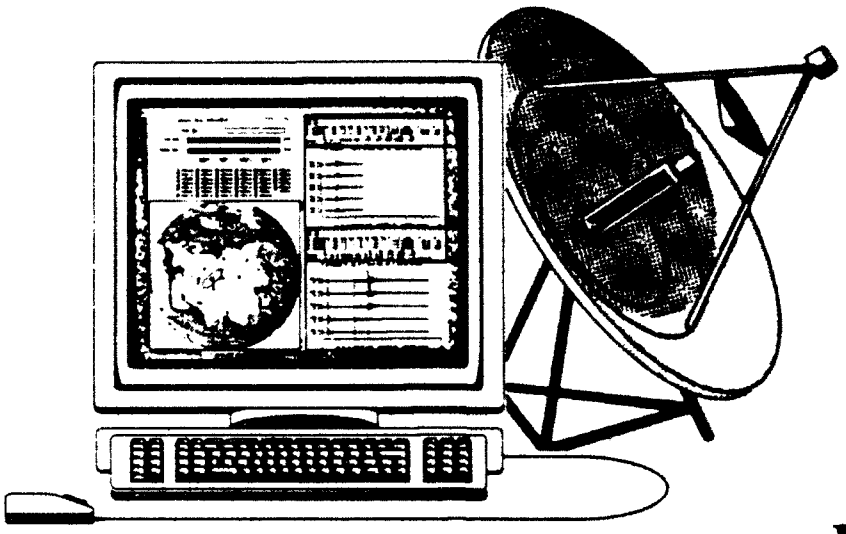
AD-A266 285

APRIL 1993

SAIC-93/1069

The Intelligent Monitoring System

Software Integration Platform



DTIC
ELECTE
JUL 01 1993
S D

SPECIAL TECHNICAL REPORT

12 April 1993

Jeffrey W. Given, Warren K. Fox, James Wang, Thomas C. Bache

Geophysical Systems Operation

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Sponsored by:
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
Nuclear Monitoring Research Office
ARPA Order Number 6266, Program Code No. 62714E
Issued by: DARPA/CMO
Contract No. MDA972-92-C-0026

Principal Investigator:
Dr. Thomas C. Bache
(619) 458-2531

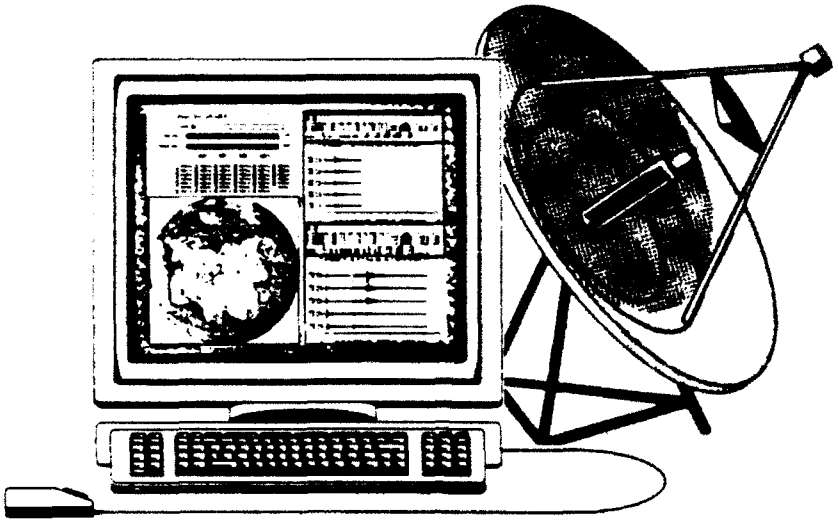
93 6 29 07 7

398290 93-14847

APRIL 1993 SAIC-93/1069

The Intelligent Monitoring System

Software Integration Platform



SPECIAL TECHNICAL REPORT

12 April 1993

Jeffrey W. Given, Warren K. Fox, James Wang, Thomas C. Bache

Geophysical Systems Operation

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Sponsored by:
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
Nuclear Monitoring Research Office
ARPA Order Number 6266, Program Code No. 62714E
Issued by: DARPA/CMO
Contract No. MDA972-92-C-0026

Principal Investigator:
Dr. Thomas C. Bache
(619) 458-2531

Table of Contents

ABSTRACT 1

I. INTRODUCTION 2
- 1.1. Background 2
- 1.2. Overview 3

II. SOFTWARE INTEGRATION PLATFORM (SIP) 9
- 2.1. Distributed Applications Control System (DACS) 9
  - 2.1.1 The CommAgent 10
  - 2.1.2 The DACS Manager 13
- 2.2. The Process Manager 15
- 2.3. Data Management System Interface 18
  - 2.3.1 CSS3.0/IMS Interface 20
  - 2.3.2 The Generic Database Interface 22
  - 2.3.3 The Dynamic Object Interface 22

III. SUMMARY 29

IV. DEFINITIONS, ACRONYMS AND ABBREVIATIONS 31

V. REFERENCES 32

APPENDIX 33

THE CSS3.0/IMS Database Interface 33
- libdb30 34
  - Synopsis 34
    - Calls From C Applications 34
    - Calls From Fortran Applications 41
  - Description 58
- libdbims 61
  - Synopsis 61
    - Calls From C Applications 61
    - Calls From Fortran Applications 68
  - Description 71

The IMS Software Integration Platform

ABSTRACT

The *Software Integration Platform* (*SIP*) supports automated and interactive distributed processing for the Intelligent Monitoring System (*IMS*). The *SIP* addresses the practical problem of integrating existing software and data archives into a processing system distributed on a network of UNIX workstations. It consists of software components that are widely applicable to other scientific data-processing operations. The *SIP* is divided into two subsystems. The Data Management System (DMS) manages shared data stored in archives distributed over a wide-area network. The Distributed Applications Control System (DACS) handles inter-process communication (IPC) and process control for user applications distributed over a local-area network. The data archives managed by the DMS consist of commercial relational database management systems (RDBMS) supplemented by UNIX file systems. User applications access data stored in the archives through the Data Management System Interface (DMSI). The DMSI allows global access to data independent of a specific physical archive. Because *IMS* requires the capabilities provided by commercial RDBMS products, the DMSI includes extensive support for these products. The DACS divides the IPC and process-control services between two applications. The CommAgent provides message routing and queueing. The DACS Manager interprets the IPC and monitors local-area network resources to decide how and when to start processes. Working together, these applications isolate user applications from network-specific details. The messaging facilities of the DACS enable the distributed system to exploit concurrency and pipelining to expedite data processing. The Process Manager is a general application developed for the DACS that manages the processing of data through complex configurable sequences of user applications. All components of the *SIP* exploit commercially available software products and anticipate current trends in software-product development.

DTIC QUALITY INSPECTED 5

Accesion For
NTIS CRA&I X
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

I. INTRODUCTION

1.1 Background

Over the past several years we have been developing a new generation of technology for automated and interactive interpretation of data from a network of seismic stations. The most complete expression of this technology is the Intelligent Monitoring System or *IMS* (Bache *et al.* 1990a, b, 1991), which was initially developed to process the data from four NORESS-type seismic arrays in Europe. These arrays produce over 100 channels of data with a total volume of about 1.2 Gybtes/day. Automated processing by the *IMS* produces about 500-1000 characterized signal detections per day (many turn out to be "noise"), and these signals are interpreted automatically to locate 50-100 events/day. Many MIPS are required for these automated processes, but they are easily organized into separate processes that can proceed in parallel or in a pipeline sequence with the time required to complete each step being large compared to the time to transfer information between steps. Organizing the processes this way also provides an architecture that scales naturally by replicating processes. Thus, as was realized when we did our initial design in 1986, this kind of problem fits naturally with emerging concepts for solving big problems by dividing them into semi-independent processes that are distributed across networks of (relatively) inexpensive UNIX workstations.

While the UNIX distributed processing concept is attractive for systems like *IMS*, the technology for fault-tolerant management of fully automated distributed processing was quite immature in 1986, and commercial products have still not emerged to handle many important management and administration tasks. That is, while there is excellent support for client-server relationships between processes via RPC (remote procedure call) mechanisms, these mechanisms must be extended to support the complicated inter-process relationships in a dynamic, data-driven distributed processing system like *IMS*. Additional mechanisms are needed for fault-tolerant inter-process communication (IPC), system reconfiguration in response to changing processing requirements, and implementation of policy for restarting or reconfiguring the system when faults occur.

The *IMS* is one product of the Nuclear Monitoring Research and Development (NMRD) project, which also includes requirements to develop several other automated and interactive systems to process seismic data from other seismic stations and networks (e.g., Bratt, 1992). Also, there is a continuing requirement to integrate (for test and evaluation) software modules developed by members of a disparate community of researchers at other institutions, and there is a general requirement for an open system architecture with minimal dependence on particular software and hardware vendors.

To meet these requirements, we have developed the *Software Integration Platform* (*SIP*) described in this report. This *SIP* provides the infrastructure for *IMS*, our largest and most complex system. However, it is also used for much smaller and simpler systems (down to those made up of only a few loosely connected processes on a single workstation) because of its convenience, flexibility and fault-tolerance.

1.2 Overview

The overall architecture of the *IMS* is data centric and data driven (Figure 1.1). *IMS* applications share *entities*[1] (i.e., data) described by a common data model and stored in data archives managed in a globally accessible Data Management System. *IMS* accepts seismograms as input entities and responds by creating entities that describe those seismograms and interpret them as seismic events. The *IMS* combines automated data acquisition, processing and interpretation with subsequent analyst interaction to validate the automated results.

Several loosely coordinated groups (or *sessions*) of closely coordinated applications share the *IMS* data-processing tasks. These sessions are distributed over a wide-area network (WAN); the applications within each session are distributed over a local-area network (LAN). Inter-session (i.e., WAN) communication requirements are minimal. A session primarily communicates with another session by polling the Data Management System for new or modified entities. Within each session the applications require message-based inter-process communication (IPC) for adequate operation and control.

The *SIP*, shown in Figure 1.2, is a collection of applications and interfaces that provide an *IMS* session with access to the Data Management System and IPC. While its development was motivated by *IMS*, the *SIP* is entirely independent of *IMS*-specific requirements. It provides support for a wide range of UNIX distributed-processing systems consisting of dozens of processes distributed on a local-area network. The *SIP* is divided into two subsystems: the Data Management System, which handles all aspects of managing shared data throughout the WAN, and the Distributed Application Control System, which provides the mechanisms for interprocess communication and control. In the following, we describe how user (i.e., *IMS*) applications interact with the *SIP* components and each other.

The user applications in each session retrieve and store entities through the Data Management System Interface (DMSI). The DMSI isolates applications from the Data Management System, which, as shown in Figure 1.1, can include multiple data archives distributed over a geographically dispersed wide-area network. In *IMS*, the Data Management System consists of a network of Oracle™ Relational Database Management Systems (RDBMS) supplemented by the UNIX file system and other media (i.e., optical disks and magnetic tape) for storing large and complex entities. Three interfaces currently make up the DMSI. These are: the *CSS3.0/IMS Interface*, which provides access to a relational database that follows the CSS3.0/IMS database schema (Anderson, *et al.*, 1990; Swanger, *et al.*, 1991); the *Generic Database Interface* (GDI), which provides schema-independent access to a relational database; and the *Dynamic Object Interface* (DOI), which provides access to general shared entities from heterogeneous data archives including relational and

1. Throughout this report, the term *entity* refers to a collection of attributes that describe a real-world thing. Where confusion is possible, the term *entity type* distinguishes between a generic collection of attribute names and a specific collection of attribute name-value pairs.

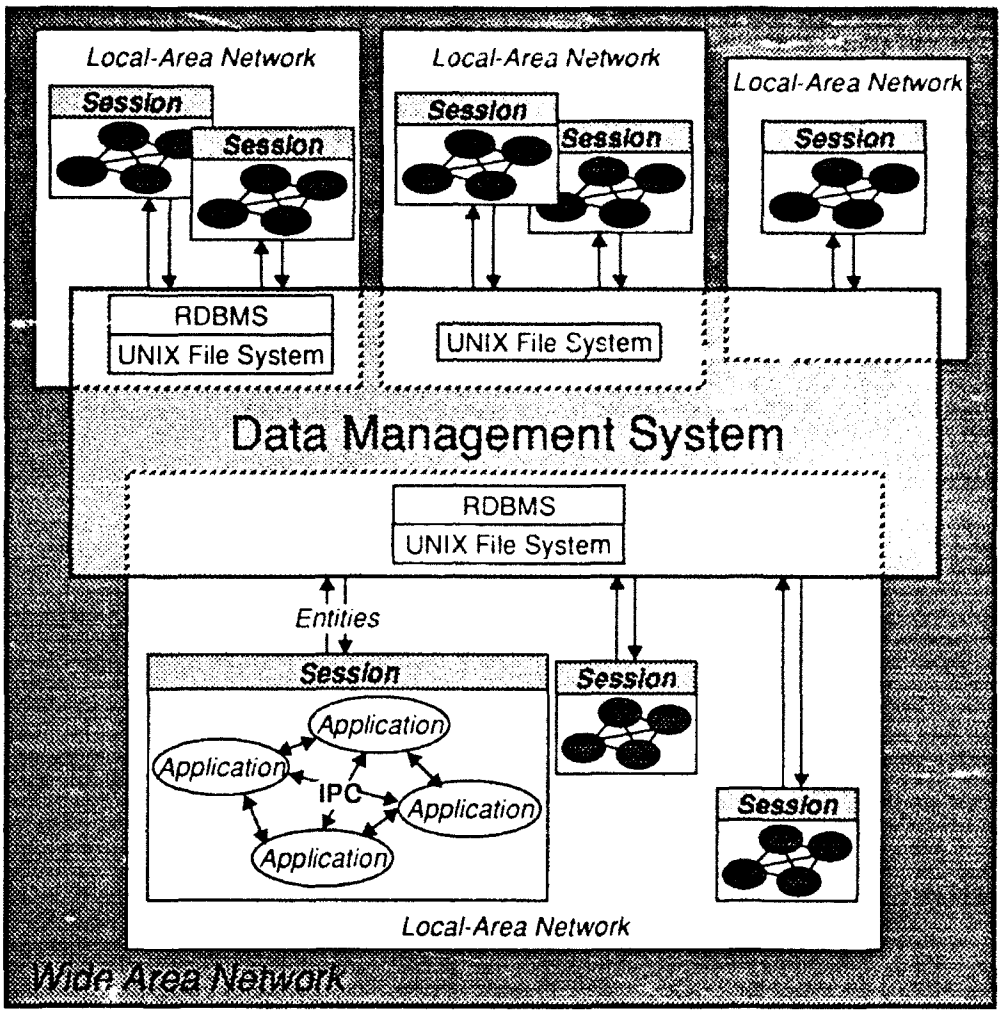


Figure 1.1. The organization of data processing in the *IMS*. Processing tasks are distributed over several local-area networks. Within each local-area network, there are sessions consisting of closely coordinated applications dedicated to a specific task. Each session shares entities managed throughout the wide-area network by the Data Management System.

non-relational databases. Of the three interfaces, the CSS3.0/IMS is the most widely used. The development of the GDI is complete, but it has not been completely integrated into existing *IMS* applications. The DOI is still under development but an early version provides critical functionality for the *IMS* Map program.

