IGORE Maintenance and Development Guide

Version 7.4

Gamma Ray Observatory
Oriented Scintillation Spectrometer Experiment

Naval Research Laboratory
Washington D. C.

NRL Document #926-224
Release Date: May 22, 1998

Editors: Gail Stephenson, John Adams, NRL

Contents

Chapter 1

Overview

IGORE (Interactive GRO OSSE Reduction Environment) is the main data analysis tool for data from the OSSE experiment. The various components fall into several major subsets.

IDL is a OTS sofware package from RSI. It features include fast array operations, strong math and statistics support, and a wide variety of plotting and display facilities. See the IDL documentation from RSI for a full description of commands and usage.

The IGORE system core is an executable image managing memory allocation and data conversion/presentation between IGORE applications and IDL procedures.

IGORE applications are generally FORTRAN sourced routines that run through a preprocessor before compilation. The preprocessor adds a front end routine, known as an AFE (Application Front End), which interfaces with IGORE system core routines to perform argument passing and conversion between IDL and the application routine. Existing FORTRAN pplication routines can be adapted and incorporated into the system relatively easily, the only source code changes being a handful of preprocessor directives to define the arguments. In some cases, preprocessor-generated AFEs have been replaced with hard-coded C sourced routines which allow IDL keyword parameters to passed to and from the "application routines". [reference AFE documentation, user's reference, cookbook?]

A collection of IDL procedures forms the fourth and final major subset of IGORE components. In most cases, several layers of IDL procedures are built up to form the primary user interfaces. In addition, a user may develop a set of IDL procedures appropriate for his or her particular needs. [reference user's reference, cookbook]

The general "structure" of IGORE is a tangled hierarchy. The following diagrams express this hierarchy. [section on IGORE startup should refer to these diagrams]

```
IGORE overview, user interfaces

----------------------------------------------
|            the happy user ;^)              |
|--------------------------------------------|
| IDL  | IDL procedures  | IGORE applications |
|_____________________________________________|


IGORE, the big picture

------------------------------------------
|            the happy user ;^)          |
|------------------------                |
|      |  IDL procedures |               |
|       ----------------------------------|
|               | IGORE applications      |
|               --------------------------|
|        IDL    | IGORE system core       |
|                --------------          |
|                             |           |
|-------------------------------         |
|________________________________________|
```

This document focuses on the IGORE system core and on its interactions with both IDL and VMS. This includes several "features" of IGORE that are dependent on undocumented features of VMS or IDL. For example, the technique used to create structured IDL records for the IGORE record pointers is highly dependent on undocumented IDL implementation details that are subject to change from one version to the next. See section ??. Another example is the technique used to modify the characteristics of the VM zone used by VAXC$MALLOC and therefore by IDL. While trouble-free to date, it depends on the consistent behaviour of what should be considered a quirk in the VMS RTL. See section ???.

No attempt is made to include documentation available elsewhere regarding IGORE, IDL, or VMS.

Chapter 2

IGORE Configuration Management

The selected tools for configuration management are VAX DEC/CMS (Code Management System) and VAX DEC/MMS (Module Management System). Refer to the associated manual for instruction in using either of these systems.

Also, see appendix X for the Igore development team CM policy.

2.1 CMS libraries

VAX DEC/CMS (Code Management System) is a library system which allows for strict control of all managed files. Access to each library is recorded in a history and all changes made to an individual file are tracked and retrievable by anyone with the proper privileges. Igore uses several CMS libraries to keep each "set" of code separate, but accessible by CMS. Each library contains a logically consistent grouping of source files.

In some cases the library will correspond directly to an Igore subdirectory. For instance the CMS library referred to as IGR_CMS_DAT (ODT2:[IGRCMS.DAT.CMSLIB]) contains all of the Igore data files, which can be found in the user directory IGR$DAT, or IGR:[DAT]. The source files for Igore routines written in IDL are located in the CMS library IGR_CMS_IDL (ODT2:[IGRCMS.IDL.CMSLIB]). This corresponds to the subdirectory IGR$IDL, or IGR:[IDL]. All of the "include" files for FORTRAN or C are kept in the CMS library IGR_CMS_ICL (ODT2:[IGRCMS.ICL.CMSLIB]) and all of the linker options files are maintained in IGR_REF_OPT (ODT2:[IGRCMS.OPT.CMSLIB]). Below is the complete list of libraries and a brief description of the contents.

```
Logical            Actual CMS library      Description of contents
IGR_CMS_AFE     ODT2:[IGRCMS.AFE.CMSLIB]   Source code for PREPRO tool
IGR_CMS_BENCH   ODT2:[IGRCMS.BENCH.CMSLIB] All routines to verify Igore
```

IGR_CMS_BLD	ODT2:[IGRCMS.BLD.CMSLIB]	MMS and DCL .COM files for building a complete Igore from CMS
IGR_CMS_CMMGR	ODT2:[IGRCMS.CMMGR.CMSLIB]	CM Manager tools
IGR_CMS_DAT	ODT2:[IGRCMS.DAT.CMSLIB]	All data files
IGR_CMS_DOC	ODT2:[IGRCMS.DOC.CMSLIB]	All documentation source files
IGR_CMS_ICL	ODT2:[IGRCMS.ICL.CMSLIB]	All include files for FORTRAN and C routines
IGR_CMS_IDL	ODT2:[IGRCMS.IDL.CMSLIB]	Source files for all routines written in IDL
IGR_CMS_OPT	ODT2:[IGRCMS.OPT.CMSLIB]	All linker options files
IGR_CMS_SDB	ODT2:[IGRCMS.SDB.CMSLIB]	All FORTRAN, C and, MACRO source code for building applications
IGR_CMS_ST	ODT2:[IGRCMS.ST.CMSLIB]	Source code build files and documentation for Structure Translator
IGR_CMS_STRUC	ODT2:[IGRCMS.STRUC.CMSLIB]	Structure definition files (source) and PICL (run-time system structures definitions)
IGR_CMS_SYS	ODT2:[IGRCMS.SYS.CMSLIB]	All FORTRAN, C and MACRO source code for Igore system

2.2 MMS Description

VAX DEC/MMS (Module Management System) is a tool that updates files by comparing the revision times of source files, which are specified in a description file. If a source file is newer than the target, MMS updates the source by executing commands, or rules, in the description file. MMS updates only those items which have been modified more recently than the date/time of the target. If the target does not exist, MMS will build it from its sources. In the case of Igore, a source could be an object module or library, an include file, or a FORTRAN, C or MACRO file.

2.3 CMS/MMS Usage

Contained in the CMS library IGR_CMS_BLD are all of the MMS description files to build the Igore system, the applications and preprocessors. The master file MAKEFILE.MMS will build a complete Igore system, using the target "COMPLETE_IGORE". It is possible to build subtargets from the file, such as "IGORE_PREPROCESSORS" or "IGORE_PDFTSF". The suggested macros to use are CLEAN=1 and CLEAN_SOURCE=1. Be aware that with the CLEAN_SOURCE macro any files you have in the current directory (where it will build Igore) which are named the same as any Igore source files will be deleted when MMS completes. The CLEAN macro will simply delete all of the temporary files which were created.

2.4 CMMGR tools

Because it is quite simple to update a interpreted IDL procedure, appropriately privileged users/programmers often update the run-time IGR$IDL directory, as well as the CMS library elements. For this reason DIFF_PRO is used to check for changes in any of the IDL files. Also, when a change is made only to the CMS library element, it is necessary to make sure that file is copied to the run-time directory once accepted by the CM manager or delegate.

2.4.1 DIFF_PRO.COM

Tool to check for any new procedures, changes to existing files or new files in the controlled Igore IGR$IDL area. Run by GETNU.COM procedure. Submits DIFF_PRO nightly. Check log GETNU.LOG at top directory for failures. DIFF_PRO.COM loops through all known Igore deliveries, checking for differences in the IDL files' dates and file identifiers. Uses a FORTRAN program GET_NAMES to collect the names of functions and procedures in each file. Uses several logicals to set up each installation.

- IGR$LOG_SYM (the IGR logical)
- IGR$LOG_LEN (Length of above)
- IGR$CMMGR_IN (input file specification to GET_NAMES)
- IGR$CMMGR_OUT (Output file specification written by GET_NAMES)

Output file IGR$CMMGR_OUT is a listing of routines in each parsed IDL file.

Output file with extension .IDS is a parallel listing of a file identification number and date stamp for each IDL file which had its routines listed in IGR$CMMGR_OUT.

After running GET_NAMES.EXE, will check differences of the .IDS file created and the one from the previous day's run. (ie. must have at least one residing in the directory before running DIFF_PRO for good results.) These differences reside in a file with extension .DIFF.

If differences exist, will generate differences of the IGR$CMMGR_OUT file as well to be checked. These differences reside in a file with extension .IDS_DIFF.

File .IDS_DIFF is mailed to CM Manager(s) for a decision on whether the changes are authorized or not.

2.4.2 CMS_HIST.COM

Run nightly to report all database changes in each CMS library for that day.

Re-submits itself nightly. Check log HIST.LOG at top directory for failures.

Must set up for which libraries to poll before running this file using the CMS SET LIBRARY command. Since this is run in batch mode in most cases, it is preferrable to set the library in the account's LOGIN.COM file for the case of non-interactive logins.

If there were any history records found (ie. through CMS HISTORY command), mails the listing of them (file CMS_HIST.LOG) to CM Manager(s) to be verified.

2.4.3 SET_CMS_PROT.COM

Used when CMS libraries are created to set the ACLs (both external and internal to CMS) for restricting use on commands and library files.

2.5 Release Scheme

For each release, there are two files USR_REL_NOTES.DOC and REL_NOTES.DOC which are, respectively, the User's release notes, organized alphabetically by topic/routine name, and the Programmer's release notes, organized chronologically.

Released as a set of savesets

2.6 Installation and Maintenance

2.6.1 NRL Installations

Built directly from CMS using main MMS file MAKEFILE.MMS after DCL command file MAKE_IGORE.COM for setting up proper directory structure.

2.6.2 OSSENU installations

Use VMSINSTAL routine to install from distribution kit. Kit consists of a series of savesets with the files to be restored; KITINSTAL.COM should be in the saveset "A". Installation must be coordinated with the OSSENU system manager as it is done from a privileged account. FTP all the savesets to an OSSENU machine and TELNET to OSSENU to run VMSINSTAL.

2.6.3 Distribution Kit

IGORE distribution kits are built for use with the VMS VMSINSTAL procedure. See the VMS documention for details on the VMSINSTAL procedure.

