17.

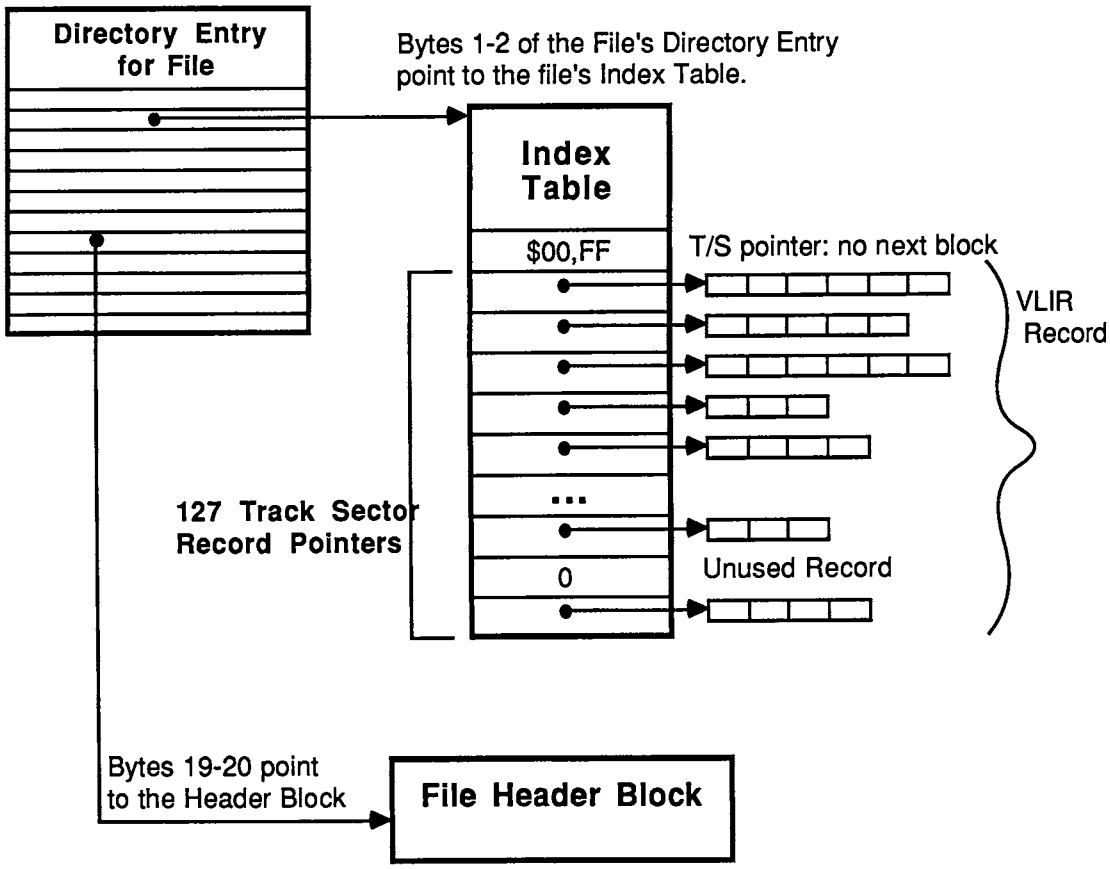
VLIR Files

The VLIR file structure was created to allow applications to grow much larger than the 30k available to them in GEOS. With a faster 1541 disk speed, it becomes practical to break an application up into several different modules, and swap them in as needed. A good way to organize such an application is to keep one module always resident while the others share a common memory area. The resident module is allowed to call subroutines in any of the other swap modules but the other modules may only call routines in the resident module. This keeps the application from getting bogged down with endless swapping. Applications tend to execute out of one module for a while, and then swap modules and execute out of another for a while.

A VLIR file is comprised of several modules referred to a records. Each record, is a chained link of blocks just like a regular Commodore file. Thus a VLIR file is somewhat like a collection of files. The same routines used to save a regular SEQUENTIAL file to disk may be used to save individual records in a VLIR file. In addition, several VLIR specific routines are provided.

The VLIR file routines allocate sectors on disk for records the same as is done for regular files, using the one block track/sector allocation table, fileTrScTab. Each record may therefore be from 0 to 127 blocks long, (just under 32k: 32,258 bytes), the maximum number of track/sector pointers fileTrScTab can hold. If the application uses the background screen buffer for program space, it has the use of memory from $400 to $8000 which is also just under 32k. An Index Table, holds the track/sector pointers to the first block in each record. The diagram below shows how the VLIR file uses an Index Table to organize the records in the file.

VLIR is an acronym for Variable Length Indexed Record. Both applications and data files may be stored in VLIR file. For example, the font files are divided into several records, one for each point size.



