IDM SYSTEM/UNIX SOFTWARE SPECIFICATION

THE INTELIGENT DATABASE MACHINE™

BRITTON LEE, INCORPORATED

Britton Lee, Inc.

IDM SYSTEM/UNIX SOFTWARE SPECIFICATION

FEBRUARY 1988
Part Number 205-1392-006

This document supercedes all previous documents. This edition is intended for use with software release number 3.5 and future software releases, until further notice.

The information contained within this document is subject to change without notice. Britton Lee assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under license and may only be used or copied by the terms of such license.

NAME

coversheet – a message to our readers

DESCRIPTION

This revision of the IDMLIB spec describes the final **Product Release** of Release 3.

I am somewhat embarrassed to realize that this spec uncovers so many internal aspects that it is almost an internals document. The intent is that every attempt to use some facility should be centralized in one routine (or family of routines). For example, *itapeopts*(3I) is intended for minimal use, but all users of IDM tape options should use this routine.

> **NOTE:** This is *not* a user document. It is intentionally terse to minimize possible inconsistencies and to minimize the size of the document. Other documents will be provided in the future directed for a user audience.

DEFICIENCIES

The following areas are known to be insufficiently addressed at the current time. They are ranked in approximate priority order.

- A forms-based screen interface is a necessity.
- There should be some way of adjusting the output field format in the *idl*(1I) and *sql*(1I) programs.

HISTORY

The following descriptions do not include sections that have changed because of minor editorial or typographical changes.

3.5 **PHI, UNIX/ULTRIX binary Product Release Version**

intro(1I)	Warning added that syntax errors in file specifications will cause the remaining parameters to be ignored. More syntax checking has been added to the Release 3.5 *pextract*(3I) routine.
idmckload(1I)	New utility checks database and trasaction logs if running RDBMS Software Release 3.5.
idmcklog(1I)	New utility checks transaction log time stamps and verifies they are complete.
idmconfig(1I)	New menu-based IDM configuration utility.
idmcopy(1I)	Many new RDBMS Software Release 3.5 features supported. Copy in accepts *.d (names ending with a ".d") as the relation list for convenience. Page lock option avoids locking the whole relation during copy in.
idmdump(1I)	Always send IDM tape error performance option to report soft tape errors to user.
idmfcopy(1I)	New 2 byte length variable length field syntax. Page lock flag avoids locking the whole relation during copy in.
idmload(1I)	Always send IDM tape error performance option to report soft tape errors to user.
idmxdbin(1I)	New DBA utility to kill a hung dbin.
ric(1I)/rsc(1I)	New **$cancel** command to cancel commands on the database server. New **$fetch** feature for execution of user stored commands.

iecontrol(3I) New "mapcc" control to allow printing of control characters for Kanji.

iesetopt(3I) Set options are now linked to the IDL/SQL command tree from the environment when the tree is sent to the database server. Set options are no longer linked into the command tree at parse time.

irget(3I) Add IP_DBIN field which returns dbin from idmrun structure.

itqstmt(3I) New interface for fast tree building of query commands such as **retrieve** or **append**.

itxdbin(3I) Kill dbin tree builder for the idmxdbin(1I) utility.

keylook(3I) New binary keyword lookup interface.

parsedate(3I) Now accepts IDM time specifications in the format "idmtime <days> [<ticks>]" where *days* is an integer representing the number of days since the epoch and *ticks* is an integer representing the number of 60ths of a second since midnight.

params(5I) MAPCC now controls printing of control characters in tupprint. Multiple hashed message files added for MESSAGES using a comma separated list. IDMSYSLINE is obsolete along with the serial multi-user kernel driver.

3.4 PHI, UNIX/ULTRIX binary Product Release Version

intro(1I) Addition of IDM tape parameters *verify* and *norewind*.

idl(1I) Auto-association of stored command definitions has been added. New %redo command to reexecute the complete log of the user session. It is no longer required to quote immediate commands containing special characters. Better syntax error reporting.

idmdump(1I) Removed –r flag. Added –or|m[clock,waitcnt] flags for doing online (read/write) dumps.

rc(1I), ric(1I) Renamed rc to ric.

sql(1I) New %redo command to reexecute the complete log of the user session. It is no longer required to quote immediate commands containing special characters. Better syntax error reporting.

bytetype(3I) Add ISZWIDTH and ISKANGI for kangi language support. Add ISPMATCH to check for IDM pattern matching characters.

crackargv(3I) Default flags added. The version (–V) flag is useful to customers needing the version number of the host software.

getclock(3I), params(5I)
New EPOCHOFFSET system parameter to change the IDMLIB epoch (defaults to Jan 1, 1900).

exc(3I) A new handler **excprbo** was added which modifies the severity to "E", reraises the exception and backs out.

ifdump(3I), irdump(3I)
New routines to dump IFILE and IDMRUN structures respectively.

igeteot(3I) Allow the "n!" (n is a digit) on IDM tape transport prompting. If a '!' is present then the tape volume name will not be verified.

iftscan(4I) added TK_INFO and giveinfo param to return token information. The new "0f" (4 byte float) and "0d" (8 byte float) radix specifications have

been added.

3.3 **3B/Pyramid–Binary Product Release Version 1.**

idl(1I),sql(1I) Added new flag –p to turn off reading of profile files.

rc(1I) New –S flag to set the size of the symbol table. RC library librcrun.a has been merged into libidmlib.a. To compile rc generated source files use –lidmlib only.

igetdone(3I) IDM warning messages are not printed if there was an error on the database server.

iftidm(4I) and

params(5I) System parameter **IDMDEV** may now have the device name coded in using a *filespec* syntax. On UNIX users may omit "/dev/" from the device name.

3.2 **UNIX/ULTRIX Product Release Version 1.** Added **A:IDMRUN.BADIDMRUN** and **A:IDMRUN.RECOMPILE** exceptions to all runtime interfaces (ir*(3I)) to check for valid structures and version ids.

idl(1I) Add the **?** and **!** help and shell commands. Mark the char to the **continuation** command as being optional. Add the silent flag **–s**. Document the **MAPCC** map control character system parameter.

idmdump(1I) Added new flags **–r** and **–w** for no-read lock and wait options during the dumping of a database.

rc(1I) (New page) Relational C precompiler.

sql(1I) (New page) SQL parser.

crackargv(3I) Exceptions with severity **U:** now exit with the RETCODE **RE_USAGE**.

exc(3I) Added *exccleanup*. Added exception handler setting routine *excahandle*. This macro takes an argument to be passed to the handling functions.

fmtfloat(3I) Precision zero suppresses printing a decimal place. Useful for printing BCD integers.

idlparse(3I) Input overflow for commands such as and no longer send data which encountered a conversion overflow error.

iftloterm(4I) (New page) Low level machine dependent file type.

iftterm(4I) Additions for cursor motion characters. Opens the *IftLoTerm* machine dependent file type.

irsql(3I) (New page) Runtime library interface to SQL.

istdio(3I) (New page) Standard I/O compatability library.

sqlparse(3I) (New page) Build query tree's from SQL program input.

params(5I) Removed parameter **CLOCKTICKS**. Add parameter **MAPCC** to map control characters in the IDL and SQL. Add parameter **IDMVERSION** for the version software running on the database server.

maketerm(8I) Add cursor motion definitions. **–C** flag may be used to generate a language source file rather than the binary data file.

3.1 **PHI Product Release Version 1.** This revision includes many content-free changes in the spec so that it will print nicely on our laser printer (yeah!!!).

idl(1I)	Add **%display** command.
idmfcopy(1I)	Change syntax of floating point precision specification from **float.*prec*(*len*)** to **float(*len*,*prec*)** for aesthetic reasons.
exc(3I)	Removed *onbackout*
ifgetc(3I)	The ifgetc macro IFGETC added.
iferror(3I)	Macro names iferror and ifeor are now capitalized.
ifputc(3I)	The ifputc macro IFPUTC added.
ifscrack(3I)	Remove insistence on a file name for some file types.
intro(3I)	Document foldcase mode in level 3 interface.
bcdtoa(3I)	Arguments changed for compatibility with *ftoa*(3I) and to clean up the interface.
dba(3I)	Retract foldcase changes; these are only in level 3.
dsc(3I)	Added. This describes internal routines to manipulate descriptor-based types that must be implemented during porting.
fmtclock(3I)	Broken off from *getclock*(3I).
fmtfloat(3I)	Broken off from *ftoa*(3I).
ftoa(3I)	Arguments changed for compatibility with *bcdtoa*(3I) and to clean up the interface.
getclock(3I)	Broken into three pages: *fmtclock*(3I), *getclock*(3I), and *parsedate*(3I).
igeteot(3I)	Add *ifp* and *env* parameters to *itapeload*.
irclose(3I)	Return value defined.
itcopy(3I)	Retract foldcase changes; these are only in level 3.
itdefine(3I)	Retract foldcase changes; these are only in level 3.
mapsym(3I)	Add 'd' tag for done status bits.
parsedate(3I)	Split off from *getclock*(3I).
xalloc(3I)	A primitive technique has been added to recover from out of memory conditions. Zero and negative sizes are specified.
iftidm(4I)	Prompting for user name/password is now controlled by the GETHUNPW parameter.
iftterm(4I)	Add ITG_BLOTCH.
params(5I)	GETHUNPW added.
symfile(5I)	Add 'd' tag for done status bits.
maketerm(8I)	Add so and si sequences and g1-blotch character.

2.11 **BetaC Prerelease.** Updates for (hopefully) the final modifications before product release. Most of this falls into the class of "tuning."

intro(1I)	Allow specification of volume lists for tape files.
idl(1I)	Semicolon is an alias for "go." **%continuation** added. Profile files added.
idmpasswd(1I)	Added.

anyprint(3I)	***Anyprint*** now actually prints the output; ***anyfmt*** has been added to provide (essentially) the old semantics.
bcdtoa(3I)	Separated from *ftoa*(3I); ***ftoa*** is environment dependent, while ***bcdtoa*** is not.
bitset(3I)	Name changed to be upper case to emphasize that it is a macro.
dba(3I)	Upper to lower case folding added.
exc(3I)	Add ***bocleanup***.
foldcase(3I)	Added.
ftoa(3I)	Add documentation of ***fmtfloat***; this routine can be used to simplify formating. ***Bcdtoa*** broken off to a separate page.
getclock(3I)	***Fmtclock*** now takes a timezone argument.
getpass(3I)	Resurrected.
iecontrol(3I)	Added.
ieopen(3I)	***Params*** added.
iesetopt(3I)	Parameter order reversed for consistency.
irget(3I)	IP_DMASK added. IP_TREE now gets the entire tree.
irset(3I)	IP_DMASK added. IP_TREE now sets the entire tree. IP_CURSTMT deleted.
itcopy(3I)	Case folding added.
itdefine(3I)	Case folding added.
makefname(3I)	Added.
operator(3I)	Radically changed to support multiple language, multiple operators, automatic tape loaders, different response characterstics, etc., etc.
sysshell(3I)	Changed to not raise an exception if the exit status was not normal, but rather to just pass it back to the calling program.
unsign(3I)	Name changed to be upper case to emphasize that it is a macro.
intro(4I)	Changed semantics of ***_ioerr*** routine and **_ioerr** ***ifcontrol*** call.
iftidm(4I)	Added parameter to ***id_ioerr*** routine.
iftltape(4I)	Allow specification of volume lists.
iftscan(4I)	TK_DPARAM added.
ienv(5I)	***Ic_flags field added.***

2.10 **BetaB Prerelease.** Support for different wording for different query languages in messages added (e.g., "relation" for IDL, "table" for SQL). Some changes as indicated by a detailed code walkthrough.

intro(1I)	Document QRYLANG parameter.
intro(3I)	Drop GDEF and GREF; these have been unused and do not have quite the right semantics anyhow.
anyprint(3I)	Added.
bintoa(3I)	Add overflow exception.

bytetype(3I)	Formerly *ctype*(3I); macro names are now upper case.
dba(3I)	Add *env* parameter to all routines.
iesetopt(3I)	Added.
iesubst(3I)	Now returns a RETCODE.
ifcontrol(3I)	**Truncate** control changed to **rewrite**; **dio** changed to **_dio** (to emphasize that it is reserved for internal use). **Flushblock** control added for blocked files.
igetdone(3I)	Restrict abortable errors from 128-191 (i.e., 192-255 are no longer front end errors) so the IDM group has room for more user errors.
itcopy(3I)	Pass *env* parameter.
itdefine(3I)	Pass *env* parameter.
itxcmd(3I)	Pass *env* parameter.
intro(4I)	**Dio** changed to **_dio**; **truncate** changed to **rewrite**; **flushblock** added.
ifthfile(4I)	**Truncate** changed to **rewrite**.
iftifile(4I)	**Truncate** changed to **rewrite**.
iftkeyed(4I)	Formerly *IftHash*(4I). Exception names changed for consistency.
iftltape(4I)	**Reset** and **rewrite** controls now give an error if they are not supported.
iftmtext(4I)	Add language flags.
messages(5I)	Change syntax to allow for language flags and clean up experience mapping.
retcode(5I)	RW_IGNORED added.
buildmsgs(8I)	Allow for language flags.

2.9 *Environments* added to include the range and substitute tables for precompiler support. Default exception handlers added. Support for IDM passwords added.

idl(1I)	**%experience** and **%substitute** commands added. **–E** flag changed to **–e**; **–E** should be reserved for the experience level. **–l** flag added.
intro(3I)	*Stdtrc* added. Added description of environments. Global variable *DoneMask* deleted; *DefEnv* added.
exc(3I)	*Excdhandle* added.
getpass(3I)	Deleted.
gethunpw(3I)	Added.
idlparse(3I)	*Env* parameter added.
ieopen(3I)	Added.
iesubst(3I)	Renamed from *itsubst*(3I). Old *tree* parameter is now the *env* parameter.
ifcontrol(3I)	**Rbf** parameter changed to **_rbf** to emphasize that it is not for use by normal users.
igetdone(3I)	*Env* parameter added.
igeteot(3I)	*Env* parameter added.
iputtree(3I)	*Env* parameter added. Exception E:IDMLIB.IDM.NOTSUB changed to E:IDMLIB.IDM.SUB.NEEDVAL for consistency with other substitution

message names. E:IDMLIB.IDM.SUB.TYPE and E:IDMLIB.IDM.SUB.VAL now raised from *iputtree* instead of from *itsubst*(3I). Special casing of **open database** command added.

irclose(3I)	Interaction with environment documented.
irget(3I)	IP_ENV added.
iropen(3I)	*Dbname* parameter added. Interaction with environment documented.
irset(3I)	IP_ENV added.
irsubst(3I)	Change to interact with *icsubst*(3I) instead of *itsubst*(3I).
itlprint(3I)	Added.
itprint(3I)	Prints on *stdtrc* instead of *stdout*.
itsubst(3I)	Renamed to *icsubst*(3I).
tupprint(3I)	Added *env* parameter to *tupsetup* to (someday) hold default print formats for domains.
intro(4I)	**Rbf** parameter renamed **_rbf** to emphasize that it is not for use by user programs.
iftidm(4I)	R:IDMLIB.IDM.GETHUNPW exception added.
iftscan(4I)	**TK_EOL** token added for SQL ad hoc parser. Added *TK_PSEUDO*.
ienv(5I)	Added.
retcode(5I)	Dropped RW_NOSUB.
vinodb(8I)	Dropped.

2.8 **BetaA Prerelease.** Minor updates to version 2.7

irparse(3I)	has been changed to iridl(3I).

2.7 Update for the BetaA prerelease.

intro(1I)	Default mode on tapes changed.
idmfcopy(1I)	**-w** flag added. Use of **type(text)** specified.
idmrollf(1I)	The log is really in *wdbname*.
inittape(1I)	**-i** flag added.
bcopy(3I)	*STRUCTASGN* added.
bintoa(3I)	*Atobin* added.
crackargv(3I)	Usage messages changed to **U:** severity.
dba(3I)	Names of dump and load routines changed to fit on machines with six-character external names.
exc(3I)	**U:** severity added. Mappings between UNIX signals and exceptions added.
getclock(3I)	*IDMTOTICKS* and *TICKSTOIDM* added. *Dt_ticks* field changed from a long to a short. Correspondence to IDM time and date clarified.
helpsys(3I)	Command syntax changed to be consistent with *idl*(1I).
ifcontrol(3I)	**Geterr** control added.

ifclose(3I)	The **remove** control changed to **_delete** to emphasize that it is for internal use only.
ifflush(3I)	Interaction with record-based files clarified.
ifopen(3I)	*Autoclose* parameter added.
igeteot(3I)	*Itapeload* added.
irxcmd(3I)	Added.
itrange(3I)	Dropped.
onexit(3I)	Restriction on number of calls removed.
sysedit(3I)	Second parameter dropped.
syserr(3I)	Catastrophic versus recoverable syserr's identified. Syntax of messages specified.
tf(3I)	*DPRINTF* added.
ifthfile(4I)	**Delete** control changed to **_delete**. **Pre**disposition parameter added.
iftidm(4I)	Semantics of *id_reopen* defined.
iftifile(4I)	**Delete** changed to **_delete**.
iftltape(4I)	Largely rewritten. **Gen**, **gver**, **offset**, **expiration**, and **format** parameters added. **Newfile** control added.
iftscan(4I)	**Mark** control dropped.
iftstring(4I)	**Mark** control dropped.
iftterm(4I)	Converted to ANSI specifications.
iftype(4I)	renamed *intro*(4I); **remove** changed to **_delete**.
intro(5I)	Added.
retcode(5I)	RW_NOSUB added.
symfile(5I)	User defined symbols specified.
intro(8I)	Added.
dumptape(8I)	Added.
inittape(8I)	Moved from section 1I.
maketerm(8I)	Added.
porting(8I)	Dropped.

Histories for versions prior to Beta release have been removed.

ASSISTANCE AND FEEDBACK

For assistance with the release, please contact Britton Lee Customer Support at (408) 378-7000.

NAME

Britton Lee Integrated Database Management Host Software Release 3 Introduction and Summary

DESCRIPTION

Section 1I documents commands, i.e., operations that the user can invoke directly, without the use of a programming language or special interpreter. In this spec the pages are specific to UNIX, although all commands run on all systems unless otherwise noted. Section 3I, describes IDMLIB. File types are assigned to section 4I. Section 5I describes various file and data structures. Installation and operation information are relegated to section 8I.

References to other pages within this spec are given as *name*(*n*I), where *name* is the name of the page and *n*I is the section in which it is found. References such as *name*(*n*) are to the *UNIX Programmer's Manual.*

PHILOSOPHY

Several philosophical points will facilitate the understanding of IDMLIB and this document.

- IDMLIB contains a complete runtime environment. Modules that must be modified to move to a new architecture or operating system are extremely limited; all other modules are intended to be completely portable between environments. This environment attempts to provide a reasonable set of primitives without becoming a superset of every operating system.
- Essentially all functionality is located in the library. That is, Britton Lee-supplied utilities are most often just calls to library routines, rather than being complex modules themselves. This centralizes code at the cost of making the library quite large. Fortunately, no program has to link the entire library.
- IDMLIB is layered. At the bottom layer are buffered I/O primitives; these will not be extensively used by application programs. On top of this is an interface layer to the shared database system. This defines basic data structures such as trees, target lists, etc, and is called the "system interface." It is used by most Britton Lee-supplied utilities. It is flexible but requires considerable sophistication to use. Above the system interface is the "application interface." This level simplifies the interface for application software.
- Bindings are normally dynamic. Decisions are put off until quite late. For example, the default size of an I/O buffer is determined at run time rather than compile time. This is intended to maximize flexibility and portability.

BUGS

The BUGS section describes quirks of the environment-independent implementation that cannot in good conscience be called "features" but which are not expected to change due to the high cost of solution. However, these should not be relied on either. Britton Lee reserves the right to change these semantics at any time without notice.

SEE ALSO

System Programmer's Manual, Britton Lee part number 205-2088-rev, for a description of the semantics of database server symbols, error codes, etc. This document is referred to as *SPM* in the remainder of this spec.

IDL Primer (Britton Lee part number 205-1024-rev). A tutorial introduction to the IDL language.

BL700 Installation Manual, Britton Lee part number **200-1077-rev.** or BL300 Installation/Operation Manual Britton Lee part number 205-1568-rev.

BL700 Operation Manual Britton Lee part number 201-1078-rev.

Host Software Coding Standards for the Britton Lee coding standards for host software in this release.

UNIX Programmer's Manual for references of the form *name*(*n*). References such as *name*(*n*I) refer to this spec.

Host Software Message Summary (IDL Version), Britton Lee part number 205-1432-rev, for a list of error messages returned by the IDL query language.

Host Software Message Summary (SQL Version), Britton Lee part number 205-1421-rev, for a list of error messages returned by the SQL query language.

A Guide to Writing an IDM Device Driver, Britton Lee part number 205-1150-rev.

NAME

ptx – permuted index

CONTENTS

NAME

Introduction to Release 3 Britton Lee Integrated Database Management (IDM) support commands

DESCRIPTION

Section *1I* describes the UNIX command line syntax for the Release 3 Britton Lee IDM support commands. These commands provide direct access to the IDL and SQL languages and database administrator utilities.

PARAMETERS

A number of system parameters can be set in the environment. For example, the command:

```
setenv IDMDEV /dev/testidm                    (csh)
        - or -
IDMDEV=/dev/testidm; export IDMDEV            (sh)
```

will set the parameter IDMDEV to have the value "/dev/testidm" for all subsequent commands. Parameters without an explicit setting are given a default. See ***params***(5i) for a complete description of the following parameters. Useful parameters (and their usual default value, shown in square brackets) are:

EXPERIENCE [beginner] The experience level of the user, chosen from "beginner," "able," or "expert," with case ignored. Only the first character is checked, so "expert," "Expert," "e," and "Excalibur" are the same.

IDMDRIVER [0] An index into a driver table for the database server. Driver zero is the standard driver. On most systems, driver one is the standalone serial driver. Drivers other than zero are normally used for experimental protocols. Consult your site manager for details.

IDMDEV [/dev/idm] The name of the file used to connect to the database server. If IDMDRIVER is not zero, this parameter may be interpreted differently or ignored.

TERM [dumb] The type of the terminal being used. On most UNIX systems, this is set automatically when you log in. On Berkeley UNIX systems, see *tset*(1) for details.

QRYLANG [idl] The query language you normally use: "idl" or "sql." This affects the wording of messages. The *idl*(1I) program always sets this to "idl;" *sql*(1I) always sets this to "sql." The setting of this variable in no way limits the query language you can use.

FLAGS

Flags that have values may or may not have a space between the flag and the value as convenient.

Several flags are available on almost all commands as noted in the individual command descriptions:

-B*device* The IDMDEV setting. For example, "idl -B /dev/newidm" runs *idl* using "/dev/newidm" as the interface to the database server, regardless of the setting of the IDMDEV variable.

-T*traceflags* Trace flag settings; see *tf*(3I) for details.

-P Turn on performance monitoring. This turns on the following IDM system options:

33	oRESP	Response time
34	oCPU	Database server CPU use
37	oINP	Input wait

38	oMEM	Database server memory wait
39	oCPUW	Database server CPU wait
40	oDISK	Database server disk wait
41	oTAPE	Database server tape wait
42	oOUTW	Output wait
43	oBLOCK	Blocked wait (for locks)
44	oDAC	Database Accelerator use
45	oOUTC	Output buffer wait
46	oHITS	Database server disk cache hits
47	oREADS	Database server disk reads
48	oTPERRS	Soft tape errors
49	oQRYBUF	Bytes of query buffer used
60	oPLAN	Decomposition plans

FILE SPECS

Names of files on many commands can be given using a *file spec*, that is, a combined file name, type, and parameter indication. The syntax:

filename%type,params

specifies the given *filename* of the selected *type* modified by the *params*. *Type* can be selected from **hfile** (host file, see *ifthfile*(4I)), **ifile** (IDM file, see *iftifile*(4I)), **htape** (host [ANSI] tape, see *iftltape*(4I)), and **itape** (IDM tape, see *itapeopts*(3I)). If a type is not given, **hfile** is assumed. See *ifscrack*(3I) for details. Filename for IDM files containing a ':' specify *filename:owner*. *Filename* is unused on IDM tape.

Parameters are specified using a comma-separated list of *name*(*value*) pairs. Valid parameters are documented in *ifcontrol*(3I), *ifopen*(3I) and section 4I.

If the required commas are omitted between each parameter, parameters after the missing comma will be ignored and default values used instead. Be sure to put a comma after the file *type* and before the *params* when specifying parameters.

Tape parameters are chosen from the list:

mode(M) I/O mode; *M* may be 'r' (read), 'w' (overwrite), or 'a' (append). Britton Lee utilities that read tapes (e.g., *idmload*(1I)) default to 'r'; utilities that write tapes (e.g., *idmdump*(1I)) default to 'a' on host tape (i.e., create a new file on the end of the tape) and 'w' on IDM tape (overwrite).

volume(VL) A comma-separated list of the names of the volumes in this set. If specified, the header of each tape is read and verified before the tape is used. If not specified any volume is accepted. Only the first volume is checked on IDM tape. Tape reads will always check volume names on tapes 2-n (but not 1).

fileset(FS) The name of the fileset to check. Host tape only. If not specified, the fileset name is not checked.

newname(V) The new volume name to write on the tape to replace the existing name. Can only be used in 'w' mode. If not specified, the volume name is unchanged. New IDM tapes (tapes not previously written by the IDM/RDMBS software) must be given a new name. IDM tape only.

fileno(N) The file number to access. Only used in read mode on IDM tape. If not specified file zero is assumed on IDM tape, or the *filename* is used on host tape. Note that files are numbered from zero on IDM tape and one on host tape. The *fileno* and *filename* must match if both are specified on host tape. This option is ignored when writing an IDM tape.

unit(N) The unit number to access. Zero by default.

density(D) The tape density in BPI. Host tape only (on IDM tape this is determined from the "configure" relation). If not specified, a system default is used.

length(L) The length of the tape in feet. Host tape only. Ignored on some systems. The UNIX implementation of *inittape*(8I) writes the tape length into a UVL1 label, which will override this parameter. The tape length is reduced by approximately 4% to allow for possible tape errors and variations in interrecord gap size.

bs(N) The (maximum) block size. Ignored in read mode if it can be determined from the tape header. Host tape only. If not specified, 2048 is used. Block sizes larger than 2048 exceed ANSI Standards X3.22-1978 and X3.39-1973 and hence may be incompatible with other systems.

format(F) The format of this file. Supported formats are 'F' for fixed length records and 'D' for variable length records. UNIX also supports 'U' for undefined; this format roughly resembles a stream. *Idmfcopy*(1I) defaults to format 'D' and *idmdump*(1I), *idmload*(1I) and *idmcopy*(1I) default to 'U' on UNIX.

erase Perform a "security erase" of the tape before writing. Only supported on some drives. Mode 'w' must be specified. IDM tape only.

xlate(X) Perform the requested translation of data on the tape. This may be one of "none" (no translation), "ascii" (translate to ASCII), "ebcdic" (translate to EBCDIC), "host" (do host translation). The default is "none." IDM tape only. Host tape is always host translated.

norewind Do not rewind tape between writing files. Default is to rewind. IDM tape only. Norewind is available for writes only in IDM Software Releases 35 and 40. Norewind applies to both reads and writes in RDBMS Software Release 3.5 and future RDBMS releases.

verify(B) Turn on (B = 1) or off (B = 0) tape sequence number verification. Default is not to verify. This parameter should only be used on tapes previously written by the IDM/RDBMS software on the database server. Like *volume*, tape reads will automatically verify the sequence numbers on tapes 2-n. IDM tape only.

AUTHENTICATION

If your shared database system is configured to require user authentication, you may be prompted for a password the first time the database is opened. The password can be set or changed using *idmpasswd*(1I).

On some systems it may be possible to set a default password. This will only be permitted if the password can be securely stored on the host.

NOTE

System V Release 2.0 (running on 3B series) does not provide access to basic tape operations. Therefore support of ANSI labeled tape (**htape**) is unavailable at this time.

SEE ALSO

idmpasswd(1I), getparam(3I), ifscrack(3I), itapeopts(3I), tf(3I), ifthfile(4I), iftifile(4I), iftltape(4I), csh(1), sh(1), tset(1)

NAME

idl – Ad hoc interactive IDL (Intelligent Database Language) parser

SYNOPSIS

idl [**–B** device] [**–P**] [**–e**] [**–a**] [**–f** infile] [**–l** linesperpage] [**–n**] [**–p**] [**–s**] [dbname]

ARGUMENTS

–B *device*	Use *device* as the connection to the database server.
–P	Turn on performance monitoring. Individual performance options can be set using the **set** pseudo-IDL command.
–a	Turn off auto-association. See *%associate* below.
–e	Echo every command as read. This can be useful when redirecting the input of the parser. In this case, the input commands as well as the replies will go into the output file.
–f *infile*	Input file name. If not specified, read the standard input in interactive mode.
–l *linesperpage*	Set the number of lines per page for output formatting. When data is being retrieved, a new header will be printed sufficiently frequently to insure that column labels are always visible. If *linesperpage* is zero, only the initial header will be printed. If not specifed, the terminal driver (*IftTerm*(4I)) is queried.
–n	Parse commands but don't execute them. The connection to the database server will not be opened. Front-end commands (e.g., *%input*) and range statements will still be executed. This can be used to verify an input script that is to be run later.
–p	Disable the reading of user and system profile (or startup) files.
–s	Run the parser in silent mode. Turns off prompting, printing of IDL banner and elaborate printing of syntax errors.
dbname	The name of the initial database to open.

DESCRIPTION

Idl implements the IDL query language. Queries typed at a terminal are translated, processed by the shared database system, and results are formatted and printed.

If the **–f** flag is specified, input is read from the named file rather than the standard input. File input is non-interactive, that is, special functions of interest only to the interactive user are disabled and input will be faster.

If the **–p** is not specifed, system and user profile files are read before user input begins. On UNIX, these are "/usr/lib/idm/idlpro.idl" and "~/.idlpro.idl" respectively.

The system parameter **MAPCC** may be used to pass control characters through the IDL front end. The default is to map control characters to blanks. See *params*(5I).

Auto association of stored command, relation, and view creation will place the user text into the *descriptions* relation of the current database using the **associate** command. Text starting at the end of the previous command up to and including the **end define** or command terminater (i.e., "go" or semi-colon) is stored in the *text* field, including comments and newlines, as it appears in the input. The *key* field of the relation has a value of iX where X ranges from 0 to 9 and a to Z to insure the sorting order of the text in the *descriptions* relation.

See the discussion of the **–a** flag, above, or the description of the *%associate* command, below. See BUGS section for warning about creating many objects within one "go".

The following list describes features of Britton Lee's IDL implementation.

- A "go" or a semicolon terminates all commands and sends them to the IDM/RDBMS software if no continuation character is set. If a continuation character is set (using the *%continuation* command – see below) then each line without a continuation character is sent immediately to the database server.
- The "exit" command exits idl.
- The "reset" command resets the command buffer like the "go" command but does not send the buffered commands to the database server.
- The "? [topic]" command invokes the help subsystem. See *helpsys*(3I).
- The "! [shcomm]" command invokes the system shell. See *sysshell*(3I).
- The commands **close**, **copy**, **dump database**, **dump transaction**, **load database**, **load transaction**, **open file**, **read file**, **write file**, **close file**, **roll forward**, **setdate**, and **settime** are not implemented here. Separate utilities provide these functions. See *idlparse*(3I) for details.
- The interrupt character (normally delete (a.k.a. rubout) or control-C on UNIX) can be used to interrupt a command.
- BCD numbers are preceeded by the '#' sign. (eg. "#1234.1234E−10"). BCD's may have 31 digits total with a decimal point embedded anywhere within the digits These digits are optionally followed by an 'E' or 'e' and an exponent from 1022 to −1023.
- Floating point constants must begin with a digit. For example, use "0.1" instead of ".1".
- The command **set** *option* causes the specified IDM system *option* to be set on all future commands. For example, "set 11" or "set CPU" causes database server CPU time to be returned. The **unset** command turns off options.

Commands

A number of front-end specific commands are available. These are all introduced with a percent sign at the beginning of a line and take effect immediately (i.e., are not buffered to a "go" command). Abbreviations are allowed for convenience.

%associate [*on* | *off*]
: If there is no argument or if the argument is **on** auto-association is enabled, so that the text description of stored commands is automatically entered into the database (using the **associate** command of IDL). If the argument is **off** then auto-association is disabled. Auto-association is normally on. See also the **−a** flag.

%continuation [*char*]
: Set the continuation character to *char*. Lines ending with the specified character are not sent directly to the parser. If this mode is set, the "go" command is not recognized; instead, the first line that does not end with the continuation character terminates the command. If the *char* parameter is omitted the "go" mode is reinstated.

  Britton Lee strongly discourages use of the continuation character. Inadvertently typing a carriage return before a command is complete may destroy data. You should use the default ("go" or semicolon) input mode.

%display *text*
: Output the *text* to the standard output. This is normally used in system profile files to provide informational messages to users.

%edit [*filename*]	Edit the transcript of the IDL session (or *filename* if given). When the editor returns, the file is submitted as input to IDL. The editor used is defined by the EDITOR parameter. See *params*(5I)
%experience *level*	Set the experience to *level*.
%help	Print all immediate commands.
%input [*filename*]	Read the specified *filename* for IDL commands. When the file ends (or an "exit" command is encountered) control returns to the standard input. If *filename* is not specified, the standard input is read.
%redo	Resubmit the transcript of the IDL session as input to IDL.
%showranges	Show the currently defined range variables.
%substitute *name value*	Assign the *name* to have the specified *value*. The "*%name*" syntax can be used to interpolate the value. This is a substitution, *not* a macro, so there are restrictions on where this substitution can occur. See *idlparse*(3I) for details. The value is typed as an iINT2 if the *name* begins with a digit, otherwise the value is typed as an iSTRING (iCHAR).
%trace *tracespec*	Send the *tracespec* to *tfset*(3I).
%?	Same as *%help*.

In addition to these commands, two special characters are recognized in the first position of a line. "?" invokes a help subsystem. It may be followed by a help topic, so "? idl append" describes the **append** command. A line beginning with the "!" character passes the remainder of the line to the UNIX shell.

EXAMPLE

idl –B /dev/gpib hostdb

Invokes IDL on the GPIB interface, database *hostdb*.

SEE ALSO

idlparse(3i), iftterm(4I), *IDL Reference Manual*, Britton Lee part number 205-1235-rev.

BUGS

If more than one **create** and/or **define** command is submitted to the parser at once, they are all *auto-associated* under the relation id of the first object.

There should be some way of controlling the format of the output. A *%format* command will probably be added to do this.

The output format should be better adapted to the terminal. For example, output lines that exceed the terminal width are not wrapped nicely. In particular, the current interface does not adapt nicely to IBM 3270-style interactions.

It should be possible to write scripts at this level that include looping based on return data so that simple applications can be prototyped easily.

In general, there should be a very sexy applications development tool available that would include report capabilities, simple applications generators, etc.

NAME

Idel2ric – convert .idm (IDEL precompiler input) syntax to .ric (RIC precompiler input) syntax

SYNOPSIS

idel2ric [**–r**] pgm.idm...

DESCRIPTION

Idel2ric converts files that were written in the dialect used by the old *idel* precompiler into the dialect used by the current *ric* precompiler. For each *idel* source file *pgm.idm* a corresponding *pgm.ric* file is generated.

If an argument of *–r* is given, the generated files will have the suffix *.rc*, rather than the suffix *.ric*. These files can be precompiled by the *rc* precompiler, a predecessor of *ric*. The option has no effect on the contents of the generated files.

The source *.idm* files should not provoke any diagnostics from *idel*. *Idel2ric* assumes its inputs are valid *idel* files, and is relatively weak at recovering from syntax errors.

The changes made are the following:

- Semicolons are added to query language statements;
- C variables embedded in database statements get a dollar-sign ("$") prefix;
- Leading dollar signs ("$") are stripped from continuation lines and lines containing only curly brackets.

The generated files will need further work before they are ready to run. In particular, you should edit them to make sure that the first executable statement in the program is **INITRIC** ("yourprogname") and that the last executable statement in the **main** procedure is an **exit**(RS_NORM).

For a more complete conversion from Release 2 I/O and Standard I/O to Release 3 I/O *R2toR3* may be used instead of *idel2ric*. *R2toR3* will call *idel2ric* (without the –r flag).

EXAMPLES

To convert an *idel* program *myguy.idm* use the command

```
idel2ric myguy.idm
```

Edit the file *myguy.ric* to make sure the **INITRIC** and **exit** are in place and checking out any lines containing the string **%%%**.

SEE ALSO

r2tor3(1i)

DIAGNOSTICS

Some *idel* syntax errors are diagnosed, but the effort made is pretty feeble.

BUGS/DEFICIENCES

Lines with detected *idel* syntax errors evaporate, rather than being passed on to the output file.

User variables used in order by clauses are not converted to the correct relation domain. This is left for the user to correct by hand.

NAME

idmckload – verify database or transaction log data files

SYNOPSIS

idmckload [**–B** device] [**–P**] [**–l**] wdbname srcspec

ARGUMENTS

–B *device*	Use *device* as the database server connection. See *intro*(1I) for details.
–P	Turn on performance monitoring.
–l	If specified, a transaction log file is verified; otherwise, a database file is verified.
wdbname	The working database. If an IDM file is specified in *srcspec* it will be found in this database.
srcspec	The specification of the input file (see *intro*(1I)).

DESCRIPTION

Idmckload verifies a database or a transaction log as previously dumped by *idmdump*(1I).

WARNING

This utility uses with option 28 to the command. If the error "bad with option option: 28" is returned, then the database server does not have code to support this utility. The mimimum requirement is D3.5 RDBMS software.

EXAMPLES

idmckload system %itape

Verifies a database data file from IDM tape file 0.

idmckload system "%itape,fileno(1)"

Verifies a database data file from IDM tape file 1. Since IDM tape files are numbered sequentially from zero, this is actually the second file on the tape.

idmckload –l system tuesday.log

Verify transaction log from the host file "tuesday.log."

SEE ALSO

intro(1I), idmcklog(1I), idmload(1I), idmdump(1I), idmrollf(1I), backup(8I), The section "Backup and Restore" in the *Database Administrator's Manual*

NAME

idmcklog – verify IDM transaction logs are complete and print their time stamps.

SYNOPSIS

idmcklog [**–B** device] [**–d** dbname] loglist

ARGUMENTS

–B *device*	Use *device* as the database server connection.
–d *dbname*	If set, will print the current time stamp for database *dbname*.
loglist	List of log file specifications to check.

DESCRIPTION

Idmcklog outputs the timestamps found in the headers of the specified transaction logs and reads the complete log to verify that the last page is present. Logs that are missing the last page (which may have occured from a user interrupt, host system crash, or database server crash) will generate an **Error** exception.

If the **–d** flag is given, the system database is queried for the current time stamp (found in the *databases* relation) for *dbname*. The ending stamp for the last transaction log dumped should match this value. Read permission of the system database is required to use this option. *Idmcklog* requires one or more transaction log file specs (see *intro*(1I) for file specs) which may be host file or tape specifications.

Currently transaction logs on the database server (IDM files and IDM tape) cannot be checked. The log could be dumped to host file or host tape and then *idmcklog* run on the host copy of the log.

EXAMPLES

idmcklog –d mydb log1 log2 log3%htape

Check transaction logs *log1*, *log2*, and *log3* (on host labelled tape). The time stamp for database *mydb* will be printed out along with the time stamps for the three logs.

SEE ALSO

intro(1I), idmdump(1I).

NAME

idmcopy – copy relation(s) between the database server and the host

SYNOPSIS

idmcopy in|out [**–B** device] [**–P**] [**–f** filespec] [**–w** wdbname] [**–l**] [**–n**] [**–p**] dbname [reln ...]

ARGUMENTS

in|out Copy direction relative to the database: **in** means to copy into relations; **out** means to copy out of relations.

–B *device* Use *device* as the database server connection.

–P Turn on performance monitoring.

–f *filespec* Copy all relations to or from the given *filespec* (see *intro*(1I)). IDM files are not supported. If not specified, host files named *reln*.**d** are used.

–w *wdbname* Use *wdbname* as the working database. Currently the main purpose for this is to use a different IDM tape permission.

–l Rather than locking the entire relation during copy in, only lock the page being modified.

–n Do not verify pages as they are copied in. Verification is performed only when copying in data pages.

–p Copy out the data in IDM system internal page format. This is more efficient than the standard (tuple) format. Copy in does not require this flag. the IDM/RDBMS software will recognize the format as the data is written to the database server.

dbname The name of the database in which to find the relations to be copied. If **–w** is not specified, this is also the working databse.

reln... The list of relations to be copied. If not specified, all user relations are copied.

DESCRIPTION

Idmcopy copies relations in or out of the shared database system. If any *reln*s are listed, then they are copied; otherwise, all user relations (i.e., objects of type 'U') in the database are copied. If the **–f** flag is specified, the relations are all copied to or from the named file. Otherwise, host files named *reln*.**d** are used. Note that for *idmcopy in* the *reln* names may contain the trailing **.d** so that pattern matching (*.d) can be used (see example below).

If *idmcopy in* is specified, data is copied from *filespec* or *reln*.**d** files to the shared database system relations. The relations will be created in the database if they do not already exist. The *reln* must match the name of the relation copied out. *Idmcopy out* copies data from shared database system relations to the named *filespec* or *reln*.**d** files in standard copy (tuple) format. If the **–p** option is specified then data is copied in IDM system internal page format. A "*:user*" may be used on relation names. If so, this tag is removed before the ".d" is appended. For example, a relation spec of "parts:user" references host file "parts.d."

Idmcopy to or from IDM files is not supported at this time. To copy IDM files, use *idmread* or *idmwrite* (see *idmread(1I)*).

EXAMPLE

idmcopy out –p –f backup hostdb

Make a copy of all user relations in *hostdb* in the host file *backup* formatted as IDM system internal pages.

idmcopy in hostdb wines stores
Copy host files *wines.d* and *stores.d* into relations *wines* and *stores*, respectively.

idmcopy in mydb *.d
Copy all files ending with *.d* in the current directory into the database *mydb*.

BUGS

Idmcopy out of two relations with the same name but different owners will use the same host file; the second will override.

SEE ALSO

intro(1I), idmfcopy(1I), idmread(1I), *System Administrator's Manual*

NAME

idmdate – set the date and/or time on the shared database system

SYNOPSIS

idmdate [**–B** device] [**–P**] [**–d**] [**–t**] [daytime]

ARGUMENTS

–B *device*	Use *device* as the database server connection.
–P	Turn on performance monitoring.
–d	Do not set the date.
–t	Do not set the time.
daytime	The date and time in free format. The syntax is described in *parsedate*(3I). If not specified, the host date and time are used. This must be a single parameter, so it will have to be quoted if it contains spaces.

DESCRIPTION

Idmdate sets the date and time for the shared database management system. If no *daytime* is given the date and time are collected from the host.

If **–d** and **–t** are both given, *idmdate* does nothing.

Only the DBA of the system database may use this command.

EXAMPLES

idmdate

Set the date and time on the shared database system to the current date and time on the host.

idmdate –d 5:32pm

Set the time on the IDM to 5:32 P.M.; do not change the date.

BUGS

Since dates are represented in GMT, the "day" as represented by the IDL **getdate** function can wrap at strange times (e.g., 4 P.M. on the West coast).

SEE ALSO

parsedate(3I)

NAME

idmdump – dump database and transaction log

SYNOPSIS

idmdump [**–B** device] [**–P**] [**–l** logname] [**–w**] [**–m** mode [clock,waitcnt]] [**–d** dbspec] **–t** logspec dbname wdbname

ARGUMENTS

–B *device*	Use *device* as the database server connection.
–P	Turn on performance monitoring.
–l *logname*	Specify the name of the transaction log on the shared database system. The default is "transact". This argument has no effect if the **–d** flag is specified, since only **transact** can be dumped during a full database dump.
–d *dbspec*	Set the database destination file specification (see *intro*(1I)).
–m *mode*	Set the dump mode, only when dumping both the transaction log and the database. Legal *modes* are **r** read only and **o** online or read/write dump. The **o** mode will reverse the order in which files are written (see below).
–w	Wait until the database is accessible to commit updates, rather than returning errors to the updating program. Ignored when using the online dump flag.
–t *logspec*	Set the transaction destination file specification. (see *intro*(1I)).
dbname	The name of the database to dump.
wdbname	Destination database to dump into if dumping within the database server (IDM file or IDM tape); *wdbname* must be different from *dbname*. *Wdbname* may be the same as dbname unless the user is dumping to IDM file.

DESCRIPTION

Idmdump dumps the transaction log of database *dbname* to an IDM file, a host file, IDM tape, or host ANSI labelled tape. The database may also optionally be dumped.

The database to be dumped is specified as *dbname*. A "working database" *wdbname* must also be specified; it must not be the same database as *dbname* if dumping to an IDM file. IDM files will be created in the working database.

The transaction and database destinations are controlled using the **–t** and **–d** flags, respectively. The **–t** flag must be specified since the transaction log must always be dumped; the database dump (**–d** flag) is optional.

Idmdump without a **–d** flag causes only the transaction log from the database *dbname* to be dumped into the file given in *logspec*. The log name is assumed to be "transact" unless specified by the **–l** flag.

If a **–d** flag is also specified, then the transaction log is dumped as above, followed by the database. The database is dumped to the file specified in *dbspec*. **Note** that if the **–mo** flag is specified the database is dumped first. The database is locked during the dump (unless using the **–mo** option), that is, no other users may update the database during the dump.

If either the log or database is going to IDM tape, then the other cannot be going to the host. An error will be reported if this is attempted. If both the log **and** the database are going to IDM tape, then the tape parameters must be put on the **–t** flag.

The mode flag **–m** sets either the read only (**–mr**) or the online (**–mo**) dump options available in Release 40 or newer RDBMS software. Online dump writes pages that are not being modified, keeping track of those marked as "dirty". Successive passes are made to write all dirty pages. The number of these passes and their frequency are user-definable as described below.

The online mode will accept optional clock time and wait count to be used by the database server in checking for updates not yet complete in the database. The syntax is two comma-separated integers: <*clock*>[, <*waitcnt*>].

The clock value for an online dump specifies the number of seconds which ***idmdump*** should wait between passes over the database. The clock value may range between 1-540 seconds and defaults to 60 seconds.

The wait count is the number of passes to make over the database before new updates are locked out and may range from 1-20 passes. The default is to make 5 passes.

A warning message is issued when using these options on a transaction only dump. For more detailed information read the IDM System Status Document for IDM Release 40.

> **NOTE:** the order in which the files are written are reversed. This could create a problem when loading the database from host or IDM tape if the user is not aware that the database is now the first file on the tape. If both the log and the database are to idm or host tape then the first file written to tape is the database followed by the transaction log.

Idmdump **–w** will suspend the database dump until all updates active in the database to be dumped have finished. Normally, if there are updates active, the dump will exit and print an error message. This flag is ignored if using online dump (**–m**) or when dumping the transaction log only.

The **–m** and **–w** options are legal only when dumping the database.

A *dbspec* or *logspec* specifying an IDM file creates an object of type "T" for the log and type "F" for the database. These can only be read by *idmload*(1I), *idmckload*(1I), *idmrollf*(1I), or the **audit** command. In particular, the *idmread*(1I) command cannot be used to read a dumped transaction log.

EXAMPLES

idmdump –d dbdest –t /dev/null parts system

Dump database "parts" to the host file "dbdest". The transaction log is not saved.

idmdump –t log%hfile –d db%ifile –mo80,10 parts system

The transaction log is dumped to the host file and the database to IDM file with the online option set. The clock value is 80 and the waitcnt 10.

idmdump –t log%hfile –d db%ifile –mo80 parts system

The transaction log is dumped to the host file and the database to IDM file with the online option set. The clock value is 80. The waitcnt defaults to 10.

idmdump –t logsave parts system

The transaction log is dumped to the host file "logsave".

idmdump –t logsave%ifile –l log1 parts backup

The transaction log is dumped to the IDM file "logsave" in database "backup" from IDM file "log1" in database "parts".

idmdump –d %htape –t partslog%ifile –B /dev/gpib parts system

Dump database "parts" using GPIB parallel driver to host tape. The transaction log is saved in IDM file "partslog" in the system database.

idmdump –t tr%ifile –d "%htape,bs(8192),density(1600)" parts backup

Dump transaction log to IDM file "tr" in the database "backup" and dump database "parts" to host tape where the tape block size is 8K and density is 1600 bpi. (Note: block sizes exceeding 2048 may not be available on all systems.)

idmdump –t "%itape,volume(old),newname(new)" –d %itape employees system
Dump database "employees" and the transaction log on IDM tape. Check the name on the tape first and make sure that it is equal to "old" then replace it with "new." Note that when using IDM tape the database destination name is not needed.

idmdump –t "elog%ifile,newname(new),mode(a)" –d %itape employees system
Dump database "employees" to the end of IDM tape with the transaction log to IDM file "elog".

BUGS

If the dump is suspended when using the **–w** flag and updates are not yet complete, the user does not receive error messages until the dump resumes.

SEE ALSO

intro(1I), idmcklog(1I), idmckload(1I), idmload(1I), backup(8I), The section "Backup and Restore" in the *Database Administrator's Manual*

NAME

idmfcopy – format and copy data to or from a relation

SYNOPSIS

idmfcopy in [**–B** devname] [**–P**] [**–b**N] [**–d** dataspec] [**–e**N] [**–f** formfile] [**–l**] [**–n**] [**–r** rejectfile] [**–s**N] [**–v**] [**–w**] [dbname [relname [formdesc]]]

idmfcopy out [**–B** devname] [**–P**] [**–d** dataspec] [**–e**N] [**–f** formfile] [**–v**] [**–w**] [dbname [relname [formdesc]]]

ARGUMENTS

in | out — If **in**, data is copied from the host to the shared database system. If **out**, data is copied from the shared database system to the host.

–B *device* — Use *device* as the connection to the database server.

–P — Turn on performance monitoring.

–b *N* — Copy records in batches of *N* records, and commit the copy of each batch automatically. If the system crashes during a long copy, records that were committed can be skipped (using the **–s** flag). If **–b** is not specified, records will be committed in batches of 5000. *Idmfcopy in* only.

–d *dataspec* — The specification of the host data file. Host files or host tapes may be specified. If not specified, a **file** statement in the specification is used. If that does not exist either, then by default standard input is used on copy in and standard output is used on copy out.

–e *N* — Stop processing after *N* errors have been encountered. The default is to never stop on error. *Idmfcopy in* only.

–f *formfile* — The name of a host file containing a description of the format of data in *dataspec*.

–l — Rather than locking the entire relation during copy in, only lock the page being modified.

–n — Check data, but do not copy it. Data format descriptions and input records are checked, but no data is transferred to the shared database system. This option is useful for debugging file descriptions and cleaning up input data. *Idmfcopy in* only.

–r *rejectfile* — The name of a host file to receive copies of records from the *dataspec* that do not match the format specification. Duplicate records deleted by the IDM/RDBMS software will not be included in *rejectfile*. This option applies to *idmfcopy in* only.

–s *N* — Skip the first *N* input records. *Idmfcopy in* only.

–v — Verbose mode. Data transferred to or from the relation is formated as a table and written to standard error. This can be useful for debugging file descriptions.

–w — Ignore warnings. Records that have warnings instead of errors (e.g., conversion overflows) will be copied into the database; otherwise, the record will be rejected to the reject file and included in the error count. *Idmfcopy in* only.

dbname — The name of the database containing the relation to copy. Overrides a **database** statement in the specification.

relname — The name of an existing relation. The relation name on the command line overrides a **relation** statement in a *formfile*.

formdesc The description of the format of data, if not specified by a *formfile*. Must be quoted on most systems.

DESCRIPTION

Idmfcopy converts and copies data to or from an external form described by *formfile* or a *formdesc*. Only the fields in the format description will be copied; it is not necessary to copy all fields in the relation. Those fields not copied are filled with the appropriate NULL value depending on the type of the field.

Idmfcopy in reads records from a host file, converts to internal (IDM system) format, and loads the database server.

Idmfcopy out reads tuples from the shared database system, converts them to external format, and writes records to a host file.

External (host) records are defined in one of three ways:

- *Physical records* are defined by the underlying file system. This includes fixed length records on stream-based files and operating-system defined variable length records.
- *Delimited records* terminate at a specific record delimiter character. For example, data formated as lines of text on UNIX terminate at a newline character.
- *Field-driven records* simply gather enough data to fill all the component fields. The use of this record type is strongly discouraged, as it is inefficient and reduces error recovery dramatically.

Records may not exceed 4096 bytes in length.

Records are composed of fields. Fields are defined in one of the following ways:

- *Fixed length fields* consume a predetermined number of bytes. The data may be text or binary.
- *Delimited fields* consume bytes until a specified delimiter character. The delimiter is consumed, but is not sent to the shared database system. The data is always text.
- *Counted fields* begin with either a single byte or two bytes that is interpreted as a binary length followed by that many bytes of data. This is normally used only for special IDM system types such as BCD.

If all fields are text types the file will be opened with **type(text)** by default (this can be overridden by specifying **type(binary)** in the file spec).

The external format is defined in *formfile* or on the command line using the *formdesc* parameter. The syntax is as follows:

<description> ::= { <statement> }+
The description is a sequence of statements.

<statement> ::= **database** <name> ;
The name of the database to access. A *database* on the command line overrides this statement.

<statement> ::= **relation** <name> ;
The name of the relation in the database server to be copied. A *relname* on the command line overrides this statement.

<statement> ::= **file** <filespec> ;
<filespec> ::= <name>
The specification of the host file in *ifscrack*(3I) format. The specification will normally need to be quoted. A *filespec* parameter on the command line overrides this statement.

<statement> ::= **delimiters** <delims> ;
The default set of field delimiters. If not specified, tab, comma, and newline are the default field delimiters.

<statement> ::= **verbose** ;
Turns on verbose mode (i.e., the same as the –v flag).

<statement> ::= **record** <extent> { <fieldspec> ; }+ **end**
This statement describes the internal structure of a record. It consists of a definition of the record followed by an ordered sequence of field specifications.

<fieldspec> ::= <attname> <typespec> [= <value>]
Every field has a name, a type (describing the type in the external file, not in the database), and an optional initial value.