Within each session, the Distributed Application Control System (DACS) controls the physical behavior of the system. The DACS provides the mechanisms for inter-process communication (IPC) between user applications. Through the DACS, user applications send messages to command, alert, and inform other user applications. The DACS monitors the IPC traffic to determine where and when to start applications. It handles all aspects of configuring a group of user applications to a specific network of workstations. The IPC

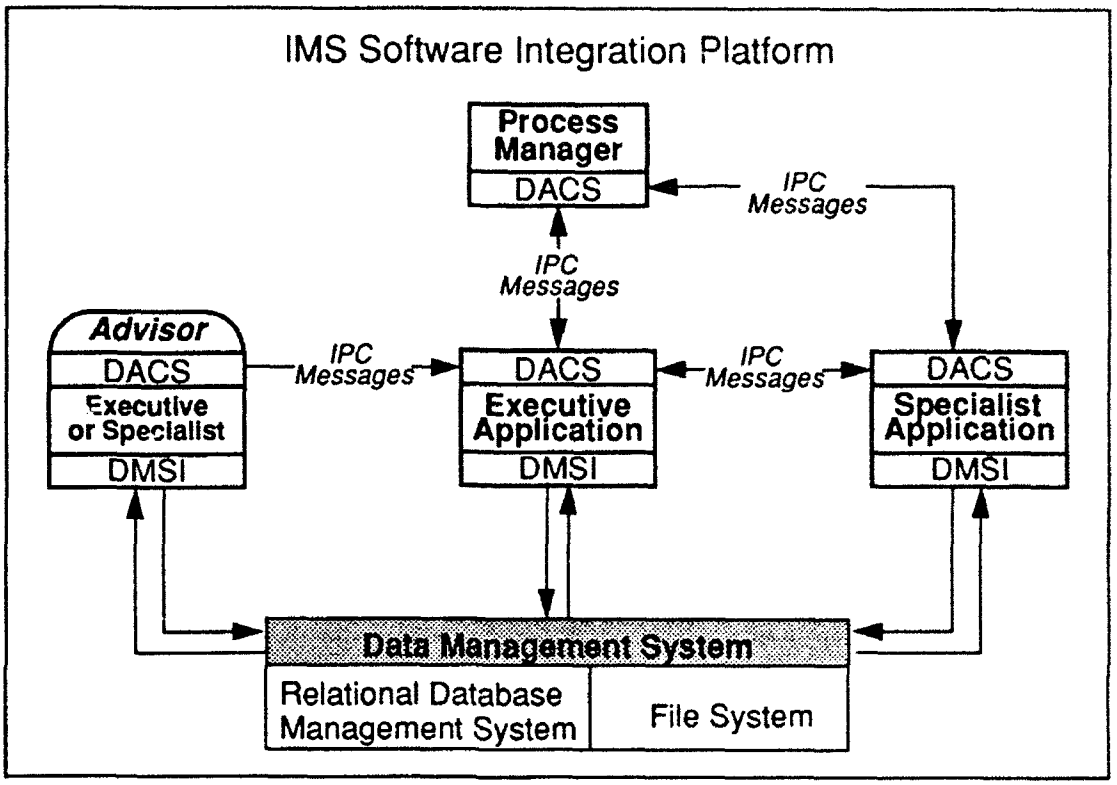


Figure 1.2 The Software Integration Platform. The applications access a common data model through a uniform interface to a global Data Management System. User applications transfer control through IPC managed by the Distributed Application Control System (DACS). Each user application is either a Specialist or an Executive based on its response to IPC messages. Advisor applications send unsolicited messages to Executives. A Process Manager provides logical control of complex sequences of Specialists.

messages passed through the DACS primarily transfer control between user applications in the *SIP*. The DACS does not directly provide facilities for efficient high-volume data-transfer, but it can coordinate such transfer between applications. Typically the IPC messages contain references to entities maintained in the Data Management System.

There are two major components in the Distributed Application Control System (DACS). One is the CommAgent, which handles the sending and receiving of messages, routes messages between physical processes, and monitors the current status (e.g., active or inactive) of the user applications. The CommAgent is independent of any specific network of workstations and isolates user applications from the physical details of network configuration. The second component is the DACS Manager, which manages network-specific information. Its primary function is to start processes when it detects unread messages for inactive applications. The CommAgent is a strictly procedural application while the DACS Manager is a highly configurable application that uses complex network-specific application-specific rules to manage a session. The DACS Manager also provides the human interface to the DACS allowing an operator to control the decision-making process.

In Figure 1.2, we distinguish between two types of user applications: Executive and Specialist applications. We assume the behavior of any application is completely determined by the messages sent to it. The behavior of a Specialist application is the result of messages from, at most, one other application. The current state of an Executive application is the result of unsolicited messages from many applications. A Specialist is essentially a server in a simple remote procedure relationship; an Executive is an event-driven application responding to messages from several other applications that are outside its control.[1]

A Specialist responds unambiguously to messages from another application. That is, the same message sent to the same Specialist will always yield the same results. An Executive responds a message according to its current state, which may be the result of messages from many sources. A message to a Specialist is a command; a message to an Executive is analogous to advice. This type of interaction motivates adding to Figure 1.2 the applications called the Advisors, which send unsolicited messages to an Executive. Advisor applications are either Specialists or Executives, but they have the additional property that they notify an Executive of events that are subsequently processed (or ignored) according to the Executive's current state.

The classification of applications is used by the DACS when it associates an abstract name of an application and a specific physical process. The DACS must restrict IPC access to a Specialist to prevent other applications from sending the Specialist messages while it is engaged in an assigned task. On the other hand, access to an Executive is unrestricted to allow any Advisor to initiate communication at any time. Therefore, the DACS maintains private identifiers for Specialist applications and public identifiers for Executives.

Because Specialist identifiers are private, several instances of the same Specialist may be simultaneously active within the same session. Thus, a session may exploit concurrency to optimize processing tasks. The *Process Manager* is an *SIP* application that coordinates the simultaneous processing of many entities of the same type through complex sequences of Specialist applications. The Process Manager exploits both concurrency and pipelining and is configurable to a wide range of specific processing requirements.

The application names in Figure 1.2 are analogous to familiar organizational structures. In a typical session, there are a few Executives with a global perspective that make the overall decisions guiding the construction of the final data-processing product. Thus, the complex decision-making functions are concentrated in the Executives, while straightforward procedural functions are distributed among the Specialists. The Executives are notified of important outside events by independent Advisors, which are either Specialists or other Executives.

---

1. All of the interprocess interactions can be described in terms of remote procedures and client/server relationships, but such a description offers little insight into the organization of user applications in the *SIP*. A familiar analogy to a Specialist is a network file server. An Executive application is analogous to a display server (e.g. an X-window server), which accepts events from several independent client applications..

A complex system like *IMS* includes dozens of applications organized into sessions of closely coordinated applications interacting as described in Figure 1.2. Typical *IMS* sessions are shown in Figure 1.3. The Analyst Review Station (ARS) is an Executive application that selects data from the Data Management System for review and interactive analysis. ARS functionality is extended by analyst tools with a simple server-to-client relationship to the ARS (i.e., Specialist applications). An example of an Advisor application to ARS is the Map program, which displays and manipulates digital maps, images, and entities tied to geographical location. With the Map, a user may select entities based on their geographical attributes and send messages to ARS identifying the selected entities. However, ARS does not request the Map to send messages, the Map does not know how ARS handles the messages, and ARS cannot prevent other applications from sending messages at the same time. Although it may be an Advisor application to ARS, the Map is also an Executive that responds to events from several external sources, including ARS.

Automated processing in *IMS* is organized in sessions consisting of Specialist applications. In these sessions one or more Specialists are responsible for periodic polling of the Data Management System for specific data conditions. These data conditions are usually the result of processing activity in another session, often at a remote site. A typical circumstance occurs when new seismic events have been formed by the session responsible for interpreting detections and locating events. In another session the polling Specialist (*NewOrid* in Figure 1.3) detects the new events and sends messages to other Specialists to retrieve waveforms from the WAN, compute magnitudes, perform event identification, and produce a top-level summary of system results. This sequence of processing is organized into a pipeline managed by the Process Manager. Bache (1990a, 1991) describes in detail those applications that comprise the automated processing system in *IMS*.

The preceding is an abstract overview of the *SIP*. In the remainder of this report we describe the major *SIP* elements in more detail. Section 2 includes subsections on the DACS and its key elements (CommAgent and DACS Manager), the Process Manager, and the DMSI. This description of the DMSI is supplemented by an Appendix that documents the CSS3.0/IMS database interface. Separate reports describe the CommAgent (Given, *et al.*, 1993), the DACS Manager (Fox, 1993) and GDI (Anderson, *et al.*, 1993).

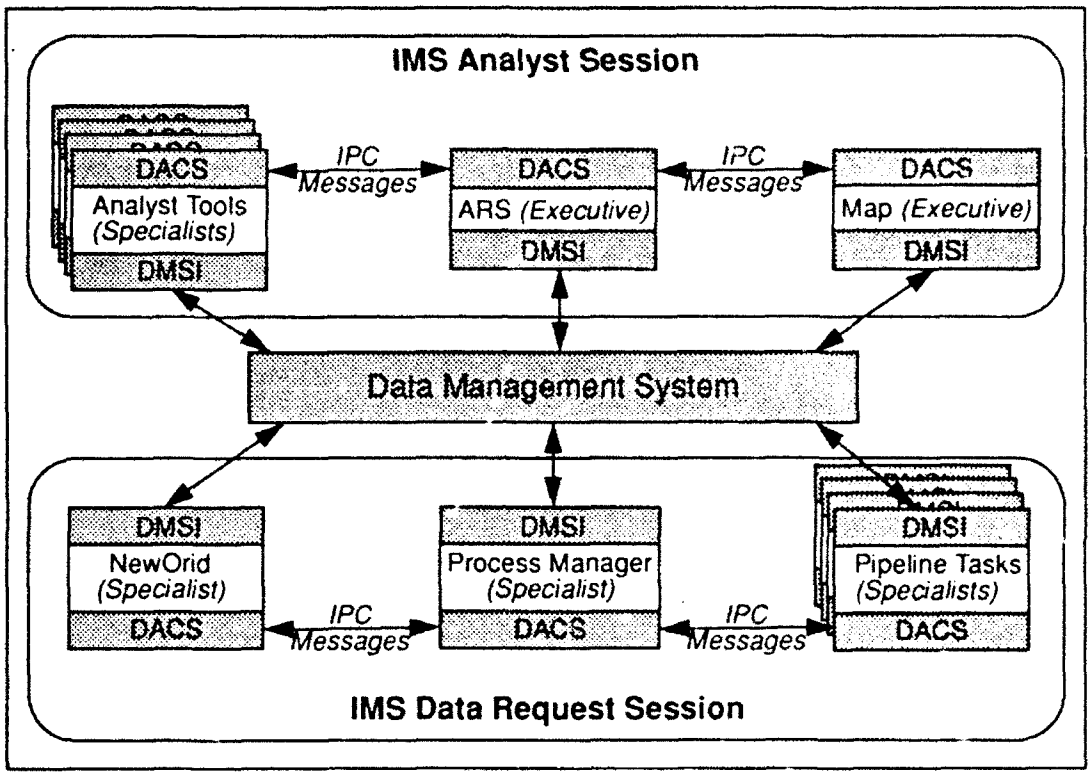


Figure 1.3. Examples of sessions from the *IMS*. In the Analyst Session, an analyst interactively reviews the results of the automated event location using the Analyst Review Station (*ARS*), the *Map*, and additional analyst tools that include specialized signal processing and data display functions. In the Data Request Session, *NewOrid* detects new seismic events created by another session and initiates processing by a *Process Manager* to collect additional seismic data distributed throughout the WAN and to process that data through tasks that include magnitude calculation and event identification.

II. SOFTWARE INTEGRATION PLATFORM (*SIP*)

The three main elements of the *SIP* are the Distributed Applications Control System (DACS), the Process Manager, and the Data Management System Interface (DMSI). These are described in this section.

2.1 Distributed Applications Control System (DACS)

The DACS provides the framework to connect distributed UNIX software processes into a coherent data processing and analysis system. It is separated into two distinct components: the *CommAgent*, which manages the network- and application-independent procedural aspects of interprocess communication (IPC), and the *DACS Manager*, which oversees the complex details of configuring a set of user applications to a specific network. These applications, working together, provide the IPC support for a session of closely coordinated applications. A session may include user-driven, interactive applications or it may consist entirely of applications performing automated data processing. The Process Manager is a closely related *SIP* element designed to work in conjunction with the DACS to control a complex logical processing sequence, and it is described in Section 2.2.

