Support Group Application Note
*Number: 012*
*Issue: 1*
*Author:*

# HOW TO WRITE A VIEWSTORE UTILITY

Applicable Hardware :

BBC B
BBC B+
BBC Master 128
BBC Master Compact

Related Application Notes: Viewstore Hints and Tips



Support Group
Acorn Computers Limited
Acorn House
Vision Park
Histon
Cambridge CB4 4AE

**HOW TO WRITE A VIEWSTORE UTILITY**
**by Mark Cotton, author of VIEW, VIEWSHEET AND VIEWSTORE**

Since the available space for ROM code is limited to 16k, ViewStore has provision for extra programs or utilities that exist outside the ROM to be used. ViewStore is supplied with several utilities that supplement the ROM, and has an interface built into it to allow utility programs to use routines inside the ROM. Certain areas of memory are also allocated for use by utility programs. Given the knowledge of the interface and memory allocation, it is possible to write extra utilities for ViewStore. The purpose of this document is to define the utility interface to enable third parties to write their own utilities. A good knowledge of ViewStore and assembly language will be essential.

**The Interface**

Various routines within the ROM are available for use by utilities. The routines provide the utility with the means to access format files and data records and indexes and assorted useful functions. The ROM has a jump table at the beginning which directs calls to the various routines in the ROM.

As well as access to the ROM routines, the utility has memory allocated to it. Three areas of memory are available to a utility:

A section of zero page
A section of language absolute workspace
A section of main memory

The amount of zero page and language workspace available to a utility depend upon which ROM routines the utility is going to use. The size and position of the piece of main memory available can only be determined when the utility is run: pointers to the start and end of the main workspace are passed when the utility is started.

The zero page, language workspace and main memory not available to the utility is used by the ViewStore ROM itself. A utility should not alter this memory, but the addresses of some locations are defined to allow utilities to read useful parameters.

**Utility Format**

The format of a utility must conform to certain rules. When a utility is located and loaded from the filing system, ViewStore relocates the code to run at a particular point in memory. This point varies according to the size of the format file loaded and the MOS "high water mark". On machines with second processors attached, ViewStore relocates the utility to run in the space left above the ROM: from &C000 to &F800. Using this relocating system, ViewStore make optimum use of the memory available.

Since it is not easy to write position independent code for the 6502, the ROM includes a relocating system. The utility must provide certain information about itself to enable this system to work. The format of a utility is as follows:-

```
.start      JMP     code
            EQUW    bitmap-start
            EQUW    start
            EQUS    "version no/copyright string"

.code               /utility code begins here

                    /end of utility code

.bitmap             /relocation bitmap

                    /end of utility
```

The first word after the initial JMP gives the offset from the beginning of the utility to the relocation bitmap.

Since the utility is relocated after being loaded, the actual assembly address is not important, but ViewStore must be told at what address the utility has been assembled, so that it can calculate how much needs to be added or subtracted from the addresses to be relocated. The second word after the JMP gives the assembly address. This &2000 for most of the supplied utilities.

The version no/copyright string is not essential, but it's a good idea to include one so that you can identify the code.

The main hurdle for anyone writing their own utilities will be the generation of the relocation bitmap. This identifies the addresses that must be relocated. There is a bit in the bitmap for every byte of code in the utility, excluding the bitmap itself. A bit set to zero indicates that an address is not to be relocated; a bit set to one indicates that an address is to be relocated.

It is only possible to relocate addresses using this system, not single bytes. This means that it is not possible to set up or move addresses using immediate data:

```
            LDA     #place
            STA     var
            LDA     #place AND &FF
            STA  var+1
```

is not allowed. You must use the form:

```
            LDA     palcew
            STA     var
            LDA     placew+1
            STA     var+1

 .placed    EQUW    place
```

In the bitmap, a bit which is set is taken to refer to a 16 bit address within the program; it is therefore impossible to have two adjacent bits set since the second bit it referring to the high byte of the address.

Given eight adjacent bytes, all represented by a single byte in the bitmap, the most significant bit in the bitmap byte corresponds to the first code byte, and the least significant bit to the last code byte.

As an example: the first byte of a relocation bitmap always corresponds to the JMP and EQUW structure at the beginning of the utility. Assuming that you have a copyright string (which contains no addresses to be relocated), the first byte of the bitmap should always be &44. The bit string for this is:

```
MSB      LSB

0 1 0 0 0 1 0 0
```

The first three bits correspond to the three bytes of the JMP instruction. The second two bytes of the JMP instruction contain the address to jump to, which must be relocated. The first bit of the two bits for this address is therefore set. The next two bits correspond to the bitmap offset word. This remains constant for any load address, so these two bits are zero. The next two bits are for the assembly address, which will alter as the utility is relocated, and the first bit is a 1 accordingly. The last bit is for the first byte of the copyright string; zero since this has no addresses to be relocated.

The bitmaps for the utilities which are supplied with ViewStore were generated automatically by an assembler which is not available on the market. If you are going to write a utility for ViewStore, you must find a way of generating relocation bitmaps. This could be done in one of three ways:

1. Generate it by hand
2. Write a program to take an assembled utility and generate the bitmap
3. Modify an assembler to generate relocation bitmaps
4. Assemble it at two different addresses and write a program to compare the two resulting code files. Those locations which have changed need a set bit in the bitmap.

For most people, the last option will be the simplest.

**The Utility Environment**

Most utilities will want to operate upon existing databases. It is possible, though, to have a utility which does not operate on existing data, but creates data, or doesn't act on data at all.

An example of this is the SETUP utility, which creates blank databases. It doesn't refer to any existing data. Utilities which need to access either format files or data files must first check that a database has been loaded, and abort with an error message if there is no loaded database.

This is done by checking the location FILMOD. If FILMOD is non-zero, then a database is loaded. If zero, there is no database loaded.

The normal sequence of operation will be to load a database in ViewStore, with the LOAD command. This loads the format file into memory, and locates the data file. A utility is then started with the UTILITY command. The utility can read the format file as it requires, and can open and process the information in the data file. It can use a subset of data identified by the select file.

Once the utility is running, it can take control of the machine as it needs to, using the memory available to it, and the ROM routines as required.

**Naming of Addresses**

In this document, all location and routine addresses will be referred to by name. Tables of addresses and values are at the end of the document.

You will notice that the names of a block of addresses being with "TEMP". These locations are available for use by the utility as temporary storage, but some are used and altered by routines in the ROM.

**Temporaries**

The numbered temporaries are all either a single byte or two bytes long. Those from TEMPFD to TEMP05 are all one byte long; those from TEMP06 to TEMP14 are all two bytes long. The two byte temporaries are used to store and pass addresses; the single byte temporaries are used to store one byte quantities.

Often, values are passed to and from routines using temporaries, as well as registers in the CPU. Many routines "corrupt" certain temporaries; a list of the temporaries a routine corrupts is given in a summary at the end. Utilities can use temporaries whenever they wish, but of course their use must not clash with any routines that you call in the ROM.

