

SIBO 'C' Software Development Kit

ISAM REFERENCE

Version 2.10

February 3, 1995

(C) Copyright Psion PLC 1990-95

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilization in machines capable of reproduction or retrieval, without express written permission of Psion PLC, is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion, Psion MC, Psion HC, Psion Series 3, Psion Series 3a and Psion *Workabout* are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software Corporation. IBM, IBM XT and IBM AT are registered trademarks of International Business Machines Corp. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Apple and Macintosh are registered trademarks of Apple Computer Inc. VAX and VMS are registered trademarks of Digital Equipment Corporation. Brief is a registered trademark of Underware Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

Contents

1 Introduction	1
Overview	1
The ISAM library	1
DBF files	2
Fields	2
Number of records	2
B-tree index files	2
Block buffering	2
Key description	2
Size of B-tree files	3
Interface to the ISAM library	3
Loading the DYL	3
Creating an ISAM object	4
Calling ISAM functions	4
Destroying the ISAM object	4
Typical use	4
Example programs	5
Building a complete (dense) index	5
Building a selective (sparse) index	7
2 Isam Functions	9
General functions	9
Initialise an ISAM object (O_IS_INIT)	9
Destroy an ISAM object (O_IS_DESTROY)	10
Set the field definition (O_IS_SET_FIELDDEF)	10
Get the field definition (O_IS_GET_FIELDDEF)	11
Set the key definition (O_IS_SET_KEYDEF)	11
Get the key definition (O_IS_GET_KEYDEF)	13
Put a value into a field (O_IS_PUT_FIELD)	13
Get a value from a field (O_IS_GET_FIELD)	13
Get the type of a field (O_IS_GET_TYPE)	14
Get the address of the record buffer (O_IS_GET_RBUF)	14
Set the radix for number conversion (O_IS_SET_RADIX)	14
Set the text format for DOUBLES (O_IS_SET_RADIX)	14
Add an entry (O_IS_ADD)	14
Erase an entry (O_IS_ERASE)	14
Update an entry (O_IS_UPDATE)	15
Data file functions	15
Open a data file (O_IS_DOPEN)	15
Flush data file buffers (O_IS_DFLUSH)	15
Get size of data file (O_IS_DSIZE)	15
Read first data record (O_IS_DFIRST)	15
Read next data record (O_IS_DNEXT)	16
Index file functions	16
Open an index (O_IS_IOPEN)	16
Close an index (O_IS_ICLOSE)	16
Flush index file buffers (O_IS_IFLUSH)	17
Set and get index flags (O_IS_IFLAGS)	17
Specify a filter method (O_IS_IFILTER)	17
Specify a duplicates method (O_IS_IDUP)	18
Build an index (O_IS_IBUILD)	18
Build an index from sorted data (O_IS_IQBUILD)	18
Add an index entry (O_IS_IADD)	19
Add an index entry in order (O_IS_IQADD)	19

Erase an index entry (O_IS_IERASE).....	19
Erase all index entries (O_IS_IERASEALL).....	19
Get the number of index entries (O_IS_ICOUNT).....	20
Get size of an index file (O_IS_ISIZE).....	20
Find a record using an index (O_IS_IFIND).....	20
Read next indexed record (O_IS_INEXT).....	21
Read previous indexed record (O_IS_IBACK).....	21
Read first indexed record (O_IS_IFIRST).....	21
Read last indexed record (O_IS_ILAST).....	21
Read current indexed record (O_IS_ICURRENT).....	21

CHAPTER 1

INTRODUCTION

Overview

This manual describes the Indexed Sequential Access Method (ISAM) dynamic library for SIBO machines (HC, S3, S3a and MC).

The ISAM library provides a powerful set of functions for rapid and efficient access of *Database files* (DBF files). Such files consist of multiple records containing one or more structured fields. They can be created, maintained and accessed from OPL programs and database applications on the HC, S3, S3a and MC (see the *Database Files* chapter of the *PLIB Reference* manual).

The ISAM library uses *B-tree index files* to provide rapid and efficient indexed and sequential access to DBF files. It is the most widely used form of index file simply because it is the most efficient general method of accessing database files. In brief, *B-tree* index files minimise the number of disk accesses per retrieval - manipulating data already in memory is much faster than accessing the disk - whilst ensuring a well balanced index i.e. no one retrieval requires an inordinately large number of disk accesses (for details see for example *Structures*, by Michael J. Folk and Bill Zoellick, published by Addison-Wesley Publishing Company).

The ISAM library

The ISAM library functions provide the following:

- Fast record retrieval on a key. For example a DBF file can be searched for all records containing the key "Smith".
- Very little degradation of the speed of retrieval as the number of records increases - even large DBF files can be manipulated with relative ease.
- Sequential access to records that are ordered by the key (that is, *next*, *back*, *first* and *last*). For example a record with key "Smalley" that precedes a current record with key "Smith" can be rapidly found using the `o_is_iback` function.
- The ability to see a selection of the records in the file (by constructing a selective or *sparse* index). For example all records containing the key "Smith" can be viewed.
- Access to record *fields* with automatic conversion from numbers to text (and vice-versa) if required.
- The ability to make powerful key definitions that facilitate access ordered on combinations of up to eight fields. For example all records with key fields "Smith" and "Slough" can be found with "Smith" given the higher priority.
- The opening of up to 31 index files for a DBF file at any one time. For example, a DBF file containing customer details can have two indexes, one for searching on customer occupation and related details and the other for searching on customer location and related details.
- The adding, erasing, and updating of records with the appropriate updating of all associated index files carried out automatically thus greatly facilitating the task of maintaining an index.

DBF files

Fields

The following field types are available:

BYTE, UBYTE, WORD, UWORD, LONG, ULONG, DOUBLE, STRING.

However, if compatibility with existing OPL or PLIB DBF functions is required, the field types must be restricted to:

WORD, LONG, DOUBLE, STRING

Number of records

The maximum number of records that can be handled by the PLIB DBF functions is 65534.

However, in ISAM the maximum is 2147483647.

B-tree index files

The ISAM library uses *B-tree index files* to provide much more powerful indexed and sequential access to DBF files than would otherwise be possible.

In general B-tree index files should not be constructed or maintained on a Flash SSD. There is however no reason not to read an index file from Flash SSD.

Block buffering

B-tree index files are block structured files having a block size of ISAM_BLOCK_SIZE (512) bytes (in the literature these blocks are sometimes referred to as pages - the terms are interchangeable).

Index files are read using a least-recently-used (LRU) block buffering scheme - blocks that are repeatedly accessed are kept in memory (typically these are blocks near the root of the B-tree index). LRU block buffering has such an impact on the performance of the B-tree algorithm that it is not reasonable to do without it.

The ISAM library implements a fixed-size LRU block pool per process regardless of the number of open index files. The size is fixed by calling the ISAM function `o_is_init` and specifying zero causes the default of 20 blocks (10K bytes) to be used. If more than one ISAM object is created, the same block pool is used; if a new size is specified in `o_is_init`, it is ignored if there are any index files still open. The new size is used once all of the original index files are closed.

The LRU block pool is implemented in an external memory segment and does not, therefore, detract from the process data segment. The segment name will be `BLK$nnnn.BLK` where `nnnn` is the process ID in hex.

Key description

A key consists of up to eight fields with the constituent fields prioritised according to their order - the first having the highest priority - subject to an overall maximum key size of 64 bytes. For example, keys can be constructed from eight DOUBLE fields or, as a further example, the first 63 characters of a single text field (stored as a leading byte count string).

The field types: BYTE, UBYTE, WORD, UWORD, LONG, ULONG and DOUBLE are compared numerically whereas STRING fields are compared lexically using a given number of characters at a given offset from the start. The comparison is subject to a programmer defined collation table.

The specification of the key is stored in the header of the B-tree index file and the structure is defined by ISAM_KEYDEF:

```

typedef struct
{
    WORD field;           /* Field number for key */
    UBYTE type;          /* Type of field in key */
    UBYTE flags;         /* ISAM_FIELDFLAG_ASCEND or ISAM_FIELDFLAG_DESCEND */
    UBYTE offset;        /* Offset for start of comparison */
    UBYTE len;           /* Number of characters to compare */
    WORD convert;        /* Convert flag or collate table */
} ISAM_KEY_FIELD;

typedef struct
{
    WORD nKeyFields; /* Number of key fields following... */
    ISAM_KEY_FIELD keyField[ISAM_MAX_KEYFIELDS];
} ISAM_KEYDEF;

```

The convert field can take one of the following values:

```

ISAM_CONVERT_NOFOLD
ISAM_CONVERT_FOLD
ISAM_CONVERT_UPPER
ISAM_CONVERT_LOWER

```

which range from 0 - 3, or it can be the address of a user defined 256 byte collate table.