The CommAgent (see Section 2.1.1 for more detail) handles the posting and delivery of IPC messages between the applications in a session. The CommAgent maintains a table associating logical application names with specific processes and message queues. This table, which is the result of the IPC, describes the physical state of the distributed system. This state information indicates, for example, which applications are active and busy performing a specific task, which applications are active without an assigned task, and which applications are inactive with a pending task (indicated by undelivered IPC messages). The CommAgent takes no external action based on its state table; it holds messages indefinitely until a suitable recipient picks them up. The CommAgent cannot start a process to pick up a message, it makes no decisions about how long to hold a message for delivery, and it has no control on the sources, destinations, or contents of the messages. The only interface to the CommAgent is through an application-program interface (i.e., a library of functions available to the applications); there is no external user interface. However, the contents of the state table can be transferred to a managing process, such as the DACS Manager, that can reconfigure the physical system in response to the CommAgent's current state.

The DACS Manager makes the decisions concerning where and when to start applications. For example, when a message to an inactive application appears in the CommAgent state table, the CommAgent notifies the DACS Manager, which selects a workstation and executes the application with the proper environment and command arguments. To make these decisions the DACS Manager monitors session resource utilization (e.g., active processes vs. overall number allowed), overall usage of the LAN, specific

hardware and software requirements (e.g., licenses), and the state of required external systems (e.g., database servers or file systems). The DACS Manager has a user interface that allows an operator to start and terminate selected applications or to modify system parameters. The interface also allows a user to monitor and modify the CommAgent state table.

The DACS is, therefore, divided between one application that follows rigorous, predictable, well-defined procedures and a second application that follows a complex, possibly heuristic, decision-making process. As the former, the CommAgent is inherently more reliable, which is a crucial property for a process that maintains critical system state information in a volatile form (i.e., in an active UNIX process). When controlling a complex system with many applications competing for limited resources, the DACS Manager must be expected to be less reliable. Therefore, the two applications have been designed so that the DACS Manager, like any other user application, does not interfere with the action of the CommAgent. In fact, the DACS Manager can be terminated at any time during a session and replaced by either another reconfigured DACS Manager or operator intervention.

The DACS is configurable to accommodate a wide range of requirements for a group of applications. Both the CommAgent and the DACS Manager are designed to be easily started and stopped by naive and novice users without restrictive administrative overhead or adverse system impact. Any number of DACS Manager - CommAgent pairs can be active on a LAN, each responsible for a distinct work session. The DACS provides a convenient interface for a novice user to control a session distributed over one or several workstations. Small sessions are simple to configure, yet there is sufficient flexibility to impose adequate control on large sessions. Although the CommAgent and the DACS Manager have evolved together, they are independent applications. The design permits further independent development of the DACS Manager to exploit expert-system techniques for managing complex systems.

2.1.1 The CommAgent

The CommAgent coordinates the delivery of messages between applications distributed over a network of workstations. It provides both message routing and message queueing services. Applications post messages with the CommAgent that are destined for other applications identified by their abstract names. Applications receive messages by requesting them from the CommAgent. The sending and receiving applications need not synchronize to pass messages; they need not even be active at the same time. The CommAgent holds messages indefinitely until a recipient requests them. The application-program interface to the message handling facilities of the CommAgent is through a few simple library functions. Applications require no network-specific information beyond a single parameter necessary to locate a specific CommAgent running on any workstation in the LAN.

The operational basis of the CommAgent is a state table that associates a *name*, a *task* (specified by a task-id), a *process* (specified by a process-id), and a *message queue* (see Figure 2.1). The *name* identifies an application that can execute somewhere on a network of workstations. In operation, *processes* send messages that define *tasks* to be completed by other *processes* specified by *name*. A *task* is not completed until a UNIX *process* (specified by a process-id) with the associated *name* starts, receives the messages, and notifies the CommAgent that it has completed the task.

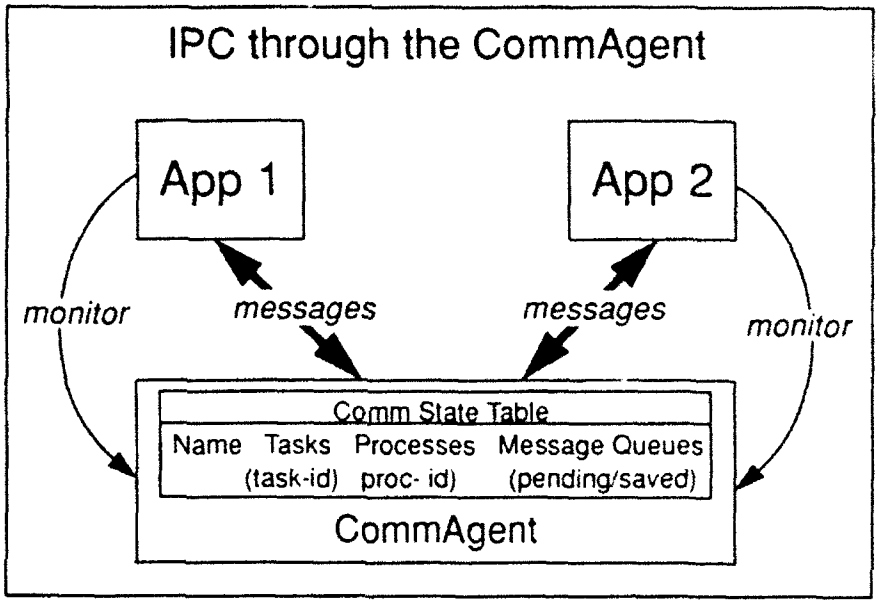


Figure 2.1 The CommAgent handles IPC between applications on a network. The state table contains *message queues* associated with a destination identified by a *name*, a (logical) *task*, and a (physical) *process*. A *process* sends a message by inserting it in the *message queue* of the destination. It receives a message by querying its queue. An application must first establish a connection to the CommAgent, which is then monitored to detect unexpected termination of the process.

When an application is started on a workstation, it is assigned a unique process-id by the operating system. It joins a session by opening a connection to the corresponding CommAgent and informing the CommAgent of its name and process-id. If the CommAgent has a pending task (e.g., pending messages) for an application with the same name, the CommAgent associates the name, process-id, task-id, and message queue. The message queue will contain the specific messages that direct the processing function. When the process completes the assigned task, it notifies the CommAgent, which will disassociate the task-id from the process and delete the task-id. When a process exits, it disconnects from the CommAgent. If the process disconnects before disassociating an assigned task-id, that task-id (and its corresponding message queue) will be reassigned to another process with the same name when it connects. The CommAgent monitors all connected processes and can, therefore, detect when one terminates abnormally (i.e., without disconnecting).

The CommAgent is an inefficient facility for transferring large quantities of data between applications. It is designed to handle short messages that transfer control between applications. In general, data transfer is by reference to entities stored in the Data Management System. Applications often access these entities through IPC mechanisms that are outside control of the DACS (e.g., through an RDBMS). Otherwise, the transfer of large volumes of data is most efficiently handled directly between applications. The DACS can coordinate this interaction by handling the network-specific details needed to establish the required communication connections and starting any requested data-server processes.

As an example, consider two applications, *SelectData* and *ViewData*, which are executable programs that can run on workstations on the LAN. When *SelectData* is started, it is assigned a process-id, *proc-id-Select*, by the operating system (e.g., from the hostname and system process-id). That instance of *SelectData* directs the CommAgent to associate its name, *SelectData*, with the specific process-id, *proc-id-Select*. After that, when any message is sent to or from that process, the CommAgent will associate it with a task, *task-id-Select* (i.e., it assigns it a unique task-id). As an active process, *SelectData* may send messages to an application named *ViewData*, for which the CommAgent has no associated process. The CommAgent will then assign the name, *ViewData*, to a new task, *task-id-View* and place the message into the associated message queue for later delivery. The process, *proc-id-Select*, can continue or it can wait (i.e., block) for a reply from an instance of *ViewData* concerning the status of the task. When a process opens a connection with the CommAgent under the name *ViewData*, the CommAgent will assign it the pending task, *task-id-View*, and make the associated messages available for delivery. The CommAgent can take no action to start *ViewData*; it does not have the necessary information. It can, however, be directed to notify another application (i.e., the DACS Manager) of all changes to its status table. That application can decide when and where to start the user applications.

The CommAgent distinguishes between two kinds of tasks, *public* and *private*. When a process initiates communication with another application, the CommAgent assigns an identifier that specifies a task. The sender may specify that the identifier be private, in which case the CommAgent generates an identifier and provides it to the sender so that additional messages can be directed to the same task. Alternatively, the sender may require that the identifier be public. In that case, the task is identified by a global address known to all applications in the session. Messages to a private application are unambiguously ordered. The messages from any single application to a public task are also strictly ordered, but they may be unpredictably interspersed with messages from other processes. A private task is a Specialist application; a public task is usually an Executive.

The CommAgent was developed using ISIS™, an IPC message management software system and application toolkit initially developed at Cornell University under DARPA support. A commercially supported version is available from ISIS Distributed Systems™, Ithaca, New York. ISIS provides a simplified and robust remote procedure (RPC) interface as well as tools for monitoring the status of processes. ISIS also has additional capabilities beyond those offered by standard RPC mechanisms. In particular, it is

possible to replicate a CommAgent process (i.e., execute several CommAgents in a session). If the CommAgent is replicated, any individual CommAgent process can fail without interrupting the session. Replication protects critical sessions from a CommAgent failure since the CommAgent process must always be active in order for a session to exist. Furthermore, replication permits a CommAgent to be moved from one machine to another without affecting the session. Using these features, the operation of a session is very tolerant of system errors. With the capabilities provided by the DACS Manager, described in the following section, the physical state of a session may be easily manipulated during operation to accommodate changes in the underlying physical system.

2.1.2 The DACS Manager

The DACS Manager provides the physical process-control services for the DACS. Through the DACS Manager, either a user or another application may request a change in the applications active in a session. Though an independent application, the DACS Manager is explicitly designed to work with the CommAgent to provide overall physical control of a session. The DACS Manager encapsulates all knowledge of the LAN so that all user applications are independent of LAN-specific details.

The DACS Manager monitors status changes in the CommAgent to decide which processes to start to complete requests for tasks. The CommAgent table associating names, tasks, and processes is a concise description of the physical system state. The DACS Manager receives and interprets the entries in this table to make its decisions. The relationship of the DACS Manager to the CommAgent is shown in Figure 2.2. The DACS Manager connects to the CommAgent the same as any other user process. It directs the CommAgent to transfer its state table and all subsequent updates to it. The CommAgent responds to the DACS Manager through the standard IPC interface as it normally responds to other user applications. That is, the DACS Manager has no special relationship with the CommAgent.

In the simplest case, the DACS Manager maps an application name to a specific system command and starts the application on any machine. The resulting process connects to the CommAgent, which then notifies the DACS Manager that the process started successfully. However, the DACS Manager does have unlimited freedom to start applications. It interprets the request for an application as a request for system *resources*. The DACS Manager then matches the resource request with the available system, session, and workstation resources to schedule when and where to start the application. To perform this function, the DACS Manager monitors resource utilization in order to enforce constraints on the configuration of the physical system. In general, what is considered a resource will be very specific to a session. Some sessions require detailed monitoring of many specific resources; others require tracking of only a few general resources. The DACS manager is designed to be configured to accommodate a wide range of session-specific requirements. In the following we discuss the general resource-allocation problems that are part of the overall DACS Manager design (although not fully addressed in the current operational version).

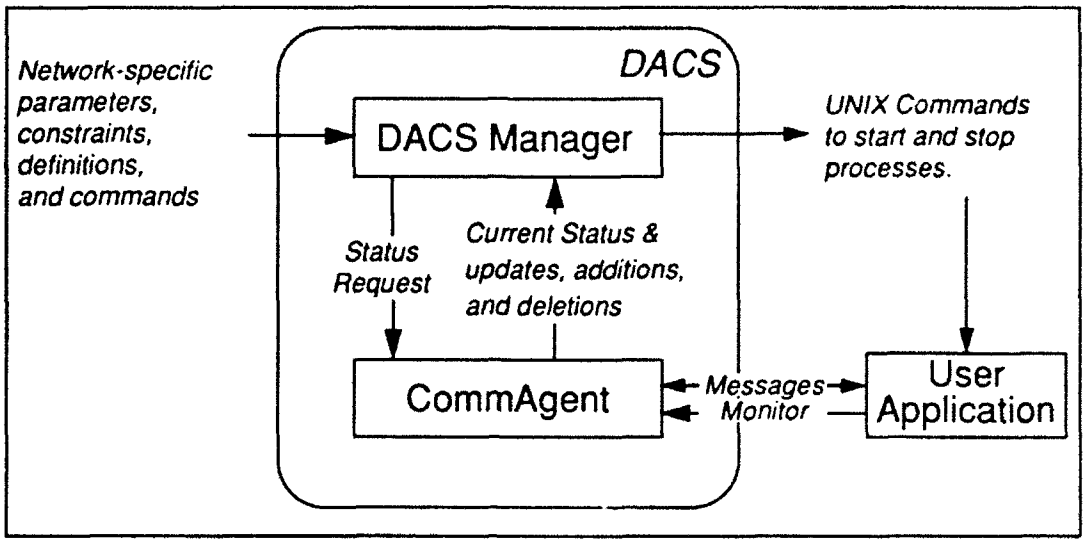


Figure 2.2. The interaction between the DACS Manager, the CommAgent and the User applications. The CommAgent notifies the DACS Manager of changes in its state table. The DACS Manager applies network-specific constraints to decide where and when to start a user application.

The simplest resources to manage are static machine attributes that must match the attributes required to support each application. For example, an application may require a specific type of hardware, a specific workstation, a software license, or some combination of these attributes. The DACS Manager uses simple pattern matching to select workstations with the required static attributes for a requested application.