<attname> ::= <name> | **all** | –
A name may be specified explicitly, which matches the attribute of the same name in the relation, specified as the keyword **all** to indicate all domains in the relation, or specified as dash for dummy fields. On *idmfcopy in* dummy fields are discarded; on *idmfcopy out* dummy fields are created.

<typespec> ::= <binspec> | <textspec>

<binspec> ::= <fixedbinspec> | <varbinspec>
<fixedbinspec> ::= **i1** | **i2** | **i4** | **f4** | **f8**
<varbinspec> ::= **bcd** <length> | **bcdflt** <length> | **bin** <length>
Binary specifications represent data that is stored in IDM system internal format. These are not recommended for use in interchange. Types **bcd**, **bcdflt**, and **bin** require a length specification (see below). The length on **bcd** and **bcdflt** is in bytes, not digits, and the data stored does not include a type or length byte.

<textspec> ::= <texttype> <extent>
Text types describe representations that have been rendered into the printable character set. These are in general usable for interchange with other operating systems and database systems. The <extent> field defines the size of the field.

<texttype> ::= **text** | <inttype> | <floattype>
<inttype> ::= [**unsigned**] **decimal** | **octal** | **hex**
<floattype> ::= **float** | **sci**
For character domains **text** represents a byte-by-byte copy. For integer numeric domains **text** is equivalent to signed **decimal**. For floating numeric domains **text** is equivalent to **float**. The <inttype>s must match an integer domain (i1, i2, or i4) and force interpretation in the indicated radix. The <floattype>s must match a floating point domain (f4, f8, bcd, or bcdflt). On output, type **sci** causes output in exponential notation. On input, types **float** and **sci** are identical.

floating point numbers at the maximum representable value may give a float point overflow error when copied in. To avoid this, reduce the precision on output to ensure that the number will correctly convert during copy in.

<extent> ::= [<length> | **to** <delims>]
<length> ::= (<integer> [, <precision>]) | (*) | (**var**)
<precision> ::= <integer>
A length specifies the total number of bytes consumed by a record or field. If an integer is specified, the field or record is fixed length, consuming or producing the number of bytes specified. The asterisk syntax indicates *counted* field

format is used for fields. The first byte of the data describes the width of the field. The **var** syntax indicates *counted* field format with two bytes of data describing the width of the field. The byte ordering is most significant followed by least significant.

An optional <precision> specifies the number of digits after the decimal point for **float** or **sci** output; in other contexts it is either ignored or illegal.

A delimiter list specifies delimiter characters that will cause input to end. On *idmfcopy out* the first delimiter specified is used to terminate the field. If neither length nor delims are specified, then a variable length string delimited by a default set of delimiters is assumed for fields. The rules for records are described below. On output the first delimiter specified is used.

<delims> ::= <delimiter> { , <delimiter> }*

<delimiter> ::= <identifier> | <integer> | <string>

Delimiters may be represented as a symbolic name, as a numeric value, or as a string. For example, the specifiers **comma**, ",", 0o54, and 0x2c all represent the same delimiter on ASCII-based machines. See below for a list of the symbolic names.

<value> ::= <integer> | <string>

Values specify verification or initialization of external fields. On *idmfcopy out* the <attname> must be '-' and the resultant output field contains the specified value. On *idmfcopy in* the input field must exactly match the specified value.

<name> ::= <identifier> | <string>

Names that have no special characters may be given directly. If necessary, names can be quoted to hide special characters.

The following is a list of reserved words that must be quoted if they are to be used as names of fields in a database server relation:

all	**bin**	**bcd**
bcdflt	**database**	**decimal**
delimiters	**end**	**file**
float	**f4**	**f8**
hex	**i1**	**i2**
i4	**octal**	**record**
relation	**sci**	**text**
to	**unsigned**	**var**
verbose		

Comments begin with '/*' and end with '*/' as in C or PL/I.

Record formats are defined as follows:

record(<integer>)

Opens the underlying host file with the **rbp** (record-based presentation) parameter and the specified record length. On physically record-based files, this may specify a variable-length file. On physically stream-based files, this specifies fixed-length records.

record to <delimiter>

Opens the file as a stream. Data will be scanned for the specified delimiter.

record

If the underlying file is record based or if all fields in the record are fixed length, acts like "**record**(*N*)," that is, opens the file with **rbp** (record based presentation). The record length is the sum of the field

lengths if all are fixed lengths, or otherwise is a system default. Otherwise (on stream based files with variable length fields) acts like "**record to nl.**"

record(*) Opens the file as a stream. Fields are read or written piecemeal. Efficiency is lost, and error recovery is reduced. The *rejectfile* option is disabled with this mode.

For example, the input:

```
/* address records */
record(60)
        name        text(20);
        address     text(40);
end
```

specifies a file containing a collection of fixed length records, sixty bytes in length, containing names and addresses.

Offsets in the host file are implied by the order of the specifications. For example, in the above example, attribute 'name' is loaded from the data in positions zero through nineteen, and 'address' is loaded from positions twenty through fifty-nine.

Symbolic delimiters may be selected from the following list:

Name	*Graphic*	*ASCII*	*EBCDIC*	*Meaning*
null		000	00	Null
tab	\t	011	05	Horizontal Tab
nl	\n	012	15	Newline
lf		012	25	Line Feed
ff	\f	014	0C	Form Feed/New Page
cr	\r	015	0D	Carriage Return
fs		034	22	Field Separator
gs		035		Group Separator
rs		036	35	Record Separator
us		037		Unit Separator
space	(space)	040	40	Space
comma	,	054	6B	Comma
dash	-	055	60	Dash/Hyphen/Minus
dot	.	056	4B	Dot/Period/Decimal Point
slash	/	057	61	Slash
colon	:	072	7A	Colon
semi	;	073	5E	Semicolon

TYPE CONVERSION

Corresponding attributes in the relation and the host file do not have to be of the same type or length. *Idmfcopy* uses *typecnvt*(3I) to convert as necessary. Britton Lee's IDM/RDBMS software does not convert floating point numbers to a standard representation. Floating point numbers generated on one machine may not be meaningful if read on a machine of a different type.

Dummy fields, denoted by a name of "-", are not transferred to or from the shared database system. ***Idmfcopy out*** will write an empty field or a value (eg. = <value>) if specified. ***Idmfcopy in*** will read but discard dummy fields.

Idmfcopy in

When the ***direction*** is ***in***, *idmfcopy* appends data into the relation from the host file. Domains in the relation which are not assigned values from the host file are assigned the default value of zero for numeric attributes, and blank for character attributes. When copying in this direction the

following special meanings apply:

text — The data is a variable length character string terminated by any field delimiter character (comma, tab, or newline if not specified with the **delimiter** command). The delimiter is thrown away.

text to <delims> — The data in the host file is a variable length character string terminated by the delimiter *delim*. If more than one *delim* character is specified, any of the characters will terminate the string.

text(<integer>) — The data in the host file is exactly *integer* bytes long.

text(*) — The data in the host file begins with a single byte that contains the number of bytes of data when interpreted as a binary number. The count field does not include itself. Usage of this field type should be with fixed length records. Record delimiters with the same binary representation as a count byte will cause the record to be prematurely terminated.

text(var) — The data is the host file begins with two bytes that contain the number of bytes of data when interpreted as a binary number (most significant byte first). The count field does not include itself. Usage of this field type should be with fixed length records. Record delimiters with the same binary representation as a count byte will cause the record to be prematurely terminated.

For example:

pnum text; — A variable length string ending in the field delimiter character (tab, comma, or newline if not set with the field-delim option) is read from the host file. The delimiter is discarded and the string is converted to an integer and copied into the *pnum* attribute.

pnum text to ",";
A variable length string ending in comma is read. It is converted and copied into the *pnum* attribute.

pnum text to "\\";
A variable length string ending in the character '\' is read. It is converted and copied into the *pnum* attribute.

pnum decimal to ",", "/";
A variable length string ending in comma or slash is read. All characters in the string must be decimal digits or spaces. The string is then converted and copied into the *pnum* attribute.

all text; — all attributes of the relation appear in the input file as variable length strings ending in comma, tab, or newline.

- text; — a variable length string ending in comma, tab, or newline appears in the record, but is not transferred to the shared database system.

Idmfcopy out

When the direction is **out,** *idmfcopy* transfers data from the relation into the host file. Any field in the host file which is not assigned a value (the *attname* is -, and no literal *field-value* is specified), is assigned the default value of zero for numeric attributes, and blank for character attributes. When copying in this direction, the following special meanings apply:

text
: The attribute value is converted to a character string and written into the host file. For character attributes, the length will be the same as the attribute length as defined when the relation was created. Integer and bcd attributes are converted to decimal, and f4, f8, and bcdfloat attributes are converted to scientific notation. A comma (or the first field delimiter specified with the **delimiter** statement) is written after the field.

text to <delimiter>
: The attribute will be converted according to the rules for **text** above. The one character delimiter will be inserted immediately after the attribute. If the record type is "to <delimiter>" and the field delimiter matches the record delimiter in the last field of the record, the field delimiter will be suppressed so that only one copy of the delimiter will be output to the record.

text(<integer>)
: Exactly ***integer*** bytes are written to the output file. The field is padded with spaces or truncated as necessary to fit.

text(*)
: A byte is written giving the length of the field, followed by the field itself. The count byte does not include itself. Usage of this field type should be with fixed length records. Record delimiters with the same binary representation as a count byte will cause the record to be prematurely terminated on ***idmfcopy in*** .

text(var)
: Two bytes are written giving the length of the field, followed by the field itself. The count bytes do not include the two bytes of count. Usage of this field type should be with fixed length records. Record delimiters with the same binary representation as a count byte will cause the record to be prematurely terminated on ***idmfcopy in*** .

text = <string>
: ***String*** is written to the output file. A tab (or the first field delimiter specified with the **delimiter** statement) is written after the field.

Numeric fields represented in text fields are generated as specified in ANSI standard X3.42-1975.

For example:

pnum text;
: The integer in ***pnum*** is converted to a character string in decimal notation and written to the host file. The field delimiter character (tab if not set with the field-delim option) is written after the string.

pnum decimal;
: This is identical to the above example.

pcost text;
: The bcd float in the ***pcost*** attribute is converted to a character string in scientific notation. The field delimiter character is written after the string.

pcost float;
: The bcd float in the pcost attribute is converted to a character string in floating point decimal format. The field delimiter character is written after the string.

pnum text to ",";
: The integer in pnum is converted to a character string in decimal notation and written to the host file. A comma is written after the string.

EXAMPLES

Example 1:

```
idmfcopy in -f empl.fmt
```

emp1.fmt:

```
database   demo;
relation   emp;
file       myfile;
record
           name  text(10);
           sal   f4;
           date  i2;
           mgr   text(10);
           -     text(1);
end
```

copies data into the "emp" relation in the "demo" database from "myfile" on the host. "Myfile" contains a string field, a float field, a two byte integer field, a string field, and a one character field that is ignored.

Example 2:

idmfcopy out –f emp2.fmt demo emp

emp2.fmt:

```
file       outfile;
record to nl
           name  text to ":";
           sal   decimal to nl;
end
```

copies employee names and their salaries to standard output. The name field is followed by a colon. Records are terminated by newlines. For example, the output may look like:

```
Fred:10000
Joe:12000
Sam:52000
```

Outfile will be opened with **type(text)**.

Example 3:

idmfcopy in demo –f parts.fmt

parts.fmt:

```
relation   parts;
file       "xyzdata%htape,unit(1)";
record(80)
           pnum  text(5);
           pname text(20);
end
```

Reads data from host ANSI tape on unit 1 into the parts relation. The ***pnum*** domain comes from the first five bytes of each eighty-byte record; the ***pname*** domain comes from the next twenty bytes. The remaining fifty-five bytes are ignored.

Example 4:

idmfcopy in demo parts 'record to nl all text; end'

Reads records from the standard input into the "parts" relation in database "demo"; each record is on one line in the external file, with fields separated by commas or tabs.

SEE ALSO

intro(1I), idmcopy(1I), typecnvt(3I), ANSI X3.42-1975, *Idmfcopy User's Guide, American National Standard Representation of Numeric Values in Character Strings for Information Interchange.*

BUGS

The reject file is always opened as a stream with default parameters.

NAME

idmhelp – access the IDM Help Subsystem

SYNOPSIS

idmhelp [*topic*]

ARGUMENTS

topic The topic for which help is desired. If not specified, the user is placed at the top of the help tree.

DESCRIPTION

Idmhelp is a menu-based help facility for users of Britton Lee's Shared Database System. It shows proper command syntax, gives the meaning of command-line arguments, and describes available features.

The help system is a tree-structured collection of topics. Each topic has some explanatory text and zero or more children associated with it. The user is shown the text for the current topic, and presented with a list of subtopics.

The user may ask for information on a subtopic by typing its name. The following commands are also recognized:

%EXIT

Exit *idmhelp*.

%UP Move up the help tree to the topic immediately above the current one.

%TOP

Move directly to the top of the help tree.

Commands and topic names may be entered in either upper or lower case.

EXAMPLE

idmhelp idl.append

Enter the help system, starting with the description of the IDL **append** command.

SEE ALSO

helpsys(3I)

NAME

idmload – load database or transaction log

SYNOPSIS

idmload [**–B** device] [**–P**] [**–l** logname] dbname wdbname srcspec

ARGUMENTS

–B *device*	Use *device* as the connection to the database server. See *intro*(1I) for details.
–P	Turn on performance monitoring.
–l *logname*	The name of the transaction log in *dbname*. If specified, a transaction log is loaded; otherwise, an entire database is loaded.
dbname	The name of the database to be loaded (if –l is not specified) or the database in which to place the loaded transaction log.
wdbname	The working database. If an IDM file is specified in *srcspec* it will be found in this database. *Wdbname* must be specified and differ from *dbname*.
srcspec	The specification of the input file (see *intro*(1I)).

DESCRIPTION

Idmload loads a database or a transaction log as previously dumped by *idmdump*(1I). If –l is specified a transaction log is loaded, otherwise a database is loaded.

After a transaction log is loaded into a database it can be applied using *idmrollf*(1I), that is, the updates described by the log can be run again. If this is intended the log must be loaded into a different database than that which is to be rolled forward.

EXAMPLES

idmload db system %itape

Load database "db" from IDM tape file 0.

idmload db system "%itape,fileno(1)"

Load database "db" from IDM tape file 1. Note that IDM tape files are numbered sequentially from zero, so tape file one is the second file on the tape.

idmload –l newlog db system tuesday.log

Load log "newlog" into database "db" from the host file "tuesday.log." The usual next step would be the command "idmrollf targetdb db newlog" to roll forward "targetdb" from newlog.

WARNING

Using the *online* option to *idmdump* will cause the order of the files written to be reversed. The database is written as the first file and the transaction log is written as the second file. This is most significant when using host or IDM tape.

SEE ALSO

intro(1I), idmckload(1I), idmcklog(1I), idmdump(1I), idmrollf(1I), backup(8I), The section "Backup and Restore" in the *Database Administrator's Manual*

NAME

idmpasswd – set password in the shared database system login relation

SYNOPSIS

idmpasswd [**–B** device]

ARGUMENTS

–B *device* Use *device* as the connection to the database server. See ***intro***(1I) for details.

DESCRIPTION

Idmpasswd resets the password for the current user as stored in the login relation in the system database. The user must specify both old and new passwords.

SEE ALSO

intro(1I), The section "System Level Security" in the *System Administrator's Manual*

NAME

idmread, idmwrite – read/write files between the host and the shared database system

SYNOPSIS

idmread [**–B** device] [**–c** count] [**–o** offset] database idmfile [destspec]

idmwrite [**–B** device] [**–c** count] [**–o** offset] database idmfile [srcspec]

ARGUMENTS

–B *device*	Use the specified IDM *device*, instead of the default, to connect to the database server.
–c *count*	A maximum of *count* bytes will be copied. If omitted, the entire file will be copied.
–o *offset*	Start copying from the byte *offset* in the IDM file. The host offset is always zero. Gaps in the IDM file caused by offsets have undefined values.
database	The database name to operate in.
idmfile	The name of the IDM file to access.
srcspec	The specification of the source of an *idmwrite* (see *intro*(1I)).
destspec	The specification of the destination of an *idmread*.

DESCRIPTION

Idmread reads the file IDM file *idmfile* in *database* to *destspec*. If *destspec* is not specified, the file is written to the standard output.

Idmwrite writes *srcspec* (or the standard input if not specified) to file *idmfile* in the specified *database*.

The sense of "read" and "write" is always with respect to the IDM file.

Neither *srcspec* nor *destspec* may specify an IDM file.

EXAMPLES

idmwrite –c 10000 db igetdone igetdone.c

Write the first ten thousand bytes of host file "igetdone.c" into the IDM file "igetdone" in database "db."

idmread db igetdone %itape

Read the IDM file "igetdone" in database "db" and write to IDM tape.

SEE ALSO

intro(1I), iftifile(4I), *System Programmer's Manual*

NAME

idmrollf – roll forward a transaction log

SYNOPSIS

idmrollf [**–B** device] [**–P**] [**–d** enddate] [**–v**] dbname wdbname **logname**

ARGUMENTS

–B *device*	Use *device* as the connection to the database server. See ***intro***(1I) for details.
–P	Turn on performance monitoring.
–d *enddate*	Do not run any updates in the transaction log dated after *enddate*. This essentially leaves all logged relations in database *dbname* in at the same state they were in at *enddate*. *Enddate* can be entered in free format (see *parsedate*(3I)).
–v	Print more information during rollforward. Useful when a date is specified and the user wants to check the idm day and ticks value after conversion.
dbname	The database to roll forward.
wdbname	The working database. This must not match *dbname*.
logname	The name of the transaction log in *wdbname*. *Logname* must have previously been created by *idmdump*(1I) or *idmload*(1I).

DESCRIPTION

Idmrollf applies the transaction log *logname* to database *dbname*. All updates logged in *logname* (created by an **idmload –l**) are re-executed against the database *dbname*.

Times as represented by the shared database system in the **audit** command may be input directly using the *idmtime*(idmdate,idmticks) syntax for *enddate*.

EXAMPLES

idmrollf vino system vinolog

Roll forward database *vino* using the tranaction log *vinolog* in the database *system*.

idmrollf –v –d "idmtime(31480,60000)" vino system vinolog

Roll forward database *vino* using the tranaction log *vinolog* in the database *system* up to IDM day 31480 at IDM time 60000. This date translates to Mon March 10, 16:16:40 1986.

SEE ALSO

intro(1I), idmdump(1I), idmload(1I), parsedate(3I), backup(8I), The section "Backup and Restore" in the *Database Administrator's Manual*

NAME

R2toR3 – convert Release 2 source to Release 3

SYNOPSIS

R2toR3

DESCRIPTION

R2toR3 is a shell script which converts release 2 source files to release 3 source files.

Directories *NEW* and *OLD* are created in the directory where *R2toR3* is invoked. Converted copies of all **.c*, **.y*, **.idm*, **.ric*, and **.rc* files are put into *NEW* and the originals are copied to *OLD*. *R2toR3* will run *idel2ric* on any *.idm* files first. This will create an output file ending with *.ric*. The *.ric* file will be copied to *NEW* and the original file will be copied to *OLD*.

The changes applied to the source files are as follows:

- Release 3 *#include* files are added to the beginning of each file. If the file is yacc source, move these *#includes* to their proper place in the file.
- Release 2 and system dependent include files are deleted or modified. These are:

bcd.h	<deleted>
ctype.h	→ bytetype.h
done.h	→ idmdone.h
idmio.h	<deleted>
options.h	<deleted>
setexit.h	<deleted>
setjmp.h	<deleted>
stdio.h	<deleted>
symbol.h	→ idmsymbol.h
useful.h	<deleted>

- Runtime system calls (eg. fetch) are renamed according to Release 3 (eg. irfetch).
- Standard I/O and Release 2 I/O calls are replaced with calls to Release 3 I/O routines.
- Signals, setexit, setjmp and longjmp are replaced by an exception raise or the setting of an exception handler. See **exc**(3I) for more information on the exception facility.

  Only Release 2 oriented setjmps and longjmps are converted to exceptions.

  Signals are converted as:

signal(SIGINT, SIG_IGN)	→ exchandle("T:IDMLIB.ASYNC.INT", excignore)
signal(SIGINT, handle)	→ exchandle("T:IDMLIB.ASYNC.INT", handle)
signal(SIGHUP, handle)	→ exchandle("T:IDMLIB.ASYNC.INT", handle)
signal(SIGTERM, handle)	→ exchandle("T:IDMLIB.ASYNC.TERM", handle)
signal(SIGALRM, handle)	→ exchandle("T:IDMLIB.ASYNC.ALARM", handle)

  All other signals, setjmps and longjmps are commented out with a **%%%** in the comment for later correction.

- A line of the form **INITIDMLIB**("%%%progname%%%"); will be added if a call to **crackargv**() already exists. If not, add this call by hand, replacing the %%%progname%%% with the name of your program. This must be the first executable statement in the program.
- Check that **exit** is called when leaving your program so that output will be flushed.
- Release 2 defined constants are replaced with Release 3 semantically equivalent constants. This includes defines such as token types (INT4 → iINT4) and done status bits

(DONE_CONTINUE → ID_CONTINUE).

- Release 2 structures and field names are converted to their equivalent in Release 3.
- Ctype macros (e.g. isalpha) are converted to bytetype macros (e.g. ISALPHA).

The converted files in *NEW* may need further work before they are ready to be compiled. In particular, you should edit the files to make sure that the first executable statement in the program is **INITIDMLIB**("yourprogname") and that the last executable statement in the **main** procedure is an **exit**(RS_NORM). Also check out any lines containing the string **%%%** in a comment; they mark changes which may require more work.

Makefiles will have to modified if they used *idel*.

SEE ALSO

idel2ric(1i)

NAME

ric – precompiler for embedding IDL in C

SYNOPSIS

ric [**–d** database [**–B** device] [**–n** progname]] [**–S** symtabsize] [**–l**] [**–q**] [**–V**] [file.ric ...]

ARGUMENTS

–d *database*	Database to use.
–B *device*	Use *device* as the database server connection. If not specified, the IDMDEV parameter is consulted. *Database* must also be specified.
–n *progname*	Use stored programs. Associate stored programs under *progname*. *Database* must also be specified.
–S *symtabsize*	Make the symbol table *symtabsize* elements large. The default size is 100 symbols.
–l, –q	Normally, the *#line* directives that *ric* writes look like this: *#line 3 "file.ric"*. If *-l* is specified, they will look like *# 3 "file.ric"*; if *-q*, like *#line 3 file.ric*; if both, like *# 3 file.ric*. See the *RIC User's Guide*.
–V	Prints the version number of the precompiler and the version number of IDMLIB used to make it on *stderr*.
file.ric ...	The file(s) to be precompiled.

DESCRIPTION

The precompiler *ric* takes file(s) with IDL commands embedded in C code and generates file(s) containing pure C. The embedded IDL is translated to appropriate calls into the Britton Lee library **IDMLIB**. After precompilation, there will be a file generated with the same name as the **.ric** source file, but with a **.c** suffix. This file is ready for compilation by the C compiler.

An input file name must either have a suffix of *.ric* or (for backward compatibility) a suffix of *.rc* or else have no suffix. A file name with no suffix is taken literally. If a directory contains the files *p* and *p.ric*, the command *ric p* will precompile *p* and not *p.ric*. If there are two input files named *x.ric* and *y*, then the two output files produced will be named *x.c* and *y.c*. If an input file name of – (a single minus) is given, then *stdin* is read and *stdout* is written. This allows *ric* to be used in pipelines.

The precompiled query language commands either may be kept within the object module (the default case) or stored in the database (if the **–n** flag is given). Storing commands in the database is much more efficient at execution time, but requires that the database schema not change during the program's lifetime. See the *RIC User's Guide* for a more complete discussion.

Programs that are precompiled with *ric* must link in the runtime library **idmlib**. See the examples below.

Example of precompiler source code:

```
main()
{
        $int    num;

        INITRIC("demo");
        $range of a is arelation;

        $retrieve($num=a.number) where a.name = "animal"
        {
                printf("%d\n", num);
```

```
        }
        exit(0);
}
```

To precompile:

```
ric -d mydb -n Xprog prog.ric
```

To compile:

```
cc -o prog prog.c -lidmlib
```

LANGUAGE SYNOPSIS

The following is a short synopsis of those IDL queries that may be used with *ric*.

$ abort transaction ;

$ append [**to**] object_name **(** target-list **)** [**where** qualification] **;**

$ associate { object-name | range . att_name } [**with**] string [, string] **;**

$ audit [**into** relation] **(** target-list **)** [**where** qualification] **;**

$ begin [**new** | **nest** n] **transaction ;**

Note that the modifiers *new* and *nest* are not part of interactive **idl**. They may only be used in embedded **idl**.

$ create relation **(** att_name = type ,... **)** [**with** options] **;**

$ create database dbname [**with** options] **;**

$ create [**unique**] [**nonclustered** | **clustered**] **index** [**on**] relation **(** att_name ,... **)** [**with** options] **;**

$ create view object-name **(** target-list **)** [**where** qualification] **;**

$ define queryname command ... **end define ;**

$ delete range [**where** qualification] **;**

$ deny protect mode [**of** | **on**] object-name [**(** att_name ,... **)**] [**to** user ,... **;**

$ destroy object-name ,... **;**

$ destroy (target-list **)** [**where** qualification] **;**

$ destroy database dbname ,... **;**

$ destroy [**nonclustered** | **clustered**] **index** [**on**] relation **(** att_name ,... **) ;**

$ end transaction ;

$ [**execute**] [**program**] query-name [[**with**] [**(**] [name =] value ,... [**)**]] { { | ; }

The sole purpose of following an **execute** with a bracketed series of statements is to associate it with one or more **obtain** commands (see below).

$ extend database dbname [**with** options] **;**

$ open dbname **;**

$ permit protect-mode [**of** | **on**] object-name [**(** att_name ,... **)**] [**to** user ,...] **;**

$ range of range **is** relation [**with** options] **;**

$ reconfigure ;

$ replace range **(** target-list **)** [**where** qualification] **;**

$ retrieve [**unique**] [**into** relation] **(** target-list **)** [**order** [**by**] order ,...] [**where** qualification] { { | ; }

$ set option_number ,... ;

$ sync ;

$ trace [**on** | **delete**] flag ;

$ truncate object_name ,... ;

$ unset option_number ,... ;

Two statements have been added that do not exist in interactive IDL.

An **$ obtain** statement has been added to allow the assignment of items retrieved by retrieve statements that are part of stored commands to be assigned to C variables. Its syntax is

$ obtain [( ] $C ,... [ )] { { | ; }

$C is defined below.

For instance, if a stored command named **foo** contained a **retrieve** statement that returned three items, then we might invoke it via the statements

```
$execute foo
{
        $obtain ( $a, $b, $c)
        {
                printf("From foo -- %d %d %d.\n", a, b, c);
        }
}
```

Obtain is a loop-controlling command, like **retrieve.** If an **obtain** is simply followed by a semicolon (";") rather than a bracketed sequence of statements, it still cycles through all the tuples returned by the **retrieve** statement, assigning them to the targets in turn, rather than just returning one tuple like the singleton **retrieve**. Thus a statement like

$ obtain ($(*p++)) ;

can be used to fill an array.

The new statement

$ cancel ;

cancels all activity in the shared database system on the current dbin and any dbins that are related to the current one as parent or child in a chain of reopens. Programs using **cancel** must be careful to exit any **retrieve** loop with **break** immediately:

```
$range of t is threeatt;
$retrieve ($att1=t.att1, $att2=t.att2, $att3=t.att3)
{
        ifprintf(stdout, "\t%d\t%s\t%f0, att1, att2, att3);
        if (att1 = 3)
        {
                $ cancel;
                ifprintf(stdout, "loop cancelled.");
```

```
                    /* must terminate loop after cancel */
                    break;
                }
        }
```

The following synopsis shows those places in the above sentence types **where** C variables or expressions may be embedded within the *IDL* statements. **$C** indicates that a **$**–prepended C variable name or parenthesized C expression may appear at the indicated location in the statement. All C variables must have been correctly declared in a statement with a prepended **$**.

For the syntax of an ***expression***, see the *IDL Language Reference* section "EXPRESSION". For the syntax of a ***qualification***, see the section "QUALIFICATION". Any place a numeric or character-string constant can appear in these, a **$**–prefaced C variable name or expression of the appropriate type may also appear.

$ append [**to**] $C (attribute = expression ,...) [**where** qualification] ;

Thus the following program fragment precompiles:

```
$ append $animal (name = $name, type = $type);
```

$ associate $C [**with** ...] ;

Unfortunately, C strings cannot be used for the ***associate*** comment strings.

$ audit into $C ... [**where** qualification] ;

It would be nice if ***audit*** was a loop-controlling command like *retrieve* and could store its results into variables, but currently it can't.

$ create [**database**] $C ... ;

$ { **create** | **destroy** } ... **index** [**on**] $C ... ;

Thus the following work:

```
$ create index on $rel (type);
$ destroy clustered index on $rel (name);
```

$ create view $C ... [**where** qualification] ;

$ delete range [**where** qualification] ;

$ { **permit** | **deny** } protect-mode **of** relation ($C ,...) ... ;

The name of the attribute to which access is being permitted or denied can be given in a C string. The following are equivalent:

```
$ permit read on animal (name) to edwin;
$ permit read on animal ($("name")) to edwin;
```

This is the only place in these two statements that C expressions can be used.

$ destroy [**database**] $C ,... ;

$ destroy ... [**where** qualification] ;

$ execute queryname [**with**] [**(**] [name =] $C ,... [**)**] ;

$ extend [**database**] $C ... ;

$ open $C ;

$ range of dynamic_range **is** $C [**with** options] ;

$ replace range (attribute = expression ,...) [**where** qualification] ;

$ retrieve [**unique**] ({ [$C =] expression | range.attribute } ,...) [**order** [**by**] expression ...] [**where** qualification] ;

$ retrieve [**unique**] [**into**] $C (attribute = expression ,...) ... [**where** qualification] ;

For a *retrieve* that is not a *retrieve into*, the retrieved values are always placed in C variables. if the name of the C variable is not given explicitly, the value is stored in the C variable that has the same name as the attribute given. Thus the following two statements are exactly equivalent:

```
$ retrieve (a.x, a.y) { ...
$ retrieve ($x = a.x, $y = a.y) { ...
```

If an expression more complicated than a simple *range.attribute* pair is given, then this must be explicitly assigned to a C variable or expression. Thus the following is valid:

```
$ retrieve ($(*i) = int4 (a.string));
```

$ truncate $C ;

Note that the **truncate** command is not documented in the *IDL Language Reference*. It is present in the grammar, however, and does compile and execute correctly when a C string expression is given as its argument:

```
$ truncate $animal ;
```

C expressions can be used as arguments to IDL functions almost anywhere integer or character-string literals may be used. This is not true for arguments that are digit, character, or byte counts; these may only be integer constants.

We give a list of IDL functions with their argument names as given in the *IDL Language Reference Manual*, with a dollar sign ("$") prepended to the names of those arguments which may be C expressions.

abs ($n)
mod ($n, $d)
concat ($a, $b)
substr (pos, len, $str)
substring (pos, len, $str)
int1 ($n)
tinying ($n)
int2 ($n)
smallint ($n)
int4 ($n)
integer ($n)
[fixed] binary ($n)
fbinary ($n)
[fixed] bcd (l, $n)
fbcd (l, $n)
[fixed] bcdflt (l, $n)
fbcdflt (l, $n)
[fixed] bcdfloat (l, $n)
[fixed] string (l, $n)
fstring (l, $n)
fchar (l, $n)
[fixed] char (l, $n)
bcdfixed (prec, frac, $n)
float4 ($n)
flt4 ($n)

smallfloat ($n)
float8 ($n)
flt8 ($n)

EXAMPLES

ric prog.ric

Generate a file named prog.c, and do not use stored commands. It is the program's responsibility at runtime to determine which database it uses by assigning the appropiate value to the (char *) variable *RcCDB* before the execution of the INITRIC code.

ric –d hostdb file.ric

Generate a file named ***file.c***. The code will not use stored commands, but the program will use the database ***hostdb*** by default. The program can override the default by assigning a value to *RcCDB* before INITRIC is executed.

ric –d hostdb –n stprog phyle.ric

The file ***phyle.c*** will execute stored programs in the database when possible. The stored programs will be stored under the name ***stprog***.

BUGS

Does not allow substitution of attribute names like *IIDEL* did.

User-level substitutions not supported yet.

SEE ALSO

initrc(3i)

RIC User's Guide, BLI part number 205-1393-rev.

IDL Reference Manual, BLI part number 205-1235-rev.

NAME

rsc – precompiler for embedding SQL in C

SYNOPSIS

rsc [**–d** database [**–B** device] [**–n** progname]] [**–S** symtabsize] [**–l**] [**–q**] [**–V**] [file.rsc ...]

ARGUMENTS

–d *database* Database to use.

–B *device* Use *device* as the connection to the database server. If not specified, the IDMDEV parameter is consulted. Only used if *database* is specified.

–n *progname* Use stored programs. Associate stored programs under *progname*. *Database* must be specified.

–S *symtabsize* Make the symbol table *symtabsize* elements large. The default size is 100 symbols.

–l, –q Normally, the *#line* directives that *rsc* writes look like this: *#line 3 "file.rsc"*. If –l is specified, they will look like *# 3 "file.rsc"*; if *–q*, like *#line 3 file.rsc*; if both, like *# 3 file.rsc*. See the *RSC User's Guide*.

–V Prints the version number of the precompiler and the version of IDMLIB it uses on *stderr*.

file.rsc ... The file(s) to be precompiled. An input file name must either have a suffix of *.rsc* or else have no suffix. A file name with no suffix is taken literally; that is, if a directory contains the files *p* and *p.rsc* and the command *rsc p* is given, then it is *p* and not *p.rsc* that is precompiled. If an input file name of – (a single minus) is given, then *stdin* is read and *stdout* is written. This allows *rsc* to be used in pipelines.

DESCRIPTION

The precompiler *rsc* takes file(s) with *SQL* commands embedded in C code and generates file(s) containing pure C. The embedded *SQL* statements have a dollar-sign ("$") prefix and are terminated by either a semi-colon (";") or an open-curly bracket ("{"). The *SQL* commmands are translated to appropriate calls into the Britton Lee library IDMLIB. After precompilation, there will be a file generated with the same name as the **.rsc** source file, but with a **.c** suffix. This file is ready for compilation by the C compiler.

Input files must either have a suffix of *.rsc* or else have no suffix. If there are two input files named *x.rsc* and *y*, then the two output files produced will be named *x.c* and *y.c*.

The precompiled query-language commands either may be kept within the object module (the default case) or stored in the database (if the **–n** flag is given). Storing commands in the database is much more efficient at execution time, but requires that the database schema not change during the program's lifetime. See the *RSC User's Guide* for a more complete discussion.

Programs that are precompiled with *rsc* must link in the runtime library IDMLIB. See the examples below.

Example of precompiler source code:

```
main()
{
        $int    num;

        INITRSC("dummy");
        $select $num=number from arelation where name = "animal"
        {
```

```
                printf("%d\n", num);
        }
        exit(0);
}
```

To precompile:

```
rsc -d mydb -n Xprog prog.rsc
```

To compile:

```
cc -o prog prog.c -lidmlib
```

LANGUAGE SYNOPSIS

The following is a quick synopsis of those SQL statements that are acceptable to **rsc** when they contain no embedded C expressions.

$ alter db_name [**with** options] ;

$ audit [**into** table_name] target_list [**from** object_name ,...] [**where** qualification] ;

$ comment on object_name [. column_name] [**is** string_1 [, string_2]] ;

$ create database dbname [**with** option_list] ;

$ create [**unique**] [**clustered** | **nonclustered**] **index on** object_name (column_name ,...) [**with** option_list] ;

$ create table table_name (name type ,...) [**with** option_list] ;

$ create view view_name [(col_name ,...)] **as** select_statement ;

$ delete from object_name [label] [**where** qualification] ;

$ drop object_name ,... ;

$ drop database dbname ,... ;

$ drop [**unique**] [**clustered** | **nonclustered**] **index** [**on**] object_name (column_name ,...) ;

$ grant protect_mode [**on** object_name [(col_name ,...)]] [**to** user ,...] ;

$ insert into object_name [(column_name ,...)] { **values** (expression ,...) | select_statement } ;

$ open dbname ;

$ reconfigure ;

$ revoke protect_mode [**on** object_name [(column_name ,...)]] [**from** user ,...] ;

$ select [**distinct**] [**into** table_name] target ,... [**from** object_name ,...] [**where** qualification] [**group by** column_name [**having** qualification]] [**order by** order_spec ,...] { { | ; }

$ set option [**on** | **off**] ;

$ start { name | **program** number } [[name =] constant ,...] { { | ; }

The sole purpose of following a **start** with a bracketed series of statements is to associate it with one or more **obtain** commands (see below).

$ store [**program**] object_name command [command , ...] **end store** ;

$ sync ;

$ truncate table_name ,... ;

$ update object_name [label] [**from** from_name ,...] **set** col_name = expression ,... [**where** qualification] ;

The *SQL* statements **commit work** and **rollback work** are not accepted by *rsc*. Instead, the three transaction control statements from the IDL language are used. These are

$ begin transaction ;

$ end transaction ;

$ abort transaction ;

See the *RSC User's Guide* .

Two statements have been added that do not exist in interactive SQL.

An **$ obtain** statement has been added to allow the assignment of items selected by select statements that are part of stored commands to be assigned to C variables. Its syntax is

$ obtain [( ] $C ,... [ )] { { | ; }

$C is defined below.

For instance, if a stored command named **foo** contained a **select** statement that returned three items, then we might invoke it via the statements

```
$start foo
{
        $obtain $a, $b, $c
        {
                printf("From foo -- %d %d %d.\n", a, b, c);
        }
}
```

Obtain is a loop-controlling command, like **select**. If an **obtain** is simply followed by a semicolon (";") rather than a bracketed sequence of statements, it still cycles through all the tuples returned by the **select** statement, assigning them to the targets in turn, rather than just returning one tuple like the singleton **select**. Thus a statement like

```
$ obtain $(*p++) ;
```

can be used to fill an array.

The new statement

$ cancel ;

cancels all activity in the shared database system on the current dbin and any dbins that are related to the current one as parent or child in a chain of reopens. Programs using **$cancel** must be careful to exit any **retrieve** loop with **break** immediately:

```
$ range of t is threeatt;
$ retrieve ($att1=t.att1, $att2=t.att2, $att3=t.att3)
{
        ifprintf(stdout, "\t%d\t%s\t%f0, att1, att2, att3);
        if (att1 = 3)
        {
                $ cancel;
                ifprintf(stdout, "loop cancelled.");
```

```
                    /* must terminate loop after cancel */
                    break;
                }
        }
```

The following synopsis shows those places where C variables or expressions may be embedded within SQL statements. C variables or expressions may appear as syntactic elements of *insert* and *select* statements. They may also appear in other statements as part of a *qualification* or an *expression*.

For the syntax of an *expression*, see the *SQL Language Reference Manual*, section "EXPRESSION". For the syntax of a *qualification*, see the section "QUALIFICATION". Any place a numeric or character-string constant may appear in these, a $–prefaced C variable name or expression of the appropriate type may also appear.

The syntax of a *nested_select* is approximately that of the **select** statement. See the *SQL Language Reference Manual*, "SUBQUERIES" and "CORRELATED SUBQUERIES", for exact details. C variables and expressions may be used in *qualifications* in *nested_selects* just as they can be in the **select** statement itself, including clauses controlled by *all*, *any*, or *in*.

Rsc does not disallow assignments to C variables within a *nested_select* target list, but these attempted assignments have no effect at execution time. For example, the statement

```
create view vyu as select $t = type from animal;
```

compiles and executes and creates the view, but the C variable *t* remains unchanged by the execution of the statement. The effect is exactly as if the phrase "$*t*=" had been left out of the statement. This is neither a bug nor a feature, simply a curiosity.

Synopsis of where C variables or expressions, and *qualifications* and *expressions* containing them, may appear in embedded *SQL* statements:

$ audit ... **where** qualification ;

It would be nice if audit were a loop-controlling command like *select* and could store its results into C variables, but currently it cannot.

$ create view ... **as** nested_select ;

$ delete ... **where** qualification ;

$ insert into $ obectname ... { **values** (expression ,...) | nested_select } ;

$ select from $ objectname ... [$C-variable =] name ,... [**where** qualification] [**group by** ... [**having** qualification]] { { | ; }

If only *name* appears, this is equivalent to saying **$name = name**.

$ start queryname [name =] $C ,... ;

$ update ... **set** column = expression ,... **where** qualification ;

C expressions can be used as arguments to SQL functions in most places that an integer or character-string literal may be used. The one exception is those arguments that are a count of digits, characters, or bytes; i.e., those identified by the words *precision*, *pos(ition)*, or *len(gth)* in the section "FUNCTION DESCRIPTIONS" in the *SQL Language Reference Manual*. The following is a list of the SQL functions that take arguments, with the names of the arguments as given in the *SQL Language Reference Manual*. The arguments prepended with a dollar sign ("$") may be C expressions.

```
table_id ($name)
```

table_name ($id)
abs ($num)
binary ($arg)
[fixed] bcd (precision, $expression)
[fixed] bcdflt (precision, $expression)
[fixed] char (len, $expression)
mod ($expr1, $expr2)
concat ($str1, $str2)
col_name ($table_id, $col_id)
substring (pos, len, $str)
char (l, $n)
bcdfixed (position, fraction, $expr)
integer ($n)
smallint ($n)
tinyint ($n)
float ($n)
smallfloat ($n)

EXAMPLES

rsc prog.rsc

Generate a file named *prog.c*, and do not use stored commands. It is the program's responsibility at runtime to determine which database it uses by assigning the appropriate value to the (char *) variable *RcCDB*, before the execution of *INITRSC*.

rsc –d hostdb file.rsc

Generate a file named *file.c*. The code will not use stored commands, but the program will reference the database *hostdb* by default. The program can override the default by assigning a value to *RcCDB* before *INITRSC* is executed.

rsc –d hostdb –n stprog phyle.rsc

The file *phyle.c* will execute stored programs in the database when possible. The stored programs will be stored under the name *stprog*.

BUGS

Does not allow substitution of attribute names like *IIDEL* did.

User level substitutions not supported yet.

SEE ALSO

initrc(3i)

The RSC User's Guide, BLI part number 205-1575-rev.

Portable Host Interface Software Specification, BLI part number 205-1190-rev.

SQL Reference Manual, BLI part number 205-1344-rev.

NAME

sql – Interactive/SQL parser

SYNOPSIS

sql [**–B** device] [**–P**] [**–f** infile] [**–c**] [**–e**] [**–l** linesperpage] [**–n**] [**–p**] [**–s**] [**–x** contchar] [dbname]

ARGUMENTS

–B *device* Use *device* as the connection to the database server.

–P Turn on performance monitoring. Individual performance options can be set using the **set** SQL command.

–c Turn off auto-commenting (auto-association). See *Auto Commenting* below and the *%comment* pseudo command.

–e Echo every command as read. This can be useful when redirecting the input of the parser. In this case, the input commands as well as the replies will go into the output file.

–f *infile* Input file name. If not specified, read the standard input in interactive mode.

–l *linesperpage*
For rudimentary output formatting. *Linesperpage* specifies the number of lines displayed before re-displaying the header. When data is being retrieved, a new header will be printed sufficiently frequently to insure that column labels are always visible. If *linesperpage* is zero, only the initial header will be printed. If not specifed, the terminal driver (*IftTerm*(4I)) is queried.

–n Parse commands, but don't execute them. The connection to the database server will not be opened. Front-end commands (e.g., "*%input*") will still be executed. This can be used to verify an input script that is to be run later.

–p Disable the reading of user and system profile (or startup) files.

–s Run the parser in silent mode. Turns off prompting, printing of SQL banner, and elaborate printing of syntax errors.

–x *contchar* Set *contchar* to be the continuation character. See *Continuation Characters* below, and see the *%continuation* pseudo command.

dbname The name of the initial database to open.

DESCRIPTION

Sql implements the SQL relational query language. Queries typed at a terminal are translated and sent to the shared database system, and results are formatted and printed.

If the **–f** flag is specified, input is read from the named file rather than the standard input. File input is non-interactive. Special functions of interest only to the interactive user are disabled and input is faster.

Continuation Characters

There are several forms of input recognized by Britton Lee Interactive/SQL. The user may choose the input format that is most familiar or comfortable.

The default is similar to that of DB2. Input is buffered, and not executed until a semicolon (";") is entered. This guarantees that incomplete input lines will not destroy data.

The user may also set the line-continuation character to any non-alphanumeric character. In this case, any line not terminated with the continuation character is executed immediately. This convention is similar to that of SQL/DS.

Britton Lee strongly discourages the use of line continuation characters. Inadvertently typing a carriage return before a command is complete may destroy data. Britton Lee recommends that customers use the semicolon to terminate commands, and leave the continuation character at its default value.

For more information, see the **–x** argument above and the *%continuation* pseudo-command, below.

Auto Commenting

Auto comment of stored commands or table and view creation will place the user text into the *descriptions* relation of the current database using the **comment** command. Text starting at the end of the previous command up to and including the **end store** or command termination is stored in the *text* field, including comments and newlines, as it appears in the input. The *key* field of the relation has a value of **s**X where X ranges from 0 to 9 and a to Z to insure the sorting order of the text in the *descriptions* relation. See also the discussions of *%comment*, below, and the **–c** flag, above. See BUGS section for warning about submitting many create statments to the parser at once.

User Profiles

If the **–p** flag is not specifed, SQL reads system and user profile files before user input begins. These files may contain any valid SQL commands. Particularly useful may be the pseudo-commands, which cause the profile file to configure SQL according to the user's individual preference. On UNIX, the profile files may be "/usr/lib/idm/sqlpro.sql" (for a system-wide profile) or "~/.sqlpro.sql" (for a user's individual profile).

User Interrupts

The interrupt character (normally delete (a.k.a. rubout) or control-C on UNIX) can be used to interrupt processing at any time.

Control Character Mapping

The system parameter **MAPCC** may be used to pass control characters through the SQL front end. The default is to map control characters to blanks. See *params*(5I).

Special SQL commands

- The special keyword "ignore" may be used anywhere in a command to cancel the entire current command and reset the line number to 1 (one).
- The "exit" command immediately exits SQL.

Pseudo-Commands

There are a number of commands do not process data but instead control the actions of the parser itself. These are all introduced with a percent sign at the beginning of a line and take effect immediately (i.e., the line cannot be extended by the line-continuation character). Pseudo-commands may be abbreviated to any length.

%comment [*on* | *off*]	If no argument is present or the argument is **on**, then auto-commenting is enabled. Text description of stored commands are automatically entered into the database (using the **comment** command of SQL). If the argument is **off** then auto-commenting is disabled. Auto-commenting is normally on. See also the **–c** flag.
%continuation [*char*]	By default, all Britton Lee Interactive/SQL commands must be terminated with a semicolon (";"). The %continuation pseudo-command allows users to set the line-continuation character to any non-alphanumeric *char*.

If the continuation character is set, input lines ending in a carriage return without the continuation character are executed immediately. Commands may extend over more than one line, if each line is terminated with the specified continuation character.

Use of the continuation character is strongly discouraged. If *char* is omitted, the input style reverts to the default.

%display *text*	Output *text* to the standard output. This is normally used in system profile files to provide informational messages to users.
%edit [*filename*]	Edit the transcript of the SQL session (or *filename* if given). When the editor returns, the file is submitted as input to SQL. The editor used is defined by the EDITOR parameter.
%experience *level*	Set the user's experience level to *level*. *Level* can be "Beginner", "Able", or "Expert".
%input [*filename*]	Read the specified *filename* for SQL commands. When the file ends (or an "exit" command is encountered) control returns to the standard input. If *filename* is not specified, the standard input is read.
%help	Print all immediate commands.
%redo	Resubmit the transcript of the SQL session as input to SQL.
%substitute *name value*	Assign the *name* to have the specified *value*. The "*%name*" syntax (within SQL commands) can be used to interpolate the value. This is a substitution, *not* a macro, so there are restrictions on where this substitution can occur. See *sqlparse*(3I) for details. The value is typed as an iINT2 if the *name* begins with a digit, otherwise the value is typed as an iSTRING (iCHAR).
%trace *tracespec*	Send the *tracespec* to the *tfset*(3I) routine. This turns on host software tracing, and should not be used in normal operation.
%?	Equivalent to "*%help*"

In addition to these commands, two special characters are recognized in the first position of a line. "?" invokes a help subsystem. It may be followed by a help topic, so "? sql insert" describes the **insert** command. A line beginning with the "!" character passes the remainder of the line to the operating system.

EXAMPLE

sql –B /dev/gpib hostdb

Invokes SQL on the GPIB interface, opening the database *hostdb*.

BUGS

If more than one **create** and/or **store** command is submitted to the parser at once, they are all *auto-associated* under the table id of the first object.

There should be some way of controlling the format of the output. A "*%format*" command will probably be added to do this.

The output format should be better adapted to the terminal. For example, output lines that exceed the terminal width are not wrapped nicely. In particular, the current interface does not adapt nicely to IBM 3270-style interactions.

It should be possible to write scripts at this level that include looping based on return data so that simple applications can be prototyped easily.

In general, there should be a very sexy applications development tool available that would include report capabilities, simple applications generators, etc.

NAME

IDM Support Library (IDMLIB) summary; INITIDMLIB

SYNOPSIS

#include <idmlib.h>

INITIDMLIB(progname);

cc –i .... –lidmlib

DESCRIPTION

The IDM Support Library (IDMLIB) contains a set of routines that may be ported to a number of different host machines and operating systems. Some of these routines are machine-independent, but others are highly machine-dependent and have to be modified or completely re-written to port to a new environment.

Given only an acceptable C compiler and an IDMLIB, our "generic code" should be able to run happily in a large number of environments.

GENERAL INFORMATION

In order to use the capabilities of IDMLIB, all source files must include the file *idmlib.h*.

The main program must use *INITIDMLIB*(*progname*) as the first IDMLIB operation. This will initialize IDMLIB and set the name of this program for use by error messages, etc. This *must* be called from *main*() to insure machine-independence.

In addition, the IDM support library (–lidmlib on UNIX) must be loaded. Sixteen-bit machines require the use of separated instruction and data space (the **–i** flag on UNIX).

Warning: Variable names and structure field names which start with an underscore ('_') are non-public unless otherwise documented. Usage of these hidden values may result in unexpected errors.

In general, any arguments passed into IDMLIB that are saved internally are copied. Thus, the space used by the argument may be reused immediately.

TYPES

The following basic types are defined either by the C language or by IDMLIB:

int	An integer in the basic size of the language. It is not fair game to assume that a pointer fits into an int. Also, there is no guarantee that a long is always the same size as an int, even though this is true on VAXes.
char	A character in the native character set.
BYTE	An eight-bit byte.
short	A two-byte integer.
long	A four-byte integer.
BOOL	A Boolean (TRUE/FALSE) value.
BCDNO	A BCD number. Direct access to fields in a BCDNO should not be attempted. This format is specific to the RDBMS system software and does not correspond to the "packed decimal" format of most hosts.
ANYTYPE	A union consisting of a large number of types. The symbols in parentheses are the associated IDM symbol. The types are: i1type – a one-byte integer (iINT1) i2type – a two-byte integer (iINT2) i4type – a four-byte integer (iINT4) inttype – an integer in the native size f4type – a four-byte float (iFLT4)

f8type – an eight-byte float (iFLT8)
chartype – a one-byte character
cvtype – a vector of characters (iCHAR)
cptype – a pointer to characters (iCHAR)
cpptype – a pointer to a pointer to characters
bvtype – a vector of BYTEs (iBIN)
bptype – a pointer to BYTEs (iBIN)
iptype – a pointer to integers
booltype – a BOOL
bcdtype – a BCDNO (iBCD, iBCDFLT)
anyptype – a pointer to another ANYTYPE

IFILE An IDMLIB file descriptor.

FUNCP A pointer to a function.

RETCODE Return status code.

"Quote bits" (the 0200 bit) cannot be safely used in characters since EBCDIC and other character representations are eight-bit codes.

The >> and << (shift) operators should be kept under close control. Although << is guaranteed to shift zero bits in to the right, >> is not guaranteed to sign-extend.

The defined constant STATIC is the null string if debugging is turned on (i.e., DEBUG is defined); "static" otherwise.

The defined constant READONLY is defined to be the null string on most compilers. If your compiler supports read-only shared data, this will be the appropriate keyword to get these semantics (usually "readonly"). The usage is as a storage class, e.g.,

```
READONLY struct foo[] = ...
```

The macro INITZERO can be used on global declarations to cause zero-initialization. For example:

```
int  LineNumber INITZERO;
```

This is necessary because some compilers (e.g., Whitesmith's) require all declarations to be initialized. Other compilers (e.g., the UNIX PCC) are less efficient if unnecessary initialization is used, especially for large arrays. INITZERO is either the null string or "= 0" as appropriate for the particular compiler.

The macro "_ _" (two underscores) acts as a cast to a pointer to BYTE; this is used in initializations and for routines that take an arbitrary type.

The following constants are used to identify data types when necessary. The type name, length, and corresponding C type are shown:

Type Name	*Length*	*C Type Fetched*
iINT1	1	char
iINT2	2	short
iINT4	4	long
iFLT4	4	float
iFLT8	8	double
iCHAR	variable	char *
iFCHAR	variable	char *
iSTRING	variable	char *
iBCD	variable	BCDNO
iBCDFLT	variable	BCDNO

iBINARY	variable	BYTE *
iFBINARY	variable	BYTE *

For the STRING type the string is terminated by a null byte ("\0") so the bind length must be one byte larger than the maximum anticipated return string or truncation may result.

I/O INTERFACE (LEVEL ONE)

IDMLIB includes a buffered I/O interface. The I/O interface gives a uniform view of the file capabilities available.

A file with associated buffering is called a *ifp* (IDMLIB file pointer), and is declared to be a pointer to a defined type **IFILE**. *Ifopen* creates certain descriptive data for a file and returns a pointer to designate the file in all further transactions.

A constant "pointer" **IFNULL** designates no *ifp* at all.

An integer constant **EOF** is returned upon end-of-file by integer functions that deal with files. This may also be returned for certain error indications.

There are four normally open files with constant pointers declared in the include file and associated with the standard open files:

stdin	standard input file
stdout	standard output file
stderr	standard error file
stdtrc	standard trace file

Standard output, standard error, and standard trace are normally line-buffered, so that no actual output will occur until a newline is output. Standard trace normally refers to the same file as *stdout*, so closing either will close the other implicitly.