These files are located OD11:[IGOREKIT.BUILDKIT] directory. They will collect all of the needed files to subdirectories under OD11:[IGOREKIT], then build the savesets in the OD11:[IGOREKIT.BUILDKIT.IGRKIT_V0nnn] where nnn is the version number, such as 740. Before running any of the collection routines, it is necessary to "clean out" the subdirectories from the last version. There is currently no DCL routine to do this. Also, the OD11:[IGOREKIT.BUILDKIT.IGRKIT_V0nnn] directory must exist prior to the saveset building.

Routines for Collection

GET_IGORE.COM is the main collection routine; uses several others

KIT_FETCH_TESTBED.COM used by GET_IGORE to get TESTBED routines from CMS library

IGR_GET_IRM.COM used by GET_IGORE to get and build the instrument response matrix software from CMS library.

GET_DOC.COM used by GET_IGORE to collect and build all relevant documentation from CMS library.

BLD_LNK_OPT.COM builds the linker options files.

Routine for Building Savesets

IGR_BLD_KIT.COM arranges all of the pieces into separate savesets for easy delivery. Will prompt for source (device) and target (directory) locations.

2.6.4 KITINSTAL.COM

KITINSTAL.COM, for use by the VMS INSTALL procedure, is the main installation routine and needs to be maintained per release. It contains information regarding which versions of IDL Igore is compatible to and the release space requirements and quotas. It follows the rules for KITINSTAL.COM in the VMS documentation.

Chapter 3

IGORE Documentation

Unless otherwise noted, all Igore documentation is managed in the CMS library IGR_CMS_DOC and user copies are available in the directory IGR$DOC. It is assumed that the full IDL documentation set is available as well as the IDL internals manual.

3.1 Installation guide

This guide is delivered with the distribution kit in the documentation saveset (.L) and has several (hopefully) useful installation tips. The section on system requirements discusses space and parameter recommendations as well as supported VMS and IDL versions. There is a sample installation that should prepare the installer for what to expect. The postscript printable file is named IGR_INSTALL.PS.

3.2 Users' Reference

This document is written for the experienced Igore user. It documents calling sequence, keywords and notes any significant usages of the user routines. Some complex routines also contain examples. There are also several appendices that contain information about AUX types and give full definitions of all Igore structures. The postscript printable file is named IGR_USER_REF.PS.

3.3 User's Guide

The Igore User's Guide document is for the novice Igore user and will be instrumental in guiding through OSSE data analysis.

3.4 System Reference

A collection of previously undocumented Igore internal and external tidbits. What you are currently reading.

3.5 Structure Translators

The structure translators consist of a user translator (TRANSL.COM), accessed via the TRANSL command, and a system translator (TRANSYS.COM), accessed via the SYSTRANSL command. This support for structures and records is necessary because it is not provided in IDL and is required for any analysis with more than the simplest data structures. The (incomplete) documentation existing for the structure translators can be found in the CMS library IGR_REF_ST in the files named by : TRANSL_DESIGN.TEX and TRANSL_GUIDE.TEX. These are LaTex source files.

3.6 AFE Generator (PREPRO) command "AFEGEN"

The AFE (Application Front End) Generator is a preprocessor to add FORTRAN programs to the Igore system. The AFE preprocessor generates an AFE subroutine from the user's FORTRAN code which only needs to be modified to contain a few special delimeters. The generator is invoked with the command AFEGEN. The (incomplete) documentation existing for the AFE generator can be found in the CMS library IGR_REF_AFE in the files named by : PREP_GUIDE.TEX and PREP_DESIGN.TEX. These are TEX (?-LaTex) source files. These documents also discuss other preprocessors, such as AFE_PROCESS (aka GEN_VCTR) NOSHR_PSECTS (aka CHECK_COMMON) and the BUILDSHR routine.

Chapter 4

IGORE DevelopmentMaintenance Notes

4.1 IGORE Dynamic Memory Management

IGORE uses dynamic memory to store and access data for analysis and processing. Capability to move data between the IGORE dynamic memory and IDL variables is provided. The capability to move data between the IGORE dynamic memory and disk files is provided by various IGORE applications.

The dynamic memory used by IGORE for user data is managed by a nested hierarchy of IGORE structures that define, for example, chunks of memory containing SDB headers. This level of structures is in turn described by control structures maintained by the IGORE memory management subsystem.

4.1.1 Control Structures and IGORE Record Structures

The control structures comprising the IGORE memory management subsystem can best be understood by studying the diagrams and include files.

The root structures are four static structures loosely called "VM tables". Each table is the root structure for a particular class of IGORE dynamic memory. The table names are VMT$REC for "direct" IGORE records, VMT$STD for IGORE structure descriptors, VMT$IND for "indirect" records, and VMT$TMP for temporary allocations. Direct records can be thought of as records for which there exist IDL variables containing record pointers, while indirect records are referenced via other IGORE records. (See figure D.1)

Each VM table contains fixed arrays for "segment" pointers. The segment is a dynamically allocated but fixed size block of memory. Each segment is mapped as an array of segment-slots. The segment-slot structure contains the information required to track a single allocation unit of dynamic memory. (See figures, Appendix D)

New segments are allocated as existing segments are filled, until the segment pointer arrays are filled. The number of allocation units for each VM table is limited by the product of slots_per_segment*number_of_segments. The number of slots per segment is parameterized separately for the requirements of each table type. The size of each allocation unit is specified by the routine requesting the allocation.

In general the entity stored in the allocation unit can be an IGORE record, an IGORE structure descriptor, or simply a piece of dynamic memory allocated for the temporary use of an IGORE application routine. With the exception of the last case, some form of secondary control structure is used. Field descriptors, dynamic record headers (DRH's), structure descriptors, structure descriptor headers, and field descriptors can all be considered secondary control structures.

The secondary level of control structures sits between the raw memory management layer and the actual user/application data layer. The mapping of specific fields and records to dynamic memory is managed by this layer.

IGORE records have several possible secondary control structures along with the actual data, depending on whether the record is a direct record (figure D.2) or an indirect record (figure D.3). Note that in both cases there is also a structure descriptor, and structure descriptor header. These latter entities are stored, not surprisingly, in the structure descriptor table. The structure descriptor is simply an array of field descriptors, one for each field of the structure.

We reserve the term IGORE record for these actual application specific records. The conception for the IGORE kernel was that it could be used for various applications beyond the OSSE data analysis system. Application data structures can be added and modified relatively easily. In OSSE usage, these IGORE records include SDR's, SDB headers, OSSE aux data records, etc.

The record structures are generated from FORTRAN "include" files using the structure translator applications, creating binary files with '.PICL' extension. The PICL files are read at run-time to load the structure and field descriptors.

See Appendices for include files defining the IGORE control structures.

4.1.2 Fragmentation Issues

Memory fragmentation occurs when multiple chunks of dynamic memory are freed and reallocated in sequences that leave some chunks of unuseably small sizes. See the VMS Run-time Library documentation (section on dynamic memory management) for example scenarios.

Fragmentation can waste large quantities of virtual address space, particularly when both large amounts of dynamic memory are used and there is significant free and reallocate activity.

IGORE Dynamic Memory

The IGORE dynamic memory management system has been tuned to minimize the fragmentation of the dynamic memory it manages. VM zones, which are VMS RTL constructs, are used with parameters specifically set up for IGORE requirements.

IGORE internal initialization creates one VM zone for each of the four general IGORE dynamic memory categories; structure descriptors, direct records, indirect records, and temporary memory. The characteristics of each zone can be set up to suit the usage of that zone.

The parameters at issue are initial size and extend size. The initial size parameter, not suprisingly, determines the size of the first "area" belonging to the zone at creation time. Once the memory in that area has been allocated, or if the remaining free space is insufficient to satisfy a request, a new area is created. The minimum size of the new area is set by the extend size. If the size of the request is larger than the extend size, multiples of the extend size are used.

Most IGORE dynamic memory is consumed by the indirect records, so the zone used for them is sized very large, with very large extend sizes (2 MB and 1 MB). The other three types of memory are created with smaller allocations and have smaller extend sizes, reflecting observed usage. The parameters reside in VM_PARAMETERS.ICL, and if modified, require VM_MANAGEMENT.FOR to be recompiled and the IGORE system image to be relinked.

As a result of using separate VM zones with parameter settings optimized for observed usage, very little, if any, fragmentation occurs in dynamic memory used by IGORE. Fragmentation of dynamic memory not directly used by IGORE is not effected.

See VAX/VMS RunTime Library manual's section on dynamic memory for full description of virtual memory zones, etc.

IDL Dynamic Memory

IDL, mostly written in C, uses dynamic memory through standard C language "malloc" and "free" calls. The C run-time library creates a separate VM zone to manage this memory, but the zone is created with sizes and characteristics more appropriate to simpler data structures and smaller amounts of data. As a result the dynamic memory used for standard IDL operations can become fragmented, wasting memory in unusably small pieces.

RSI, the developers and vendors of IDL, are not willing to address this problem with a VMS-specific solution, apparently because of concerns about portability and even the long-term viability of VMS. It is small consolation that memory fragmentation problems are even worse on some Unix systems.

One effective work-around is to create a large array, then set it to a scalar zero. This creates a large contiguous chunk of memory but leaves it free for re-use. An array size of four to ten MB is usually sufficient, it may be wise to start with four MB (a 1000 x 1000 long integer array, for example) and increase as necessary. The "showquo" routine can be used to assess the effectiveness of a given array size.

Modification of VAXC$MALLOC zone

There is fragmentation caused by the way IDL allocates memory. IDL, by default, uses the VAXC$MALLOC routine which, because the call is internal to IDL, cannot be changed. However, the zone used by the C run-time library on behalf of IDL, (and consequently by VAXC$MALLOC) has been modified in order to allocate a larger initial size and extend size which should keep the fragmentation to a minimum. Also, the characteristics/flags of the zone have been modified to include boundary tags and the extend area attribute.

The modifications include allocating memory from a C routine to create the VAXC$MALLOC zone, searching through the list of zones to get the last one created, setting up new characteristics for the zone, deleting the VAXC zone and immediately recreating it with the new characteristics. This method relies on the observed behaviour that if the characteristics of the zone do not change radically (ie. no more space being taken up by information about the zone) the zone id will remain the same. This behaviour could change with a future version of VMS. Note that this modification must be performed before there has been any use of the VAXC$MALLOC zone.

4.1.3 Related routine SHOWQUO

a.

Usage

The SHOWQUO utility displays several memory related statistics. Inside IGORE, the utility is invoked by:

```
IGR> SHOWQUO [,/FULL]
```

The default display includes initial pagefile quota, remaining pagefile quota, remaining virtual pages, and the number of free pages actually remaining in the pagefile, as well and the number of calls to LIB$GET_VM and LIB$FREE_VM, and the number of bytes (of dynamic memory) still allocated.