The priority for the field comparisons is the order they are specified with the first one having highest priority.

Size of B-tree files

The size of a B-tree file depends upon the key size; therefore, it is good practice to keep the key as compact as possible. A worst-case B-tree file is only 50% full so the maximum size of a B-tree file is approximately:

$$2 * (\text{number of records}) * (\text{key length} + 6)$$

For example, an index file with an eight byte key for a 1000 record file would require approximately 28K bytes of storage.

However, under typical use, a B-tree will be approximately 67% full and with the `ISAM_MINIMISE` flag set (see the `O_IS_IFLAGS` function) this is increased to approximately 86%. Note also that B-trees built from sorted data using `O_IS_IQBUILD` or `O_IS_IQADD` are significantly more compact, typically 98% or over for large numbers of records.

Interface to the ISAM library

The ISAM library is written using object-oriented programming (OOP). A class hierarchy has been implemented such that OOP programmers can subclass the library to handle record files other than DBF files and to use index files other than B-tree files.

Most of the ISAM functions can be called using conventional C (using the PLIB `p_send` or `p_entsend` functions) and the library is compatible with any combination of CLIB and PLIB libraries.

The rest of this section describes how to use the ISAM library from conventional C. In any program using ISAM, the following steps will be involved:

- Load the ISAM DYL.
- Create an ISAM object
- Call the ISAM functions
- Destroy the ISAM object

Loading the DYL

The ISAM library is supplied as one file `ISAM.DYL`. Before any ISAM functions can be called, this must be *loaded*. This will get a *category handle* which is needed for an ISAM *object* to be created. The loading must be done in one of two ways depending on whether it has been linked into a multiple DYL file (normally an executable) or not.

- If not linked to a multiple DYL file, the PLIB function `p_loadlib` must be used, for example:

```
HANDLE isamCat;
```

```
p_loadlib("ISAM.DYL",&isamCat,TRUE);
```

will load ISAM.DYL from the current directory and write the category handle to `isamCat`.

- If linked to a multiple DYL, the two PLIB functions `p_openlib` and `p_loadfilelib` must be used, for example, if ISAM.DYL has been linked as the first DYL in an executable:

```
HANDLE isamCat;
```

```
VOID *chan;
```

```
chan=NULL;
```

```
if (p_openlib(&chan,DatCommandPtr)==0)
```

```
    p_loadfilelib(chan,0,&isamCat,TRUE);
```

```
p_close(chan);
```

will load ISAM.DYL from the current executable and write the category handle to `isamCat`.

Creating an ISAM object

Once the ISAM library has been loaded, an ISAM object can be created. The *class* of the object required is defined in *isam.g* as `C_BTDBF`, which is an ISAM object using B-trees to access DBF files (ISAM may be subclassed to produce objects accessing other types of file).

The PLIB functions `p_newlibh` or `f_newlibh` can be used to create the object, for example:

```
VOID *pIsam;
```

```
pIsam=p_newlibh(isamCat,C_BTDBF);
```

Since there is always some initialisation to be carried out, it is generally more convenient to use the PLIB function `f_newlibhsend`, for example:

```
VOID *pIsam;
```

```
pIsam=f_newlibhsend(isamCat,C_BTDBF,O_IS_INIT,4096,0);
```

will create an ISAM object and then call the function `O_IS_INIT` to initialise a record buffer of 4096 bytes and the default index buffer (see the *ISAM Functions* chapter for the details of the `O_IS_INIT` function).

Calling ISAM functions

The ISAM functions are called by *sending a message* to an ISAM object using the PLIB functions `p_send` or `p_entsend`. The messages are defined in *isam.g* and take the form `O_IS_xxx` where `xxx` is the name of the function. Since all ISAM functions which can fail return zero for success or leave with a negative error number, `p_entsend` can easily be used to catch the error or `p_send` can be used if the error handling is done at a higher level. For example:

```
INT c;
```

```
if ((c=p_entsend4(pIsam,O_IS_DOPEN,"TEST.DBF",P_FOPEN))<0)
```

```
    p_printf("Open data file failed with error %d",c);
```

catches the error whereas:

```
p_send4(pIsam,O_IS_DOPEN,"TEST.DBF",P_FOPEN);
```

passes any error to a higher level.

Destroying the ISAM object

The ISAM function `O_IS_DESTROY` will close the data file and all open index files and free all memory used and does not leave or return errors. However, since data is buffered, it could fail. Carefully written applications should therefore make sure that all data has been flushed and any error dealt with before calling `O_IS_DESTROY`. The functions `O_IS_DFLUSH` and `O_IS_IFLUSH` must be used.

Typical use

There are many ISAM functions available; here is a sample in the order that they are typically used:

- Create and initialise an ISAM object, use PLIB function `f_newlibhsend`.

- Set the field definition for the data file, use `O_IS_SET_FIELDDEF`.
- Open or create a data file, use `O_IS_DOPEN`.
- Set the key definition for an index file, use `O_IS_SET_KEYDEF`.
- Open or create one or more index files, use `O_IS_IOPEN`.
- Build an index file, use `O_IS_IBUILD`.
- Add or erase records, use `O_IS_ADD` or `O_IS_ERASE`.
- Access ordered data with keys, use `O_IS_IFIND` or `O_IS_INEXT` etc.
- Flush buffers, use `O_IS_DFLUSH` and `O_IS_IFLUSH`.
- Destroy the ISAM object, use `O_IS_DESTROY`.

Example programs

Note that the following two examples do not include complete error handling.

Building a complete (dense) index

This program creates a DBF file called `EXAMPLE.DBF` with one field of type `LONG` and appends 100 random numbers to it. It then creates and builds an index file called `EX1.BTX` of these numbers in ascending order.

Note that in the argument list for the `setFieldDef` and `setKeyDef` subroutines, the TopSpeed C compiler understands the three dots to indicate an undefined number of arguments of undefined type.

```
#include <p_std.h>
#include <p_file.h>
#include <p_sys.h>

#include <epoc.h>
#include <isam.g>

GLREF_C TEXT *DatCommandPtr;

LOCAL_C HANDLE isamCat;
LOCAL_C VOID *pIsam;

LOCAL_C VOID pErr(UBYTE *mess, INT err)
{
    UBYTE buf[E_MAX_ERROR_TEXT_SIZE];

    p_errs(&buf[0], err);
    p_printf("%s failed - %s", mess, &buf[0]);
    p_getch();
    p_exit(TRUE);
}

LOCAL_C VOID loadIsam(VOID)
{
    INT c;
    TEXT dylname[P_FNAME_SIZE];

    p_fparse("ISAM.DYL", &dylname[0], NULL);
    if ((c = p_loadlib(&dylname[0], &isamCat, TRUE)) < 0)
        pErr("Load ISAM.DYL", c);
}

LOCAL_C VOID createIsam(VOID)
{
    pIsam = f_newlibhsend(isamCat, C_BTDBF, O_IS_INIT, 4096, 0);
}

LOCAL_C VOID setFieldDef(INT nFields, ...)
{
    p_send4(pIsam, O_IS_SET_FIELDDEF, nFields, &nFields + 1);
}
```

```
    }

LOCAL_C VOID setKeyDef(INT nKeyFields,...)
{
    p_send4(pIsam,O_IS_SET_KEYDEF,nKeyFields,&nKeyFields+1);
}

LOCAL_C VOID openData(TEXT *name,UINT mode)
{
    INT c;

    if ((c=p_entsend4(pIsam,O_IS_DOPEN,name,mode))<0)
        pErr("Open data file",c);
}

LOCAL_C INT openIndex(TEXT *name,UINT mode)
{
    INT c;

    if ((c=p_entsend4(pIsam,O_IS_IOPEN,name,mode))<0)
        pErr("Open index file",c);
    return(c);
}

GLDEF_C VOID main(VOID)
{
    ULONG seed;
    LONG l;
    INT nRecords;
    INT indexId;
    INT i;

    loadIsam();
    createIsam();
    setFieldDef(1,ISAM_FIELDTYPE_LONG);
    openData("EXAMPLE.DBF",P_FREPLACE|P_FUPDATE);
    seed=0L;
    nRecords=100;

    p_printf("Generating %u records",nRecords);
    for (i=0;i<nRecords;i++)
    {
        l=p_randl(&seed);
        p_send5(pIsam,O_IS_PUT_FIELD,0,&l,ISAM_FIELDTYPE_LONG);
        p_send2(pIsam,O_IS_ADD);
    }

    p_printf("Building index");
    setKeyDef(1,0,ISAM_FIELDTYPE_LONG,ISAM_FIELDFLAG_ASCEND);
    indexId=openIndex("EX1.BTX",P_FREPLACE|P_FUPDATE);
    p_send3(pIsam,O_IS_IBUILD,indexId);

    p_printf("Sorted data:");
    p_send3(pIsam,O_IS_IFIRST,indexId);
    for (i=0;i<nRecords;i++)
    {
        p_send5(pIsam,O_IS_GET_FIELD,0,&l,ISAM_FIELDTYPE_LONG);
        p_printf("%ld",l);
        p_send3(pIsam,O_IS_INEXT,indexId);
    }
    p_send2(pIsam,O_IS_DESTROY);
    p_getch();
    p_exit(0);
}
```