**Entry Parameters**

When the utility is started, the following data is provided:

| | |
|---|---|
| TEMP14 | contains the start address of free main memory |
| VWSLIM | contains the address of the byte after the last free byte in main memory |

VWSLIM will not change while the utility is running, and the utility must not alter VWSLIM. TEMP14 may be altered by a ROM call, so it is best to store it somewhere else for later reference.

The utility is called with a JSR instruction, and ViewStore expects the utility to hand back control, when it is finished, with an RTS instruction.

```
.start      TSX
            STX     stksav
            .
            .
.error      LDX     stksav
TXS
RTS
```

**Zero Page**

Zero page is divided up into four areas of different types:

| | |
|---|---|
| ViewStore variables | read only for utilities. Below &50. |
| Temporaries | TEMPFD-TEMP14. Start at &50. read/write for utilities; also used as parameter areas and workspace for ROM routines. |

| | |
|---|---|
| Floating point accumulators | FACCxxand FWRKxx. Start at &6B. read/write for utilities; if the floating point calls are not used by the utility. this area can be used as general workspace. |
| General workspace | VWSXTZ-&8F inclusive. |

**Language Workspace**

| | |
|---|---|
| LWORK | 16 byte parameter block used by ROM routines. |
| FBLOCK | 27 byte filename work area. |
| LINBUF | 256 byte work area used by one or two ROM routines. |
| VWSXTL-VWSITL | area used by ISAM index system. Can be used as general workspace if the utility is not using indexes. |
| General workspace | VWSITL-&7FF inclusive. |

**ROM Routines**

I will describe the ROM routines in the categories that they fall into. The routine addresses and parameters are summarised in table 1. All routines should be called with a JSR instruction, except for CALUTI which should be called with a JMP instruction, since returning control to the utility is not usually sensible as the new utility will overwrite the old one in memory.

**Data File Control**

These routines give the utility access to the database data file, using the current select file if required. It is only possible to read sequentially through the data file using these calls, but the data will be returned in sorted order if the selected data was sorted.

The utility should not close the intermediate file if it uses it. This is done automatically when control is returned to the ROM.

| | |
|---|---|
| INIIMF | Initialise data sequence. Called to start the data reading sequence. |
| MXTIMF | Get next from data sequence. Each call returns the next data record in the sequence. |

**INIIMF**

This routine is called to initiate a sequence of data transfers. It opens the main data file, and stores its handle in the location EFILE. According to the state of the carry flag on entry, it asks the user if he wishes to use a select file. The user responds with a yes or no, and ViewStore opens the select file (S.database) accordingly. After this, the select file is transparent to the utility; repeated calls to NXTIMF will either return all the records in the data file if the select file is not being used, or the subset of records in the select file, if specified.

| | | |
|---|---|---|
| On entry: | CC | Don't ask "Use select file (Y,N)?" question. |
| | CS | Ask select file question. |
| On exit: | VC | No error. |
| | VS | Error; error code in A. |

**NXTIME**

After starting the sequence with a call to INIIMF, repeated calls to MXTIMF return the records in the data file one by one.

| | | |
|---|---|---|
| On entry: | A | low byte of address to store record. |
| | Y | high byte of address to store record. |
| | TEMP13 | address of the byte after the last byte available to store the record. |
| On exit: | VC | No error. |
| | VS | Error; error code in A. |
| | CC | Not end of file. |
| | CS | End of file (returned on the call after the last record has been processed). |

**Errors**

Many of the ROM routines can return an error status. An error is usually indicated by either the Carry flag (C), or the Overflow flag (V). When an error is indicated, the error code is in the A register. To report the error to the user, call the routine REPERL with this code A. The various error codes and messages are summarised later. A utility can use a ROM error message by loading the appropriate code into A, and calling REPERL.

All file calls have the same error trapping system: after a call, V is set to indicate an error, clear if there was no error. This includes errors causing a BRK, that is control is returned to the calling routine even when a BRK is occurred. When you call REPERL with the returned error code, the BRK message will be reported as normal.

**REPERL**

Reports the error message for the error code in A:

| | | |
|---|---|---|
| On entry: | A | contains error code. |
| On exit: | A,X,Y | undefined. |

**Field and Record Control**

Much of ViewStore's manipulation is on fields and records; accordingly, there are several routines available to make this easier. There are some routines to locate fields in the header, format file or current record; routines to compare field values; and routines to find the size of a given field.

Most of these routines use the two temporaries TEMP06 and TEMP07. TEMP06 points to either a field within the format file, or a field within the current record. TEMP07 points to a field within the current record. The Y indirect indexed addressing mode is used in conjunction with these temporaries to access the field contents: the temporary points to the beginning of the field, and the Y register gives the offset from there.

Remember that the format file itself is in the same format as a data file. The same routines are used to process information in records of the database as in the format file itself. For each field in the database, there is a record in the format file, and this record details the characteristics of its corresponding field in the database. The header record is the first record in the format file.

**Data Format**

The data format is summarised in table 10 at the end of the document. All fields in ViewStore are stored in ASCII, even numbers and dates; each field ends with an end of field marker; each record ends with an end of record marker; and the file ends with an end of file marker; after the end of file marker, the file is padded up to the physical end of file with null characters. If you are processing a field's contents, you should test for the end of field using the CHKEOF and CHKEOR routines. These routines set the flags according to the character that they find. Don't check for the character value explicitly.

Generally, when you make a call to a routine that locates a field, the x register indicates where the field is to be found:

| | |
|---|---|
| X=0 | Field in header; A has field number. |
| X=1 to 254 | Field in format file; X gives format file record number; A has field number. The field numbers of the various format file fields are summarised below. |
| X=225 | Field in current record; A has field number; Y has record number. |

Fields within a record are numbered from 1 to 254. If you ask for a field which is not in the record, the routine will return with the Carry flag set. Whereas ViewStore knows where the format file is located, the address of the current record could be anywhere, and before fields within the current record can be accessed, you must tell ViewStore its address with the SETDPS routine.

| | |
|---|---|
| GETFLD | General field locate routine; can locate a field in the header, format file or current record. |
| GETFRC | Find the address of a field in the current record. |
| GETXFL | Return first non-space character of a field in the A register, folded to upper case if applicable. |
| SETDPS | Set the address of the beginning of the current record. |
| CHKEOF | Check the character in the A register for an end of field character. |
| CHKEOR | Check the character in the A register for an end of record character. |
| SIZFLD | Return the size of a given field. |
| CMPFLD | Compare two fields of the same type and set the flags. |
| SCHFLD | Return the number of a field, given its name. |
| SCHFLN | Return the next field number, given a name, for an ambiguous name specification. |

| | |
|---|---|
| GETWID | Return the display width for a particular field. |
| GETKYW | Return the key width for a particular field. |
| CALSBN | Calculate the number of spaces required to be output before a numeric field to right justify it within the display width. |

**GETFLD**

This routine locates a field in either the database header, the format file or the current record. If the field is in the current record, the address of the first character of the field is set into TEMP06 and also into TEMP07. If the field is in the format file, TEMP07 is left unaltered, and the address of the field is put into TEMP06.