Adding the /FULL keyword results in the above display plus a detailed display of information for each VM zone currently in use.

See Appendix A for an annotated sample of SHOWQUO output.

4.2 IGOREIDL Structures

4.2.1 IDL Internal Structure Notes

IDL internal structure definitions can be found in the file IDL_DIR:[SOURCE]EXPORT.H. This file may be updated with each release of IDL, so code including this file should point to this particular file in the IDL distribution directory, rather than a copy. The file contains standard C language structure and macro definitions.

The IDL external, or "user", structures are a special case particularly important for IGORE. While the corresponding internal structures are similar to those used for array variables, there is an otherwise spare field used in the descriptor to point to the structure definition. Once created, this structure definition is fixed until program exit. The structure definition is created at the creation of the first record(s) of that structure. Subsequent to that, new records of that structure reference the same definition. The layout of the structure definition itself it not entirely clear, but the first field is a null-terminated string containing the structure name. One possibility is that the name is used in a table search to locate the definition proper.

4.2.2 IGORE Record Pointer Structure

The external IGORE record pointer structure, named IGR$REC_PTRS, is an IDL user structure defined at IGORE startup. It contains two fields and is defined as:

```
{ IGR$REC_PTRS, HDR: long(0), REC: long(0) }
```

The IGORE startup procedure (and IDL language script) defines this structure by the creation of one record. That record is passed to an internal IGORE routine that saves the address of the structure definition in a global variable. Once that address is saved, the record itself is deleted, but the definition is fixed, and the address of that definition is available to other internal IGORE routines.

Records of this structure are used to store Igore record pointers as IDL variables. Igore records are typically created by one routine, populated by another routine, and accessed by many other routines. The Igore records are never directly manipulated by IDL, instead, pointers to these records are returned to IDL by the Igore routine creating the records, then passed to other Igore routines to reference the records.

Requirements.

Examination of the IGR$REC_PTRS structure, as defined above, reveals that each element contains two longwords for a total of eight bytes. The IGORE system design requires the record pointer to actually consist of two pointers, including a pointer to a record header as well as one to the record proper. At the same time, the ability to conveniently reference the whole two-part pointer as a single entity was required.

To accomplish this, the original version of IGORE used the only simple eight byte data type, the double precision float variable. The float was mapped internally by IGORE as two long integers to contain the pointer fields. Using this method, "dirty zeroes" (negative zeros) and subsequent normalization led to corrupted record pointers.

To avoid problems with corrupted record pointers, the structured record was determined to be the best option and is used in all distributed versions of Igore.

Risks

RSI, the developers and vendors of IDL do not support this technique, so it is subject to modification as changes in IDL's implemention of record structures occur.

Changes in the IDL's internal implementation of structure definitions occurred in IDL Version 3.6. Prior to V3.6, the structure name was at the address specified by the "sdef" field of the variable descriptor. Starting with V3.6, the "sdef" field appears to point to another structure which in turn contains a pointer to the structure name. Note that while only the value of the "sdef" pointer is strictly required, the structure name is used extensively for consistency checking.

As a result, Version 7.4 of IGORE passes a version dependent flag to the GET_PTR_STRUC() routine which controls where that routine looks for the structure name.

While such changes in IDL's internal implementation can be reverse-engineered, the process is time consuming and the results must be carefully tested to insure the continued reliablity of IGORE.

Related routines

Several internal Igore routines have been created to access and test the IGR$REC_PTRS structures and records. These routines can all be found in the source module IGR_IDL_STRUCT_UTILS.C.

GET_PTR_STRUCT This routine expects to be called from IDL and to be passed a record of the type IGR$REC_PTRS. It locates the structure definition and saves the address of the definition in a global variable. As of IGORE Version 7.4, it accepts an optional second argument, a flag which should be set to 1 if the IDL version is 3.6 or later, otherwise set to zero.

```
void get_ptr_struct (int argc, VPTR *argv)
```

The IDL call name for this procedure is GET_PTR_STRC. It is normally by IGORE$SYSTARTUP with the correct arguments for the running IDL version.

FORCE_PTR_REC This routine expects to be passed an IDL VPTR (variable descriptor) to an array of pointers. It modifies the descriptor to an acceptable VPTR for an IDL structured record. The data is not modified in any way.

Routines which create Igore record pointers must call this routine to convert the output IDL variable to a structured record. This routine must be called after the var_copy() routine, and should only be used inside a subroutine. Record pointers should not be returned as function values.

```
void force_ptr_rec (VPTR temp)
```

Example:

```
long example (int argc, VPTR *argv) {
#include "idl_dir:[source]export.h"
/*
 * The IDL call dispatcher loads argc with the argument count
 * and argv with a pointer to an array of VPTR'S. Each VPTR
 * is the address of a VARIABLE structure, which is a descriptor
 * for an IDL command line argument.
 */
long array_dims[MAX_ARRAY_DIMS], out_array, tmp_array, temp;
/*
 * Next we create a temporary IDL array. make_temp_array()
 * returns the address of the datablock, the temp argument
 * receives the address of the descriptor for the temporary
 * array. The created array is a double precision float
 * necessary to contain an 8-byte record pointer. It is
 * not possible to specify TYP_STRUCT for this argument.
 */
array_dims[0] = dims;
out_array = make_temp_array(TYP_DOUBLE, 1, array_dims,
BASICARR_INIT_ZERO, &temp);

/*
 *  here would be a call to a routine that populates the
 * array with record pointers.
 */
/* example_create_pointer_array (&dims, out_array); */

/*
 * With our populated temporary array ready, use var_copy()
 * to create a permanent IDL array.
```

```
 */
var_copy (temp, argv[1]; /* where the pointers are returned in
                            the second argument to this routine*/

/*
 * Now that a permanent IDL variable has been created, we can
 * convert it to a structured record, using the force_ptr_rec()
 * routine.
 */
force_ptr_rec (argv[1]);
}
/* end of example routine*/
```

CHK_STRUCT This function expects to be passed a private internal Igore record of the the type CTRL_REC. It tests the IDL record pointed to by the idl_var_dptr sub-field, which should indicate a valid IDL record of type IGR$REC_PTRS, and returns a true (1) or false (0) value accordingly. The function is primarily for the convenience of low-level AFE utility routines.

```
long chk_struct (CTRL_REC *ctrl)
```

STRUCT_OK This function expects to be passed a VPTR (variable descriptor). It tests that the descriptor describes a valid record of type IGR$REC_PTRS, and returns a true (1) or false (0) value.

This is the general purpose entry point for keyword front ends to test the data type of input pointers arrays.

```
char struct_ok (VPTR temp)
```

4.2.3 Record Pointers as Function Returns

The Problem

The problem surfaces any time user code returns an IDL structured record as a function value. Comparison with the subprogram case is useful in the explanation.

In the case of a subprogram, when the code allocates space for a temporary variable, the code must specify a data type for the variable. The IDL routines to allocate variable space do not allow us to specify TYP_STRUC to allocate space directly for a structure, so we allocate (in the case of the record pointers) real*8 arrays. To return this as a argument, we use the IDL copy_var() routine to copy the temporary to the argument, then patch up the descriptor to the ARGUMENT, so that it looks like an IDL sturctured record, before returning.

We do the descriptor patching after calling copy_var, so that the temporary descriptor is returned with the same data type as it had at allocation time. If a temporary is allocated with one data type, but returned with another, the original allocation is still considered to be outstanding, and a warning is issued on return to IDL.

The approach above works fine with subprogram routines. In the case of function return values, there is no argument with a descriptor we can patch. If we modify the temporary descriptor, IDL considers a temporary to be still checked out when we return and issues an error message. We chose to return a real*8 pointer instead, which is still vulnerable to floating point fixups.

Potential Solutions

There are really only two alternatives. A variation of the first alternative is actually used starting with IGORE Version 7.0.

First alternative Modify the code in question (GETFLD,etc.) to work as subprograms rather than as functions. We can and do return structured record pointers as arguments. To preserve the current calling sequence, the subprograms could be front-ended with IDL functions. For example,

```
Igore> x = getfld ('@hdr',sdr(0))

... would invoke an IDL function

function getfld (field, record)
; which calls a application subprogram
;
getfld_p, field, record, return_value
return, return_value
end
```

Second alternative Convince RSI to document the IDL entry points needed to create and manipulate records and structures. If we're going to ask RSI for some enhancements, and offer to pay for same, maybe this could be included.

4.3 IGORE Startup

4.3.1 Overview

IGORE startup processing is important in itself, and useful in understanding the overall organization of a complete running IGORE system. The sequence of events can be summarized as follows:

* The system is started with the symbol, "IGORE". The symbol equates to the foreign command:

```
$IGORE$IMAGE IGORE$SYSTARTUP
```

The single argument to the command is a logical name pointing to an IDL procedure that performs some of the startup processing.

* The command invokes the image pointed to by the logical IGORE$IMAGE, invoking the IGORE main routine.

* The IGORE main routine, which is linked with the IDL shareable image, makes several calls to the IDL procedure "add_system_routine()". These calls define the entry points of the IGORE "system" routines. These entry points can later be used by the IDL call dispatcher. The source code for the IGORE main routine can be found in IGORE_SYSTEM_DEFINITIONS.C.

* The IGORE main routine then calls the IDL internal entry point ur_main(), the callable interface to the IDL main routine. The argument to IGORE main is passed to ur_main().

* IDL startup invokes the user startup procedure identified by the logical IDL_STARTUP, if it exists.

* IDL's standard command processing invokes the procedure named in the argument passed, normally the logical IGORE$SYSTARTUP.

* The procedure identified by the logical IGORE$SYSTARTUP, which normally translates to IGR$ROOT:IGORE_SYSTEM_STARTUP.PRO builds the !PATH variable.

* The IGORE$SYSTARTUP procedure creates a record of the structure IGR$REC_PTR, calls the IGORE system routine get_ptr_strc() to save the address of the structure definition, then deletes the record.

* The IGORE$SYSTARTUP procedure calls the procedure AUXDEF_V7 which defines a structure with a field for the type code for each auxiliary data type mnemonic.

* The IGORE$SYSTARTUP procedure calls the procedure MSG_CODE to define the !ERRV system variable for IDL error codes.

* The IGORE$SYSTARTUP procedure calls the IGORE system routine INIT_IGORE. This routine initializes the IGORE dynamic memory tables and structures.

* The IGORE$SYSTARTUP procedure calls the procedure IGORE_SYSLINK.

* The procedure IGORE_SYSLINK translates the list logical IGORE$SYSTEM_APPS. For each translation, IGORE_SYSLINK tries to invoke a procedure named as IGORE_'translation'.PRO. The value of each translation identifies one of the IGORE "applications".