Building a selective (sparse) index

The following routine opens the DBF file called EXAMPLE.DBF which was created in the previous example. It then creates and builds an index file called EX2.BTX of those numbers which are divisible by ten in ascending order.

```
LOCAL_C VOID div10(VOID)
{
    LONG l;
    INT indexId;
    INT eof;

    createIsam();
    openData("EXAMPLE.DBF",P_FOPEN);
    p_printf("Build index of numbers divisible by 10");
    setKeyDef(1,0,ISAM_FIELDTYPE_LONG,ISAM_FIELDFLAG_ASCEND);
    indexId=openIndex("EX2.BTX",P_FREPLACE|P_FUPDATE);
    eof=p_send2(pIsam,O_IS_DFIRST);
    while (eof==FALSE)
    {
        p_send5(pIsam,O_IS_GET_FIELD,0,&l,ISAM_FIELDTYPE_LONG);
        if ((l%10)==0)
            p_send3(pIsam,O_IS_IADD,indexId);
        eof=p_send2(pIsam,O_IS_DNEXT);
    }
    p_printf("Sorted data:");
    eof=p_send3(pIsam,O_IS_IFIRST,indexId);
    while (eof==FALSE)
    {
        p_send5(pIsam,O_IS_GET_FIELD,0,&l,ISAM_FIELDTYPE_LONG);
        p_printf("%ld",l);
        eof=p_send3(pIsam,O_IS_INEXT,indexId);
    }
    p_send2(pIsam,O_IS_DESTROY);
}
```


CHAPTER 2

ISAM FUNCTIONS

ISAM functions are called using either `p_send` or `p_entsend` to the ISAM object created with (for example) the PLIB function `p_newlibh`.

Most functions which can fail will return zero or a positive number for success or *leave* with a negative error number, giving the caller the option of catching the error using `p_entsend` or leaving the error handling to a higher level by calling `p_send`.

Functions that can generate `E_FILE_EOF` (for example, `O_IS_INEXT`), will *return* this negative error rather than *leave*.

All functions can be called from standard C but `O_IS_IFILTER` and `O_IS_IDUP` are only applicable when using object oriented programming (OOP).

There are three types of functions:

- General functions
- Data file functions
- Index file functions

General functions

This section describes functions to perform the following:

- Initialise and destroy an ISAM object.
- Define and read the *field definition* for a *data file*.
- Define and read the *key definition* for an *index file*.
- Read and write data to the *record buffer*.
- Set the format for conversion of numbers to text.
- Add, erase and update records in the *data file* with appropriate updates to all associated *index files*.

O_IS_INIT

Initialise an ISAM object

```
VOID p_send4(VOID *pIsam, O_IS_INIT, INT rBufSize, INT iBufBlocks);
```

Initialise the ISAM object with a *record buffer* of length `rBufLen` and an *index buffer* in an external memory segment with `iBufBlocks` blocks (a block is `ISAM_BLOCK_SIZE` (512) bytes long). If `iBufBlocks` is zero, the default number of blocks, `BL_DEFAULT_BLOCKS` (20) is used.

Performs the following initialisation:

- Allocates `rBufLen` bytes for the *record buffer*.
- Sets the *field definition* to the default of 32 string fields.
- Sets the *key definition* to the default of the first eight characters of the first string field.

- Sets the radix for number conversion to the default of 10.
- Sets the format for doubles converted to text to the default of `P_DTOB_GENERAL` with a width of 20 and a decimal point character of '.'.

`rBufLen` sets the size of the *record buffer* and must be in the range `ISAM_MIN_RECORD_BUFFER` (512) to `ISAM_MAX_RECORD_BUFFER` (16384). If it is not, `p_panic` will be called. The record buffer is used to read records from the *data file* and must be at least as long as the longest record to be read. A length of `ISAM_MAX_RECORD_LENGTH+2` (4096) is guaranteed to be large enough for all records. When an *index file* is being built, data is read from the file in blocks as large as the record buffer; the bigger the buffer, the faster the build will be.

`iBufBlocks` must be less than or equal to `ISAM_MAX_SEG_BLOCKS` (1024). Any value greater than this will cause `p_panic` to be called.

Leaves with `E_GEN_NOMEMORY` if there is insufficient memory to initialise everything.

Typically, this function would be called when the object is created and can be combined into one call using (for example) the PLIB function `f_newlibhsend`:

```
pIsam=f_newlibhsend(isamCatNum,C_BTDBF,O_IS_INIT,4096,0);
```

This will create an ISAM object and initialise it with a *record buffer* of 4096 bytes and the default number of *index buffer* blocks.

Creating an ISAM object is explained further in the *Introduction* chapter.

O_IS_DESTROY Destroy an ISAM object

```
VOID p_send2(VOID *pIsam, O_IS_DESTROY);
```

Destroy the ISAM object. The *data file* and all open *index files* are closed and all memory is freed.

Note that if more than one ISAM object is created in the same process, the external segment memory used for buffering the indexes is shared and will only be freed when the last ISAM object is destroyed.

`O_IS_DESTROY` can fail due to one or more write operations but will always succeed in destroying the ISAM object (freeing all memory used). No error is ever returned and the function will not leave.

Carefully written applications avoid this problem by using `O_IS_DFLUSH` and `O_IS_IFLUSH`, to flush all buffered data (and taking appropriate action if this fails) before destroying the object without risk of failure.

O_IS_SET_FIELDDEF Set the field definition

```
INT p_send4(VOID *pIsam, O_IS_SET_FIELDDEF, INT nFields, VOID *pArgs);
```

Set the field definition to contain `nFields` with types in the list at `pArgs`. This field definition will be used when a *data file* is created.

Alternatively `pArgs` can point to an `ISAM_FIELDDEF` structure containing the field definition while `nFields` is passed as zero.

The `ISAM_FIELDDEF` structure is defined in *isam.g* as:

```
typedef struct
{
    WORD nFields;
    UBYTE type[ISAM_MAX_DEFINED_FIELDS];
} ISAM_FIELDDEF
```

For example:

```
LOCAL_C VOID defineFields(VOID)
{
    ISAM_FIELDDEF fieldDef;

    fieldDef.nFields=2;
    fieldDef.type[0]=ISAM_FIELDTYPE_WORD;
    fieldDef.type[1]=ISAM_FIELDTYPE_LONG;
    p_send4(pIsam,O_IS_SET_FIELDDEF,0,&fieldDef);
}
```

is equivalent to:

```
LOCAL_C VOID setFieldDef(INT nFields,...)
{
    p_send4(pIsam,O_IS_SET_FIELDDEF,nFields,&nFields+1));
}

LOCAL_C VOID defineFields(VOID)
{
    setFieldDef(2,ISAM_FIELDTYPE_WORD,ISAM_FIELDTYPE_LONG);
}
```

and both define the field structure of the data file to be two fields, the first a WORD field, the second a LONG.

The maximum number of fields that can be specified is ISAM_MAX_DEFINED_FIELDS (32) and each field type must be one of the following types:

ISAM_FIELDTYPE_BYTE	for a signed BYTE field.
ISAM_FIELDTYPE_UBYTE	for an unsigned UBYTE field.
ISAM_FIELDTYPE_WORD	for a signed WORD field.
ISAM_FIELDTYPE_UWORD	for an unsigned UWORD field.
ISAM_FIELDTYPE_LONG	for a signed LONG field.
ISAM_FIELDTYPE_ULONG	for an unsigned ULONG field.
ISAM_FIELDTYPE_DOUBLE	for a DOUBLE field.
ISAM_FIELDTYPE_STRING	for a leading byte count string field.