If X is equal to 255, then the routine uses the value in the Y register to locate a record in a list of current records. The list of records is numbered from 0 onwards. The usual way to use this part of the routine will be with Y set to zero, in order to locate a field within a single current record. If you are using a list of records, then you must not set Y to too high a value, so that the routine runs off the end of the list, unless you have an end of file marker after the last record.

Before you use this routine, you must have set the position of the first record in the list by using the SETDPS call.

| | | |
|---|---|---|
| On entry: | A | field number of field to locate; field start at 1. |
| | X=0 | find field in header record. |
| | X=1 to X=254 | find field in format file record; X gives the number of the format file record. |
| | X-255 | find field in list of current records; Y has the number of the record to search, starting at zero. |
| | Y | only significant if X=255. |
| On exit: | CS | field or record not found; TEMP06 (and TEMP07 if X=255) point to the end of record marker if field not found, the end of file marker if record not found. |
| | CC | field and record found; TEMP06 points to the beginning of the field; if the call was made with X=255 then TEMP07 also points to the beginning of the field. |
| | A,Y | undefined. |
| | X | preserved. |

**GETFRC**

This call first sets X to 255, and then calls the GETFLD routine. The entry and exit conditions are as for GETFLD when X-255, except that X will always return set to 255.

**GETXFL**

GETXFL returns the first non-space character in a field, folded to upper case if alphabetic. It is intended primarily for reading the value of single character fields in the format file, such as the "Field type" field.

GETXFL first calls the routine GETFLD. The entry conditions are the same as GETFLD. If the call to GETFLD fails, ie the Carry flag is set, then the A register is cleared, and the routine ends. If the field is found, then the first character of the field is returned, folded to upper case if alphabetic. If the field is blank, then the end of field marker will be returned.

| | | |
|---|---|---|
| On entry: | | See GETFLD |
| On exit: | | TEMP06 and TEMP07 set as for GETFLD. |
| | CS | field or record not found, as GETFLD; A set to zero. |
| | CC | field found; A contains first non-space character, folded to upper case if alphabetic. |
| | X,Y | See GETFLD. |

**SETDPS**

This routine stores the address of the records to be used when using one of the field locate routines with X set to 255. It should be called whenever the address of one of the records in the list or of one of the fields in the records has altered. It need not be called if none of the fields has moved, since ViewStore will keep track of its position in the list of records, and move backwards or forwards as necessary to find the field you have asked for.

If you are reading records one by one using the NXTIMF call, for example, then you must call SETDPS with their address of the record for each record that you read: the alignment of the fields will alter for each record.

| | | |
|---|---|---|
| On entry: | A | contains low byte of the address of the first record in the list. |
| | Y | contains the high byte of the address of the first record in the list. |
| On exit: | A,X,Y | undefined. |

**CHKEOF**

CHKEOF checks the character in the A register for an end of field marker. It should be used rather than checking for the character explicitly since it handles classes of characters rather than single values. Generally, it is not necessary to detect illegal characters explicitly, it is enough to detect them as an end of field marker.

| | | |
|---|---|---|
| On entry: | A | contains character value to be checked. |
| On exit: | EQ | end of field. |
| | CS | end of record (EQ also set). |

| | | |
|---|---|---|
| | VS | illegal character (EQ also set). |
| | A,X,Y | preserved. |

**CHKEOR**

CHKEOR checks the character in the A register for an end of record marker.

| | | |
|---|---|---|
| On entry: | A | contains character to be checked. |
| On exit: | EQ | end of record. |
| | CS | end of file (EQ also set). |
| | VS | space character (EQ also set). |
| | A,X,Y | preserved. |

**SIZFLD**

SIZFLD is provided to allow you to determine the size of a field. First of all you should locate the field, using one of the field locator routines such GETFLD, which set up TEMP06. Then call SIZFLD.

| | | |
|---|---|---|
| On entry: | TEMP06 | points to the beginning of the field. |
| On exit: | A,Y | have length of field in characters. |
| | EQ | zero length field. |
| | X | preserved. |
| | TEMP06 | preserved. |

**CMPFLD**

CMPFLD compares the values of two fields of the same type, and sets the 6502 flags register like the CMP instruction. TEMP07 points to the first field (equivalent to the contents of the 6502 A register in the CMP instruction), and TEMP06 points to the second field. If the two fields being compared are strings, then wildcards are allowed in the second string.

| | | |
|---|---|---|
| On entry: | A | contains the field type: A, N, D or Y; must be in upper case. |
| | TEMP07 | points to field 1. |
| | TEMP06 | points to field 2. |
| On exit: | VS | error in one of the fields supplied: eg illegal date; result not valid. |
| | C flag | set according to compare. |
| | Z flag | set according to compare. |
| | A,X,Y | undefined. |

**SCHFLD**

SCHFLD is used to find the number of a field, given its name. It searches the list of fields in the format file until it finds one that fits the name given. The name that you specify can obtain wild cards: the single wildcard "?", and the multiple wildcard "*" are both allowed. SCHFLD will always return the first field in the format file that fits the name you have given. You can either continue searching for more fields by using the SCHFLD call described next.

The name of the field is set up in the 16 byte LWORK area. It should be terminated by a null, or an end of field marker. It must not be longer then 16 bytes, including the delimiter.

| | | |
|---|---|---|
| On entry: | LWORK | contains name to search for; maximum of 16 bytes including delimiter; delimiter null or end of field marker; wildcards "?" and "*" valid. |
| On exit: | CS | no field found to match the name; A has error code. |
| | CC | field found OK; field number in X. |

**SCHFLN**

After calling SCHFLD, you can search for other fields which also fit the field specification that you gave given, by making repeated calls to SCHFLN. Before you call SCHFLN, you must have called SCHFLD first, to start the sequence, and this call must have successfully found a field.

You can keep calling SCHFLN until the call returns with the Carry flag set to indicate that it has found no more fields.

| | | |
|---|---|---|
| On entry: | | Must have called SCHFLD first, and this must have returned with the Carry flag clear. |
| | A | Contains field number to start searching from. This should be one more than the last value that SCHFLD or SCHFLN returned in X. |
| On exit: | CS | no more fields found; A has error code (report only if required). |
| | CC | field found; field number in X. |

**GETWID**

GETWID returns the display width of a field, as defined in the format file. If there is no display width defined, a series of defaults comes into action.

| Display width defined | Display width |
|---|---|
| No display width, Sheet mode | 18 |
| No display width, Card mode | 0 |

| | | |
|---|---|---|
| On entry: | X | contains number of field for which width is required. |
| On exit: | CS | field doesn't exit. |
| | CC | field found; A has display width. |
| | X | preserved. |
| | Y | undefined. |

**GETKYW**

This routine finds the key width for a given field. It uses the value defined in the format file, if any; otherwise a system of defaults operates:

| | |
|---|---|
| Key width defined | Key width |
| No key width, display width defined | Display width |
| No key width or display width | 10 |

| | | |
|---|---|---|
| On entry: | X | contains number of field for which width is required. |
| On exit: | CS | field doesn't exist. |
| | CC | field found; A has already width. |
| | X | preserved. |
| | Y | undefined. |