* Each procedure invoked by the last step executes a series of IDL linkimage statements defining the entry points in the relevant application shareble image.

* The IGORE$SYSTARTUP procedure calls the procedure SDR$INIT_SDRH, which initilizes structures used by the AUX access routines.

* The IGORE$SYSTARTUP procedure calls the procedure IGORE_LINKIMAGE, which is merely a placeholder, by default. The user has the option of placing a procedure of the same name such that it occurs in the IDL_PATH before the directory IGR:[IDL], allowing customized applications to start automatically.

* The IGORE$SYSTARTUP procedure sets the prompt, cleans up work variables, and exits. Control passes to the normal IDL command line processor. IGORE is started.

4.3.2 IDL Path Processing

Because much of the IGORE functionality resides in IDL procedures, the IDL system variable !PATH must include either the directory or a text library containing the procedures.

Several distinct issues have determined the rules used in modifying the IDL !PATH variable so that it includes the elements required by IGORE along with any user specified elements in a compatible, reasonably friendly way.

IDL limits the number of text libraries that can be present in !PATH before version 3.6.1. So, historically, flat directories have been the best approach to preserving scarce IDL text library slots while at the same time keeping the IDL !PATH variable to a meaningful size.

IGORE_SYSTEM_STARTUP.PRO will build an IDL !PATH system variable as follows:

- The user's IDL_PATH logical is not touched by INIT_IGORE.COM. This means that the user can now modify the IDL_PATH logical at anytime, in ALMOST any way, without endangering Igore functionality.
- Any elements in IDL_PATH that are duplications of elements required by Igore are removed when building !PATH. Only the instance in the Igore-required list is included. See below for the final order of !PATH.
- Elements of IDL_PATH that come from IDL_DIR are still added to the end of the !PATH variable. To force them to the beginning, they must be entered in the IGORE$PATH which is included in the !PATH variable the time-honoured old-fashioned historic CPLOD (comma position list order determination) way.

The final order of the various pieces in !PATH looks like this:

```
stuff from IGORE$PATH   (if followed by comma)
    +stuff from IDL_PATH   (if not from IDL_DIR)
    +stuff required by IGORE (igr:[idl],stat,lib,array,color,data_io, -
      environ,fonts,graphics,math,obsolete,prog,widgets)
    +stuff from IDL_PATH   (if from IDL_DIR)
    +stuff from IGORE$PATH   (if preceded by comma)
```

Usage guidelines are simple. One, if you want something in the IDL_DIR tree to take precedence over something in IGR:[IDL], include the item from IDL_DIR in IGORE$PATH. Two, if you get message at Igore startup about "too many procedure libraries", you'll probably want to remove some libraries from your path, as some essential functionality may be missing.

The "working copy" of each IDL procedure is in a flat directory for configured and supported procedures. The "source" for these procedures are in subdirectories by category. The following directories are used for the IGORE IDL procedures.

```
IGR:[IDL]  ! Working copies of all supported/configured IDL procedures.
IGR:[IDL.SRC]        ! Source only in following subdirectories

IGR:[IDL.SRC.ANAL]              ! Contains generic analysis routines
IGR:[IDL.SRC.CMDSTATE]          ! Contains routines for command state
IGR:[IDL.SRC.DBMS]              ! Contains database management system routines
IGR:[IDL.SRC.DTW_EFF]           ! Contains DTW Efficiency routines
IGR:[IDL.SRC.ECAL]              ! Contains Energy Calculations routines
IGR:[IDL.SRC.EFOLD]             ! Contains Epoch Folding routines
IGR:[IDL.SRC.FDB]               ! Contains FDB routines
IGR:[IDL.SRC.FFOLD]             ! Contains Forward Folding routines
IGR:[IDL.SRC.FFOLDM             ! Contains Forward Folding Model routines
IGR:[IDL.SRC.FIT]               ! Contains FIT routines
IGR:[IDL.SRC.MISC]              ! Contains miscellaneous routines
IGR:[IDL.SRC.OGRE]              ! Contains OGRE (OSSE Graphical
                                !   Reduction Environment) routines
IGR:[IDL.SRC.OSSE]              ! Contains OSSE specific routines
                                !   (non-distribution items)
IGR:[IDL.SRC.PDFTSF]            ! Contains PDF, TSF & PDR routines
IGR:[IDL.SRC.PLAN]              ! Contains PLAN routines
IGR:[IDL.SRC.PLOT]              ! Contains plotting routines
IGR:[IDL.SRC.RSI]               ! Contains routines changed from RSI
                                !   delivered (not CMed)
IGR:[IDL.SRC.SDB]               ! Contains SDB routines
IGR:[IDL.SRC.SYS]               ! Contains System routines
IGR:[IDL.SRC.TIME]              ! Contains Time routines
```

4.3.3 Application startup

The startup applications are defined by the logical IGORE$SYSTEM_APPS which is currently the three applications: SDB, PDFTSF and IRM. The applications SDB, PDFTSF (two of the startup defaults) as well as FDB, MISC and INTGRAPH follow the above startup procedures. All of the sharable executables are located in the directory IGR$EXE, except the IRM (Instrument Response Matrix) application, which is handled differently. The directory IGR$IRM contains sharables and IDL routines which define the IRM$APPS.

4.3.4 IGORE-specific IDL System Variables

!AUX

The !AUX system variable is created by the procedure AUXDEF_V7.PRO. !AUX is an IDL structure with one field for each IGORE auxiliary data type. The value of each field is set to the corresponding bit number in the SDB header's AUXDEF field. It is intended solely to provide a consistent mnemonic for specifying auxiliary data types to routines.

See the SDB Header Version 7 document for details on IGORE auxiliary data.

!ERRV

The !ERRV system variable is created by the procedure MSG_CODE.PRO. !ERRV is an IDL structure defining fields for several IDL I/O error codes. It is intended to provide consistent mnemonics for I/O errors across IDL versions.

4.3.5 Internal initialization

Done by INITIALIZE_VM_TABLES to create zones and initialize first segment. Also, see section IIIA2c on modification of VAXC$MALLOC VM zone.

4.3.6 Record pointer structure initialization

Done by routine GET_PTR_STRUCT. See above section on record pointer structure.

4.4 Miscellaneous Routines

4.4.1 CTRLC_ABORT

The routine CTRLC_ABORT() is an IGORE internal entry to the condition handler facility. When called, it checks an internal flag indicating whether or not a CONTROL-C has been entered by the user, and if so gracefully aborts the program. Because of unwanted interactions with the RTL I/O routines, the normal out-of-band AST approach does not work properly, so the CTRLC_ABORT routine was created to allow high level code to be aborted by CONTROL-C. It should be placed inside of do-loops and other repetitive structures.

4.4.2 WRTALL

The routine WRTALL() provides a messaging flexiblity for C-language sourced portions of IGORE and its applications. It writes to both the user terminal and to the IDL journal file (if open).

4.4.3 Preprocessors

The IGORE preprocessors are standalone programs used for converting source code for IGORE applications, IGORE structures, and linker option files.

- BUILDSHR is used to create a sharable image for user programs to be linked with Igore. Builds transfer vectors.
- MAKE_IGORE_UTILITIES_VCTR builds transfer vectors for Igore system routines. Uses IGORE_ENTRY_POINTS.TXT as input to create and compile a macro file of corresponding transfer vectors. Output file IGORE_UTILITIES_VCTR is referenced by IGORE.OPT for linking.
- MAKE_SDR_UTILITIES_VCTR builds transfer vectors for system routines referenced by the SDR routines. Uses SDR_ENTRY_POINTS.TXT as input to create and compile a macro file of corresponding transfer vectors. Output file SDR_UTILTIES_VCTR is referenced by ???
- AFE_PROCESS /command "GEN_VCTR" - used to generate transfer vector macros as well as link options for common block psects.
- NOSHR_PSECTS /command "CHECK_COMMON" - used to identify common blocks in FORTRAN source files and build appropriate options file entries.
- PREPRO /command "AFEGEN" - used to generate an AFE (Application Front End) to be linked via linkimage command to the Igore application.

4.4.4 IGR_GET_KW

The routine IGR_GET_KW() is used to simplify and speed up handling keywords from a C interface routine for an AFE. This routine allows caller to accept a large list of keywords, including a keyword for each AUX data type, or any subset of that list. It populates: (1) the AUX1 and AUX2 bit fields, (2) the pargs array (positional arguments) used by IDL'S GET_KW_PARAMS routine and (3) a user passed record of the structure GET_KW_LIST. By defining a bit mask for each routine, the keywords available to a particular routine can be well defined. IGR_GET_KW will search through the list of possible keywords and set the appropriate bits in the user's kw record (type GET_KW_LIST). This is mainly used by the read, write, get and free routines, but the preface fully documents how to go about adding new keywords and using it from other routines.

4.4.5 File and LUN managment routines

IGORE provides a set of file and LUN management routines. The source code for these routines is in FILE_LUN_MGMNT.FOR, and FILE_LUN_MGMNT_KW.C. These routines manage the set of FORTRAN logical units as well currently open files. Command line interfaces are available although the primary usage for these routines is in the SDB applications which support "transparent" file access.

Transparent file access in IGORE is considered to that access which is not the result of explicit file open, file read, file write sequences. For example SDR$GET creates and populates auxiliary data records in an SDR by accessing the SDR's parent file. The field SDB.FILE is passed to the file / LUN management routines. The management table is searched for the file name, and if the file is already open for transparent access, the lun from the table is used by the SDB I/O routines. If the file is not already open for transparent access, the file is opened, entered into the management table, and the lun is passed to the SDB I/O routines.

There are several disadvantages to this scheme. Because the higher level application passes an array of SDRs one element at a time, the table search takes place once for each array element. Another disadvantage is that, at least as currently implemented, routines that do explicit file access such as SDB$READ, SDB$WRITE, etc., share the same low level code. As a result, we get the following scenario:

```
Igore> SDB$OPEN,LUN,FILE,/GET_LUN  ; open file on lun allocated
                                   ; from management table
Igore> NTAG = SDR$NTAG(LUN)        ; get record count
Igore> SDR$CREATE, SDR, NTAG       ; create record array
Igore> SDB$READ,LUN,SDR,LONARR(NTAG) ; checks that file is open
                         ; then transparently accesses each
                         ; record via file/LUN and SDB I/O
                         ; routines.
```

This scenario can create problems for routines that must open identically named files in different directories, for instance, and forces higher level IDL procedures to incorporate cumbersome workarounds.

More reliable approaches are still being sought as of 06/20/95.

