgs |= PV_ADDDEL;
ret = xt_putspd(fd, &new_privs);
if (ret < 0) { /* error handling */
    ...
}
xt_close(fd);
```

The default permissions now apply to every user not “different“ in some way from the default. So if you change the default permissions, say to have a different default form type, the users who in some way differ from the default before will not be changed. You may want to run over all users and make appropriate adjustments.

Chapter 7

Example API program

The following program is an example program to provide for an “alternative printer” to be activated when a machine running the main printer is or goes offline. The program runs on the “secondary” machine:

```
/*
 * altprin.c:  created by John Collins.
 */

#include <stdio.h>
#include <sys/types.h>
#include "xtapi.h"
#include <unistd.h>
#include <netdb.h>
#include <string.h>

int  had_prod;
char *primary,            /* Primary host name */
     *secondary,          /* Secondary host name */
     *primary_prin,       /* Primary printer name */
     *secondary_prin;     /* Secondary printer name */

netid_t  prim_hostid, sec_hostid;

int  xtfd;

/*
 *   Routine to call when printer event occurs.
 *   Just set flag and let the main loop look at it
 *   when it is ready.
 */

void    prodder(const int fd)
{
     had_prod++;
}

void  process(void)
{
     /*
      *  Say we want to know about events affecting printers.
      */

     xt_ptrmon(xtfd, prodder);

 gotpri:
     for (;;)  {
         int     nump, cnt, ret;
         slotno_t *slp;
         struct  apispptr res;

         /*
          *  Wait until something interesting happens to a printer.
          */

         pause();
         if (!had_prod)  /* Huh???  */
             continue;
         had_prod = 0;

         /*
          *  Get list of printers "slot numbers" into "slp", number
          *  into "nump".
          *  We don't really need to do this on each loop if printer
          *  slot numbers don't change too much, which they don't
          */

         if (xt_ptrlist(xtfd, 0, &nump, &slp) < 0)
             exit(255);

         /*
          *  Search list for primary printer.
          *  If found, all is ok, and we go back to sleep.
          */

         for (cnt = 0;  cnt < nump;  cnt++)  {
             if (xt_ptrread(xtfd,
                              XTAPI_FLAG_IGNORESEQ,
                              slp[cnt],
                              &res) < 0)
                 exit(254);
             if  (res.apispp_netid != prim_hostid)
                 continue;
             if  (strcmp(res.apispp_ptr, primary_prin) == 0)
                 goto  gotpri;
         }

         /*
          *  We didn't find primary printer, so we start up the
          *  secondary printer.  First find the thing.
          */

         for (cnt = 0;  cnt < nump;  cnt++)  {
             if (xt_ptrread(xtfd,
                              XTAPI_FLAG_IGNORESEQ,
                              slp[cnt],
                              &res) < 0)
                 exit(254);
             if (res.apispp_netid != sec_hostid)
                 continue;
             if (strcmp(res.apispp_ptr, secondary_prin) == 0)
                 goto  gotsec;
         }
         fprintf(stderr, "Cannot find secondary printer, %s\n", secondary_prin);
             exit(200);

         /*
          *  Found secondary printer, print a warning message
          *  if already running.
          */

gotsec:
         if (res.apispp_state >= API_PRPROC) {
             fprintf(stderr,
           "I think that the secondary printer is already running\n");
             exit(0);
         }

         /*
          *  Tell the world, start it up, and exit
          */

         fprintf(stderr, "Activating secondary printer
                    %s:%s\n", secondary, secondary_prin);

         if ((ret = xt_ptrop(xtfd, XTAPI_FLAG_IGNORESEQ, slp[cnt], PRINOP_PGO))
< 0) {
             printf("Error starting printer - %d\n", ret);
             exit(0);
         }
}

int main(int argc, char **argv)
{
     extern  intoptind;
     extern  char    *optarg;
     int     ch;
     struct  hostent*hp;
     char        *cp;
     static  char    myname[256];

     /*
      *  Get "my" host name.
      */

     myname[sizeof(myname) - 1] = '\0';
     gethostname(myname, sizeof(myname) - 1);
```

```
if (!(hp = gethostbyname(myname))) {
    fprintf(stderr, "Who am I???\n");
    return 10;
}

/*
 *  Get arguments giving primary and secondary printers.
 */

while ((ch = getopt(argc, argv, "p:s:")) != EOF)  {
    switch  (ch)  {
    default:
        fprintf(stderr,
            "Usage:  altprin -p primary -s secondary\n");
        return  1;
    case 'p':
        primary = optarg;
        break;
    case 's':
        secondary = optarg;
        break;
    }
}

if (!primary)  {
    fprintf(stderr, "No primary host:printer name given\n");
    return 2;
}
if (!secondary)  {
    fprintf(stderr, "No secondary host:printer name given\n");
    return 3;
}

/*
 *  Split host:printer names into separate strings.
 *  If not host name, tack on "my" name.
 */

if (cp = strchr(primary, ':'))   {
    *cp = '\0';
    primary_prin = cp+1;
}
else {
    primary_prin = primary;
    primary = myname;
    fprintf(stderr, "Primary printer on local host?\n");
}
if (cp = strchr(secondary, ':'))   {
    *cp = '\0';
    secondary_prin = cp+1;
}
else {
    secondary_prin = secondary;
    secondary = myname;
```

```
     }

     if (strcmp(primary, secondary) == 0) {
         fprintf(stderr, "Sorry both printers on the same host\n");
         return  4;
     }

     /*
      *  Get host ids, used in scanning printer list.
      */

     if (!(hp = gethostbyname(primary))) {
         fprintf(stderr, "Sorry, unknown primary host name %s\n",
                         primary);
         return  5;
     }
     else
         prim_hostid = *(netid_t *) hp->h_addr;

     if (!(hp = gethostbyname(secondary))) {
         fprintf(stderr, "Sorry, unknown secondary host name %s\n",
                 secondary);
         return  6;
     }
     else
         sec_hostid = *(netid_t *) hp->h_addr;

     /*
      *  Open API link.
      */

     if ((xtfd = xt_open(secondary, (char *) 0, 0)) < 0)  {
         fprintf(stderr,
             "Sorry, cannot open connection to secondary host\n");
         return 7;
     }

     /*
      *  Fork off to leave a daemon process.
      *  (You might want to set process group, ignore
      *  signals and/or reconnect
      *  stdout/stderr).
      */

      if (fork() != 0)
          return 0;

      /*
       *  Do the business (no return).
       */

      process();
}
```