VLIR - Variable Length Indexed Rec ord File Structure

A VLIR file can be identified by looking at the GEOS Structure type byte in the file's Directory Entry. In addition, the Directory Entry contains a track/sector pointer to the file's Index Table. In a regular SEQUENTIAL file this word usually point to the first data block in the file. See the beginning of the file system section for more details on the Directory Entry structure. The Index Table consists of 127 entries, numbered 0 to 126, where each entry is a pointer to a record. The rest of the entries in the Directory Entry, such as the pointer to the Header Block, are the same.

VLIR Routines

The routines for reading and writing records, closely resemble those one might expect for manipulating objects in a linked list: NextRecord, PreviousRecord, and others.

This "linked list" concept makes use of a pointer to the current record. This pointer may be set directly or set to the next or previous record. The current record may be deleted, read from, or written to. At each access, the full record must be dealt with. Thus the application should provide sufficient RAM at any one time to accomadate the largest possible record it could be processing. New empty records may be inserted before, or appended after the current record. New Records are empty and may be written to. Presently there is no way to detach a record and re-attach it somewhere else. DeleteRecord is destructive, i.e., frees up the sectors, and InsertRecord only works with empty records.

The Index Table may be stored in memory, often in the fileHeader buffer, to make it possible to go directly to a record using PointRecord instead of advancing one record at a time with NextRecord or PreviousRecord.

An attempt has been made to return meaningful error flags concerning operations onthe structure. The following is a list of possible errors as returned in the x register by VLIR Record routines.

Error Messages

UNOPENED_VLIR_FILE

This error is returned upon an attempt to Read/Write/Delete/Append a record of a VLIR file before it has been opened with OpenRecordfile.

INVALID_RECORD

This error will appear if an attempt is made to Read/Write/Next/Previous a record what doesn't exist (isn't in the Index Table). This error is not fatal, and may be used to move the Record pointer to the end of the record chain.

OUT_OF_RECORDS

This error occurs when an attempt is made to Insert/Append a record to a file that alread contains the maximum number of records allowed (127 currently).

STRUCT_MISMATCH

This error occurs when a routine supporting a function for one type of file structure is called to operate on a file of different type.

Creating a VLIR File

Use the SaveFile routine to initially create a VLIR file. The File Header should contain the following values:

c64 File Type	- USER
GEOS File Structure Type	- VLIR
For Data Files:	
Start Address:	0
End Address:	FFFF (-1)
For Applications:	
Start Address:	Location to load the first record when the application is loaded.
End Address:	The Start Address - 1. This causes an empty VLIR structure to be created by SaveFile.

This creates a VLIR file on disk with an Index Table with no records. The current Record pointer is set to -1: a null pointer. Before any manipulation of the file is possible, it must be opened with OpenRecordFile. This loads certain internal buffers GEOS needs. With a completely empty Record file like this, the first record must be created with AppendRecord. After that calls to InsertRecord, AppendRecord, and Delete Record are possible.

When through with the file, it is imperative that the programmer close it by calling CloseRecordFile. This will update the file's Index Table, the disk BAM, and the "blocks used" entry in the file's Directory Entry. Note that at present only one VLIR may be opened at at time.

A description of the routines available specifically for VLIR files appears below.

OpenRecordFile

Function: Open an existing VLIR file for access given its filename.

Calls: FindFile, GetBlock

Pass: r0 - pointer to null-terminated **filename**

Initialized **Drive** - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - disk error status (0 =OK), see Appendix for disk errors.
fileHeader - **IndexTable** stored in fileHeader block.
usedRecords - **number of Records** in file.
fileWritten - flag indicating if file has been written to since last change to BAM or IndexTable. Zero = no change yet.
curRecord - Zero if at least one record in file, else set to -1 for empty structure
dirEntryBuf - **Directory Entry** for file.
curDirHead - The **Directory Header** of the disk.

Destroyed: a, y, r1, r4 - r6

Synopsis: OpenRecordFile sets up the RAM variables above as expected by the ReadRecord and WriteRecord routines. OpenRecordFile calls FindFile to check the disk for the file. If found, the values for several variables are retrieved and the file is error checked to make sure it is a VLIR file.

CloseRecordFile

Function: Update the VLIR file's IndexTable and the disk BAM. Indicate no open VLIR file.

Calls: UpdateRecordFile

Pass: **usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf** - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized **Drive** - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: IndexTable - updated
BAM - updated
fileWritten - reset to 0,
Directory Block- if record was modified, update the **Blocks Used** entry in theDirectory Entry. The **time/date** variables in the Directory Entry are updated from the year, month, day, hour, minutes, seconds variables in RAM.

Destroyed: a, y, r1, r4, r5

Synopsis: Calls UpdateRecordFile to update the Record variables mentioned above. If the file has changed since the last write, the time/date stamp in the Directory Entry is updated. An internal GEOS variable is set to indicate no presently open VLIR files.