4.5 Miscellaneous Notes

4.5.1 Local IGORE Records

IDL, like most programming languages, distinguishes between global variables and local variables. The only global variables recognized by IDL are variables contained in "common" blocks, and the so-called "system variables", which can be identified by the initial exclamation point (!) in the variable name. All other IDL variables are either local to a procedure or local to the main program, or else are arguments of a procedure.

Inside a procedure, the arguments are known as "formal" or "dummy" arguments. Except for variables in "common" blocks, all other variables inside a procedure are local to that procedure, meaning that these variables are temporary and automatically deleted on exit from procedure.

IGORE records do not fit the local/global model very well. The record pointers can be local IDL variables, or IDL argument variables, while the IGORE records themselves are not known to IDL at all, and cannot be automatically deleted. This has some impact on programming IDL procedures which create IGORE records. It is possible to generate "orphan" records inside a procedure when IGORE records are created with the record pointers in a local variable. When the procedure exits, the pointer variable is deleted but the IGORE records remain unless explicitly deleted. (Note that although it is possible to put IGORE record pointers into "common" blocks, it is inconvenient and not generally useful.)

These "orphan" records are troublesome, and should be avoided. If the records are created with the /UNIQUE qualifier available to many routines, significant amounts of memory may be wasted. If an attempt is made to create an IGORE record of the same name by recalling the same procedure or from some other procedure, the following error will result:

```
SDR$CREAT%IGR-E-IMDMSG, Record name already exists in Create_Rec_Array
```

Good programming practice allows two options for creating IGORE records inside IDL procedures. One option is to ensure that the IGORE record pointers are created in an argument to the procedure, and are therefore preserved when the procedure exits. (An alternate form of this option would be to ensure that the record name is returned as an argument.)

Using option two, if the application requires it, a pseudo-local IGORE record can be used safely. To do so, the procedure should create the record with the /UNIQUE qualifier to eliminate naming conflicts with existing records, and it MUST delete the record before exiting.

By sticking to one of the two recommended techniques, IDL procedures can be written which avoid the problems associated with "orphan" records. When problems are caused by older IDL procedures, the procedures can usually be adapted by using one of these two techniques.

4.5.2 PROCSECTCNT Sysgen Parameter

The "section table full" message is a can of worms, but here is a (over)simple description. The process section table (PST) and the working set list (WSL) share a memory region. The PST starts out at the size required to contain PROCSECTCNT entries, and grows toward the WSL. The WSL starts as being sized by WSDEF and grows toward the PST, limited by WSMAX. Once the WSL has reached size WSMAX, the PST cannot grow beyond PROCSECTCNT, (well, with some complicated exceptions that don't apply here) and no further process sections can be created.

```
addr  ---------------------------
incr  |
  |   |    WSL
  V   |         growth |
      |                v
      |
      |
      |
      |
      |                           WSMAX
      ----------------------------
      |                           PROCSECTCNT
      |    PST                 ^
      |         growth |
      |
      |-----------------------------
```

If a user has a large WSQUOTA on a system that has a small WSMAX (Ulmer on any NU node but OSSENU), the WSL list will grow to WSMAX when running a large image as long as there is sufficient free memory. Once it has grown, the WSL can only shrink if entries can be removed from the high address end of the list, which is particulary unlikely when WSMAX is small.

So a scenario would go like this. A user runs a large image that uses few PST entries but WSMAX WSL entries, filling the WSL. Image rundown is unable to reduce the WSL because of the small WSMAX. So far so good. Now the user runs Igore. He can only get PROCSECTCNT number of PST entries, not enough. Crash. If the user runs Igore without previous running this other large image (IDL's VM usage counts, too), the PST entries are taken before the WSL has grown to the limit, and things work, although probably slowly.

The current IGRTST code uses 76 process sections, so 64 PST entries is still not sufficient. Try 96, so that there is room for additional merged images (via linkimage), if necessary.

4.5.3 Physical vs. Virtual Memory

This note is in response to a question that came up at the recent DAWG meeting. The question was whether or not we are getting "heap" size scaled to system physical memory in IDL.

The simple answer is "no", but may be misleading. On VMS systems, the "heap", or dynamic memory, is not directly related to physical memory capacity.

Instead, dynamic memory is limited by the smaller of the system parameter VIRTUALPAGECNT or the user quota PGFLQUO. The system VIRTUALPAGECNT sets an upper limit on the virtual size of any process on a given system. PGFLQUO sets an upper limit for any given user on the system.

These parameters are typically set according to the memory requirements of the applications in use, and are not related to hardware in any way, although it is critical that a system's pagefile(s) be sufficiently large to contain all of the pages that are not in physical memory. For example, on a system with 64 MB of physical memory, two processes might have an address space of 80 MB each. Obviously, 160 MB cannot be contained in physical memory simultaneously, so some 96 MB will be in the pagefile. Allowing space for other smaller processes and the pageable parts of the operating system, a reasonable value might be 200 MB or 400 K pages. If sufficient space is not available in the pagefile, the system will hang!

Igore has built in checks to ensure that it will not over-use the pagefile and hang a system, but IDL does no such checking.

Systems with more physical memory can may take advantage of it by increasing the amount available to a process. Again there are both system-wide and user-specific parameters that control the size of the physical memory, or working set, for processes on a system. The system parameter WSMAX sets a system-wide limit and the user quota WSEXTENT sets a user-specific limit. The smaller of the two values is the upper limit for each user. Increasing the size of either or both of these will NOT effect the amount of virtual memory the user is allowed, but can effect the speed of the memory access, since phyical memory is always much faster than the pagefile on disk. It should be noted that VMS retains a cache in memory of pages removed from process working sets, but not yet moved to the pagefile. Access to this memory is only slight slower than to pages in the working set.

The adjustment of working set sizes should be performed very carefully. Gross adjustments have side effects that must be considered and may require other changes. At a minimum, the VMS Guide to Performance Tuning should be consulted.

In summary, the amount of virtual memory available is unrelated to physical capacity, although it does have implications for pagefile sizing. Virtual memory parameters VIRTUALPAGECNT and PGFLQUO should therefore be set according to application requirements.

To take advantage of greater amounts of physical memory, the working set parameters WSMAX and WSEXTENT should be cautiously adjusted.

Appendix A

Annotated sample of output from SHOWQUO utility

This sample output from SHOWQUO is annotated with item numbers used in the explanatory text, these item numbers are appear as <n>.

```
Original pagefile quota:        131071              <1>
Remaining pagefile quota:        84438              <2>
Remaining virtual pages:        248001              <3>
Free page file pages:           266672              <4>
 796 calls to LIB$GET_VM, 368 calls to LIB$FREE_VM, 145392 bytes still
allocated
Zone Id = 00094468,  Zone name = "DEFAULT_ZONE"     <5>

   Algorithm = LIB$K_VM_FIRST_FIT               <6>

   Flags = 00000020
           LIB$M_VM_EXTEND_AREA                 <7>

   Initial size =     124 pages     Current size = 124 pages in 1 area <8>
   Extend size  =     128 pages     Page limit   = None                <9>

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   1168 bytes have been freed and not yet reallocated      <10>

   144 bytes are used for zone and area control blocks, or 0.2% overhead <11>

   Area Summary:

         First     Last       Pages    Bytes not yet
        address   address    assigned    allocated
```

```
       --------  --------  ----------  -------------
       00439E00  004495FF         124          36288     <12>

    Scanning Free List for Area at 00439E00
      Number of blocks = 10, Min blocksize = 8, Max blocksize = 576 <13>


Zone Id = 00094508,  Zone name = "STRING_ZONE"

    Algorithm = LIB$K_VM_QUICK_FIT  with 17 Lookaside Lists ranging from
                a minimum blocksize of 16, to a maximum blocksize of 144

    Flags = 00000021
            LIB$M_VM_BOUNDARY_TAGS
            LIB$M_VM_EXTEND_AREA

    Initial size =       4 pages     Current size = 4 pages in 1 area
    Extend size  =      32 pages     Page limit   = None

    Requests are rounded up to a multiple of 8 bytes,
    naturally aligned on 8 byte boundaries

    1728 bytes have been freed and not yet reallocated

    280 bytes are used for zone and area control blocks, or 12.0% overhead

    Quick Fit Lookaside List Summary:

            List        Block   Number of
           number        size     blocks
           ------  ----------  ----------
             1          16           2
             3          32           1
             4          40           1
             5          48           1
             6          56           1
             7          64           1
             8          72           1

    Area Summary:

          First     Last       Pages    Bytes not yet
         address   address    assigned   allocated
        --------  --------  ----------  -------------
        00449600  00449DFF           4            280

    Scanning Lookaside Lists in Zone Control Block
    Scanning Free List for Area at 00449600
      Number of blocks = 3, Min blocksize = 272, Max blocksize = 608


Zone Id = 00449E00,  Zone name = ""     <14>
```

```
   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 0000002B
           LIB$M_VM_BOUNDARY_TAGS
           LIB$M_VM_GET_FILL0
           LIB$M_VM_FREE_FILL0
           LIB$M_VM_EXTEND_AREA

   Initial size =    1024 pages     Current size = 1024 pages in 1 area
   Extend size  =    1024 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   136320 bytes have been freed and not yet reallocated

   144 bytes are used for zone and area control blocks, or 0.0% overhead

   Area Summary:

         First     Last       Pages    Bytes not yet
        address   address   assigned    allocated
       --------  --------  ----------  -------------
       00459600  004D95FF        1024         288864

   Scanning Free List for Area at 00459600
     Number of blocks = 19, Min blocksize = 16, Max blocksize = 56424


Zone Id = 00449E50,  Zone name = "VMT$REC "    <15>

   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 00000003
           LIB$M_VM_BOUNDARY_TAGS   <16>
           LIB$M_VM_GET_FILL0

   Initial size =    2048 pages     Current size = 2048 pages in 1 area  <17>
   Extend size  =     512 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   0 bytes have been freed and not yet reallocated

   160 bytes are used for zone and area control blocks, or 0.0% overhead

   Area Summary:

         First     Last       Pages    Bytes not yet
        address   address   assigned    allocated
```

```
       --------  --------  ----------  -------------
       006C8000  007C7FFF        2048        1048576

   Scanning Free List for Area at 006C8000

Zone Id = 00449EB0,  Zone name = "VMT$STD "    <18>

   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 00000003
           LIB$M_VM_BOUNDARY_TAGS
           LIB$M_VM_GET_FILL0

   Initial size =      64 pages     Current size = 64 pages in 1 area  <19>
   Extend size  =      16 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   0 bytes have been freed and not yet reallocated

   160 bytes are used for zone and area control blocks, or 0.4% overhead

   Area Summary:

         First     Last       Pages    Bytes not yet
        address   address   assigned    allocated
       --------  --------  ----------  -------------
       0044A600  004525FF          64          16392

   Scanning Free List for Area at 0044A600

Zone Id = 00449F10,  Zone name = "VMT$IND "   <20>

   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 00000003
           LIB$M_VM_BOUNDARY_TAGS
           LIB$M_VM_GET_FILL0

   Initial size =    4096 pages     Current size = 4096 pages in 1 area  <21>
   Extend size  =    4096 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   0 bytes have been freed and not yet reallocated

   160 bytes are used for zone and area control blocks, or 0.0% overhead

   Area Summary:
```

```
        First     Last       Pages    Bytes not yet
       address   address   assigned    allocated
      --------  --------  ----------  -------------
      007C8000  009C7FFF        4096        2097152

   Scanning Free List for Area at 007C8000

Zone Id = 00449F70,  Zone name = "VMT$TMP "   <22>

   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 00000003
           LIB$M_VM_BOUNDARY_TAGS
           LIB$M_VM_GET_FILL0

   Initial size =     512 pages     Current size = 512 pages in 1 area
   Extend size  =     128 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   296 bytes have been freed and not yet reallocated

   160 bytes are used for zone and area control blocks, or 0.0% overhead

   Area Summary:

        First     Last       Pages    Bytes not yet
       address   address   assigned    allocated
      --------  --------  ----------  -------------
      009C8000  00A07FFF         512         261848

   Scanning Free List for Area at 009C8000
     Number of blocks = 1, Min blocksize = 296, Max blocksize = 296


Zone Id = 00449FD0,  Zone name = ""   <23>

   Algorithm = LIB$K_VM_FIRST_FIT

   Flags = 00000021
           LIB$M_VM_BOUNDARY_TAGS
           LIB$M_VM_EXTEND_AREA

   Initial size =      16 pages     Current size = 16 pages in 1 area
   Extend size  =      16 pages     Page limit   = None

   Requests are rounded up to a multiple of 8 bytes,
   naturally aligned on 8 byte boundaries

   0 bytes have been freed and not yet reallocated
```

```
   144 bytes are used for zone and area control blocks, or 1.7% overhead

   Area Summary:

         First     Last       Pages    Bytes not yet
        address   address   assigned    allocated
        --------  --------  ----------  -------------
        00452600  004545FF          16           4888

Scanning Free List for Area at 00452600
```