By default, files are presented to the application as a *stream*, that is, as a continuous stream of bytes with no inherent delimiters except the beginning and end of the file. An application may also request a record-based presentation, which limits access to the record-at-a-time primitives.

All operations (in particular, *IFGETC*, *IFPUTC*, *ifungetc*, *ifread*, and *ifwrite*) are available on stream-based presentations. *Ifread* and *ifwrite* are logically equivalent to sequences of *IFGETC*s and *IFPUTC*s respectively, although the actual implementation allows performance improvements.

Record-based presentation may be requested by the application by specifying the **rbp** parameter in an *ifopen* or *ifcontrol* call. The only I/O operations available to a file with record-based presentation are *ifread* and *ifwrite* to read and write one record respectively. All other operations are undefined and must not be used. Record-based presentations are best suited to files containing fixed-format data, or where record boundaries may be confused with byte values of 012 (the C "newline" character).

High-level operations (e.g., *ifgets*, *ifprintf*, *igetdone*) are built on top of *ifgetc*/*ifputc* and hence are limited to streams. By default all files are presented as streams.

Files also have an inherent physical structure that cannot be changed. This structure depends on the operating system, the media, and the file format. For example, labelled tape is always physically record-based. On UNIX, all disk files are stream-based from the point of view of a user process. VMS has both record- and stream-based files.

IDMLIB permits a record-based file to be presented as a stream and vice versa, with a few obvious constraints. Stream-based presentation of a record-based file removes record boundaries on input, placing newline characters in their place for text-type files, or ignoring record boundaries for binary-type files. On output, records are assumed to end at newline characters for text files. For binary files, one call to *ifwrite*() results in one record. Record-based presentation of stream-based files only accomodates fixed-length records, that is, one fixed-length record is presented to

the application at a time on input, and on output the application must give a complete record to *ifwrite*.

Not all differences will be hidden. For example, the format of file names will not be standardized across operating systems. However, *makefname*(3I) will build default file names as needed.

The constants (e.g., EOF) and many of the "functions" (such as *IFPUTC*) are implemented as macros.

IDM INTERFACE (LEVEL TWO)

The level-two IDM interface routines operate on data structures specific to the RDBMS system software. A family of routines create and manipulate trees that represent queries. Another family of routines interacts with the database server itself.

Trees can be created using *idlparse*(3I), *itxcmd*(3I), *itcopy*(3I), or one of the DBA routines described in *dba*(3I). *Idlparse* parses an IDL string. *Itxcmd* creates a tree for an **execute** command. The others create trees to perform DBA commands.

Once a tree is created, *iputtree*(3I) sends it to the database server. *Igettl*(3I) reads the target list for commands that return data. Retrieved tuples can be read back by successive calls to *igettup*(3I). The DONE token is read by *igetdone*(3I), as well as ERROR or other trailing data.

Iputtl(3I) and *iputtup*(3I) send target-list descriptions and tuple data respectively to a file, normally used by routines doing bulk copies.

When a tree is no longer needed, it must be explicitly deallocated using *itfree*(3I). Similarly, when a target list is no longer needed, it must be explicitly deallocated using *itlfree* (see *igettl*(3I)).

Many of these routines use an *environment* that maintains miscellaneous control and state information. In most cases, passing an environment of IENVNULL will default to a system global environment. Environments are stacked. New environments are created using *ieopen*(3I) and destroyed using *ieclose*(3I).

The level-two routines are suited to system program interfaces. Application programs will typically find the level-three interface more convenient.

IDMRUN INTERFACE (LEVEL THREE)

The IDMRUN subsystem provides a high-level programming-language interface to the shared database system. Most of the details of data structures and operations are hidden. This interface is appropriate for the application programmer. All modules using the IDMRUN interface must include the file <*idmrun.h*> after <*idmlib.h*>:

```
#include <idmlib.h>
#include <idmrun.h>
```

Any modules that include level-two include files (e.g., <*idmtree.h*>) must include these before <*idmrun.h*>.

All character strings passed into these routines that do not refer to data (e.g., object or parameter names) have all uppercase letters folded to lowercase if **foldcase** mode is set in the underlying environment; see *irget*(3I) and *iecontrol*(3I) for details. The default setting of this mode depends on the local system conventions: on for VMS, off for UNIX, etc.

The IDMRUN Structure

The IDMRUN structure contains all of the state information necessary for the run-time system to determine the legality of operations. An IDMRUN structure has the following characteristics:

- An IDMRUN structure is associated with a single database server.
- IDMRUN structures can only have one parsed IDL statement list associated with them at a time. Every time new statements are created, the previously parsed statements are discarded and the newly parsed statements take their place.

- The IDMRUN structure contains the return status from the shared database system. After a command is executed, all of the return data must be processed. An attempt to create another set of statements before this is done will result in an error.

Iropen(3I) returns an IDMRUN structure. If the database name is known in advance, it can be specified in this command. If not, a database can be opened later.

Upon completion of the use of the IDMRUN structure, it should be closed via *irclose*.

All use of the IDMRUN structure must follow in the strict order:

(1) Select a statement to process using *iridl*(3I), *irsql*(3I), *irxcmd*(3I), or *irxprog*, or by setting a command tree using *irset*(3I) with the IP_TREE option.

(2) Send the tree to the IDM/RDBMS software using *irexec*(3I).

(3) Retrieve results (if appropriate) using *irfetch*(3I). Results can be described using *irdesc*(3I) and bound to programming-language variables using *irbind*(3I).

(4) Proceed to the next set of commands and/or results using *irnext*(3I). This step is necessary since several trees can be in an IDMRUN structure at the same time — even a tree created with *irxcmd* can reference a stored command that contains several primitive commands, so that multiple batches of data can be returned. The program should then cycle back to step three.

(1) Selecting a Command

IDL statements in text form are parsed and associated with an IDMRUN structure using *iridl*(3I) or *irsql*(3I). This is the usual way of inputting statements. Special trees to execute stored commands (or programs) can be created quickly using *irxcmd*(3I) or *irxprog* (documented in the same section). Parameters are added incrementally using *irxsetp* (also documented in *irxcmd*(3I)).

(2) Starting Execution

Successfully parsed statements can be executed using *irexec*(3I). This sends the first command in a list to the IDM/RDBMS software. *Irexec* "peeks ahead" at the results coming back from the IDM/RDBMS software — if data is returned it determines the type of the fields; if no data is returned it reads and processes the status information.

(3) Reading Results

If the statement returns tuple values the program can bind program variables to receive the retrieved target-list elements with *irbind*(3I). Descriptions of the types of the retrieved target-list elements can be requested to aid in the binding process by calls to *irdesc*(3I). *Irbind* causes conversion from any of the numeric IDM types to any of the numeric types and from any of the IDM types to STRING. No other automatic conversions are guaranteed.

Each call to *irfetch*(3I) reads the next tuple into the bound programming-language variables. Any target-list element values which have not been bound to programming variables are discarded. Automatic type conversion from the type of the target-list element as stored on the database server to the type of the bound programming variable is performed. If the program decides not to process all of the retrieved tuples (e.g., by exiting a retrieve loop early) then *irflush*(3I) can be called to remove the remaining tuples by reading them all and throwing the results away. *Ircancel*(3I) will flush the remaining tuples without reading them but it will also cancel any pending commands to be executed on the database server as well as any return information. *Ircancel* can also be called to stop the current executing command on the database server (this is useful when responding to interrupts). In general, *irflush* should be used when responding to normal conditions where the data is no longer of interest, while *ircancel* should be used to do a full abort.

(4) The Next Command

When a command stream contains more than one executable statement, due either to parsing several statements in a single call to *iridl* or *irsql* or to executing a stored command containing

several commands, the routine *irnext*(3I) must be called for each primitive command. *Irnext* moves on to the next command in the stream and otherwise acts like *irexec*.

All data must be processed before processing can continue. *Irflush* will discard all of the return information for the next statement. However, if there was more than one executable IDL statement parsed (or if there was an IDL *irexec* of a stored command which contained more than one executable statement), an *irflush* must be performed for each command which returns data (i.e., after every call to *irnext*). Alternatively a single call to *ircancel* will clear all of the return information.

Status

All error tokens and done packets are handled automatically by the run-time system. Done packet information is known immediately upon executing commands which do not return data, i.e., *irexec* will read in the done packet information for the first command unless there is tuple data to be processed. For commands returning data the done packet information is returned automatically when all of the data has been read or a flush has been performed. *Irnext* reads in the done packet for a statement if there is no data to be returned.

Return Values

Some routines return only a status code (typed as RETCODE). The status code can be RS_NORM to indicate successful, normal completion, RE_FAILURE to indicate failure, or a warning status. If the status is RE_FAILURE, an exception will be raised giving more detail. Warning returns will typically not have an associated exception raised. See *retcode*(5I) for details.

Types

In addition to the types supported in level two, the type **iDSC** may be used at this level. This type is intended to be a "descriptor-based" type defined by the host architecture and operating system. Most commonly this will be used for scaled types, decimal types, etc.

Type **iDSC** may be passed to *irsubst*(3I) and *irbind*(3I).

The routines used to manipulate descriptors are described in *dsc*(3I).

Programs using descriptors are inherently non-portable.

EXCEPTIONS

The exception package helps formalize the handling of special conditions that require abnormal flow of control. When a procedure "raises an exception" a search is made backward through the invocation stack until a "handler" is found for this exception. The handler is then called; it can perform any necessary cleanup operations and can then ignore the exception, back out (i.e., abort the procedure that raised the exception), or re-raise the exception to start the process over again. See *exc*(3I) for details.

Routines that raise exceptions list the exceptions and the semantics of the parameters to the exception.

The following exceptions can occur from a number of places:

W:IDMLIB.ARITH.OVERFLOW
: Arithmetic overflow occurred.

W:IDMLIB.ARITH.UNDERFLOW
: Arithmetic underflow occurred.

W:IDMLIB.ARITH.DIVZERO
: Division by zero occurred.

W:IDMLIB.ARITH.PRECISION
: Precision was lost during a conversion operation.

T:IDMLIB.ASYNC.INT
A terminal interrupt occurred.

T:IDMLIB.ASYNC.TERM
A terminate signal occurred.

A:IDMLIB.ASYNC.NOFP
The program attempted to use floating-point with no floating-point hardware or software emulation.

A:IDMLIB.IO.BADIFP(detail)
The *ifp* passed to some routine was determined to be bad.

A:IDMLIB.IO.IOERR(filetype, filename, detail)
An I/O error occurred during some operation. This typically indicates some sort of hardware problem.

A:IDMLIB.RECOMPILE
The program is out of date with respect to the library.

This document does not list all IDMLIB exceptions. For a complete list, see the *IDL Message Summary* and *SQL Message Summary* for your host system.

GLOBALS

The following globals are used by IDMLIB for communication with the application:

CnvtCount	Set to the count of the number of characters converted by *atof*, *atoi*, and *atol*. See *atof*(3I).
ProgName	The name of this program; used by routines that print messages.
FileName	This may be set to the current input file name for printing with error messages.
LineNumber	If this variable is greater than zero, it will be printed with error messages. *IftScan* will increment the *LineNumber* if requested.
DefEnv	The default environment for use by the IDM routines.
DefMpool	The default memory pool for *xalloc*(3I), et. al.
SysMpool	The global memory pool.

COMPILATION FLAGS

The following flags are defined to handle exceptional cases. Their use should be kept to an absolute minimum.

One of the following flags is set to tell what hardware we are running on:

IBM370	This is an IBM 370-architecture processor (this includes the 43xx and 30xx series).
M68K	This is a 68000-based processor.
MV	This is a Data General MV processor.
PDP	This is a PDP-11 processor.
PYRAMID	This is a RISC processor.
VAX	This is a VAX processor.
U3B2	This is a Western Electric 3B2 series processor.
U3B5	This is a Western Electric 3B5 series processor.
U3B20S	This is a Western Electric 3B20S series processor.

U3B20AP This is a Western Electric 3B20AP series processor.

One of these flags is set to define the operating system being run:

UNIX We are running 4.2 BSD UNIX.

UNIX5 We are running UNIX System V.

VMS We are running DEC VMS.

CMS We are running IBM CMS.

MVS We are running IBM MVS.

MSDOS We are running MS-DOS.

AOS_VS We are running Data General AOS/VS.

The native character set of the machine is defined using one of the following:

ASCII This machine uses the ASCII character set, as defined by ANSI standard X3.4-1977, *American National Standard Code for Information Interchange.*

EBCDIC This machine uses the EBCDIC character set.

The following constants are always defined; their value describes certain parameters of the hardware and the underlying system:

WORDBITS The size of an integer, normally 16 or 32.

ADDRBITS Set to the number of bits of address space available for user programs, normally 16, 17, 24, 31, or 32.

The following constants may be defined to enable special features.

NOFP This machine may not have floating-point hardware.

DEBUG Compile in debugging flags.

WHITESMITHS The Whitesmith's pseudo-compiler is being used.

IMPLEMENTATION NOTES

This spec is both a functional spec for users of IDMLIB and an implementation spec. Implementation notes are broken out into a special section.

Where necessary to rename an IDMLIB routine to avoid conflict with a system routine, "#define"s can be used. Routine names with leading underscores should not be used in regular programs to provide a namespace that can be used freely by the IDMLIB implementor.

Since IDMLIB can be linked with user programs, it will be important for IDMLIB to coexist with the host run-time library. However, it should be possible to link IDMLIB without linking in unused portions of the system run-time library.

NAME

anyprint, anyfmt — print or format any possible type for printing

SYNOPSIS

```
anyprint(type, length, value, ifp)
int type;
int length;
BYTE *value;
IFILE *ifp;

char *anyfmt(type, length, value)
int type;
int length;
BYTE *value;
```

DESCRIPTION

Anyprint prints any *type* and *length* datum pointed to by *value* onto the file *ifp*. Regular data types (e.g., iINT1 etc.) are stored as a direct conversion. Certain IDM/RDBMS nodes (such as iRANGE et al.) are formatted with labels. Other types are converted to a hexadecimal string.

Anyfmt formats the result and returns a pointer to the converted string. The string will be destroyed on the next call.

The format of the following "vanilla" types (basic user data types) are guaranteed suitable for normal user consumption:

iINT1	iINT2	iINT4	
iFLT4	iFLT8	iBINARY	iFBINARY
iCHAR	iFCHAR	iSTRING	iSUBSTITUTE
iBCD	iBCDFLT	iFBCD	iFBCDFLT

The formats of more obscure types are for gurus only.

SEE ALSO

intro(3I), itlprint(3I), itprint(3I), printf(3I), typecnvt(3I)

NAME

ASSERT — verify fixpoints in a program

SYNOPSIS

ASSERT(expression)

DESCRIPTION

ASSERT indicates that ***expression*** is expected to be true at this point in the program. It syserrs with a diagnostic comment when ***expression*** is FALSE.

DIAGNOSTICS

Some message will be given containing sufficient information to find the problem in the source code. It is not intended that a naive user be able to understand the message. For example: "Assertion failed: file *f* line *n*." *F* is the source file and *n* the source line number of the *ASSERT* statement.

SEE ALSO

syserr(3I)

IMPLEMENTATION NOTES

Some C compilers define the pseudo-macros "__FILE__" and "__LINE__" to describe the current file and line number. These should be used if available.

NAME

atobcd — alpha to BCD conversion

SYNOPSIS

```
#include <bcd.h>

BCDNO *atobcd(buf, res)
char *buf;
BCDNO *res;
```

DESCRIPTION

Atobcd converts a character string to BCD and stores the result in *res*.

EXCEPTIONS

W:IDMLIB.BCD.OVERFLOW
An overflow occurred.

W:IDMLIB.BCD.UNDERFLOW
An underflow occurred.

SEE ALSO

intro(3I), bcd(3I), *System Programmer's Manual (SPM)* for BCD representations and semantics.

NAME

atof, atos, atol — convert characters to numbers

SYNOPSIS

```
double atof(nptr)
char *nptr;

atos(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a character string pointed to by *nptr* to double precision floating point, short integer, and long integer representation respectively. The first unrecognized character ends the string.

Atof recognizes an optional string of tabs and spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

Atos and *atol* recognize an optional string of tabs and spaces, then an optional sign, an optional "0x" to force hexadecimal radix interpretation or "0o" to force octal radix interpretation, then a string of digits.

GLOBALS

CnvtCount Set to the number of bytes consumed from *nptr*.

EXCEPTIONS

W:IDMLIB.CNVT.OVERFLOW(nptr, limit)

An arithmetic error occurred during processing.

IMPLEMENTATION NOTES

Environment-independent versions of *atos* and *atol* exist. An environment-dependent version of *atof* must be supplied by the OEM.

Atoi is not defined in IDMLIB. If the system C runtime library does not define an *atoi* then the machine dependent header file *machdep.h* can add the appropriate #define line depending on the system integer size.

```
#define  atoi(p)  (int) atos(p)
#define  atoi(p)  (int) atol(p)
```

BUGS

Atof neither sets CnvtCount nor detects overflow, and is unlikely to in the near future.

Atos fails on the value −32768; *atol* fails on −2147483648.

These routines do not understand unsigned numbers; they will cause overflow exceptions.

NAME

bcdadd, bcdsub, bcddiv, bcdmult, bcdcmp, bcdround — BCD arithmetic

SYNOPSIS

```
#include <bcd.h>

BCDNO *bcdadd(srca, srcb, res)
BCDNO *srca;
BCDNO *srcb;
BCDNO *res;

BCDNO *bcdsub(srca, srcb, res)
BCDNO *srca;
BCDNO *srcb;
BCDNO *res;

BCDNO *bcdmult(srca, srcb, res)
BCDNO *srca;
BCDNO *srcb;
BCDNO *res;

BCDNO *bcddiv(srca, srcb, res, domod)
BCDNO *srca;
BCDNO *srcb;
BCDNO *res;
BOOL domod;

bcdcmp(srca, srcb)
BCDNO *srca;
BCDNO *srcb;

bcdround(bcdnum, prec)
BCDNO *bcdnum;
int prec;
```

WARNING

These routines are not supported at this time.

DESCRIPTION

Bcdadd, bcdsub, bcdmult, and bcddiv each perform an arithmetic operation between the two source operands *srca* and *srcb* and place the result in *res*. The type of the result will be BCDFLT if either of the source operands are BCDFLT and will be BCD otherwise. If *domod* is true when *bcddiv* is called then the modulo operation is performed. Modulo operations are not defined for BCDFLTs.

BCD and BCDFLT comparisons can be done with *bcdcmp*. It returns a negative, zero, or positive number depending on whether the first operand is less than, equal to, or greater than the second operand, respectively.

A BCDFLT number can be rounded to a specified precision or a BCD number can be truncated using *bcdround*. The specified *bcdnum* is left with at most *prec* digits right of the decimal point in the case of BCDFLT, or *prec* digits altogether in the case of a BCD number.

EXCEPTIONS

W:IDMLIB.BCD.OVERFLOW
: An overflow occurred during BCD arithmetic.

W:IDMLIB.BCD.UNDERFLOW
: An underflow occurred during BCD arithmetic.

W:IDMLIB.BCD.DIVZERO
An attempt was made to divide by zero.

W:IDMLIB.BCD.PRECISION
Precision was lost during a conversion operation.

BUGS

Bcdcmp fails on zero value comparisons if one BCD was retrieved from the database and the other created via ***atobcd***. This is due to the many possible representations of a zero BCD.

SEE ALSO

intro(3I), atobcd(3I), bcdtobcdf(3I), bcdtol(3I), ftoa(3I), *SPM* for BCD representations and semantics.

NAME

bcdtoa — BCD to alpha conversion

SYNOPSIS

```
#include <bcd.h>

char *bcdtoa(bcd, buf, width, fmt, scale, prec)
BCDNO *bcd;
char *buf;
int width;
char fmt;
int scale;
int prec;
```

DESCRIPTION

Bcdtoa converts the BCD number *bcd* into a string stored in *buf* of length at most *width*. There will be at most *prec* digits after the decimal point. Six formats are defined by *fmt*. These are:

F Regular floating-point.

E Exponential format.

G E or F format, whichever produces the smallest number of characters.

H E or F as appropriate to fit, with F preferred.

A Like H, but with decimal points aligned on F's. Alignment is done only within E and F formats, that is, E format align with E format, F format with F formats, but E and F format do not align.

P Like F, but with the number padded out to *prec* digits after the decimal point even if they are not present in the input.

If the number is output in E format, *scale* digits will be placed before the decimal point.

SEE ALSO

atobcd(3I), fmtfloat(3I), ftoa(3I), *SPM* for BCD representations and semantics.

NAME

bcdftobcd, bcdtobcdf — BCD conversion

SYNOPSIS

```
#include <bcd.h>

BCDNO *bcdftobcd(bcdnum, res)
BCDNO *bcdnum;
BCDNO *res;

BCDNO *bcdtobcdf(bcdnum, res)
BCDNO *bcdnum;
BCDNO *res;
```

DESCRIPTION

Conversions between BCD and BCDFLT can be performed using ***bcdftobcd*** and ***bcdtobcdf***. The former operation can cause an OVERFLOW exception, but the second is guaranteed to succeed.

EXCEPTIONS

W:IDMLIB.BCD.OVERFLOW
An overflow occurred.

SEE ALSO

intro(3I), bcd(3I), *System Programmer's Manual* for BCD representations and semantics.

NAME

bcdtol, ltobcd — BCD to long integer conversion

SYNOPSIS

```
#include <bcd.h>

long bcdtol(src, res)
BCDNO *src;
long *res;

BCDNO *ltobcd(src, res, restype)
long *src;
BCDNO *res;
BYTE restype;
```

DESCRIPTION

Bcdtol Converts a BCD or BCDFLT number to a long integer.

Ltobcd Converts a long integer to the desired ***restype*** bcd. ***Restype*** may be either iBCD or iBCDFLT.

EXCEPTIONS

W:IDMLIB.BCD.OVERFLOW
An overflow occurred.

W:IDMLIB.BCD.UNDERFLOW
An underflow occurred.

W:IDMLIB.BCD.PRECISION
Precision was lost during conversion.

SEE ALSO

intro(3I), bcd(3I), *System Programmer's Manual* for BCD representations and semantics.

NAME

bcopy, bfill, bzero, STRUCTASGN — copy, set, or zero a block of memory

SYNOPSIS

```
bcopy(from, to, size)
BYTE *from, *to;
int size;

bfill(to, size, ch)
BYTE *to;
int size;
BYTE ch;

bzero(to, size)
BYTE *to;
int size;

STRUCTASGN(dst, src)
struct ??? dst;
struct ??? src;
```

DESCRIPTION

Bcopy copies *size* bytes from *from* to the block of memory at *to*.

Bfill fills *size* bytes of memory at *to* with copies of the given character *ch*. *Bzero* acts like *bfill* except that the character is the zero byte.

STRUCTASGN is a macro that copies the struct *src* to *dst* (note: these are not pointers to the structs, but the structs themselves); *src* and *dst* must be compatible structures. On compilers supporting structure assignment this macro expands to "dst = src"; otherwise it is a *bcopy*.

LIMITATIONS

The *from* and *to* areas in *bcopy* should not overlap in any way to allow most efficient implementation on any machine. Specifically, left-to-right copy is not guaranteed.

Size should never exceed 65535. This also limits the size of the structures in *STRUCTASGN*.

IMPLEMENTATION NOTES

Although environment-independent implementations exist, these may be implemented as an in-line macro instruction using an assembly language massager.

The "size == 0" case must be handled properly.

Bfill and *bzero* are provided as separate commands because zeroing memory is typically less expensive than filling it with an arbitrary byte.

SEE ALSO

string(3I)

NAME

bintoa, atobin — binary to alpha conversion

SYNOPSIS

```
bintoa(inptr, inlen, outptr, outlen)
BYTE *inptr;
int inlen;
char *outptr;
int outlen;

atobin(inptr, inlen, outptr, outlen)
char *inptr;
int inlen;
BYTE *outptr;
int outlen;
```

DESCRIPTION

Bintoa converts a string of bytes of length *inlen* starting at *inptr* to a character string stored into *outptr*. There are *outlen* bytes available at *outptr* for data storage.

Each input byte is converted to two output characters representing the hexadecimal value of that byte. For example, the input byte with value 31 (decimal) is converted to the characters "1F" on output.

A trailing null byte is added.

If *outlen* is not large enough to store all the bytes from the input, input bytes are truncated on the right. Note that binaries represent byte strings rather than integers: leading zeros are significant, while trailing zeros are insignificant.

Atobin performs the inverse operation.

EXCEPTIONS

W:IDMLIB.CNVT.OVERFLOW(input, limits)
: The output overflowed.

W:IDMLIB.CNVT.ATOBIN(char)
: The specified *character* is not a valid hexadecimal character (0–9, a–f, A–F).

SEE ALSO

typecnvt(3I), xdump(3I)

NAME

BITSET — test to see if a bit is set

SYNOPSIS

BOOL BITSET(bits, word)
int bits;
int word;

DESCRIPTION

BITSET returns TRUE if any of the *bits* are set in *word*. For example, typical usage might be:

```
if (BITSET(ID_ERROR, dp→id_stat))
        ...
```

To set one or more *bits*, use

```
word |= bits;
```

To clear one or more bits, use

```
word &= ~bits;
```

BITSET is implemented as a macro.

DISCLAIMER

BITSET actually returns an **int**, not a BOOL (or **char**).

NAME

ISALPHA, ISUPPER, ISLOWER, ISDIGIT, ISXDIGIT, ISALNUM, ISSPACE, ISPUNCT, ISPRINT, ISGRAPH, ISCNTRL, ISCHAR, ISPMATCH, ISZWIDTH, ISKANJI, TOCHAR, TOUPPER, TOLOWER — character classification and conversion

SYNOPSIS

#include <bytetype.h>

ISALPHA(c)

. . .

DESCRIPTION

The *IS*xxx macros classify character-coded integer values by table lookup. Each is a predicate returning TRUE if the indicated condition is satisfied. The *TO*xxx macros do character-specific conversions.

ISCHAR is defined on all integer values; the rest are defined only where *ISCHAR* is true and on the single out-of-band value EOF (see *intro*(3I)).

ISALPHA	*c* is a letter [a-z, A-Z]
ISUPPER	*c* is an uppercase letter [A-Z]
ISLOWER	*c* is a lowercase letter [a-z]
ISDIGIT	*c* is a digit [0-9]
ISXDIGIT	*c* is a hexadecimal digit [0-9, A-F, a-f]
ISALNUM	*c* is an alphanumeric character [a-z, A-Z, 0-9]
ISSPACE	*c* is a space, tab, carriage return, newline, or formfeed
ISPUNCT	*c* is a punctuation character (neither control nor alphanumeric)
ISPRINT	*c* is a printing character, ASCII codes 040 (space) through 0176 (tilde).
ISGRAPH	*c* is a printing character, like *isprint* except false for space
ISCNTRL	*c* is a delete character (ASCII 0177) or ordinary control character (less than ASCII 040).
ISCHAR	*c* is a character in the native character set of the host computer.
ISPMATCH	*c* is an IDM pattern matching character ('*', '?', or '[') or the internal equivalent thereof.
ISZWIDTH	*c* is nominally a zero-width character when printed.
ISKANJI	*c* is one byte of a two-byte Kanji character. This is always FALSE in American and European versions of IDMLIB.
TOCHAR	Converts a character into the legal range by stripping off special bits.
TOUPPER	If the argument is a lowercase letter, returns the uppercase equivalent; undefined on other values.
TOLOWER	If the argument is a uppercase letter, returns the lowercase equivalent; undefined on other values.

SEE ALSO

string(3I)

IMPLEMENTATION NOTES

Although the descriptions of the domain of these routines refer to ASCII characters, the implementation also handles EBCDIC. The EBCDIC codes are derived from the IBM System/360

Reference Card, order number GX20-1703-7.

NAME

crackargv, usage — take apart an argument vector or print a usage message

SYNOPSIS

#include <crackargv.h>

crackargv(argv, template)
char **argv;
ARGLIST *template;

usage(template, fmt, a1, a2, a3)
ARGLIST *template;
char *fmt;

DESCRIPTION

Crackargv parses command-line arguments as necessary for the host environment. Traditionally in C, the command line is passed to the subroutine *main*() as the arguments *argc* and *argv*, without provisions for special command options and differing command syntax with different operating systems. *Crackargv* accepts a NULL-terminated *argv* and a template data structure describing the allowable arguments for the command and where the argument values should be stored. *Crackargv* takes apart the argument vector, storing argument values in the program's variables.

Command arguments are either positional arguments or flag arguments:

Positional arguments have no explicit name in the command invocation; They must be specified in a particular order. Required positional arguments must precede optional positional arguments. The template specifies required and optional positionals for this program. The number of positional arguments that the user specifies in the command invocation must be at least as many as the number of required arguments. Because of limitations on some host operating systems, at most six positional arguments may be specified.

Flag arguments have names, and may be specified in any order. They are almost always optional. A flag having no value associated with it is called a boolean flag. The template lists all flag arguments with their names and the type of their argument.

Given a template and an *argv*, *crackargv* finds the argument values in the most user-friendly manner possible.

On UNIX, flags with values have the form **–x***value* or **–x** *value* as convenient. Boolean flags can be concatenated; for example, "–abc" is the same as "–a –b –c". A flag taking an argument must be the last flag in the sequence; for example, "–abcx 7" is legal (assuming the "–x" flag takes a value) but "–abxc 7" is not. A minus sign '–' preceding the argument of a short, integer, or long must be abutted to the flag, e.g., "–x-7".

The template is an array of structures describing the parameters. The fields are:

flag_cname — The character that names this flag, on operating systems like UNIX that use single-character flag names. If it is FLAGPOS then this entry represents a positional argument. Order is important; positional arguments will be matched in the order listed. In general, all positional argument templates should come after all flag argument templates for readability. The last entry in the list has this argument equal to the null character, '\0'.

flag_type — The type of the value for this argument. These may be

FLAGBOOL	boolean (takes no value)
FLAGSHORT	short integer
FLAGLONG	long integer
FLAGINT	native integer
FLAGCHAR	single character
FLAGSTRING	text string
FLAGLIST	vector of string (last positional only)
FLAGTRACE	trace specification
FLAGPARAM	global IDMLIB parameter
FLAGVER	show IDMLIB version number (takes no value)

Native integers are short or long, depending on the underlying hardware. Lists are sequences of strings. There should be no more than one parameter of type FLAGLIST and it should be the last flag in the description. For example, on VMS this will turn a comma-separated list of elements on the command line into one list. On UNIX it will match the rest of the *argv* argument vector after the other positionals are consumed. FLAGPARAMs are passed to *setparam* with the *flag_value* pointing to a null-terminated parameter name. Trace flags are passed to *tfset* (see *tf*(3I)). FLAGVER is used so that a user can determine the exact version of the library being used.

flag_mlength — Minimum length (flags only). In the string form (for VMS-like systems) this is the minimum number of characters that must be specified on the command line to match this *flag_lname*. Normally this is just enough to make the name unique. For instance, if the *lnames* are "fig", "plum", "process", and "protect", "fig" would have an *mlength* of one, "plum" two, and the last two would both require four. "Dangerous" flags can set the *mlength* equal to their total length; for example, the parameter "initialize" could have an *mlength* of ten to insure that it could not accidently be specified.

flag_lname — The long (string) form of the flag name. This is used on VMS and other systems that use full-word qualifier names. These should be unique in the first four characters.

flag_aname — An alternate string form, for Multics-like systems. This will normally be very short and incomprehensible. If NULL, it will be ignored.

flag_value — A pointer to a place to put the result. It should be a pointer prepended by two underscores ("__") which will do necessary type coercion. If this value is to have a default it should be set before *crackargv* is called. For FLAGSTRINGs, this points to a character pointer that will end up pointing to the string which has been statically allocated by *crackargv*. For FLAGPARAMs, this pointer instead points to a constant character string that specifies the system parameter name that should receive the value.

flag_prompt — Prompt string. If this is not NULL, the argument is required. If the user does not specify it and the operating system supports prompting, this prompt will be printed (followed by a question mark) and the value read from the standard input.

flag_usage — A text string to print in a usage message; the name of this argument. If this is NULL, *flag_prompt* is used. If that is NULL, *flag_lname* is used. If this is the zero-length string ("") then this flag will never be printed in

a usage message; this is used for "hidden" flags, i.e., flags intended for BLI use only.

Crackargv must consume all arguments.

An implementation must accept some default flags; that is, flags that are not listed in a *template* should be availiable in a default list. The following list is the minimum set of default flags that must be implemented:

```
-B  FLAGPARAM   "IDMDEV"
-E  FLAGPARAM   "EXPERIENCE"
-T  FLAGTRACE   —
-V  FLAGVER     —
```

The routine ***usage*** can be used to print a usage message in a machine-independent fashion. It prints the *fmt* and arguments in *printf*(3I) style followed by a usage message built from *template*. *Usage* then raises "U:*name*.USAGE" (where *name* is the program name specified by *INITIDMLIB*) and exits with status RE_USAGE. This message can give more detail about the use of the command.

GLOBALS

ProgName Used by ***usage*** to print the name of this program.

EXAMPLES

In order for the argument template in the following example to fit completely on the page the definition of *_CN* for CHARNULL is included. Note that it is **not** defined by the include files.

```
#include <idmlib.h>
#include <crackargv.h>

#define _CN CHARNULL

short   ShortV;
BOOL    Xact;
int     Count;
char    *DbName;

ARGLISTArgs[ ] =
{
/* cname      type           mlen lname     aname   value          prompt     usage */
   's',       FLAGSHORT,     1,   "short",   "fs",   __ &ShortV,    _CN,       _CN,
   'x',       FLAGBOOL,      4,   "trans",   "tx",   __ &Xact,      _CN,       _CN,
   'r',       FLAGINT,       1,   "rep",     _CN,    __ &Count,     "count",   _CN,
   'B',       FLAGPARAM,     1,   "idmdev",  _CN,    __ "IDMDEV",   _CN,       _CN,
   'T',       FLAGTRACE,     4,   "trace",   _CN,    BYTENULL,      _CN,       "",
   FLAGPOS,   FLAGSTRING,    0,   _CN,       _CN,    __ &DbName,    "dbname",  _CN,
   '\0'
};

main(argc, argv)
        int argc;
        char **argv;
{
        INITIDMLIB("testprog");
        crackargv(argv, Args);
         (etc)
}
```

Legal command-line syntax includes:

```
% testprog hostdb
% testprog -xs5 -B/dev/other -T50.9 bigdb
```

LIMITATIONS

It is not possible to have multiple occurrences of named flags.

IMPLEMENTATION NOTES

An implementation exists to parse UNIX argument vectors. This version should be examined before doing further development.

It may be reasonable to check the experience level to decide whether to prompt for missing required arguments rather than diagnosing an error.

EXCEPTIONS

E:IDMLIB.CRACKARGV.BADINT(str)

An illegal value was specified for an integer.

U:*progname*.USAGE

This program was invoked incorrectly.

SEE ALSO

getparam(3I), printf(3I), tf(3I)

NAME

itdbdump, ittxdump, itdbload, ittxload, itrollf — build trees for database administration functions

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

ITREE *itdbdump(dbname, dbfile, txfile, tape, env)
char *dbname;
char *dbfile;
char *txfile;
char *tape;
IENV *env;

ITREE *ittxdump(dbname, txname, txfile, tape, env)
char *dbname;
char *txname;
char *txfile;
char *tape;
IENV *env;

ITREE *itdbload(dbname, dbfile, tape, env)
char *dbname;
char *dbfile;
char *tape;
IENV *env;

ITREE *ittxload(dbname, txname, txfile, tape, env)
char *dbname;
char *txname;
char *txfile;
char *tape;
IENV *env;

ITREE *itrollf(dbname, txname, datetime, env)
char *dbname;
char *txname;
CLOCK *datetime;
IENV *env;
```

DESCRIPTION

The DBA functions build trees to perform certain database administration functions. These functions are not provided as part of the IDL grammar implemented by *idlparse*(3I). *Itcopy*(3I) is also of interest.

Each of these operates on a particular database whose name is passed as *dbname*. The working database is the database that is open when the commands are sent by *iputtree*(3I).

The dump and load operations all take an optional *dbfile* and/or *txfile* to represent the name of an IDM file in the working database to use as a source or destination for the database dump or the transaction dump respectively. If these are CHARNULL and the tape parameter is provided then IDM tape is used. If the tape parameter is also CHARNULL then I/O is engaged with the host; it is up to the user program to ensure that this I/O is handled properly, since none of these routines return data structured as a target list.

The *tape* options are defined in *itapeopts*(3I).

Itrollf produces a tree to roll forward a database from a transaction log until the given date. If the date is not given (i.e., if CLOCKNULL is passed), the entire log is rolled forward. The date is specified as a CLOCK datum; see *getclock*(3I).

In all cases, options set in the *env*ironment are added to the tree. If *env* is IENVNULL a default environment is used.

All functions return a tree that will execute the specified function when sent to the IDM/RDBMS software using *iputtree*(3I). It is then up to the user program to send or receive any additional data that the IDM/RDBMS software expects, such as a load image. The routines *ifread*(3I) and *ifwrite* are the usual means of accomplishing this.

EXAMPLES

The call:

```
t = itdbdump("db", CHARNULL, "tx", CHARNULL, IENVNULL);
```

produces a tree which, when executed, will dump the database "db" to the host and the transaction log for "db" to the IDM file "tx" in the working database.

```
t = itdbload("db", CHARNULL, "volume(d123),unit(1)", IENVNULL);
```

produces a tree that will load database "db" from IDM tape, verifying that volume "d123" is mounted before the load begins. File zero from unit one will be read.

SEE ALSO

getclock(3I), iesetopt(3I), ifread(3I), ifopen(3I), iputtree(3I), itapeopts(3I), itcopy(3I), iftltape(4I)

NAME

_dsctoidm, _idmtodsc — descriptor-based type (iDSC) conversion hooks

SYNOPSIS

```
int _dsctoidm(dsc, ptype, len, val)
BYTE *dsc;
int *ptype;
int len;
BYTE *val;

_idmtodsc(type, len, val, dsc)
int type;
int len;
BYTE *val;
BYTE *dsc;
```

DESCRIPTION

N.B.: These routines are used internally by IDMLIB routines. They are not for use by applications. System porters must provide these routines if they wish to support descriptor-based types.

_Dsctoidm converts types represented by the descriptor ***dsc*** to one of the legal IDM system types. The resulting type is stored indirectly through ****ptype*** and the value is stored into the buffer ***val***. The value may not exceed ***len***. The actual length of the resulting value is returned.

_Idmtodsc converts an IDM system datum represented by ***type***, ***len***, and ***val*** to the type indicated by the descriptor ***dsc***.

Descriptors are assumed to contain a buffer for (or a pointer to) the actual value.

These routines are invoked when a datum of type iDSC is passed to one of the level three IDMLIB routines.

Programs using descriptors are inherently non-portable.

SEE ALSO

intro(3I)

NAME

exchandle, excahandle, excdhandle, excraise, excvraise, excignore, excprint, excfprint, excbackout, excprbo, excabort, excalock, excaunlock, exccleanup, bocleanup — exception and message handling package

SYNOPSIS

```
#include <exc.h>

int exchandle(pattern, func)
char *pattern;
FUNCP func;

excahandle(pattern, func, arg)
char *pattern;
FUNCP func;
BYTE *arg;

excdhandle(pattern, func, arg)
char *pattern;
FUNCP func;
BYTE *arg;

excraise(exc, arg1, arg2, ..., CHARNULL)
char *exc;
char *arg1, *arg2, ...;

excvraise(excv)
char **excv;

int excignore(excv, arg)
char **excv;
BYTE *arg;

int excprint(excv)
char **excv;

int excfprint(excv, outifp)
char **excv;
IFILE *outifp;

int excbackout(excv, arg)
char **excv;
BYTE *arg;

int excprbo(excv, arg)
char **excv;
BYTE *arg;

int excabort(excv, arg)
char **excv;
BYTE *arg;

excalock()

excaunlock(force)
BOOL force;

exccleanup(func, arg)
FUCNP func;
BYTE *arg;
```

MPOOL *bocleanup(idmifp, oldmpool)
IFILE *idmifp;
MPOOL *oldmpool;

DESCRIPTION

The exception package is a general-purpose facility to help formalize the handling of special conditions that require abnormal flow of control. A function or procedure represents a context; if it agrees to handle a particular exception by declaring a handler routine, any time that exception is raised in that function or in a subordinate function, that routine will get control. Exceptions handlers nest, so if *f*() calls *g*() calls *h*(), and *f* and *h* both agree to handle EXCXXX, then if EXCXXX is raised in *f* or *g*, control will return to *f*, but if EXCXXX is raised in *h*, control will be returned to *h* rather than *f*.

Exchandle agrees to handle any exception matching the pattern (described in *pmatch*(3I)). It returns zero on first return, and the return value of the handler for subsequent returns. *Excraise* or *excvraise* cause an exception to happen, i.e., "be raised". When an exception EXCXXX is raised, the package looks backwards on the stack of exception handlers built by *exchandle* until it finds the most recent handler with a pattern matching the exception being raised. The handler procedure *func* is then called with an argument vector *excv* and the argument *arg*. The zeroth element of that *excv* is the actual exception being raised (e.g., EXCXXX), and the remaining arguments in the vector correspond to the remaining arguments passed to *excraise*. *Excahandle* is identical to *exchandle* except that a second argument may be passed to the handler procedure. *Excraise* and *excvraise* are identical except that the latter passes the *excv* directly. The final argument must be CHARNULL. The arguments are copied before processing the exception.

The handling function *func* may:

- Return with value zero which will cause the *excraise* to return.
- Return non-zero which will cause the *exchandle* that set the handler to return again with that value. This is refered to as "backing out" to the handler. See the section below on Backout Functions for special backout handlers.
- Raise the exception again (after possibly modifying the severity or arguments), which causes it to be passed back to the previous willing handler.

If there are no handlers willing to handle this exception, a default handler is invoked. Default handlers are like regular handlers, except:

- They are not removed automatically when the procedure that sets them exits, that is, they remain in force until explicitly removed.
- The handler may not back out (return non-zero) since the context they were set in may no longer exist. If it does, the process is aborted.
- They are set using *excdhandle* instead of *exchandle*. Since they can never back out, *excdhandle* returns no value.

If no default handler is specified, then the exception name is used to select a message using *IftMText*(4I) which is printed on the diagnostic output. The exception then returns or the process exits, depending on the "severity" of the exception (see below). This is analogous to the default action of a signal. This technique should be used for printing *all* messages generated by libraries in order to support multilanguage I/O and to insure that the user can do special message formatting as required.

The handler *func* executes in a subordinate context to the function executing the raise call. Thus, if it raises another exception, the new exception will be interpreted relative to the function that called *excraise* rather than relative to the function that called *exchandle*.

As a special case, *excraise* and *excvraise* will never return if a message of severity "abort" is raised. The procedure may return nonlocally; otherwise the program is terminated.

Messages always begin with at least two asterisks for easy recognition. The number of asterisks reflects the severity of the message.

If the *func* argument to *exchandle* is FUNCNULL, the exception is no longer handled at this level, i.e., it is passed back to anyone who previously preferred to handle it. The handler is also removed when the routine that sets it returns.

Five canned functions are supplied that may be passed to *exchandle*:

- *Excignore* will cause the exception to be ignored.
- *Excbackout* will cause the *exchandle* call to return again with value one.
- *Excprint* causes the exception to be printed and otherwise ignored.
- *Excfprint* causes the exception to be printed on the output file specified and otherwise ignored.
- *Excprbo* arranges to print the exception (by reraising the exception) and then returns one, causing backout. Abort severity exceptions are first downgraded to Error exceptions.
- *Excabort* converts the exception to an Abort severity exception and reraises it; the usual effect is to print the exception and then abort the process exactly as though no one had been willing to handle the exception.

Critical sections can be protected using *excalock* and *excaunlock* to lock and unlock asynchronous exceptions respectively. It is almost always an error to leave these exceptions locked for a long time; these routines are intended to be used to lock modification of a critical global data structure (i.e., no more than a few instructions) rather than large blocks of code. *Excaunlock* will process any exceptions that were raised during the locked interval. *Excalock* and *excaunlock* nest if the *force* parameter to *excaunlock* is FALSE. If TRUE, exceptions are completely unlocked regardless of the nesting level; this is normally used during exception backout.

Procedures that must get control during exception backout to do cleanup operations should use *exccleanup*. "Cleanup functions" are called when the stack is being unwound due to an exception handler returning non-zero (backing out). Note that cleanup functions are **NOT** passed the argument vector from *excraise*. Any number of cleanup functions may be set. These handlers will be called (and the functions removed) in the reverse order from setting. Cleanup functions are removed as they are called so that duplicates will not exist in the exception handler list should the code continue execution. See the example below. Cleanup functions are normally used to release local resources.

After a major backout where memory may have to be freed, etc., the routine *bocleanup* may be called to do cleanup actions. The *idmifp* will be canceled if supplied (i.e., if not IFNULL). If an *oldmpool* is supplied, this memory will be released and a new pool created and returned. The new pool is guaranteed to be at the same position in the memory pool tree as the old pool. See *xalloc*(3I) for details of memory pools. Asynchronous exceptions will be reenabled.

EXCEPTION CODES

Exception codes are text strings. Every exception code is also a message code. They must be in the format:

S:EXCCODE

The *S* field is a one character severity indication, selected from the set:

I	Information
S	Success
C	Continue

R	Respond
W	Warning
T	Transient
E	Error
U	Usage
A	Abort

Information — These exceptions give no information that the user must know, but such information may be convenient. For example, copy utilities may raise an "I:" exception periodically with the expectation that it will be printed to let the user know how far they have gotten.

Success — These tell the user of the successful completion of a step. They may be omitted for expert users. For example, copy utilities may terminate with a success message including the number of tuples actually copied; expert users may prefer to have this information suppressed.

Continue — These exceptions invite the user to continue with some action; for example, in a screen-based system, a continue message might be generated between each frame.

Respond — These exceptions indicate that an unusual but not erroneous condition has occurred that requires human intervention, e.g., "End of tape; mount next volume."

Warning — These exceptions are raised when some condition has occurred that may be an error.

Transient — Transient exceptions are usually caused by asynchronous events, operator interrupts, transient resource exhaustion, or some problem that is due not to a user error but rather to a condition that is unlikely to occur again. The user is invited to try again later. Programs raising transient exceptions are not expected to behave in the same way if run again.

Error — Error exceptions are due to a user error. The program will normally try to continue processing if possible, but it is certain that incorrect results will occur.

Usage — Raised only by ***usage*** (see ***crackargv***(3I)) when a program is invoked incorrectly. If there is a message associated with this exception it will be printed. In any case, this terminates the process exactly like an "Abort" severity exception (see below).

Abort — These indicate catastrophic errors that immediately abort processing if some exception handler does not arrange to back out. It is not possible for the current routine to continue processing.

The *EXCCODE* field uniquely identifies the exception and the associated message. It is a structured field, consisting of a series of dot-separated names reading from most to least significant. Each of these names should be descriptive but "reasonably" short, consisting exclusively of upper case letters, digits, and underscores. For example, the code "IDMLIB.IO.WLR" might represent a wrong length record error in the IO submodule of IDMLIB.

Note that the severity is ***not*** considered part of the name, so codes "E:XXX" and "A:XXX" are the same message, but with different severities.

Conventions

Exceptions that represent error messages, measure tokens, or done bits from the database server begin with the word "IDM". Exceptions from level-one or level-two IDMLIB modules begin with the word "IDMLIB". Exceptions from the level-three IDM interface module begin with the word "IDMRUN". Exceptions generated by applications (e.g., **idmfcopy**) begin with the name of the

application.

Within IDMLIB, the second word of a three-or-more-part exception code identifies the major module that raised the exception. Common modules are "IDM" for IDM-specific interfaces, "IO" for the Input/Output module, "CNVT" for the data conversions, or the name of the routine generating the error.

Within the IDMLIB.IO module, file-type-specific messages have the name of the file-type module (with the "Ift" removed) as the third word, e.g., "IDMLIB.IO.SCAN.NOROOM" is the error "NOROOM" from the *IftScan*(4I) module.

GLOBALS

FileName	If set, print as the input file name with messages.
LineNumber	If non-negative, printed with messages.

EXAMPLE

```
#include <idmlib.h>
#include <exc.h>
#include <idmmpool.h>

main()
{
        MPOOL *mympool = MPOOLNULL;
        extern maincatch();
        extern MPOOL *bocleanup();
        extern MPOOL *DefMpool;

        INITIDMLIB("demo");

        exchandle("*:USER.EXC", maincatch);

        /* handle interrupts and back out */
        if (exchandle("T:IDMLIB.ASYNC.*", excbackout) == 0)
                DefMpool = mympool = newmpool(0, MPOOLNULL);
        else
                DefMpool = mympool = bocleanup(IFNULL, mympool);

        /* this call will cause maincatch to be called */
        subr();

        exchandle("*:USER.EXC", FUNCNULL);

        /* this call will abort the process */
        subr();
}

maincatch(excv)
        char **excv;
{
        printf("caught exception %s\n", excv[0]);

        /* return zero to cause excraise to return */
        return (0);
}
```

```
subr()
{
        MPOOL *temppool = MPOOLNULL;
        extern freempool();
        extern subrcatch();

        /* create a new memory pool to illustrate resource release on backout */
        temppool = newmpool(0, MPOOLNULL);

        if (exchandle("T:IDMLIB.ASYNC.INT", subrcatch) != 0)
                return;

        /*
        ** Backout function -- freempool(temppool);
        **      Release resource when subrcatch backs out
        **      after interrupt.
        */

        excCleanup(freempool, _ _ temppool);

        printf("try interrupt now\n");
        sleep(5);
        excraise("E:USER.EXC", CHARNULL);
}

subrcatch(excv)
        char **excv;
{
        printf("congratulations!  you typed ^C!\n");

        /* return non-zero to cause exchandle to back out */
        return (1);
}
```

WARNINGS

It may not always be possible to build an efficient implementation of the exception handler. Avoid calling ***exchandle*** inside inner loops, or inside functions that get called frequently. In general it is safe to use ***excraise*** however.

The use of ***setjmp*** and ***longjmp*** in programs that link to ***libidmlib.a*** is not recommended. If the user code performs a ***longjmp*** over active contexts which called ***exchandle***, then the exception stack will become out of sync and strange behaviour will occur.

Since it is very hard to predict all calling sequences (to know if a context on the stack set a exception handler), it is recommended that user code convert to using only the exception facility.

IMPLEMENTATION NOTES

The UNIX implementation is quite flexible and can probably be adapted to your environment. This implementation requires that your system supply you with the ***setjmp***(3) primitives to do non-local gotos. You must supply two internal assembly-language routines that manipulate the run-time program stack: ***_excpra*** which returns a pointer to the return address of your parent, and ***_excdisable*** which cleans up a context at a given level. ***Exchandle*** is actually a macro that calls ***_excvect*** and then does a ***setjmp*** on the return to save the possible backout address.

When *_excvect* is called, it calls *_excpra* to find the return address of the function that called it. If it is not the address of *_excdisable* then this is a first call at this level, and initialization must occur: a context is allocated, the old return address is stored in the context, and the return address is replaced with the address of *_excdisable*. Then in any case the context is adjusted to reflect this exception handler.

When the function returns, *_excdisable* will be executing in the stack frame of the caller of the function that placed the handler. It should deallocate the context. It then does a jump to the saved return address, simulating the last part of the return statement.

On UNIX, the following mappings of signals to exceptions apply:

UNIX	EXCEPTION
SIGHUP	T:IDMLIB.ASYNC.INT
SIGINT	T:IDMLIB.ASYNC.INT
SIGILL	A:IDMLIB.ASYNC.NOFP*
SIGALRM	A:IDMLIB.ASYNC.ALARM
SIGTERM	T:IDMLIB.ASYNC.TERM
SIGTSTP	T:IDMLIB.JOB.SUSPEND
SIGCONT	T:IDMLIB.JOB.CONTINUE

(*Only on systems that have no floating point hardware.)

Other signals have default actions.

The routine *_excinit* is called by *INITIDMLIB* to do initialization; it must be defined by the implementation. On UNIX, it arranges to catch signals. The job-control signals, SIGTSTP and SIGCONT, are caught and handled in *IftLoTerm*(4I).

SEE ALSO

exit(3I), pmatch(3I), IftMText(4I), IftLoTerm(4I), messages(5I), signal(2), setjmp(3)

NAME

exit — terminate program

SYNOPSIS

exit(stat)
RETCODE stat;

DESCRIPTION

Exit is the normal means of terminating a program. *Exit* performs necessary cleanup actions and returns *stat* to the operating system.

This call can never return.

The *stat* should be an error code as defined in *geterr*(3I).

IMPLEMENTATION NOTES

It may be necessary to map *stat* to a system exit status code.

This routine must call *_icleanup* before exiting to invoke *onexit*(3I) routines. Possible recursive invocations of *exit* will be handled by *_icleanup*.

If the system *exit* performs additional cleanup actions it may be necessary to redefine the name of this routine (for example, using **#define exit _iexit**) so that the IDMLIB exit routine can perform its cleanup and then call the system exit routine.

If the program calls *fork*(2), the child process will need to **#undef exit** before calling *exit*() to avoid freeing resources inherited from the parent process. In particular, a parent database server connection will be closed if the IDMLIB *exit* is called by the child process.

SEE ALSO

onexit(3I), retcode(5I)

NAME

fmtclock, fmtdate, fmtintvl — date/time output formatting

SYNOPSIS

```
#include <clock.h>

char *fmtclock(clock, zone)
CLOCK *clock;
int zone;

char *fmtdate(date)
DATE *date;

char *fmtintvl(clock, verbose)
CLOCK *clock;
BOOL verbose;
```

DESCRIPTION

Fmtdate and *fmtclock* turn the specified *date* or *clock* value (described in *getclock*(3I)) into a string in the system default format. For example, this might produce "Tue Mar 29 16:59:46 1983" or "29-MAR-83 16:59:46" depending on the host computer's operating system.

The *zone* parameter to *fmtclock* specifies the time zone in which the value should be interpreted; the semantics are identical to the *zone* parameter to *clocktodate* (see *getclock*(3I)).