**GETKYW**

This routine finds the key width for a given field. It uses the value defined in the format file, if any; otherwise a system of defaults operates:

| | |
|---|---|
| Key width defined | Key width |
| No key width, display width defined | Display width |
| No key width or display width | 10 |

| | | |
|---|---|---|
| On entry: | X | contains the field number for which the key width is required. |
| On exit: | A | contains key width; if field doesn't exist, a default key width of 10 is returned. |
| | X | preserved. |
| | Y | undefined. |

**CALSBN**

This routine is used when displaying numeric fields, to calculate the number of spaces to be output before the number in order to right justify the number within its field width. This also takes into account the decimal places specified in the format file.

| | | |
|---|---|---|
| On entry: | X | contains the field number. |
| | TEMP07 | points to the beginning of the field in question. |
| | TEMP03 | contains the field display width, as returned by the GETWID routine. |
| On exit: | A | gives number of spaces to output, zero if the number is wider than the field, or there are too many decimal places. |
| | X | undefined. |
| | Y | undefined. |
| | TEMP03 | preserved. |
| | TEMP07 | preserved. |

**File Control**

File handling in ViewStore is centred around three things: error handling, FBLOCK and prefixes. Since the error handling provided by the normal filing system interface provided is completely unsatisfactory for a program such as ViewStore, I have developed a system which gives control of what happens after a disc or filing system error. For the utility writer, this system is transparent: you can forget about it as long as you use the calls provided, and don't call the filing system directly. If you do this, you can forget all about BRK errors and handlers.

The system is the same for all file calls: the state of the Overflow (V) flag indicates after a call whether an error has occurred. If there has been an error, then the V flag is set, and the A register contains the error code. If you detect an error, you should unravel yourself from any routines, report the error by calling the REPERL routine, and then close any files that you have opened yourself, before returning to control of the ROM. You shouldn't close the intermediate file: this is done automatically when control is passed back to the ROM.

**Filenames**

FBLOCK is a small of memory used to store and manipulate filenames. Several routines are provided which work on the filename in FBLOCK, altering directories and prefixes.

A filename in ViewStore is made up of threes parts:

Prefix
Directory
Name

ViewStore maintains a list of the current prefixes for each different file type: data; format; sort and so on. A routine which adds a specified prefix to a directory and name stored in FBLOCK is available. The maximum length of a prefix is 13 characters, excluding delimiter. The current prefixes can only be altered with the PREFIX command in ViewStore's Command Mode.

ViewStore considers directories to be single character; of course the prefix can include multiple character directories, but the filenames of data and format files, for example, begin with directories, and these are always single character directory names, whatever the filing system. Directories must be separated from the name itself by a dot, making the total size of the directory section of a filename 2 characters.

The name part of the filename can be up to 10 characters long. This does not include the directory and separator, or the delimiter. Names are always delimited with a Carriage Return character.

| Part | Max. Size (exc. delimiter) | Example |
|---|---|---|
| Prefix | 13 | :2. |
| Director | 2 | d. |
| Name | 10 | datafile |

| | |
|---|---|
| MOVFBK | Moves a filename from the store area into FBLOCK. |
| MOVNAY | Moves a filename out of FBLOCK into the store area. |
| CHKDIR | Checks for the presence of a directory in a filename in FBLOCK, and returns the directory character, if there is one. |
| SETDIR | Sets a directory into a filename in FBLOCK. |
| STXPRE | Stores the required prefix with a filename in FBLOCK. |
| OPFILE | Gives access to the filing system OSFIND call. |
| OSHCAL | Gives access to the filing system OSFILE call. |
| XOSARG | Vector with error trapping to OSARGS. |
| XOSBGE | Vector with error trapping to OSBGET. |
| XOSBPU | Vector with error trapping to OSBPUT. |
| XOSCLS | Vector with error trapping to OSFIND with A=0; used to close a file. |
| XOSGBP | Vector with error trapping to OSGBPB. |
| CALUTI | Loads and runs a utility format file. |

**MOVFBK**

MOVFBK moves a filename from ViewStore's list into FBLOCK. A summary of the filenames available and their offsets is given in table 8. Filenames are not stored with prefixes attached, but they do include the directory. You must use the routine STXPRE to add a prefix to the filename once it has been moved to FBLOCK, before calling one of the file routines.

| | | |
|---|---|---|
| On entry: | Y | contains the offset of the filename to be moved into FBLOCK. |
| On exit: | A,X,Y | undefined. |

**MOVNAY**

This routine is the inverse of MOVFBK; it moves a filename from FBLOCK into ViewStore's list of names. The filename in FBLOCK should not include the prefix when this routine is called. The list of filename offset values is given in table 8.

| | | |
|---|---|---|
| On entry: | X | contains the offset of the filename area in which the name currently in FBLOCK is to be stored. |
| On exit: | A,X,Y | undefined. |

**CHKDIR**

CHKDIR checks whether the filename in FBLOCK has a directory, and if it does, it returns the directory. The filename should not include the prefix when this routine is called.

| | | |
|---|---|---|
| On entry: | | filename to check in FBLOCK. |
| On exit: | CS | filename contains directory; directory character found returned in A; offset from beginning of FBLOCK to the directory character is in X. |
| | CC | no directory found. |
| | Y | undefined. |

**SETDIR**

SETDIR forces a directory into the filename in FBLOCK. It doesn't matter whether the filename there contains a directory or not; SETDIR will make space for it. The filename must not include a prefix when SETDIR is called.

| | | |
|---|---|---|
| On entry: | A | contains directory character to set into name in FBLOCK. |
| On exit: | A,X,Y | undefined. |

**STXPRE**

This is the routine that you use to add a prefix to a name. The name itself, including directory, should be in FBLOCK. Normally, adding the prefix to a name is the last thing you do before calling the filing system to do some operation on the file: opening or deleting the file, for example. Names themselves should be stored without prefix attached, the prefix being added only when calling the filing system itself.