1 Original pagefile quota - equivalent to the UAF parameter PGFLQUO. May be rounded up slightly to a value VMS likes.

2 Remaining pagefile quota - equivalent to original quota minus number of virtual pages currently mapped by the process.

3 Remaining virtual pages - equivalent to SYSGEN parameter VIRTUALPAGECNT minus number of virtual pages currently mapped by the process.

4 Free page file pages - actually number of free pages remaining in pagefile(s).

5 The total count of calls to the RTL LIB$GET_VM and LIB$FREE_VM, along with the count of bytes still allocated. Note that the count reflects all dynamic memory managed by the RTL routines, but does not include dynamic memory that is created in other ways.

5 The zone id uniquely identifies the VM zone. It can also be used to locate the VM zone control block in memory. The zone name is for informational purposes only. As a rule, the zone named "DEFAULT" is the zone used by calls to LIB$GET_VM and LIB$FREE_VM which do not include a zone-id parameter.

6 The algorithm parameter specifies which of several algorithms is to be used for locate, free, and organization of areas within the zone. See the VMS RTL documentation on Dynamic Memory facilities for details.

7 The flags parameter indicates which of various options are in use. LIB$M_VM_EXTEND_AREA, for example, specifies that, if possible, an existing VM area is to be extended to satisfy an allocation request. Otherwise a new area is always created if no existing area has sufficient free space.

8,9 The initial and extend size parameters are also displayed. Note the page limit parameter, which can be used to set an upper limit on the size of a zone.

10 Count of bytes that have been allocated then freed, but not yet reused.

11 Overhead statistics. Very little overhead is required by the RTL dynamic memory facilities.

12 The area summary lists information about each area in the zone. Note that the size of the area in pages, as well as the number of bytes not yet allocated are displayed. The latter is NOT identical to the number free of bytes in the area.

13 Free list statistics are also displayed for each area in the zone. Note the "Max block-size" value. In this sample, the value is 1184 bytes, representing the largest chunk (or node) of free memory in the area. This does not include the memory that has not yet been allocated.

14 This particular un-named zone is the zone used by the C run-time library on behalf of IDL. It can be expected to appear immediately after the "STRING_ZONE" in the list of zones. See section IIIA2c on modification of VAXC$MALLOC VM zone.

15 Zone created at IGORE initialization to be used for IGORE "direct records".

16 All zones created by IGORE at initialization enable the boundary tag feature. This feature has a minor performance benefit, as well as allowing free operations to specify an address without a size. A beneficial side effect is that the RTL VM facilities are justifiably fussy about the correctness of the boundary tags. Corrupted boundary tags detected by these facilities can often served as an "early warning" system for memory corruption caused by IGORE bugs.

17 The initial size of the "direct record" zone is one MB, and it extends in units of 256KB. Observed usage has shown this to be sufficient.

18 Zone used for IGORE structure descriptors.

19 Small initial and extend sizes. This zone rarely, if ever grows after initialization.

20 Zone used for "indirect records".

21 Note the large initial and extend sizes, two MB each. A typical IGORE session requires 20 to 40 MB for "indirect records", so 20 or more areas will be created. Observed usage shows this particular size is large enough to minimize or eliminate fragmentation, while not greatly increasing virtual size unnecessarily.

22 Zone created for temporary dynamic memory. In the IGORE context, temporary dynamic memory can be defined as memory allocated and then freed by IGORE system code or IGORE application code before control returns to the IDL command line.

23 This zone appears to be used by debug.

Appendix B

IGORE Structure Definition Source Files

B.1 VM_TABLES.ICL

```
c......  IGORE VM Management Table ......
c  Jim Pendleton, IGORE, GRO/OSSE NU
c  6/5/89
c
c  First include IGORE status codes and VM table structures
c
   Include 'VM_Structures.ICL/NoList'
c
c  Implement the VM management table structure.
c  Communication is through a (gulp!) common block.
c
   Record /VM_MGNT/ IGR_VM_Table(vmt$ntables)
   Common /VM_MGNT_Block/ IGR_VM_Table
c
c...... End IGORE VM Management Table include file ......
```

B.2 VM_STRUCTURES.ICL

```
c......  IGORE VM Management Table structures ......
c  Jim Pendleton, IGORE, GRO/OSSE NU
c  6/5/89
c
   Include 'VM_Parameters.ICL/NoList'
c
```

```
c  This data structure defines the entries in the IGR_VM_TABLE.
c
   Structure /VM_MGNT/
      Integer*2 Max_Entries      !diff. for diff. table types
      Integer*2 Slot_Size        !bytes per segment slot
      Integer*4 zone_id          !vm zone identification
      Integer*4 init_size        !initial size of zone
      Integer*4 extend_size      !extend size of zone
      Integer*2 Nxt_Indx(vmt$max_seg)  !next avail. slot in seg(i)
      Integer*2 Cur_Ent(vmt$max_seg)   !no. active ents in seg(i)
      Integer*4 Seg_Add(vmt$max_seg)   !address of seg(i)
   End Structure

   Structure /VM_Slot_Structure/
      Union
         Map
            Integer*4 Address/0/    !add returned Lib$Get_VM
            Integer*4 Size/0/       !nbytes input Lib$Get_VM
            Integer*2 Stat/0/       !takes up space
         End Map
         Map
            Character*8 Seg_Desc    ! add & size as one
         End Map
      End Union
      Character*15 VM_ID/' '/ !char ID for alloc {types R and S only}
   End Structure
c
c......  End IGORE VM Management Table structures include file......
```

B.3 VM_PARAMETERS.ICL

```
c......  IGORE VM Management Table parameters
c  Jim Pendleton, IGORE, GRO/OSSE NU
c  8/12/90
c
   include 'igr$status(igr$status)'
c
c  These parameters define the VM management table itself.
c
   Byte vmt$ntables
   Integer*2 vmt$max_seg
   Integer*4 vmt$size
   Byte vmt$rec, vmt$std, vmt$ind, vmt$tmp
   Integer*2 vmt$maxrec, vmt$maxstd, vmt$maxind, vmt$maxtmp

   Parameter (vmt$ntables = 4)   ! number of entries in vm table
   Parameter (vmt$max_seg = 20)  ! max segments per table
   Parameter (vmt$size = vmt$ntables*(vmt$max_seg*8 + 12))
```

```
                                  ! byte alloc. of vm table
      Parameter (vmt$rec = 1)       ! mnemonic for record segments
      Parameter (vmt$std = 2)       ! mnemonic for struc desc segments
      Parameter (vmt$ind = 3)       ! mnemonic for indirect rec vm segments
      Parameter (vmt$tmp = 4)       ! mnemonic for temporary segments
      Parameter (vmt$maxrec = 20)   ! records : max slots per segment
      Parameter (vmt$maxstd = 10)   ! struc descr : " "
      Parameter (vmt$maxind = 50*vmt$maxrec) ! indirect rec : " "
      Parameter (vmt$maxtmp = 20)   ! temp : " "
c
c  set vm zone characteristics
c
      parameter (vmt$rec_init = 2048) ! initial size of vm zone (pages)
      parameter (vmt$std_init =   64) !  "    "       "  "     "
      parameter (vmt$ind_init = 4096) !  "       "       "  "     "
      parameter (vmt$tmp_init =  512) !  "       "       "  "     "
      parameter (vmt$rec_extnd =  512) ! extend size of vm zone (pages)
      parameter (vmt$std_extnd =   16) !  "      "       "  "     "
      parameter (vmt$ind_extnd = 4096) !  "      "       "  "     "
      parameter (vmt$tmp_extnd =  128) !  "      "       "  "     "
c
c  These parameters define table segments.
c
      Integer*4 vmt$maxslotsize, vmt$vmidsize, vmt$vmidoff

      Parameter (vmt$maxslotsize = 25)! byte alloc for table slots
      Parameter (vmt$vmidsize = 15) ! size of vm_id field
      Parameter (vmt$vmidoff = 10)  ! byte offset of vm_id field
c
c
c...... End IGORE VM Management Table parameters include file ......
```