More fields than are specified in the field definition can be accessed, but they are always assumed to be of type ISAM_FIELDTYPE_STRING.

Note that this specifies the field types to be used in the actual *data file*; however, conversion to and from ISAM_FIELDTYPE_STRING is performed automatically by the functions O_IS_PUT_FIELD and O_IS_GET_FIELD. See the descriptions of these two functions for further details.

Similarly, *index files* may specify field types which differ from those specified for data file fields. The field types will be converted automatically, provided the conversion is to or from ISAM_FIELDTYPE_STRING. See the description of O_IS_SET_KEYDEF.

Returns zero if successful or leaves with E_GEN_ARG if the given field definition is invalid.

O_IS_GET_FIELDDEF

Get the field definition

```
ISAM_FIELDDEF *p_send2(VOID *pIsam, O_IS_GET_FIELDDEF);
```

Get the current field definition, returning a pointer to an ISAM_FIELDDEF structure.

O_IS_SET_KEYDEF

Set the key definition

```
INT p_send4(VOID *pIsam, O_IS_SET_KEYDEF, INT nKeyFields, VOID *pArgs);
```

Set the key definition (to be used for the next created index file) to contain nKeyFields with the parameters for each key field specified in the list at pArgs. This key definition will be used for the next *index file* created.

Alternatively pArgs can point to an ISAM_KEYDEF structure containing the key definition while nFields is passed as zero.

The ISAM_KEYDEF structure is defined in *isam.g* by:

```
typedef struct
{
    WORD field;           /* Field number for key */
    UBYTE type;           /* Type of field in key */
    UBYTE flags;          /* ISAM_FIELDFLAG_ASCEND or ISAM_FIELDTYPE_DESCEND */
    UBYTE offset;         /* Offset for string comparison */
    UBYTE len;            /* Length for string comparison */
    WORD convert;         /* Convert flag or collate table */
} ISAM_KEY_FIELD;

typedef struct
{
    WORD nKeyFields;
    ISAM_KEY_FIELD keyField[ISAM_MAX_KEYFIELDS];
} ISAM_KEYDEF;
```

For example:

```
LOCAL_C VOID defineKey(VOID)
{
    ISAM_KEYDEF keyDef;

    keyDef.nKeyFields=2;
    keyDef.keyField[0].field=3;
    keyDef.keyField[0].type=ISAM_FIELDTYPE_STRING;
    keyDef.keyField[0].flags=ISAM_FIELDFLAG_ASCEND;
    keyDef.keyField[0].offset=0;
    keyDef.keyField[0].len=8;
    keyDef.keyField[0].convert=ISAM_CONVERT_NOFOLD;
    keyDef.keyField[1].field=1;
    keyDef.keyField[1].type=ISAM_FIELDTYPE_LONG;
    keyDef.keyField[1].flags=ISAM_FIELDFLAG_ASCEND;
    p_send4(pIsam,0_IS_SET_KEYDEF,0,&keyDef);
}
```

is equivalent to:

```
LOCAL_C VOID setKeyDef(INT nKeyFields,...)
{
    p_send4(pIsam,0_IS_SET_KEYDEF,nKeyFields,&nKeyFields+1);
}

LOCAL_C VOID defineKey(VOID)
{
    setKeyDef(2,3,ISAM_FIELDTYPE_STRING,ISAM_FIELDFLAG_ASCEND,0,8,ISAM_CONVERT_NOFOLD,1,
    ISAM_FIELDTYPE_LONG, ISAM_FIELDTYPE_ASCEND);
}
```

The fields: offset, len and convert must only be supplied if the type is ISAM_FIELDTYPE_STRING.

The key definition is subject to the following validation:

- The number of key fields must be in the range 1 to ISAM_MAX_KEYFIELDS (8).
- The field number for each key field must be in the range 0 to ISAM_MAX_RECORD_FIELDS (4094).
- The type of each key field (the type actually stored in the *index file*) must be valid and must be either the same as the type specified in the current *field definition* or be a conversion from or to an ISAM_FIELDTYPE_STRING.
- The length of a key produced by this definition must be no greater than ISAM_MAX_KEY_LENGTH (64).

Returns the length of a key resulting from this definition if successful or leaves with E_GEN_ARG if the key definition is invalid.

O_IS_GET_KEYDEF**Get the key definition**

```
ISAM_KEYDEF *p_send3(VOID *pIsam, O_IS_GET_KEYDEF, INT indexId);
```

Get the key definition for the index `indexId`, returning a pointer to an `ISAM_KEYDEF` structure. The value `indexId` is returned when an index is opened (see `O_IS_IOPEN` below).

O_IS_PUT_FIELD**Put a value into a field**

```
INT p_send5(VOID *pIsam, O_IS_PUT_FIELD, INT field, VOID *pData, INT type);
```

Put the data at address `pData` into the *record buffer* at field number given by `field`. If any preceding fields have not been set, they will be automatically set to contain zeros.

The `type` parameter is the type of the data pointed to by `pData`. If `type` is `ISAM_FIELDTYPE_STRING`, the data at `pData` must be in the form of a leading byte string.

If `type` is different from that specified for field number `field` (as reported by `O_IS_GET_FIELDDDEF`) and either of these two types is `ISAM_FIELDTYPE_STRING`, the field data will be converted between numeric and string types as necessary according to the current radix (as set by `O_IS_SET_RADIX`).

For example, if the *field definition* is set to 32 strings and the current radix is decimal (the defaults), the following code will put the value "31" into field zero in the *record buffer* as a leading byte count string.

```
WORD value;
```

```
value=31;
```

```
p_send5(pIsam,O_IS_PUT_FIELD,0,&value,ISAM_FIELDTYPE_WORD);
```

Returns zero if successful or leaves with one of the following negative error numbers:

<code>E_FILE_RECORD</code>	the assignment to the field would make the total record length greater than <code>ISAM_MAX_RECORD_LENGTH</code> (4094) or the size of the buffer specified in <code>O_IS_INIT</code> .
<code>E_GEN_ARG</code>	the type conversion required is not to or from <code>ISAM_FIELDTYPE_STRING</code> .
<code>E_GEN_FAIL</code>	the type conversion failed because the string could not be recognised as a number.
<code>E_GEN_OVER</code>	the type conversion failed because the number is out of range.

O_IS_GET_FIELD**Get a value from a field**

```
INT p_send5(VOID *pIsam, O_IS_GET_FIELD, INT field, VOID *pData, INT type);
```

Get the data from `field` in the *record buffer* into the buffer at `pData`.

The `type` parameter is the type of the data required at `pData`. If `type` is `ISAM_FIELDTYPE_STRING`, the data will be written to `pData` in the form of a leading byte string.

If `type` is different from that specified for field number `field` (as reported by `O_IS_GET_FIELDDDEF`) and either of these two types is `ISAM_FIELDTYPE_STRING`, the field data will be converted between numeric and string types as necessary according to the current radix (as set by `O_IS_SET_RADIX`).

For example, if the *field definition* is set to 32 strings (the default), the following code will get the value from field 10 in the *record buffer* and convert it to a `LONG`.

```
LONG lValue;
```

```
p_send5(pIsam,O_IS_GET_FIELD,10,&lValue,ISAM_FIELDTYPE_LONG);
```

Returns zero if successful or leaves with one of the following negative error numbers:

<code>E_GEN_ARG</code>	the type conversion required is not to or from <code>ISAM_FIELDTYPE_STRING</code> .
<code>E_GEN_FAIL</code>	the type conversion failed because the string could not be recognised as a number.
<code>E_GEN_OVER</code>	the type conversion failed because the number is out of range.

O_IS_GET_TYPE**Get the type of a field**

```
INT p_send3(VOID *pIsam, O_IS_GET_TYPE, INT field);
```

Return the actual type of field in the *data file*.

This provides the same functionality as calling `O_IS_GET_FIELDDDEF` and then inspecting the required field. It is included for convenience.

O_IS_GET_RBUF**Get the address of the record buffer**

```
ISAM_RECORDBUF *p_send2(VOID *pIsam, O_IS_GET_RBUF);
```

Get the actual address of the *record buffer* allocated in `O_IS_INIT`, returning a pointer to an `ISAM_RECORDBUF` structure. This struct is not defined in any header file - users of this method function should declare the struct in their own code as:

```
typedef struct
{
    WORD len;                /* Length of record */
    UBYTE data[2];          /* len bytes of data... */
} ISAM_RECORDBUF;
```

This function is provided to allow direct access to the current record in the buffer, if needed. Typically, it will be more convenient to use `O_IS_GET_FIELD` and `O_IS_PUT_FIELD`.