A list of the prefix offsets for different file types is given in table 7.

| | | |
|---|---|---|
| On entry: | X | contains the offset of the prefix required. |
| On exit: | A,X,Y | undefined. |

**OPFILE**

OPFILE is an equivalent of the filing system "OSFIND" call, used for opening files. It assumes, however, that the filename is ready in FBLOCK. OPFILE cannot be used to close a file, as OSFIND can; use the XOSCLS call to close a file. OPFILE also uses the ViewStore error trapping system.

| | | |
|---|---|---|
| On entry: | A | contains file open code, as for OSFIND; eg. &40 is open for input. |
| On exit: | VS | error occurred; error code in A. |
| | VC | no error occurred; file handle is in both A and Y registers; usual OSFIND error of file handle being zero when the file can't be found is trapped: V is set, and the ViewStore "File not found" error code is in A; Y is zero. |
| | X | undefined. |

**OSHCAL**

OSHCAL gives the utility access to the filing system OSFILE routine. It uses the OSFARA area as its control block. Note that this spills over into the LWORK area, which will be corrupted after an OSHCAL call. OSHCAL assumes that the filename (when required) is set up in FBLOCK. OSHCAL also sets the high order addresses into the control block at OSFARA +4 and +5; +&C and +&D; and at +&10 and +&11. ViewStore error trapping is also enabled.

| | | |
|---|---|---|
| On entry: | A | contains reason code as for normal OSFILE (see filing system manual). |
| | OSFARA | set up with whatever the reason code action requires, eg. start and end addresses for file save. |
| On exit: | VS | error occurred; error code in A. |
| | VC | completed successfully; file type in A when relevant. |
| | X,Y | undefined. |

**XOSARG**

XOSARG is equivalent to OSARGS, except that ViewStore error handling is enabled.

| | | |
|---|---|---|
| On entry: | | See filing system manual for OSARGS. |
| On exit: | VS | error occurred; error code in A. |
| | VC | no error; see filing system manual for results. |

**XOSBGE**

XOSBGE is equivalent to OSBGET, except that ViewStore error handling is enabled.

| | | |
|---|---|---|
| On entry: | | See filing system manual for OSBGET. |
| On exit: | VS | error occurred; error code in A. |
| | VC | no error; see filing system manual for results. |

**XOSBPU**

XOSBPU is equivalent to OSBPUT, except that ViewStore error handling is enabled.

| | | |
|---|---|---|
| On entry: | | see filing system manual for OSBPUT. |
| On exit: | VS | error occurred; error code in A. |
| | VC | no error; see filing system manual for results. |

**XOSCLS**

XOSCLS is equivalent to OSFIND with A=0, in order to close a file. ViewStore error trapping is also enabled.

| | | |
|---|---|---|
| On entry: | Y | contains handle of file to be closed. |
| On exit: | VS | error occurred; error code in A. |
| | VC | file closed successfully. |
| | X,Y | undefined. |

**XOSGBP**

XOSGBP is equivalent to OSGBPB, except that ViewStore error handling is enabled.

| | | |
|---|---|---|
| On entry: | | See filing system manual for OSGBPB. |
| On exit: | VS | error occurred; error code in A. |
| | VC | no error; see filing system manual for results. |

**CALUTI**

CALUTI is provided to allow utilities to call other programs stored in utility file format. The SELECT utility, which is provided with ViewStore, uses this call to call the SORT program when it is required. Parameters can be passed between the programs using temporaries, or other memory space. The CALUTI call can be thought of as the logical equivalent of the Basic CHAIN statement.

The utility should exit under the prefix given for utilities; CALUTI applies the utility prefix to the name given automatically. If the utility is not found, the "Insert utility disc and hit a key" message will be generated, and ViewStore will wait until a key is pressed before continuing.

| | | |
|---|---|---|
| On entry: | A | contains low byte of address of name of utility to be loaded. Name must be terminated by Carriage Return. |
| | Y | contains high byte of address of name of utility to be loaded. |
| | TEMP14 | contains the address of the first byte of free memory that the utility being loaded is to use. This will normally be the same value as passed to the initial utility. |
| On exit: | VS | error occurred; error code in A. Otherwise, control is passed to the new utility. |

**Floating Point**

Unfortunately, a discussion of floating point is outside the scope of this document. However, certain of the key routines in a floating point package are in the ROM itself, and entry points to these routines are provided. The REPORT utility which is provided with ViewStore uses these routines, and implements many more inside itself. The routine most obviously missing from this set is a routine to output a floating point number in ASCII. If you wish to do this, you will have to work out how to do it yourself.

I can refer you to the "Advanced Basic Rom User Guide", published by the "Cambridge Microcomputer Centre", which contains useful information about the floating point package in Basic, and how it works. The floating point in ViewStore works in the same way.

Remember that there is no need to understand floating point to write a utility, since the date in ViewStore files is stored in ASCII format, not as floating point numbers. Use of floating is only necessary when you wish your utility to provide floating point arithmetic.

The two accumulators, FWRK and FACC are in zero page. If you are not using the floating point package in your utility, you can use the zero page allocated to the accumulators for your own purposes.

| | |
|---|---|
| FONE | Sets FACC to value of one. |
| FTENFX | Multiplies FACC by 10 (not normalised). |
| FTENGQ | Divides FACC by 10 (not normalised). |
| FADDWI | Adds FACC and FWRK; answer in FACC, not rounded. |
| FTENX | Multiplies mantissa by 10. |
| FRDDK | Reads in ASCII number to FACC. Low byte of address of string in A; high byte of address of string in Y. |
| FTST | Tests number of FACC and sets flags. |
| FNEG | Swaps sign of number in FACC. |
| FCLR | Sets FACC to zero. |
| FADDW | Adds FACC and FWRK; answer in FACC, rounded. |
| FDIVA | Divides FACC by FWRK; answer in FACC. |
| FMULX | Multiples FACC by FWRK; answer om FACC. |

**General**

These routines are an assortment of useful subroutines for which entry points are provided.

| | |
|---|---|
| GETDEC | Get a decimal number. |
| KINCH | Flush keyboard buffer, and get an input character. |
| MULPLY | Multiply two single byte integers. |
| OUTDEC | Output a decimal number. |
| PSTRNG | Output a string. |
| RELLIN | Read a line of input. |
| SKPCBL | Skip blanks. |

**GETDEC**

GETDEC reads a number as an ASCII string and converts it into binary. The maximum size of a binary number that it can return is two bytes. No errors are generated for overflows; GETDEC will return the bottom sixteen bits of an arbitrarily large number.

| | | |
|---|---|---|
| On entry: | TEMP06 | points to start of string. |
| | Y | gives offset from beginning of string to start of ASCII number; no leading blanks allowed. |

| | | |
|---|---|---|
| On exit: | A | contains low byte of number. |
| | X | contains high byte of number. |
| | Y | is updated to point to the non-numeric character that terminated the number. |
| | EQ | no number was found; A,X,Y undefined. |

**KINCH**

Call this routine to get a character of input from the keyboard. The keyboard buffer is flushed first. ESCAPEs are detected and acknowledged automatically, using the OSBYTE 126 call, and the Carry flag indicates whether ESCAPE was detected.

| | | |
|---|---|---|
| On entry: | | No entry conditions. |

| | | |
|---|---|---|
| On exit: | CC | A contains input character. |
| | CS | ESCAPE was detected. |

**MULPLY**

MULPLY multiplies two eight bit numbers together, giving a sixteen bit result:

| | | |
|---|---|---|
| On entry: | A | contains an 8-bit number. |
| | Y | contains an 8-bit number. |

| | | |
|---|---|---|
| On exit: | A | contains the low byte of the result. |
| | Y | contains the high byte of the result. |

| | | |
|---|---|---|
| | CS | if Y is non-zero (the result is greater than 255). |
| | X | preserved. |

**OUTDEC**

Call OUTDEC to output a sixteen bit decimal number to the VDU.

| | | |
|---|---|---|
| On entry: | X | contains the low byte of the number. |
| | Y | contains the high byte of the number. |
| On exit: | A,X,Y | undefined. |

**PSTRNG**

PSTRNG outputs a string in-line with the code to the VDU. The string must be delimited with a null, zero byte.