Fmtintvl is similar to *fmtclock* except that it assumes that the clock represents an interval; typically the output will be something like "3+12:03:00" or "3 days, 12 hours, 3 minutes" depending on the setting of the *verbose* flag.

WARNINGS

The return values point to static data whose content is overwritten by each call.

Fmtclock and *fmtdate* may silently fail for dates before Jan. 1, 1900 or after Feb. 28, 2100.

IMPLEMENTATION NOTES

If the time zone is not available from the system, it should be supplied as a system parameter (see *getparam*(3I)).

The routines *fmtclock* and *fmtdate* are environment-dependent; *fmtintvl* is environment-independent.

SEE ALSO

getclock(3I), parsedate(3I)

NAME

getclock, clocktodate, datetoclock, diffclock, IDMTOTICKS, TICKSTOIDM — date/time manipulation

SYNOPSIS

```
#include <clock.h>

CLOCK *getclock()

DATE *clocktodate(clock, zone)
CLOCK *clock;
int zone;

CLOCK *datetoclock(date)
DATE *date;

CLOCK *diffclock(c1, c2)
CLOCK *c1;
CLOCK *c2;

long TICKSTOIDM(ticks)
long ticks;

long IDMTOTICKS(idmtime)
long idmtime;

typedef struct
{
    long    cl_day;     /* days since the epoch */
    long    cl_ticks;   /* clock ticks since midnight */
} CLOCK;

typedef struct
{
    short   dt_ticks;   /* ticks (parts of a second) */
    short   dt_sec;     /* seconds */
    short   dt_min;     /* minutes */
    short   dt_hour;    /* hour */
    short   dt_mday;    /* day of the month */
    short   dt_mon;     /* month of the year */
    short   dt_year;    /* year */
    short   dt_wday;    /* day of the week */
    short   dt_yday;    /* day of the year */
    short   dt_zone;    /* timezone */
    BOOL    dt_isdst;   /* TRUE if daylight savings time ever used in your area */
} DATE;
```

DESCRIPTION

There are two representations for dates. The first is a *CLOCK* value, having days (gross resolution) and clock ticks (fine resolution). The day is stored as days since the ***epoch***. The time is stored as ticks (1/TICKSPERSEC of a second) since midnight. GMT is always used for the clock. It can be used to store either dates or intervals.

NAME

ftoa — floating-point to alpha conversion

SYNOPSIS

ftoa(f, buf, width, fmt, scale, prec)
double f;
char *buf;
int width;
char fmt;
int scale;
int prec;

DESCRIPTION

Ftoa converts the floating-point number *f* into a string stored in *buf* of length at most *width* (including the trailing null byte). There will be at most *prec* digits after the decimal point. Six formats are defined by *fmt*. These are:

F Regular floating-point.

E Exponential format.

G E or F format, whichever produces the smaller number of output digits.

H E or F format, with F preferred. That is, if F format will fit in the specified *width* field it will be used; E format will be used only if the number will not fit when represented in F format.

A Like H, but with decimal points aligned on the numbers represented in F format. This format is convenient for columns of numbers. Alignment is done only within E and F formats, that is, E format align with E format, F format with F formats, but E and F format do not align.

P Like A, but with the precision padded out. This is provided for compatibility with *bcdtoa*(3I).

If the number is ultimately formatted in E style, there will be *scale* digits before the decimal point.

IMPLEMENTATION NOTES

This routine must be supplied by the environment-dependent implementation for use by *printf*(3I). It may use the internal routine *fmtfloat*(3I). This routine is intended to print in a format compatible with *bcdtoa*(3I).

SEE ALSO

atof(3I), bcdtoa(3I), fmtfloat(3I), printf(3I), ecvt(3)

NAME

foldcase — fold upper to lower case in a string

SYNOPSIS

foldcase(src, dst, cnt)
char *src;
char *dst;
int cnt;

SYNOPSIS

Foldcase copies up to *cnt* bytes from *src* to *dst* folding uppercase alphabetics to lowercase as it goes. The copy terminates when *cnt* is exceeded or a null byte is encountered. The null byte will be copied.

Src and *dst* may point to the same string.

SEE ALSO

string(3I)

NAME

fmtfloat — internal floating-point output formatting routine

SYNOPSIS

```
fmtfloat(digits, neg, expon, buf, width, fmt, scale, prec)
char *digits;
BOOL neg;
int expon;
char *buf;
int width;
char fmt;
int scale;
int prec;
```

DESCRIPTION

N.B.: This routine is for internal use by *bcdtoa*(3I) and *ftoa*(3I) only — it should not be used by end-user routines.

Fmtfloat takes a string of *digits* representing a floating-point value and adds the sign, decimal point, exponent, etc. in the correct places for normal output representation. *Digits* is a string of digits converted to alpha notation. A decimal point is implied before the first digit. *Neg* is TRUE if the number is negative. *Expon* is the exponent, that is, the number of digits that should be to the right of the decimal point. It may be negative. The result is stored in *buf*; at most *width* characters (including the trailing null byte) will be stored. There will be at most *prec* digits after the decimal point. A precision of zero suppresses the printing of a decimal point, useful for printing BCD integers. Six formats are defined by *fmtfloat*:

F Regular floating-point.

E Exponential format.

G E or F format, whichever produces the smaller number of output digits.

H E or F format, with F preferred. That is, if F format will fit in the specified *width* field it will be used; E format will be used only if the number will not fit when represented in F format.

A Like H, but with decimal points aligned on the numbers represented in F format. This format is convenient for columns of numbers. Alignment is done only within E and F formats, that is, E format align with E format, F format with F formats, but E and F format do not align.

P Like A, but with the precision padded out. This is provided for compatibility with *bcdtoa*(3I).

If the number is ultimately formatted in E style, there will be *scale* digits before the decimal point.

SEE ALSO

ftoa(3I), bcdtoa(3I)

BUGS

Output buffer overflow is not properly detected with format E.

Format F does not always round correctly when the exponent is negative.

specifications may not be intermixed with other textual time information.

EXAMPLES

The following all represent October 6, 1950:

```
Oct. 6, 1950
october 6, 1950 14:30:12 edt
friday, 6 oct 50, 2 pm
TUES 6-OCT-50 1400 H
noon, 50/10/6
10-6-50 143012
6.10.50 14:30
50/10/6-14:00-PDT
6-Oct-50 10:37:19-PDT (Tue)
```

EXCEPTIONS

E:IDMLIB.CLOCK.PARSE(input)

The specified input could not be parsed.

WARNINGS

The return value points to static data whose content is overwritten by each call.

SEE ALSO

fmtclock(3I), getclock(3I)

NAME

parsedate — free-format date/time conversion

SYNOPSIS

#include <clock.h>

CLOCK *parsedate(string)
char *string;

DESCRIPTION

Parsedate reads a string that represents the date and turns it into a CLOCK structure. A heuristic parse is used that accepts a wide variety of formats. Either upper or lower case may be used within date strings. *Parsedate* can only handle dates between Jan. 1, 1900 and Feb. 28, 2100.

Unspecified date fields are copied from the current system date; unspecified time fields are set to their minimum possible values. For example, if the current date is September 12, 1983 at 11:32:05, the input "10AM September 20" would mean "September 20, 1983 at 10:00:00" and "3 PM 1980" would mean "September 12, 1980, at 3:00:00 PM."

Parsing an empty string returns the current date.

Parsedate returns CLOCKNULL and raises an exception if the input cannot be recognized or is inconsistent.

CLOCK structures are described in *getclock*(3I).

The following time zones are supported:

STD, DST	local standard, daylight-savings times, respectively.
GMT, GST	Greenwich mean time.
AST, ADT	Atlantic standard, daylight-savings time.
EST, EDT	Eastern standard, daylight-savings time. Synonymous with AST, ADT
CST, CDT	Central standard, daylight-savings time.
MST, MDT	Mountain standard, daylight-savings time.
PST, PDT	Pacific standard, daylight-savings time.
YST, YDT	Yukon standard, daylight-savings time.
HST, HDT	Hawaii standard, daylight-savings time.

Parsedate also recognizes military time zones represented by the characters 'A' through 'Z' (except for 'J') where 'H' is Pacific Standard Time and 'Z' is Greenwich Mean Time.

Specifications indicating daylight-savings times are ignored if daylight savings was not in effect on the specified date. For example, in the date string "dec 20, 2:30 pm dst" the time is known to be standard, not daylight.

Four-digit numbers are interpreted as times if possible, otherwise as dates. The string "1915" parses to the time 7:15 PM, while the string "1970" parses to the year 1970.

Date formats may be syntax-sensitive. For example, the date "9/2/84" parses to September 2, 1984, while "2.9.84" is interpreted as February 9, 1984.

Six-digit numbers are interpreted as dates in "YYMMDD" format, if possible, otherwise as military time specifications.

Parsedate accepts IDM time specifications in the format "idmtime < ***days*** > [< ***ticks*** >]" where ***days*** is an integer representing the number of days since the epoch and ***ticks*** is an integer representing the number of 60ths of a second since midnight. The ***ticks*** are optional. IDM time

HTAPE.ERR.WRONGVOLUME
The wrong volume was mounted.

HTAPE.FILENOTFOUND
File not found on host tape.

HTAPE.MOUNT(volume, unit)
Mount the specified *volume* on host tape unit *unit*.

HTAPE.NEXTVOLUME
Ready for next volume.

ITAPE.MOUNT(volume, unit)
Mount the specified *volume* on IDM tape unit *unit*.

ITAPE.NEXT
Mount the next IDM tape volume. Respond with the unit number of the drive.

WARNINGS

Askoperator may return CHARNULL even if *hasoperator* previously returned TRUE if the operator logs out; in this case the user program must be careful not to go into a loop.

EXCEPTIONS

W:IDMLIB.OPERATOR.NONE
Raised by *askoperator* and *telloperator* if there is no operator available.

IMPLEMENTATION NOTES

Care must be taken to insure that these implementations are extensible, that is, that new operators and new messages may be added easily.

On UNIX, this just communicates with the user. *Hasoperator* tests whether input is coming from the terminal. On other systems this is likely to test whether the operator is currently in attendance, or may just return TRUE.

On VMS, all communications go to the operator named by the system parameter OPERATOR (determined from the logical name IDM_OPERATOR). Only IDM tape messages are implemented using this facility, as host tape messages are handled automatically by RMS. The OPERATOR parameter may be set to any of the standard VMS operator identifiers: TAPES, CARDS, CENTRAL, DEVICE, DISKS, NETWORK, PRINT, and OPER1 through OPER12. You may also direct IDMLIB operator messages to your own terminal by specifying SELF or ME.

SEE ALSO

getprompt(3I)

NAME

telloperator, askoperator, hasoperator — communicate with the system operator

SYNOPSIS

```
telloperator(oper, msgcode, param, ..., CHARNULL)
char *oper;
char *msgcode;
char *param;

char *askoperator(buf, len, oper, msgcode, param, ..., CHARNULL)
char buf[ ];
int len;
char *oper;
char *msgcode;
char *param;

BOOL hasoperator(oper)
char *oper;
```

DESCRIPTION

Telloperator sends the message to the specified system operator.

Askoperator sends the message to the specified operator exactly like *telloperator* and then waits for an operator response. It returns *buf* if the response was successful, CHARNULL if the operator is not in attendance.

Hasoperator returns TRUE if it is possible to communicate with someone acting as the specified system operator.

The system may have several operators. The following *operators* are specifically defined:

ITAPE The IDM tape operator (for IDM tape mount requests).

HTAPE The host tape operator (for host tape mount requests).

PRINTER The line printer operator (for special forms requests).

The *msgcode* and *params* behave like exceptions, where *msgcode* is modified to be an exception name. The last parameter must be CHARNULL.

Defined operator, message code, and parameter combinations are:

HTAPE.EOV
: At end of volume.

HTAPE.ERR.INVALID
: Host tape is not a valid format.

HTAPE.ERR.NODRIVE(drivename, error)
: Cannot open *drivename*: system reported *error* as the cause.

HTAPE.ERR.NOHDR1
: No HDR1 label on tape.

HTAPE.ERR.NOTONLINE(volume, unit, error)
: The specified tape *unit* could not be accessed when trying to read the named *volume*.

HTAPE.ERR.WRONGTAPE
: Incorrect tape.

HTAPE.ERR.WRONGVOL(needed, actual)
: Incorrect volume: *needed* required, *actual* mounted.

NAME

onexit, offexit — transfer control on exit

SYNOPSIS

```
onexit(exitfn, arg)
FUNCP exitfn;
BYTE *arg;

offexit(exitfn, arg)
FUNCP exitfn;
BYTE *arg;
```

DESCRIPTION

Onexit specifies functions to be called when the process exits. Each *exitfn* is called with the specified *arg*. The functions will be called in the reverse order in which they were established. Duplicate calls to *onexit* are ignored.

Offexit removes the entry that matches. It is not an error if no entries match.

IMPLEMENTATION NOTES

Exit(3I) must call *_icleanup* to invoke the exit routines set by *onexit*.

On VMS, *exit*() either calls the system service *SYS$EXIT*() or the *exit* routine in the VAX C Run-time Library, depending on how a program is linked. In either case, a VMS exit handler is declared in *INITIDMLIB*() that will call *_icleanup* to invoke the exit routines set by *onexit*. This way, all exit handlers will be called regardless of how the program exits.

SEE ALSO

exit(3I)

NAME

mapsym — translate symbol name into integer value

SYNOPSIS

int mapsym(prefix, sym)
char prefix;
char *sym;

DESCRIPTION

Mapsym translates a symbolic name having the given *prefix* into an integer by doing a file lookup in the file specified by the SYMFILE parameter (see *getparam*(3I) and *params*(5I)).

If the parameter begins with a digit, it is converted to integer and returned directly.

The following prefixes are defined:

- d IDM done status bits.
- o IDM option values.
- t IDMLIB trace flags.
- * IDM trace flags.

Upper case prefixes are reserved for customer use. All other prefix characters are reserved for Britton Lee use.

Case is ignored in *sym* comparisons.

EXAMPLES

mapsym('t', "PROTECT") → 26
mapsym('t', "Protect") → 26
mapsym('x', "38") → 38

EXCEPTIONS

E:IDMLIB.MAPSYM.NOSYM(prefix, symbol)
No mapping for the specified symbol exists.

SEE ALSO

atoi(3I), getparam(3I), params(5I), symfile(5I)

NAME

makefname — make file name from components

SYNOPSIS

char *makefname(file, directory, filetype)
char *file;
char *directory;
char *filetype;

DESCRIPTION

Makefname makes a fully qualified host file name from the constituent pieces: *file* is the basic file name, *directory* is the name of the directory in which to find *file*, and *filetype* is the filetype part of the file name.

If *directory* is CHARNULL or the null string then the current directory is used. The following special strings are also recognized and interpolated:

LOGIN The current user's login directory.

USRPROFILE The profile directory for the current user, that is, a directory in which to find user startup and configuration files.

SYSPROFILE A system profile directory.

If *filetype* is CHARNULL then no filetype is added to the file name. Filetypes compiled into programs should never exceed three characters for maximum portability.

Components that are already present in *file* are not replaced or added. That is, if *file* already had a directory and a filetype *makefname* would return *file*.

Since the syntax of directories cannot be standardized it is expected that this routine will always be called with one of the builtin directory names or by calling *getparam*(3I).

EXAMPLES

The call

```
makefname("iqppro", "_USRPROFILE_", "idl")
```

might return the following strings:

UNIX	/a/sw/eric/.iqppro.idl
VMS	DBA0:[eric]iqppro.idl
CMS	iqpro.vmuserid

The call

```
makefname("/usr/idl/x", "/tmp", "idl")
```

might return:

UNIX	/usr/idl/x.idl
CMS	x.idl

IMPLEMENTATION NOTES

This routine is machine dependent.

On UNIX, a "filetype" is defined to be anything after a dot found after the second position of the final component of the pathname. This allows a leading dot in the filename that will not be considered the beginning of a filetype. Correspondingly, the "_USRPROFILE_" directory is actually the home directory plus a leading dot as shown in the examples above.

SEE ALSO

getparam(3I)

LIMITATIONS

Keylook can only handle string/integer pairs. This is insufficient for some applications.

NAME

keylook, usage — perform binary search on a given table

SYNOPSIS

#include <keylook.h>

keylook(string, table)
char *string;
KEYTABLE *table;

DESCRIPTION

Keylook looks up the given *string* in the given *table*, and returns the integer token associated with the table entry.

Keylook uses a fast binary search algorithm, so it is very efficient for medium-sized tables. Very small tables are probably better handled by linear search, very large tables by some hashing method.

The first entry of the table is the default returned if the search string is not found. The method of specifying a lookup table is shown below in the example.

EXAMPLE

```
#include <idmlib.h>
#include <keylook.h>


KEYWORD             Keywrds[] =
{
        /* keyword      token returned  */
        {CHARNULL,      -1          }, /* default */
        {"and",         I_AND       },
        {"any",         I_ANY       },
        {"as",          I_AS        },
        {"by",          I_BY        },
        {"from",        I_FROM      },
        {"in",          I_IN        },
        {"on",          I_ON        },
        {"set",         I_SET       },
        {"to",          I_TO        },
        {"with",        I_WITH      }
};

KEYTABLE Keytable =
        { Keywrds, _KTAB_SIZE(Keywrds) };

int
get_token(str)
        char *str;
{
        /* this returns -1 if 'str' not found in Keytable */
        return (keylook(str, &Keytable));
}
```

The macro *_KTAB_SIZE* is provided in keylook.h for convenience. The KEYWORD array is referenced only in the KEYTABLE declaration.

SEE ALSO

idlparse(3I), iesetopt(3I), iputtree(3I), *System Programmer's Manual* .

NAME

itxcmd, itxprog, itxsetp — build trees to execute stored commands/programs

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

ITREE *itxcmd(cmdname, env)
char *cmdname;
IENV *env;

ITREE *itxprog(progid, env)
long progid;
IENV *env;

itxsetp(t, name, type, len, val)
ITREE *t;
char *name;
int type;
int len;
BYTE *val;
```

DESCRIPTION

Itxcmd and *itxprog* produce trees for the **execute command** and **execute program** operations respectively. The tree returned includes no parameters. Parameters may be added using successive calls to *itxsetp*. The *name* of the parameter may be CHARNULL to specify unnamed parameters. The *type* and *len* describe both the data in the host and to be sent to the IDM/RDBMS software. Type iSTRING is converted to iCHAR but is otherwise semantically equivalent (i.e., if a length of −1 is specified then the *strlen* of the argument is used). Values of type iPCHAR will have the standard pattern characters mapped to internal form.

Options set in the *env*ironment will be set in the command tree. If *env* is IENVNULL, a default environment will be used.

EXAMPLES

```
/* execute update with name = "mike", amount = 44 */
t = itxcmd("update", IENVNULL);
itxsetp(t, "name", iSTRING, −1, __ "mike");
itxsetp(t, "amount", iINT2, 2, __ &amnt);

/* help "relation" */
t = itxcmd("help", IENVNULL);
itxsetp(t, CHARNULL, iSTRING, −1, __ "relation");

/* execute program 2112001 with ("foobar", 7) */
t = itxprog(2112001L, IENVNULL);
itxsetp(t, CHARNULL, iSTRING, −1, __ "foobar");
itxsetp(t, CHARNULL, iINT1, 1, __ &seven);
```

EXCEPTIONS

E:IDMLIB.IDM.ITXCMD
: No name was specified to *itxcmd*.

E:IDMLIB.IDM.ITXSETP.BADTYPE(type)
: The tree specified is not an **execute command** or **execute program** tree.

E:IDMLIB.IDM.ITXSETP.NOTREE
: The user did not correctly specify a value.

```
**      retrieve (r.name, a.name)
**      order by a.name
**      where r.relid = a.relid
**      and r.name ≠ "relation";
*/

/* build the range table */
rlist[0] = "relation";
rlist[1] = "attribute";
rlist[2] = CHARNULL;

/* build the target list */
tlist[0] = itvar(0, "name");
tlist[1] = itvar(1, "name");
tlist[2] = ITNULL;

/* build the qualification */
l = itvar(0, "relid");
r = itvar(1, "relid");
qlist[0] = itnode(l, r, iEQ, 0, BYTENULL);
l = itvar(0, "name");
r = itnode(ITNULL, ITNULL, iCHAR, -1, "relation");
qlist[1] = itnode(l, r, iNE, 0, BYTENULL);
qlist[2] = ITNULL;

/* build the order list */
olist[0] = 2;
olist[1] = 0;

/* now create the entire tree */
it = itqstmt(iRETRIEVE, rlist, tlist, qlist, olist, IENVNULL);
```

SEE ALSO

idlparse(3I), itnode(3I), sqlparse(3I)

NAME

itqstmt — build a tree for a general query statement

SYNOPSIS

```
#include <idmtree.h>
#include <idmsymbol.h>
#include <idmenv.h>

ITREE *itqstmt(cmnd, rlist, tlist, qlist, olist, env)
int cmnd;
char **rlist;
ITREE **tlist;
ITREE **qlist;
int *olist;
IENV *env;
```

DESCRIPTION

Itqstmt builds query trees for most of the general query statements (**retrieve**, **append**, etc.) without calling a full parser such as *idlparse*(3I) or *sqlparse*(3I). It is intended for use in environments that require ad hoc queries of some sort (so a precompiler is insufficient) but which still have memory or performance requirements that prohibit linking of the full parser — specifically, 4th Generation interpreters.

The user is still required to build some subtrees; additional documentation can be found in the *System Programmer's Manual* . In other words, this routine encapsulates the non-public interfaces.

Cmnd is the type of the tree, e.g., iRETRIEVE or iDELETE. *Rlist* is CHARNULL-terminated list of relation names used in the query. The relation number is determined by the index into the vector. Any VAR nodes in the other lists must match this index.

Tlist, an ITNULL-terminated list of targets, can be simple VAR nodes or complex expressions. Each entry will have a iRESDOM node tacked on. *Qlist* is an ITNULL-terminated list of qualification terms. These are conjoined to create the qualification.

If a particular ordering is required, *olist* may be specified as a zero-terminated list of order terms. Each integer entry is an index into *tlist*. For the purposes of this array, *tlist* is assumed to have an origin of one — that is, if *olist*[0] == 1, that implies that the first target (i.e., *tlist*[0]) should be ordered. If the entry is negative, the ordering is descending instead of ascending.

Env is an environment used for execution as in the other routines.

Itqstmt returns a tree that can be executed as though it had been returned from *idlparse*(3I) or one of the other tree creation routines. This tree will have iRESDOM nodes rather than iRESATTR nodes, so the retrieved data will be unnamed. Also, there is no way to specify **.all** at this time.

DEFICIENCIES

Possibly should check its arguments more carefully; as it stands the database server will give a diagnostic, but it may be quite obscure.

The **iATTRALL** (a.k.a., **.all**) should be supported.

EXAMPLE

In the following example, the routine *itvar* is used to create VAR nodes.

```
/*
** Handcraft the query:
**
**      range of r is relation;
**      range of a is attribute;
```

NAME

itprint — print a tree for debugging

SYNOPSIS

#include <idmtree.h>

itprint(tree, all)
ITREE *tree;
BOOL all;

DESCRIPTION

Itprint prints a representation of the given *tree* on the standard trace. This is not expected to be readable by mortals. If *all* is set the entire tree is printed, otherwise only the root node is printed.

SEE ALSO

itnode(3I), itfree(3I)

NAME

itnode, itvar, itroot — build an IDM tree node, VAR node, or ROOT node

SYNOPSIS

```
#include <idmtree.h>
#include <idmsymbol.h>

ITREE *itnode(left, right, type, len, valp)
ITREE *left;
ITREE *right;
int type;
int len;
BYTE *valp;

ITREE *itvar(relno, attname)
int relno;
char *attname;

ITREE *itroot(left, right, val1, val2)
ITREE *left;
ITREE *right;
int val1;
int val2;
```

DESCRIPTION

Itnode creates a new tree node. The *it_left*, *it_right*, and *it_type* fields are filled in directly from *left*, *right*, and *type* respectively. If *len* is given, it is used as the length of the node. If omitted (by passing −1), an attempt is made to determine the length from the type. If the type is a fixed length symbol, then that length is used. If it is a "length follows symbol" type, then *valp* must be non-NULL, and the string length of the value field is used.

The value field of the generated node is filled in from the *valp* if non-NULL, otherwise zeroed.

Since space for the node is allocated off of the default heap, the space must always be released when done. This can be done easily using *itfree*(3I).

VAR nodes can be created using *itvar*, supplying the range variable number and the name of the attribute desired. ROOT nodes can be created using *itroot*, supplying the left and right child pointers, and two bytes of value to put in the ROOT node itself.

SEE ALSO

itfree(3I), itree(5I)

NAME

itlprint — print IDM target list (ITLIST) for debugging

SYNOPSIS

#include <idmtlist.h>

itlprint(itl, all)
ITLIST *itl;
BOOL all;

DESCRIPTION

Itlprint prints a representation of the IDM target list *itl* on *stdtrc* for debugging. The resulting output is intended to edify gurus.

If *all* is set the entire target list is printed; otherwise only the first node is shown.

SEE ALSO

igettl(3I), itlist(5I)

NAME

itfree — free an ITREE

SYNOPSIS

#include <idmtree.h>

itfree(tree)
ITREE *tree;

DESCRIPTION

Itfree frees the space used by an IDM tree. The space must not be touched again.

All fields in all tree nodes must be allocated using *xalloc*(3I) (*itnode*(3I) has the equivalent effect).

SEE ALSO

itnode(3I), xalloc(3I)

NAME

itdefine — create tree for define command

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

ITREE *itdefine(treelist, name, definep, env)
ITREE *treelist;
char *name;
BOOL definep;
IENV *env;
```

DESCRIPTION

Itdefine encapsulates the *treelist* into a DEFINE command with given *name*, returning the resultant tree. If *definep* is TRUE then a DEFINE PROGRAM is created, otherwise a simple DEFINE is created.

Options set in the *env*ironment are set in the resultant tree. If *env* is IENVNULL a default environment is used. (These options are unused at this time.)

When a DEFINE PROGRAM is executed, the done count field is set to the command number to be passed to an EXECUTE PROGRAM. It is the responsibility of the user program to save this information.

SEE ALSO

idlparse(3I), iesetopt(3I), iputtree(3I), itxcmd(3I), *System Programmer's Manual*

NAME

itcopy — build tree for bulk copy function

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

ITREE *itcopy(dbname, in, rellist, tape, env)
char *dbname;
BOOL in;
char **rellist;
char *tape;
IENV *env;
```

DESCRIPTION

Itcopy builds a tree to execute the IDM **copy** function. If *in* is set, a COPY IN tree is built, otherwise a COPY OUT tree is built. *Rellist* is a CHARNULL-terminated array of pointers to names of relations to be copied to or from database *dbname*; if NULL all user relations in database *dbname* are copied. If *tape* is not CHARNULL then IDM tape will be used; the format of the *tape* parameter is described in *dba*(3I).

Options set in the *env*ironment are included in the copy tree. If *env* is null a default environment is used.

After the copy tree is complete, it can be sent to IDM/RDBMS using *iputtree*(3I). The database system will then return results formated to look like a series of **retrieve** statements; the routines *igettl*(3I) and *igettup*(3I) can be used to simplify this. If it is not necessary to interpret the results (e.g., if copy is being used to back up a relation) then data can be read until end-of-file.

SEE ALSO

dba(3I), iesetopt(3I), igettl(3I), igettup(3I), iputtl(3I), iputtree(3I), iputtup(3I), ienv(5I), *System Programmer's Manual* .

NAME

itapeopts — parse IDM tape options

SYNOPSIS

BYTE *itapeopts(optlist)
char *optlist;

DESCRIPTION

Itapeopts converts a text description of IDM tape options to a twenty-eight byte option value as described in *SPM*. This string is suitable for direct use by the IDM/RDBMS software.

Optlist is a comma-separated list of *name(value)* pairs chosen from the list:

mode(M) I/O mode; *M* may be **r** (read), **w** (overwrite), or **a** (append). Defaults to **a**.

volume(VL) A comma-separated list of the names of the volumes in this set. If specified, the header of each tape is read and verified before the tape is used. If not specified any volume is accepted. Only the first volume name is actually checked, although all will be presented to the operator. Tape reads will always check volume names on tapes 2-n (but not 1).

newname(V) The new volume name to write on the tape to replace the existing name. Can only be used in **w** mode. If not specified, the volume name is unchanged. New IDM tapes (tapes not previously written by Britton Lee's IDM/RDBMS software) must be given a new name.

fileno(N) The file number to access when reading the tape. If not specified file zero is assumed. This option is ignored when writing a tape. File numbers on IDM tape always begin at zero.

unit(N) The unit number to access. Zero by default.

erase Perform a "security erase" of the tape before writing. Only supported on some drives. Mode **w** must be specified.

norewind Do not rewind tape between writing files. Default is to rewind. Norewind is available for writes only in IDM Software Releases 35 and 40. Norewind applies to both reads and writes in RDBMS Software Release 3.5 and future RDBMS releases.

xlate(X) Perform the requested translation of data on the tape. *X* may be one of **none** (no translation), **ascii** (translate to ASCII), **ebcdic** (translate to EBCDIC), **host** (do host translation). The default is **none**.

verify(B) Turn on (B = 1) or off (B = 0) tape sequence number verification. Default is to not verify. This parameter should only be used on tapes previously written by Britton Lee's IDM/RDBMS software. Like *volume*, tape reads will automatically verify the sequence numbers on tapes 2-n.

Other fields may be specified but are ignored.

After creation, these options may be added to a tree using *itaddopts*(3I). More typically, tape options are set directly using *itcopy*(3I) or one of the routines in *dba*(3I).

SEE ALSO

intro(1I), dba(3I), igeteot(3I), itaddopts(3I), itcopy(3I), pextract(3I), *SPM*.

NAME

itaddopts — add options bytes to a tree

SYNOPSIS

```
#include <idmtree.h>

itaddopts(tree, len, options)
ITREE *tree;
int len;
BYTE *options;
```

DESCRIPTION

Itaddopts adds *len* bytes of *options* to a *tree*. No check is made to see if any of the options are already set. There is no way to delete options from an existing tree.

Options are normally set in the environment using *iesetopt*(3I). The sole reason for this routine is to allow IDM tape options.

BUGS

This routine is totally bogus.

SEE ALSO

idlparse(3I), iesetopt(3I), *IDL or SQL Reference Manual* for a description of the available options.

```
        /* write to file "outfile" and standard output */
        while (fgets(buf, sizeof(buf), fp) != CHARNULL)
        {
                puts("standard I/O ");
                printf("should be flushed if there is not a newline");
                fflush(stdout);

                ifputs(buf, istdout);
                ifflush(istdout);

                ifputs(buf, ifp);
        }

        fclose(fp);
        ifclose(ifp);

        /* must call exit */
        exit();
}
```

doerror.c:

```
#include <idmlib.h>

doerror(msg)
        char *msg;
{
        /*
        ** Note that we now use stdout,
        ** not istdout.
        */

        ifputs(msg, stdout);
}
```

To compile the program:

```
cc -o demo main.c doerror.c -listdio -lidmlib
```

If using curses, etc:

```
cc -o demo main.c doerror.c -listdio -lidmlib -lcurses -ltermcap
```

IDMLIB cursor control and graphic characters **must** go through IDMLIB I/O.

WARNINGS

It is safest to do a flush on the appropriate standard I/O file before changing I/O systems.

Be careful not to pass *iprintf* the standard I/O file (e.g., *stdout*). IDMLIB will warn you about this, but the standard I/O system will dump core.

CAVEATS

Reading on IDMLIB *istdin* for large amounts of data is not efficient due to the limitations of the I/O interleaving mechanism. When mixing the I/O systems it is preferable to use standard I/O *stdin*.

SEE ALSO

intro(3S), *UNIX Programmer's Manual*

NAME

libistdio.a — standard I/O compatibility library

SYNOPSIS

```
#include <istdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

IFILE *istdin;
IFILE *istdout;
IFILE *istderr;

iprintf(fmt [, arg ] ... )
char *fmt;

char *isprintf(buf, fmt [, arg ] ... )
char *buf;
char *fmt;
```

DESCRIPTION

Standard I/O may be used along with the IDM library I/O (IDMLIB) system by changing one include declaration and linking in the appropriate libraries **before** the standard C runtime libraries. The first operation in *main*() must initialize IDMLIB by calling the macro *INITIDMLIB*(*progname*).

Include the file *<istdio.h>* in the module containing *main*() and link the libraries *istdio* and *idmlib*.

It is not necessary to add includes of *<istdio.h>* except in modules that will also use IDMLIB. If standard I/O is not used it is simpler to only include *<idmlib.h>*.

To access IDMLIB's standard I/O system, use the files *istdin*, *istdout*, and *istderr*. IDMLIB *printf* and *sprintf* are renamed so that standard I/O versions are used. IDMLIB versions are *iprintf* and *isprintf* when *<istdio.h>* is included.

If the file *<istdio.h>* is not included in a module in which *<idmlib.h>* is included, then *stdin* refers to IDMLIB's standard input, not standard I/O's input.

EXAMPLE

main.c:

```
#include <istdio.h>

main()
{
        IFILE *ifp;
        FILE *fp;
        char buf[100];

        INITIDMLIB("demo");

        ifp = ifopen("outfile", &IftHFile, "mode(w)", IFNULL);

        fp = fopen("somefile", "r");
        if (fp == (FILE *) NULL)
        {
                doerror("can't open somefile\n");
        }
```

NAME

isleep — sleep for a real-time interval

SYNOPSIS

isleep(ticks)
long ticks;

DESCRIPTION

Isleep delays the current process by *ticks* clock ticks (as defined in *getclock*(3I)), that is, in 1/TICKSPERSEC intervals). This will be rounded as necessary to the resolution of the host clock.

Since the resolution may be crude, this should not be used for precise intervals; these are perforce environment dependent.

If *ticks* is negative, then *isleep* will simply return.

EXAMPLE

To sleep for four seconds:

```
isleep(4 * TICKSPERSEC);
```

To sleep for one-half second:

```
isleep(TICKSPERSEC / 2);
```

IMPLEMENTATION NOTES

If the host system does not have sufficient resolution to delay for the exact interval, rounding (not truncation) should be employed.

If the host system has no way of delaying a process, the exception E:IDMLIB.ISLEEP.NOCLOCK should be raised.

Isleep was added to allow for an environment-independent identify daemon; general use is probably risky.

NAME

isforegnd — are we in foreground (interactive)?

SYNOPSIS

BOOL isforegnd()

DESCRIPTION

Isforegnd returns TRUE if the process is running in foreground, i.e., if it is connected to a terminal.

IMPLEMENTATION NOTES

On UNIX and VMS, this tests to see if the standard input is a terminal. The intent of this routine is to see if we should operate interactively (e.g., give prompts).

This may be called fairly frequently, so the implementation should be reasonable efficient.

On CMS, *isforegnd* returns FALSE if *username* returns "cmsbatch" or if the user is disconnected.

A:IDMRUN.RECOMPILE("irx(cmd | prog | setp)")
Must recompile from source.

E:IDMLIB.USENEXTCMD("irx(cmd | prog)")
You should be using *irnext*(3I) instead.

SEE ALSO

intro(3I), iridl(3I), itxcmd(3I), *System Programmer's Manual*

NAME

irxcmd, irxprog, irxsetp — arrange to execute a stored command

SYNOPSIS

```
#include <idmrun.h>

RETCODE irxcmd(idmrun, cmdname)
IDMRUN *idmrun;
char *cmdname;

RETCODE irxprog(idmrun, progid)
IDMRUN *idmrun;
long progid;

irxsetp(idmrun, name, type, len, val)
IDMRUN *idmrun;
char *name;
int type;
int len;
BYTE *val;
```

DESCRIPTION

Irxcmd is a fast, special purpose version of *iridl*(3I) for **execute command** operations. A call to *irxcmd* creates a tree that will execute the stored command named *cmdname* with no parameters. Subsequent calls to *irxsetp* will add parameters with the given *name* of the specified *type*, *len*gth, and *val*ue. Parameter *name s* may be CHARNULL to specify unnamed parameters.

Irxprog is identical except that it sends an **execute program** operation.

Both *irxcmd* and *irxprog* free any existing command trees in the *idmrun* structure.

RETURN VALUES

RS_NORM The tree was successfully created.

RE_FAILURE The tree could not be created; detail is given by an exception.

EXAMPLES

The calls:

```
(void) irxcmd(idmrun, "cmd");
irxsetp(idmrun, "a", iCHAR, 2, __ "xx");
irxsetp(idmrun, "b", iPCHAR, 2, __ "r*");
irxsetp(idmrun, CHARNULL, iSTRING, -1, __ "relation");
```

are equivalent to (and faster than):

```
iridl(idmrun, "execute cmd (a = \"xx\", b = \"r*\", \"relation\")");
```

The following code saves the program id for a **define program** command.

```
(void) iridl(idmrun, "define program ... end define");
(void) irexec(idmrun);
(void) irget(idmrun, IP_DINT, __ &progid, 0);
(void) irxprog(idmrun, progid);
```

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irx(cmd | prog | setp)")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.NOTEXEC("irx(cmd | prog)")
Commands have been parsed, but not executed.

Bcd numbers can be substituted with:

```
(void) irsubst(idmrun, "xx", b→bcd_type, b→bcd_len, _ _ b→bcd_str);
```

EXCEPTIONS

As described in *icsubst*(3I).

A:IDMRUN.BADIDMRUN("irsubst")
Closed, NULL or bad IDMRUN structure.

A:IDMRUN.RECOMPILE("irsubst")
Must recompile from source.

SEE ALSO

intro(3I), iesubst(3I), iridl(3I), irexec(3I), irnext(3I), ienv(5I)

NAME

irsubst — perform substitutions in trees

SYNOPSIS

```
#include <idmrun.h>

RETCODE irsubst(idmrun, name, type, length, value)
IDMRUN *idmrun;
char *name;
int type;
int length;
BYTE *value;
```

DESCRIPTION

Irsubst associates a value with a substitution name in an IDMRUN structure almost exactly analagously to *icsubst*(3I). *Irsubst* operates on IDMRUN structures rather than directly on environments.

Type, *length*, and *value* describe the value to be substituted. If *type* is iSTRING, the *length* is ignored in favor of the string length of *value*.

If the *type* is iPCHAR, then any pattern matching characters in the string (e.g., "*", "?" in IDL, "%", "_" in SQL) will be interpreted as documented. Values of type iCHAR or iSTRING will not interpret pattern-matching characters as magic. Note that type iPCHAR does not require that pattern-matching characters be present; it only instructs IDMLIB to treat them specially if they are. iPCHAR values used in target lists will generate IDM error E39. They should be used in qualifications only.

All iSUBSTITUTE nodes must have a value associated before *irexec*(3I) may be called. However, values can be reassigned and the query rerun without reparsing the query, and without reassigning all iSUBSTITUTE nodes.

The *value* is copied; that is, changes to the memory that *value* points to will not affect the value of the substitution. When substituting BCD numbers, pass the *bcd_str* data area of the BCDNO as the *value*. *Bcd_len* should be passed in as *length* to ensure that the correct number of *bcd_str* bytes are copied.

RETURN VALUES

RS_NORM The substitution has proceeded normally.

RE_FAILURE The substitution has failed; an exception has explained why.

EXAMPLE

```
qry = "replace x (a = %q) where x.b = %r";
(void) iridl(idmrun, qry);
val = 1;
(void) irsubst(idmrun, "q", iINT2, 2, __ &val);
val = 2;
(void) irsubst(idmrun, "r", iINT2, 2, __ &val);
(void) irexec(idmrun);
val = 3;
(void) irsubst(idmrun, "r", iINT2, 2, __ &val);
(void) irexec(idmrun);
```

runs the two queries:

```
replace x (a = 1) where x.b = 2
replace x (a = 1) where x.b = 3
```

NAME

irsql — parse SQL statements

SYNOPSIS

#include <idmrun.h>

RETCODE irsql(idmrun, string)
IDMRUN *idmrun;
char *string;

DESCRIPTION

Irsql parses the SQL statements in *string* and associates the resulting query tree with *idmrun*.

Irsql accepts a sequence of SQL statements so that SQL statements can be processed in groups (see *irnext*(3I)).

The language accepted by *irsql* is described in *sqlparse*(3I).

RETURN VALUES

RS_NORM The input has sucessfully been parsed and may now be executed using *irexec*(3I).

RE_FAILURE The input could not be parsed. An exception has been raised giving details.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irsql")
Null IDMRUN or not an IDMRUN structure.

A:IDMRUN.RECOMPILE("irsql")
Must recompile from source.

E:IDMRUN.USENEXTCMD("irsql")
You should be using *irnext*(3I) instead.

Many others, described in *sqlparse*(3I).

SEE ALSO

intro(3I), sqlparse(3I), irexec(3I), irnext(3I), irxcmd(3I)

NAME

irset — set values into the IDMRUN structure

SYNOPSIS

#include <idmrun.h>

RETCODE irset(idmrun, addr, field, item)
IDMRUN *idmrun;
BYTE *addr;
int field;
int item;

DESCRIPTION

Irset sets the value contained in *addr* into the IDMRUN structure. *Field* specifies what action to take. The legal field and type is:

IP_TREE	Set the head of the command tree list to a copy of the argument (ITREE *).
IP_ENV	Set the environment to the argument (IENV *). The environment is *not* copied.
IP_DMASK	Set the done mask (int). See *igetdone*(3I).

Item is currently unused.

Improper use of this routine can cause grave damage.

RETURN VALUES

RS_NORM	The *irset* was successful.
RE_FAILURE	The set could not be performed. An exception will have been raised explaining why.

EXAMPLES

irset(idmrun, _ _ newenv, IP_ENV, 0);

Set the environment to *newenv* for all future commands associated with the *idmrun* structure.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irset")

Null IDMRUN structure or not an IDMRUN structure.

E:IDMRUN.MOREDATA("irset")

There is data remaining to be read from the previous command.

A:IDMRUN.RECOMPILE("irset")

Must recompile from source.

E:IDMRUN.SETFLD(field)

Illegal field identifier.

E:IDMRUN.SETTREE(treenum)

Cannot set specified tree number.

E:IDMLIB.USENEXTCMD("irset")

You should be using *irnext*(3I) instead.

SEE ALSO

intro(3I), igetdone(3I), irget(3I), ienv(5I), itree(5I)

```
(void) irclose(idmrun);
```

EXCEPTIONS

A:IDM.E46
No open database

A:IDMRUN.BADIDMRUN("irreopen")
Null IDMRUN structure or not an IDMRUN structure.

E:IDMRUN.MOREDATA("irreopen")
There is data remaining to be read from the previous command.

A:IDMRUN.RECOMPILE("irreopen")
Must recompile from source.

E:IDMRUN.USENEXTCMD("irreopen")
You should be using *irnext*(3I) instead.

SEE ALSO

irclose(3I), iropen(3I), *System Programmer's Manual*

NAME

irreopen — reopen an IDMRUN structure

SYNOPSIS

```
#include <idmrun.h>

IDMRUN *irreopen(oldidmrun)
IDMRUN *oldidmrun;
```

DESCRIPTION

Irreopen creates a new IDMRUN structure much like *iropen*(3I). A database must be opened and a **begin transaction** executed on *oldidmrun* before issuing the reopen request. The new IDMRUN structure is a child of *oldidmrun* as described in section the *System Programmer's Manual*.

Reopened IDMRUN structures may be closed using *irclose*(3I) before executing an **abort transaction** or **end transaction**. All reopened IDMRUN structures must be closed before the parent is closed.

The IDMRUN structure returned by *irreopen* can be used just like an IDMRUN structure returned by *iropen*.

EXAMPLES

The following code fragment illustrates irreopen:

```
idmrun = iropen("db");

/* a reopen must be done within a transaction */
(void) iridl(idmrun, "begin transaction");
(void) irexec(idmrun);

/* reopen to do updates while retrieving */
child = irreopen(idmrun);

(void) iridl(idmrun, "range of x is x");
(void) iridl(idmrun, "retrieve (x.a)");
(void) irexec(idmrun);
(void) irbind(idmrun, 1, iINT4, 4, __ &i);

/* set up an append to do during the parent's retrieve */
(void) iridl(child, "append to x (a = %value)");

/* add new values to x.a */
while (RETSUCCESS(irfetch(idmrun)))
{
        if (i > 100)
        {
                /* substitute "i" into the child's append command */
                (void) irsubst(child, "value", iINT4, 4, __ &i);

                /* append i to relation "x" */
                (void) irexec(child);
        }
}
(void) irclose(child);
(void) iridl(idmrun, "end transaction");
(void) irexec(idmrun);
```

NAME

iropen — open an IDMRUN structure for use

SYNOPSIS

#include <idmrun.h>

IDMRUN *iropen(dbname)
char *dbname;

DESCRIPTION

Iropen creates a new IDMRUN structure. An IDMRUN structure must be opened before any commands can be executed by the IDM/RDBMS software on the database server. The server accessed is determined by the IDMDEV parameter (see ***getparam***(3I) and ***IftIdm***(4I)).

The indicated ***dbname*** is opened. If it is CHARNULL then no database is opened initially; an ***iridl***(3I) of an **open** command will perform this operation.

Every subroutine taking an IDMRUN structure as an argument takes it as the first argument.

The initial environment (see ***ienv***(5I)) is the default at the time of the open. This can be changed using ***irset***(3I).

Many IDMRUN structures can coexist.

EXAMPLE

The following example is included to also show how to get IDM DONE warning messages of the form W:IDM.***bitname***. See ***igetdone***(3I) and ***idone***(5I) for more details.

```
#include <idmenv.h>
#include <idmrun.h>

opendb()
{
        IDMRUN *idmrun;

        /* enable printing of some warnings from the idm */
        DefEnv→ie_donemask |= ID_DUP|ID_OVERFLOW|ID_DIVIDE;

        /* open system database */
        idmrun = iropen("system");

        /* more code */
}
```

SEE ALSO

intro(3I), getparam(3I), irclose(3I), irreopen(3I), iftidm(4I)

NAME

irnext — check for next executed statement

SYNOPSIS

#include <idmrun.h>

RETCODE irnext(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Irnext checks to see if there is another statement to be executed on the database server. It is used in conjunction with an IDL **execute** statement which executes a stored command that contains more than one executable IDL statement or when more than one executable IDL statement is processed with a single call to *iridl*(3I).

Irnext does not flush any return information. If this is desired *irflush*(3I) must be called. All data must be consumed before *irnext* can be called. The DONE struct information from the next command is read in if no data is returned.

RETURN VALUES

RS_NORM — The information for the next command is available. *Irdesc*(3I), *irbind*(3I), or *irfetch*(3I) should normally be the next routine called.

RW_DONECMDS — All commands have been processed.

RE_FAILURE — This command was not legal, probably because results are pending from the previous *irexec*.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irnext")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.MOREDATA("irnext")
The is data remaining to be read from the previous command.

E:IDMRUN.NOCMDS("irnext")
There were no commands to be executed.

E:IDMRUN.NOTEXEC("irnext")
Commands have been parsed, but not executed.

A:IDMRUN.RECOMPILE("irnext")
Must recompile from source.

E:IDMRUN.USEIREXEC("irnext")
Use irexec before calling irnext.

SEE ALSO

intro(3I), irdesc(3I), irexec(3I), irflush(3I), iridl(3I)

NAME

iridl — parse IDL statements

SYNOPSIS

#include <idmrun.h>

RETCODE iridl(idmrun, string)
IDMRUN *idmrun;
char *string;

DESCRIPTION

Iridl parses the IDL statements in *string* and associates the resulting query tree with *idmrun*.

Iridl accepts a sequence of IDL statements so that IDL statements can be processed in groups (see *irnext*(3I)).

The language accepted by *iridl* is described in *idlparse*(3I).

RETURN VALUES

RS_NORM The input has sucessfully been parsed and may now be executed using *irexec*(3I).

RE_FAILURE The input could not be parsed. An exception has been raised giving details.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("iridl")
Null IDMRUN or not an IDMRUN structure.

A:IDMRUN.RECOMPILE("iridl")
Must recompile from source.

E:IDMRUN.USENEXTCMD("iridl")
You should be using *irnext*(3I) instead.

Many others, described in *idlparse*(3I).

SEE ALSO

intro(3I), idlparse(3I), irexec(3I), irnext(3I), irxcmd(3I)

SEE ALSO

intro(3I), irset(3I), idone(5I), ienv(5I), itree(5I)

NAME

irget — get information from the IDMRUN structure

SYNOPSIS

```
#include <idmrun.h>

RETCODE irget(idmrun, addr, field, item)
IDMRUN *idmrun;
BYTE *addr;
int field;
int item;
```

DESCRIPTION

Irget extracts requested information from the *idmrun* structure into the address specified by *addr*. *Field* specifies what information to return. Compound fields include an *item* number.

The possible field values and types are:

IP_NUMSTMTS	Number of commands (int)
IP_CURSTMT	Number of current command (int)
IP_CMDTYP	Type of *item*'th command (int)
IP_DSTAT	Current done status (int)
IP_DINT	Current done integer (int)
IP_DCNT	Current done count (long)
IP_IFILE	Address of IFILE (IFILE *)
IP_TREE	A copy of the tree (ITREE *). This must be freed when done using *itfree*(3I).
IP_ENV	A pointer to the environment (IENV *)
IP_DMASK	The current done mask (short).
IP_DBIN	The current dbin (int).

RETURN VALUES

RS_NORM	The *irget* was successfully performed.
RE_FAILURE	It was not possible to satisfy the call.

EXAMPLES

irget(idmrun, _ _ &dstat, IP_DSTAT, 0);

Sets the variable *dstat* to the value of the done status field in the IDONE structure associated with the IDMRUN structure specified.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irget")

Null IDMRUN structure or not an IDMRUN structure.

E:IDMRUN.GETFLD(field)

Illegal field specifier.

E:IDMRUN.GETTREE(treenum)

Specified tree not available.

A:IDMRUN.RECOMPILE("irget")

Must recompile from source.

NAME

irflush — flush tuples for current command

SYNOPSIS

```
#include <idmrun.h>

RETCODE irflush(idmrun)
IDMRUN *idmrun;
```

DESCRIPTION

Irflush discards any tuple data for the current command. This is used when a retrieve loop is to be exited before all of the tuples have been fetched or when the user wishes to ignore any returned tuples and would simply like to see the status information for the command.

If the returned tuples are not desired they must be flushed before further processing can occur on the IDMRUN structure. For instance, one could send a **retrieve** command, then go into a loop calling *irfetch*(3I) for each iteration. If it became necessary to leave the loop without having fetched all of the tuples *irflush* must be called.

Ircancel(3I) flushes the current command as well as any commands waiting to be processed. The status information for all waiting commands will be lost. *Irflush* and *irnext*(3I) allow the user to view all of the status information without processing all of the return data.

RETURN VALUES

RS_NORM The values were successfully flushed.

RE_FAILURE Some failure occured during processing; an exception was raised explaining why.

EXAMPLES

The following code fragment illustrates early exit of a retrieve loop.

```
(void) iridl(idmrun, "retrieve (r.name)");
(void) irexec(idmrun);
(void) irbind(idmrun, 1, iCHAR, sizeof name, _ _ name);
while (RETSUCCESS(irfetch(idmrun)))
{
        /* code to handle each tuple goes here */
        if (flag)
        {
                (void) irflush(idmrun);
                break;
        }
}
```

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irflush")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.NOTEXEC("irflush")
Commands have been parsed, but not executed.

A:IDMRUN.RECOMPILE("irflush")
Must recompile from source.

E:IDMLIB.USENEXTCMD("irflush")
You should be using *irnext*(3I) instead.

SEE ALSO

intro(3I), ircancel(3I), irexec(3I), irfetch(3I), irnext(3I), iridl(3I)

```
                {
                        /* code to process the tuple goes here */
                }
        }
```

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irfetch")
Closed, NULL or bad IDMRUN structure.

W:IDMRUN.NEWTL
A new target list was read.

E:IDMRUN.NOCMDS("irfetch")
There were no commands to be executed.

E:IDMRUN.NOTEXEC("irfetch")
Commands have been parsed, but not executed.

A:IDMRUN.RECOMPILE("irfetch")
Must recompile from source.

BUGS

Single pseudo-commands which are parsed, executed, and fetched will return RW_TUPEND rather than RW_NOTUPS. A range and retrieve statement together will return RW_NOTUPS if the user attempts to apply *irfetch* after executing the range statement.

SEE ALSO

intro(3I), irbind(3I), ircancel(3I), irdesc(3I), irexec(3I), irflush(3I), iridl(3I)

NAME

irfetch — fetch a retrieved tuple

SYNOPSIS

#include <idmrun.h>

RETCODE irfetch(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Irfetch reads a tuple from the IDM/RDBMS software. Each target-list element is converted and stored into programming-language variables previously specified by calls to *irbind*(3I). *Irfetch* is used after an IDL or SQL statement which returns tuples (such as **retrieve** or **select**) has been parsed and executed. Information about the retrieved target-list elements can be found by calls to *irdesc*.

Any unbound target-list element values are discarded. If all of the target-list elements are unbound, tuples are read but not converted or stored.

If a retrieve loop is exited before all of the tuples have been fetched then *irflush*(3I) or *ircancel*(3I) must be called.

Irfetch or *irflush* must be called after a **retrieve** or **select** statement has been executed.

If a new target-list is read without an intervening DONE packet (i.e., an **audit** command is being processed), "W:IDMRUN.NEWTL" is raised. If the application program does not rebind using *irdesc*(3I) and *irbind*(3I), additional targets will be fetched but not bound.

RETURN VALUES

RS_NORM — A tuple has been successfully retrieved and bound to programming-language variables specified in *irbind*(3I).

RW_NOTUPS — The IDM command passed to *irexec* never returns data. Some IDM system commands, like **delete**, never return tuples.

Other commands, like **select** or **retrieve**, can return tuples. If no tuples satisfied the qualification, then RW_TUPEND will be the first value returned by *irfetch* (see below).

When RW_NOTUPS is returned, *irnext*(3I) will normally be the next routine called. If it is certain that there no other commands, another command can be parsed using *iridl*(3I) or *irsql*(3I).

RW_TUPEND — All available tuples have been retrieved. If this is the first value returned by *irfetch*, then no tuples were retrieved.

RE_FAILURE — It was not possible to execute this command, probably because it was called at the wrong time. An exception has been raised giving details.

EXAMPLE

A sample retrieve loop. No calls to *irdesc* are used since the storage type of the target-list element is known.