UpdateRecordFile

Function: Update the VLIR file's IndexTable, disk BAM, and Time/date stamp.

Called By: CloseRecordFile

Calls: GetBlock, PutBlock

Pass: **usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf** - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: Index Table - updated
BAM - updated
fileWritten - reset to 0
Directory Block -update the **Blocks Used** entry in the Directory Entry. The **time/date** variables in the Directory Entry are updated from the year, month, day, hour, minutes, seconds variables in RAM.

Destroyed: a, y, r1, r4, r5

Synopsis: UpdateRecordFile updates the Record variables mentioned above. If the file has changed since the last write, the time/date stamp in the Directory Entry is updated.

PreviousRecord
NextRecord
PointRecord

Function: Adjust current record pointer to the previous, next, or to a specific record in the VLIR file.

Pass: a - Contains the **record number** for PointRecord. Not used for Next or PreviousRecord.

usedRecords, curRecord, fileWritten, fileHeader,curDirHead,dirEntryBuf - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: curRecord - current **record number**
x - **error status,**
0 - pointing to previous record
nonzero - location didn't exist. pointer unmoved.
y - **empty flag**: zero if no error but record is empty, else nonzero.
The actual value loaded into y is the track of the record as stored in the index table. This is zero if the record contains no blocks, i.e., nothing to point to.
r1L,r1H - The track and sector of the first block in the recordusedRecords,
fileHeader - unchanged

Destroyed: nothing

Synopsis: PreviousRecord, PointRecord and NextRecord adjust the current record pointer to point to a new record. You must pass a record number to point at to PointRecord. If some error occured in moving the pointer to a new record, such as calling NextRecord when already pointing to the last record, then the error condition is returned in x and the current record pointer is unchanged. If there was no error in reading the disk, the track and sector number of the first block of the requested record is retrieved form the index table and loaded into r1. r1L, the track, is copied to y. A track of 0 indicated an empty record.

DeleteRecord

Function: Deletes current record and leaves curRecord pointing to the following record.

Calls: GetDirHead (probably redundant)

Pass: **usedRecords, curRecord, fileWritten, fileHeader,curDirHead,dirEntryBuf** -These variables initialized by call to OpenRecordFile. file Headercontains index table.

Initialized **Drive** - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - **error status**, 0 = OK
Current Record - left pointing to **following record**, or last record in list.

Destroyed: a, y, r0 - r9

Synopsis: The Current Record is deleted and curRecord is left pointing at the following Record. If the deleted Record was the last Record in the VLIR file, then the CurrentRecord pointer is left pointing at the new last Record.

WriteRecord

Function: Writes contents of memory area out to Current Record.

Calls: GetDirHead, WriteFile, BlkAlloc

Pass: r2 - **Number of bytes** to write
r7 - **Beginning address** of data block in RAM to write to disk

usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized **Drive** - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - Disk **error status**, 0 = OK. See Appendix for disk errors.
fileHeader - Contains the updated **Index Table**
fileSize - Current **size in blocks** of record.
fileWritten - Loaded with $FF indicating file modified since opening.
fileTrScTab - Table of sectors used to store the record.
disk - Old Record deleted, **new Record written.**
r8L - number of data bytes stored in last sector.
r3 - Track and sector of **last block** allocated
curDirHead - **BAM** portion of Directory Header was modified to reflect allocation of new blocks to the record. Directory Header not written to disk. Use PutDirHead for that.

Destroyed: a, y, r0 - r9

Synopsis: The Current Record on disk is deleted (blocks freed in BAM) and the contents of the indicated block of RAM is written out. **Note**: The old version of the file is deleated before the new one is written. If there is not enough room on the disk to write the new version of the file, the old file will be lost. Use CalcBlocksFree to make sure the file can be written. The time/date stamp is not updated until the file is closed.

ReadRecord

Function: Reads Current Record into memory.

Calls: ReadFile

Pass: r2 - max **number of bytes** expected. If the Current Record contains more than this, a BUFFER_OVERFLOW error will be generated.
r7 - **beginning address** of data block in RAM to read into.
usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf -These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - Disk **error status**, 0 = OK. See Appendix for disk error listing.
r7 - Pointer to byte following **last byte** read in (if non-empty record)
a - **EmptyFlag**: 0 if record was empty.
fileTrScTab - Track/sector table for the record.
r5L - Points to the **last (word length) entry** in fileTrScTab. The value of this word is [$0, index to last data byte in block]
r1 - In case of BUFFER_OVERFLOW error, r1 contains the track/sector number of the **block which didn't fit**, and was not read in.

Destroyed: a, y, r1 - r4

Synopsis: The Current Record is read into memory at the indicated location. The fileTrScTab is built. The first two entries in fileTrScTab are identical: both store the track/sector of the first data block in the record.