O_IS_SET_RADIX**Set the radix for number conversion**

```
VOID p_send3(VOID *pIsam, O_IS_SET_RADIX, INT radix);
```

Set the base radix for number conversion to radix. By default this is set to 10.

O_IS_SET_DFORMAT**Set the text format for DOUBLES**

```
VOID p_send3(VOID *pIsam, O_IS_SET_DFORMAT, P_DTOB *dFormat);
```

Set the format for DOUBLES converted to text to be that specified by `dFormat`.

O_IS_ADD**Add an entry**

```
INT p_send2(VOID *pIsam, O_IS_ADD);
```

Add the record currently in the record buffer to the data file and attempt to add entries to all open index files which do not have `ISAM_FLAG_MANUAL` set (see `O_IS_IFLAGS`).

One or more of the index files may *refuse* to actually add the entry for any of the following reasons:

- A *filter method* returns FALSE (see `O_IS_IFILTER`).
- The entry matches an existing entry in the index and a *duplicate method* returns FALSE (see `O_IS_IDUP`).
- The entry matches an existing entry in the index and `ISAM_FLAG_ALLOWDUP` is not set (see `O_IS_IFLAGS`).

but no error will be given.

Returns FALSE if successful or leaves with one of the negative errors returned by the PLIB function `p_write`.

O_IS_ERASE**Erase an entry**

```
INT p_send2(VOID *pIsam, O_IS_ERASE);
```

Erase the record currently in the record buffer from the data file and from all open index files which do not have `ISAM_FLAG_MANUAL` set (see `O_IS_IFLAGS`).

The record buffer must contain a record read by a successful call to one of the following functions:

`O_IS_DFIRST`, `O_IS_DNEXT`, `O_IS_IFIND`, `O_IS_INEXT`, `O_IS_IBACK`, `O_IS_IFIRST`, `O_IS_ILAST`, `O_IS_ICURRENT`.

Returns FALSE if successful or E_FILE_EOF if the current record is end-of-file or leaves with one of the negative errors returned by the PLIB function `p_write`.

O_IS_UPDATE

Update an entry

```
INT p_send2(VOID *pIsam, O_IS_UPDATE);
```

Replace the record which was read by a successful call to one of the following functions: `O_IS_DFIRST`, `O_IS_DNEXT`, `O_IS_IFIND`, `O_IS_INEXT`, `O_IS_IBACK`, `O_IS_IFIRST`, `O_IS_ILAST`, `O_IS_ICURRENT`, by the new contents of the record buffer in the data file and all open index files which do not have `ISAM_FLAG_MANUAL` set (see `O_IS_IFLAGS`).

Returns FALSE if successful or E_FILE_EOF if the current record is end-of-file or leaves with one of the negative errors returned by the PLIB function `p_write`.

Data file functions

This section describes functions to perform the following:

- Open or create a *data file*.
- Flush the buffers used for the data file.
- Get the current size of the data file.
- Sequentially read the data file.

O_IS_DOPEN

Open a data file

```
INT p_send4(VOID *pIsam, O_IS_DOPEN, TEXT *name, UINT mode);
```

Open a *data file* specified by the zero terminated file specification *name*.

The *mode* parameter is the same as for the PLIB function `p_open(P_FSTREAM)`. If `P_FCREATE` or `P_FREPLACE` is specified, the data file is created with the current *field definition* (see `O_IS_SET_FIELDDF`). If an existing data file is opened, the current *field definition* is set to that of the file opened.

Only one *data file* can be opened per ISAM object. If an attempt is made to open more than one, `p_panic` will be called.

Returns zero if successful or leaves with one of the negative errors returned by `p_open(P_FSTREAM)`.

O_IS_DFLUSH

Flush data file buffers

```
INT p_send2(VOID *pIsam, O_IS_DFLUSH);
```

Flush all data written to the data file and write the file's modification date.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_write`.

See the PLIB function `p_iow(P_FFLUSH)` for further details on flushing.

O_IS_DSIZE

Get size of data file

```
INT p_send3(VOID *pIsam, O_IS_DSIZE, LONG *size);
```

Write the size (in bytes) of the open data file to *size*.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_seek`.

O_IS_DFIRST

Read first data record

```
INT p_send2(VOID *pIsam, O_IS_DFIRST);
```

Read the first record in the data file into the record buffer.

Returns zero if successful or `E_FILE_EOF` if there are no records or leaves with one of the negative errors returned by the PLIB function `p_read`.

O_IS_DNEXT

Read next data record

```
INT p_send2(VOID *pIsam, O_IS_DNEXT);
```

Read the next record in the data file into the record buffer.

Should be used in conjunctions with `O_IS_DFIRST` to provide sequential access to the data file records in the order they are stored without the need for an index file.

Returns zero if successful or `E_FILE_EOF` if the current record is the last record in the file or leaves with one of the negative errors returned by the PLIB function `p_read`.

Index file functions

This section describes functions to perform the following:

- Open or create an *index file*.
- Flush the buffers used for the index file.
- Build a complete (dense) index file from a sorted or unsorted data file.
- Build an index for a selection of records (sparse) by one of two mechanisms:
 - 1) Specify a function which will be called during the building of the index (OOP only).
 - 2) Add and erase entries from the index directly.
- Get the size and number of entries in an index.
- Provide fast indexed access to records in the data file from a given index and match key.
- Provide sequential access to records in the data file in the order defined by an index.

The function `O_IS_IOPEN` returns an `indexId` which is then passed as a parameter to other index functions.

O_IS_IOPEN

Open an index

```
INT p_send4(VOID *pIsam, O_IS_IOPEN, TEXT *name, UINT mode);
```

Open an *index file* specified by the zero terminated file specification `name`.

The mode parameter has the same meaning as for the PLIB function `p_open(P_FSTREAM)`, except that `P_FAPPEND` should not be specified. If `P_FCREATE` or `P_FREPLACE` is specified, the data file is created with the current *key definition* (see `O_IS_SET_KEYDEF` above). If an existing index file is opened (with a mode of `P_FOPEN`, optionally ORED with `P_FUPDATE`) its key definition can be obtaining using `O_IS_GET_KEYDEF`.

Note that the key definition is stored in the index file itself and cannot be changed once the index file has been created.

Returns an `indexId` from 1 to 31 if successful or leaves with one of the negative errors returned by `p_open(P_FSTREAM)` or `E_GEN_FAIL` if an attempt is made to open more than 31 indexes.

O_IS_ICLOSE

Close an index

```
INT p_send3(VOID *pIsam, O_IS_ICLOSE, INT indexId);
```

Close the index specified by `indexId` and return zero.

`O_IS_ICLOSE` can fail due to one or more write operations but will always succeed in closing the channel (the `indexId` should not be used subsequently). No error is ever returned and the function will not leave.

Carefully written applications avoid this problem by using `O_IS_IFLUSH` to flush all buffered data (and taking appropriate action if this fails) before closing the channel without risk of failure.

O_IS_IFLUSH**Flush index file buffers**

```
INT p_send3(VOID *pIsam, O_IS_IFLUSH, INT indexId);
```

Flush all buffers used for index `indexId` and write the index file's modification date.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_write`.

See the PLIB function `p_iow(P_FFLUSH)` for further details on flushing.

O_IS_IFLAGS**Set and get index flags**

```
INT p_send5(VOID *pIsam, O_IS_IFLAGS, INT indexId, INT mask, INT value);
```

Modify the flag bits given by `mask` to the state (set or clear) given by `value` in index `indexId`.

Returns the new value of the flags; passing `mask` as zero just reads the current flag settings.

The flags available are:

<code>ISAM_FLAG_ALLOWDUP</code>	allow duplicate entries in the index. If this flag is set, duplicate entries will be added when the index is built using <code>O_IS_BUILD</code> or <code>O_IS_@BUILD</code> or when entries are added with <code>O_IS_ADD</code> or <code>O_IS_IADD</code> .
<code>ISAM_FLAG_MANUAL</code>	the index will not be automatically updated by calls to the <i>general functions</i> <code>O_IS_ADD</code> , <code>O_IS_ERASE</code> or <code>O_IS_UPDATE</code> . The <i>index functions</i> <code>O_IS_IADD</code> and <code>O_IS_IERASE</code> must be used to add or erase entries explicitly; typically, it is used to build an index to a selection of records only (a sparse index).
<code>ISAM_FLAG_MINIMIZE</code>	Minimize the storage requirement of the index by altering the way keys are inserted. The index is not compressed when the flag is set but any future additions to the index are inserted using the minimize algorithm. With the flag set, the actual insertion will be slower but the index produced will be smaller and hence a greater portion of it can be held in internal buffers which means finding the insertion point is faster.