At the next level of complexity, the DACS Manager tracks dynamic session resources. Each session is allocated quotas that limit the resources it can obtain from the LAN. These quotas limit the use of machines and software subsystems, both on an individual (i.e., machine by machine) basis and in the aggregate (i.e., groups of machines). The session is free to allocate these resources to its applications as necessary to achieve its objectives. If the quotas allocated to a session are much smaller than available LAN resources, many of the constraints imposed by overall LAN resource limits can be ignored. The DACS Manager has complete knowledge of the session environment, so tracking session resources is a relatively simple procedure. The DACS Manager provides a configurable framework for identifying, tracking, and allocating dynamic session resources.

The most difficult resources to track and manage are the dynamic attributes of the hardware (e.g., machines) and software (e.g., file systems, database management systems) that are shared with other sessions and users in a heterogenous work environment. In the simplest cases, it is sufficient for the DACS Manager to determine whether a specific workstation is operational. Other circumstances require an estimate of a workstation's current and anticipated extra-session load. Some sessions need to know the status of the data management systems (e.g., data-storage capacity, or the number of active server connections). In these more complex sessions, the DACS Manager must access sensors that monitor the status of hardware and software throughout the LAN. These sensors are usually

external processes designed to track the state of a specific external system (e.g., an Oracle RDBMS). Different sessions will require quite different information. The DACS Manager has an open design that allows it to be modified and configured to handle complex resource management situations. At the same time, small sessions are not unnecessarily complicated.

2.2 The Process Manager

The *Process Manager* (Figure 2.3) is a general application that manages a group of Specialist applications to exploit concurrency and pipelining. It is independent of any specific processing objective or group of applications. The Process Manager allows the configuration of a complex sequence of tasks organized around the processing of many entities of a specific type. It is designed around a specific processing model that is frequently encountered in a variety of data processing situations. In the following we describe that model and the Process Manager.

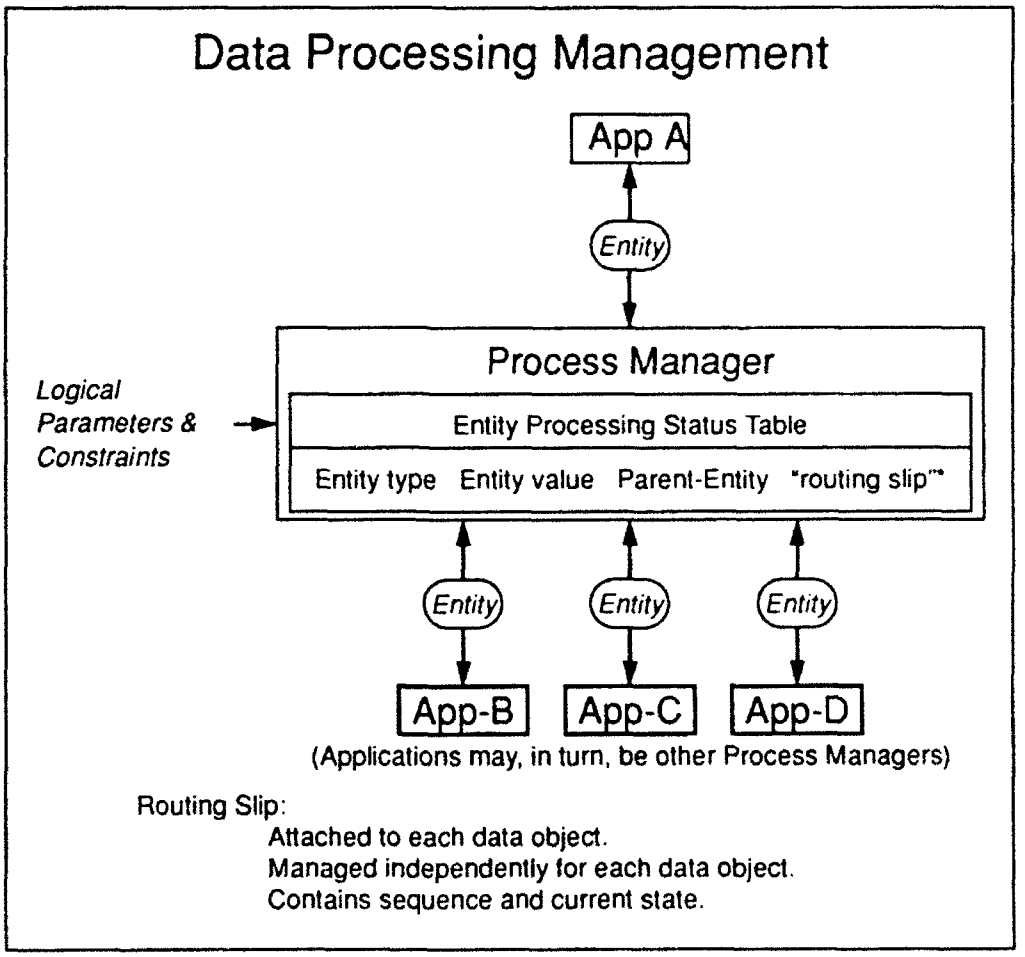


Figure 2.3 The Process Manager manages the processing of many entities of the same type through a processing sequence specified by a "routing slip" attached to each entity.

Consider a data-processing system organized around the processing of independent entities, which are collections of attributes that describe something (i.e., data). There may be several distinct types of entities with many instances of each. Entities of different types may be related hierarchically. That is, several child (or dependent) entities may be related to a single parent entity. Processing of the entities is analogous to an assembly line. Initially, only the primary attributes of an entity may be present in the Data Management System. During processing, the entity is sent through a sequence of applications that result in new attributes added to the entity or in the creation of related entities. At any time there usually are many instances of several types of entities in the system, and we seek to exploit distributed processing methods expedite their processing.

Specialist applications usually operate as procedures on a single instance of a particular type of entity. Thus, processing of an entity usually involves sending it to several Specialists in a particular sequence. For example, the entity may first be sent to application A; when A is complete it is to be sent to B; when B is complete it can be sent to C, D, and E in any order; when these are complete it is sent to F, which completes the overall sequence. The processing sequence and state for each entity can be concisely described in a set of routing instructions (referred to in Figure 2.3 as the "routing slip"). One way to speed processing of many entities is to organize a pipeline, which follows from the assembly line analogy. Another way is to exploit concurrency for those time-consuming Specialist applications, which extends the analogy to multiple assembly lines. The following discusses how the Process Manager allows the configuration and control of such a processing system.

For each type of entity, the processing sequence in the Process Manager is specified through configurable instructions, called the "routing slip", attached to each instance of the entity. An example processing sequence is shown in Figure 2.4. As implied by that figure, the routing slip must specify which processing tasks are to be executed sequentially, which may be executed concurrently, and which are conditional on the status of the previous task. The synchronization of processing is handled by the Process Manager, which, for each entity, keeps track of what tasks are complete, active, inactive, or in an error state. An individual application has no knowledge about the applications that precede or follow it. Therefore, processing instructions may be reconfigured to add new applications or rearrange the existing sequence without changing any of the existing applications. Furthermore, the instructions for an individual entity may be modified during the processing sequence.

The Process Manager recognizes hierarchical relationships between different types of entities. While processing one entity (the parent), an application may send several new dependent entities to the Process Manager. These are processed according to their own instructions. The parent entity may be instructed to wait at some point during its processing for its all of its dependents to reach a specified state in their sequences, (e.g., the parent

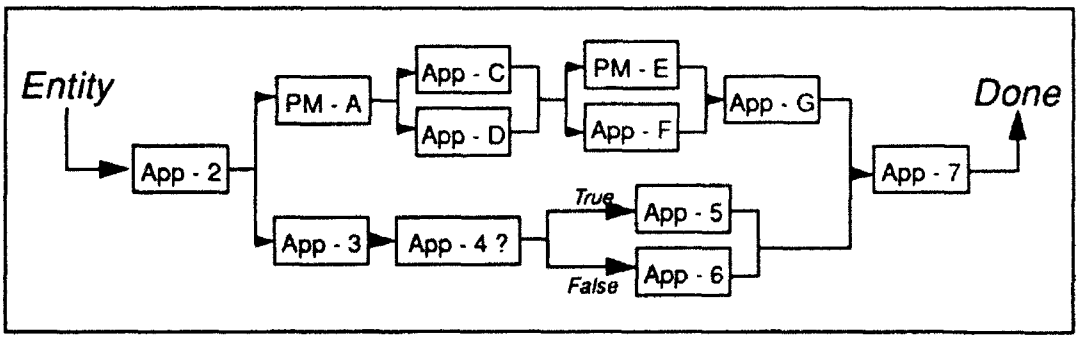


Figure 2.4. An example processing sequence managed by the Process Manager. Under the supervision of the Process Manager, an entity is passed between applications as indicated by the arrows.

waits for all dependents to complete their sequences). Thus, while each entity is processed independently, there are mechanisms to synchronize processing between hierarchically related entities. This design allows a divide and conquer approach to achieve an overall processing objective.

The Process Manager must assume some of the responsibility for scheduling the execution of applications in the session. The DACS Manager cannot schedule processing based on specific knowledge of any entity, yet processing a particular entity may require priority based on one of its attributes (e.g., its age). Furthermore, since the CommAgent maintains its message queues in a volatile form, there are limits on the message traffic that can be handled. Its overall reliability and performance is compromised if too many messages are sent simultaneously. With more knowledge of the logical processing objectives, the Process Manager can impose ordering and assign priority to ensure that overall processing is expedited. To allow configurable scheduling, the Process Manager maintains its own processing queues that can be reordered as required. By maintaining its own queues, the Process Manager also strictly limits the number of messages passed through the DACS and avoids stress on the IPC infrastructure.

The Process Manager handles conditions raised by logical processing errors. An application that encounters a logical error while processing an entity returns a status to the Process Manager. The Process Manager interprets the status to decide what action to take for that entity: continue processing the entity through the sequence, reschedule the application, branch to another instruction, or wait for operator intervention. The implementation of a particular error-handling policy is configurable according to the session-specific processing requirements.

The Process Manager maintains a log (e.g., a disk file) of its current state, which includes the status of each entity. In the event of a Process Manager failure, the last logged state may be used to recover. Any applications that are actively processing entities are also assumed to have failed when a Process Manager fails; those entities will be rescheduled

for those applications upon recovery. These recovery rules assume that any entity can be processed through the same application any number of times without affecting the final result. User applications controlled by the Process Manager must comply with this assumption to avoid inconsistencies in the processing results.

To exploit the Process Manager, many user applications and system commands must be integrated with the DACS. This is a simple but tedious task, since there are many useful Specialist applications. Furthermore, some applications from external developers are either difficult or impossible to interface directly with the DACS IPC mechanisms. Instead, these applications require command-line parameters that reference or otherwise depend on the entity being processed. To facilitate integration of these applications, the *SIP* includes a utility, the PMshell, to act as a proxy for a Specialist application. The PMshell receives an IPC message from the Process Manager, translates the enclosed entity into required command line arguments, executes its assigned application, and responds to the Process Manager when that execution is complete. Thus, the Process Manager has access to many applications and system commands to meet its processing objectives.

2.3 Data Management System Interface

The *IMS* applications are a collection of heterogenous software components that share common entities (i.e., data) through a distributed globally accessible data-management system. Thus, a key element of all *IMS* applications is a common data model based on well-described entities (e.g., Anderson, *et al.* 1990, Swanger, *et al.* 1991). In order to integrate diverse applications and data archives, the applications must be insulated from the physical details of how the data model is implemented in each archive. Ideally, an application should access entities through an interface that requires only an abstract description of the entity. That is, any entity represented by the data model should be accessible through an interface that is independent of any archive-specific hardware or software component. Another key requirement for *IMS* is that the data model itself must be dynamic. *IMS* is a testbed for new research developments; it must be anticipated that new entities will be defined and incorporated into the data model as new knowledge is acquired.

Most of the entities used by *IMS* applications can be represented in a relational data model, allowing the use of reliable commercially available Relational Database Management Systems (RDBMS) to manage the physical data archive. The relational model is a powerful abstraction that permits entities and their relationships to be succinctly specified and accessed using a query language (e.g., the Structured Query Language, SQL). An RDBMS was selected over competing alternatives because it supported the most flexible way of organizing data that satisfied both the requirements of routine data processing and knowledge acquisition. Knowledge acquisition, in particular, is enhanced by the flexibility afforded by SQL. Commercial RDBMS also provide reliable transaction management and concurrent data access that is crucial to the development of an automated, distributed, near-real-time system.

It has been impractical to extend the relational model and RDBMS products to incorporate all data shared by the *IMS* applications. Some entities (time-series, images, a algorithm-specific parameters) must be managed through the file system. Some *IMS* s ware is used in processing environments that do not provide access to an RDBMS. A a centralized, monolithic, data-management system, as typified by current RDBMS products, is an impediment to realizing some of the advantages of distributed data processing. Thus, our applications require a Data Management System Interface to entities stored in one or more RDBMS as well as data in local and remote file systems and other archives.

The design and development of the Data Management System Interface (DMSI) has been guided by emerging methods and technologies (e.g., relational database management systems, object-oriented programming methods, object-oriented database management systems) while stressing the scientific objective of actually building and operating *IMS*. It ultimately will provide a uniform application interface to all external entities that is independent of the detailed structure and implementation of the data archive. However, the DMSI will still allow applications to exploit specific features of the data archive. For example, if a relational model is adopted, then the DMSI will allow applications to take advantage of SQL in accessing data. Some applications will not be able to function without access to an RDBMS. But the interface should not explicitly depend on SQL and, instead, treat query-language constructs as configurable arguments. In that way individual applications may be easily reconfigured to access archives managed by both RDBMS and non-RDBMS, if appropriate.

The DMSI provides the following benefits to *IMS* software:

- The effects of modifying the data model and the physical data archive are isolated from application code. That is, the application software need not change if a different RDBMS product is used was chosen, or the underlying data model is expanded. Eccentricities in a particular database management system are handled in the DMSI support libraries, not by each application.

- The uniform DMSI interface improves developer productivity by providing reusable software modules to handle data access, which is often a large element of an application program. All applications are subject to consistent programming standards with respect to the data-access interface, which enhances quality control. Also, consistent source code across all software products is easier to maintain.

- The software for data access is produced in a uniform way by a small group of specialized developers, improving its consistency and quality.