```
relnames()
{
        char name[15];

        (void) iridl(idmrun, "retrieve (r.name)");
        (void) irexec(idmrun);
        (void) irbind(idmrun, 1, iCHAR, sizeof name, _ _ name);
        while (RETSUCCESS(irfetch(idmrun)))
```

NAME

irexec — execute parsed IDL statements

SYNOPSIS

#include <idmrun.h>

RETCODE irexec(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Irexec sends the first IDL statement associated with the IDMRUN structure to the IDM/RDBMS software and retrieves any initial status information. The statements must have already been parsed by a call to *iridl*(3I).

If multiple executable IDL statements are parsed with one call to *iridl*, then *irnext*(3I) must be used to send the second and subsequent commands. Status information is associated with each statement (and possibly other return data as well). *Irnext* reads in the status information for the next executed command (or in the case of returned data warns the user of this fact).

If the user program attempts to perform another *irexec* on an IDMRUN structure without having processed all of the input data, an error will be returned and the *irexec* will be ignored. The user must either process all of the return data or call *ircancel*(3I) before another *irexec* can be performed. *Irflush*(3I) is also useful in processing return information quickly.

RETURN VALUES

RS_NORM — The command has been successfully sent. If the command returns data, the first tuple is available (via *irfetch*(3I)); otherwise, the return status is available. If there is any return data, it can be described using *irdesc*(3I) and/or bound to programming language variables using *irbind*(3I).

RE_FAILURE — It was not possible to execute this command. An exception has been raised describing the problem in detail. Most likely, *irexec* has been called at the wrong time or a required substitution (as described in *irsubst*(3I)) has not been performed.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irexec")
Null IDMRUN or not an IDMRUN structure.

E:IDMRUN.MOREDATA("irexec")
There is data remaining to be read from the previous command.

E:IDMRUN.NOCMDS("irexec")
There were no commands to be executed.

A:IDMRUN.RECOMPILE("irexec")
Must recompile from source.

E:IDMRUN.USENEXTCMD("irexec")
You should be using *irnext*(3I) instead.

SEE ALSO

intro(3I), ircancel(3I), irflush(3I), irnext(3I), iridl(3I)

NAME

irdump — dump an IDMRUN structure for debugging

SYNOPSIS

irdump(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Irdump prints the contents of *idmrun* onto *stdtrc* in a format suitable for gurus. The intent is to support debugging by very sophisticated users.

```
while (RETSUCCESS(irfetch(idmrun)))
{
        /* process the data */
}
```

DIAGNOSTICS

A return value of RW_TARGEND means target-list element number is too large by one; this allows a person to have a simple loop starting with target-list element number one and incrementing until the return code is equal to RW_TARGEND to get the descriptions of all the retrieved target-list elements. If a target-list element number is a value other than the number of domains in the target list plus one then the IDMRUN.TARGNUM exception is raised.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irdesc")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.NOCMDS("irdesc")
There were no commands to be executed.

E:IDMRUN.NOTEXEC("irdesc")
Commands have been parsed, but not executed.

A:IDMRUN.RECOMPILE("irdesc")
Must recompile from source.

E:IDMRUN.TARGNUM("irdesc", targnum)
An impossible target number was specified.

SEE ALSO

intro(3I), irbind(3I)

NAME

irdesc — get type and name information about a retrieved target-list element

SYNOPSIS

```
#include <idmrun.h>

RETCODE irdesc(idmrun, tl_num, type, length, name)
IDMRUN *idmrun;
int tl_num;
int *type;
int *length;
char **name;
```

DESCRIPTION

Irdesc returns type and name information about the retrieved target-list elements. It is typically called once for each target-list element. The target-list elements are specified by *tl_num*, numbered starting at one from left to right in the target list. Normal usage has a loop calling *irdesc* with *tl_num* incremented each time through the loop. When the return value of *irdesc* is RW_TARGEND then all of the target-list elements have been described.

The type of the target-list element as stored on the database server is placed at *type*. The length of the target-list element as stored on the database server is placed at *length*. The address of the name of the target-list element is placed at *name*, stored as a null-terminated string.

The names of the target-list elements correspond to the name given them in the target list (e.g., for the target-list element "cost = p.number * p.price", the name is "cost", and for the target-list element "p.number * p.price", there is no name). If no name is specified then the address of an empty string will be placed in *name*. The storage for the name is owned by the run-time system and cannot be modified by the user; it may change after the next call to the run-time system and must be copied if it is to be saved.

RETURN VALUES

RS_NORM The target was succesfully described. Normally the application program will bind the target to some program variable using *irbind*(3I).

RW_TARGEND There are no more targets left to describe.

RW_NOTUPS This command does not return any data.

RE_FAILURE This operation could not be satisfied; an exception has been raised giving the complete description.

EXAMPLES

The following loop illustrates how *irdesc* can be called to get information about each target-list element.

```
auto int type, length;
auto char *name;

i = 1;
while (RETSUCCESS(irdesc(idmrun, i, &type, &length, &name)))
{
        /*
        ** code to process the information goes here,
        ** e.g. save it in an array
        */
        i++;
}
```

NAME

irclose — close an IDMRUN structure

SYNOPSIS

#include <idmrun.h>

RETCODE irclose(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Irclose releases the specified IDMRUN structure. The IDMRUN structure can no longer be used. Any open database associated with the IDMRUN structure is closed.

The environment associated with the IDMRUN structure is *not* closed automatically.

RETURN VALUES

RS_NORM The IDMRUN structure was successfully closed.

RE_FAILURE The IDMRUN could not be closed because it was active. An exception is raised explaining the problem.

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irclose")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.MOREDATA("irclose")
Data remains to be read from the previous command.

A:IDMRUN.RECOMPILE("irclose")
Must recompile from source.

E:IDMRUN.USENEXTCMD("irclose")
You should be using *irnext*(3I) instead.

SEE ALSO

intro(3I), ircancel(3I), iropen(3I)

NAME

ircancel — cancel current operations on an IDMRUN structure

SYNOPSIS

#include <idmrun.h>

ircancel(idmrun)
IDMRUN *idmrun;

DESCRIPTION

Ircancel aborts any command currently being processed on the specified IDMRUN structure and flushes any pending return data and any commands waiting to be processed. No DONE packet results are available. *Ircancel* is intended for use whenever further processing of the current activity is to be ceased, for example, upon the receipt of a user interrupt.

If the user parsed several executable IDL statements with a single call to *iridl* and a cancel was performed while the first of the statements was being executed, the others would be discarded.

To simply discard the return data from the current command (e.g., when exiting a retrieve loop before processing all of the tuples) the program may call *irflush*(3I).

EXAMPLES

The following code fragment illustrates responding to user interrupts. This assumes that only one IDMRUN structure is used by operations affected by the interrupts.

```
#include <idmlib.h>
#include <idmrun.h>
#include <exc.h>

IDMRUN      *Idmrun;

toplevel()
{
        extern intr();
        Idmrun = iropen(CHARNULL);
        (void) exchandle("T:IDMLIB.ASYNC.INT", intr);
        for (;;)
        {
                ... etc ...
        }
}
intr()
{
        ircancel(Idmrun);
        return (1);
}
```

EXCEPTIONS

A:IDMRUN.BADIDMRUN("ircancel")
Closed, NULL or bad IDMRUN structure.

A:IDMRUN.RECOMPILE("ircancel")
Must recompile from source.

SEE ALSO

intro(3I), irexec(3I), irfetch(3I), irflush(3I), iridl(3I)

The length argument given in the call to *irbind* declares the total number of bytes available in the variable.

```
#include <idmlib.h>
#include <idmrun.h>

char relname[14];
long relid;

idmrun = iropen("demo");
(void) iridl(idmrun, "range of r is relation");
(void) iridl(idmrun, "retrieve (r.name, r.relid)");
(void) irexec(idmrun);
(void) irbind(idmrun, 1, iSTRING, 14, __ relname);
(void) irbind(idmrun, 2, iINT4, 4, __ &relid);
while (RETSUCCESS(irfetch(idmrun)))
        printf("rel=%s, relid=%ld\n", relname, relid);
```

Bcd numbers can be substituted with:

```
BCDNO b;

(void) irbind(idmrun, 1, iBCD, sizeof(BCDNO), __ &b);
```

EXCEPTIONS

A:IDMRUN.BADIDMRUN("irbind")
Closed, NULL or bad IDMRUN structure.

E:IDMRUN.BINDTYPE(usertype, idmtype)
It was not possible to bind the specified user variable to the IDM target list.

A:IDMRUN.RECOMPILE("irbind")
Must recompile from source.

E:IDMRUN.TARGNUM("irbind", targnum)
An impossible target number was specified.

E:IDMRUN.NOCMDS("irbind")
There were no commands to be executed.

E:IDMRUN.NOTEXEC("irbind")
Commands have been parsed, but not executed.

SEE ALSO

intro(3I), bcd(3I), irdesc(3I), irexec(3I), irfetch(3I), iridl(3I), typecnvt(3I)

NAME

irbind — bind program variables to retrieved target list elements

SYNOPSIS

#include <idmrun.h>

RETCODE irbind(idmrun, tl_num, type, length, address)
IDMRUN *idmrun;
int tl_num;
int type;
int length;
BYTE *address;

DESCRIPTION

Irbind associates data from domain *tl_num* retrieved from the database server with a program variable. After being bound, each *irfetch*(3I) call will convert the tuple data to the specified *type* and *length* and store it into the data area specified by *address*. *Irbind* may only be called after a statement that returns data (such as **retrieve**) has been executed.

The parameter *tl_num* specifies the index (numbered from one) into the target list after expansion of **.all** clauses. For example, consider the query:

retrieve (x.a, y.all, x.b)

If the relation indicated by "y" had three domains (e.g., "y.q", "y.r", and "y.s") then the following bindings would apply:

tl_num	*domain*
1	x.a
2	y.q
3	y.r
4	y.s
5	x.b

The types and names of each domain can be determined by *irdesc*(3I).

The *type*, *length*, and *address* of the program data area are specified using the final three parameters. The IDM system types iINT1, iINT2, iINT4, iFLT4, iFLT8, and iCHAR are supported in the obvious way. Types iBCD and iBCDFLT may be bound to data areas of type BCDNO; the routines described in *bcd*(3I) may be used to manipulate them. Type iBINARY is treated identically to type iCHAR except that it is padded with zero-bytes instead of spaces. In addition to the IDM system types, the host type iSTRING (null-terminated string, for C) is supported; *length* represents the maximum length of the string, including the trailing null byte.

For complete details of conversions, see *typecnvt*(3I). For a description of the difference between the RW_TARGEND return value and the IDMRUN.TARGNUM exception see the Diagnostics section in *irdesc*(3I).

RETURN VALUES

RS_NORM	The target was successfully bound.
RW_TARGEND	There are no more targets left.
RW_NOTUPS	This query does not return any tuples.
RE_FAILURE	It was not possible to do the bind; an exception has been raised giving more detail.

EXAMPLES

The following code fragment parses and executes an IDL **retrieve** statement and prints the return data.

NAME

iputtup — put a tuple from a target list to the database server

SYNOPSIS

#include <idmtlist.h>

RETCODE iputtup(itl, idm)
ITLIST *itl;
IFILE *idm;

DESCRIPTION

Iputtup writes a tuple from the specified target list *itl* to the specified *idm*. *Itl* is typically created by *igettl*(3I).

Iputtup sends an iTUPLE token followed by the tuple information from the target list. The type, length, and name information should have been sent previously by *iputtl*(3I).

Returns RS_NORM on success, RE_FAILURE on failure.

This routine is normally used for copy in/out.

SEE ALSO

igettl(3I), igettup(3I), iputtl(3I), itlist(5I)

EXCEPTIONS

W:IDMLIB.IDM.LONGNODE(type, len)
A node with a length field that was too long to transfer to the database server (that is, greater than 255) was truncated.

E:IDMLIB.IDM.ALL.NOREL(relname)
The specified relation name does not exist. This error occurs during expansion of an **.all** clause.

E:IDMLIB.IDM.BADORDER(ordervalue)
An attempt was made to order the query output by *ordervalue*. It was not found in the target list.

E:IDMLIB.IDM.ILLEGPARAM(cmd, paramvalue)
Illegal use of stored command parameter *paramvalue* in command *cmd*. Parameters to stored commands are only legal within **define**, **define program**, **exec**, and **exec program** commands.

E:IDMLIB.IDM.NOTCOMMAND(rootnode)
The tree passed was not a command tree; the type of the initial node was *rootnode*.

E:IDMLIB.IDM.SUB.NEEDVAL(subname)
The substitute node named *subname* has not had a value bound.

E:IDMLIB.IDM.SUB.TYPE(subname, symbol)
Illegal type *type* for substitution *subname* in some context in the tree, e.g, an integer used as a result attribute name.

E:IDMLIB.IDM.SUB.VAL(subname, value, min, max)
The *value* specified for substitution *subname* is out of range for the context in which it is used; *min* and *max* specify the acceptable range of values.

R:IDMLIB.IDM.GETHUNPW(database)
A host user name and password were required to open the specified *database*. This exception has a default handler associated with it to prompt for the name and password as required.

A:IDMLIB.IDM.TLOVFLOW(max)
The query had too many target list entries to send in one query; the maximum number of entries is *max*. Break up the query or dispose of domains you don't really need.

SEE ALSO

intro(3I), dba(3I), idlparse(3I), iesubst(3I), ifcontrol(3I), itxcmd(3I), iftidm(4I), ienv(5I), itree(5I)

NAME

iputtree — put a tree to the database server

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

RETCODE iputtree(tree, ifp, env)
ITREE *tree;
IFILE *ifp;
IENV *env;
```

DESCRIPTION

Iputtree translates a tree from the fully connected internal form produced and manipulated by the tree routines into the list form expected by Britton Lee's IDM/RDBMS software. This form is written to the file specified by *ifp*, which is normally a file of type *IftIdm*(4I).

As the tree is sent, all substitute nodes present in the tree will have values interpolated from the *env*ironment. If *env* is IENVNULL a default environment is used. Values must be supplied for all substitutions in the tree. If a value cannot be found in the current environment, the parent environments will be searched recursively until the value is found.

In some cases the tree can be modified, always resulting in the same semantics. In particular, **.all** nodes are converted to iATTRALL nodes.

The **open database** command is captured if *ifp* is a true IDM file (type *IftIdm*(4I) or *IftReopen*) and turned into an *ifcontrol*(3I) "opendb" call. This allows all special processing (IDM system user name/password processing and saving of the "dbin") to be centralized in one module. RW_PSEUDO is returned so that applications will not call *igetdone*(3I) inappropriately.

After a tree is sent, the program should call *igettl*(3I) to check for returned data. If a target list is returned, the data should be retrieved using *igettup*(3I). The iDONE token should then be read using *igetdone*(3I). If it specifies that more results are to be read, a new target list should be read.

RETURN VALUES

RW_PSEUDO	A pseudo-tree (e.g., a tree for a **range** statement) was ignored by *iputtree*.
RS_NORM	The tree was successfully sent to the file.
RE_FAILURE	The tree could not be sent; an exception is raised.

EXAMPLE

The following code provides a template for the generic case:

```
if (iputtree(tree, idmifp, IENVNULL) != RS_NORM)
    return;
do
{
    while ((itl = igettl(idmifp)) != ITLNULL)
    {
        while (igettup(idmifp, itl) == RS_NORM)
            process_tup(itl);
        itlfree(itl);
    }
    dp = igetdone(idmifp, IENVNULL);
} while (BITSET(ID_CONTINUE, dp→id_stat));
```

NAME

iputtl — write a target list to a file

SYNOPSIS

#include <idmtlist.h>

iputtl(itl, ifp)
ITLIST *itl;
IFILE *ifp;

DESCRIPTION

Iputtl writes the description of the target list *itl* to the specified *ifp*, consisting of the of names the target fields followed by one or more iFORMAT tokens and the the format information. See *igettl*(3I) for creating the target list.

Itl is a target list as described in *itlist*(5I). *Iputtl* puts the types and names, but not the values themselves. The values are put using *iputtup*(3I).

This routine is normally used for copy in/out.

SEE ALSO

igettl(3I), iputtup(3I), itree(5I), itlist(5I), *System Programmer's Manual*

```
#include <crackargv.h>
...
static char *Dbname, *Device;

static ARGLIST Args [] =
{
        /* argument template for database name */
  'd', FLAGSTRING, 4, "dbname", CHARNULL, _ _ &Dbname, CHARNULL, CHARNULL,
        /* argument template for IDM device name */
  'B', FLAGSTRING, 4, "idmdev", CHARNULL, _ _ &Device, CHARNULL, CHARNULL,
        /* other argument templates */
...
};

main (argc, argv)
   char **argv;
{
        /* declarations */
        ...

        /* initialize the runtime system */
        INITRIC (argv[0]);
        /* get command line arguments */
        crackargv (argv, Args);
        /* reset device or database names, if given on command line */
        RCDEVICE (Device);
        RCDBNAME (Dbname);
        /* begin processing */
        ...
}
```

SEE ALSO

ric(1I), rsc(1I), crackargv(3I), params(5i).

NAME

INITRC, INITRIC, INITRSC, RCDEVICE, RCDBNAME — macros for RIC and RSC precompiler source files

SYNOPSIS

INITRIC(progname);
INITRC(progname);
INITRSC(progname);
RCDEVICE(devicename);
RCDBNAME(databasename);
char *databasename, *devicename, *progname;

SYNOPSIS

INITRIC does the run-time setup for programs that have been run through the **ric** (IDL/C) precompiler. Its single argument is a character string that names the program, for use by the run-time system in writing error messages. For UNIX users, it is usually appropriate to say *INITRIC(argv[0])*.

INITRIC in turn invokes *INITIDMLIB*, so the user does not have to. Usually *INITRIC(progname)* will be the first executable statement in the program. It must be executed before the first executable IDL or SQL statement is executed.

INITRC is a synonym for *INITRIC*, maintained for historical reasons.

INITRSC does the exact same job in the exact same way for programs that have been run through the **rsc** (SQL/C) precompiler.

RCDEVICE is used for changing at run time the system device name of the device to use as the connection to the datbase server. Ordinarily, the device name is set at precompile time, either by giving an argument to the **–B** flag on the **ric** or **rsc** command, or (if this flag is omitted) from the value of the IDMDEV system parameter at precompile time. (Note that the value of the IDMDEV parameter at run time is never automatically used.) If neither of these values is appropriate, the device name can be given to *RCDEVICE* at run time. Often, this name will be obtained from the command line.

If ***devicename*** is neither the null pointer nor the null string, then it will be used as the name of the device to open.

RCDBNAME is used to reset the name of the database to access at run time. Ordinarily this value is inherited from the precompiler **–d** argument. If ***databasename*** is neither the null pointer nor the null string, then it will be used as the name of the database to open.

Either *RCDEVICE* or *RCDBNAME* may appear either before or after *INITRIC* or *INITRSC*, but they must appear before the first executable IDL or SQL statement if they are to have any effect.

All these macros are defined in the header file ***rcinclude.h***, which is automatically included by the precompiler in all files it produces.

EXAMPLE

The following canned lines are appropriate in most **ric** programs:

NAME

INITIDMLIB — initialize the IDM support library

SYNOPSIS

#include <idmlib.h>

INITIDMLIB(progname);

cc –i .... –lidmlib (on UNIX)

DESCRIPTION

In order to use the capabilities of IDMLIB, all source files must include the file *idmlib.h*. In addition, the IDM support library (–lidmlib on UNIX) must be loaded. Sixteen-bit machines require the use of separated instruction and data space (the –i flag on UNIX).

The main program *must* use *INITIDMLIB*(*progname*) as the first operation. This will initialize IDMLIB and set the name of this program for use by error messages, etc. This *must* be called from *main*().

SEE ALSO

intro(3I)

NAME

igettup — get a tuple from a database server into a target list

SYNOPSIS

#include <idmtlist.h>

RETCODE igettup(idm, itl)
IFILE *idm;
ITLIST *itl;

DESCRIPTION

Igettup reads a tuple from the specified *idm* into the target list *itl*. *Itl* is typically created by *igettl*(3I).

Igettup returns RS_NORM on success. If the input did not begin with a TUPLE token *igettup* ungets the errant token and returns RW_TUPEND.

SEE ALSO

igettl(3I), tupprint(3I), itlist(5I)

NAME

igettl, itlfree — read a target list from a database server

SYNOPSIS

```
#include <idmtlist.h>

ITLIST *igettl(idm)
IFILE *idm;

itlfree(itl);
ITLIST *itl;
```

DESCRIPTION

Igettl reads the description of a target list as described in *itlist*(5I) from the specified *idm*, consisting of the description of the types, lengths, and names of values to be returned by the IDM/RDBMS software. A target list is built, including sufficient space to hold the values when they are retrieved using *igettup*(3I).

The IDM SENDFORMAT option (option one) must be set for the target list to be built. If the SENDNAMES option (option two) is set then the names will be available in the iTLELM node.

If the next token in the input stream is not FORMAT or CHAR, then *igettl* pushes back the input token and returns ITNULL.

A query tree must be written (typically using *iputtree*(3I)) before *igettl* is called.

A target list must be explicitly freed using *itlfree* when it is no longer needed.

SEE ALSO

igettup(3I), iputtl(3I), iputtree(3I), itree(5I), itlist(5I)

NAME

igeteot, itapeload — get DONE blocks until end of IDM tape

SYNOPSIS

```
#include <idmdone.h>
#include <idmenv.h>

IDONE *igeteot(ifp, env)
IFILE *ifp;
IENV *env;

itapeload(optlist, ifp, env)
char *optlist;
IFILE *ifp;
IENV *env;
```

DESCRIPTION

Igeteot acts almost exactly like *igetdone*(3I) except that it understands intermediate DONE tokens asking the operator to mount another tape. Intermediate DONE tokens are handled automatically by communicating with the operator using the *operator*(3I) primitives.

Igeteot returns the IDONE value from the final volume.

Igeteot only prompts the operator for the second and subsequent tape. The initial tape should be requested using *itapeload* or a direct operator request.

A user response of the form *n*! where *n* is a digit, will turn off tape volume verification in the IDM/RDBMS software. This is useful when a partially completed dump needs another tape and there are no more initialized tapes available.

Itapeload should be called before the command tree that includes the tape option is sent to the IDM/RDBMS. *Optlist* describes the tape(s) to be mounted (see *itapeopts*(3I) for details). *Ifp* refers to the database server that will be used. *Env* refers to the associated environment, with the value IENVNULL mapped to the current default environment.

LIMITATIONS

Since *igeteot* must interact with *ifp*, this file must be of type *IftIdm*(4I) or *IftReopen*.

EXCEPTIONS

A:IDMLIB.IDM.TAPE.NOOPER

The job was aborted because there was no operator available to change the tape.

SEE ALSO

igetdone(3I), itapeopts(3I), operator(3I), iftidm(4I), idone(5I), *System Programmer's Manual*

NAME

igetdone — read ERROR, MEASURE, and DONE packets from the database server

SYNOPSIS

```
#include <idmdone.h>
#include <idmenv.h>

IDONE *igetdone(ifp, env);
IFILE *ifp;
IENV *env;
```

DESCRIPTION

Igetdone reads the database server connection pointed to by *ifp* for zero or more ERROR and/or MEASURE packets followed by a done packet. The arguments to the error and measure packets are formatted into exceptions which are raised.

The done packet is read and returned. The done status word is masked with the *ie_donemask* field in the *env*ironment and any bits remaining on cause an exception to be raised as described below. If *env* is IENVNULL a default environment is used.

The range table from *env* is used to select the range variable name for tokens of type iVAR that are returned in messages. If the variable name can not be determined the variable number is used instead.

WARNINGS

The done packet that is returned points to static memory. It must be copied if it is to be saved.

EXCEPTIONS

E:IDM.E*nnn*(*message-dependent arguments*)
: An IDM error packet has been read with an error value between 1 and 127 or 192 and 255.

A:IDM.E*nnn*(*message-dependent arguments*)
: An IDM error packet with error value between 128 and 191 inclusive has been read.

I:IDM.M*nnn*(*message-dependent arguments*)
: A MEASURE token was read. These tokens give performance information about the IDM.

W:IDM.*bitname*
: Here, *bitname* is INTERRUPT, OVERFLOW, DIVIDE, DUP, ROUND, UNDFLO, BADBCD, LOGOFF, or XABORT. The corresponding bit is set in the status word. This must be enabled by setting the corresponding bit in the done mask in *env*. Warning messages are not printed in the event that an IDM system error is returned in this call to *igetdone*.

I:IDM.INXACT
: The INXACT bit is set in the status word. This must be enabled by setting the corresponding bit in the done mask in *env*.

I:IDM.*bitname*(*value*)
: Here, *bitname* is COUNT or TIMER. The corresponding bit is set in the status word. *Value* is the value from the appropriate field of the done packet. The MINUTES bit is interpreted properly. This must be enabled by setting the corresponding bit in the done mask in *env*.

SEE ALSO

exc(3I), igeteot(3I), idone(5I), ienv(5I), *System Programmer's Manual*

NAME

ifwrite — write a block of memory

SYNOPSIS

ifwrite(ifp, ptr, cnt)
IFILE *ifp;
BYTE *ptr;
int cnt;

DESCRIPTION

Ifwrite appends at most *cnt* bytes of data beginning at *ptr* to the named output *ifp*. It returns the number of bytes actually written.

This routine is efficient on large transfers, doing the output directly from the user's buffer if possible.

Ifwrite is the only output primitive defined on files with the **rbp** (record-based presentation) attribute set. In this case *ifwrite* writes exactly one record; if *cnt* exceeds the maximum record length, a full record is written, the rest of the data is discarded, and an exception is raised.

When a stream-based file has a record-based presentation, short records (i.e., records where *cnt* is less than the **rs**) will be padded to the full **rs** using the **padchar** character (binary zero default).

EXCEPTIONS

E:IDMLIB.IO.WLR(filetype, filename)
An attempt was made to write a record that was too long.

A:IDMLIB.IO.WOROF(filetype, filename)
An attempt was made to write on a read-only file.

SEE ALSO

ifcontrol(3I), ifopen(3I), ifputc(3I), ifputs(3I), ifread(3I), printf(3I)

NAME

ifungetc — put a character back into input buffer

SYNOPSIS

ifungetc(c, ifp)
int c;
IFILE *ifp;

DESCRIPTION

Ifungetc pushes the byte *c* back on an input *ifp*. That character will be returned by the next *IFGETC* call on that *ifp*. *Ifungetc* returns *c*.

Attempts to push EOF are ignored.

LIMITATIONS

One character of pushback is guaranteed provided something has been read from the *ifp* and the *ifp* is actually buffered.

This primitive is only defined on files with stream-based presentations.

EXCEPTIONS

E:IDMLIB.IO.IFUNGETC(filetype, filename)
If there is no room to hold the pushback character.

E:IDMLIB.IO.UOWOF(filetype, filename)
You have tried to invoke *ifungetc* on a write-only file.

SEE ALSO

ifgetc(3I)

A:IDMLIB.IO.CRACK.BADTYPE(spec)
Illegal type is spec <spec>.

IMPLEMENTATION NOTES

The percent sign can be changed on a per-system basis. If necessary this routine could be completely rewritten to provide a different syntax.

SEE ALSO

intro(1I), ifopen(3I), ifthfile(4I), iftifile(4I), iftltape(4I)

NAME

ifscrack, ifstype — crack file specification string

SYNOPSIS

```
#include <ifscrack.h>

char *ifscrack(spec, ptype, fnbuf, fnlen)
char *spec;
int *ptype;
char fnbuf[];
int fnlen;

char *ifstype(type, filename)
int type;
char *filename;
```

DESCRIPTION

Ifscrack takes an IDM file *spec* and breaks it up into a file name, a file type, and a set of params. The syntax of a file spec is:

[filename][%params]

where *params* is a comma-separated list of parameters in *pextract*(3I) format. *Params* may include a type, which must be one of:

hfile	host file
ifile	IDM file
htape	host (ANSI) tape
multi	multi-diskette file
itape	IDM tape

Hfile is the default. For example:

myfile	host file
myfile%hfile	same
myfile%ifile	IDM file
myfile%htape,bs(4096)	host tape, block size = 4096
%itape	IDM tape

The file name is stored into the *fnbuf* buffer. At most *fnlen* characters, including a terminating null byte, will be stored. An exception is raised if the filename in the spec is too long.

A pointer to the *params* is returned. If no parameters exist, the zero length string is returned.

The type field is decoded and stored indirectly into **ptype* as a bit mask. The IFS_TAPE bit is set if tape is used, and IFS_IDM is set if an IDM file or tape is specified. The constants IFS_HFILE, IFS_HTAPE, IFS_IFILE, and IFS_ITAPE represent the valid combinations.

Htape and *multi* are equivalent (both returning IFS_HTAPE), with the latter intended for use in the PC environment.

The syntax accepted by *ifscrack* is intended to be used for external specification of files, e.g., file names specified by users on command lines.

Ifstype takes an encoded *type* field and a *filename* and returns a string suitable for printing. For example, it might produce "host file "xyzzy"" for a *type* of IFS_HFILE and a *filename* of "xyzzy".

EXCEPTIONS

E:IDMLIB.IO.CRACK.NAMETOOLONG(name, maxlen)

The name specified was too long.

NAME

ifread — read a block of memory

SYNOPSIS

```
ifread(ifp, ptr, cnt)
IFILE *ifp;
BYTE *ptr;
int cnt;
```

DESCRIPTION

Ifread reads *cnt* bytes of data from the named input *ifp* into a block beginning at *ptr*. It returns the number of bytes actually read.

This routine is efficient on large transfers, doing the I/O directly into the user's buffer if possible.

Ifread is the only input primitive defined on files with the **rbp** (record-based presentation) attribute set. In this case *ifread* reads exactly one record; if the *cnt* is smaller than the length of the next record *cnt* bytes are read, the remainder of the record is discarded, and W:IDMLIB.IO.SHORTREAD is raised.

EXCEPTIONS

A:IDMLIB.IO.ROWOF(filetype, filename)
Read on write-only file.

W:IDMLIB.IO.SHORTREAD(filetype, filename)
Data was discarded.

SEE ALSO

ifcontrol(3I), ifopen(3I), ifgetc(3I), ifgets(3I), ifwrite(3I)

DIAGNOSTICS

Ifread returns zero upon end of file.

NAME

ifputs — put a string on a text file

SYNOPSIS

ifputs(s, ifp)
char *s;
IFILE *ifp;

DESCRIPTION

Ifputs copies the null-terminated string *s* to the named output *ifp*. It does not copy the terminating null character.

Since *ifputs* is built on top of *ifputc*(3I), all *IFPUTC* restrictions apply to *ifputs*.

SEE ALSO

iferror(3I), ifgets(3I), ifopen(3I), ifputc(3I), ifwrite(3I), printf(3I)

BUGS

Ifgets and *ifputs* are not inverse operations, since *ifgets* strips the newline but *ifputs* does not add one.

NAME

IFPUTC, ifputc — put a byte to a file

SYNOPSIS

int IFPUTC(c, ifp)
BYTE c;
IFILE *ifp;

int ifputc(c, ifp)
BYTE c;
IFILE *ifp;

DESCRIPTION

IFPUTC and ***ifputc*** append the character *c* to the named output ***ifp***. The return value is not defined. I/O errors are reported via ***excraise***(3I).

IFPUTC is functionally the same as ***ifputc***, but is implemented as a macro for efficiency.

The standard files ***stdout*** and ***stderr*** are normally line buffered. When an output file is line buffered, information appears on the destination file or terminal as soon as one line is written; when it is fully buffered, many characters are saved up and written as a block. ***Ifflush***(3I) may be used to force the block out early. However, on record-based files ***ifflush*** will start a new record.

LIMITATIONS

Some systems may not allow completely unbuffered output, i.e., a newline may be required to force output. To print a prompt and read input, use ***getprompt***.

EXCEPTIONS

A:IDMLIB.IO.WOROF(filetype, filename)
Write on read-only file.

SEE ALSO

getprompt(3I), ifflush(3I), ifgetc(3I), ifopen(3I), ifputs(3I), ifwrite(3I), printf(3I)

WARNINGS

Because *IFPUTC* is implemented as a macro, an ***ifp*** argument with side effects functions improperly. In particular "IFPUTC(c, *f++);" doesn't work sensibly.

Errors can occur long after the call to ***ifputc*** or *IFPUTC*.

Ifputc is undefined on files with a record-based presentation; ***ifwrite***(3I) must be used instead.

W:IDMLIB.IO.RSIZE(type, name, user, file)
The user has specified an explicit *rs* for the specified file that does not match the information associated with the file. The user parameter overrides.

W:IDMLIB.IO.BRSIZE(type, name, bs, rs)
The *rs* is larger than the *bs* for this file. The *bs* will be increased to accommodate the *rs*.

Others as described in section 4I.

SEE ALSO

ifclose(3I), ifcontrol(3I), ifgetc(3I), ifputc(3I), ifread(3I), ifwrite(3I), section 4I for descriptions of the various file types.

mode(M) This may be **r** for read, **w** for write, **a** for append, **u** for update. The file is created if it does not already exist in **w**, **a**, and **u** modes.

padchar(C)† Set the pad character. Used by some file types to **pad out** short records. Defaults to binary zero if not overridden by the file type. See below for details.

rbp(B)† Use a record-based presentation.

rs(RS) The logical record size for this file. If not specified, the block size is used.

trace(B)† Enables detailed tracing on this file. This normally includes showing all traffic on the file if trace flag IOTRAFFIC.6 is set.

Other parameters may be defined by the file-type module; see the individual descriptions for these parameters.

The basic primitives that may be applied to an *ifp* are *IFGETC* (get character), *IFPUTC* (put character), *ifcontrol* (perform control operations), *ifflush* (force output), and *ifclose* (close file). Other operations are built from these primitives.

Update mode has highly restricted semantics. In general, a file must have the **reset** or **rewrite** *ifcontrol*(3I) call applied to it before switching from read to write operations or vice versa. Individual file types may have less restrictive semantics; see section 4I for details. Any usage other than those explicitly defined will give undefined results (probably without an error message).

The **padchar** is used when simulating records on physically stream-based files. Short logical records will be padded with this character to the full record length on output. It will *not* be stripped on input. The **padchar** defaults to binary zero on most file types.

IMPLEMENTATION NOTES

Parameters that are not recognized by an implementation should be ignored rather than diagnosed. If there are parameters that will cause important functional differences, they should be parsed and diagnosed specially.

EXAMPLES

```
extern IFTYPE IftHFile, IftIdm, IftIFile;

hifp = ifopen(InputFile, &IftHFile, "mode(r),vms(dsp=prt)", IFNULL);
iifp = ifopen("system", &IftIdm, "trace", IFNULL);
fifp = ifopen("myfile", &IftIFile, "bs(8192)", iifp);
```

EXCEPTIONS

A:IDMLIB.IO.BADMODE(filetype, filename, mode)
An I/O mode was specified that was incompatible with the file type.

A:IDMLIB.IO.CANTOPEN(filetype, filename, mode, why)
The file cannot be opened.

A:IDMLIB.IO.NOBASE(filetype, filename)
The specified file was not supplied with a required base file.

A:IDMLIB.IO.NEEDNAME(filetype)
The specified file type was not supplied with a required file name.

A:IDMLIB.IO.CANTNAME(filetype, filename)
This file type does not accept names. For example, strings do not have names.

A:IDMLIB.IO.NOMODE(filetype, filename)
The specified file did not have a required I/O mode supplied.

W:IDMLIB.IO.BSIZE(type, name, user, file)
The user has specified an explicit *bs* for the specified file that does not match the information associated with the file. The user parameter overrides.

NAME

ifopen — open a file

SYNOPSIS

IFILE *ifopen(filename, type, params, baseifp)
char *filename;
IFTYPE *type;
char *params;
IFILE *baseifp;

DESCRIPTION

Ifopen opens the file named by *filename* and associates a file pointer with it. *Ifopen* returns a pointer to be used to identify the file in subsequent operations.

Type specifies the file type. A few of the important types are *IftHFile* (host file), *IftIdm* (IDM channel), and *IftString* (in-core string). See section 4I for details and other file types.

Certain file types do not implement a true file. Rather, they act somewhat like a UNIX filter, reading from one file, performing a transformation, and writing to another file. For example, the *IftIFile* (IDM file) type module does not actually open any files on the host. However, reads or writes on an *IftIFile* are transformed into I/O on an underlying file of type *IftIdm* (IDM channel). This allows the *IftIFile* type to emulate a stream on top of the database server. *Baseifp* is used to pass in this underlying file.

Params is a character string describing characteristics of the file. It consists of a series of comma-separated file parameters, each of the form *name*(*value*). Individual parameters may be in any order. Parameters that are not recognized by an implementation should be silently ignored.

In most cases binary *params* (that is, parameters that are either on or off) are asserted by specifying "*name*" or "*name*(1)"; they can be explicitly deasserted by specifying "*name*(0)" (e.g., to override the default setting on a file type).

STANDARD PARAMS

Params that apply to all file types are listed below. Params marked with † are identical to the corresponding *ifcontrol*(3I) calls. Defaults listed are for the usual case, although a file type may specify a different default. That is, explicit *ifopen* parameters are preferred, followed by the file-type default, followed by the default listed here. See the appropriate writeup in section 4I for exact information regarding defaults.

autoclose(B) Close this file automatically on exit. This is normally used only in the default parameters for a file type.

bs(BS) The value is the block size for this file. The block size must be at least as large as the largest record in the file. On output, this defines the physical block size. This is advice only on input, i.e., the file type can (and should) override this value if it can be determined from the file itself.

disp(D)† Set the file disposition, i.e., what should happen to the file when it is closed. Defined values are **delete** for a file that should be deleted when closed and **keep** for files that should be kept when closed (default).

global Allocate the resources for this file in a global arena. Files without this attribute may be automatically deallocated by a *freempool* (see *xalloc*(3I)) call. In general, always specify this attribute if the file pointer is stored in a global variable; never specify it if the file pointer is a local variable.

linebuffer(B)† Enables line buffering, that is, an automatic flush of each newline-terminated line. This operation should normally be reserved to the file type, since it can affect the normal functioning of the file.

NAME

ifgets — get a line from a text file

SYNOPSIS

char *ifgets(s, n, ifp)
char *s;
int n;
IFILE *ifp;

DESCRIPTION

Ifgets reads *n*–1 characters, or up to a newline character, whichever comes first, from the *ifp* into the string *s*. The last character read into *s* is followed by a null character. *Ifgets* returns its first argument. The trailing newline is deleted. If the input line is longer than *n*–1 bytes, the remainder of the line is thrown away.

Since *ifgets* is built on top of *ifgetc*(3I) all restrictions on *ifgetc* also apply here.

SEE ALSO

iferror(3I), ifgetc(3I), ifputs(3I), ifread(3I)

DIAGNOSTICS

Ifgets returns the constant pointer CHARNULL upon end of file.

NAME

IFGETC, ifgetc — get a byte from a file

SYNOPSIS

int IFGETC(ifp)
IFILE *ifp;

int ifgetc(ifp)
IFILE *ifp;

DESCRIPTION

IFGETC and *ifgetc* return the next byte from the named input *ifp*, and the integer constant EOF at end of file.

IFGETC is functionally the same as *ifgetc*, but is implemented as a macro for efficiency.

EXCEPTIONS

A:IDMLIB.IO.ROWOF(filetype, filename)
Read on write-only file.

WARNINGS

Ifgetc and *IFGETC* are undefined on files having a record-based presentation; *ifread*(3I) must be used instead.

Because it is implemented as macro, *IFGETC* treats an *ifp* argument with side effects incorrectly. In particular, "IFGETC(*f++);" doesn't work sensibly.

SEE ALSO

ifgets(3I), ifopen(3I), ifputc(3I), ifread(3I), ifungetc(3I)

NAME

ifflush — flush a file

SYNOPSIS

int ifflush(ifp)
IFILE *ifp;

DESCRIPTION

Ifflush causes any buffered data for the named output *ifp* to be written to that file. The EOF bit is cleared and the file remains open. *Ifflush* returns zero on success and negative on failure.

This call may be safely applied to a read-only file. This will cause the EOF bit to be cleared.

If *ifp* specifies a file with stream-based presentation of a physically record-based file, *ifflush* will terminate a record. *Ifflush* is undefined on files with record-based presentations.

SEE ALSO

ifcontrol(3I), ifopen(3I)

NAME

IFERROR, ifeof, IFEOR — file status inquiries

SYNOPSIS

RETCODE IFERROR(ifp)
IFILE *ifp;

BOOL ifeof(ifp)
IFILE *ifp;

BOOL IFEOR(ifp)
IFILE *ifp;

DESCRIPTION

IFERROR returns the error code of the most recent error that has occurred reading or writing the named *ifp*. Unless cleared by the *clrerr* control, the error indication lasts until the file is closed. The value RS_NORM is returned if no error has occurred.

Ifeof returns TRUE after end-of-file has been read on the named input *ifp*, otherwise FALSE.

IFEOR returns TRUE at end-of-record. It is only meaningful on input. *IFEOR* is also TRUE at the beginning of the file. On a stream-based-file, it means that the buffer is empty.

SEE ALSO

geterr(3I), ifcontrol(3I), ifopen(3I)

NAME

ifdump — dump an IDMLIB file pointer for debugging

SYNOPSIS

ifdump(ifp)
IFILE *ifp;

DESCRIPTION

Ifdump prints the contents of *ifp* onto *stdtrc* in a format suitable for gurus. The intent is to support debugging by very sophisticated users.

should be taken using numeric values since they are unlikely to be portable between ASCII and EBCDIC environments. Use of this parameter is discretionary to the file type.

_rbf(B)* Set or clear record-based-file mode. Since this is an attribute of the physical file, use is strictly limited to file-type modules.

rbp(B)† Set or clear record-based-presentation mode. When set, *ifread*(3I) and *ifwrite* are the only legal interfaces.

reset Resets the internal pointers to the beginning of the buffer, clears the EOF bit, and (where possible) rewinds the device.

rewrite Reset the file (as above) and destroy any existing contents (i.e., truncate the file to zero length). The file must be writable for this to succeed.

trace(B)† Set or clear trace mode on this file.

EXCEPTIONS

E:IDMLIB.IO.REWRITE(filetype, filename, why)
The file could not be rewritten.

SEE ALSO

ifopen(3I), ifflush(3I), ifclose(3I), iodefs(5I), section 4I for specific controls for different file types.

NAME

ifcontrol — perform control operations on files

SYNOPSIS

int ifcontrol(ifp, params, args)
IFILE *ifp;
char *params;
BYTE *args;

DESCRIPTION

Ifcontrol performs control operations on the named file. These can set or retrieve parameters or perform special operations on the file.

The syntax of the *params* argument is identical to the *ifopen*(3I) call. The semantics are defined by the file type. The *args* parameter is used by some control operations; the semantics vary. Controls which return a value via the *args* parameter (*getbs*, *getflags*, etc.) of a type other than **BYTE *** are noted in their description.

The return value depends on the control operation performed. Normally zero means success, negative means failure.

STANDARD CONTROLS

The following controls are implemented on all files where they are possible, as detailed in section 4I. Controls marked with † are identical to the corresponding *ifopen*(3I) params. Controls marked with * should be implemented by the file-type module if possible, but should *never* be used by application programs; they typically implement some internal feature.

cancel — Resets the internal pointers to the beginning of the buffer and sets the EOF bit. Used to cancel input from a device and fool any other code into thinking that the transfer is complete.

clrerr — Clear the current error indication on the file.

_delete* — Remove the underlying file. This is *not* intended for use by the end user. It is guaranteed that the file will be closed. This call is issued by *ifclose*(3I).

_dio(B)* — Set or clear direct I/O capability. Setting this mode is a dangerous operation on some file types; use is reserved to file-type modules.

disp(D)† — The file disposition, i.e., what will be done with the file when the file is closed. Currently defined values are **keep** and **delete** to keep and delete the file respectively.

flushblock — Flush the file using *ifflush*(3I), then force any blocked data to the physical media. This is the only way to guarantee that output to a blocked file is actually on the media.

getbs — Return the buffer size of this file into the integer pointed to by *args*.

getflags — Return the flag bits for this file into the integer pointed to by *args*. These bits are defined in <*idmiodefs.h*>, described in *iodefs*(5I).

getrs — Return the record size of this file into the integer pointed to by *args*.

linebuffer(B)†* — Enables line buffering, that is, an automatic flush of each newline-terminated line. This operation should normally be reserved to the file type, since it can affect the normal functioning of the file.

padchar(C)† — Set the pad character to be used to pad the record out for fixed-length records. This is used when simulating records on streams or by file types that want to provide settable padding. By default this is a zero byte. This may be a single character used directly, or a numeric character value converted by *atoi*. Care

NAME

ifclose — close a file

SYNOPSIS

ifclose(ifp)
IFILE *ifp;

DESCRIPTION

Ifclose causes any buffers for the named *ifp* to be emptied using *ifflush*(3I), and the file to be closed. Buffers allocated by the standard input/output system are freed. If the file disposition is **delete** the file is removed.

Ifclose is invoked automatically by *freempool* (see *xalloc*(3I)) if the memory pool being freed is bound by *ifopen*(3I) to the file.

SEE ALSO

ifcontrol(3I), ifflush(3I), ifopen(3I), xalloc(3I).

IMPLEMENTATION NOTES

The **delete** file disposition is implemented by issuing the **_delete** *ifcontrol*(3I) call. (This module is environment-independent.)

meaning.

All SUBSTITUTE nodes must have a value associated before *iputtree*(3I) may be called. However, values can be reassigned and the query rerun without reparsing the query, and without reassigning all SUBSTITUTE nodes. See *irsubst*(3I) for an example.

Iesubst returns RS_NORM if the substitution was successful; RE_FAILURE if the value was not legal.

EXAMPLES

Bcd numbers can be substituted with:

```
(void) iesubst(env, "xx", b→bcd_type, b→bcd_len, _ _ b→bcd_str);
```

EXCEPTIONS

E:IDMLIB.IDM.SUB.TYPE(subname, symbol)

Illegal type for *value*. This only occurs if the specified type can *never* be an acceptable substitution type, i.e., if it is not a constant. This exception can also be raised by *iputtree*(3I) if the type is unacceptable in a particular context, e.g., a string must be specified in a context where only a domain name may occur.

SEE ALSO

idlparse(3I), iputtree(3I), irsubst(3I), ienv(5I)

NAME

iesubst — perform substitutions in environments

SYNOPSIS

#include <idmenv.h>

RETCODE iesubst(env, name, type, length, value)
IENV *env;
char *name;
int type;
int length;
BYTE *value;

DESCRIPTION

Iesubst associates the *value* of specified *type* and *length* with the *name* in the *environment*. If *env* is IENVNULL a default environment is used. The *type* must specify a constant.

Substitutions are a way of putting placeholders into an ITREE using the *%name* syntax in *idlparse*(3I) or *sqlparse*(3I). Values may be substituted later into the tree without reparsing. Substitutions may occur

- Any place where an *object-name* might appear.
- Any place where an *expression* might appear.
- As the first or second parameter to a *substring* or *bcdfixed* function, or as the first parameter to a [*fixed*]*bcd*, [*fixed*]*bcdfloat*, or [*fixed*]*string* function.
- As an attribute name on the left-hand side of an equal sign; the substitution must be a character type.
- As the **with** part of an **associate** command.

When a tree is sent using *iputtree*(3I), all substitute nodes are replaced with associated values from the *environment*. For example, the calls

```
tree = idlparse("retrieve (r.relid) where r.name = %rel", env);
stat = iesubst(env, "rel", iSTRING, -1, "parts");
stat = iputtree(tree, idmifp, env);
stat = iesubst(env, "rel", iPCHAR, 2, "p*");
stat = iputtree(tree, idmifp, env);
```

are equivalent to

```
tree = idlparse("retrieve (r.relid) where r.name = \"parts\"", env);
stat = iputtree(tree, idmifp, env);
tree = idlparse("retrieve (r.relid) where r.name = \"p*\"", env);
stat = iputtree(tree, idmifp, env);
```

"Call by value" semantics apply to *value*; changes to the memory that *value* points to will not affect the value of the substitution.

If the *type* is iSTRING the length is ignored and the *strlen* (see *string*(3I)) is taken instead. *Name* is a null-terminated string.

When substituting BCD numbers, pass the *bcd_str* data area of the BCDNO as the *value*. *Bcd_len* should be passed in as *length* to ensure that the correct number of *bcd_str* bytes are copied.

When passing character or string data, the iPCHAR type indicates that the datum may contain pattern matching characters. Pattern matching may not be used in target lists; doing so will return IDM error E39. Types iCHAR and iSTRING indicate that all characters should be interpreted literally. Meta-characters (e.g., "*", "?" in IDL, "%", "_" in SQL) have no magic

NAME

iesetopt, ieclropt — set or clear options

SYNOPSIS

#include <idmenv.h>

RETCODE iesetopt(env, option)
IENV *env;
int option;

RETCODE ieclropt(env, option)
IENV *env;
int option;

SYNOPSIS

Iesetopt sets the specified IDM system *option* in the *env*ironment. This option will be associated with all further commands sent with the *env*ironment using *iputtree*(3I). Options are linked into the tree when it is sent rather than at parse time when the tree is built. This allows programs such as *ric*(1I) to set options at runtime rather than at precompile time.

Returns RS_NORM if the option is successfully set, RE_FAILURE if the option is illegal, or RW_IGNORED if the option is already set.

If *env* is IENVNULL the default environment is used.

The IDM tape option is rejected in *iesetopt*. To use IDM tape, use one of the *dba*(3I) routines, *itcopy*(3I), or set the option using *itaddopts*(3I).

The optNAMES and optFORMAT options may not be changed. These are set by default, and any attempt to alter their values will result in an exception.

Ieclropt functions identically, except that the option is cleared. If the option was not previously set, *ieclropt* raises a warning exception.

EXCEPTIONS

W:IDMLIB.IDM.OPT.SET(option)
The specified option is already set.

W:IDMLIB.IDM.OPT.NOTSET(option)
The specified option is already clear.

E:IDMLIB.IDM.OPT.ILLEGAL(option)
The specified option number is unknown.

E:IDMLIB.IDM.OPT.ILLEGAL.TYPE(option)
The specified option may not be set or unset.

W:IDMLIB.IDM.OPT.TAPE(option)
Cannot set IDM tape option in iesetopt.

SEE ALSO

dba(3I), idlparse(3I), itaddopts(3I), itcopy(3I), sqlparse(3I), ienv(5I)

NAME

ieopen, ieclose — open and close IENV's (IDM environments)

SYNOPSIS

```
#include <idmenv.h>

IENV *ieopen(parent, params)
IENV *parent;
char *params;

ieclose(env)
IENV *env;
```

DESCRIPTION

Ieopen opens (creates) a new environment with the specified *parent*. The range and substitute tables are initially empty. The done mask is copied from the parent unless modified by *params*.

Properties of the environment may be set or modified using a comma-separated list of *params* (see below).

Ieclose closes (destroys) the specified environment. *Env* must not be used again. If *env* is the default environment (DefEnv) the default is replaced by its parent. It is a grave error to destroy the root environment (the environment with no parent).

These routines take IENVNULL as values for *env* to indicated the default environment.

PARAMS

donemask(D)	Set the done mask literally to the list of done bits named in *D* (see *idone*(5I)).
setdonemask(D)	Set the done mask to the *parent*'s done mask plus the named bits.
clrdonemask(D)	Set the done mask to the *parent*'s done mask minus the named bits.
foldcase(B)	If *B* is '0', turn foldcase mode off. If *B* is '1' or omitted, turn foldcase mode on. If **foldcase** is not specified, set foldcase to the value of the FOLDCASE parameter (see *getparam*(3I)). If foldcase mode is set, routines taking an *env* parameter will fold uppercase letters in character string arguments to lowercase.
mapcc(B)	Set the mode where output control characters are mapped to ITG_BLOTCH *iftterm*(4I) in *tupprint*(3I) if *B* is missing or '1'; print control characters if *B* is '0'. Defaults from the MAPCC parameter at *ieopen*(3I) time.

EXAMPLE

Environments can be stacked easily using:

```
DefEnv = ieopen(IENVNULL, CHARNULL);
/* use new environment */
ieclose(IENVNULL);
```

SEE ALSO

idlparse(3I), iecontrol(3I), iesubst(3I), iputtree(3I), sqlparse(3I), idone(5I), ienv(5I), params(5I)

NAME

iecontrol — perform control operations on environments

SYNOPSIS

```
#include <idmenv.h>

iecontrol(env, params, args)
IENV *env;
char *params;
BYTE *args;
```

DESCRIPTION

Iecontrol adjusts fields in the *env*ironment. If *env* is IENVNULL the default environment is used.

Params describes what is to be done. *Args* are used by some control calls as specified by *params*.

CONTROLS

donemask(D) Set the done mask to the list of done bits named (see *idone*(5I)). Any bits not explicitly named are cleared. If *D* is missing, *args* contains the literal bits to use for the new value.

clrdonemask(D) Clear the named bits in the done mask. If *D* is missing, *args* contains the literal bits to use for the new value.

setdonemask(D) Set the named bits in the done mask. If *D* is missing, *args* contains the literal bits to use for the new value.

foldcase(B) Set the foldcase mode if *B* is missing or '1'; clear the foldcase mode if *B* is '0'. This defaults from the FOLDCASE parameter at *iecopen*(3I) time.

mapcc(B) If *B* is missing or 1, map control characters to ITG_BLOTCH (see *iftterm*(4I)) in *tupprint*(3I). If *B* is '0', do not change control characters. Defaults from the MAPCC parameter at *iecopen*(3I) time.

SEE ALSO

idone(5I)

NAME

idmsymbol, idmwsymbol — return name of IDM symbol or WITH node

SYNOPSIS

```
char *idmsymbol(sym)
int sym;

char *idmwsymbol(wsym)
int wsym;
```

DESCRIPTION

Idmsymbol returns the name of the symbol passed as the argument. If the symbol is not recognized, a printable version of the numeric value is returned.

Idmwsymbol returns the name of the WITH node symbol in a manner analagous to *idmsymbol*.

WARNINGS

The return value of either routine may be a pointer to a static data area that will be destroyed on the next call.

EXAMPLES

idmsymbol(0342) → "DBOPEN"
idmsymbol(0204) → ">="
idmsymbol(0543) → "(token 0x163)"

idmwsymbol(5) → "demand"

SEE ALSO

System Programmer's Manual (SPM) for the tokens and their semantics.