Whether it is worth setting this flag is clearly application dependent and the best way to decide is by trial and error.

The flag can be set or cleared at any stage without harming the index structure.

By default, all the above flags are not set.

Note that these settings are stored in the index file but can be modified at any time. For example, if `ISAM_FLAG_ALLOWDUP` is not set, it does not necessarily mean there are no duplicate entries in the index. It just means that the current intention is to not add duplicates.

Example

```
testId=p_send4(pIsam,O_IS_IOPEN,"TEST.INX",P_FREPLACE|P_FUPDATE);
p_send5(pIsam,O_IS_IFLAGS,testId,ISAM_FLAG_ALLOWDUP,ISAM_FLAG_ALLOWDUP);
```

creates an index file in which duplicate entries will be allowed.

O_IS_IFILTER**Specify a filter method**

```
VOID p_send5(VOID *pIsam, O_IS_IFILTER, INT indexId, VOID *pObj, INT method);
```

This function is only applicable when using object oriented programming (OOP).

Specify the object at `pObj` to be called with `method` whenever entries are added to index `indexId`, to facilitate the building of an index to a selection of records (a sparse index).

The method is called with a pointer to the key that is being inserted and the address of a `LONG` which is the data file reference. The method must return `FALSE` if the entry should be omitted from the index or `TRUE` if it should be added.

To remove the filter method, `pObj` must be passed as `NULL`.

O_IS_IDUP**Specify a duplicates method**

```
VOID p_send5(VOID *pIsam, O_IS_IDUP, INT indexId, VOID *pObj, INT method);
```

This function is only applicable when using object oriented programming (OOP).

Specify the object at `pObj` to be called with method whenever a duplicate entry is added to index `indexId`.

The method is called with a pointer to the key that is being inserted and the address of a `LONG` which is the data file reference. The method must return `FALSE` if the duplicate entry should be omitted from the index or `TRUE` if it should be added anyway. There is no regard for the setting of the flag `ISAM_FLAG_ALLOWDUP` in either case, ie the method called has higher priority.

To remove the duplicates method, `pObj` must be passed as `NULL`.

O_IS_IBUILD**Build an index**

```
INT p_send3(VOID *pIsam, O_IS_IBUILD, INT indexId);
```

Build the index specified by `indexId` using its key definition to extract keys from the records in the data file.

If the data file is already ordered by the key for this index, `O_IS_IQBUILD` should be used to build the index *much* faster.

The index is always built from scratch; any existing index entries are erased first.

Returns `FALSE` if the complete build was successful and there were no duplicate entries; returns `TRUE` if successful but there were duplicates or leaves with one of the negative errors returned by the `PLIB` function `p_write` in which case all entries made so far will be erased.

Duplicate entries

The value `TRUE`, returned by `O_IS_IBUILD` is simply to inform the caller that one or more keys extracted from the data during the build were identical. It does not, however, indicate whether more than one copy of the key has been stored in the index or not. This is controlled by the following:

- If a duplicates method has been specified by `O_IS_IDUP`, its return value of `TRUE` or `FALSE` determines whether to insert or not. (This is applicable to OOP only.)
- If there is no duplicates method, the setting of the flag `ISAM_FLAG_ALLOWDUP` is used.

O_IS_IQBUILD**Build an index from sorted data**

```
INT p_send3(VOID *pIsam, O_IS_IQBUILD, INT indexId);
```

Build the index specified by `indexId` using its key definition to extract keys from the records in the (sorted) data file.

For this function to work, the records in the data file must be in the correct order defined by the key definition for this index.

There are two advantages of this function over `O_IS_IBUILD`:

- 1) It is *much* faster for a large number of records. So much faster, in fact, that it is often quicker to sort the data file (using, for example, a quick-sort algorithm) and then call `O_IS_IQBUILD` to build an index than to call `O_IS_IBUILD` with an unsorted data file.
- 2) The index file produced will generally be smaller (more compressed) than that produced using `O_IS_IBUILD`. Note that if the flag `ISAM_FLAG_MINIMIZE` is set before the call to `O_IS_IQBUILD`, an even smaller index file will be produced at a small cost to the time taken (see the `O_IS_IFLAGS` function, above).

The index is always built from scratch, so any existing index entries are erased first.

Returns `FALSE` if the complete build is successful or leaves with one of the negative errors returned by the `PLIB` function `p_write` in which case all entries made so far will be erased.

Note that there is no checking for duplicate entries as there is with the `O_IS_IBUILD` function.

O_IS_IADD**Add an index entry**

```
INT p_send3(VOID *pIsam, O_IS_IADD, INT indexId);
```

Add an entry to the index `indexId` using its key definition to extract a key from the record currently in the record buffer.

The entry is added to the given index only, unlike the function `O_IS_ADD` which adds entries to all appropriate indexes automatically. This provides, therefore, the mechanism for building and maintaining a selective (sparse) index:

- Read records into the record buffer using, for example, `O_IS_DNEXT`, or `O_IS_IFIND`, `O_IS_INEXT` with another index.
- Selectively add entries to this index if certain conditions are met.

This method of building an index will typically be slower than building a sparse index by calling `O_IS_IBUILD` or `O_IS_IQBUILD` after specifying a filter method but does not require OOP.

If the entries are to be added in key order, the function `O_IS_IQADD` should be used to add the entries faster.

Returns FALSE if successful or TRUE if successful but the entry is a duplicate or leaves with one of the negative errors returned by the PLIB function `p_write`.

The decision whether to add a duplicate entry is made in exactly the same way as with `O_IS_IBUILD`.

O_IS_IQADD**Add an index entry in order**

```
INT p_send4(VOID *pIsam, O_IS_IQADD, INT indexId, INT finish);
```

Add an entry to the index `indexId` in order using its key definition to extract a key from the record currently in the record buffer.

The entries must be added in the correct order given by the key for this index and the parameter `finish` should be passed as FALSE for all entries except the last entry, when it should be passed as TRUE.

The index should contain no entries before the first `O_IS_IQADD` is called.

The advantages of this function over `O_IS_IADD` are the same as `O_IS_IQBUILD` over `O_IS_IBUILD`.

Returns FALSE if successful or leaves with one of the negative errors returned by the PLIB function `p_write`.

Note that there is no checking for duplicate entries as there is with the `O_IS_IADD` function.

O_IS_IERASE**Erase an index entry**

```
INT p_send3(VOID *pIsam, O_IS_IERASE, INT indexId);
```

Erase an entry from the index `indexId`, which corresponds to the record currently in the record buffer.

The entry is erased from the given index only, unlike the function `O_IS_ERASE` which erases entries from all appropriate indexes automatically. This provides, therefore, the mechanism for maintaining a selective (sparse) index.

Returns FALSE if successful or TRUE if there is no entry in the index which corresponds to the record in the record buffer or leaves with one of the negative errors returned by the PLIB function `p_write`.

O_IS_IERASEALL**Erase all index entries**

```
INT p_send3(VOID *pIsam, O_IS_IERASEALL, INT indexId);
```

Erase all entries in the index `indexId`.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_write`.

O_IS_ICOUNT Get the number of index entries

```
INT p_send4(VOID *pIsam, O_IS_ICOUNT, INT indexId, LONG *count);
```

Write the number of entries in the open index file `indexId` to `count`.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_seek`.

O_IS_ISIZE Get size of an index file

```
INT p_send4(VOID *pIsam, O_IS_ISIZE, INT indexId, LONG *size);
```

Write the size (in bytes) of the open index file `indexId` to `size`.

Returns zero if successful or leaves with one of the negative errors returned by the PLIB function `p_seek`.

O_IS_IFIND Find a record using an index

```
INT p_send4(VOID *pIsam, O_IS_IFIND, INT indexId, VOID **pArgs);
```

Find an entry in the index `indexId` which matches the key generated by the arguments at `pArgs` and read the corresponding record from the data file into the record buffer.

`pArgs` is the address of an array of pointers to key fields to be matched, terminated by NULL if fewer key fields are supplied than are specified in the key definition for this index.

Returns the following:

FALSE	if a matching entry is found and the corresponding record is read into the record buffer.
TRUE	if no match is found and the record corresponding to the <i>following</i> index entry is read into the buffer.
E_FILE_EOF	if no match is found and there is no following index entry (no record is read into the record buffer).

or leaves with a negative error returned from the PLIB function `p_read`.