- The effective use of emerging computer-aided software engineering (CASE) tools quires an underlying structure to the basic software components. These tools promise remove muc f the tedium (and cost) from the assembly of complex software products if the assembly process can be sufficient abstracted. The DMSI is designed to impose such a structure on the data-acc ss functions.

During the *IMS* development, the DMSI has evolved along several paths resulting in the three distinct products shown in Figure 2.5. The earliest version consists of an interface library to a pre-defined set of entities described by the Center for Seismic Studies Version 3.0 database schema plus *IMS* extensions (Anderson *et al.*, 1990, Swanger *et al.*, 1991) and managed in an Oracle RDBMS. The next version provides an RDBMS interface that is independent of the underlying database schema and uses the full power of SQL to define and select entities. This interface, called the *Generic Database Interface* (GDI), is built on a support library that simplifies transition from Oracle to other RDBMS products. Neither the CSS3.0/IMS interface nor the GDI support access to entities that are not managed in an RDBMS. The *Dynamic Object Interface* (DOI) provides a single application-program interface for all data access, including data managed in various database management systems and the UNIX file system. The DOI provides the additional capability to define, construct, and manipulate entities during application execution. This simplifies the development of general software components that are applicable to a wide range of specific data analysis tasks. These three stages of DMSI evolution are described in the following sections.

2.3.1 CSS3.0/IMS Interface.

The *CSS3.0/IMS Interface* provides access to the entities described in the CSS3.0/*IMS* schema and maintained in an RDBMS. Only those entities that exist as tables documented in the schema are accessible through the interface. Although the application-program interface is independent of any vendor-specific RDBMS, only access to an Oracle RDBMS is supported in *IMS* applications[1]. The CSS3.0/IMS interface consists of a library of functions that invoke data selection criteria specified through SQL constructs. The following discusses the important characteristics of the CSS3.0/IMS interface; it is described in detail in the Appendix.

For each entity, the access functions for data retrieval and insertion are similarly structured, which allows them to be generated automatically from the entity's definition. Therefore, the interface libraries are easy to build and maintain[2]. When the database schema is extended or modified, access functions are easily added or modified. Since the interface functions are so similar for each entity, there is uniformity in the data-access code used in the various software modules, which improves maintenance. Finally, the interface is simple and developers are do not need to involve themselves with the specific details of an RDBMS product. This greatly simplifies the integration of software from external developers.

---

1. This is only true to the extent that SQL itself is RDBMS product independent. Further, RDBMS products deal with the details of transaction management in very different ways, so that true isolation from a specific RDBMS is very difficult to achieve.
2. *IMS* was developed with versions of RDBMS products that had not been widely distributed and tested in an operational environment of similar complexity. Our design allowed us to successfully develop a large, complex software system while working with a new (and thus, changing) RDBMS product in its initial stages of release.

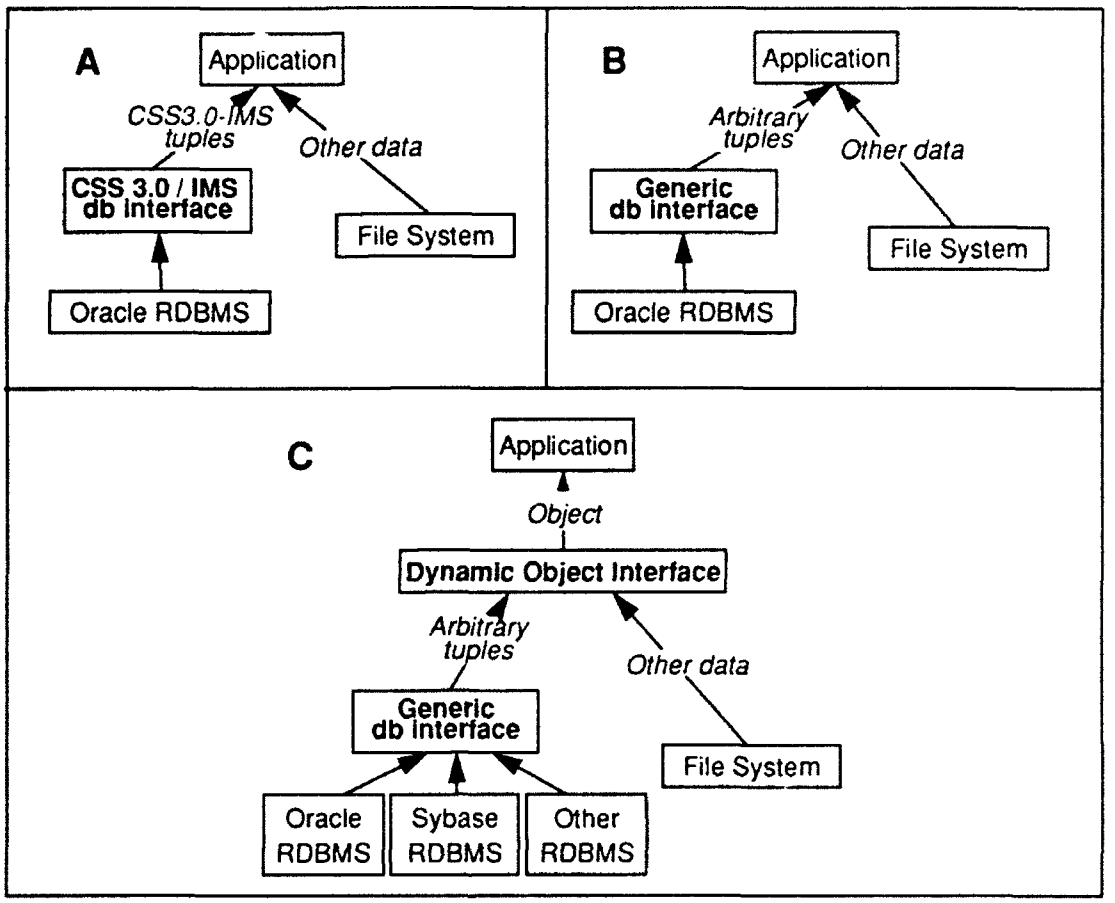


Figure 2.5 The evolution of the DMSI during the development of the *IMS*. A: the CSS3.0/IMS database interface provides access to an Oracle CSS3.0/IMS RDBMS; applications access other data from the file system through application-specific interfaces. B: GDI extends the RDBMS interface to an arbitrary schema, but does not address non-RDBMS data. C: DOI unifies the RDBMS and the non-RDBMS data-access interfaces with the GDI providing access to data managed in an RDBMS. New support libraries allow access to additional RDBMS products.

Although the CSS3.0/IMS interface has many productivity and maintenance advantages, is has some important limitations. In particular, the database libraries are restricted to entities enumerated in the CSS 3.0 and *IMS* schema. Therefore, complex entities that are formed by relational operations on the fundamental entities cannot be accessed through the libraries. These entities must be constructed within individual applications, which sacrifices some of the advantages of a relational database. Thus, software developed with this interface has an explicit dependence on the CSS3.0 and *IMS* schema. Finally, uniform access is not extended to those entities that cannot be represented relationally, that cannot be practically maintained in a relational database, or that require access to several independent RDBMS or data archives. In the *IMS*, the most important example is time-

series (i.e., waveform) data. Although the CSS schema provides a relational mechanism for managing references to these data, the actual entities are accessed in ways incompatible with a relational data model. Furthermore, efficient storage and access of large numbers of these entities precludes using the CSS3.0 and *IMS* schema at all.

2.3.2 The Generic Database Interface.

The GDI is a generic, schema-independent, vendor-independent interface to an RDBMS. Through this interface, an application can take full advantage of a relational data model (and the SQL) to define entities based on an arbitrary database schema. The GDI also simplifies the adaptation of applications to other RDBMS products by further isolating and minimizing the library support modules that depend on vendor-specific components.

In the CSS3.0/IMS interface, knowledge about each accessible entity is explicit. If a new entity is required, a new set of library modules to access that entity must be developed and added to the library. In other words, the set of accessible entities is enumerable and restricted to a subset of all entities that could be derived by relational operations on the underlying data model. The GDI relaxes this restriction and allows access to any entity defined by a relational query without modification to the underlying support library. With the GDI, an application accesses an entity by specifying a complete and valid SQL query. The query both defines the entity and constrains the selection of specific instances. This feature decouples an application from the underlying database schema since the complete entity-defining query is merely a parameter that can be easily configured to a particular schema without modification to the GDI. The only constraint is, obviously, that the requested entity be derived from relational operations on the underlying database schema.

The GDI significantly improves software development productivity and product maintainability. However, it does not address access to entities that are not managed in an RDBMS. Also, it does not provide any interface to the overall heterogeneous data-management system. In the *IMS*, for example, these required entities must be accessed on an *ad hoc* basis that depends on the detailed structure of the specific archive. For each of these entities, a significant effort has been devoted to systematically building interfaces to these entities. It remains, however, to provide the user applications access to these entities through a single, consistent interface. The Dynamic Object Interface provides that unified interface along with other important (and related) functionality.

2.3.3 The Dynamic Object Interface

The Dynamic Object Interface (DOI) provides a uniform object-oriented approach to retrieving, manipulating, and accessing the entities maintained in the Data Management System. The DOI is under development and is ultimately intended to be the Data Management System Interface for *IMS* applications.

In the DOI, an entity is generalized to an object, which includes both *methods* (i.e., functions for manipulation) and *attributes* (i.e., data or other objects). These objects are encapsulated in a form that allows an application to define, interpret and manipulate them through commands that are configurable during runtime. Through the DOI, an application builds its required objects from primitive objects and entities available from a particular data archive. The manipulations necessary to build these objects, which usually depend on the details of the archive, are configurable through an extension language interpreted by the application at runtime. If data access is through DOI, an application can be reconfigured to access a new archive without the need to rebuild the application. With DOI, the separation between an application and its data access is complete, allowing for easier maintenance and wider applicability of software components. Furthermore, with such an interface it is possible to take full advantage of a client/server architecture to partition the data access and the application into separate physical processes. The final executable application is then completely independent of a data archive.

DOI also provides a flexible interface to dynamically define, construct and manipulate objects through interactive user commands. This functionality is important for developing general-purpose applications to be adapted to specific analysis tasks. An *IMS* example is the Map program, which manages the display of user-defined objects on digital maps and images. Through DOI, users define and construct very general objects that are linked to geographical location. The display of these objects is specified by user-defined methods that interpret the user-specified attributes of the object. Thus, the Map program has no explicit knowledge of the objects it is displaying, and can be applied to many different geographic visualization problems.

A fundamental objective of the *SIP* is the integration of diverse and disparate applications and data archives into a single system. Achieving this goal requires that we deal with ubiquitous problems in converting entities and reformatting data. DOI provides a systematic solution to these problems.

Figure 2.6 shows the typical data management problems encountered in software integration. In #1, there are two sets of applications, *A-[1...n]* and *B-[1...n]*, that require, respectively, entities e_A and e_B from archives *DataServer-A* and *DataServer-B*. The *A* and *B* applications are well designed software modules that access their respective entities through remote procedures in a client-server architecture. Consider a scenario in which entities e_A and e_B are similar enough to be procedurally (although perhaps incompletely) derived from each other: $e_A = f(e_B)$, $e_B = g(e_A)$. Further assume that the *A* and *B* applications have complementary functionality and are all required in an integrated analysis system. Both sets of applications require access to both data archives. The design issue is to determine the best place to perform the mapping between entities anticipating that a similar situation will be encountered when expanding to other archives and applications.

One solution (#2 in Figure 2.6) is to modify each program to recognize the new entities and make the required conversions. This solution requires each application to be modified, rebuilt, and redistributed to users. This procedure eliminates many of the advantages of the client-server architecture. Another solution (#3) is to modify the entity-access

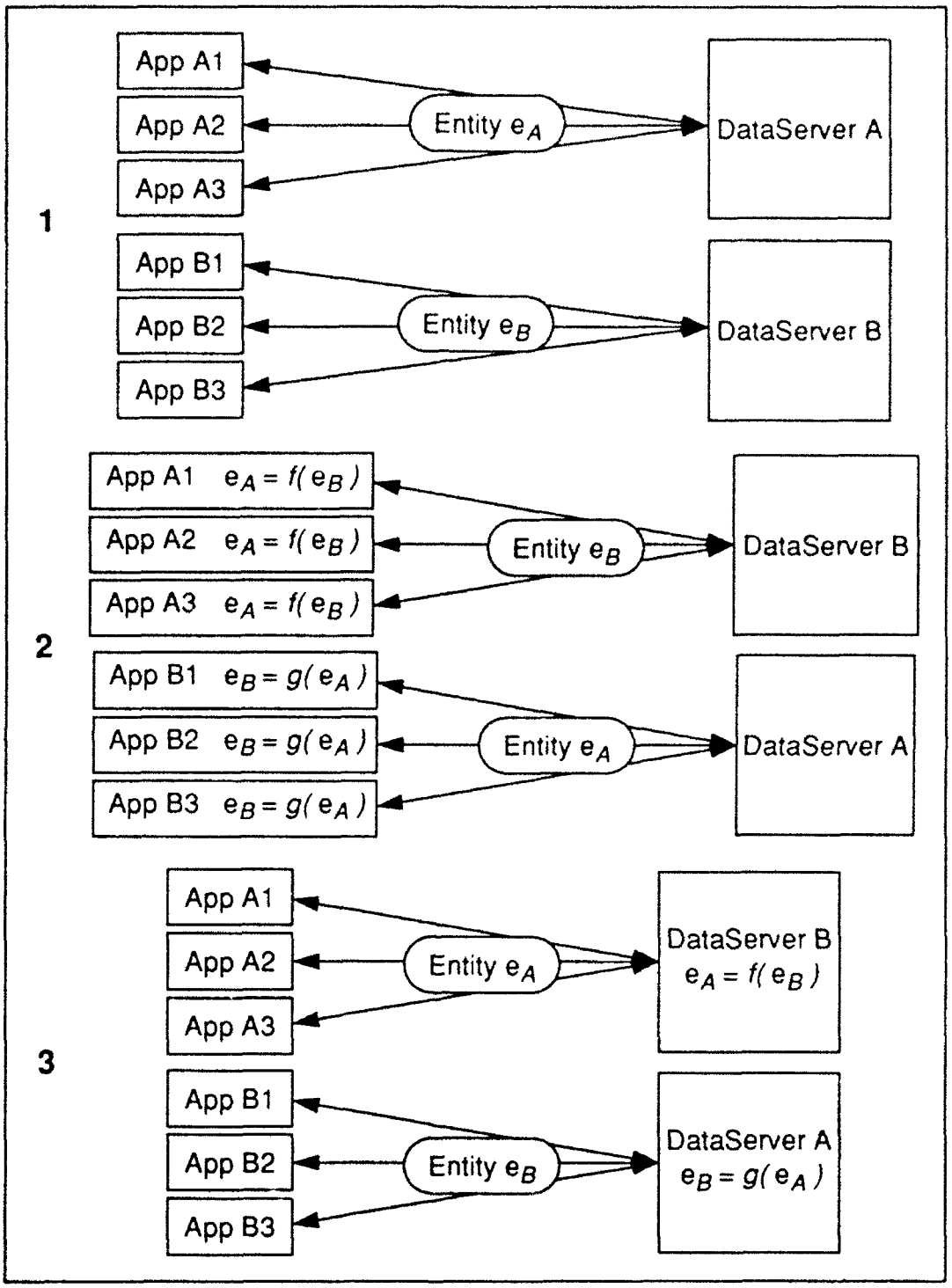


Figure 2.6. The different approaches to adapting applications to access entities from different data archives in a client/server architecture. *1:* independent applications and archives; *2:* applications are adapted by data transformations within the application; *3:* archives are adapted by transformations within the archive management software.

libraries for each archive to convert between the specific entities. This seems like a better solution since there are fewer archives than applications. However, in an environment of heterogenous databases distributed among different operations, this alternative frequently requires more inter-institutional cooperation than can normally be mustered.