Syntax errors should try to give you a pointer into the input line, rather than just a line number, so that user-friendly error messages can be generated.

SEE ALSO

dba(3I), ieopen(3I), iesubst(3I), iputtree(3I), itfree(3I), itxcmd(3I), ienv(5I), itree(5I)

E:IDMLIB.IDM.MAPC.ESCAPE(string)
Illegal pattern-matching string.

E:IDMLIB.IDM.NOTINT(type)
Constant in %N substitution name was not an integer.

E:IDMLIB.IDM.NOTFUNC(name)
The specified name was used in a context that would imply that it must be a function or aggregate name, but it cannot be recognized.

E:IDMLIB.IDM.NUMARGS.TOOMANY/TOOFEW(function, nargs)
The wrong number of arguments were given to the specified function. The correct number of arguments is given.

E:IDMLIB.IDM.OPT.ILLEGAL(option)
An attempt was made to set an impossible or unknown option.

W:IDMLIB.IDM.OPT.NOTSET(option)
An attempt was made to **unset** an option that was not set.

W:IDMLIB.IDM.OPT.SET(option)
An attempt was made to **set** an option that was already set.

E:IDMLIB.IDM.OPT.TOOMANY
Too many options have been set.

E:IDMLIB.IDM.RANGE.NOTDECL(rvar)
The specified range variable was not declared in a **range** statement.

E:IDMLIB.IDM.RANGE.BADNO(rangenum)
Internal error — used an illegal range variable number.

E:IDMLIB.IDM.RANGE.BADOPT(optname)
The specified range option is not valid.

W:IDMLIB.IDM.RANGE.GRAB(newrv, oldrvar, oldreln)
A range table entry has been changed.

E:IDMLIB.IDM.RANGE.TOOMANY(nvar, maxvar)
Too many range variables were used in a single query.

E:IDMLIB.IDM.RANGE.ILLEGOPTVAL(optname)
The specified option does not accept a value.

E:IDMLIB.IDM.RANGE.NEEDOPTVAL(optname)
The specified option requires a value.

E:IDMLIB.IDM.PERMDENY(cmd)
A required object was missing from a **permit** or **deny** command.

E:IDMLIB.IDM.SET.SYNTAX(valuetype)
Wrong type of value to **set** command.

E:IDMLIB.IDM.SYNTAX(lasttoken)
A syntax error was detected during parsing.

E:IDMLIB.IDM.TRACE.SYNTAX(type)
An invalid type was passed as an IDM trace specification.

E:IDMLIB.IDM.WITH(withoption)
An option value for the specified **with** option was not a constant.

BUGS

No reasonable recovery from syntax errors is made at this time.

- The **set** command sets IDM options to be used on all subsequent commands. For example, "set 11" causes IDM option 11 (return database server CPU time) to be sent on all future commands. **Unset** can be used to turn off options.

Idlfparse treats input conversion overflow as an error. The command tree will not be sent to the IDM/RDBMS software. This mainly affects the **append** and **replace** commands.

EXCEPTIONS

E:IDMLIB.CNVT.OVERFLOW(input, max)
: The user's input overflowed during conversion. The maximum value or size is also printed.

W:IDMLIB.CNVT.OVERFLOW(datatype, max)
: Conversion overflowed during data output. The maximum value or size is printed.

E:IDMLIB.IDM.BADARG(problem, argument, func)
: The specified argument to an IDL function was not valid.

E:IDMLIB.IDM.BADDIREC(direction)
: An unknown sort *direction* has been specified. *Direction* can be **ascending** or **descending** (or may be abbreviated to **a** or **d**).

E:IDMLIB.IDM.BADORDER(domain)
: An attempt was made to order by a domain that was not specified in the target list.

E:IDMLIB.IDM.BADTYPE(type)
: An unknown type was specified in a **create** statement.

E:IDMLIB.IDM.BADWITHOPT(optname)
: The specified **with** option is invalid.

E:IDMLIB.IDM.CANTBY(func)
: The specified function cannot accept a **by** clause.

E:IDMLIB.IDM.CANTFIX(func)
: The specified function cannot accept a **fixed** specification.

E:IDMLIB.IDM.CANTUNIQUE(func)
: The specified function cannot accept a **unique** specification.

E:IDMLIB.IDM.CANTWHERE(func)
: The specified function cannot accept a **where** clause.

E:IDMLIB.IDM.CONSTTOOLONG(type, maxlen)
: A constant was too long.

E:IDMLIB.IDM.EXEC.PROGID(type)
: An illegal **execute program** name was specified.

E:IDMLIB.IDM.EXEC.PARAM(cmdname, argnum)
: The specified argument to an **execute** was not a constant.

E:IDMLIB.IDM.FIELDSIZE(type)
: An illegal size was specified for a domain in a **create** statement.

E:IDMLIB.IDM.ILLEGPRTCT(mode)
: Illegal mode to **permit** or **deny**.

W:IDMLIB.IDM.LONGNODE(type, len)
: Long node was truncated.

W:IDMLIB.IDM.LONGTOKEN(token, maxlen)
: A token was too long and was truncated.

NAME

idlparse, idlfparse — build query trees from IDL program input

SYNOPSIS

```
#include <idmtree.h>
#include <idmenv.h>

ITREE *idlparse(text, env)
char *text;
IENV *env;

ITREE *idlfparse(ifp, env)
IFILE *ifp;
IENV *env;
```

DESCRIPTION

Idlparse reads and parses the given *text* as IDL input in the given *environment* and produces a list of trees corresponding to the statements in *text*. The return value points to a list of iCOMMAND nodes as described in *itree*(5I). The number of iCOMMAND nodes equals the number of commands in *text*. If *env* is IENVNULL, a default environment is used.

Certain commands consisting only of side effects take place immediately, although they continue to have an entry in the tree list. For example, the **range** statement updates the range table in the environment immediately upon being parsed.

The trees should be presented one at a time to *iputtree*(3I) to be sent to Britton Lee's IDM/RDBMS software.

When the tree is no longer needed, it must be explicitly freed using *itfree*(3I).

Idlfparse takes an IFILE pointer which must return tokens as described in *IftScan*(4I); it is in all other ways identical to *idlparse*. This input stream may be macro processed or otherwise manipulated before being parsed.

Idlparse accepts the language described in *System Programmer's Manual* , with the following changes:

- **Close** is not supported.
- Database administration functions (**dump database**, **dump transaction**, **load database**, **load transaction**, **roll forward**, **copy in**, **copy out**) are not supported using this interface; see *dba*(3I) and *itcopy*(3I) for details.
- The syntax *%name* creates placeholder nodes in the tree; values can be assigned using *icsubst*(3I).
- **Open file**, **close file**, **create file**, **read**, **write**, and **write eof** commands are not supported; *IftIFile*(4I) provides this functionality.
- **Reopen** is not supported; *IftIdm*(4I) gives equivalent functionality.
- **Setdate** and **settime** are available only using *idmdate*(1I).
- Syntax "0xNNN" accepts hexadecimal radix integers; "0oNNN" accepts octal radix integers.
- "0bNNN" accepts binary constants in hexadecimal radix.
- BCD constants are preceded by a '#' mark.
- Floating-point constants must begin with a digit. For example, use "0.1" instead of ".1" If an exponent is present, it must abut the final digit.
- Floating-point constants preceded by "0f" or "0d" indicate four- or eight-byte representations respectively. The default is eight-byte constants.

NAME

helpsys — interactive help subsystem

SYNOPSIS

helpsys(topic)
char *topic;

DESCRIPTION

Helpsys implements an interactive tree-structured help subsystem. Once ***helpsys*** is invoked, direct communication with the user is maintained until they explicitly exit ***helpsys***.

At any point the user is at a certain node in the help tree. The text associated with that node is printed, and the user is prompted for input. The user can enter a subtopic, causing descent through the tree, or one of the following commands:

%EXIT Exit ***helpsys***. End of file (control-D on UNIX, control-Z on VMS and the IBM Personal Computer, etc.) also works.

%TOP Return to the top menu of the help tree.

%UP Move one level back up the help tree.

The messages are printed out using the facilities of *IftMText*(4I), so recognition of the experience level applies.

An initial ***topic*** may be specified. For example, a ***topic*** of "IDL.APPEND" would start the help session at the section describing the IDL **append** command. If ***topic*** is CHARNULL, the help session begins at the top of the help tree.

If the session is not interactive, only one frame is printed. For example, "helpsys("IDL.APPEND")" would print the frame describing the IDL **append** command and then return immediately if the input was not a terminal.

The parameter feature of IftMText is not currently used.

DISCLAIMER

This module requires more evaluation. The human interface may change at some point in the future to be more user-friendly.

EXCEPTIONS

E:HELP.ATTOP
Already at top of help tree.

E:HELP.NONEXT
There is no automatic next frame.

E:HELP.UNKNOWN(topic)
The topic is unknown.

SEE ALSO

idmhelp(1I), IftMText(4I), the *INFO* facility on MIT ITS systems.

BUGS

The method of interaction does not extend gracefully to screen-based interfaces.

NAME

getprompt — get string with a prompt

SYNOPSIS

```
char *getprompt(buf, size, prompt)
char buf[];
int size;
char *prompt;
```

DESCRIPTION

Getprompt prints the *prompt* on the standard output and then reads a line from the standard input into *buf*. *Buf* can be at most *size* bytes long. The trailing newline is deleted. *Buf* is null-terminated.

In general, this is the only way to output a line to a terminal that is not terminated by a newline.

Getprompt returns CHARNULL on end of file, *buf* otherwise.

SEE ALSO

ifgets(3I)

NAME

getpass — get password securely from terminal

SYNOPSIS

char *getpass(prompt)
char *prompt;

DESCRIPTION

Getpass prints *prompt* and reads a password from the user. The null-terminated result is returned.

IMPLEMENTATION NOTES

Getpass is responsible for ensuring that the password is not visible. On a full-duplex terminal echo should be turned off. On a half-duplex terminal the password should be obliterated promptly.

SEE ALSO

getpass(3)

NAME

getparam, setparam — get/set a system parameter

SYNOPSIS

```
char *getparam(param)
char *param;

setparam(param, value)
char *param;
char *value;
```

DESCRIPTION

Getparam returns a pointer to the string value of the named parameter. The parameters are described in *params*(5I). If the parameter is unknown, a syserr occurs.

Setparam sets the named parameter to the specified value.

WARNINGS

The return value from *getparam* points to static memory that will be destroyed on the next call. Be sure to copy the return value before calling *getparam* again.

IMPLEMENTATION NOTES

On UNIX, this looks in the file "/usr/lib/idm/params" for the default set of names. Names in the user environment override these names.

On VMS, parameters are implemented as logical names. To avoid collision with other VMS logical names, "IDM_" is prepended to each IDMLIB parameter name. Both routines add "IDM_" to the beginning of the name passed in. *Getparam* then performs one logical name translation to get the value. *Setparam* defines the indicated logical name in the user mode process logical name table; its scope is the current running image.

Under AOS/VS, this looks for a file called "params" first in the current working directory, then in the user's search path, and finally in :idm:etc.

On CMS this looks in the table "IDMPARAM ASSEMBLE" and "userid.idmparam.*" for the default set of names. The file "IDMPARAM ASSEMBLE" contains system defaults, and is always processed. The file "userid.idmparam.*" is local to the user, and is processed if present.

The implementation ***must*** be extensible, that is, it must be possible to add new parameters without changing the code.

The implementation must define the entry point *_initparams* to be called by the initialization code. Since the I/O subsystem is not yet initialized when this is called, this should use native I/O.

The implementation should allow "secure parameters" — parameters that cannot be imported from the user. These are used for relatively static parameters that may have disasterous effects on users (e.g., the system call number used to access the database server). The system parameter file must contain the necessary information to decide which parameters are secure and which are not; that is, these must ***not*** be hard-coded into the implementation.

SEE ALSO

crackargv(3I), getenv(3), params(5I)

NAME

gethunpw — get host user name and password

SYNOPSIS

int gethunpw(excv)
char **excv;

DESCRIPTION

Gethunpw is the default exception handler for the R:IDMLIB.IDM.GETHUNPW exception (see *exc*(3I)). This exception is raised by *IftIdm*(4I) when the IDM complains that access is denied on an **open database** command.

Gethunpw determines the user name and/or password, setting the IDMHUNAME and IDMPASSWD parameters respectively (see *getparam*(3I)). It then returns zero to retry the open. This could be done by reading a file in the user's home directory, prompting the user, or whatever is appropriate for the host environment.

If it is not possible to determine the user name and/or password (e.g., if the program is run in background and the user must be contacted) then *gethunpw* will reraise the exception using *excabort*. This will print the message and abort the process.

SEE ALSO

getparam(3I), getpass(3I), iftidm(4I), params(5I), The section "System Level Security" in the *System Administrator's Manual*

IMPLEMENTATION NOTES

It may not be necessary to read the user name. On "trusted" hosts using user numbers, the host name is not strictly necessary. See the *System Administrator's Manual* for details.

The password should be read with echo turned off if possible, or should obscure the password echoed to the user.

If a file is read, care should be taken to ensure that it is not readable except by the owner to encourage security.

On some environments it is appropriate to set only the password, since setting the name changes the semantics of IDM authentication.

On CMS, the user is prompted for a user name and password if *isforegnd* is TRUE. If the user can supply the correct password for any "login" relation tuple, the user takes on the "huid" for that user. This supports applications authorized for only one user but executed by all users given the password.

BUGS

RETERROR et al **should work on all RETCODEs.**

NAME

geterr, clrerr, seterr, errstring, errclass, RETSUCCESS, RETWARNING, RETERROR — get, clear, set, classify, or interpret error codes

SYNOPSIS

RETCODE geterr()

clrerr()

seterr(code)
RETCODE code;

char *errstring(code)
RETCODE code;

RETCODE errclass(code)
RETCODE code;

BOOL RETSUCCESS(code)
RETCODE code;

BOOL RETWARNING(code)
RETCODE code;

BOOL RETERROR(code)
RETCODE code;

DESCRIPTION

Geterr returns a magic number that describes the current error. These reflect the full level of detail available from the operating system. *Errclass* classifies them into a limited range, which are a subset of the total set of error codes. *Errclass* can return values as described in *retcode*(5I).

Clrerr clears the system's idea of the current error. On some systems this may be done automatically at the next system call, so it is wise not to depend on it being sticky.

Seterr sets the system's idea of the error code. Normally this is only used in cases where an error is detected and exception is to be raised, so that an appropriate exit status may be returned to the host system. In particular, it is almost certainly an error to use *seterr* when the system error is already set, since this will most probably cause information to be lost.

Errstring returns a string describing the specified error *code*. Typical uses are "errstring(geterr())" and "errstring(IFERROR(ifp))". The return value may point to a static value that will be destroyed on the next call.

RETSUCCESS, *RETWARNING*, and *RETERROR* are predicates returning TRUE if their argument is a Sucess, Warning, or Error severity respectively. They are not guaranteed to work on return values from *geterr*, but will work on return values from *errclass* and other IDMLIB routines.

IMPLEMENTATION NOTES

In many systems *errstring* will use operating system services to get the string. For example, UNIX will use *syserrlist*[]. Whereever possible it should return the most specific message possible.

A namespace must be chosen such that the detailed system errors and the error codes may coexist. *Errclass* must map values from its range onto themselves, e.g., "errclass(RE_PERM)" → "RE_PERM".

On VMS, RETCODEs are VMS condition codes and *errstring* uses the *SYS$GETMSG* system service.

SEE ALSO

exc(3I), exit(3I), iferror(3I), retcode(5I)

The second is a *DATE* value, containing the date broken down into component fields. These quantities give the time on a 24-hour clock (including ticks), day of month (1–31), month of year (1–12), day of week (Sunday = 0), year (1900–), day of year (1–366), a flag that is nonzero if daylight saving time is ever used in your area, and an offset in minutes from Greenwich Mean Time.

Getclock returns the current system clock value. *Clocktodate* converts a CLOCK value into a DATE. The *zone* argument specifies an adjustment in minutes westward from GMT (e.g., the adjustment for California is 480 minutes westward from GMT; Amsterdam has an adjustment of −60 minutes westward from GMT, i.e., one hour eastward). The value LOCALTIME can be used to get local time adjustment including Daylight Savings Time (if the system parameter ISDST is set). Valid *zone* values are multiples of 30; if an illegal *zone* is used the TIMEZONE and ISDST system parameters are used to determine the time zone. *Datetoclock* performs the inverse function.

Diffclock determines the difference between two clocks. These may represent intervals or absolute times.

Correspondence with IDM Time and Date

Under Release 3, the IDM system stores time under GMT since the epoch. Thus, the *cl_day* field exactly matches the **getdate** IDL primitive. The *cl_ticks* field must be scaled between the host and the shared database system; the macros *IDMTOTICKS* and *TICKSTOIDM* provide this scaling.

Selection of the Epoch

The default epoch is January 1, 1900. If a later epoch is desired, the *EPOCHOFFSET* parameter can be set to the number of days between January 1, 1900 and the desired epoch. For example, an epoch of January 1, 1970 can be achieved by setting EPOCHOFFSET to 25568.

Extreme care should be taken if the offset is changed from the default, especially in environments where several hosts are connected to a single database server. Since dates are stored by IDM system software relative to this epoch, *all* hosts must agree on the epoch.

WARNINGS

The return values point to static data whose content is overwritten by each call.

This family of routines is defined only for dates in the range of Jan. 1, 1900 through Feb. 28, 2100.

IMPLEMENTATION NOTES

If the time zone is not available from the system, it should be supplied as a system parameter (see *getparam*(3I)).

The routine *getclock* is environment-dependent; the others are environment-independent.

SEE ALSO

fmtclock(3I), parsedate(3I), params(5I), date(2), ctime(3),

NAME

pextract — extract parameter value from list

SYNOPSIS

char *pextract(field, list)
char *field;
char *list;

DESCRIPTION

Pextract finds the named *field* in *list* and returns a copy of the value. If the field does not exist, *pextract* returns CHARNULL. If the field exists but has no value, the zero length string ("") is returned.

The list is a comma-separated list of *name(value)* pairs. For example, the list:

bs(512),linebuffer,mode(r)

specifies three parameters, two with values and one with no value. Elements of the list may be empty.

The value can have at most 256 characters. It may have commas and parentheses, but the parentheses must be properly nested.

By convention, arguments specifying "yes or no" options assert the option if no value is specified or if it has value of '1'. Digit '0' explicitly deasserts the option. For example, "linebuffer" and "linebuffer(1)" both assert the linebuffer option, while "linebuffer(0)" turns off the option.

Pextract with a zero length *field* parameter checks the parameter *list* for syntactical accuracy and raises an exception on errors. This should always be done before using *pextract* to extract values, as syntactically incorrect lists have undefined results.

WARNING

The routine value points to static data whose content is overwritten by each call.

EXAMPLES

list = "mode(r),linebuffer,bs(512),sync";

pextract("bs", list) → "512"
pextract("sync", list) → ""
pextract("trace", list) → CHARNULL

EXCEPTIONS

E:IDMLIB.PEXTRACT.SYNTAX(list)
Syntax error in list.

BUGS

Blanks are significant unless immediately following a comma. For example:

pextract("a", "a , b")

will return CHARNULL (that is, parameter "a" not found).

SEE ALSO

intro(1I), ifcontrol(3I), ifopen(3I)

NAME

pmatch — text pattern matching

SYNOPSIS

BOOL pmatch(pattern, string)
char *pattern;
char *string;

DESCRIPTION

Pmatch compares the pattern and the string, returning TRUE if they match. FALSE may be returned on non-matching patterns or malformed patterns.

Special characters in *pattern* are:

*	Matches zero or more characters.
?	Matches exactly one character.
[abc...]	Matches any single character listed. If the first character is a caret (' ^ ') then it matches any character *not* listed. The square brackets turn off special meaning for most other characters. After an open square bracket ("["), only the backslash character ("\") and the close square bracket character ("]") are magic. Thus the string "[*?^]" will match any single asterisk, question mark, or caret.
{abc,def,...}	Matches any of the comma-separated patterns listed.

Lowercase letters match themselves or the corresponding uppercase letter. Other characters match themselves. Any character can be preceded by a backslash to disable any possible magic interpretation.

Patterns may nest.

For single-character matching, the notation "[abc]" is more efficient than "{a,b,c}" and "?" is more efficient than "*".

EXAMPLES

pmatch("A*Z", "AZ") → TRUE
pmatch("A*Z", "AMNZ") → TRUE
pmatch("A*Z", "AZY") → FALSE
pmatch("?ABC", "AABC") → TRUE
pmatch("?ABC", "EABC") → TRUE
pmatch("?ABC", "A:ABC") → FALSE
pmatch("[ABC]", "A") → TRUE
pmatch("[ABC]", "B") → TRUE
pmatch("[ABC]", "X") → FALSE
pmatch("[^ABC]", "X") → TRUE
pmatch("A:{ABC,DEF}", "A:ABC") → TRUE
pmatch("A:{ABC,DEF}", "A:DEF") → TRUE
pmatch("A:{ABC,DEF}", "A:XYZ") → FALSE
pmatch("?:[UVW]*{COM,MOC}*[^XYZ]", "E:VERY COMPLICATED") → TRUE

SEE ALSO

string(3I)

NAME

printf, ifprintf, sprintf, tprintf — formatted output conversion

SYNOPSIS

printf(format [, arg] ...)
char *format;

ifprintf(ifp, format [, arg] ...)
IFILE *ifp;
char *format;

sprintf(s, format [, arg] ...)
char s[];
char *format;

tprintf(format [, arg] ...)
char *format;

DESCRIPTION

Printf places output on the standard output text file *stdout*. *Ifprintf* places output on the named output *ifp*. *Sprintf* places output in the string *s*, followed by the character '\0'. *Tprintf* prints to the trace file *stdtrc*. Because of the problems of mixed-language environments, *printf* should not be used in libraries that may be loaded with other languages.

Each of these functions converts, formats, and prints its arguments after the first under control of the first argument. The format argument is a character string which contains two types of objects: plain characters, which are simply copied to the output, and conversion specifications, each of which causes conversion and printing of the next successive *arg*.

Each conversion specification is introduced by the character '%'. Following the '%', there may be

- An optional minus sign '–' which specifies *left adjustment* of the converted value in the indicated field;
- An optional digit string specifying a *field width* ; if the converted value has fewer characters than the field width it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be done instead of blank-padding;
- An optional period '.' which serves to separate the field width from the next digit string;
- An optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string;
- The character **l** specifying that a following **d**, **o**, **x**, or **u** corresponds to a long integer *arg*.
- A character which indicates the type of conversion to be applied.

A field width or precision may be '*' instead of a digit string. In this case an integer *arg* supplies the field width or precision. A negative *arg* is equivalent to specifying no width or precision. Note that "%*" with an *arg* of −5 differs from "%−5."

In summary, in all formats except the floating-point formats, the ***width*** **specifies** the minimum number of characters that will be output and the *prec* specifies the **maximum** number of characters that will be output.

The conversion characters and their meanings are

duox The integer *arg* is converted to decimal, unsigned decimal, octal, or hexadecimal notation respectively.

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits are printed.

e The float or double *arg* is converted in the style "[–]d.ddde±dd" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double *arg* is printed in style **f** or in style **e**, whichever gives full precision in minimum space.

c The character *arg* is printed.

C The character *arg* is printed with nonprintable characters turned into a printable sequence.

s *Arg* is taken to be a string (character pointer) and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however if the precision is negative or missing all characters up to a null are printed. If the pointer is null it prints "[null]".

S *Arg* is printed as a string with non-printable characters escaped as in %C.

p *Arg* is printed as a pointer.

% Print a '%'; no argument is converted.

Use of any other keyletter is specifically undefined.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

RESTRICTIONS

Ifprintf is undefined on files with record-based presentations.

No more than fourteen parameters may be passed in one call to any of these routines. On 16-bit machines, longs and floats count as two parameters, doubles as four parameters. On 32-bit machines, doubles count as two parameters.

SEE ALSO

ifputc(3I)

NAME

rccount — subroutine for RSC and RIC source files

SYNOPSIS

long ntups, rccount();
ntups = rccount();

DESCRIPTION

Rccount returns the number of tuples affected by the last SQL or IDL command executed. For *select/retrieve* loops, *rccount()* should be called after the loop has finished executing.

If the last command was killed, or if there was some kind of error, the value is unreliable. *Rccount* must not be used to check for error or abnormal termination; rather, it should only be used when a query completes normally.

NAME

sqlparse, sqlfparse — build query trees from SQL program input

SYNOPSIS

ITREE *sqlparse(text, env)
char *text;
IENV *env;

ITREE *sqlfparse(ifp, env)
IFILE *ifp;
IENV *env;

DESCRIPTION

Sqlparse reads and parses the given *text* as SQL input in the given *env*ironment and produces a list of trees corresponding to the statements in *text*. The return value points to a list of iCOMMAND nodes as described in *itree*(5I). The number of iCOMMAND nodes equals the number of commands in *text*. If *env* is IENVNULL, a default environment is used.

Certain commands consisting only of side effects take place immediately, although they continue to have an entry in the tree list.

The trees should be presented one at a time to *iputtree*(3I) to be sent to the IDM/RDBMS software.

When the tree is no longer needed, it must be explicitly freed using *itfree*(3I).

Sqlfparse takes an IFILE pointer which must return tokens as described in *IftScan*(4I); it is in all other ways identical to *sqlparse*. This input stream may be macro processed or otherwise manipulated before being parsed.

Sqlparse accepts the language described in *SQL Reference Manual*. The following features are not documented in the manual:

- The syntax *%name* creates placeholder nodes in the tree; values can be assigned using *iesubst*(3I).
- Syntax "0xNNN" accepts hexadecimal radix integers; "0oNNN" accepts octal radix integers.
- "0bNNN" accepts binary constants in hexadecimal radix.
- BCD constants must be preceded by a '#' mark.
- Floating-point constants must begin with a digit. For example, use "0.1" instead of ".1" If an exponent is present, it must abut the final digit.
- Floating-point constants preceded by "0f" or "0d" indicate four- or eight-byte representations respectively. The default is eight-byte constants.

Sqlfparse treats input conversion overflow as an error. The command tree will not be sent to the IDM/RDBMS software. This mainly affects the **insert** and **update** commands.

EXCEPTIONS

E:IDMLIB.CNVT.OVERFLOW(input, max)
The user's input overflowed during conversion. The maximum value or size is also printed.

W:IDMLIB.CNVT.OVERFLOW(datatype, max)
Conversion overflowed during data output. The maximum value or size is printed.

E:IDMLIB.IDM.ALL.NOTONE
The "*" operator was used in a context in which too many tables were specified.

E:IDMLIB.IDM.BADARG(problem, argument, func)
The specified argument to an SQL function was not valid.

E:IDMLIB.IDM.BADDIREC(direction)
An unknown sort *direction* has been specified. *Direction* can be **ascending** or **descending** (or may be abbreviated to **a** or **d**).

E:IDMLIB.IDM.BADORDER(domain)
An attempt was made to order by a domain that was not specified in the target list.

E:IDMLIB.IDM.BADTYPE(type)
An unknown type was specified in a **create** statement.

E:IDMLIB.IDM.BADWITHOPT(optname)
The specified **with** option is invalid.

E:IDMLIB.IDM.CANTFIX(func)
The specified function cannot accept a **fixed** specification.

E:IDMLIB.IDM.CANTUNIQUE(func)
The specified function cannot accept a **unique** specification.

E:IDMLIB.IDM.CONSTTOOLONG(type, maxlen)
A constant was too long.

E:IDMLIB.IDM.EXEC.PROGID(type)
An illegal **start program** name was specified.

E:IDMLIB.IDM.EXEC.PARAM(cmdname, argnum)
The specified argument to a **start** command was not a constant.

E:IDMLIB.IDM.FIELDSIZE(type)
An illegal size was specified for a domain in a **create** statement.

E:IDMLIB.IDM.ILLEGPRTCT(mode)
Illegal mode to **grant** or **revoke.**

W:IDMLIB.IDM.LONGNODE(type, len)
Long node was truncated.

W:IDMLIB.IDM.LONGTOKEN(token, maxlen)
A token was too long and was truncated.

E:IDMLIB.IDM.MAPC.ESCAPE(string)
Illegal pattern-matching string.

E:IDMLIB.IDM.MATCHLIST.NOMAT
Two lists failed to match (usually in the **insert** command).

E:IDMLIB.IDM.NOTABLE(column-name)
There was no table specified for the given *column-name*.

E:IDMLIB.IDM.NOTFUNC(name)
The specified name was used in a context that would imply that it must be a function or aggregate name, but it cannot be recognized.

E:IDMLIB.IDM.NOTINT(type)
An integer was expected in the context.

E:IDMLIB.IDM.NUMARGS.TOOMANY(what, function, nargs)
The wrong number of arguments were given to the specified **function**. The correct number of arguments is given.

E:IDMLIB.IDM.OBJECT.SYNTAX(type)
Bad syntax for *object_name*.

E:IDMLIB.IDM.OPT.ILLEGAL(option)
An attempt was made to **set** an impossible or unknown option.

W:IDMLIB.IDM.OPT.NOTSET(option)
An attempt was made to **unset** an option that was not set.

W:IDMLIB.IDM.OPT.SET(option)
An attempt was made to **set** an option that was already set.

E:IDMLIB.IDM.OPT.TOOMANY
Too many options have been set.

E:IDMLIB.IDM.QUAL.AGG
An aggregate was found in a **where** clause.

E:IDMLIB.IDM.RANGE.BADOPT(optname)
The specified **from** clause option is not valid.

E:IDMLIB.IDM.RANGE.TOOMANY(nvar, maxvar)
Too many table references were used in a single query.

E:IDMLIB.IDM.RANGE.ILLEGOPTVAL(optname)
The specified option does not accept a value.

E:IDMLIB.IDM.RANGE.NEEDOPTVAL(optname)
The specified option requires a value.

E:IDMLIB.IDM.PERMDENY(cmd)
A required object was missing from a **grant** or **revoke** command.

E:IDMLIB.IDM.SET.SYNTAX(lasttoken)
Incorrrect syntax in the **set** command.

E:IDMLIB.IDM.SYNTAX(lasttoken)
A syntax error was detected during parsing.

E:IDMLIB.IDM.TRACE.SYNTAX(type)
An invalid type was passed as a trace specification.

E:IDMLIB.IDM.WITH(withoption)
An option value for the specified **with** option was not a constant.

BUGS

No reasonable recovery from syntax errors is made at this time.

Syntax errors should try to give you a pointer into the input line, rather than just a line number, so that user-friendly error messages can be generated.

SEE ALSO

dba(3I), ieopen(3I), iesubst(3I), iputtree(3I), itfree(3I), itxcmd(3I), ienv(5I), itree(5I)

NAME

stredit — do sophisticated output editing of numeric string

SYNOPSIS

```
char *stredit(str, exp, neg, pic)
char *str;
int exp;
BOOL neg;
char *pic;
```

DESCRIPTION

Stredit edits *str* under the control of *pic*. *Exp* represents the position of a decimal point in *str* as characters rightward from the end of *str*. If *neg* is set, *str* represents the magnitude of a negative number. For example, a *str* value of "123" with *exp* = −2 represents the number 1.23.

Pic is a series of characters describing the output stream. The values are:

9 Copy a digit from *str*.

Z Set the fill character to space. Copy a digit from *str*. Leading zeros are replaced by the fill character.

* Same as 'Z' except the fill character is set to '*'.

0 Same as 'Z' except the fill character is set to '0'.

, Replaced by itself unless we are currently suppressing zeros, when it is replaced by the fill character (space, zero, or asterisk). Also true of '.' and ' ' (space).

$ If present, represents a floating dollar sign. The dollar sign is moved to be adjacent to the first non-blank output character.

(The American pound mark/sharp sign and the British currency symbol overlap in the ASCII character set.) Behaves the same as '$'.

− Same as '$' except that it is only output if the *neg* flag is set. The same is true of '(' and '<'. This can be used in conjunction with '$' or '#'.

) Replaced by itself if the *neg* flag is set; otherwise, replaced by the fill character. The same is true of 'D', 'B', 'C', 'R', and '>'.

V Matches the decimal point in the input as specified by *exp*. Does not produce any output. If not specified, the end of *pic* is assumed.

The input is first aligned with a 'V' spec in *pic* (or the end of *pic* if no 'V' spec is present). Zeros are implicitly added to the front of *str* as necessary to match all replacement characters ('9', 'Z', etc.) in *pic*.

EXAMPLES

In the following examples, 'x' in the output field represents a space.

str	*exp*	*neg*	*pic*	*output*
123456	0	F	ZZZ,ZZZ,ZZ9.99	xxxxxx1,234.56
123456	0	F	***,***,**9.99	******1,234.56
123456	0	F	999,999,999.V99	000,123,456.00
123456	−2	F	ZZZ,ZZZ,ZZ9.V99	xxxxxx1,234.56
123456	0	F	$ZZ,ZZZ,ZZ9.99	xxxxx$1,234.56
123456	0	F	($Z,ZZZ,ZZ9.99)	xxxxx$1,234.56x
123456	0	T	($Z,ZZZ,ZZ9.99)	xxxx($1,234.56)
123456	0	F	−ZZ,ZZZ,ZZ9.99	xxxxxx1,234.56
123456	0	T	−ZZ,ZZZ,ZZ9.99	xxxxx−1,234.56

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strmcpy, strlen, strchr, strrchr — string operations

SYNOPSIS

```
char *strcat(dst, src)
char *dst, *src;

char *strncat(dst, src, n)
char *dst, *src;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(dst, src)
char *dst, *src;

char *strncpy(dst, src, n)
char *dst, *src;

char *strmcpy(dst, src, m)
char *dst, *src;

strlen(s)
char *s;

char *strchr(s, c)
char *s, c;

char *strrchr(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string *src* to the end of string *dst*. *Strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *Strncmp* makes the same comparison but examines at most *n* characters.

Strcpy copies string *src* to *dst*, stopping after the null character has been moved. *Strncpy* copies exactly *n* characters, truncating or null-padding *src;* the target may not be null-terminated if the length of *src* is *n* or more. *Strmcpy* copies a maximum of *m* characters, including a trailing null byte. All three return *dst*.

Strlen returns the number of non-null characters in *s*.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or CHARNULL if *c* does not occur in the string.

WARNINGS

Strcmp uses native character comparison, which is signed on PDP11s and VAX-11s, unsigned on other machines.

All string movement is performed character by character starting at the left. Thus overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

SEE ALSO

bytetype(3I)

NAME

sysedit — call system editor on a file

SYNOPSIS

RETCODE sysedit(name)
char *name;

DESCRIPTION

Sysedit calls the system editor on the named file. The file must be of type *IftHFile*(4I) and must be closed. It is normally expected to be a temp file.

Sysedit prepares the file for editing if necessary (e.g, setting locking modes, etc.), and then invokes an editor on the named file. If the system supports multiple editors, the pathname of the editor is specified by the EDITOR parameter (see *getparam*(3I)).

The return value is the exit status of the editor.

EXCEPTIONS

W:IDMLIB.CANTFORK(why)
Cannot create a new process to run the editor.

E:IDMLIB.SYSEDIT(file, problem)
Could not edit the file as noted.

IMPLEMENTATION NOTES

Sysedit should take care of such issues as file locking, file version numbers, etc. An exception should be raised if the editor cannot be invoked, and RE_CANT should be returned if a more specific error is not available.

Interrupts should be ignored while the editor is running. Locking them using *excalock* (see *exc*(3I)) is not sufficient, as interrupts will be improperly queued for delivery when *excaunlock* is called.

On VMS, only DEC-supported editors are supported by IDMLIB as editors that can be called by ***sysedit***. Any others may cause unexpected side-effects, especially in terminal settings.

On CMS, ***sysedit*** raises no exceptions.

SEE ALSO

exc(3I), getparam(3I), sysshell(3I)

NAME

syserr — print a fatal system error and abort

SYNOPSIS

syserr(format, arg, ...)
char *format;

DESCRIPTION

Syserr interpolates the *args* into *format* in the same format as *printf*(3I). The maximum number of *args* is three **integer** or pointer arguments. Formatting of **double** (64 bit) arguments will not work. They must first be converted into a string buffer and passed as a pointer to **char**. The result is printed on the standard error together with any other information about the state of the process that *syserr* can divine.

Format should have the syntax:

[!][module/]routine: text

where:

- If present, ! indicates that this is a catastrophic error from which recovery should not be attempted. If this is not included, *syserr* can raise "A:IDMLIB.SYSERR" after printing the message to attempt to back out to a top loop. Otherwise, *syserr* has no recourse except to immediately abort the process; no cleanup actions should be attempted. Preferably, a core dump will be generated.
- *Module/* is the name of the module, to be included if the routine name may not be meaningful in itself.
- *Routine:* is the name of the routine that is generating this error.
- *Text* is the text of the syserr. This is not intended to be "user-friendly", but is supposed to give a sophisticated systems maintainer whatever information is necessary to determine the problem. It should be terse, but complete.

Syserr is only to be used on internal errors. Users should never see any such error if the system is properly installed.

Syserr should prefix its output with a distinctive indication so that the user will easily understand that this is an internal system error.

If recovery is attempted, the magic variable *_ILibState* should be set to zero after backing out to indicate to IDMLIB that the syserr recovery has been successful.

EXCEPTIONS

A:IDMLIB.SYSERR(message)

Raised on non-catastrophic errors. *Message* will have been printed already.

IMPLEMENTATION NOTES

Extreme care must be taken to avoid using any unnecessary resources in this routine, since *syserr* may be called due to resource exhaustion. Also, *syserr* should not use the buffered primitives, since they may not be properly initialized. Typically, *syserr* should *sprintf* to a local buffer, and then do a physical write to the standard error file.

If the error is not catastrophic, *syserr* may invoke a routine to interactively log a Problem Report before raising the exception.

On VMS, this always signals the VMS condition IDMLIB-F-SYSERR, which is defined in IDMLIB. It never raises A:IDMLIB.SYSERR. You can catch this signal using the usual VMS conventions or you can link with IDMOBJ and provide your own *syserr*() routine. As supplied, *syserr* always causes the program calling to exit. In addition, the message generated by *syserr* bypasses the IDMLIB I/O system so it can't be redirected by internal manipulation to IDMLIB.

SEE ALSO

printf(3I)

BUGS

The number of arguments to *syserr* is limited to three.

NAME

sysshell — execute system command

SYNOPSIS

RETCODE sysshell(cmd)
char *cmd;

DESCRIPTION

Sysshell executes the system command ***cmd***. If ***cmd*** is CHARNULL an interactive command interpreter is created.

The exit status of the command interpreter is returned.

EXCEPTIONS

W:IDMLIB.CANTFORK(why)
Cannot create a new process to run the shell.

E:IDMLIB.SYSSHELL(problem)
The shell could not be executed.

IMPLEMENTATION NOTES

An exception should be raised if the command cannot be executed. An exception should be raised if ***sysshell*** cannot be emulated on the host system, and RE_CANT should be returned if a more specific error is not available.

Interrupts should be ignored while the subshell is running. Locking them using ***excalock*** (see ***exc***(3I)) is not sufficient, as interrupts will be improperly queued for delivery when ***excaunlock*** is called.

On VMS, this is implemented using the library routine *LIB$SPAWN*. Only DCL is currently supported as a shell since the DEC/Shell has not been tested with IDMLIB. This may change in a future release of IDMLIB.

On CMS, the ***cmd*** string must explicitly request CP or EXEC as required, e.g.,

"CP Q N"
"EXEC MYEXEC"
"MYPROGRAM"

Note that all are uppercase names; if a program name is given, it must not be the name of a CMS user area program.

SEE ALSO

sysedit(3I), system(3)

NAME

tempname — create a unique file name

SYNOPSIS

char *tempname()

DESCRIPTION

Tempname returns a file name that is unique on the system. The file is not created. The string is a copy, so it need not be saved before use.

If a file with this name is created, it will not be automatically deleted unless other arrangements are made, such as setting a file disposition in *ifopen*(3I).

The file name is dynamically allocated, and must be freed using *xfree*.

DIAGNOSTICS

This routine is guaranteed to work for at least twenty-six calls. After that, it will raise an exception.

EXCEPTIONS

A:IDMLIB.TEMPNAME.NOFILES

All temporary files are in use.

IMPLEMENTATION NOTES

On UNIX, the temp file should be in "/tmp". Other systems should behave analogously if possible. The implementation is encouraged to allow more than twenty-six calls.

On VMS, the file name is SYS$SCRATCH:IDMxxxxxx.n, where *xxxxxx* is the lower six characters of your process ID in hexadecimal, and *n* is a decimal number that is incremented once for each call to *tempname*(), starting at 0.

On CMS, the file name is vmuserid.IDLUTxy.A1, where *x* and *y* belong to the set A–Z, $, #, and @.

SEE ALSO

ifopen(3I)

NAME

tfset, tf, tflev, DPRINTF — trace package

SYNOPSIS

#include <idmtrace.h>

tfset(flags)
char *flags;

tf(flag, level)
int flag;
int level;

tflev(flag)
int flag;

DPRINTF(flag, level, (args))
int flag;
int level;
(LIST) args;

DESCRIPTION

Every process has available a vector of 100 trace flags, numbered 0-99. Flags 50-99 are reserved for use by IDMLIB itself and other Britton-Lee-supplied libraries; flags 0-49 may be used by the application. *Tfset* sets the trace flags as described by its argument. The syntax of *flags* is approximately as follows:

```
<flaglist>   ::= <flagclause> [ , <flagclause> ]*
<flagclause> ::= <flagrange> | <flagname>
<flagrange>  ::= <flagid> - <flagname>
<flagname>   ::= <flagid> [ . <flaglevel> ]
<flagid>     ::= <integer> | <identifier>
<flaglevel>  ::= <integer> | <identifier>
```

An individual flag name, e.g., "flag.level" specifies setting the *flag* to *level*. A range specification sets all the named flags to the specified level. A missing level is assumed to be one. The identifiers are looked up in a special file using *mapsym*(3I), using a prefix of 't'. IDMLIB identifiers are defined in the include file *idmtrace.h*.

The boolean routine *tf()* may be used to test if a given flag is at least at a particular level. The routine *tflev()* returns the level of a trace flag.

Calls to the trace package should be surrounded by *#ifdef*s to simplify deletion for small hosts. Trace information must always be printed using *tprintf()* (see *printf*(3I)).

The macro *DPRINTF* combines calls to *tf()* and *tprintf()* if the precomiler flag DEBUG is defined. *DPRINTF* expands to

```
if (tf(flag, level))
    tprintf args
```

Args must be enclosed in parentheses (see example below).

If DEBUG is not defined, *DPRINTF* is defined as the null string.

EXAMPLE

```
/* set flag 20 to level 2 */
tfset("20.2");

/* test flag 20 for level 1 or greater (TRUE in this example) */
if (tf(20, 1))
    tprintf( ... );
```

```
/* test for level 5 or greater (FALSE in this example) */
if (tf(20, 5))
        tprintf( ... );

/*
** Print index if flag 32 is level 4 or greater.
** Note that arguments to tprintf() are enclosed in parentheses
** when DPRINTF is invoked.
*/
DPRINTF(32, 4, ("index=%d\n", index));
```

SEE ALSO

crackargv(3I), mapsym(3I), printf(3I)

NAME

tupsetup, tupsep, tuphead, tupprint — print tuples

SYNOPSIS

```
#include <idmtlist.h>
#include <idmenv.h>

tupsetup(itl, env)
ITLIST *itl;
IENV *env;

tupsep(itl, where, ifp)
ITLIST *itl;
int where;
IFILE *ifp;

tuphead(itl, ifp)
ITLIST *itl;
IFILE *ifp;

tupprint(itl, ifp)
ITLIST *itl;
IFILE *ifp;
```

DESCRIPTION

This family of routines prints tuples as in *idl*(1I).

Tupsetup sets up the target list for printing. This involves computing the width of fields, etc., storing the results in the target list. Defaults are determined from the specified *env*ironment (someday). If *env* is IENVNULL a default environment will be used.

Tupsep prints a line between parts of the output. *Where* is −1, 0, or +1 for the line above, amidst, and at the bottom of the table respectively.

Tuphead prints a line with the titles.

Tupprint prints the data in a tuple.

For example, the following sample table shows which lines are generated by which routine:

```
|--------|----|     tupsep(−1)
|name    |  x |     tuphead
|--------|----|     tupsep(0)
|greg    |  12|     tupprint
|dave    | 114|     tupprint
|--------|----|     tupsep(0)
|name    |  x |     tuphead
|--------|----|     tupsep(1)
```

Tupsetup fills in print information in the target list. This can be modified before printing by the application. The ITL_PRINTABLE bit is set in *itl_flags* to indicate that this attribute can be printed; if cleared, the attribute is ignored by all routines. The other fields are:

itl_pwidth	The width of the output field.
itl_pprec	For floating point or BCD attributes, the number of digits after the decimal point.
itl_pfmt	The output format.

RESTRICTIONS

In all cases, the *ifp* should be type *IftTerm*(4I), since special terminal sequences are generated.

SEE ALSO

igettl(3I), igettup(3I), printf(3I)

NAME

typecnvt, cktypecnvt — generalized type conversion

SYNOPSIS

int typecnvt(intype, inlen, inval, outtype, outlen, outval)
int intype;
int inlen;
BYTE *inval;
int outtype;
int outlen;
BYTE *outval;

BOOL cktypecnvt(intype, inlen, outtype, outlen)
int intype;
int inlen;
int outtype;
int outlen;

DESCRIPTION

Typecnvt converts the data of type *intype* of length *inlen* pointed to by *inval* into the specified *outtype/outlen* into the buffer pointed to by *outval*. Returns the actual length of *outval* on success, negative on failure. An exception will also be raised on failure.

Cktypecnvt checks to see if the conversion can be performed, returning TRUE if it can and FALSE if it cannot.

CONVERSIONS

The following types are supported both as input and output types:

iCHAR	iFCHAR	iSTRING
iINT1	iINT2	iINT4
iFLT4	iFLT8	iBCD
iFBCD	iBCDFLT	iFBCDFLT
iBINARY	iFBINARY	

All conversions are supported except that iBINARY and IFBINARY can only be converted to or from one of the string types.

For convenience, if *intype* is iSTRING and *inlen* –1, *typecnvt* will use the *strlen*() of *inval* as the input length.

Conversion to iBCD and iBCDFLT should be done in the manner of the following example, which converts a character string named *x* into a BCD number named *b*:

```
BCDNO  b;
char x[12];
...
b.bcd_len = typecnvt(iCHAR, 12, x, iBCD, 11, _ _ b.bcd_str);
b.bcd_type = iBCD;
```

Bcd_len, bcd_str, and *bcd_type* are the three fields defined in *idmlib.h* for a structure of type *BCDNO*. Note that the field *bcd_str* is an array, and therefore the expression *_ _ b.bcd_str* in the above example does not need an ampersand preceding the *b.bcd_str*.

The following example shows how to convert a number stored in a BCDNO structure to some other type. We use *b* and *x* as defined in the previous example.