For example:

```
JSR     PSTRNG
EQUS    "This will be output to the VDU"
EQUB    0
```

| | | |
|---|---|---|
| On entry: | | Immediately following the JSR call, there is a string delimited with a null. The string must not be more then 256 characters long, including the delimiter. |
| On exit: | A,Y | undefined. |
| | X | preserved. |

**RELLIN**

RELLIN reads a line of input, and puts it into LINBUF. The input is terminated with a Carriage Return or an ESCAPE. The OSWORD 0 call is used to get the input. TEMP06 is left pointing to the beginning of LINBUF, and Y gives the offset to the first non-space character in LINBUF.

| | | |
|---|---|---|
| On entry: | | No entry parameters. |
| On exit: | LINBUF | contains the line of input; maximum 256 characters. |
| | TEMP06 | points to LINBUF. |
| | CMDPAR | gives offset from TEMP06 to the first non-space character. |
| | CS | ESCAPE terminated input. |

**SKPCBL**

SKPCBL skips spaces in a line of input, terminated by a Carriage Return.

| | | |
|---|---|---|
| On entry: | TEMP06 | points to beginning of input string. |
| | CMDPAR | gives offset from start of string to start skipping spaces. |
| On exit: | CMDPAR | gives offset to first non-space character after initial value. |
| | A | contains first non-space character. |
| | EQ | hit CR at end of line, before non-space character was found. |

**The Index System**

The ViewStore ROM contains a set of routines for creating and maintaining index files. These are used internally by code in the ROM, and also by the INDEX utility. They are very powerful and could be used by extra utilities to great effect.

You will be familiar with the way the ViewStore itself uses the index system. A utility could use indexes side by side with the ROM, or it could build and maintain indexes for its own purpose.

In a ViewStore index file, you can store a "key", and associate with the key a pointer value, 4 bytes in size. They key can be any string of ASCII characters, and the pointer value any four byte integer, but usually the pointer value is used to remember a record file address.

The characters that make up an index key must be between the values 32 and 254, inclusive. Since the alphabetical value of numbers and dates is not the same as the value we generally wish these data types to have (that is numbers in numerical order, and dates in age order), the ASCII number and date fields as they are stored in ViewStore data files must be converted into another form before being sent to the index system. A routine, ADJVAL, is provided to do this. Remember, if you are building or altering an index file, to use the ADJVAL routine on the key.

The index system uses a technique akin to that of IBM's ISAM (indexed sequential access method) and VSAM (virtual sequential access method) systems.

Nearly all the calls to alter an index file are made through one routine, with a reason code: ISAM.

ISAM uses some workspace in the language workspace area. If you re not using the index system, the utility can use this workspace for its own purposes.

| | |
|---|---|
| ADJVAL | Routine to adjust values of different key types. |
| CCRTIX | Create an index file. |
| GETIXN | Return index name for a given field number. |
| IDXSCH | Search for an index, given a field name. |
| ISAM | General entry point to ISAM package. |

**ADJVAL**

This routine is called before sending a key to ISAM for an operation. It adjusts the value of number and date fields into "index format", ready for ISAM.

If you give it a date value to adjust, it will check the validity of the date as it is processing the value. An error code is returned if a problem is found with the date; in this case the value left in the buffer will be legal, but will be an incorrect conversion from the date that you supplied.

| | | |
|---|---|---|
| On entry: | X | has field number of field being adjusted. |
| | LINBUF | has field value to be adjusted, delimited with an end of field marker. |
| On exit: | LINBUF | contains adjusted field value. |
| | CS | error was found in date value. |

**CRTIX**

CRTIX is used to create a new index file. You must supply the name of the file, and the number of bytes of disk space that you wish to reserve for the file. If a file with the name that you have exists already, it will be overwritten.

The maximum amount of space that you can reserve is 65535 bytes.

| | | |
|---|---|---|
| On entry: | FBLOCK | has filename of file that you wish to create, with PREFIX already inserted. |
| | A | contains keysize of file. |
| | X | contains low byte of number of bytes to reserve. |
| | Y | contains high byte of number of bytes to reserve. |
| On exit: | VC | file created OK. |
| | VS | error occurred; error code in A. |

**GETIXN**

GETIXN extracts the name of the index for a particular field from the format file, and places it in FBLOCK, with the index prefix automatically inserted.

| | | |
|---|---|---|
| On entry: | X | contains the field number of the field for which you require the index name. |
| On exit: | CC | no error; name is in FBLOCK with prefix. |
| | CS | error occurred; error code is in A. |

**IDXSCH**

Given a field name specification (which may include the wildcards "?" and "*"), IDXSCH looks for a field with this name, and checks whether this field has an index switched on.

| | | |
|---|---|---|
| On entry: | LWORK | contains field name to search for, which may contain wildcards. |
| | A | contains field number of field to start searching from. |
| On exit: | CC | field found OK; X contains number of field found. |
| | CS | no field found; error code in A. |

**ISAM**

ISAM is the routine that you call to perform operations on an index file. The reason code of the operation you wish to perform is loaded into A. A summary of reason code values is given at the end in table 4.

All calls to ISAM update the Carry and Overflow flags. The carry flag indicates whether an "internal" error occurred - such as "No key found". The Overflow flag indicates when a filing system error occurred - such as "Disk fault". A list of internal ISAM errors is given in table 5. Note that you can't call the error reporting routine REPERL with an internal ISAM error code; the internal code is only for checking within a program.

Key values are passed to ISAM in LINBUFl 4 byte pointer values are passed in REG1. ISAM can handle a maximum of nine indexes open at one time. The maximum size of a key is 105 bytes.

Indexed sequential files have the two features that you can locate a particular key by giving its value, and that you can also read up and down the index in key order. ISAM works by having a "position". Certain calls set the file "position", some move the position up and down, and some calls destroy the position altogether. The "Search" call sets the file position; the "Next" and "Previous" calls move the position, and the "Insert" and "Delete" calls destroy the position.

If you execute a "Next", or a "Previous" call on an index file with no position, the index is said to be set at the beginning.

The A, X and Y registers are all undefined after a call to ISAM.

Reason code Effect

A=ISMFLO Tell ISAM that file is open.

Before you make this call, you should have opened the file with the filing system. This call just informs ISAM that you have opened the file, and tells it to reverse some workspace for the file.

| | | |
|---|---|---|
| On entry: | Y | contains handle of already opened file, as returned by the filing system. |

| | | |
|---|---|---|
| On exit: | | The file position is reset. |
| | VS | file already open; internal error code in A. |
| | CS | internal error occurred; internal error code in A. |

A=ISMSCH Search for key in index.