A better approach combines aspects of both solutions and is shown at the top of Figure 2.7. In this approach, data conversion is handled in two steps. The entities are transformed into a standard form in the database server. In our example, that standard form is obtained using: $e_{AB} = U(e_B)$ or $e_{AB} = V(e_B)$. The standard entity is transformed back to the application-specific (or site-specific) form in the client using $e_B = V^*(e_{AB})$ or $e_A = U^*(e_{AB})$. Standardization is an important step, but real entities defy complete description by an enumerable set of standard attributes. Standards and, therefore, software must be expected to evolve. This is especially true in scientific data management in which a fundamental objective is to describe entities in ever increasing detail.

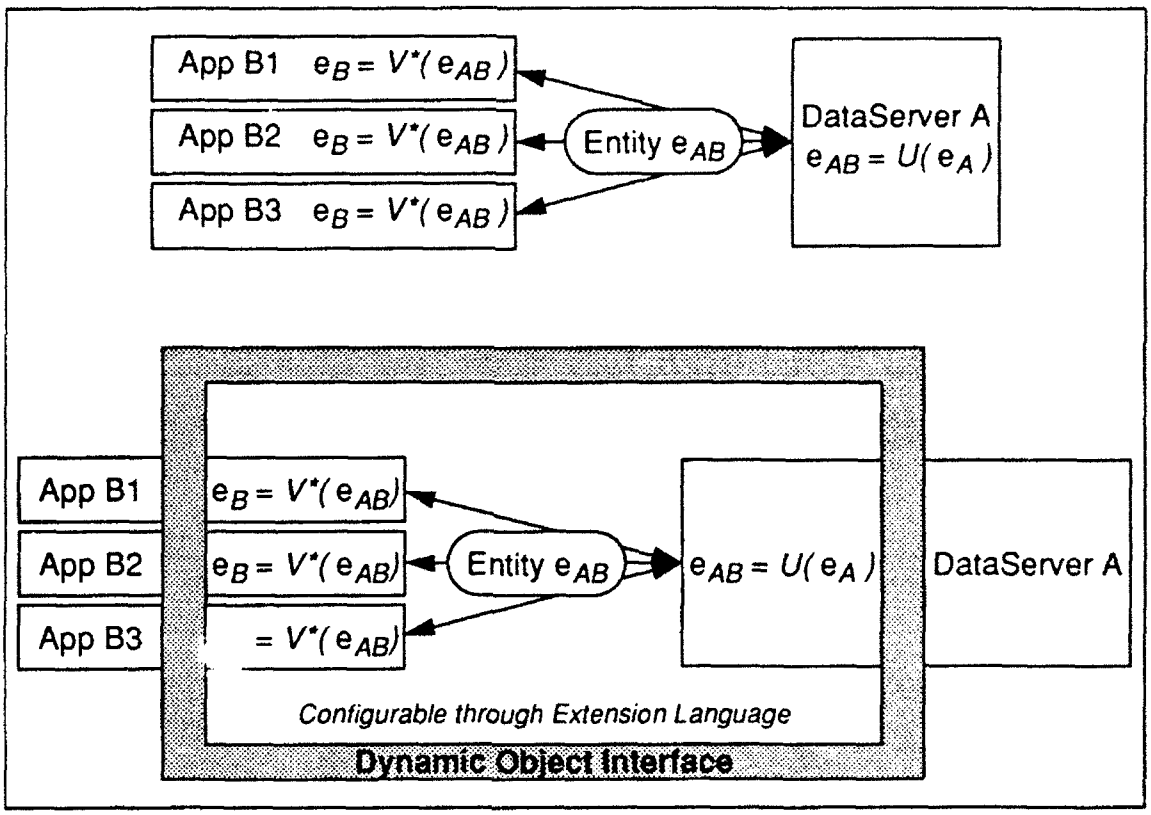


Figure 2.7 The inter-operability of applications is facilitated by agreeing on a standard form for all entities. With the DOI the necessary transformations may be configured at runtime in both the client and server.

The DOI solution is shown at the bottom of Figure 2.7. With the DOI, each application accesses its required entities through a general interface that is configured at runtime to handle the conversions between the entities. Entities e_A and e_B are encapsulated and each application accesses them through the DOI. The conversions required by each application are specified at runtime, which reduces the need to rebuild applications and data-

base servers. The servers, *DataServer-A* and *DataServer-B*, independently package the entities, e_A and e_B, into objects that are provided to the requesting applications. Since the servers must package the objects into a form interpretable through DOI, the server can use DOI to manipulate the objects into a standard form. In this way, the configurability is extended to the server as well as the client. Thus, DOI simplifies the development, maintenance, and extensibility of both client applications and data servers.

The DOI interprets a rich command (i.e., programming) language at runtime. The command language has access to primitive archive-specific data-access functions in order to construct primitive objects. These basic objects are subsequently manipulated through DOI for inclusion into more complex objects. The DOI command language is Scheme, which has a Lisp-like syntax and structure. Scheme has a standard definition (IEEE Std. 1178-1990, Dybvig, 1987) and is a widely used extension language in the software-development community. A Scheme interpreter can be implemented in a very compact form that is easily embedded in applications written in more widely used languages (e.g., Fortran, C, C++). Several suitable versions are available in the public domain.

Both DOI and GDI provide encapsulation of entities and an abstract data-access interface. DOI extends access to more general data types and provides mechanisms to manipulate the object. Thus, GDI and DOI together provide powerful capabilities to access diverse data archives. The DOI encapsulates an object so that it can be defined and manipulated at runtime. The details of accessing a specific physical data archive must be addressed on a case by case basis, but with GDI, the data-access functions are provided for those archives managed by an RDBMS.

We illustrate DOI with a simple *IMS* example. Over a dozen *IMS* applications require an entity (here called *seismic event)* with attributes that specify the hypocentral parameters (the *origin*: geographical coordinates, time, depth, magnitude) and attributes that describe observations of signals from the event (the *arrivals*). The *seismic event* entity is widely recognized in seismology and these *IMS* applications may be applied to any data archive that supports the entity. Also, there are many non-*IMS* applications that recognize *seismic event*. The *IMS* data archives are organized in a relational model in which the *seismic event* object consists of three fundamental entities: *origin*, *arrival*, and *assoc*, which relates *origin* and *arrival*. Non-*IMS* data archives are organized differently and may, for example, manage origin and arrivals as one entity. In either case, the user (i.e., client) applications in question should be independent of such details of the data-management systems.

Figure 2.8 shows the interfaces and the sequence of procedures through which a client application retrieves a *seismic event* entity through the DOI. The client requests a *seismic event* object using its class name, "Seismic Event", and identifiers and parameters needed to identify and construct it. For example, *seismic event* might be specified by *bulletin*, which identifies the organization or operation that created it, and an *event-id*, which uniquely identifies it in *bulletin*. The client application retrieves *seismic event* by invoking a function (labelled 2.0 in Figure 2.8) providing the class name and identifiers as parameters. The function will return the object, *seismic event*. At this point, *seismic event* is not

yet in data structures managed by the client. The client queries the object (3.0) to move the attributes into the structures it needs. These data-access functions (1.0, 2.0 and 3.0) make up the *application-program interface* of DOI. The mechanics of retrieving the object are configurable and depend on where and how the data is physically stored. These details are part of the data-management system and isolated from the client application by the DOI.

Below the application-program interface of DOI in the client, the object must first be located (labelled 2.1 in Figure 2.8). To perform this task, the client relates *bulletin* and *event-id* to a server, a server object (or objects), and server-specific parameters. For example, *bulletin* may map to a remote database and the *event-id* may map to physical identifiers (i.e., hypocentral coordinates). After the object is located, the client sends a request to the server and awaits a reply (2.2). Upon reply, the client converts the server object into a client object (2.3). The details of constructing the client object depends on the server providing the object. The instructions for locating and constructing the client object are specified to the DOI in Scheme through the *configurable interface*. These instructions are loaded into the application at runtime for interpretation by DOI. For example, depending on the data archive, the client may need to modify the server object to rename attributes, change units of measure, or transform coordinate systems. In more complex situations, a client may require multiple server objects from different servers to construct its object.

The server translates the request for *seismic event* into a request for primitive objects (labelled 2.2.1 in Figure 2.8). It invokes archive-specific functions to create these objects, access the physical archive, and insert the resulting entities into these objects (2.2.2). As shown in the figure, the server invokes the archive-specific functions as instructed through the configurable interface of DOI. In our example, the archive-specific functions query the RDBMS (2.2.2.1) through either the GDI or the CSS3.0/IMS interface for the *origin*, *assoc*, and *arrival* entities that will be used to construct *seismic event*. The archive-specific functions then use the DOI application-program interface (procedure 2.2.2.2) to construct primitive objects. The server subsequently manipulates these primitive objects into the requested server objects as specified through the configurable interface (procedure 2.2.3) and sends the object to the client.

The previous example illustrates several important software-design principles: modularization, encapsulation, and isolation. User applications are often complex interactive programs for analysis, visualization, and decision making. DOI isolates these applications from the application-independent details of accessing a specific physical data store. The DOI consists of a powerful configurable interface separating the application from the data archive. With DOI, data access can be divided into several layers, each containing specific knowledge about those objects it can access and those it must produce. Complex client applications are easier to maintain, since the data-access layer can be independently developed, tested, and modified for a new data archive. Finally, DOI can exploit the Distributed Application Control System to create user objects by invoking Specialist applications as servers. Again, all of the necessary complexity introduced by accessing the DACS is completely isolated from the user application.

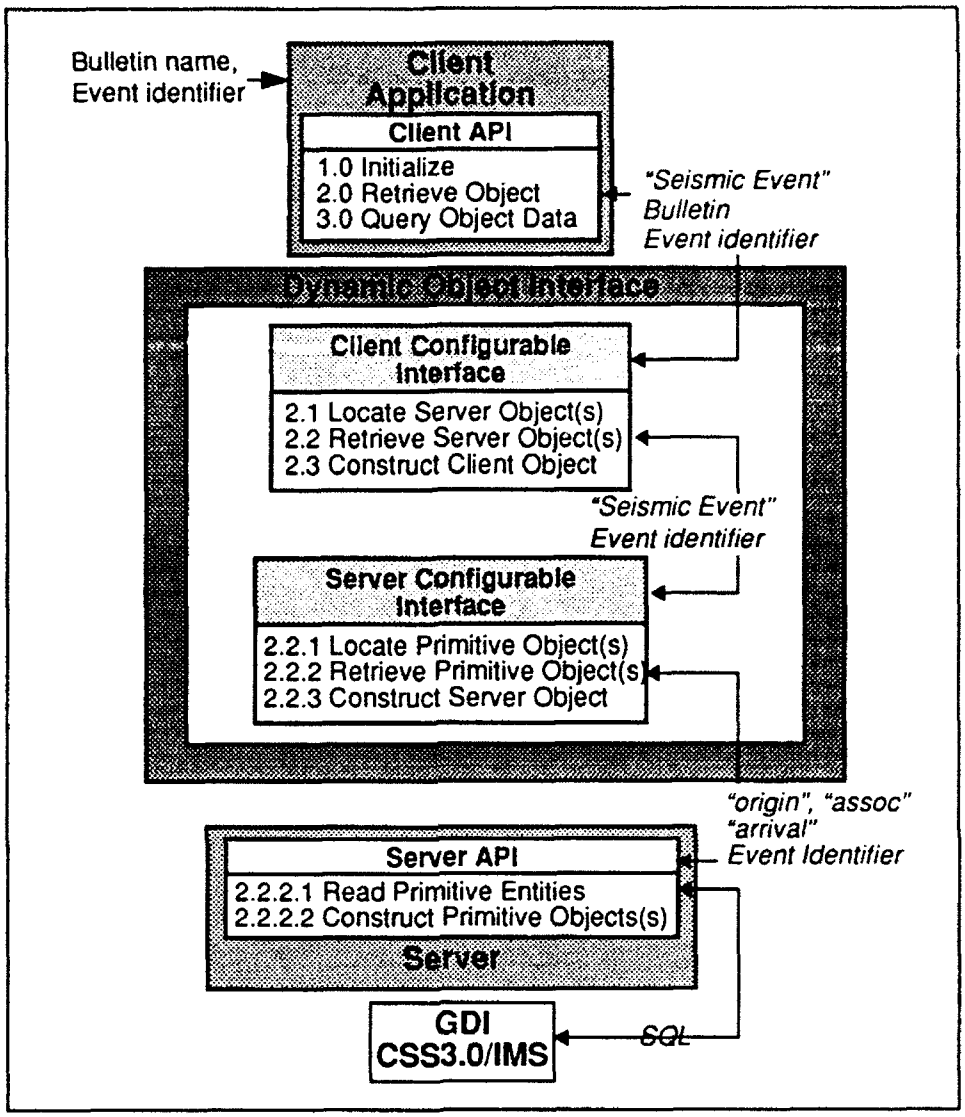


Figure 2.8 An *IMS* example of data access through DOI. The client user application and the server data access is through a similar application-program interface. The configurable interfaces control those details that depend on a specific physical data archive.