```
(void) typecnvt(b.bcd_type, b.bcd_len, _ _ b.bcd_str, iCHAR, 12, x);
```

EXCEPTIONS

E:IDMLIB.CNVT.CANT(intype, outtype)
The type conversion cannot be performed.

W:IDMLIB.CNVT.OVERFLOW(input, limits)
Data has been truncated.

W:IDMLIB.CNVT.GARBAGE(input, type)
Garbage (non-numeric data) was found on the end of the input stream during conversion to the specified type.

E:IDMLIB.CNVT.BADTYPE(type)
The *type* was unknown.

BUGS

Input types that have extra blanks or zeros that overflow a non-fixed output type overflow, even though they would not if the extraneous cruft were stripped.

SEE ALSO

string(3I)

NAME

UNSIGN — remove sign-extension bits from a byte

SYNOPSIS

```
int UNSIGN(byte)
int byte;
```

DESCRIPTION

UNSIGN strips the sign-extension bits off of the low-order byte of an *int* value, leaving eight bits.

UNSIGN must be used for comparisons of bytes which may have the high-order bit set; for example:

```
BYTE cmd;
if (UNSIGN(cmd) == iRANGE)
    ....
```

UNSIGN is implemented as a macro.

EXAMPLES

UNSIGN(0123) → 0123
UNSIGN(0200) → 0200
UNSIGN(0177600) → 0200

NAME

username — get user name

SYNOPSIS

char *username()

DESCRIPTION

Username returns a pointer to the current user's login name.

WARNINGS

This call may be expensive in some environments. It is wise to save the result if needed inside loops.

The return value points to static data space. However, since each call will return the same value, this should be irrelevant.

IMPLEMENTATION NOTES

This routine is intended to log names of users in logs and to store dynamically generated user profiles. As such it should make every attempt to identify the individual. For example, on UNIX a distinction is made between the logged in user and the executing user (which can be changed using the *su*(1) command); the former should be used.

This routine must always return a value; if the user name cannot be determined, then the numeric userid should be converted to a string and returned.

On VMS, this is implemented with the *LIB$GETJPI(JPI$_USERNAME)* system service.

SEE ALSO

su(1), getlogin(3)

NAME

itiutree, ituitree — convert to and from user tree (UTREE) representations

SYNOPSIS

```
#include <idmtree.h>

BYTE *itiutree(tree, &size)
ITREE *tree;
int *size;

ITREE *ituitree(utree)
BYTE *utree;
```

DESCRIPTION

UTREE's are a representation of an IDM tree structure with pointers removed. This form can be moved in memory, sent to a different process, written to a file, or otherwise moved and still be viable. The most common use is to pass compiled trees to a program generated by a precompiler.

Itiutree converts a normal *tree* such as might be returned by *idlparse*(3I) into a position independent byte stream, referred to as a *UTREE*. The UTREE form is dynamically allocated via *xalloc*(3I) and should be freed when no longer needed. The length of the UTREE in bytes is stored into the integer pointed to by *size*.

Ituitree converts a UTREE into a normal fully linked tree suitable for passing to further routines. This tree should be freed using *itfree*(3I) when no longer needed.

EXCEPTIONS

A:IDMLIB.IDM.UTREE.BADVER(tree, me)

A UTREE was passed to *ituitree* marked as version *tree*. Version *me* is the version that is understood.

A:IDMLIB.IDM.UTREE.TRASH

The tree that was passed to *ituitree* could not be decoded.

SEE ALSO

idlparse(3I), itfree(3I), xalloc(3I), itree(5I)

NAME

xalloc, zalloc, savestr, xfree, newmpool, mergempool, freempool, showmpool — main memory allocator

SYNOPSIS

```
#include <idmmpool.h>

BYTE *xalloc(size, mpool)
int size;
MPOOL *mpool;

BYTE *zalloc(size, mpool)
int size;
MPOOL *mpool;

char *savestr(str, mpool)
char *str;
MPOOL *mpool;

xfree(ptr)
BYTE *ptr;

MPOOL *newmpool(quantum, parentmpool)
int quantum;
MPOOL *parentmpool;

mergempool(oldmp, newmp)
MPOOL *oldmp;
MPOOL *newmp;

freempool(mpool)
MPOOL *mpool;

showmpool(mpool, flags)
MPOOL *mpool;
int flags;
```

DESCRIPTION

Memory is arranged into a collection of *memory pools*. Each pool contains a collection of zero or more *segments*, allocated by one of the allocation routines. Pools are organized into trees: except for the root, each pool has a unique parent and some number of children. If a memory pool is freed, all segments in that pool and all child memory pools are freed.

There are two special memory pools: *SysMpool* is the root memory pool, and *DefMpool* is the default memory pool. *SysMpool* can never be freed or subsumed into another pool. *DefMpool* is used if no memory pool is explicitly referenced. Initially, *DefMpool* is set to *SysMpool*.

Xalloc returns a pointer to a block of at least *size* bytes suitably aligned for storage of any type object out of the specified memory pool. If the memory pool is specified as MPOOLNULL *DefMpool* is used. *Zalloc* promises to return zeroed memory; in other respects it is identical to *xalloc*. *Savestr* allocates enough memory to store the string and copies it.

If *size* is zero then any pointer may be returned. If possible this pointer should be an illegal value so that attempts to reference it will be caught and rejected.

If memory cannot be allocated in any of these routines, the function pointed to by the global variable *NoMemFunc* will be called. This function must free memory and return non-locally. It must *NOT* raise an exception before freeing memory, since the process of raising an exception consumes memory. If *NoMemFunc* is not specified or returns the program is irrevocably aborted.

The argument to *xfree* is a pointer to a block previously allocated by *xalloc*, *zalloc*, or *savestr*; this space is made available for further allocation. Grave disorder will result if this pointer does not point to area that has been allocated — no special validation is performed.

Newmpool allocates a new memory pool. *Parentmpool* is the memory pool that should "own" this pool; when a pool is freed, all child pools are also freed. The *quantum* is advice from the application to the memory allocator about the size of blocks allocated from the system for this pool. It is *not* a limit on the maximum allocation size. If zero, a system default is used. This represents the nominal size of a new extent to be requested from the system if the existing memory pool cannot honor an allocation request. Applications that wish to allocate a large number of small segments may want to set the quantum high to minimize memory fragmentation.

Freempool frees the memory pool, all memory that was allocated out of it, and all child memory pools. *SysMpool* can never be freed; any attempt to do so will abort the program.

Mergempool merges *oldmp* into *newmp*; that is, all memory owned by *oldmp* is given to *newmp* and *oldmp* is deleted *a la freempool*. If *oldmp* is MPOOLNULL then *DefMpool* is used; if *newmp* is MPOOLNULL then the parent of *oldmp* is used.

Showmpool prints some information about the memory allocation in the given memory pool. MPOOLNULL may be used to see the system default pool. This routine is for debugging. If debugging is not enabled, it will act as a no-op. The *flags* are a map consisting of the following bits:

MPS_RECURSE
: Show subordinate memory pools as well.

MPS_SUMMARIZE
: Print one number instead of a report for every memory pool you look at.

MPS_NDISPLAY
: Don't display any memory pool information (but the total number of segments is returned).

MPS_DETAIL
: Print a detailed summary of memory utilization. This requires that trace flag ILIB-MEMORY.101 be turned on. This only works on some implementations.

The include file *<idmmpool.h>* must be included by all files using any of the memory pool routines and by any files passing non-default memory pools to any of the other routines.

To safely create a memory pool to be released in the event of an exception backout, use the code:

```
excalock();
if (exchandle("T:IDMLIB.ASYNC.*", excbackout) != 0)
    freempool(DefMpool);
DefMpool = newmpool(0, MPOOLNULL);
excaunlock(TRUE);
```

EXCEPTIONS

A:IDMLIB.XALLOC.SIZE(size)
: An illegal *size* (that is, less than zero) has been specified.

IMPLEMENTATION NOTES

Zalloc is provided as a separate function since some operating systems may have a particularly efficient way of getting zeroed memory.

UNIX implementations must also provide *malloc*, *realloc*, and *free* so that host programs using these primitives may coexist with IDMLIB. Similar comments apply to other operating systems.

Where possible, memory pools should be physically clustered to improve paging behavior.

The ***quantum*** is the number of bytes that should be requested from the system if there is no room in the memory pool to allocate the current request. It must *never* be interpreted as a maximum limit.

The implementation of memory pools must include a field named ***mp_flist*** that contains a list of routines to be invoked when the memory pool is freed. These routines should be invoked using:

```
_fl_call((FLIST **) &mp→mp_flist);
```

The FLIST structure is defined in <*idmflist.h*>.

Comments in the UNIX version explain more details.

The VMS version is the same as the UNIX version, except that the MPS_DETAIL feature of ***showmpool***() is not supported. The primitives *malloc*() and *free*() have been implemented as calls to *LIB$GET_VM* and *LIB$FREE_VM*.

GLOBALS

DefMpool	The default memory pool, used if MPOOLNULL is passed to one of the allocation routines.
SysMpool	The global memory pool. This pool is never freed. IDMLIB system resources are allocated from it.

BUGS-UNIX

UNIX ignores the ***quantum*** parameter to *newmpool*; in fact, memory pools are simulated, and memory can become horribly fragmented.

SEE ALSO

malloc(3)

NAME

xdump — dump bytes in hexadecimal to standard trace

SYNOPSIS

xdump(p, n)
BYTE *p;
int n;

DESCRIPTION

Xdump prints *n* bytes from *p* on *stdtrc* in hexadecimal. A character representation is also included.

Duplicate lines of output are suppressed (not printed) to compress the output. When a line is encountered which is different, printing resumes and a '*' character is output next to the byte count.

NAME

IDM file type introduction and implementation

SYNOPSIS

```
IFTYPE
{
        FUNCP  _ift_open;      /* open file */
        FUNCP  _ift_close;     /* close file */
        FUNCP  _ift_read;      /* physical read bytes */
        FUNCP  _ift_getbuf;    /* get a buffer of data */
        FUNCP  _ift_write;     /* physical write bytes */
        FUNCP  _ift_putbuf;    /* put a buffer of data */
        FUNCP  _ift_control;   /* control file */
        char   *_ift_name;     /* name of type */
        char   *_ift_params;   /* default open params */
};
```

DESCRIPTION

The IFTYPE structure defines the interface between the buffering system (*ifopen, ifgetc, ifwrite, ifcontrol,* etc.) and the type-dependent implementation. The routines in this interface are not intended to be called by application programs. However, it may prove convenient for a sophisticated application to define a special purpose file type.

The interface consists of seven procedures and two strings. The procedures implement file open, file close, reads of bytes, writes of bytes, and performing of control operations. The strings give a name of the file type for messages (e.g., "host file" or "IDM channel") and the set of defaults for *ifopen* parameters.

_ift_open(name, params, ifp) opens the named file with the specified *params*. It may store information necessary to access the file into the following fields of *ifp*: *_if_fd* (file descriptor or control block), *_if_dbin* (database instantiation number), *_if_lflags* (local flag bits), and *_if_x* (a pointer to a local control block used to store any additional information). It should return zero on success. On failure it may return −1 to issue a generic "cannot open" message or may raise a more specific exception with "Abort" severity. Memory should be allocated from the memory pool *ifp→_if_mpool*; this memory is deallocated automatically when the file is closed.

_ift_close(ifp) closes the file indicated by *ifp*. Any resources allocated in the *_ift_open* module should be released (memory allocated from *ifp→_if_mpool* will be deallocated automatically). If necessary, closing protocol should be sent.

_ift_read(ifp, buf, cnt) reads up to *cnt* bytes from *ifp* into *buf*. It should return the number of bytes actually read. It may return zero on end of file and −1 on error. This will only be called if the **_dio** attribute is set. If the **_rbf** attribute is set, *cnt* is guaranteed to be the block size.

_ift_getbuf(ifp) gets a buffer's worth of data and sets *ifp→_if_irbase* to point to it, returning the number of bytes available. For streams, the canned routine *_igetbuf* can be used, which calls *_ift_read* with the appropriate arguments.

_ift_write(ifp, buf, cnt) writes *cnt* bytes from *buf* onto *ifp*. The actual number of bytes written is returned, or −1 on error. This will only be called if the **_dio** attribute is set. If the **_rbf** attribute is set, *cnt* is guaranteed to be the block size.

_ift_putbuf(ifp, cnt) Put the buffer pointed to by *ifp→_if_orbase* containing *cnt* useful bytes to the file. This routine may pad out the buffer to up to *ifp→_if_rsize* bytes, but *ifp→_if_orptr* may not be used. The field *ifp→_if_orbase* is then reset to point to a clean buffer that must be at least of size *ifp→_if_rsize*, usually *ifp→_if_obbase* which points to the base of the buffer area. A *cnt* of zero indicates a zero-length record; a negative *cnt* must not put any data, but must still return a pointer to a new buffer. Returns the actual count of bytes written. This routine may

manipulate the fields *ifp→_if_obcnt* and *ifp→_if_obptr* to implement blocked files; on the first call (with *cnt* < 0) these will be zero.

_ift_control(ifp, params, args) performs the control operation(s) specified by the *params* field on *ifp*. The *args* field may point to additional arguments as needed by the control operation. The return value is passed back to the user, normally zero for success, negative for failure.

These routine should be bundled together into one module, and a new file type declared. For example:

```
extern int myopen(), myclose(), mycontrol();
extern int myread(), mywrite(), _igetbuf(), _iputbuf();
IFTYPE IftMyFile =
{
        myopen,    myclose,
        myread,    _igetbuf,
        mywrite,   _iputbuf,
        mycontrol, "My File", "bs(512)"
};
```

The routines in the interface can be declared STATIC; only the declaration of *IftMyFile* need be extern.

All control operations on the file should be implemented as an *ifcontrol* call rather than through ad hoc routines or global variables. This ensures maximum consistency, flexibility, and portability.

CONTROLS

The following controls should be implemented on all files where they make sense. The individual pages document all the controls that apply to that type — if the description is "standard" then they behave as described below. See also *ifcontrol*(3I). Controls beginning with underscore should never be issued by an application program.

cancel	Stop I/O on the file.
_delete	Remove the file indicated by *ifp→_if_name*. The file is guaranteed to be closed. This control is issued from *ifclose*(3I) if the file disposition is **delete**.
flushblock	Flush any blocked I/O that may be stored. This should be ignored on any file type that does not support blocked I/O (that is, more than one record per block).
_ioerr	Try to recover from an I/O error. If it returns a generic message will be raised. Files that can generate more specific messages or which can recover in some way may back out using another message. This is generated from the routine *_ioerr*.
reset	Reset the file to the beginning.
rewrite	Reset and truncate the file to zero length. The file must be enabled for writing for this to succeed. Writing will begin at the beginning of the file.

Control operations that are not understood should be ignored by the file type. However, some file types may want to catch operations that they cannot implement and flag them as errors if their failure would cause confusion.

Several flag bits can be set using *ifcontrol*(3I) to control the presentation of data to the type module. In general these should never be used by an application, since the correct functioning of the type module may depend on their setting.

_dio (Direct I/O) When set, the I/O subsystem will attempt to use *_ift_read* and *_ift_write* under some conditions. When clear, all I/O will be performed using *_ift_getbuf* and *_ift_putbuf*.

linebuffer (Buffer output one line at a time) When set, A call to *_ift_putbuf* will be made every time *ifputc* is called with a 'newline' argument. Systems that store text files as variable length records or that must convert newlines to carriage-return/line-feed combinations can set this mode to help simplify the file type module.

nameopt(O) (Can be used only in *_ift_params*.) If *O* is **r**, a non-null name is required on the *ifopen*(3I) call. If *O* is **n**, no name is allowed. If *O* is **o** or *nameopt* is not specified, a file name is optional.

_rbf (Record Based File) When set, all calls to *_ift_read* and *_ift_write* will have a count equal to the block size. This flag should only be asserted from inside the file type module; use by a user program can cause unexpected results.

EXCEPTIONS

Exceptions should normally be labelled

application.**IO**.*type.cause*

where *application* is the name of the application or library that defines this type, *type* is the name (or a permutation of the name) of this file type, and *cause* uniquely identifies the exception.

Exceptions common to all files are:

A:IDMLIB.IO.IOERR(filetype, filename, reason)
An I/O error occured on the specified file.

A:IDMLIB.IO.ROWOF(filetype, filename)
An attempt was made to read a write-only file.

A:IDMLIB.IO.WOROF(filetype, filename)
An attempt was made to write a read-only file.

RECORD-BASED VS. STREAM-BASED

Files can be physically record-based or stream-based. Stream-based files may have physical I/O performed on them of any length at arbitrary offsets; when read, any boundaries from the write that produced the data will not appear. UNIX files, strings, and the database server all fit this model. Record-based files have distinct record boundaries created by a write, and reads must pair one-to-one with writes.

SUPPORT ROUTINES

The following routines are provided for use by the file type modules:

_ioerr(ifp)
Signal an I/O error on *ifp*. This saves the error code in *ifp→_if_error* and calls *ifcontrol*(*ifp*, "_ioerr", BYTENULL) to attempt error recovery. If this *ifcontrol* call returns, then *_ioerr* will raise A:IDMLIB.IO.IOERR.

_ifsetbuf(ifp, bs, rs)
Creates the appropriate buffers on *ifp* of size *bs* with the record size set to *rs*. The order for selecting the size is: (1) the size specified by the user, (stored by *ifopen*(3I) in the *_if_bsize* and *_if_rsize* fields), (2) the *bs* and *rs* parameters, assuming they are positive, (3) the default for this file type, determined by the *_ift_params* field, and (4) a system default, determined by the IOBSIZE system parameter for *bs* or the copied from the *bs* to the *rs* specification. This call should only be used in the *open* routine, and *must* be called before attempting any I/O on *ifp*.

EXAMPLES

Examples of getbuf and putbuf routines are included here for a mythical type "xx" file.

Unblocked getbuf

```
xxgetbuf(ifp)
        register IFILE *ifp;
{
        ifp→_if_irbase = ifp→_if_ibbase;
        return (xxread(ifp, ifp→_if_ibbase, ifp→_if_bsize));
}
```

Unblocked putbuf

```
xxputbuf(ifp, cnt)
        register IFILE *ifp;
{
        ifp→_if_orbase = ifp→_if_obbase;
        if (cnt < 0)
                return (0);
        return (xxwrite(ifp, ifp→_if_orbase, cnt));
}
```

Blocked getbuf

```
xxgetbuf(ifp)
        register IFILE *ifp;
{
        if (ifp→_if_ibcnt ≤ 0)
        {
                int i;

                i = xxread(ifp, ifp→_if_ibbase, ifp→_if_bsize);
                if (i ≤ 0)
                        return (i);
                ifp→_if_ibcnt = i;
                ifp→_if_ibptr = ifp→_if_ibbase;
        }

        ifp→_if_irbase = ifp→_if_ibptr;
        ifp→_if_ibptr += ifp→_if_rsize;
        ifp→_if_ibcnt -= ifp→_if_rsize;
        if (ifp→_if_ibcnt < ifp→_if_rsize)
                ifp→_if_ibcnt = 0;
        return (ifp→_if_rsize);
}
```

Blocked putbuf

This example assumes that only fixed-length records are being delivered from the upper level, which is not a good assumption.

```
xxputbuf(ifp, cnt)
        IFILE *ifp;
        int cnt;
{
        int i;

        if (cnt > 0)
```

```
        {
                ifp→_if_orbase += cnt;
                ifp→_if_obcnt -= cnt;
        }
        if (ifp→_if_obcnt < ifp→_if_rsize)
        {
                i = ifp→_if_orbase - ifp→_if_obbase;
                if (i > 0)
                        xxwrite(ifp, ifp→_if_obbase, i);
                ifp→_if_orbase = ifp→_if_obbase;
                ifp→_if_obptr = ifp→_if_obbase;
                ifp→_if_obcnt = ifp→_if_bsize;
        }
        return (cnt);
}
```

SEE ALSO

ifclose(3I), ifcontrol(3I), ifopen(3I), pextract(3I)

NAME

IftHFile — host file file type

SYNOPSIS

extern IFTYPE IftHFile;

ifp = ifopen(filename, &IftHFile, params, IFNULL);

DESCRIPTION

This file type implements an interface to host operating system files. The *filename* is the name of the file on the host, in the host syntax.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
bs(N)	Buffer size. Default depends on the host operating system. In a record-based file system, this parameter may define the maximum record or buffer size as convenient; larger records may be truncated on reads and disallowed on writes. Note that in some cases, such as magnetic tape, buffer size and block size are the same.
cms(X)	*X* is passed directly to CMS for further interpretation. Ignored by other systems.
disp(D)†	Standard.
global	Standard.
linebuffer(B)†*	Standard.
mode(M)	Standard. **Mode(u)** is not required to work except on **temp** files.
padchar(B)†*	Standard.
pred(P)	If the file predisposition *P* is **new** then the file must not already exist; if **old** then the file must already exist. Otherwise the file must exist in read mode, and is created if necessary in the other modes.
rbp(B)†	Standard.
rs(N)	Standard.
temp	This is to be used as a temporary file. It may have more restrictive permissions, and it should be removed if the process exits.
trace(B)†	Standard.
type(T)	Detailed type information: currently **text** or **binary**. Where possible, this information is defaulted from the operating system. **Type(text)** may imply line buffering.
vms(X)	*X* is specific to VMS. It is ignored by other systems.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel	Standard.

clrerr	Standard.
_delete*	Remove the underlying file. This call should only be issued from *ifclose*(3I) if the **disposition** is **delete.** The file will have been closed already.
flushblock	Standard.
getbs	Standard.
getflags	Standard.
getrs	Standard.
reset	Standard. Should raise an exception if a reset is not possible on the file.
rewrite	Standard. Should raise an exception if a rewrite is not possible on the file.

EXCEPTIONS

A:IDMLIB.IO.HFILE.DELETE(filename, reason)

If the file cannot be deleted.

RESTRICTIONS

Update mode is only required to work in the following limited manner: a file opened for update may be written, reset, and read; writing may continue at end of file or after truncation. This is only required to work with disk files with the **temp** attribute.

IMPLEMENTATION NOTES

This file type is used for all types of host files, including disk files, unit record files (e.g., line printers), and terminals (however, see *IftTerm*(4I)). The implementation should be prepared to do any extra multiplexing necessary. In general, it is *not* a requirement that tape drives be supported; tapes should be accessed using the *IftLTape* module. The **type** parameter can be used if necessary to determine the detailed host file type. Where possible, the implementation should be flexible in the interpretation of this parameter, and it must never be required.

On CMS, this file type can be used for all types of host files including tapes and terminals. In both cases, the user opens the desired type, which then internally accesses *IftHFile*.

This module must ensure that special mappings are performed as necessary, e.g., mapping newline to carriage-return/line-feed on output to a text file if required.

On VMS, ANSI labeled tape is already supported by the operating system. *IftLTape* is defined to be *IftHFile*.

SEE ALSO

exc(3I), ifopen(3I), iftmtext(4I), iftterm(4I)

NAME

IftIdm — IDM channel file type

SYNOPSIS

extern IFTYPE IftIdm, IftReopen;

ifp = ifopen(dbname, &IftIdm, params, IFNULL);

rifp = ifopen(NULL, &IftReopen, "", ifp);

DESCRIPTION

IftIdm is the type descriptor for a raw connection to the database server. The *dbname* is used as the database name. If it is NULL, no database is opened.

IftReopen is used to get a reopened connection to the database server (see the *System Programmer's Manual* . The *name* parameter is unused, but the *ifp* of an existing connection of type *IftIdm* must be passed as the *baseifp* parameter.

The name of the device used to create the connection is divined from the IDMDEV system parameter. The syntax is similar to file specifications described in *ifscrack*(3I): "*device%driver*" specifies the *device* using *driver*. *Drivers* vary from system to system; common values are **multi** for the normal multiuser driver, **stand** for the standalone serial driver, and **xns** for the XNS ethernet driver. On UNIX the "/dev/" part of a device name may be omitted. For example, an IDMDEV set to idm%multi specifies the multiuser driver and device */dev/idm*. If no *driver* is specified, the IDMDRIVER parameter is interpreted as an *integer* index into the driver table. This use is discouraged.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard. Defaults off.
bs(N)*	Underlying block size. This is set to 2048 by default.
device(D)	Overrides the IDMDEV parameter.
disp(D)†	Ignored.
global	Standard.
lifeline	Used only on XNS connections. If set, this socket may be the "lifeline socket" — otherwise, opening the lifeline socket is illegal.
linebuffer(B)†*	Meaningless.
mode(M)*	Only **u** mode accepted (default). Reads and writes may be intermixed without intervening **reset** calls; however, the output should always be *ifflush*(3I)'ed before a read is attempted.
padchar(B)†*	Ignored.
rbp(B)†*	Illegal.
rs(N)	Standard.
trace(B)†	Standard. Defaults on.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(**4I**). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel	Cancel the current query. Sends a CANCEL command to the database server.
clrerr	Standard.
flushblock	Standard.
getbs	Standard.
getdbin	Put the "dbin" into the short pointed to by *args*.
getflags	Standard.
getrs	Standard. Identical to **getbs**.
_ioerr*	Used internally to signal an I/O error.
lifeline	Return TRUE into the BOOL variable pointed to by *args* if this is the lifeline socket; FALSE otherwise.
opendb(DB)	Open the named database.
setdbin	Set the "dbin" to the short pointed to by *args*.
reset*	Undefined.
rewrite*	Undefined.

IMPLEMENTATION NOTES

A machine-independent implementation of most of this module is provided. Physical interface to the host operating system is via the following dispatch table, defined in <*idmdriver.h*>:

```
struct idmdriver
{
    BYTE   *(*id_open)();  /* open a connection */
    FUNCP  id_huid;        /* send user id (XNS only) */
    FUNCP  id_close;       /* close a connection */
    FUNCP  id_read;        /* read bytes/packets */
    FUNCP  id_write;       /* write bytes/packets */
    FUNCP  id_cancel;      /* send a cancel (non-XNS only) */
    FUNCP  id_ioerr;       /* recover from I/O error */
};
```

There are two styles of system interface supported. The first is used for the serial or parallel IDM-HOST interface. The second is the XNS interface.

Serial/Parallel Interface

It is expected that these will map to one system or supervisor call if a multiuser driver is available. If not, these are expected to implement the single user (READWAIT/WRITEWAIT) protocol.

id_open(device, ifp)
: Open the named device and set all appropriate modes (e.g., baud rate on serial lines). Return the file descriptor for the channel. The specified *ifp* may be used if other parameters must be set.

id_huid
: Unused. This should *always* be specified as FUNCNULL for serial or parallel interfaces.

id_close(fd)
: Close the IDM connection.

id_read(fd, dbin, buf, count)
: Read *count* bytes from the given *fd* and *dbin* into *buf*. Return the number of bytes actually read as returned by the database server (including the EOR bit, which should be the

0x8000 bit).

id_write(fd, dbin, buf, count)
: Write *count* bytes from *buf* to the database server indicated by *fd* and *dbin*. Return the actual number of bytes written.

id_cancel(fd, dbin, what)
: Send a CANCEL or a CANCELP on the given *fd/dbin*. *What* is either CANCEL or CANCELP (defined in <*idmchan.h*>) to send the corresponding command.

id_ioerr(ifp)
: Handle an I/O error. In some cases this may require reading error tokens from the channel. In this case the processing should back out by raising an abort exception. On UNIX, the canned routine *_idmioerr* can be used for vanilla drivers. This depends on the driver setting the *errno* variable to one of the distinguished values listed in *idmcherr.h*.

XNS Interface

These calls are specific to XNS network implementations.

id_open(hostname, ifp)
: Open a connection to the specified *hostname*. Otherwise identical to the Serial/Parallel interface. This routine will probably want to call

```
_ifsetbuf(ifp, 0, MAXPACK);
```

where MAXPACK is the maximum packet size to be sent over the connection. Larger packets will still be accepted.

id_huid(fd)
: Send an HUID packet on the specified *fd* for identification purposes.

id_close(fd)
: Identical to the Serial/Parallel interface.

id_read(fd, buf, cnt, ptype)
: Read a single XNS SPP (Sequential Packet Protocol) packet from the socket indicated by *fd* into *buf*, which is of maximum size *cnt*. Return the actual number of data bytes read (excluding SPP header bytes) as the value, and the one byte Datastream Type into the byte indicated by *ptype*.

id_write(fd, buf, cnt, type)
: Write a single XNS SPP packet of specified *type* to the connection indicated by *fd*. The data part, if any, is specified by *buf* and *cnt*. If *buf* is BYTENULL no data is to be sent with this packet (i.e., the *type* completely specifies the content of the packet). If *type* is ATTENPACK (defined in <*idmxns.h*>) this packet *must* be sent "out of band" — that is, with the ATTENTION bit set in the Connection Control field of the SPP header.

id_cancel
: *MUST* be FUNCNULL for XNS based drivers.

id_ioerr(ifp)
: Same as specified in the Serial/Parallel interface.

Out-of-band data (that is, data with the ATTENTION bit set in the SPP Connection Control field) must be caught, normally by the *id_open* module. This must set two global variables: *_Attention* to TRUE to indicate that an attention packet has been received, and *_AttnFd* to the file descriptor of the file blessed with the out-of-band data. *_AttnFd* is not examined unless *_Attention* is set, so a possible implementation might set *_AttnFd* on every call to *id_write*, setting *_Attention* only when out-of-band data is actually received.

WARNINGS

Sending an "open database" command will not cause the "dbin" to be set automatically from the associated done packet. Use the **setdbin** *ifcontrol* call to set the "dbin" in this case, or open the database using the **opendb** call.

EXCEPTIONS

A:IDMLIB.IO.IDM.NODEVICE(devicename, why)

The database server device cannot be accessed.

R:IDMLIB.IDM.GETHUNPW(database)

Raised if the specified *database* is inaccessible on an **opendb** *ifcontrol* or on an initial open. If the system parameter GETHUNPW is set to '1' (see *params*(5I)), the default handler *gethunpw*(3I) will try to divine a user name and password (by asking the user if necessary) and return so that the open can be retried.

A:IDMLIB.IO.IDM.NODRIVER(options)

You have specified an unknown driver specifier in your IDMDRIVER parameter. *Options* gives the list of legal driver names.

A:IDMLIB.IO.IDM.TIMEOUT(device)

When you tried to read results from the database server you found that they had been cancelled because of an excessive delay.

SEE ALSO

gethunpw(3I), getparam(3I), ifcontrol(3I), ifopen(3I), igetdone(3I), igettl(3I), igettup(3I), params(5I), *System Programmer's Manual* .

NAME

IftIFile — IDM file file type

SYNOPSIS

extern IFTYPE IftIFile;

ifp = ifopen(filename, &IftIFile, params, idmifp);

DESCRIPTION

This file type interfaces with an IDM file. The *filename* names a file in the current database for the database server connection opened by the file *idmifp*, which must be of type *IftIdm*.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
bs(N)	Standard.
disp(D)†	Standard.
global	Standard.
linebuffer(B)†	Standard.
mode(M)	Standard. **Mode(a)** is simulated at open time, so multiple writers may trash each other. On a **mode(u)** file, writes may follow reads with an intervening **seek**, **reset**, or **rewrite** call; reads may follow writes with any of the above calls or an *ifflush*(3I) call intervening.
padchar(B)†*	Unused.
rbp(B)†	Standard.
rs(N)	Standard.
trace(B)†	Standard.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel	Standard.
clrerr	Standard.
_delete*	Delete the file. Invoked internally from *ifclose*(3I) if the **disposition** is **delete**.
flushblock	Standard.
getbs	Standard.
getflags	Standard.
getfn	Put the "file number" into the int pointed to by *args*. Used for the commands *idmread*(1I) and *idmwrite*(1I) to IDM tape.
getrs	Standard.
reset	Standard. Equivalent to **seek**(0).
rewrite	Standard. Reset file and truncate to zero length.

seek(N) Seek to byte *N* in the file.

weof Write an end-of-file at the current location in the file. The file must be writable. This truncates the file to the current offset, discarding any following data.

EXCEPTIONS

A:IDMLIB.IO.NOMODE(filetype, filename)
No **mode** parameter was passed to the open.

A:IDMLIB.IO.BADMODE(filetype, filename, mode)
An illegal *mode* was requested.

A:IDMLIB.IO.NOBASE(filetype, filename)
A base *ifp* was not supplied as required.

SEE ALSO

ifopen(3I), iftidm(4I), *System Programmer's Manual* .

BUGS

Mode(a) does not guarantee to write at the end of the file if other users are also writing to the same file.

NAME

IftKeyed — keyed host file type

SYNOPSIS

extern IFTYPE IftKeyed;

ifp = ifopen(filename, &IftKeyed, params, IFNULL);

DESCRIPTION

IftKeyed provides access to keyed host files. This is intended primarily for use by *IftMtext*(4I), and not for database applications.

The *ifopen* call returns a handle on the keyed file. The file may be opened for read-only or write-only. After being opened, a key may be specified using the **setkey** operation to *ifcontrol*. If the file is read-only the key must exist. If the file is write-only the key must not already exist; the key is created when set. *Ifcontrol* returns zero on success, negative on failure.

After setting a key in read mode, *ifgetc* will return bytes of the value associated with the key. End-of-file is returned when the key is exhausted.

In write mode, writes to the file are stored as the value of the key set by **setkey.**

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
disp(D)†*	Undefined.
global	Standard.
linebuffer(B)†*	Undefined.
mode(M)	Standard. Mode **u** is not supported.
padchar(B)†*	Unused.
rbp(B)†*	Undefined.
rs(N)	Standard.
tablesize(SZ)	Set the length of the hash-table to be *SZ* entries. This parameter is ignored unless the file is being newly created. Hash implementations only.
trace(B)†	Standard.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel*	Undefined.
clrerr	Standard.
flushblock	Standard.
getbs	Standard.
getflags	Standard.
getrs	Standard.
reset*	Undefined.

rewrite* Undefined.

setkey Set the key to the *arg* field of *ifcontrol.*

EXCEPTIONS

A:IDMLIB.IO.KEYED.BADFILE(filename)
The file opened is not in hashed-index format. Only files created with the *IftKeyed* module may be accessed as hashed-index files.

E:IDMLIB.IO.KEYED.DUPKEY(key, filename)
The file is opened for write operations, and the key passed to *ifcontrol* has a duplicate already in the file. Duplicate keys are not allowed.

I:IDMLIB.IO.KEYED.NOTFOUND(key, filename)
The file is opened for read operations. The key passed to *ifcontrol* was not found in the file. This is not necessarily an error condition.

E:IDMLIB.IO.KEYED.NOKEY
Ifcontrol has been called to perform a *setkey* operation, but no key was passed in the *arg* parameter.

IMPLEMENTATION NOTES

On UNIX, this module is heavily dependent on *lseek*(2), and is therefore considered to be machine dependent. A reliable *lseek* (or equivalent) is critical.

On VMS, this is implemented using the VMS Librarian facility. The keyed file is a VMS text library accessed by keys using the Librarian (LBR) routines.

SEE ALSO

ifthfile(4I), iftmtext(4I), ifopen(3I), ifcontrol(3I), lseek(2)

NAME

IftLoTerm — physical terminal file type

SYNOPSIS

#include <iftterm.h>

(Opened only on IDMLIB initialization; see below for details.)

DESCRIPTION

IftLoTerm is the machine-dependent terminal module. It is used as an underlying file type for *IftTerm*(4I). The standard files *stdout*, *stdin*, *stderr*, and *stdtrc* are all opened as type *IftTerm* with the underlying file of type *IftLoTerm*.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only. Note that this file is only opened during initialization, so these parameters are really moot.

autoclose(B)	Standard.
bs(N)	Standard.
disp(D)†	Ignored.
T*	Where *T* is one of the following characters: E standard error I standard input O standard output T standard trace
global	Standard.
linebuffer(B)†*	Standard.
mode(M)	Standard. Will always be either **r** or **w** as compatible with the **f** parameter.
padchar(B)†*	Unused.
rbp(B)†*	Standard. (Should never be set.)
rs(N)	Standard.
trace(B)†	Standard. It is a grave error to set **trace** mode on the file *stdtrc*, since tracing occurs on *stdtrc*.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel	Standard. Can be used to insure that any (potentially) buffered output will not actually appear on the screen, for example, on an interrupt.
clrerr	Standard.
cmode(B)	On *stdin*, **cmode**(1) turns off all buffering of the input, making each character available to the program as it is entered. Also, echoing is turned off, and it is the program's responsibility to echo characters entered on the keyboard. On output terminal files, **cmode**(1) initiaiizes the terminal for executing cursor motion commands. This may be a no-op on some systems. Cmode processing is used for screen-oriented applications, and the cmode controls prepare the terminal for such processing. **Cmode**(0) restores the terminal file to its original condition.

flushblock	Standard. Should be used if you really truly want data to actually kid-me-not get onto the user's screen.
getbs	Standard.
getflags	Standard.
getrs	Standard.
reset*	Meaningless.
rewrite*	Meaningless.

NONSTANDARD INTERFACES

This module must include several routines that lie outside the normal *Ift* protocols. These provide information to *IftTerm*(4I) about the nature of the physical terminal. The routines are:

_gettermdesc(termtype)

Here, *termtype* is a pointer to a character string specifying the name of the terminal. *_gettermdesc* returns a pointer to a TERMDESC structure, or TDNULL if the terminal description could not be found.

_isterm(ifp)

Returns TRUE if the *ifp* refers to a physical terminal.

_termtype()

Returns a pointer to a character string specifying the name of the user's terminal. *_gettermdesc*(*_termtype*()) should return a TERMDESC pointer for the user's terminal, if such a description exists.

IMPLEMENTATION NOTES

This module is machine-dependent. The *T* parameter is guaranteed to be the first parameter in the params list. The module is opened exactly four times. The first time it is opened with *T* equal to E (standard error), the second time with *T* equal to O (standard output), the third time with *T* equal to I (standard input), and the fourth time with *T* equal to T (standard trace). This order is guaranteed.

The open module should do any necessary initialization including setting the name of the file that will be used for printing, e.g.,

```
ifcontrol(ifp, "name(SYS$INPUT)", BYTENULL);
```

Two versions of this module may be necessary to implement the scheme for being compatable with the standard C library. For more information see *istdio*(3I).

In *cmode* processing, the host must insure that no special processing is done on output or input. Systems that 'add' carriage control to output strings must be discouraged from this practice.

On Berkeley UNIX systems, the SIGTSTP (i.e., the ^Z signal) is caught and an exception, "T:IDMLIB.JOB.SUSPEND" is raised. This exception is "invisible" — the default handler returns without printing any messages. When the process is continued after the stop, *IftLoTerm* raises the "T:IDMLIB.JOB.CONTINUE" exception, again transparently. These exceptions can be handled by forms-oriented applications that need to refresh screens.

SEE ALSO

exc(3I), iftterm(4I), ifthfile(4I), maketerm(8I)

NAME

IftLTape — ANSI labeled tape file type

SYNOPSIS

extern IFTYPE IftLTape;

ifp = ifopen(filename, &IftLTape, params, IFNULL);

WARNING

System V Release 2.0 (running on 3B series) does not provide access to basic tape operations. Therefore support of ANSI labeled tape is unavailable at this time.

The *hfile* file spec (see *ifthfile*(4I)) may be used as an alternative.

For example, to dump the transaction log from *mydb* to */dev/rmt/0m* with a block size of 1024 the following command would be used:

idmdump –t/dev/rmt/0m%hfile,bs\(1024\) mydb system

HARDWARE WARNING

On some tape controllers, record sizes that fall below some number of bytes (40 or so) will confuse the controller and cause unpredictable results. It is recommended that the user avoid writing extremely small records.

DESCRIPTION

This file type implements ANSI labeled tape, as specified by ANSI X3.27-1978. A level two implementation, including multivolume files and multifile volumes is guaranteed on most systems; higher level implementations may be supported on some systems.

Systems on which multivolume tapes are nonsensical (in particular, personal computers) may redefine this module to implement multivolume diskette files instead of ANSI tape. Multivolume files *must* be supported across all implementations however, and whereever possible arguments must maintain these semantics.

Tape mount requests are handled by communicating with the system operator using the *operator*(3I) primitives.

Files may be accessed by file number, file name, generation, and/or generation-version. If file number is specified, that file must match the other parameters. If file number is not specified, the first file encountered on the tape matching the name, generation, and generation-version is selected. Unspecified values of name, generation, and generation-version match anything. In read mode, unnamed files on the tape match anything: use fileno to correctly select the file. If none of file number, name, generation, or generation-version are specified, the first file on the tape is accessed.

The protection on all volumes and files must be blank. Files being written must be expired.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B) Standard. Defaults on.

bs(N) The block size. When a file is read, the block size is read off the tape. The default is 2048. Block sizes larger than 2048 exceed ANSI Standards X3.22-1978 and X3.39-1973 and hence may be incompatible with other operating systems.

density(N) The tape density. Some systems may be able to determine the density of a tape automatically, ignoring this parameter. *N* may be 800, 1600, or 6250. The default depends on the system, normally 1600.

disp(D)†* Ignored. Files are only deleted by being overwritten.

expiration(N) The expiration period in days. Ignored in read mode.

fileno(N) The file number desired. If both filename and fileno are supplied, they must match. If only one is supplied, the other is not checked. At least one must be supplied.

fileset(FS) The name of the fileset. If not supplied, any fileset is accepted in read mode. In write mode, any fileset will be accepted if we are appending to the tape.

format(F) The format of this file. Supported formats are 'F' for fixed length records and 'D' for variable length records. UNIX also supports 'U' for undefined; this format roughly resembles a stream.

gen(N) The generation number of this file. This may be viewed as an extension to the file name.

global Standard.

gver(N) The generation-version number. This may be viewed as an extension to the file name and generation number.

length(L) The tape length in feet. This is ignored if it can be determined in any other way. 2400 feet default.

linebuffer(B)† Standard.

mode(M) 'r' or 'w' for read or write mode. Writing a file destroys all files following on the volume set. If filename does not exist on the volume set, the file is appended at the end of the volome. 'a' appends to a volume set; fileno may not be specified. 'u' is not supported.

padchar(B)†* Standard. Used to pad fixed-length ('F' format) records out to full length. Defaults to '^'.

rbp(B)† Standard.

rs(N) Standard.

trace(B)† Standard.

unit(N) Do I/O on unit *N*. Unit zero is the default.

volume(VL) A comma-separated list of the names of the volumes comprising this volume set. If not supplied, any volume names are accepted.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel* Undefined.

clrerr Standard.

flushblock Standard.

getbs Standard.

getflags Standard.

getrs Standard.

newfile(FN) Terminate the current file being written, and start a new file named *FN*. Parameters **gen**, **gver**, **offset**, **expiration**, **format**, **bs**, **rs**, and **fileset** may also be specified, having the same semantics as on the open. This call may only be

used in 'w' or 'a' mode. The application must insure that the file is uniquely identified on the tape. This need not be supported on all implementations.

reset* Gives an error on some systems because of the difficulty of resetting to the beginning of a multi-volume file.

rewrite* Same as **reset**.

EXCEPTIONS

A:IDMLIB.IO.LTAPE.ABORT(filename)
The operator aborted the job, typically because the requested tape was not available.

E:IDMLIB.IO.LTAPE.DENSITY(density)
An impossible tape density was requested.

A:IDMLIB.IO.LTAPE.BADMODE(filename, mode)
An impossible I/O mode was requested.

W:IDMLIB.IO.LTAPE.NOOPERATOR
No operator is available; if the job requires operator assistance it will be aborted.

A:IDMLIB.IO.LTAPE.NOTEXPIRED
An attempt was made to write a file that was not expired.

A:IDMLIB.IO.LTAPE.PERM(protection)
You do not have permission to access this tape.

A:IDMLIB.IO.LTAPE.CANT(operation, reason)
One of the low level tape operations (e.g., backspace record) failed for the specified reason.

A:IDMLIB.IO.LTAPE.NOFILE(name)
The specified file could not be found on the tape.

E:IDMLIB.IO.LTAPE.RESET
Cannot use the **reset** control on labeled tape.

E:IDMLIB.IO.LTAPE.REWRITE
Cannot use the **rewrite**

E:IDMLIB.IO.LTAPE.SMALLBLOCK(blocksize, minblocksize)
Blocksize is smaller than the system mimimum **minblocksize**.

I:IDMLIB.IO.LTAPE.FILENO(fileno)
The specified file number will be accessed.

A:IDMLIB.IO.LTAPE.UNAVAILABLE
Issued if this system does not support labeled tape at all.

IMPLEMENTATION NOTES

Systems that support labeled tape should use the available system services.

If you must count tape usage using the **length** parameter, the total should be reduced slightly to allow for variant interrecord gap sizes and tape errors. The UNIX implementation uses 95.83% of the available length.

Systems that don't support any way to backspace a tape drive (notably UNIX System V) only allow overwrites of the tape (i.e., params of "fileno(1),mode(w)"). Fortunately this is consistent with other tape utilities on such systems.

On VMS, record sizes for tape vary, depending on the record format. The range for fixed-length records is 1 to 65,534 bytes; The range for variable-length records is 4 to 9,999 bytes, including the 4-byte Record Control Word. Therefore, the maximum length of the data area of a variable-length record is 9,995 bytes. IDMLIB will read or write variable-length records by

default, but fixed-length records may be specified with the **vms(rfm(fix))** parameter to open.

To comply with ANSI standards, the record size should not be larger than the maximum block size of 2,048 bytes.

BUGS-UNIX

Generations should be handled automatically.

It should be possible to set a buffer offset on output.

There should be some way to generate UHLa labels.

SEE ALSO

ifopen(3I), itapeopts(3I), operator(3I), dumptape(8I), inittape(8I), ANSI X3.27-1978, *American National Standard Magnetic Tape Labels and File Structure for Information Interchange.*

NAME

IftMText — Message-text file type

SYNOPSIS

extern IFTYPE IftMText;

ifp = ifopen(msgfile, &IftMText, params, IFNULL);

(void) ifcontrol(ifp, "setvect", (BYTE *) excvect);

DESCRIPTION

IftMText allows read-only access to the text of messages as described in *messages*(5I). The text returned is determined by the message name and arguments specified by the **setvect** control, the user's experience level (**Beginner**, **Able**, or **Expert**), and the query language being used.

The message to be read is set by the **setvect** parameter to *ifcontrol*. The argument is a vector as passed to an exception handler. The first element of the vector is the message name, and the rest of the arguments are parameters. The parameters are substituted into the text of the message as described in *messages*(5I).

Once a message vector is set, reads may be performed on the file to return the text of the message with the parameters substituted. End-of-file is returned at the end of the message. A new message may be selected using **setvect** without reopening the message file.

The query language can be specified by using the **qrylang** parameter to *ifopen*(3I) or *ifcontrol*(3I). If not explicitly set, the query language specifier is set from the QRYLANG parameter from *getparam*(3I).

The experience level can be set using the **exp** parameter to *ifopen*(3I) or *ifcontrol*(3I). If not explicitly set, the experience level is set from the EXPERIENCE parameter from *getparam*(3I).

WARNINGS

Writing to an *IftMText* file will always result in failure.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
bs(N)	Standard. Limits the length of a line.
disp(D)†*	Ignored
exp(EXP)	The experience level (**Beginner**, **Able**, or **Expert**). This argument, if used, overrides the EXPERIENCE parameter. This is useful for applications that require a fixed expertise level, such as a screen-based application that always wants a single line description for the status line.
global	Standard.
linebuffer(B)†*	Ignored.
mode(M)*	Only **r** accepted (default).
noerr	Force success on open. Read calls will return a canned **message**. Used for system messages.
padchar(B)†*	Ignored.
rbp(B)†*	Standard.

rs(N)*	Undefined.
trace(B)†	Standard.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel*	Undefined.
clrerr	Standard.
flushblock*	Undefined.
getbs	Standard.
getflags	Standard.
getrs	Standard.
reset*	Undefined.
rewrite*	Undefined.
setvect	Set the key-code of the message and the message arguments. If the message code is unknown the *ifcontrol* returns −1, but reads will still succeed.

IMPLEMENTATION NOTES

This module currently opens the underlying file as a file of type *IftKeyed*(4I).

An environment independent implementation exists, but on some environments it might be appropriate to redefine this module. For example, on a large-address-space machine it may be appropriate to cache frequently used messages.

IftMText is used by the exception handler, so the handling of exceptions within the module must be done very carefully in order to avoid infinite recursion.

SEE ALSO

exc(3I), getparam(3I), ifcontrol(3I), ifopen(3I), iftkeyed(4I), messages(5I)

NAME

IftScan, TK_PSEUDO — break an input stream up into tokens

SYNOPSIS

```
#include <iftscan.h>

extern IFTYPE IftScan;

ifp = ifopen(NULL, &IftScan, params, baseifp);

BOOL TK_PSEUDO(tok)
BYTE tok;

typedef struct
{
    int     tk_line;    /* line num of this token */
    BYTE  *tk_pdiff;    /* offset from base of baseifp→_if_irbase */
} TOKINFO;

#define  TOKINFNULL  ((TOKINFO *) NULL)
```

DESCRIPTION

Reading from an *ifp* of type *IftScan* reads characters from the underlying *baseifp* and turns them into tokens. Each token has a byte of type, two bytes of length, most significant byte first, and some amount of value defined by the length.

Token types are:

TK_ID — An identifier, i.e., a string of letters, digits, and underscores. If the KANJI compilation option is on, pairs of Kanji characters are accepted as letters.

TK_INT — An integer constant. Formats "0oNNN" and "0xNNN" are accepted.

TK_FLT — A floating point constant. Constants preceeded by "0f" or "0d" (intended to force four- and eight-byte floating-point representations respectively) are accepted.

TK_BCD — A BCD constant. The leading '#' is stripped off.

TK_SQSTR — A string constant set off by single quotation marks (' ' '). The quotation marks are stripped off.

TK_DQSTR — A string constant set off by by double quotation marks (' " '). The quotation marks are stripped off.

TK_BINARY — A binary constant (that is, a hexadecimal string beginning "0b"). The "0b" is stripped by *IftScan*.

TK_DPARAM — A parameter specifier, that is, a string beginning with a dollar sign or an ampersand. These are used to interpolate parameters into IDM stored-command definitions.

TK_OP — An operator (i.e., something containing special characters). Generally anything not fitting into the above classes is an operator. Recognized multi-character operators include:

```
>=     <=     !=
>=*    <=*    !=*     =*
*>=    *<=    *!=     *=
>>     <<     *!      :=      ->
```

TK_LINE — Returned at the beginning of each line. The value is taken from the global variable *LineNumber*. The *givenl* option must be set for these tokens to be generated.

TK_EOL A pseudo-token returned at the end of each line.

TK_INFO A pseudo-token may be returned before each token. The value is the TOKINFO structure containing the linenumber and the offset of this token in the input buffer of *baseifp*. The **giveinfo** option must be set for these tokens to be generated. The buffer size must be increased to allow these on each token.

Numeric tokens are not converted, i.e., they are returned as strings.

ANSI quote escaping is supported for both single quotation marks (' ' ') and double quotation marks (' " '). Within a quoted string, if two quotation marks of the same type (eg. single) are encountered, the first is discarded and the second is passed through uninterpreted. For example:

"he said ""hi there"""

will return a TK_DQSTR with a value of:

he said "hi there"

Comments (delimited by /* and */) are silently deleted, as is any unquoted white space.

If case folding is specified (i.e., if the **fold** parameter is specified or the FOLDCASE system parameter has the value 1) then all uppercase letters will be converted to lowercase. String constants are excepted.

The macro *TK_PSEUDO(token)* returns TRUE if the *token* is a pseudo-token (TK_LINE, TK_EOL, or TK_INFO).

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I) Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
bs(BS)	Standard. Limits the size of a single token.
countlines(B)†	Increment LineNumber on each input line.
disp(D)†*	Ignored.
fold(B)†	If set, uppercase is folded to lowercase except in strings. If not explicitly specified, defaults to the value of the FOLDCASE option.
giveinfo(B)†	Return TK_INFO tokens.
givenl(B)†	Return TK_LINE tokens.
global	Standard.
linebuffer(B)†*	Meaningless.
mode(M)*	May be **r** only (default).
padchar(B)†*	Ignored.
rbp(B)†	Standard.
rs(N)*	Undefined.
trace(B)†	Standard.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel*	Undefined.
clrerr	Standard.
flushblock*	Undefined.
getbs	Standard.
getflags	Standard.
getrs	Standard.
reset*	Undefined.
rewrite*	Undefined.

GLOBALS

LineNumber — The current line number. If countlines mode is set, this will be incremented on each input newline character. It is returned in TK_LINE and TK_INFO tokens if the **givenl** or **giveinfo** option (respectively) is set.

EXCEPTIONS

A:IDMLIB.IO.SCAN.CANTWRITE
An attempt was made to write to the scanner.

A:IDMLIB.IO.NOBASE(name, openname)
The **baseifp** passed in was IFNULL.

E:IDMLIB.IO.SCAN.EOFINCOMMENT(type, name)
An end of file was found while scanning a comment while reading the specified underlying file.

E:IDMLIB.IO.SCAN.EOFINSTRING(type, name)
An end of file was found while scanning a comment while reading the specified underlying file.

E:IDMLIB.IO.SCAN.NLINSTRING(type, name)
A newline was found while scanning a quoted string while reading the specified underlying file.

E:IDMLIB.IO.SCAN.NOROOM(type, name)
No room was available to store a token while reading the specified underlying file.

NAME

IftString — in-core string file type

SYNOPSIS

extern IFTYPE IftString;

ifp = ifopen(CHARNULL, &IftString, params, IFNULL);
(void) ifcontrol(ifp, "setstring,bs(−1)", buffer);

DESCRIPTION

This file type causes "input/output" to happen into an incore buffer. Only read (**r**) and write (**w**) modes are supported. Reads return successive bytes from ***buffer*** until the buffer size is reached, when they return EOF. Writes put characters into ***buffer***.

A flush on a **w** mode file puts a null (zero) byte into the next position of ***buffer*** and resets the pointer to the beginning.

WARNINGS

Care must be taken not to overwrite the buffer if the size is not specified.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Parameters marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

autoclose(B)	Standard.
bs(BS)†	The size of the buffer. If not specified, it is set to be very large. If the open is for read mode and the *BS* is negative, it is set to the ***strlen*** of the string (see *string*(3I)). The resulting length (excluding trailing null byte) will *not* be reflected into the block size (which will still be negative), but *will* be returned by a **getrs** control call. Note that the **bs** may also be set by *ifcontrol*.
disp(D)†*	Ignored.
global	Standard.
linebuffer(B)†*	Undefined.
mode(M)	Mode **r** or **w** only.
padchar(B)†*	Unused.
rbp(B)†*	Undefined.
rs(N)*	Undefined.
trace(B)†	Standard.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel*	Undefined.
clrerr	Standard.
flushblock	Standard.
getbs	Standard. Note that this may not actually show the string length; **getrs** should be used instead.
getflags	Standard.

getrs	Standard.
reset	Reset the pointer to the beginning of the buffer.
rewrite*	Undefined.
setstring	Change to a buffer selected by the *arg* field to *ifcontrol*.

EXCEPTIONS

E:IDMLIB.IO.STRING.OVERFLOW

No room is available to put more characters into the buffer.

IMPLEMENTATION NOTES

Since there is no need for a special intermediate buffer in this file type the normal buffering is bypassed.

NAME

IftTerm — terminal file type

SYNOPSIS

#include <iftterm.h>

(Opened only on IDMLIB initialization; see below for details.)

DESCRIPTION

IftTerm accepts device-independent terminal escape sequences and interprets them for a specific device. The files *stdin*, *stdout*, *stderr*, and *stdtrc* are type *IftTerm*. *IftTerm* is a machine-independent module with one major exception: it will only work with ASCII terminals.

IftTerm must open a machine-dependent underlying file type. If no base *ifp* is passed from *ifopen*(3I), *IftTerm* will open a file of type *IftLoTerm*(4I).

The protocol uses an eight-bit path, i.e., all 256 possible codes are reserved for use. Non-printable characters are used as control codes.

Control and escape sequences comply with American National Standards X3.41-1974 and X3.64-1979 except as noted below.

Graphics

Special graphics may be output by sending the ITC_SS2 (Single Shift 2) character followed by one of the following:

ITG_TLC	top left corner
ITG_TRC	top right corner
ITG_BLC	bottom left corner
ITG_BRC	bottom right corner
ITG_TT	top 'tee'
ITG_BT	bottom 'tee'
ITG_LT	left 'tee'
ITG_RT	right 'tee'
ITG_VB	vertical bar
ITG_HB	horizontal bar
ITG_X	cross (like '+')
ITG_BLOTCH	an out-of-band 'blotch' character

Command Sequences

Certain control operations may be performed using a "command sequence" beginning with the CSI (Command Sequence Introducer) character, followed by parameters. The parameters are decimal numbers, expressed as numeric digit strings. The parameters are separated by semicolons, and terminated by a "final character" that determines the actual operation to be performed. For example, the sequence

ITC_CSI 0 ; 1 ITC_SGR

invokes SGR (Select Graphic Rendition) with arguments zero and one.

Valid final characters for CSI sequences are

ITC_SGR	select graphic rendition
ITC_CUF	move cursor right (forward)
ITC_CUD	move cursor down
ITC_CUB	move cursor left (backward)
ITC_CUU	move cursor up
ITC_CUP	absolute cursor position
ITC_ED	erase display

IftTerm translates these sequences into the actual control signals required by the terminal. The

information required to perform this translation is obtained from the underlying file type.

Graphic Renditions
Parameters to ITC_SGR may be

ITP_PRIMARY	primary (default) rendition
ITP_BOLD	bold or increased intensity
ITP_FAINT	faint, decreased, or colored
ITP_ITALIC	italic
ITP_UNDER	underscore
ITP_BLINK	slow blink (under 150/minute)
ITP_FLASH	fast blink (over 150/minute)
ITP_REVERSE	reverse video

Note that these are integer values rather than strings.

Cursor Control
The CSI sequences that control cursor motion and clear the screen are not guaranteed to work unless ***stdout*** is in "cmode" (cursor-motion mode). ***Stdout*** may be set in this mode by using the **cmode** control (see the section on controls below).

The rules for screen control and cursor motion follow the ANSI standards. The "home" position of the screen is line 1 (one) and column 1 (one). The absolute cursor motion sequence CUP takes two arguments: the line number followed by the column number. The other cursor-control sequences take no arguments.

Erase Display
The ITC_ED (Erase Display) command ***must*** be preceeded by the ITP_ED_ALL parameter to specify erasure of the entire display. Partial erasure is not supported at this time.

Extensions to the Standards
The following characters represent extensions to the ANSI standards:

ITX_RESET	Reset the terminal to a known state.

The ASCII characters SO (Shift Out, octal 016) and SI (Shift In, octal 017) **do not** shift to the G1 character set as specified by X3.41. Instead they "quote" characters that are passed directly through to the terminal without interpretation. This is intended to support an additional graphic set such as required for Korean ideographs. These are not otherwise supported in the code.

Multi-byte characters such as Kanji are supported. Both bytes must be in the range 0xA0 through 0xFE inclusive. This preempts use of the G1 character set as specified by X3.41.

Shorthands
As a convenience, certain common sequences are defined as individual strings:

ITS_PRIMARY	primary graphic rendition
ITS_BOLD	bold rendition
ITS_UNDER	underscore
ITS_BLINK	blink
ITS_REVERSE	reverse video
ITS_CUF	move cursor right
ITS_CUD	move cursor down
ITS_CUU	move cursor up
ITS_CUB	move cursor left
ITS_CUP	absolute cursor position (template)
ITS_CLEAR	clear screen

For example, to print "STRING" in bold, the sequences