B.4 RECORD_STRUCTURES.ICL

```
c...... IGORE Dynamic Record structures
c  Jim Pendleton,  IGORE GRO/OSSE NU
c  9/25/91
c
      Structure /Structure_Descriptor_Header/
         Integer*2 Segment/0/
         Integer*2 Index/0/
         Character*15 StructureName/' '/
         Logical*1 Indirect/0/
         Byte NestingDepth/0/
         Integer*4 NFields/0/
         Integer*4 RecordLength/0/
         Integer*4 SDAddress/0/
      End Structure
```

```
Structure /Field_Descriptor/
   Union
      Map
         Byte Type/0/
         Integer*2 Size/0/
         Union
            Map
               Integer*4 Offset/0/
            End Map
            Map
               Integer*2 Segment
               Integer*2 Index
            End Map
         End Union
         Byte NestingMode/0/
         Byte NDims/0/
         Union
            Map
               Integer*4 Dimensions(8)
            End Map
            Map
               Character*15 StructureName/' '/
               Integer*4 SDAddress/0/
            End Map
         End Union
      End Map
      Map
         Character*41 Clump
      End Map
   End Union
End Structure

Structure /Dynamic_Record_Header/
   Integer*4 AutoAddress/0/
   Union
      Map
         Integer*2 Segment/0/
         Integer*2 Index/0/
      End Map
      Map
         Integer*4 AuxFieldFlag
      End Map
   End Union
   Character*15 Name/' '/
   Integer*4 Address/0/
   Integer*4 RecordLength/0/
   Integer*4 NRec/0/
   Character*15 StructureName/' '/
   Integer*4 SDAddress/0/
End Structure
```

```
Structure /Record_Pointer_Structure/
   Union
      Map
         Integer*4 HdrAddress/0/
         Integer*4 RecAddress/0/
      End Map
      Map
         Real*8 R8
      End Map
   End Union
End Structure

Structure /Record_File_Index/
   Logical*1 DirectField
   Union
      Map
         Integer*4 NDimensions
         Character*15 RecordName
         Integer*4 RecordLength
      End Map
      Map
         Integer*4 IndirectSize
         Integer*4 Offset
         Byte NestingMode
      End Map
   End Union
   Character*15 StructureName
End Structure


Structure /Dynamic_String_Desc/
   Integer*2 Length/0/
   Byte DType/14/
   Byte Class/2/
   Integer*4 Pointer/0/
End Structure

Integer*4 rec$sdh_size
Integer*4 rec$fd_size
Integer*4 rec$drh_size
Integer*4 rec$rps_size
Integer*4 rec$direct
Integer*4 rec$indrec
Integer*4 rec$indfld
Integer*4 rec$auxfld

Parameter (rec$sdh_size = 33) ! Byte alloc. for Struc Descr Hdr
Parameter (rec$fd_size = 41)  ! Byte alloc. for Field Descr
Parameter (rec$drh_size = 54) ! Byte alloc. for Dynam Rec Hdr
Parameter (rec$rps_size = 8)  ! Byte alloc. for Rec Ptr
Parameter (rec$direct = 0) ! Value of nesting mode for direct
```

```
Parameter (rec$indrec = 2) ! Nesting mode for ind. record
Parameter (rec$indfld = 1) ! Nesting mode for ind. field
Parameter (rec$auxfld = -60208)  ! Flag for AUX record type (no -1,0,1!)
```

Appendix C

IGORE/OSSE CM Policy

C.1 Procedure for CM'ing IGORE PROs

C.1.1 Programmer:

1. Code must have at least a minimal header comment block describing who wrote it, when, and what part of igore uses it (if it is not user interface code). Code which is intended to be general user interface or generalized programming tool must have a standard header block which can be accessed via idl procedure, doc_library.

2. Code which has been extracted from CMS and ready for reconfiguration must have "Modified" section in header block which identifies who, when what was done. Changes to the procedure interface, formal parameters or keywords, must be clearly identified. Changes to/addition of common blocks must be clearly identified.

3. On penalty of death (or at least, severe ridicule) the programmer guarantees that the code will compile WITH NO EXTENSIONS TO THE IGORE STANDARD PATH. In the case of changes to several routines which must be configured together, all of these routines must be placed in a test directory which can then be added to the standard IGORE path. The CM Manager must be given a list of all the routines that must be configured together.

4. In the test environment with only standard IGORE path (possibly extended as described above), the programmer shall verify appropriate execution of the new version of the routine. Details

5. The programmer shall verify that, for user interface routines, the header comment accessed via doc_library shall accurately describe the new interface or functionallity of the routine as modified.

6. Programmer must provide to CM Manager, for each routine:

    1. Programmers notes on changes made to routine - ie. the same stuff that might be in header comment block of routine.

2. Recommendation as to inclusion of associated changes into Users Release Notes. In general, changes to user interface should be included in user's release notes. Cosmetic, reorganization, or minor bugs changes probably should not. CM Manager will use this recommendation in addition to her/his own judgement in maintaining consistent content and level of detail in User Release Notes.
3. Programmer shall provide "Difference List" showing the differences in latest CM'd version of code wrt new submission.

7. At request of Software Config. Board, programmer will provide detailed user documentaion for inclusion into User's Ref document. This could include examples of usage or discussion of algorithms used.

C.1.2 CM Manager (CMM) or Designee:

1. CM Manager shall verify that the programmer has performed and provided as specified above.
2. CMM shall independently compile the routines to verify programmer work.
3. If interface, common block change is specified, CMM must search all CM'd code for usage of the routine. If there calls to the routine in other configured code which has not been identified by the programmer, Configuration of the code SHALL NOT PROCEED until impact of the change on the additional code has been resolved among the programmer, the CMM, AND the SOFTWARE CONFIGURATION BOARD (Leas, Purcell, Strickman, Johnson).
4. CMM or designee shall be responsible for extracting/duplicating doc header changes from routines into associated TEX files for IGORE REFERENCE GUIDE.
5. CMM shall maintain a Programmers Release Notes for IGORE which is is available on-line to identify the current state of IGRTST/DEV environments.
6. CMM shall maintain Users Release Notes which are created at each release of an IGORE version.
7. CMM shall maintain a list of known errors (extracted from programmers release notes of subsequent versions and OMARs) in the IGRCM version of IGORE.