III. SUMMARY

The *Software Integration Platform* provides a general framework for interconnecting UNIX processes to cooperate on large data-driven processing and analysis tasks. Although, it was designed to support *IMS*, the *IMS*-specific requirements have led to general solutions that are widely applicable in both automated and interactive scientific data-processing systems. The *SIP* explicitly addresses the problem of integrating diverse software components and data archives that are distributed over a geographically dispersed wide-area network of UNIX workstations. It further addresses the practical problems of connecting existing applications to existing data-management systems. Indeed, it is explicitly designed to take advantage of and co-exist with external software subsystems.

The overall system supported by the *SIP* is data driven. That is, the state of the system is the response to entities created by user applications monitoring the external environment. These applications detect external events and create new entities that describe and identify those events. However, these entities are distributed in heterogeneous data archives, which precludes the close global coordination of data and process control. Therefore, the *SIP* is designed around widely distributed loosely coordinated sessions of locally distributed closely coordinated applications.

There are two distinct subsystems in the *SIP*: the Data Management System and the Distributed Application Control System. The Data Management System provides applications access to entities (i.e. data) distributed over a wide-area network. The DACS provides inter-process communication and control in a session made up of a group of applications distributed over a local-area network. In a system like *IMS*, sessions poll the Data Management System for results from other sessions operating in the wide-area network and initiate complex sequences of applications in response.

The Data Management System Interface (DMSI) provides uniform access to entities that are described by a data model common to all user applications. The physical data archives consist of commercial relational database management systems (RDBMS), UNIX file systems, and other media. The DMSI isolates the user applications from any specific physical archive. Early versions of the DMSI consist specifically of interfaces to commercial RDBMS products. The CSS3.0/IMS interface provides access to entities described by the CSS3.0/IMS database schema. The Generic Database Interface is schema independent. Under development, the Dynamic Object Interface (DOI) is an object-oriented interface to all entities in the Data Management System, including those not managed in an RDBMS. However, an RDBMS interface will continue to be an important component of the Data Management System.

The DACS is further divided into two applications: the CommAgent, which handles the mechanics of IPC, and the DACS Manager, which handles the details of managing specific physical processes. The CommAgent is responsible for message routing and message queueing in a session and is the interface to the subsystem that provides physical message transport. The DACS Manager is responsible for deciding when and where to

start applications on a specific configuration of workstations with limited physical resources. It provides the physical interface to the specific workstations and encapsulates all network-specific information. The DACS Manager also provides a user interface to allow an operator to monitor and manipulate the physical state of the session.

The Process Manager is a general application that exploits pipelining and concurrency to expedite processing. It simultaneously tracks the processing of many instances of the same type of entity through a complex sequence of applications. The Process Manager provides a simple interface through which to configure complex processing sequences and to make decisions based on the logical processing state.

The *SIP* is designed to take advantage of currently available commercial software and anticipate trends in product development. For example, the GDI is explicitly designed to simplify extending the DMSI to allow access new RDBMS products. The current version of the CommAgent is built on the ISIS Distributed Toolkit™ to satisfy stringent fault-tolerance requirements. These can be relaxed in some circumstances and the CommAgent can be built on any messaging subsystem that supports reliable remote-procedure calls. The DACS Manager is configurable to any UNIX workstation operating system. Thus, the *Software Integration Platform* is not a replacement for commercial software products but instead provides a framework that allows them to work together.

IV. DEFINITIONS, ACRONYMS AND ABBREVIATIONS

CSS	Center for Seismic Studies.
CSS3.0/IMS schema	The database schema describing the entities shared by IMS applications in a relational database.
CSS3.0/IMS interface	An application-program interface to the entities represented in the CSS3.0/IMS schema.
CommAgent	The component of the DACS that routes messages between other applications.
DACS	Distributed Application Control System. A set of computer programs that provide interprocess communication and control.
DACS Manager	The component of the DACS that controls starting and stopping applications.
DMS	Data Management System; the collection of data archives, including RDBMS products and UNIX file systems, that store the data shared by SIP applications.
DMSI	Data Management System Interface; the application interface to the Data Management System.
GDI	Generic Database Interface; a schema-independent application-program interface to an RDBMS.
IMS	Intelligent Monitoring System; a seismic monitoring system.
IPC	Inter-process communication; communication of discrete messages between independent computer processes.
ISIS	An toolkit that provides fault-tolerant inter-process communication.
LAN	Local-area network; a collection of computers linked together.
RDBMS	Relational Database Management System.
SIP	Software Integration Platform. A collection of applications and interfaces that allow other applications to work together in a distributed environment.
WAN	Wide-area network; a collection of local-area networks linked together.

V. REFERENCES

Anderson, J., W. E. Farrell, K. Garcia, J. W. Given, and H. J. Swanger (1990) The Center for Seismic Studies Version 3 Database Schema Reference Manual, *Tech Rep. SAIC-C90-01*.

Anderson, J, M. Mortell, and B. MacRitchie (1993). The Generic Database Interface (GDI) User's Manual, *Tech. Rep SAIC-93/1003*.

Bache, T., J. T. Anderson, D. Baumgardt, S. R. Bratt, W. E. Farrell, R. F. Fung, J. W. Given, A. S. Henson, J. C. Kobryn, H. J. Swanger, and J. Wang (1990a). The Intelligent Array System, *Final Rep. SAIC-90/1437*.

Bache, T., S. R. Bratt, J. Wang, R. Fung, C. Kobryn, and J. W. Given (1990b). The Intelligent Monitoring System, *Bull. Seismol. Soc. Am., 80*, 1833-1851.

Bache, T., S. R. Bratt, J. W. Given, T. Schroeder, H. Swanger (1991). The Intelligent Monitoring System Version 2, *Tech. Rep. SAIC-91/1137*.

Birman, K. P., R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood (1990). The ISIS System Manual V2.0. 411 pp.

Bratt, S. R. (1992). GSETT-2: An experiment in rapid exchange and interpretation of seismic data, *Eos Trans. AGU*, 73, 520.

Dybvig, R. Kent (1987). *The Scheme Programming Language*, Prentice Hall, Englewood Cliffs, NJ

Fox, Warren K (1993) The DACS Manager Program, *Tech. Rep. SAIC-93/1012*.

Given, J. W, Howard Turner, and Jerry Jackson (1993). The CommAgent Program, *Tech. Rep. SAIC-93/1013*.

IEEE Std. 1178-1990 (1991) *IEEE Standard for the Scheme Programming Language*, New York, NY

Swanger, H. J., J. W. Given, and J. Anderson (1991). IMS Extensions to the Center for Seismic Studies Version 3 Database Schema, *Tech. Rep. SAIC-91/1138*.

APPENDIX

THE CSS3.0/IMS Database Interface

The CSS3.0/IMS Database Interface provides access to CSS3.0/IMS entities represented in the Center for Seismic Studies Version 3.0 database schema and the extensions for the *IMS*. Reports by Anderson *et al.* (1990) and Swanger *et al.* (1991) describe these entities. The CSS3.0/IMS entities correspond explicitly to rows from tables in a relational data model, which is implemented in *IMS* using an Oracle RDBMS. However, the entities and interface definition are independent of the underlying RDBMS product. With some limitations and caveats, developing a consistent file-based interface is possible. Thus, external developers without access to an RDBMS can build *IMS* applications for that are easily converted to run in the *IMS* operational framework where most data access is via an RDBMS.

The CSS3.0/IMS Interface consists of general functions to connect and disconnect to an RDBMS, manage transactions, report errors, execute SQL commands, and read and write CSS3.0/IMS entities. The functions provide a simple application-program interface for applications written in both the Fortran and C programming languages. Wide-area access to a network of RDBMS is transparent to the interface, and user applications need take no special measures to access a remote RDBMS.

The read and write access for CSS3.0/IMS entities is especially simple. For each entity described in the CSS3.0/IMS database schema, there are specific C and Fortran data structures. To read a specific entity, an application invokes a function that returns an array of those structures. To write an array of those structures containing a specific type of entity, the application passes the array to a function that inserts the entities into the corresponding table. The interface has limitations. Specifically, derived entities that are defined by joins between CSS3.0/IMS entities are not accessible through the interface. Nevertheless, the simple interface is compatible with application development in the absence of an RDBMS. If external developers follow the overall structure of the CSS3.0/IMS interface and use similar data structures, converting an application from file to RDBMS data access will involve minimal modifications. Similar guidelines apply to IMS applications that must operate in environments without an RDBMS.

The CSS3.0/IMS interface uses SQL to select specific entities for reading. Therefore, while the types of entities retrieved are limited to those described in the schema, an application can take full advantage of SQL to select specific entities it requires. Thus, the entity-selection criteria can directly use relationships between different types of entities. Usually, construction of more complex entities within each application is not difficult. In large systems, this is often a more efficient strategy, since the operational RDBMS resources are always limited and often heavily used.

The following UNIX manual pages describes the specific details of the CSS3.0/IMS interface. We document two libraries: **libdb30**, which consists of functions for RDBMS access, transaction management, error reporting, and access to CSS3.0 entities; and **libdbims**, which provides functions for access to *IMS*-specific entities.

NAME

Database Communications:

dbopen, dbclose, dbwhoami, dbcancel

Transaction Management:

dbcommit, dbrollback

Key Counter Assignment:

dbgetcounter

String Handling:

cstr_to_pad, pad_to_cstr

Error Handling:

dberror_app, dberror_get, dberror_init, dberror_sqlca, dberror_unix

Fetch Routines:

get_affiliatio, get_arrival, get_assoc, get_event, get_gregion, get_instrument, get_netmag, get_network, get_origerr, get_origin, get_remark, get_sensor, get_site, get_sitechan, get_sregion, get_stamag, get_stassoc, get_wfdisc, get_wftag, get_wftape

Insert Routines:

affiliati_Aadd, arrival_Aadd, assoc_Aadd, event_Aadd, gregion_Aadd, instrumen_Aadd, netmag_Aadd, network_Aadd, origerr_Aadd, origin_Aadd, sensor_Aadd, site_Aadd, sitechan_Aadd, sregion_Aadd, stamag_Aadd, stassoc_Aadd, wfdisc_Aadd, wftag_Aadd, wftape_Aadd, qa_attrib

SYNOPSIS

CALLS FROM C APPLICATIONS:

include files

```
#include "dborac.h"
#include "dbsqlc.h"
#include "libdb30_defs.h"
```

Database Communications:

```
int dbopen (uid)
char *uid;

int dbclose()

int dbwhoami(result)
char *result;

int dbcancel()
```

Transaction Management:

```
int dbcommit()
int dbrollback();
```

Counters (see dbgetcounter.3):

```
int dbgetcounter (ctype, increment, value)
char *ctype;
int increment *value;
```

String Manipulation:

```
int cstr_to_pad (array, string, array_length)
char *array, *string;
int array_length;

int pad_to_cstr (string, array, string_length, array_length)
char *string, *array;
int string_length, array_length;
```

Error Handling (see dberror.3):

```
int
dberror_init (print_flag, warn_flag)
int print_flag;
int warn_flag;

int
dberror_get (error_code, error_text, print_flag, warn_flag)
int *error_code;
char *error_text;
int *print_flag;
int *warn_flag;

int
dberror_sqlca (ptr_sqlca)
struct sqlca *ptr_sqlca;

int
dberror_app (error_code, ptr_buf)
int error_code;
char *ptr_buf;

int
dberror_unix (err_text)
char *err_text;
```

C Insert Routines (see array_insert.3):

```
#include "db_affiliation.h"
int affiliati_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct affiliation *ptr;
int     recnum;
int     qa_flag;

#include "db_arrival.h"
int arrival_Aadd(table, ptr, recnum, qa_flag)
char    *table;
```

```
struct arrival *ptr;
int    recnum;
int    qa_flag;

#include "db_assoc.h"
int assoc_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct assoc *ptr;
int    recnum;
int    qa_flag;

#include "db_event.h"
int event_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct event *ptr;
int    recnum;
int    qa_flag;

#define "db_gregion.h"
int gregion_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct gregion *ptr;
int    recnum;
int    qa_flag;

#include "db_instrument.h"
int instrumen_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct instrument *ptr;
int    recnum;
int    qa_flag;

#include "db_netmag.h"
int netmag_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct netmag *ptr;
int    recnum;
int    qa_flag;

#include "db_network.h"
int network_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct network *ptr;
int    recnum;
int    qa_flag;

#include "db_origerr.h"
int origerr_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct origerr *ptr;
int    recnum;
```

```
int    qa_flag;

#include "db_origin.h"
int origin_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct origin *ptr;
int    recnum;
int    qa_flag;

#include "db_remark.h"
int remark_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct remark *ptr;
int    recnum;
int    qa_flag;

#include "db_sensor.h"
int sensor_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct sensor *ptr;
int    recnum;
int    qa_flag;

#include "db_site.h"
int site_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct site *ptr;
int    recnum;
int    qa_flag;

#include "db_sitechan.h"
int sitechan_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct sitechan *ptr;
int    recnum;
int    qa_flag;

#include "db_sregion.h"
int sregion_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct sregion *ptr;
int    recnum;
int    qa_flag;

#include "db_stamag.h"
int stamag_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct stamag *ptr;
int    recnum;
int    qa_flag;
```

```
#include "db_stassoc.h"
int stassoc_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct stassoc *ptr;
int     recnum;
int     qa_flag;

#include "db_wfdisc.h"
int wfdisc_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct wfdisc *ptr;
int     recnum;
int     qa_flag;

#include "db_wftag.h"
int wftag_Aadd(table. ptr, recnum, qa_flag)
char    *table;
struct wftag *ptr;
int     recnum;
int     qa_flag;

#include "db_wftape.h"
int wftape_Aadd(table, ptr, recnum, qa_flag)
char    *table;
struct wftape *ptr;
int     recnum;
int     qa_flag;
```