For example:

```
LOCAL_C INT findEntry(INT indexId,...)
{
    return(p_send4(pIsam,O_IS_IFIND,indexId,&indexId+1));
}

LOCAL_C VOID find(VOID)
{
    UBYTE str[16];
    LONG l;

    l=99999L;
    str[0]=3;
    str[1]='A';
    str[2]='B';
    str[3]='C';
    if (findEntry(index1,&l,&str[0],NULL)==FALSE)
        p_printf("Found");
    else
        p_printf("Not found");
}
```

will search `index1` for the first entry with field zero equal to 99999L and field one equal to ABC. Note that the field types passed must be the same as those specified in the key definition of the index.

If there is more than one entry in the index which matches the given fields, `O_IS_IFIND` will always read the first one. To read any others, use `O_IS_INEXT`.

O_IS_INEXT**Read next indexed record**

```
INT p_send3(VOID *pIsam, O_IS_INEXT, INT indexId);
```

Read the record corresponding to the next entry in index `indexId` into the record buffer.

Returns zero if successful or `E_FILE_EOF` if the current index entry is the last one or leaves with one of the negative errors returned by the PLIB function `p_read`.

If two or more indexes are open at the same time, this method function may return `E_FILE_EOF` erroneously. If you use two or more indexes simultaneously, you should access this method via a utility function of the following form:

```
GLDEF_C INT DoInext(VOID *pIsam, INT indexId)
{
    INT ret;

    ret=p_send3(pIsam,O_IS_INEXT,indexId);
    if (ret==E_FILE_EOF)
    {
        p_send3(pIsam,O_IS_IBACK,indexId);
        p_send3(pIsam,O_IS_INEXT,indexId);
        ret=p_send3(pIsam,O_IS_INEXT,indexId);
    }
    return(ret);
}
```

O_IS_IBACK**Read previous indexed record**

```
INT p_send3(VOID *pIsam, O_IS_IBACK, INT indexId);
```

Read the record corresponding to the next entry in index `indexId` into the record buffer.

Returns zero if successful or `E_FILE_EOF` if the current index entry is the first one or leaves with one of the negative errors returned by the PLIB function `p_read`.

O_IS_IFIRST**Read first indexed record**

```
INT p_send3(VOID *pIsam, O_IS_IFIRST, INT indexId);
```

Read the record corresponding to the first entry in index `indexId` into the record buffer.

Returns zero if successful or `E_FILE_EOF` if there are no entries in the index or leaves with one of the negative errors returned by the PLIB function `p_read`.

O_IS_ILAST**Read last indexed record**

```
INT p_send3(VOID *pIsam, O_IS_ILAST, INT indexId);
```

Read the record corresponding to the last entry in index `indexId` into the record buffer.

Returns zero if successful or `E_FILE_EOF` if there are no entries in the index or leaves with one of the negative errors returned by the PLIB function `p_read`.

O_IS_ICURRENT**Read current indexed record**

```
INT p_send3(VOID *pIsam, O_IS_ICURRENT, INT indexId);
```

Read the record corresponding to the current entry in index `indexId` into the record buffer.

Returns zero if successful or `E_FILE_EOF` if the current index entry is set to end-of-file or leaves with one of the negative errors returned by the PLIB function `p_read`.

INDEX

O_IS_ADD 14
O_IS_DESTROY 10
O_IS_DFIRST 15
O_IS_DFLUSH 15
O_IS_DNEXT 16
O_IS_DOPEN 15
O_IS_DSIZE 15
O_IS_ERASE 14
O_IS_GET_FIELD 13
O_IS_GET_FIELDDEF 11
O_IS_GET_KEYDEF 13
O_IS_GET_RBUF 14
O_IS_GET_TYPE 14
O_IS_IADD 19
O_IS_IBACK 21
O_IS_IBUILD 18
O_IS_ICLOSE 16
O_IS_ICOUNT 20
O_IS_ICURRENT 21
O_IS_IDUP 18
O_IS_IERASE 19
O_IS_IERASEALL 19
O_IS_IFILTER 17
O_IS_IFIND 20
O_IS_IFIRST 21
O_IS_IFLAGS 17
O_IS_IFLUSH 17
O_IS_ILAST 21
O_IS_INEXT 21
O_IS_INIT 9
O_IS_IOPEN 16
O_IS_IQADD 19
O_IS_IQBUILD 18
O_IS_ISIZE 20
O_IS_PUT_FIELD 13
O_IS_SET_DFORMAT 14
O_IS_SET_FIELDDEF 10
O_IS_SET_KEYDEF 11
O_IS_SET_RADIX 14
O_IS_UPDATE 15