Appendix D

IGORE Control Structures

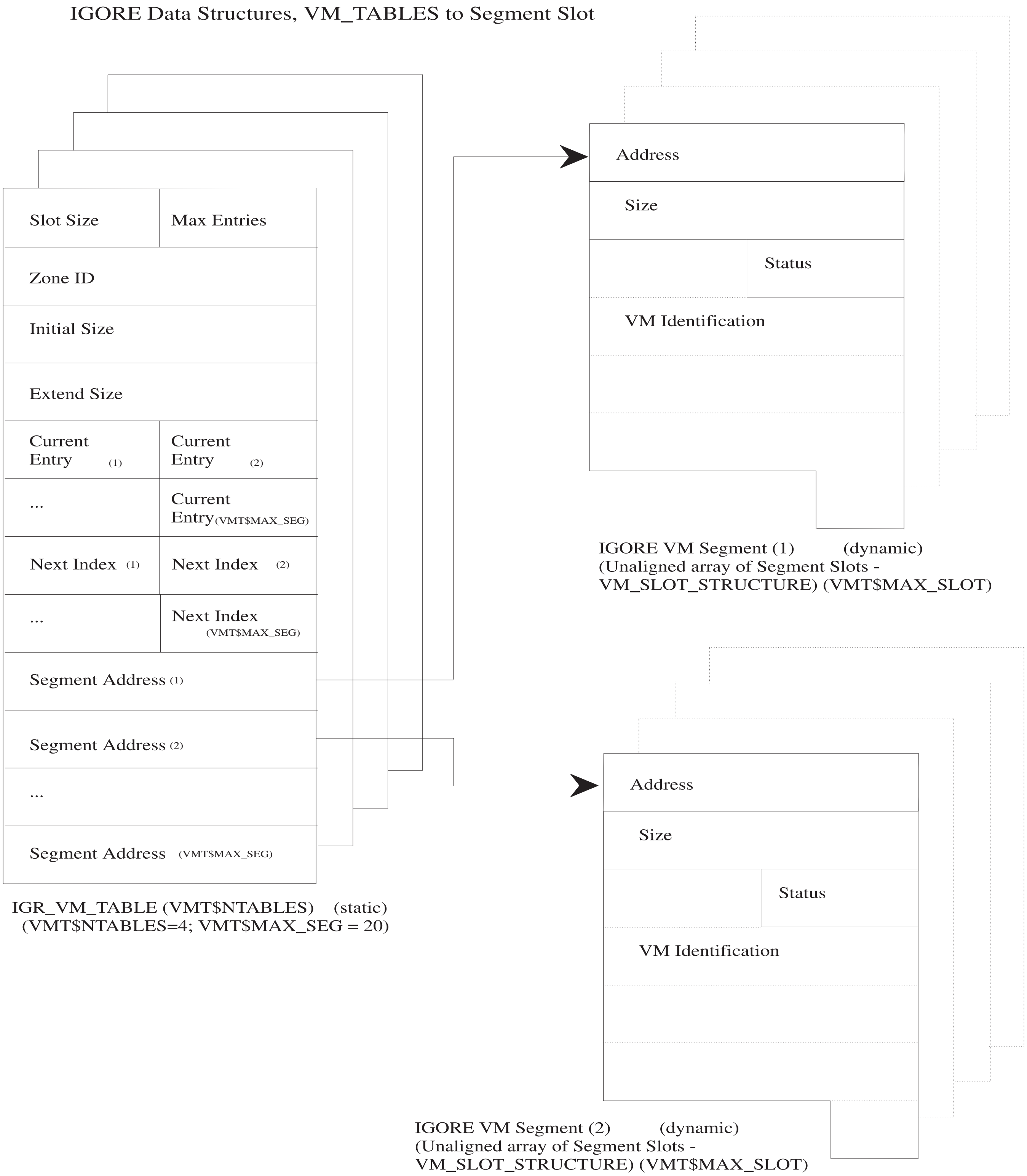

Figure D.1: IGORE virtual memory tables

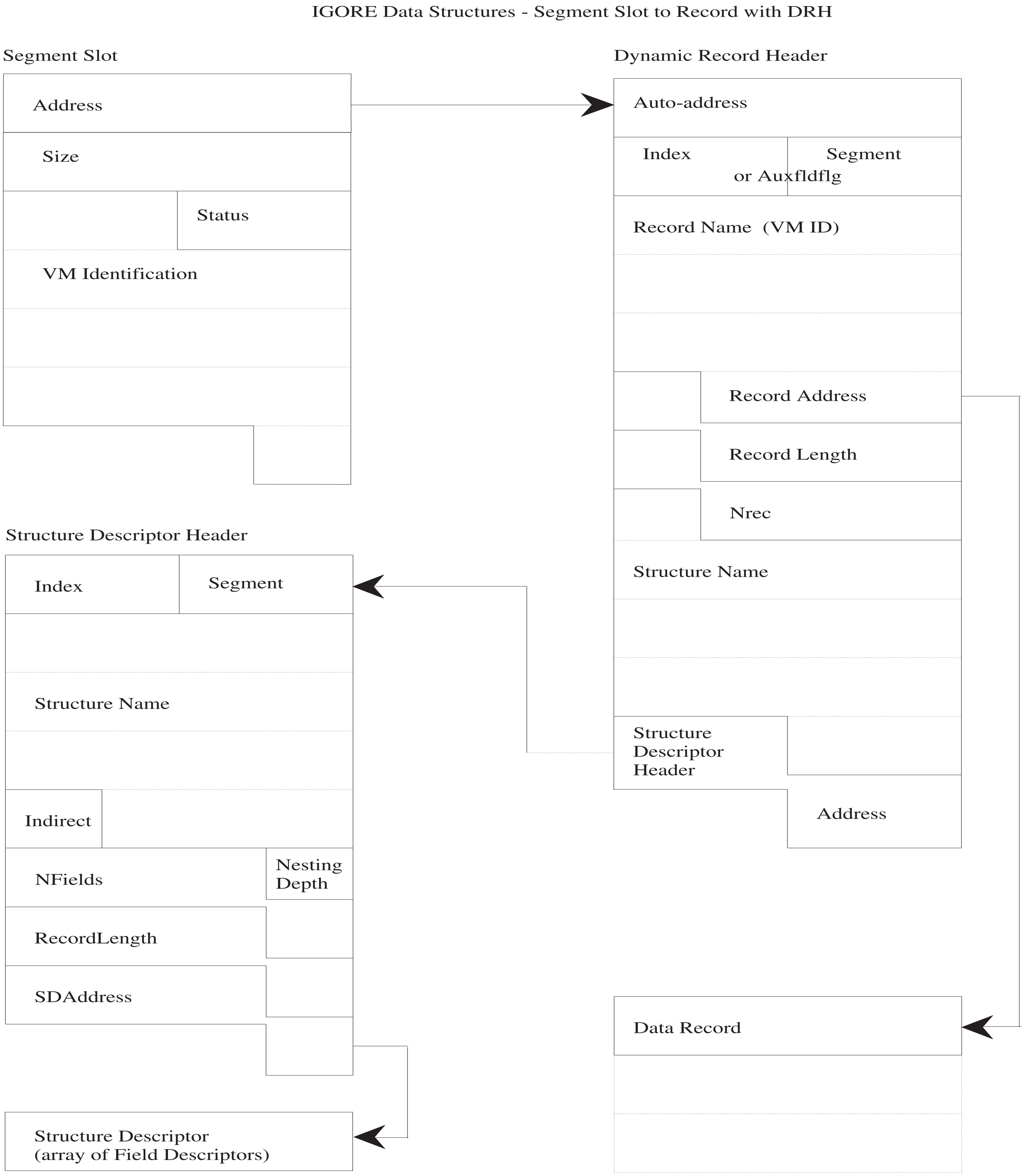

Figure D.2: Segment Slot to Dynamic Record Header

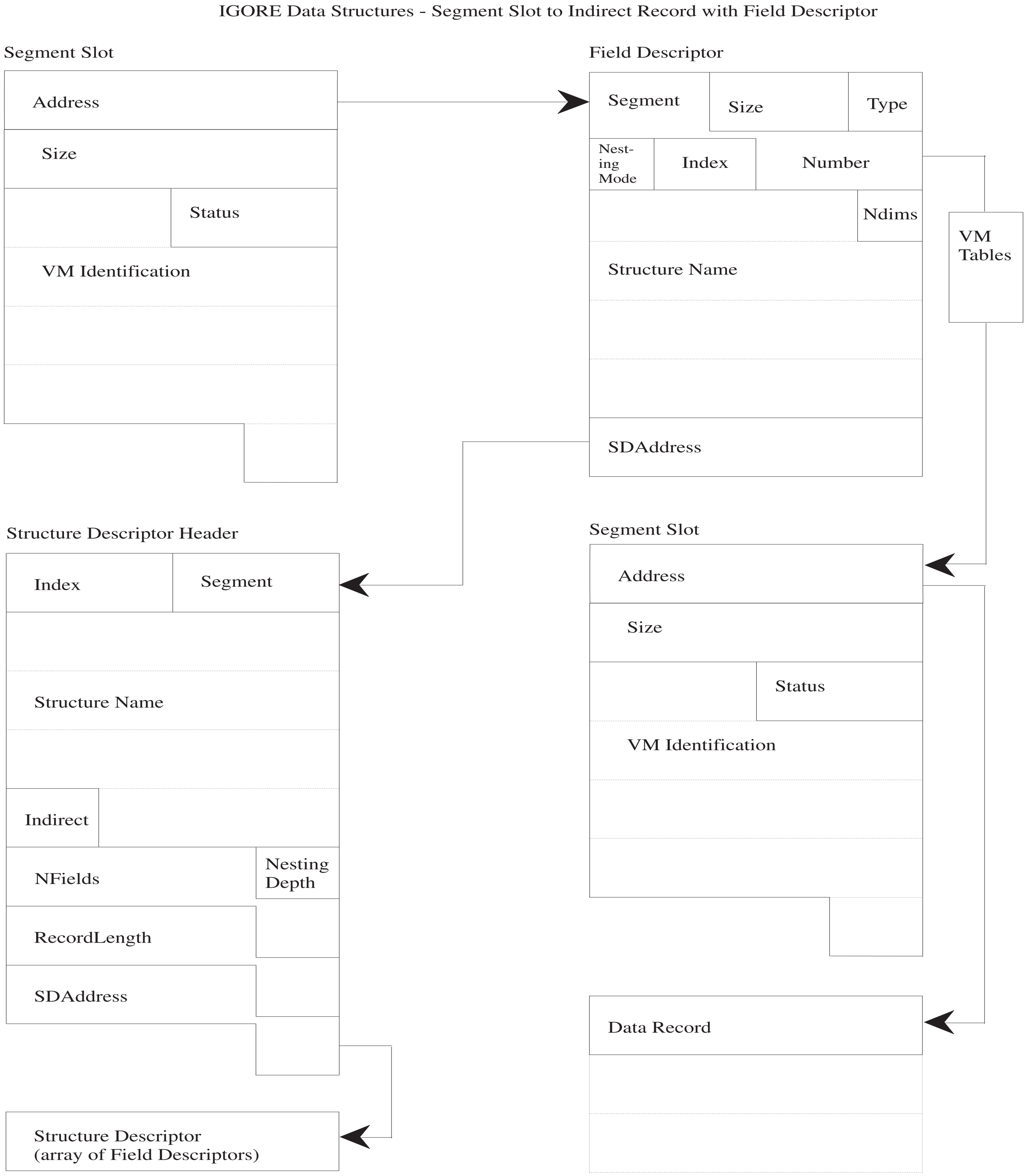

Figure D.3: Segment Slot to Indirect Record

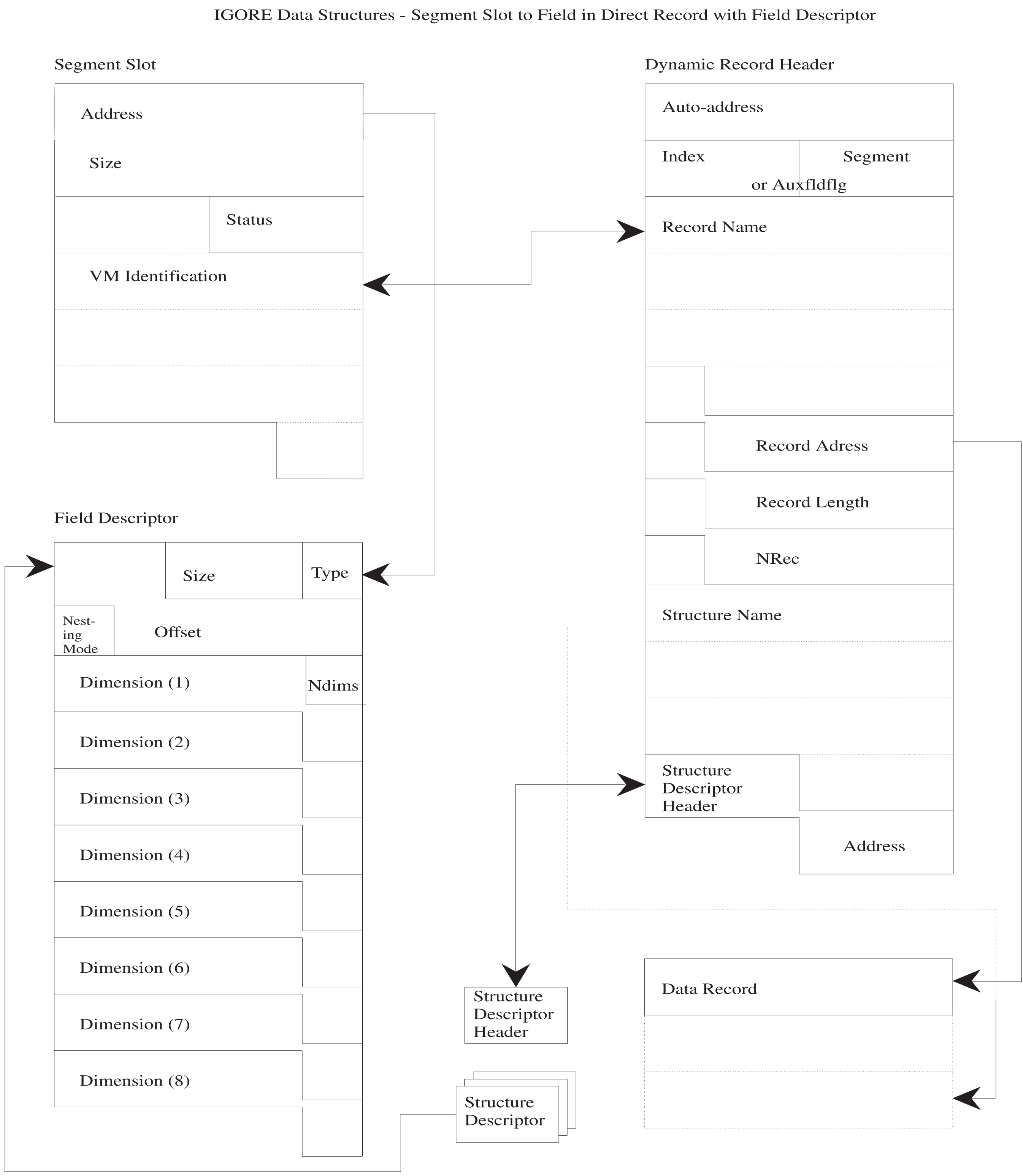

Figure D.4: Segment Slot to Indirect Field