This call attempts to find the key in LINBUF in the index that you specify. If no key is found, the index is still "positioned", and you can use the "Get next key" and "Get previous key" calls. A subsequent "Get next key" call after a key was not found, returns the next highest value key that is in the index.

| | | |
|---|---|---|
| On entry: | Y | contains the handle of the file. |
| | LINBUF | contains key to seaborne for. |
| On exit: | VS | filing system error occurred; error code in A. |
| | CS | key not found; internal error code in |
| | A. | The file position is set. |
| | REG1 | contains pointer value of key, if found. |

A=ISMINS Insert key into index.

This call inserts the key in LINBUF into the index file.

| | | |
|---|---|---|
| On entry: | Y | contains the handle of the file. |
| | LINBUF | contains the key to insert. |
| | REG1 | contains pointer value to be associated with the key. |
| On exit: | | The file position is reset. |
| | VS | filing system error occurred; error code in A. |
| | CS | index is full; internal error code in A. |

A=ISMNXT Return next sequential key.

This call returns the pointer value of the next sequential key, from the current file position.

| | | |
|---|---|---|
| On entry: | Y | contains the file handle. |
| On exit: | VS | filing system error occurred; error code in A. |
| | CS | At end of file; internal error code in A. The file position is advanced by one. |

| | | |
|---|---|---|
| | REG1 | contains the pointer value associated with the key. |

A=ISMDEL Delete key from index.

This call deletes the specified key and associated pointer value from the index file.

| | | |
|---|---|---|
| On entry: | Y | contains the file handle. |
| On exit: | | The file position is reset. |
| | VS | filing system error occurred; error code in A. |
| | CS | Key not found; internal error code in A. |
| | REG1 | contains the pointer value associated with the key. |

A=ISMCLS Close file.

This call closes the file; it both calls the filing system to close the file, and closes the file within ISAM as well. This is slightly different from the ISMOPN call, which requires the filing system open call to be separate.

| | | |
|---|---|---|
| On entry: | Y | contains the file handle. |
| On exit: | VS | filing system error occurred; error code in A. |

A=ISMPRE Return previous key in index file.

This call returns the pointer value of the previous sequential key in the index, from the current file position.

| | | |
|---|---|---|
| On entry: | Y | contains the file handle. |
| On exit: | VS | filing system error occurred; error code in A. |
| | CS | Beginning of file; internal error code in A. The file position is moved back by one. |
| | REG1 | contains the pointer value associated with the key. |

**Printer Control Routines**

The printer control routines handle the printer driver for the utility. A printer driver must be loaded from ViewStore Command Mode before calling the utility, or the default printer driver in the ROM is used.

Highlights can be sent to the printer driver from a utility, but there is no provision for handling of the printer options byte or microspacing. Highlights begin at 128 for highlight 1.

A utility can test the state of the printer by examining the location PRNFLG. If bit 7 of PRNFLG is set, then the printer is switched on. If bit 6 of PRNFLG is set, then the printer is not actually on, but waiting: a call to PRNON will switch on the printer.

The printer should be switched off before reporting errors.

| | |
|---|---|
| ASKPRN | Ask about printer. |
| PRNON | Switch waiting printer on. |
| PRNOFF | Switch printer off. |
| PSCOUTT | Send character to printer/screen vector. |

**ASKPRN**

This routine prompts the user with the question "Screen or Printer (S,P)?". According to the response, bit 6 of PRNFLG is updated to indicate whether the printer is waiting. If bit 6 is set, a subsequent call to PRNON by the utility will switch on the printer.

| | | |
|---|---|---|
| On entry: | | No entry conditions. |
| On exit: | PRNFLG | Bit 6 set if printer is waiting; clear if screen is to be used. |
| | A,X,Y | undefined. |

**PRNON**

If the printer is waiting, that is bit 6 of PRNFLG is set, a call to PRNON will switch it on, calling the printer on routine in the printer driver.

| | | |
|---|---|---|
| On entry: | | No entry conditions. |
| On exit: | A,X,Y | undefined. |

**PRNOFF**

If the printer is switched on, a call to PRNOFF will switch it off.

| | | |
|---|---|---|
| On entry: | | No entry conditions. |
| On exit: | PRNFLG | Bit 7 clear. Bit 6 unaltered. |
| | A,X,Y | undefined. |

**PSCOUT**

This routine vectors characters either to the screen or the printer, depending which is enabled. A utility which wishes to use the printer optionally should send all output to this routine, and in conjunction with the ASKPRN, PRNON and PRNOFF calls, output can be directed by the user of the utility to the screen or printer, depending on his answer to the ASKPRN question.

PSCOTT automatically strips off trailing spaces from printer output.

| | | |
|---|---|---|
| On entry: | A | contains character to be printed. Highlight codes begin at 128 for highlight 1. |
| On exit: | A,X,Y | preserved. |

**Summaries**

Table 1 - Routines and Addresses

| Routine | Address | Temporaries Altered |
|---|---|---|
| PSCOTT | &802A | |
| KINCH | &802D | |
| PRNON | &8030 | 10 |
| PRNOFF | &8033 | 10 |
| ISAM | &8036 | 01,02,03,04,05,06,08,10,14 |
| FONE | &8039 | |
| FTENFX | &803C | |
| FTENFQ | &803F | |
| FADDW1 | &8042 | |
| FTENX | &8045 | |
| GETXFL | &8048 | 04,05,06,06,11 |
| INIIMF | &804B | 00,04,05,06,07,10,11,12 |
| CMPFLD | &804E | 01,02,03,04,05 |
| RELLIN | &8051 | 06,07,08 |
| FRDDK | &8054 | 04,05,10 |
| FTST | &8057 | |
| FNEG | &805A | |
| FCLR | &805D | |
| FADDW | &8060 | |
| FDIVA | &8063 | LWORK |
| FMULX | &8066 | |
| REPERL | &8069 | 04,05,06,07,10,11,12 |
| SETDPS | &806C | |
| GETFRC | &806F | 04,05,06,07,11 |
| OUTDEC | &8072 | 04,10,11 |
| NXTIMF | &8075 | 03,04,05,10 |
| SCHFLD | &8078 | 02,03,04,05,06,07,11 |
| CHKEOR | &807B | |
| CHKEOF | &807E | |
| PSTRNG | &8081 | 12 |

| | | |
|---|---|---|
| MULPLY | &8084 | 05,10 |
| GETDEC | &8087 | 05,10 |
| ADJVAL | &808A | 01,02,03,04,05,06,07,10,11 |
| CALUTI | &808D | 03,04,05,06,07,08,09,10,11,12,LWORK |
| SKPCBL | &8090 | |
| STXPRE | &8093 | 05 |
| MOVFBK | &8096 | |
| MOVNAY | &8099 | |
| CHKDIR | &809C | 05 |
| SETDIR | &809F | 04,05 |
| OSHCAL | &80A2 | |
| CRTIX | &80A5 | |
| GETKYW | &80A8 | 02,04,05,06,07,10,11 |
| IDXSCH | &80AB | |
| GETIXN | &80AE | |
| GETWID | &80B1 | 04,05,06,07,10,11 |
| CALSBN | &80B4 | 04,05,06,07,10,11 |
| SIZFLD | &80B7 | |
| GETFLD | &80BA | 04,05,06,07,11 |
| SCHFLN | &80BD | 02,03,04,05,06,07,11 |
| ASKPRN | &80C0 | 06,07,08,12 |
| OPFILE | &80C3 | |
| XOSCLS | &80C6 | |
| XOSBGE | &80C9 | |
| XOSBPU | &80CC | |
| XOSGBP | &80CF | |
| XOSARG | 780D2 | |