INDEX

1. Introduction	1
2. Objectives	2
3. Scope	3
4. Methodology	4
5. Results	5
6. Discussion	6
7. Conclusion	7
8. References	8
9. Appendix	9
10. Glossary	10
11. Bibliography	11
12. Index	12
13. Acknowledgments	13
14. Executive Summary	14
15. Abstract	15
16. Introduction	16
17. Objectives	17
18. Scope	18
19. Methodology	19
20. Results	20
21. Discussion	21
22. Conclusion	22
23. References	23
24. Appendix	24
25. Glossary	25
26. Bibliography	26
27. Index	27
28. Acknowledgments	28
29. Executive Summary	29
30. Abstract	30
31. Introduction	31
32. Objectives	32
33. Scope	33
34. Methodology	34
35. Results	35
36. Discussion	36
37. Conclusion	37
38. References	38
39. Appendix	39
40. Glossary	40
41. Bibliography	41
42. Index	42
43. Acknowledgments	43
44. Executive Summary	44
45. Abstract	45
46. Introduction	46
47. Objectives	47
48. Scope	48
49. Methodology	49
50. Results	50
51. Discussion	51
52. Conclusion	52
53. References	53
54. Appendix	54
55. Glossary	55
56. Bibliography	56
57. Index	57
58. Acknowledgments	58
59. Executive Summary	59
60. Abstract	60
61. Introduction	61
62. Objectives	62
63. Scope	63
64. Methodology	64
65. Results	65
66. Discussion	66
67. Conclusion	67
68. References	68
69. Appendix	69
70. Glossary	70
71. Bibliography	71
72. Index	72
73. Acknowledgments	73
74. Executive Summary	74
75. Abstract	75
76. Introduction	76
77. Objectives	77
78. Scope	78
79. Methodology	79
80. Results	80
81. Discussion	81
82. Conclusion	82
83. References	83
84. Appendix	84
85. Glossary	85
86. Bibliography	86
87. Index	87
88. Acknowledgments	88
89. Executive Summary	89
90. Abstract	90
91. Introduction	91
92. Objectives	92
93. Scope	93
94. Methodology	94
95. Results	95
96. Discussion	96
97. Conclusion	97
98. References	98
99. Appendix	99
100. Glossary	100
101. Bibliography	101
102. Index	102
103. Acknowledgments	103
104. Executive Summary	104
105. Abstract	105
106. Introduction	106
107. Objectives	107
108. Scope	108
109. Methodology	109
110. Results	110
111. Discussion	111
112. Conclusion	112
113. References	113
114. Appendix	114
115. Glossary	115
116. Bibliography	116
117. Index	117
118. Acknowledgments	118
119. Executive Summary	119
120. Abstract	120
121. Introduction	121
122. Objectives	122
123. Scope	123
124. Methodology	124
125. Results	125
126. Discussion	126
127. Conclusion	127
128. References	128
129. Appendix	129
130. Glossary	130
131. Bibliography	131
132. Index	132
133. Acknowledgments	133
134. Executive Summary	134
135. Abstract	135
136. Introduction	136
137. Objectives	137
138. Scope	138
139. Methodology	139
140. Results	140
141. Discussion	141
142. Conclusion	142
143. References	143
144. Appendix	144
145. Glossary	145
146. Bibliography	146
147. Index	147
148. Acknowledgments	148
149. Executive Summary	149
150. Abstract	150
151. Introduction	151
152. Objectives	152
153. Scope	153
154. Methodology	154
155. Results	155
156. Discussion	156
157. Conclusion	157
158. References	158
159. Appendix	159
160. Glossary	160
161. Bibliography	161
162. Index	162
163. Acknowledgments	163
164. Executive Summary	164
165. Abstract	165
166. Introduction	166
167. Objectives	167
168. Scope	168
169. Methodology	169
170. Results	170
171. Discussion	171
172. Conclusion	172
173. References	173
174. Appendix	174
175. Glossary	175
176. Bibliography	176
177. Index	177
178. Acknowledgments	178
179. Executive Summary	179
180. Abstract	180
181. Introduction	181
182. Objectives	182
183. Scope	183
184. Methodology	184
185. Results	185
186. Discussion	186
187. Conclusion	187
188. References	188
189. Appendix	189
190. Glossary	190
191. Bibliography	191
192. Index	192
193. Acknowledgments	193
194. Executive Summary	194
195. Abstract	195
196. Introduction	196
197. Objectives	197
198. Scope	198
199. Methodology	199
200. Results	200
201. Discussion	201
202. Conclusion	202
203. References	203
204. Appendix	204
205. Glossary	205
206. Bibliography	206
207. Index	207
208. Acknowledgments	208
209. Executive Summary	209
210. Abstract	210
211. Introduction	211
212. Objectives	212
213. Scope	213
214. Methodology	214
215. Results	215
216. Discussion	216
217. Conclusion	217
218. References	218
219. Appendix	219
220. Glossary	220
221. Bibliography	221
222. Index	222
223. Acknowledgments	223
224. Executive Summary	224
225. Abstract	225
226. Introduction	226
227. Objectives	227
228. Scope	228
229. Methodology	229
230. Results	230
231. Discussion	231
232. Conclusion	232
233. References	233
234. Appendix	234
235. Glossary	235
236. Bibliography	236
237. Index	237
238. Acknowledgments	238
239. Executive Summary	239
240. Abstract	240
241. Introduction	241
242. Objectives	242
243. Scope	243
244. Methodology	244
245. Results	245
246. Discussion	246
247. Conclusion	247
248. References	248
249. Appendix	249
250. Glossary	250
251. Bibliography	251
252. Index	252
253. Acknowledgments	253
254. Executive Summary	254
255. Abstract	255
256. Introduction	256
257. Objectives	257
258. Scope	258
259. Methodology	259
260. Results	260
261. Discussion	261
262. Conclusion	262
263. References	263
264. Appendix	264
265. Glossary	265
266. Bibliography	266
267. Index	267
268. Acknowledgments	268
269. Executive Summary	269
270. Abstract	270
271. Introduction	271
272. Objectives	272
273. Scope	273
274. Methodology	274
275. Results	275
276. Discussion	276
277. Conclusion	277
278. References	278
279. Appendix	279
280. Glossary	280
281. Bibliography	281
282. Index	282
283. Acknowledgments	283
284. Executive Summary	284
285. Abstract	285
286. Introduction	286
287. Objectives	287
288. Scope	288
289. Methodology	289
290. Results	290
291. Discussion	291
292. Conclusion	292
293. References	293
294. Appendix	294
295. Glossary	295
296. Bibliography	296
297. Index	297
298. Acknowledgments	298
299. Executive Summary	299
300. Abstract	300
301. Introduction	301
302. Objectives	302
303. Scope	303
304. Methodology	304
305. Results	305
306. Discussion	306
307. Conclusion	307
308. References	308
309. Appendix	309
310. Glossary	310
311. Bibliography	311
312. Index	312
313. Acknowledgments	313
314. Executive Summary	314
315. Abstract	315
316. Introduction	316
317. Objectives	317
318. Scope	318
319. Methodology	319
320. Results	320
321. Discussion	321
322. Conclusion	322
323. References	323
324. Appendix	324
325. Glossary	325
326. Bibliography	326
327. Index	327
328. Acknowledgments	328
329. Executive Summary	329
330. Abstract	330
331. Introduction	331
332. Objectives	332
333. Scope	333
334. Methodology	334
335. Results	335
336. Discussion	336
337. Conclusion	337
338. References	338
339. Appendix	339
340. Glossary	340
341. Bibliography	341
342. Index	342
343. Acknowledgments	343
344. Executive Summary	344
345. Abstract	345
346. Introduction	346
347. Objectives	347
348. Scope	348
349. Methodology	349
350. Results	350
351. Discussion	351
352. Conclusion	352
353. References	353
354. Appendix	354
355. Glossary	355
356. Bibliography	356
357. Index	357
358. Acknowledgments	358
359. Executive Summary	359
360. Abstract	360
361. Introduction	361
362. Objectives	362
363. Scope	363
364. Methodology	364
365. Results	365
366. Discussion	366
367. Conclusion	367
368. References	368
369. Appendix	369
370. Glossary	370
371. Bibliography	371
372. Index	372
373. Acknowledgments	373
374. Executive Summary	374
375. Abstract	375
376. Introduction	376
377. Objectives	377
378. Scope	378
379. Methodology	379
380. Results	380
381. Discussion	381
382. Conclusion	382
383. References	383
384. Appendix	384
385. Glossary	385
386. Bibliography	386
387. Index	387
388. Acknowledgments	388
389. Executive Summary	389
390. Abstract	390
391. Introduction	391
392. Objectives	392
393. Scope	393
394. Methodology	394
395. Results	395
396. Discussion	396
397. Conclusion	397
398. References	398
399. Appendix	399
400. Glossary	400
401. Bibliography	401
402. Index	402
403. Acknowledgments	403
404. Executive Summary	404
405. Abstract	405
406. Introduction	406
407. Objectives	407
408. Scope	408
409. Methodology	409
410. Results	410
411. Discussion	411
412. Conclusion	412
413. References	413
414. Appendix	414
415. Glossary	415
416. Bibliography	416
417. Index	417
418. Acknowledgments	418
419. Executive Summary	419
420. Abstract	420
421. Introduction	421
422. Objectives	422
423. Scope	423
424. Methodology	424
425. Results	425
426. Discussion	426
427. Conclusion	427
428. References	428
429. Appendix	429
430. Glossary	430
431. Bibliography	431
432. Index	432
433. Acknowledgments	433
434. Executive Summary	434
435. Abstract	435
436. Introduction	436
437. Objectives	437
438. Scope	438
439. Methodology	439
440. Results	440
441. Discussion	441
442. Conclusion	442
443. References	443
444. Appendix	444
445. Glossary	445
446. Bibliography	446
447. Index	447
448. Acknowledgments	448
449. Executive Summary	449
450. Abstract	450
451. Introduction	451
452. Objectives	452
453. Scope	453
454. Methodology	454
455. Results	455
456. Discussion	456
457. Conclusion	457
458. References	458
459. Appendix	459
460. Glossary	460
461. Bibliography	461
462. Index	462
463. Acknowledgments	463
464. Executive Summary	464
465. Abstract	465
466. Introduction	466
467. Objectives	467
468. Scope	468
469. Methodology	469
470. Results	470
471. Discussion	471
472. Conclusion	472
473. References	473
474. Appendix	474
475. Glossary	475
476. Bibliography	476
477. Index	477
478. Acknowledgments	478
479. Executive Summary	479
480. Abstract	480
481. Introduction	481
482. Objectives	482
483. Scope	483
484. Methodology	484
485. Results	485
486. Discussion	486
487. Conclusion	487
488. References	488
489. Appendix	489
490. Glossary	490
491. Bibliography	491
492. Index	492
493. Acknowledgments	493
494. Executive Summary	494
495. Abstract	495
496. Introduction	496
497. Objectives	497
498. Scope	498
499. Methodology	499
500. Results	500
501. Discussion	501
502. Conclusion	502
503. References	503
504. Appendix	504
505. Glossary	505
506. Bibliography	506
507. Index	507
508. Acknowledgments	508
509. Executive Summary	509
510. Abstract	510
511. Introduction	511
512. Objectives	512
513. Scope	513
514. Methodology	514
515. Results	515
516. Discussion	516
517. Conclusion	517
518. References	518
519. Appendix	519
520. Glossary	520
521. Bibliography	521
522. Index	522
523. Acknowledgments	523
524. Executive Summary	524
525. Abstract	525
526. Introduction	526
527. Objectives	527
528. Scope	528
529. Methodology	529
530. Results	530
531. Discussion	531
532. Conclusion	532
533. References	533
534. Appendix	534
535. Glossary	535
536. Bibliography	536
537. Index	537
538. Acknowledgments	538
539. Executive Summary	539
540. Abstract	540
541. Introduction	541
542. Objectives	542
543. Scope	543
544. Methodology	544
545. Results	545
546. Discussion	546
547. Conclusion	547
548. References	548
549. Appendix	549
550. Glossary	550
551. Bibliography	551
552. Index	552
553. Acknowledgments	553
554. Executive Summary	554
555. Abstract	555
556. Introduction	556
557. Objectives	557
558. Scope	55