```
printf("%c%d%cSTRING%c%d%c\n",
   ITC_CSI, ITP_BOLD, ITC_SGR,
```

ITC_CSI, ITP_PRIMARY, ITC_SGR);

and

printf("%sSTRING%s\n", ITS_BOLD, ITS_PRIMARY);

are equivalent.

To move the cursor to line 24, column 10, the sequences

printf("%c%d;%d%c", ITC_CSI, 24, 10, ITC_CUP);

and

printf(ITS_CUP, 24, 10);

are equivalent.

This file type will almost certainly be extended greatly in the future.

PARAMS

Params marked with † are also legal controls. Descriptions reading "standard" are documented in *ifopen*(3I). Params marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only. Note that this file is only opened during initialization, so these parameters are really moot.

autoclose(B)	Standard.
bs(N)	Standard.
disp(D)†	Ignored.
T*	See *IftLoTerm*(4I).
global	Standard.
linebuffer(B)†*	Standard.
mode(M)	Standard. Will always be either **r** or **w** as compatible with the **f** param.
padchar(B)†*	Unused.
rbp(B)†*	Standard. (Should never be set.)
rs(N)	Standard.
trace(B)†	Standard. It is a grave error to set **trace** mode on the file *stdtrc*, since tracing occurs on *stdtrc*.

CONTROLS

Controls described as "standard" are documented in *ifcontrol*(3I) and *intro*(4I). Controls marked with * may have unexpected side effects; they should normally be reserved by internal use by the file type only.

cancel	Standard. Can be used to ensure that any (potentially) buffered output will not actually appear on the screen; for example, on an interrupt.
clrerr	Standard.
cmode(B)	On *stdout* or *stderr*, **cmode**(1) turns on cursor motion mode, enabling the use of cursor motion CSI sequences. On *stdin*, **cmode**(1) turns off all buffering of the input, making each character available to the program as it is entered. Also, echoing is turned off, and it is the program's responsibility to echo characters entered on the keyboard. **Cmode**(0) restores the terminal file to its original condition. Also passed to *IftLoTerm*(4I).
flushblock	Standard. Should be used if you really truly want data to actually kid-me-not get data onto the user's screen.

getbs	Standard.
getcols	Return the number of columns on the screen into the integer pointed to by *args*.
getflags	Standard.
getlines	Return the number of lines on the screen into the integer pointed to by *args*.
getrs	Standard.
reset*	Meaningless.
rewrite*	Meaningless.
term(T)	Set the terminal type to *T*. If *T* is unknown, the type will not be changed if a type is already set, otherwise it will be set to **dumb**. If no type *T* is specified, the terminal type is divined from the operating system, e.g., by the TERM parameter (see *getparam*(3I)). The **term** control may be re-issued to change the terminal type, but the application must be prepared to handle parameters that change, such as the screen size.

NONSTANDARD INTERFACES

This module calls several routines that are defined in *IftLoTerm*(4I). These are:

_gettermdesc
: Returns a pointer to a structure that provides a description of the physical terminal.

_isterm
: Returns a BOOL indicating whether output is to a terminal.

_termtype
: Returns the type of the terminal currently in use.

These routines are described at length in *IftLoTerm*(4I).

IMPLEMENTATION NOTES

This module is machine-independent except for its ASCII dependence. It opens the underlying file as a file of type *IftLoTerm*(4I).

The module is opened exactly four times. The first time it is opened with *T* equal to E (standard error), the second time with *T* equal to O (standard output), the third time with *T* equal to I (standard input), and the fourth time with *T* equal to T (standard trace). This order is guaranteed.

SEE ALSO

iftloterm(4I), ifthfile(4I), maketerm(8I), ANSI X3.4-1977, *American National Standard Code for Information Interchange;* ANSI X3.41-1974, *American National Standard Code Extension Techniques for use with the 7-bit Coded Character Set of American National Standard Code for Information Interchange;* ANSI X3.64-1979, *American National Standard Additional Controls for use with American National Standard Code for Information Interchange.*

NAME

Introduction to file and data formats.

DESCRIPTION

This section describes the file formats used by Britton Lee libraries and applications and the data structures used by IDMLIB.

NAME

idmtokens — values of IDM communication tokens

DESCRIPTION

octal	hex	l	name	semantics
0001	0x01	0	TLEND	end of target list
0002	0x02	0	QLEND	end of qualification list
0003	0x03	0	TIME	substitute current time
0004	0x04	0	USERID	substitute current userid
0005	0x05	0	DBA	substitute uid of database admin
0012	0x0a	0	HOST	substitute host id
0013	0x0b	0	DATE	substitute current date
0014	0x0c	0	DATABNAME	substitute current database name
0023	0x13	0	EXITIDM	close database
0030	0x18	0	REP_OLD	audit: old value before replace
0031	0x19	0	REP_DUP	audit: a replace was a duplicate
0032	0x1a	0	APP_DUP	audit: an append was a duplicate
0033	0x1b	0	SYNC	flush cache memory to disk
0040	0x20	F	HUNAME	host user name for login id
0041	0x21	F	PASSWORD	password for login id
0042	0x22	F	ATTR	attribute name
0043	0x23	F	BCDFLT	floating point BCD
0044	0x24	F	FBCDFLT	fixed length BCDFLT
0045	0x25	F	FBINARY	fixed length binary
0046	0x26	F	FBCD	fixed length BCD
0047	0x27	F	FCHAR	fixed length CHAR
0050	0x28	F	VAR	reference variable
0051	0x29	F	OPTIONS	specify processing options
0052	0x2a	F	FORMAT	define format of returned tuples
0053	0x2b	F	PARAM	stored command parameter
0054	0x2c	F	PCHAR	string that may contain pattern matching characters
0055	0x2d	F	BINARY	binary string
0056	0x2e	F	BCD	binary coded decimal
0057	0x2f	F	CHAR	character string
0060	0x30	1	INT1	one byte integer
0061	0x31	1	ORDERA	order ascending
0062	0x32	1	ORDERD	order descending
0063	0x33	1	ERROR	error follows
0064	0x34	2	INT2	two byte integer
0065	0x35	2	TYPE	
0066	0x36	2	NVAR	count any attribute
0070	0x38	4	INT4	four byte integer
0071	0x39	4	FLT4	four byte floating point number
0074	0x3c	8	FLT8	eight byte floating point number
0100	0x40	0	ABS	take absolute value
0101	0x41	0	MINUS	take arithmetic inverse
0102	0x42	0	NOT	take logical not
0103	0x43	0	CNVTI1	convert to i1
0104	0x44	0	CNVTI2	convert to i2
0105	0x45	0	CNVTI4	convert to i4
0106	0x46	0	CNVTF4	convert to f4
0107	0x47	0	CNVTF8	convert to f8
0110	0x48	0	CNVTBINARY	convert to binary

0111	0x49	0	CNVTFBINARY	
0112	0x4a	0	AOPCNT	"count" aggregate
0113	0x4b	0	AOPCNTU	"count unique" aggregate
0114	0x4c	0	AOPSUM	"sum" aggregate
0115	0x4d	0	AOPSUMU	"sum unique" aggregate
0116	0x4e	0	AOPAVG	"average" aggregate
0117	0x4f	0	AOPAVGU	"average unique" aggregate
0120	0x50	0	AOPMIN	"min" aggregate
0121	0x51	0	AOPMAX	"max" aggregate
0122	0x52	0	AOPANY	"any" aggregate
0123	0x53	0	CNVTRNAME	convert relid to relname
0124	0x54	0	CNVTRID	convert relname to relid
0126	0x56	0	AOPONE	return err if more than 1 value
0127	0x57	0	AOPONEU	return err if more than 1 distinct value
0160	0x70	1	CNVTBCD	convert to bcd
0161	0x71	1	CNVTCHAR	convert to character
0162	0x72	1	CNVTFBCD	
0163	0x73	1	CNVTFCHAR	
0164	0x74	2	SUBSTR	take substring
0165	0x75	2	CNVTFLTBCD	convert to BCDFLT
0166	0x76	2	CNVTFFLTBCD	
0167	0x77	2	FIXEDPT	
0200	0x80	0	RESDOM	specify result domain
0201	0x81	0	EQ	==
0202	0x82	0	NE	!=
0203	0x83	0	GT	>
0204	0x84	0	GE	>=
0205	0x85	0	LT	<
0207	0x87	0	RNEOUT	!=*
0206	0x86	0	LE	<=
0210	0x88	0	AND	conjoin conditions
0211	0x89	0	OR	disjoin conditions
0212	0x8a	0	ADD	+
0213	0x8b	0	SUB	-
0214	0x8c	0	MUL	*
0215	0x8d	0	DIV	/
0216	0x8e	0	BYHEAD	head of by list in aggr function
0217	0x8f	0	AGHEAD	head of aggregate list in aggr (function)
0220	0x90	0	CONCAT	concatenate strings
0221	0x91	0	MOD	%
0222	0x92	0	QUALDOM	
0223	0x93	0	ORDERDOM	
0224	0x94	0	CNVTANAME	
0225	0x95	0	LOUT	*=
0226	0x96	0	ROUT	=*
0227	0x97	0	LGTOUT	*>
0230	0x98	0	RGTOUT	>*
0231	0x99	0	LGEOUT	*>=
0232	0x9a	0	RGEOUT	>=*
0233	0x9b	0	LLTOUT	*<
0234	0x9c	0	RLTOUT	<=*
0235	0x9d	0	LLEOUT	*<=

0236	0x9e	0	RLEOUT	<=*
0237	0x9f	0	LNEOUT	*!=
0241	0xa1	F	RESATTR	
0242	0xa2	F	QUALATT	
0260	0xb0	1	WITH	specify param to various commands
0261	0xb1	1	ATTRALL	target list of all attributes
0263	0xb3	1	MEASURE	performance token
0264	0xb4	2	ROOT	root of query tree
0301	0xc1	0	RETRIEVE	retrieve command
0302	0xc2	0	RET_INTO	retrieve into command
0303	0xc3	0	APPEND	append command
0304	0xc4	0	DELETE	delete command
0305	0xc5	0	REPLACE	replace command
0306	0xc6	0	CREATE	create relation
0307	0xc7	0	DESTROY	destroy relation
0310	0xc8	0	INDCREATE	create index
0311	0xc9	0	INDDESTROY	destroy index
0312	0xca	0	TRUNCATE	truncate relation to zero length
0313	0xcb	0	DBCREATE	create database
0314	0xcc	0	DBDESTROY	destroy database
0315	0xcd	0	PERMIT	give permissions
0316	0xce	0	DENY	remove permissions
0317	0xcf	0	VIEW	define a view
0320	0xd0	0	ENDOFCOMMAND	this command is done
0321	0xd1	0	TUPLE	mark returned tuple
0322	0xd2	0	ABORT	abort transaction
0324	0xd4	0	BEGINXACT	begin transaction
0325	0xd5	0	ENDXACT	end transaction
0326	0xd6	0	EXTEND	extend allocation for relation
0327	0xd7	0	DBEXTEND	extend allocation for database
0330	0xd8	0	REOPEN	reopen database
0331	0xd9	0	AUDIT	audit transaction log
0332	0xda	0	AUDIT_INTO	audit xact log into ...
0333	0xdb	0	ASSOCIATE	associate text with object
0334	0xdc	0	CONFIGURE	please configure I/O
0335	0xdd	0	KILLDBIN	dba kill dbin command
0336	0xde	0	FILECREATE	create unstructured file
0340	0xe0	F	EXEC	execute a stored command
0341	0xe1	F	DEFINE	define a stored command
0342	0xe2	F	DBOPEN	open a database
0343	0xe3	F	RANGE	declare range variable
0344	0xe4	F	DUMPDB	dump database
0345	0xe5	F	LOADDB	load database
0346	0xe6	F	NEWPWORD	new password
0350	0xe8	F	FILEOPEN	open unstructured file
0352	0xea	F	ROLLFORWARD	roll forward database from xact log
0353	0xeb	F	DUMPXACT	dump transaction log
0354	0xec	F	COPYIN	copy relation in
0355	0xed	F	COPYOUT	copy relation out
0356	0xee	F	DEFINEP	define stored program
0357	0xef	F	LOADXACT	load transaction log
0360	0xf0	1	FILECLOSE	close a file

0361	0xf1	1	FILEREAD	read a file
0362	0xf2	1	FILEWRITE	write a file
0363	0xf3	1	FILEEOF	write and truncate a file
0364	0xf4	2	TRACE	turn on trace information
0365	0xf5	2	TRACEOFF	turn off trace information
0370	0xf8	4	EXECP	execute stored program
0371	0xf9	4	SETDATE	set current date
0372	0xfa	4	SETTIME	set current time
0373	0xfb	4	PLAN	decomposition plan
0375	0xfd	8	DONE	done packet

NAME

IDONE — IDM DONE token

SYNOPSIS

```
#include <idmdone.h>

typedef struct
{
    short   id_stat;    /* status bits, see below */
    short   id_int;     /* defined by the command */
    long    id_count;   /* # of tuples or blocks affected */
} IDONE;

#define IDNULL  ((IDONE *) NULL)

/* bit values for id_stat; see SPM for details */
#define ID_CONTINUE     0000001 /* more results are available */
#define ID_ERROR        0000002 /* an error occurred in processing */
#define ID_INTERRUPT    0000004 /* the command was interrupted */
#define ID_ABORT        0000010 /* xact abort, typically deadlock */
#define ID_COUNT        0000020 /* the count field is valid */
#define ID_OVERFLOW     0000040 /* overflow detected */
#define ID_DIVIDE       0000100 /* divide by zero detected */
#define ID_DUP          0000200 /* duplicates encountered */
#define ID_TIMER        0000400 /* opt 5 or 11: id_int is wallclock */
#define ID_INXACT       0001000 /* currently in a transaction */
#define ID_ROUND        0002000 /* rounding occurred on BCDFLT */
#define ID_UNDERFLOW    0004000 /* exponent underflow on BCDFLT */
#define ID_BADBCD       0010000 /* illegal BCD(FLT) sent by host */
#define ID_TMINUTES     0020000 /* id_int is in minutes */
#define ID_LOGOFF       0040000 /* please log off */
#define ID_VOLUME       0100000 /* current volume exhausted */
```

DESCRIPTION

The IDONE structure represents the IDM DONE token, as described in the *System Programmer's Manual*. This structure is read using *igetdone*(3I).

The *ie_donemask* field of the environment (see *ienv*(5I)) contains a mask for *id_stat*; bits that match between these two fields have exceptions raised by *igetdone*.

The symbolic names of done bits used by *iecontrol*(3I) and *ieopen*(3I) are identical to the constants listed above with the "ID_ " stripped off. For example *CONTINUE* is the name to pass when setting the environment's done mask (see *symfile*(5I)).

SEE ALSO

iecontrol(3I), igetdone(3I), ienv(5I), symfile(5I), *System Programmer's Manual* .

NAME

IENV, DefEnv — IDM environment

SYNOPSIS

Note: field names and layout of this structure are not guaranteed.

```
#include <idmenv.h>

typedef struct
{
        IENV    *ie_parent;   /* parent environment for inheritance */
        short   ie_donemask;  /* igetdone status mask */
        short   ie_flags;     /* flag bits, see below */
        short   ie_rtstamp;   /* timestamp in range table */
        BYTE    *ie_rtab;     /* range table */
        BYTE    *ie_subst;    /* substitution table */
        BYTE    *ie_options;  /* options table */
} IENV;

#define IENVNULL                ((IENV *) NULL)

/* flag bits values for ie_flags */
#define IEF_FOLDCASE            1       /* fold case in character arguments */
#define IEF_NOMAPCC             2       /* do not map control chars in tupprint */

extern IENV     *DefEnv;        /* default environment */
```

DESCRIPTION

Many IDM operations are performed in a particular environment. This environment contains:

- A pointer to the parent environment. The default environment *DefEnv* has no parent.
- The mask of bits from the IDONE *id_stat* (status) field that will have associated exceptions raised automatically by *igetdone*(3I).
- Assorted flags. If IEF_FOLDCASE is set, character string arguments to routines creating IDM trees have upper case letters mapped to lower case for systems that prefer to consider them semantically equivalent. If IEF_NOMAPCC is set, *tupprint*(3I) will pass control characters through unchanged.
- A time stamp for the range table, used internally.
- The range variables that have been declared. Set by *idlparse*(3I) immediately when a **range** statement is parsed.
- The current substitution values. Set by *iesubst*(3I), used by *iputtree*(3I).
- The set of options that will be attached by default to each tree. Modified by *idlparse*(3I) immediately when a **set** or **unset** is parsed.

If a value is not found when an environment is searched for a substitution variable or a range declaration, the parent environment will be searched. If that fails, the parent's parent will be searched, and so on, recursively.

Most routines requiring an environment parameter will accept the constant IENVNULL as the *env* parameter to mean the *DefEnv* environment. On initialization, this is set to a special static environment that has no parent and can never be deallocated.

Range, option, and substitution tables are created as needed. The format of these tables is internal to IDMLIB.

SEE ALSO

ieopen(3I), iesubst(3I)

NAME

IODEFS — Input/output flag definitions

SYNOPSIS

#include <idmiodefs.h>

DESCRIPTION

Definitions for flag bits (available using the **getflags** control to *ifcontrol*(3I)) contain file status. These flags are accessible for the convenience of extremely sophisticated applications and are not guaranteed to be available in this form in future releases. The flags are:

IFF_READ	This file is enabled for reading.
IFF_WRITE	This file is enabled for writing.
IFF_APPEND	This file is enabled for appending.
IFF_PRBF	This file is physically record-based.

SEE ALSO

ifcontrol(3I)

NAME

ITLIST — IDM target list descriptor

SYNOPSIS

```
#include <idmtlist.h>

typedef struct
{
        ITLIST      *itl_next;              /* next target in list */
        short       itl_type;               /* type of data */
        short       itl_len;                /* actual length of data */
        short       itl_alloc;              /* number of bytes allocated for data */
        short       itl_flags;              /* flag bits; see below */
        ANYTYPE     *itl_valp;              /* pointer to value buffer */
        char        *itl_name;              /* name of this domain */
        union                               /* application dependent info */
        {
            struct                          /* level 3 (runtime) binding info */
            {
                short       itlb_type;      /* type of prog lang var */
                short       itlb_len;       /* length of prog lang var */
                ANYTYPE     *itlb_addr;     /* address of prog lang var */
            }   itl_binding;
            struct                          /* IDL print format info */
            {
                short       itlp_width;     /* print field width */
                short       itlp_prec;      /* precision */
                char        itlp_fmt;       /* print format */
                char        *itlp_pic;      /* edit picture */
            }   itl_print;
        }   itl_un;
} ITLIST;

#define ITLNULL         ((ITLIST *) NULL)

/* some macros to simplify access of nested fields */
#define itl_btype       itl_un.itl_binding.itlb_type
#define itl_blen        itl_un.itl_binding.itlb_len
#define itl_baddr       itl_un.itl_binding.itlb_addr

#define itl_pfmt        itl_un.itl_print.itlp_fmt
#define itl_pwidth      itl_un.itl_print.itlp_width
#define itl_pprec       itl_un.itl_print.itlp_prec
#define itl_ppic        itl_un.itl_print.itlp_pic

/* bit values for itl_flags */
#define ITL_BOUND       0000001  /* binding info is present */
#define ITL_IGNORE      0000002  /* ignore in iputtl & iputtup */
#define ITL_PRINTABLE   0000004  /* tupprint info present */
```

DESCRIPTION

The ITLIST data structure holds tuple data retrieved from the database server. The fields are:

itl_next — The pointer to the next element of a target list.

itl_type — The type of the data.

itl_len The length of the data actually stored.

itl_alloc The length of the space allocated to store the data; this represents the maximum length of a field.

itl_flags Flag bits.

itl_valp A pointer to the buffer used to hold the value.

itl_name The name of this domain, if known.

The union field is used by various applications as necessary. In particular, the level three IDMLIB interface uses it to store binding information from *irbind*(3I); *idl*(1I) uses it to store print format information.

Target lists are built with *igettl*(3I) and freed with *itlfree*.

SEE ALSO

idl(1I), igettl(3I), igettup(3I), iputtl(3I), iputtup(3I), irbind(3I)

NAME

ITREE — IDM tree data structure

SYNOPSIS

```
#include <idmtree.h>

typedef struct
{
    ITREE     *it_left;   /* left child pointer */
    ITREE     *it_right;  /* right child pointer */
    short      it_type;   /* type of this node */
    short      it_len;    /* length in bytes of itval */
    ANYTYPE    it_val;    /* variable length value */
} ITREE;

#define ITNULL  ((ITREE *) NULL)
```

DESCRIPTION

ITREEs represent query trees destined for Britton Lee's IDM/RDBMS software.

Each node contains a left and right pointer. These fields normally implement child pointers, although in some cases one or the other may be used as a sibling pointer.

The node type is typically the same type as will be passed to the database server. Nodes with no correspondence in IDM/RDBMS are assigned values greater than 255. The values zero and 255 are reserved for use by the host.

The value field is variable length. The length is explicitly specified in the ***it_len*** field. The value always abuts the remainder of the node; a pointer to the value is never used.

A special node is the iNOTOKEN node. This node is always zero length, and essentially has "no type" — it is used as a placeholder. During tree walks in *iputtree*(3I), iNOTOKEN nodes are ignored completely, except for their pointers.

A query tree for a complete command always has a iCOMMAND node as the root. This node contains status and flag information about the command. The left child must be a command node.

The left child of the command node is the query tree as described in the ***System Programmer's Manual*** for most commands; commands that do not accept query trees either leave this node null or have a pseudo tree that describe any additional parameters.

The right child of the command node is a right-linked list of control information. The first element is the range table, the second element the order table, and the third element the options clause. Other elements may be added at Britton Lee's discretion as needed.

The iCOMMAND node has an eight byte value. The first two bytes are used for status flags. The remainder of the node is reserved for use by Britton Lee.

SEE ALSO

itnode(3I), iputtree(3I), itlist(5I)

NAME

messages — messages file format

DESCRIPTION

The ***messages*** file contains the text for all message codes used by IDMLIB. This section describes the format of the master file as distributed by Britton Lee. This format will generally be massaged by an implementation-dependent program (e.g., ***buildmsgs***(8I)) into a form that can be read efficiently by the *IftMText*(4I) module.

Each line of the file begins with a code that gives the semantics of the rest of the line. These are:

$msg.code	This line introduces an entry for the named message code.
@n name	Indicates the semantics of parameter *n*. This is used to prepare documentation.
code<*tab*>text	Text for this message. *Code* is a set of one or more letters or blanks. These letters specify conditions for display of the line at execution time. The letters specify the experience levels: B=Beginner, A=Able, E=Expert, and the query language being used: I=IDL, S=SQL. The code and the text are separated by a tab character, which will be stripped from the message before printing. Note that the default for the query language specifier is SI — display the line to both query languages. However, the default for the experience level is null — hide the line from all users.

Any other lines should be ignored.

Within the text, parameters are substituted using *%n*. For example, "%2" should substitute the second parameter.

Macros are substituted using a *?c* syntax. For example, "?S" in the text of a message will be substituted by its definition when ***buildmsgs*** is run. Definitions are contained in the file messages.mac (in etc/ on UNIX systems) and should be configured at each site to indicate the correct local person to report problems to.

Each message should contain a single line message that is sufficiently rich as to be adequate to be understood by most users. The remainder of the message should be explanatory information for beginning users. These should include the following subheadings, as appropriate: Explanation (of the message), System Action (what happened to your job), and User Action (what the user should do next).

The same syntax is used to store the help file. In the help file, lines of the form *code*<*tab*>*$help-command* represent a default command to be executed if the user enters a blank line.

IMPLEMENTATION NOTES

The master version of this file will be supplied by Britton Lee. The OEM will be responsible for writing a program to convert from the standard format into the format needed by the local system. See ***buildmsgs***(8I) for details.

EXAMPLE

```
$IDM.E19
@1 <domname>
BAE I   Result for attribute: %1 has wrong type.
BAE S   Result for column: %1 has wrong type.
BA  I   Explanation: The tuple was invalid because the value
BA  S   Explanation: The row was invalid because the value
BA  I                specified for the attribute %1 was
BA  S                specified for the column %1 was
```

BA of the wrong type.
B I User Action: Determine the actual type of the attribute,
B S User Action: Determine the actual type of the column,
B and correct the query.

If the user has an experience level of ABLE, and is running IDL, the following message should be seen:

Result for attribute: esalary has wrong type.
Explanation: The tuple was invalid because the value specified for the attribute esalary was of the wrong type.

SEE ALSO

iftmtext(4I), buildmsgs(8I), *IBM OS/360: Messages and Codes* for an excellent example.

NAME

/usr/lib/idm/params — default *getparam*(3I) parameter file

DESCRIPTION

The "params" file contains the default settings for all system parameters. On UNIX, this file is structured as a series of lines of the form

name=[=]value

No spaces are allowed in the line unless they are part of the value. By convention, the name is in upper case. If the second "=" is present, the name will not be imported from the UNIX environment (see *getenv*(3)).

For the UNIX system, the required entries are:

EDITOR The pathname of the system editor to use.

EPOCHOFFSET This is used to offset the beginning of the "epoch" for date and time routines. The default epoch is January 1, 1900. EPOCHOFFSET is given as a number of days from January 1, 1900. For instance, to change the epoch to January 1, 1901, change EPOCHOFFSET to 365. In general, the use of this parameter is discouraged.

EXPERIENCE The default experience level, chosen from the set **Beginner**, **Able**, and **Expert**. Normally **Beginner**.

FOLDCASE Perform upper to lower case folding if set to 1. UNIX command lines are currently not folded.

GETHUNPW Constant 1 if shared database system password processing is desired, otherwise 0. This should match the "untrustworthy" bit for this host in the "configure" relation. The user will be prompted for a password on "permission denied" messages on the first open database if this is set.

HELPFILE The location of the help file; see *messages*(5I).

IDMBAUD The baud rate for serial connections. These are used directly in the *stty*(2) call; for example, 13 means 9600 baud. See *stty*(2) and *tty*(4) for details.

IDMDEV The default device name for database server connections; normally "/dev/idm". On UNIX, the "/dev/" part is optional. For convenience the driver may also be specified in IDMDEV using the *filespec* syntax of *device%driver* (see *intro*(1I)). For example, an IDMDEV set to "idm%0" or "idm%multi" specifies the "system standard" driver and device "idm" ("/dev/idm"). IDMDRIVER is used if *driver* is not specified. Other values accepted are "idm%stand" (idm%1), "idm%xns" (idm%2) and "idm%tcp" (idm%4).

corwin %xns works

IDMDRIVER The offset into the IDM driver table for the low-level interface. Driver 0 is always the "system standard" driver. Driver 1 is normally the standalone serial driver. Other drivers are typically used for experimental protocols. This value must be an integer.

IDMHOSTID The host id to use for the standalone commands.

IDMHUNAME The user name to be passed to the IDM/RDBMS software for identification. If the value is null, no user name is known.

IDMPASSWD The IDM/RDBMS password for this user.

IDMPKTSIZE The size of communication packets to the database server for standalone serial connections. If your line is flakey or if your UNIX system has a small line-length limitation, this can be adjusted.

IDMSERROR	If nonzero, simulates a flakey line for protocol testing. Should always be zero.
IDMSYSCALL	The system call number used to access the database server. This must match the entry in the kernel *sysent* or *vmsysent* table.
IDMSYSLINE	Since the multi-user serial driver is no longer supported, this parameter is not used. It used to represent the line discipline for the multi-user serial driver. It *had* to match the installed line discipline in the kernel.
IDMUSER	The (numeric) user id to use for the standalone IDM drivers.
IDMVERSION	The version of IDM/RDBMS you are running. The minimum version is 30. The version configures in features supported by newer IDM/RDBMS versions.
IOBSIZE	The default I/O buffer size.
ISDST	Constant 1 if daylight savings time ever applies in this area, otherwise 0.
MAPCC	Map control characters to blanks (input) and blotch (output) characters if set to 1. When cleared, control characters are passed through unchanged. Currently used by the IDL and SQL front ends and tupprint to allow terminals to switch character sets to Kangi.
MESSAGES	A comma-separated list of files containing messages; see *messages*(5I). The files are searched in order by *excprint* (see *exc*(3I)). Changing this parameter after the first message is output has no effect.
NOPROFILE	Disable reading of profile (or startup) files in IDL or SQL if set to 1. See *idl*(1I) and *sql*(1I) for the command-line **-p** (noprofile) flag.
QRYLANG	The query language normally used, either IDL or SQL. The setting of this flag changes the wording of messages. It in no way limits the query languages that may be used.
SHELL	The pathname of the system shell to use.
SYMFILE	The location of the symbol file; see *symfile*(5I).
TERM	The terminal type.
TERMPATH	On UNIX, the prefix of the pathname (with TERM concatentated) containing a terminal descriptor as created by *maketerm*(8I). Normally "/usr/lib/idm/term".
TIMEZONE	The local time zone in chronological minutes westward from GMT. Negative values are minutes eastward from GMT. The maximum (absolute) value is ±720 (minutes), representing the time in Western Samoa.

In most cases the parameter is only examined once, so any adjustments should be made early in processing.

Other parameters may be required by particular implementations.

SEE ALSO

gethunpw(3I), getparam(3I), iftterm(4I), messages(5I), symfile(5I), maketerm(8I), csh(1), sh(1), stty(2), getenv(3), tty(4), *System Administrator's Manual* .

NAME

retcode — return/status/error code

SYNOPSIS

#include <machdep.h>

DESCRIPTION

Type *RETCODE* is used by IDMLIB routines that return a status code, for status returns from programs, and for system service error codes.

The following codes are defined in all environments. They are of the form R*x*_*code*, where *x* is S, W, or E for success, warning, and error respectively. Codes marked with an 'R' are returned by normal IDMLIB routines; codes marked with an 'E' are returned by *errclass* (see *geterr*(3I)).

RS_NORM	E	R	Normal return
RW_DONECMDS		R	There are no more commands during an *irnext*(3I).
RW_IGNORED		R	This request was ignored because it would have no effect, e.g., setting an option that was already set.
RW_NOTUPS		R	No tuples available.
RW_PSEUDO		R	Tree represents pseudo-command.
RW_TARGEND		R	Target list exhausted.
RW_TRUNCATE		R	Data truncation occured.
RW_TUPEND		R	No more tuples.
RE_CANT	E		Impossible operation requested, e.g., write on a read-only file.
RE_FAILURE		R	An error occured; an exception will have been raised giving more information.
RE_IDMQRY	E		IDM query error occured.
RE_INTR	E		Program or routine was interrupted.
RE_IOERR	E		Hard I/O error.
RE_MISC	E		A miscellaneous (unclassifiable) error has occured.
RE_NOSPACE	E		Write failed because of lack of space.
RE_NOOUTPUT	E		Cannot create output.
RE_NOINPUT	E		Cannot open input.
RE_PERM	E		You do not have permission to perform this operation.
RE_USAGE	E		Bad arguments or parameters.

This list will be expanded as necessary in the future.

IMPLEMENTATION NOTES

Type RETCODE and R*x*_ codes are defined in <*machdep.h*>. Codes should match operating system conventions if possible.

The code RS_INFO exists and is identical to RS_NORM. It is intended for use with *sysshell*(3I) so that VMS commands run by *sysshell* that return STS$K_INFO can return that value as the exit value of an IDMLIB application.

On VMS, RETCODEs are VMS condition codes. In addition to the codes listed above, there are several codes returned by IDM drivers. The numerical value of all of these codes may be found in the <*retcode.h*> include file.

All IDMLIB RETCODEs have associated messages defined by the VMS Message Utility.

SEE ALSO

intro(3I), exit(3I), geterr(3I)

NAME

/usr/lib/idm/symfile — symbol to integer value mapping file

DESCRIPTION

The symbol file contains the information to map symbols to integers. The format is the symbol name, one or more space or tab characters, and the integer value. Every symbol must begin in the first position of the line, and there may be only one symbol per line. Comments may be added after the value, separated by more white space, or may be on a line by themselves beginning with '#'.

The first character of each symbol is a tag indicating the class of symbol. Assigned tags are:

- d IDM done status bits.
- o IDM option names.
- t Host trace flags.
- * IDM trace flags.

Uppercase alphabetic tags are reserved for use by the customer. All other characters are reserved for use by Britton Lee.

The symbols input to *mapsym*(3I) are converted to uppercase before matching (except for the tag character). Thus, symbols containing lowercase characters will never match.

Syntax errors are silently ignored.

EXAMPLE

Note: this is an example only. It does not match the actual values used in the system.

```
# IDMLIB basic flags (50-59)
tILIBGEN        50      /* general utility routines */
tILIBEXC        52      /* exception handler */
tILIBBCD        53      /* BCD routines */
tILIBCNVT       54      /* type conversion module */
tILIBOS         55      /* host O/S interface (except I/O) */
tILIBCLOCK      56      /* clock routines */

# IDM-specific modules (60-65)
tIDMTREE        60      /* print tree */
tIDMGEN         61      /* general utility routines */
tIDMCNVT        62      /* type conversion */
tIDMPARSER      63      /* parser, scanner and tables */
tIDMUTREE       64      /* UTREE routines */
tIDMRANGE       65      /* range variables */

# RUNTIME-specific modules (66-69)
tRUNTREE        66      /* print tree support */
tRUNTL          67      /* print target lists */
tRUNGEN         69      /* general tracing */

# I/O subsystems (70-79)
tILIBIO         70      /* basic I/O calls */
tIFTLTAPE       71      /* labeled tape */
tIFTHASH        72      /* hash file type */
tIFTSIDM        73      /* standalone IDM access */
```

SEE ALSO

atof(3I) (for atoi), mapsym(3I), idone(5I)

NAME

/usr/lib/idm/xnshosts — XNS host name mapping file

DESCRIPTION

Xnshosts specifies the numeric addresses used for particular symbolic names. The format of this file is

physical_address logical_name [*alias* ...]

The physical address is in the form:

n1.n2.n3.n4:h1.h2.h3.h4.h5.h6

where *n1* through *n4* are the four nibbles of the network number and *h1* through *h6* are the six nibbles of the host number.

The *logical_name* or any of the optional *aliases* may be used to identify the host.

EXAMPLE

```
0.0.0.1:8.0.44.0.0.8        host idm
0.0.0.1:8.0.44.0.0.2        p3
0.0.0.1:8.0.44.0.0.1        p3spy
0.0.0.1:8.0.44.0.0.10       p7
0.0.0.1:8.0.44.60.186.252   tsa
0.0.0.1:8.0.44.74.184.20    tsb
```

NAME

Introduction to Administrative and Machine-Dependent Commands and Procedures

DESCRIPTION

Section 8I describes commands and procedures used in release administration. This section is *not* part of the spec. Commands described herein are not guaranteed to be supported on non-UNIX based systems. Several of the commands are for Britton-Lee internal use only.

N.B.: Most of these pages are UNIX-dependent.

NAME

ansitape – write files on an ANSI standard labelled tape

SYNOPSIS

ansitape [**–f** files] [**–t** tapespec]

ARGUMENTS

–ffiles	A file containing a list of files to write to labelled tape. *Files* will be written as the first file on the tape. If not specified standard input is read for the list of files.
–ttapespec	Labelled tape parameters for use when opening tape. See *iftltape*(8I) for a list of tape parameters.

DESCRIPTION

Ansitape writes files listed in the file *files* or from standard input to an initialized labelled tape (see *inittape*(8I)) using a fileset name of Write mode, record based presentation of 512 byte records blocked every 2048 bytes are the default *iftltape*(8I) open parameters. Tape parameters passed in via *tapespec* will override the default values.

Under record based presentation, *ansitape* will read and write one line of data from the host file to the tape file. Otherwise 8K blocks are read/written from host to tape file.

EXAMPLE

inittape –f listoffiles –t"mode(a),rbp(0)"

Writes files listed in *listoffiles* at the end of the tape without record based presentation (in 8K blocks).

IMPLEMENTATION NOTES

This module is UNIX-specific. Systems that support ANSI tape will have another module to perform this function.

SEE ALSO

iftltape(4I), inittape(8I), ANSI X3.27-1978, *American National Standard Magnetic Tape Labels and File Structure for Information Interchange.*

NAME

backup – Shared database system backup procedures using idmdump, idmload, and idmrollf

DESCRIPTION

Databases should be copied ("backed up" or "dumped") periodically to guard against the unnecessary loss of data due to database server disk crashes or failures. A database can be backed up to an IDM file in a different database, an IDM tape, a host file, or a host tape. The procedures are similar for all cases.

As databases are accessed and modified a "transaction log" is maintained. The transaction log contains information describing all the changes made to the database on relations created "with logging". The transaction log does not contain information on non-logged relations or on files. If you have a copy of the database at some point and a transaction log describing all the changes made since that point, you can recreate the contents of the database that were active at the end of the transaction log.

The transaction log is interesting since it is normally much smaller than the database itself. Obviously, if the transaction log is allowed to grow forever, it will eventually become larger than the database.

The program *idmdump*(1I) will dump either entire databases or transaction logs. *Idmload*(1I) will load a database or a transaction log. *Idmrollf*(1I) will "roll forward" (that is, make the changes specified by a transaction log) a database.

Databases should normally be dumped in toto periodically. The frequency of your dumps depends on how much the database is updated. For example, a database that is updated frequently should probably be dumped every day. A database that is only updated occasionally could only be dumped once per month. An average database should probably have a full dump once a week. A database dump for RDBMS software before release 35 requires that all users stop using the database while it is being dumped, so backups should be scheduled for off hours. RDBMS release 35 and above defaults to allowing users to read a database that is being dumped.

Transaction dumps should occur more frequently. For example, if you dump your entire database once a week, you might want to dump your transaction log at least once every day. The frequency of the transaction log is critical: if you only dump the transaction log once per week, you may lose up to a week's worth of work if the database server fails. If you dump the transaction log once per hour but only dump the database once per month, then a crash at the end of the month may require loading the database and then over seven hundred transaction logs (31 days/month times 24 hours/day = 744 transaction dumps/month). A good rule is to dump the database once for every three to ten transaction dumps.

Loading a transaction log is not useful by itself; the transaction must be "applied to the database" for the changes to occur. That is, the **roll forward** utility must read a transaction log that has been loaded and make all the changes indicated. This is the same as asking all your users to make all the changes they have made, but much less painful. The *idmrollf*(1I) program will perform this operation for you.

Dumping to another database has some good points as well as some drawbacks. On the negative side, if a hardware failure destroys the entire disk you will have lost your database regardless of the dump going into another backup database. On the positive side, the dumps will be very fast, and the roll forward operation can happen without requiring an *idmload* first.

For details on developing a complete backup strategy, see the *Database Administrator's Manual* .

SEE ALSO

idmdump(1I), idmload(1I), idmrollf(1I).

NAME

buildmsgs – build keyed message text file

SYNOPSIS

buildmsgs [**–s**] [**–h**] [**–a**] [**–l** length] outfile infiles

ARGUMENTS

–h Key ("hash") the output file instead of outputting text.

–l*length* Specify the length of the hash table, i.e., the number of hash buckets. The default is 512. For efficiency, the hash table should be about 30 percent larger than the number of keys. Ignored if the *IftKeyed*(4I) implementation does not require a table size.

–a Append new materal to *outfile* rather than creating a new file.

–s Create "subtopic" lists from the keys in the input file.

DESCRIPTION

Buildmsgs reads the files in the list *infiles* and creates *outfile*. The **–h** flag causes *outfile* to be a keyed file accessible using the *IftKeyed*(4I) module; otherwise it is a text file. The **–a** flag specifies that output will be appended to *outfile* if it already exists; otherwise *outfile* will always be created as a new file. Input lines beginning with '$' are interpreted as index keys. Lines beginning with a mask-code are text associated with the most recent key. (A mask code is one or more upper-case letters and one or more blank, followed by a tab.) All other input lines are ignored.

The **–s** flag builds subtopic lists for use by the help facility. This may be combined with the **–h** flag.

EXAMPLES

buildmsgs –h –l 1024 messages.uvax messages.txt
Create a keyed file named "messages.uvax" from the text file "messages.txt". The length of the hash table is set to 1024.

buildmsgs –h –s helpfile.uvax helpfile.t1 helpfile.t2
Create a keyed help file named "helpfile.uvax" from the text files "helpfile.t1" and "helpfile.t2".

SEE ALSO

iftkeyed(4i), iftmtext(4i), messages(5i)

NAME

dumptape – report on contents of an ANSI tape

SYNOPSIS

dumptape [**–r**] [**–v**] [**–t** tapefile]

ARGUMENTS

–r	Raw dump mode. Every tape record is dumped in abtruse detail.
–v	Verbose mode. Gives even more detail.
–ttapefile	The name of the UNIX device to reference; **/dev/rmt8** (4.2 BSD) by default.

NOTE

System V Release 2.0 (running on 3B series) does not provide access to basic tape operations. Therefore support of ANSI labeled tape is unavailable at this time.

DESCRIPTION

Without **–r** specified, ***dumptape*** produces a report of the tape contents in a one line per file format. The **–v** flag adds several fields; this format is suitable for output on a line printer. The fields output are:

SEQN	Sequence number of the file on the tape.
—FILE-NAME—	File name.
SECT	File section number. A multivolume file will be in several sections.
GEN#	Generation number.
GV	Generation version number.
CDATE	Creation date.
XDATE	Expiration date.
A	Access code.
-SYSTEM-CODE-	System code for the system that created the file.
F	Format.
BSIZE	Maximum block size.
RSIZE	Maximum record size.
BO	Buffer offset.
BLOCKS	Number of blocks in the file. This is computed rather than being read from the labels.

In raw format (i.e., with the **–r** flag specified) the output is suitable for system debuggers.

SEE ALSO

iftltape(4I), inittape(8I), ANSI X3.27-1978, *American National Standard Magnetic Tape Labels and File Structure for Information Interchange.*

NAME

idmboot – load the IDM/RDBMS software

SYNOPSIS

idmboot [**–B** device] [**–V**] [**–2**] [source]

ARGUMENTS

–B *device* — Use *device* as the connection to the database server. The device must be connected to the database server console or maintenance port. If not specified, the system parameter **IDMCONS** is used.

–V — Verbose mode.

–2 — Run the older two-port load. This flag is necessary if the database server has dbp proms rev. 28 or earlier.

source — This must be a single parameter, so it will have to be quoted if it contains spaces. If not specified, the default will be the host system's default IDM/RDBMS software source. (On 4.2 BSD UNIX, this is usually "/dev/rmt8" – the 1600 BPI tape-drive.)

DESCRIPTION

Idmboot in one-port mode (the default) allows the user to access the database server's console port, issuing dse server console commands. If the user issues 'load' or 'list' commands, *idmboot* will obtain the necessary files to transmit to the database server.

The older, two-port load requires that a terminal be connected to the IDM console port. The database server connection specified by the **–B** option must be to the database server maintenance port. All console commands must be issued via the console terminal – two port *idmboot* only handles the actual transmission of files to the database server.

Idmboot does not know when the console session is finished, so the only way to terminate *idmboot* is by user interrupt.

Please refer to the *BLI 700 Operation Manual* for a description of database server console commands and their use.

EXAMPLES

idmboot

Access the database server console port specified by the system parameter IDMCONS. Any files required by the IDM system will be read from the system's default RDBMS software source (on 4.2 BSD UNIX, the 1600 BPI tape-drive "/dev/rmt8"). This is the one-port load.

idmboot –2 –V –B/dev/idmmaint /dev/rmt8

Run the two-port load in verbose mode. The database server console port is connected to the device "/dev/idmmaint." The RDBMS software is read from the 1600 BPI tape-drive "/dev/rmt8" (the default IDM/RDBMS software source on UNIX).

SEE ALSO

BL 700 Operation Manual.

NAME

idmidyd — IDM XNS identify daemon

SYNOPSIS

/usr/lib/idm/idmidyd [**–B** device] [**–h** interval] [**–p**] [idmname ...]

ARGUMENTS

–B *device* Specify the database server device connection. Ignored if any names are specified as positionals.

–h *interval* Poll every *interval* seconds. The default is set from the **XNSHELLOINT** parameter.

–p Force poll mode. This mode is assumed if more than one database server is specified.

idmname The name of one or more database servers to be controlled by this identify daemon.

DESCRIPTION

Idmidyd opens a "lifeline connection" to the named database server(s). An IDENTIFY packet is sent to establish the host characteristics, and the connection is held open.

In "poll" mode a message is sent to the database server periodically to verify that both the host and the database server are both working. If the database server fails to respond, the connection is closed and ***idmidyd*** goes into a loop trying to open the connection again.

In non-poll mode ***idmidyd*** hangs on a read on the connection. If the read ever fails then the database server must be down and ***idmidyd*** enters the loop to attempt to re-open the connection.

This program is normally started on all database servers during system startup.

IMPLEMENTATION NOTES

An implementation is supplied that is machine independent assuming that a routine:

sleep(N)

is supplied which suspends the execution of the process for *N* seconds.

NAME

inittape – initialize ANSI standard labelled tape

SYNOPSIS

inittape [**–a** access] [**–d** density] [**–i**] [**–l** length] [**–o** owner] [**–t tapefile**] volumeid

ARGUMENTS

–a*access*	The access character for this tape volume. The default is space, meaning all access for everyone. If any other character is chosen, the current implementation will refuse to access the tape in any way.
–d*density*	The tape density in bits per inch. Default is 1600 bpi.
–i	Create an initial empty file on the tape. This option is required if the tape is to be used on another system before being written.
–l*length*	The tape length in feet, 2400 default. If this is not specified properly volume switching may be defeated.
–o*owner*	The name of the owner of the tape. If not specified, the name of the user running *inittape* will be used. The owner name is truncated to fourteen characters.
–t*tapefile*	The name of the file to be opened to access the tape. The default is "/dev/rmt8."
volumeid	The name that this volume should have. The volume name is truncated to six characters.

WARNING

Use of this program can cause destruction of valuable data. Some installations may want to limit access to this command to system personnel.

System V Release 2.0 (running on 3B series) does not provide access to basic tape operations. Therefore support of ANSI labeled tape is unavailable at this time.

DESCRIPTION

Inittape initializes a tape by writing an ANSI standard label set. A tape must be initialized before using the *iftltape*(4I) module, implicit in most of the IDM utilities. Initializing a volume destroys any previous contents.

Every tape must have a volume name. This name should be unique among all tapes at your installation to insure that important data is not accidently overwritten.

The volume name should be copied onto the physical tape reel for easy identification. A good technique is to initialize and physically label all tapes as soon as they arrive at your installation.

Characters in user and volume names must be chosen from the set of letters, digits, and the special characters:

```
!  "  %  &  '  (  )  *  +  ,
-  .  /  :  ;  <  =  >  ?  space
```

Lower case letters are automatically mapped to upper case.

EXAMPLE

inittape a00452

Initializes the tape to have the label "A00452."

IMPLEMENTATION NOTES

This module is UNIX-specific. Systems that support ANSI tape will have another module to perform this function.

UNIX writes a UVL1 label containing:

CP	*Field Name*	*L*	*Content*
1 to 3	Label Identifier	3	UVL
4	Label Number	1	1
5 to 17	System Code	13	Identifies this implementation.
18 to 22	Tape Density	5	Density in bits per inch.
23 to 27	Tape Length	5	Length in feet.

BUGS-UNIX

The density must be consistent with the value of the –t flag.

Use of –l is a hack.

SEE ALSO

idmcopy(1I), idmdump(1I), iftltape(4I), mt(4), ANSI X3.27-1978, *American National Standard Magnetic Tape Labels and File Structure for Information Interchange.*

NAME

Make – clever interface to make(1)

SYNOPSIS

Make make arguments

DESCRIPTION

Make (with a capital-M) is a front end to *make*(1) which creates a Makefile from a Makefile.m4 using *m4*(1) if necessary. If the Makefile does not exist or is out of date with respect to a Makefile.m4, the command

```
m4 $IDMCONFIG Makefile.m4 > Makefile
```

is executed. IDMCONFIG may be defined in your environment to select a configuration file; if not specified, /a/host/etc/config.m4 is used.

All other arguments are exactly as described in *make*(1). The **–f** flag is not correctly processed.

SEE ALSO

make(1), m4(1)

FILES

Makefile
Makefile.m4
RCS/Makefile.m4,v
/a/host/etc/config.m4

NAME

maketerm – compile a terminal descriptor

SYNOPSIS

maketerm [**–C**] term

ARGUMENTS

–C	Create a "C" language source-file instead of a binary data file.
term	The name of a terminal type.

DESCRIPTION

Maketerm reads a terminal description text file named *term*.**tty** and creates the file *term*.**td.***N* (where *N* is the a version number on the binary format) containing a compact representation to be read by *IftTerm*(4I). *Term*.**tty** must exist in the current directory. If the **–C** argument is used, a **C** source file will be produced instead of a ".td" file. The file will be named *term*.c, and the data structure will be BYTE array named Td_*term* . It is the programmers responsibility to cast the pointer to Td_*term* to be of type (TERMDESC *). (See *IftLoTerm(4I)* for a description of the _gettermdesc interface.)

TERMINAL DESCRIPTIONS

A terminal description consists of a series of "field=value" lines. Lines beginning with a '#' mark and blank lines are comments.

Field names are:

flags	A list of terminal flags
init	Initialization string
reset	Reset string
so-g1	The G1 "shift out" (alternate char set) sequence
si-g1	The G1 "shift in" (normal char set) sequence
so	Same as "so-g1"
si	Same as "si-g1"
so-g2	The G2 "shift out" sequence
si-g2	The G2 "shift in" sequence
so-g3	The G3 "shift out" sequence
si-g3	The G3 "shift in" sequence
so-g4	The G4 "shift out" sequence
si-g4	The G4 "shift in" sequence
so-g5	The G5 "shift out" sequence
si-g5	The G5 "shift in" sequence
so-g6	The G6 "shift out" sequence
si-g6	The G6 "shift in" sequence
so-g7	The G7 "shift out" sequence
si-g7	The G7 "shift in" sequence
g2-tlc	Top Left Corner sequence
g2-trc	Top Right Corner sequence
g2-blc	Bottom Left Corner sequence
g2-brc	Bottom Right Corner sequence
g2-lt	Left Tee sequence
g2-rt	Right Tee sequence
g2-tt	Top Tee sequence
g2-bt	Bottom Tee sequence
g2-x	Cross sequence
g2-vb	Vertical Bar sequence
g2-hb	Horizontal Bar sequence
g2-blotch	Out-of-band Blotch sequence

lines	Number of lines
cols	Number of columns
e-primary	Primary enhancement string
e-bold	Bold enhancement string
e-faint	Faint enhancement string
e-italic	Italic enhancement string
e-under	Underscore enhancement string
e-blink	Blink enhancement string
e-flash	Flash enhancement string
e-reverse	Reverse enhancement string
c-cuf	move cursor right
c-cud	move cursor down
c-cuu	move cursor up
c-cub	move cursor left
c-cup	absolute cursor motion
c-clr	clear screen
c-con	start cursor-motion mode
c-coff	end cursor-motion mode
padch	padding character (if not NUL)
speed	Baudrate

The "speed" field is ignored on systems that can automatically determine the baudrate.

If a field specification is missing from the terminal descriptor file, the terminal is assumed not to have that capability.

The strings are specified using the following mappings:

\b	BS (backspace) character
\e	ESC (escape) character
\f	FF (form feed) character
\i	SI (shift in) character
\n	NL (newline) character
\o	SO (shift out) character
\r	CR (carriage return) character
\\	backslash character
^*x*	Control-*x*
NNN	The octal representation

Arguments

Arguments are indicated by a "%" character, and a literal "%" may be specified by "%%". In most strings, the only recognized argument is for padding. Padding is specified as follows:

%*n* p

where *n* is a decimal integer represented by a string of digits. Actual padding times are calculated at run-time relative to the baudrate of the terminal. For most control strings, padding is absolute. For absolute cursor motion, the padding specified is for each line affected. That is, if the cursor is moved down 6 lines, the padding value will be multiplied by 6.

There are two other arguments recognized in the "c-cup" (absolute cursor motion) control string. These are the line and column to position to. The format of the argument specifications is as follows:

%*c* [*o*][*w*]*t*

where the meta-characters have the following special meanings: *c* is either "x", specifying that the argument is the column, or "y" for the line. *o* (optional) is an offset to be added to the line or column number, and is in the format of a decimal integer string followed by a "+" or a "-". *w* (optional) is a decimal integer indicating the width of the argument, in bytes. *t* is either "b", specifying that the argument is to be interpolated as a binary byte, or "d", specifying decimal digits.

As an example, the "c-cup" string for an adm3a would be as follows:

```
\e=%y31+b%x31+b
```

The following specifications all work for the Concept avt:

```
\e[%yd;%xdH
\e[%y2d;%x2dH
\e[%y0+2d;%x0+2dH
```

Graphics

The graphic characters have a single character which may be a single-quoted character or an integer representation followed by a series of flags:

g1	Terminal must be in G1 mode
so	Same as "g1"
g2	Terminal must be in G2 mode
g3	Terminal must be in G3 mode
g4	Terminal must be in G4 mode
g5	Terminal must be in G5 mode
g6	Terminal must be in G6 mode
g7	Terminal must be in G7 mode

G4 mode is reserved for Katakana mode.

In general, the terminal is normally in normal (G0) mode. When a special graphic is printed, it is shifted into the mode specified by the terminal descriptor and then the specified translation is printed. (Katakana characters are always shifted into G4 mode and are passed through untranslated.) For example, the description "g2-tlc = 54 g6" would cause the terminal to be shifted into G6 mode (as specified by the "so-g6" string), the byte with value 54 to be sent, followed by the appropriate shift-in string ("si-g6").

EXAMPLE

```
#
#    Descriptor for VT100 terminal
#

flags=          fancy
init=           \e)0
reset=          ^X\i\e[0m
so-g1=          \o
si-g1=          \i

g2-tlc=         108 so
g2-trc=         107 so
g2-blc=         109 so
g2-brc=         106 so
g2-lt=          116 so
g2-rt=          117 so
g2-tt=          119 so
g2-bt=          118 so
```

```
g2-x=           110 so
g2-vb=          120 so
g2-hb=          113 so
g2-blotch=      097 so

lines=          24
cols=           80

e-primary=      \e[0m
e-bold=         \e[1m
e-faint=        \e[2m
e-italic=       \e[3m
e-under=        \e[4m
e-blink=        \e[5m
e-flash=        \e[6m
e-reverse=      \e[7m

c-cuf=          %2p\e[C
c-cud=          %2p\e[B
c-cuu=          %2p\e[A
c-cub=          %2p\e[D
c-cup=          %5p\e[%yd;%xdH
c-clr=          %50p\e[1;1H\e[J
```

SEE ALSO

iftterm(4I)

NAME

sgrep – structured grep

SYNOPSIS

sgrep [**–o** output-spec] [**–c** comment-char] [**–t** tab-char] [**–d** keyword=default] selection-criteria

ARGUMENTS

–o *output-spec* Set the output specification.

–c *comment-char*
Set the comment character; "#" by default.

–t *tab-char* Set the character to be used to separate fields; comma by default.

–d *keyword=default*
Set a default value for a field name.

DESCRIPTION

Sgrep selects lines from the standard input and copies them to the standard output under control of the *output spec* and the *selection criteria.*

The input is structured as a set of "keyword=value" pairs separated by "tab characters" (comma by default). There is no implied ordering of fields on a line. Alternative values can be separated by vertical bars. For example, the input line

file=Makefile.m4, type=base|ext

will match selection criteria matching either "type=base" or "type=ext."

Lines are selected by a series of criteria of the form "keyword=pattern" where *pattern* is a list of alternatives separated by vertical bars or is null (to match any line that has that keyword present). Criteria may be combined using **and, or,** and **not;** the expression must be a disjunction of conjunctions. For example, the criteria:

type=base and ver=2 or type=ext

will select all lines where the **type** field is "base" and the **ver** field is "2" *or* where the **type** field is "ext."

If an output spec is given, selected lines are formated. Characters are copied from the output spec to the standard output except for field names enclosed in braces ("{ }"). For example, the output spec:

–o'co –r{ver} {file}'

will output a series of *RCS* commands that can in turn be input to the shell. Some special field names are supplied by *sgrep.* "{$input$}" is the input line as read; this is the default output spec. "{$lineno$}" is the line number of the input. For example, to get a numbered list of all lines that match, use:

–o'{$lineno$}: {$input$}'

It is an error to specify a field name in an output spec that is not in the input line. However, defaults can be specified in the command line using the **–d** flag.

Lines in the input beginning with the *comment-char* ("#" by default) are ignored.

SEE ALSO

grep(1)

Britton
Lee, Inc.
14600 Winchester Boulevard
Los Gatos, California 95030
(408) 378-7000
Telex: 172-585