**Table 2 - Field Numbers of the Header**

| | | |
|---|---|---|
| DIFDES | 1 | Description |
| DIFDSM | 2 | Display mode S/C |
| DIFRCS | 3 | Record size |
| DIFCAP | 4 | Capacity |
| DIFIDX | 5 | Index field |
| DIFSCM | 6 | Screen Mode |

**Table 3 - Field Numbers of the Format File**

| | | |
|---|---|---|
| RFFFNA | 1 | Name |
| RFFWID | 2 | Width |
| RFFTYP | 3 | Type |
| RFFPOX | 4 | X screen position |
| RFFPOY | 5 | Y screen position |
| RFFALS | 6 | Scroll Y/N |
| RFFDCP | 7 | Decimal places |
| RFFRLO | 8 | Low limit |
| RFFRHI | 9 | High limit |
| RFFIDX | 10 | Index Y/N |

| | |
|---|---|
| RFFKYW 11 | Key width |
| RFFIXN 12 | Index name |
| RFFPRO 13 | Prompt |
| RFFVLS 14 | Value list |

**Table 4 - ISAM Commands**

| | |
|---|---|
| ISMFLO 0 | open file |
| ISMSCH 1 | search |
| ISMINS 2 | insert key |
| ISMNXT 3 | next key |
| ISMDEL 4 | delete key |
| ISMCLS 5 | close file |
| ISMPRE 6 | previous key |

**Table 5 - ISAM Internal Errors**

| | |
|---|---|
| ISEFLO 0 | file already open |
| ISEKXP 1 | beginning of file for previous key |
| ISELSK 2 | end of file for next key |
| ISEKXI 3 | key already exits |
| ISENKF 4 | no key found |

**Table 6 - Memory Layout**

ViewStore variables available for read by the utility

| | | |
|---|---|---|
| VWSLIM | &B | memory limit |
| | | |
| FILMOD | &44 | editing file or not |
| PRNFLG | &47 | printer flag |
| XSSAVE | &48 | stack save for filing system |
| SSAVE | &49 | stack save for ISAM & FP |
| CURCHN | &4A | intermediate file channel |
| EFILE | &4B | main file channel |
| REG1 | &4C | 4-byte register |

**Temporaries**

| | | |
|---|---|---|
| TEMPFD | &50 | single byte temporaries |
| TEMPFE | &51 | |
| TEMPFF | &52 | |
| TEMP00 | &53 | |
| TEMP01 | &54 | |
| TEMP02 | &55 | |
| TEMP03 | &56 | |
| TEMP04 | &57 | |

| | | |
|---|---|---|
| TEMP05 | &58 | |
| TEMP06 | &59 | two byte temporaries |
| TEMP07 | &5B | |
| TEMP08 | &5D | |
| TEMP09 | &5F | |
| TEMP10 | &61 | |
| TEMP11 | &63 | |
| TEMP12 | &65 | |
| TEMP13 | &67 | |
| TEMP14 | &69 | |
| | | |
| VWSSTZ | &6B | start of zero page workspace |
| | | |
| FACCS | &6B | floating point accumulator |
| FACCXH | &6C | |
| FACCX | &6D | |
| FACCMA | &6E | |
| FACCMB | &6F | |
| FACCMC | &70 | |
| FACCMD | &71 | |
| FACCMG | &72 | |
| | | |
| FWRKS | &73 | floating point work accumulator |
| FWRKXH | &74 | |
| FWRKX | &75 | |
| FWRKMA | &76 | |
| FWRKMB | &77 | |
| FWRKMC | &78 | |
| FWRKMD | &79 | |
| FWRKMG | &7A | |
| | | |
| VWSXTZ | &7E | start of free if using FP |
| | | |
| OSFARA | &500 | OSFILE/OSHCAL work area |
| | | |
| LWORK | &50D | work area - 16 bytes |
| | | |
| FBLOCK | &563 | filename work area - 27 bytes |
| | | |
| VWSSTL | &5D3 | start of language workspace |
| LINBUF | &5DC | line buffer/ISAM key buffer (256 bytes) |
| VWSXTL | &6DC | language workspace after LINBUF |
| VWSITL | &799 | start of language workspace if using ISAM. |

**Table 7 - Offsets for Prefixes**

| | | |
|---|---|---|
| DATPRE | 0 | data prefix |
| FMTPRE | &E | format prefix |
| IDXPRE | &1C | index prefix |
| SRTPRE | &2A | sort prefix |

| | | |
|---|---|---|
| UTIPRE | &38 | utility prefix |

**Table 8 - Offsets for Filenames**

| | | |
|---|---|---|
| EFLNAM | &1B | data file name |
| FFLNAM | &2B | format file name |
| PRNAME | &35 | printer name |
| UTINAM | &42 | utility name |
| ARGNAM | &4F | name work area |

**Table 9 - Error Codes**

| | | |
|---|---|---|
| MEMERR | 1 | not enough money |
| MISERR | 2 | mistake |
| NEMERR | 3 | no end marker |
| BDFERR | 4 | bad file |
| ENDERR | 5 | end of data |
| NUMERR | 6 | not numeric |
| RANERR | 7 | range error |
| VLSERR | 8 | value not in list |
| TBGERR | 9 | overflow |
| REAERR | 10 | read error |
| RTBERR | 11 | record too big |
| BDIERR | 12 | bad directory |
| BDNERR | 13 | bad name |
| FLNERR | 14 | field not found |
| FNOERR | 15 | file not open |
| TMFERR | 16 | too many files |
| SKFERR | 17 | stack overflow |
| NOFERR | 18 | no index field |
| DATERR | 19 | bad date |
| NFSERR | 20 | no fields on screen |
| BDMERR | 21 | bad mode |
| ESCERR | 22 | escape |
| NDSERR | 23 | normal display |
| FDSERR | 24 | format edit disabled |
| DMOERR | 25 | data mode only |
| BPRERR | 26 | bad prefix |
| DCPERR | 27 | too many places |
| FNIERR | 28 | field not indexed |
| BDPERR | 29 | bad pointer |
| BFSERR | 30 | bad FS |
| FNXERR | 127 | x not found |

**Table 10 - File Format**

| | |
|---|---|
| &9 | end of field marker |
| &D | end of record marker |
| &1 | end of file marker |
| &3 | space character |
| &2 | deleted character |
| &0 | file pad character |

Each Field in the record is terminated by an end of field marker.

Each record in the field is terminated by an end of record record.

The file is terminated by an end of file marker.

Any expansion space in a record is represented by multiple space characters after the last end of field marker, but before the end of record marker.

A deleted record is represented by multiple deleted characters followed by an end of record marker.

Any padding space between the end of file marker and the physical end of file is filled with file pad characters.