C Fetch Routines (see array_fetch.3):

```
#include "db_affiliation.h"
int get_affiliatio(table_name, where_clause, affil_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct affiliation **affil_tuples;
int     maxrec;

#include "db_arrival.h"
int get_arrival(table_name, where_clause, arriv_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct arrival **arriv_tuples;
int     maxrec;

#include "db_assoc.h"
int get_assoc(table_name, where_clause, assoc_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct assoc **assoc_tuples;
int     maxrec;
```

```
#include "db_event.h"
int get_event(table_name, where_clause, event_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct event **event_tuples;
int     maxrec;

#include "db_gregion.h"
int get_gregion(table_name, where_clause, gregion_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct gregion **gregion_tuples;
int     maxrec;

#include "db_instrument.h"
int get_instrument(table_name, where_clause, instr_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct instrument **instr_tuples;
int     maxrec;

include "db_netmag.h"
int get_netmag(table_name, where_clause, netmag_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct netmag **netmag_tuples;
int     maxrec;

#include "db_network.h"
int get_network(table_name, where_clause, nwork_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct network **nwork_tuples;
int     maxrec;

#include "db_origerr.h"
int get_origerr(table_name, where_clause, origerr_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct origerr **origerr_tuples;
int     maxrec;

#include "db_origin.h"
int get_origin(table_name, where_clause, origin_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct origin **origin_tuples;
int     maxrec;
```

```
#include "db_remark.h"
int get_remark(table_name, where_clause, remark_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct remark **remark_tuples;
int     maxrec;

#include "db_sensor.h"
int get_sensor(table_name, where_clause, sens_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct sensor **sens_tuples;
int     maxrec;

#include "db_site.h"
int get_site(table_name, where_clause, site_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct site **site_tuples;
int     maxrec;

#include "db_sitechan.h"
int get_sitechan(table_name, where_clause, s_chan_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct sitechan **s_chan_tuples;
int     maxrec;

#include "db_sregion.h"
int get_sregion(table_name, where_clause, sreg_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct sregion **sreg_tuples;
int     maxrec;

#include "db_stamag.h"
int get_stamag(table_name, where_clause, stamag_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct stamag **stamag_tuples;
int     maxrec;

#include "db_stassoc.h"
int get_stassoc(table_name, where_clause, stassoc_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct stassoc **stassoc_tuples;
int     maxrec;
```

```
#include "db_wfdisc.h"
int get_wfdisc(table_name, where_clause, wfdisc_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct wfdisc **wfdisc_tuples;
int     maxrec;

#include "db_wftag.h"
int get_wftag(table_name, where_clause, wftag_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct wftag **wftag_tuples;
int     maxrec;

#include "db_wftape.h"
int get_wftape(table_name, where_clause, wftape_tuples, maxrec)
char    *table_name;
char    *where_clause;
struct wftape **wftape_tuples;
int     maxrec;
```

CALLS FROM FORTRAN APPLICATIONS:

include files

```
include '../../dboraf.h'
include '../../dbsqlf.h'
```

Connecting to the Database:

```
integer function dbopen(uid)
character*80 uid

integer function dbclose()

integer function dbwhoami(user_name)
character*80 user_name
```

Transaction Management:

```
integer function dbcommit()

integer function dbrollback();
```

Counters (see dbgetcounter.3):

```
integer function dbgetcounter(ctype, increment, value)
character*15 ctype
integer*4 increment
integer*4 value
```

Error Handling (see dberror.3):

```
integer function dberror_init(print_flag, warn_flag)
integer*4 print_flag
integer*4 warn_flag
```

```
integer function dberror_get(error_code, error_text, print_flag, warn_flag)
integer*4 error_code
character*70 error_text
integer*4 print_flag
integer*4 warn_flag

integer function dberror_sqlca(sqlcode, sqlwarn0, sqlwarn1, sqlwarn2,
sqlwarn3, sqlwarn4, sqlwarn5, sqlwarn6, sqlwarn7, sqlerrmc
integer*4 sqlcode
character*1 sqlwarn0
character*1 sqlwarn1
character*1 sqlwarn2
character*1 sqlwarn3
character*1 sqlwarn4
character*1 sqlwarn5
character*1 sqlwarn6
character*1 sqlwarn7
character*70 sqlerrmc

integer function dberror_app(error_code, err_msg)
integer*4 error_code
character*70 err_msg

integer function dberror_unix(err_text)
character*70 err_text
```

FORTRAN Insert Routines (see array_insert.3):

```
integer function affiliati_Aadd(table, recnum, qa_flag,
   x     net, sta, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*8  net(recnum)
character*6  sta(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function arrival_Aadd(table, recnum, qa_flag, sta,
   x     time, jdate, stassid, chanid, chan, iphase, stype, deltim, azimuth,
   x     delaz, slow, delslo, ema, rect, amp, per, logat, clip, fm, snr,
   x     qual, auth, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
real*8 time
integer*4 arid
integer*4 jdate
integer*4 stassid
integer*4 chanid
character*8  chan(recnum)
character*8  iphase(recnum)
character*1  stype(recnum)
real*4 deltim
```

```
real*4 azimuth
real*4 delaz
real*4 slow
real*4 delslo
real*4 ema
real*4 rect
real*4 amp
real*4 per
real*4 logat
character*1  clip(recnum)
character*2  fm(recnum)
real*4 snr
character*1  qual(recnum)
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function assoc_Aadd(table, recnum, qa_flag, arid, orid,
   x     sta, phase, belief, delta, seaz, esaz, timeres, timedef,
   x     azres, azdef, slores, slodef, emares, wgt, vmodel, commid,
   x     lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 arid
integer*4 orid
character*6  sta(recnum)
character*8  phase(recnum)
real*4 belief
real*4 delta
real*4 seaz
real*4 esaz
real*4 timeres
character*1  timedef(recnum)
real*4 azres
character*1  azdef(recnum)
real*4 slores
character*1  slodef(recnum)
real*4 emares
real*4 wgt
character*15  vmodel(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function event_Aadd(table, recnum, qa_flag, evid, evname,
   x     prefor, auth, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 evid
character*15  evname(recnum)
```

```
integer*4 prefor
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function gregion_Aadd(table, recnum, qa_flag, grn, grname,
   x     lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 grn
character*40  grname(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function instrumen_Aadd(table, recnum, qa_flag, inid, insname,
   x     instype, band, digital, samprate, ncalib, ncalper, dir,
   x     dfile, rsptype, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 inid
character*50  insname(recnum)
character*6  instype(recnum)
character*1  band(recnum)[1]
character*1  digital(recnum)
real*4 samprate
real*4 ncalib
real*4 ncalper
character*64  dir(recnum)
character*32  dfile(recnum)
character*6  rsptype(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function netmag_Aadd(table, recnum, qa_flag, magid, net,
   x     orid, evid, magtype, nsta, magnitude, uncertainty, auth,
   x     commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 magid
character*8  net(recnum)
integer*4 orid
integer*4 evid
character*6  magtype(recnum)
integer*4 nsta
real*4 magnitude
real*4 uncertainty
character*15  auth(recnum)
integer*4 commid
```

```
character*17  lddate(recnum)
integer*4 len_f_table

integer function network_Aadd(table, recnum, qa_flag, net, netname,
   x     nettype, auth, commid, lddate, len_f_table, len_f_query)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*8  net(recnum)
character*80  netname(recnum)
character*4  nettype(recnum)
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function origerr_Aadd(table, recnum, qa_flag, orid, sxx, syy,
   x     szz, stt, sxy, sxz, syz, stx, sty, stz, sdobs, smajax, sminax,
   x     strike, sdepth, stime, conf, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 orid
real*4 sxx
real*4 syy
real*4 szz
real*4 stt
real*4 sxy
real*4 sxz
real*4 syz
real*4 stx
real*4 sty
real*4 stz
real*4 sdobs
real*4 smajax
real*4 sminax
real*4 strike
real*4 sdepth
real*8 stime
real*4 conf
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function origin_Aadd(table, recnum, qa_flag, lat, lon, depth,
   x     time, orid, evid, jdate, nass, ndef, ndp, grn, srn, etype, depdp,
   x     dtype, mb, mbid, ms, msid, ml, mlid, algorithm, auth, commid,
   x     lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
real*4 lat
```

```
real*4 lon
real*4 depth
real*8 time
integer*4 orid
integer*4 evid
integer*4 jdate
integer*4 nass
integer*4 ndef
integer*4 ndp
integer*4 grn
integer*4 srn
character*7  etype(recnum)
real*4 depdp
character*1  dtype(recnum)
real*4 mb
integer*4 mbid
real*4 ms
integer*4 msid
real*4 ml
integer*4 mlid
character*15  algorithm(recnum)
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function remark_Aadd(table, recnum, qa_flag, commid, lineno,
   x     remark, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 commid
integer*4 lineno
character*80  remark(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function sensor_Aadd(table, recnum, qa_flag, sta, chan, time,
   x     endtime, inid, chanid, jdate, calratio, calper, tshift,
   x     instant, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
character*8  chan(recnum)
real*8 time
real*8 endtime
integer*4 inid
integer*4 chanid
integer*4 jdate
real*4 calratio
real*4 calper
real*4 tshift
```

```
character*1  instant(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function site_Aadd(table, recnum, qa_flag, sta, ondate,
   x     offdate, lat, lon, elev, staname, statype, refsta, dnorth,
   x     deast, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
integer*4 ondate
integer*4 offdate
real*4 lat
real*4 lon
real*4 elev
character*50  staname(recnum)
character*4  statype(recnum)
character*6  refsta(recnum)
real*4 dnorth
real*4 deast
character*17  lddate(recnum)
integer*4 len_f_table

integer function sitechan_Aadd(table, recnum, qa_flag, sta, chan, ondate,
   x     chanid, offdate, ctype, edepth, hang, vang, descrip, lddate,
   x     len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
character*8  chan(recnum)
integer*4 ondate
integer*4 chanid
integer*4 offdate
character*4  ctype(recnum)
real*4 edepth
real*4 hang
real*4 vang
character*50  descrip(recnum)
character*17  lddate(recnum)
integer*4 len_f_table

integer function sregion_Aadd(table, recnum, qa_flag, srn, srname,
   x     lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 srn
character*40  srname(recnum)
character*17  lddate(recnum)
integer*4 len_f_table
```

```
integer function stamag_Aadd(table, recnum, qa_flag, magid, sta,
   x     arid, orid, evid, phase, magtype, magnitude, uncertainty,
   x     auth, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 magid
character*6  sta(recnum)
integer*4 arid
integer*4 orid
integer*4 evid
character*8  phase(recnum)
character*6  magtype(recnum)
real*4 magnitude
real*4 uncertainty
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function stassoc_Aadd(table, recnum, qa_flag, stassid, sta,
   x     etype, location, dist, azimuth, lat, lon, depth, time, imb,
   x     ims, iml, auth, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
integer*4 stassid
character*6  sta(recnum)
character*7  etype(recnum)
character*32  location(recnum)
real*4 dist
real*4 azimuth
real*4 lat
real*4 lon
real*4 depth
real*8 time
real*4 imb
real*4 ims
real*4 iml
character*15  auth(recnum)
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table

integer function wfdisc_Aadd(table, recnum, qa_flag, sta, chan, time,
   x     wfid, chanid, jdate, endtime, nsamp, samprate, calib, calper,
   x     instype, segtype, datatype, clip, dir, dfile, foff, commid,
   x     lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
character*8  chan(recnum)
```

```
real*8 time
integer*4 wfid
integer*4 chanid
integer*4 jdate
real*8 endtime
integer*4 nsamp
real*4 samprate
real*4 calib
real*4 calper
character*6  instype(recnum)
character*1  segtype(recnum)
character*2  datatype(recnum)
character*1  clip(recnum)
character*64  dir(recnum)
character*32  dfile(recnum)
integer*4 foff
integer*4 commid
character*17  lddate(recnum)
integer*4 len_f_table
integer*4 len_f_query

integer function wftag_Aadd(table, recnum, qa_flag, tagname, tagid,
   x     wfid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*8  tagname(recnum)
integer*4 tagid
integer*4 wfid
character*17  lddate(recnum)
integer*4 len_f_table

integer function wftape_Aadd(table, recnum, qa_flag, sta, chan, time,
   x     wfid, chanid, jdate, endtime, nsamp, samprate, calib, calper,
   x     instype, segtype, datatype, clip, dir, dfile, volname,
   x     tapefile, tapeblock, commid, lddate, len_f_table)
character *80 table
integer*4 recnum
integer*4 qa_flag
character*6  sta(recnum)
character*8  chan(recnum)
real*8 time
integer*4 wfid
integer*4 chanid
integer*4 jdate
real*8 endtime
integer*4 nsamp
real*4 samprate
real*4 calib
real*4 calper
character*6  instype(recnum)
character*1  segtype(recnum)
character*2  datatype(recnum)
```

```
character*1  clip(recnum)
character*64  dir(recnum)
character*32  dfile(recnum)
character*6  volname(recnum)
integer*4 tapefile
integer*4 tapeblock
integer*4 commid
character*17 lddate(recnum)
integer*4 len_f_table
```

FORTRAN Fetch Routines (see array_fetch.3):

```
integer function get_affiliatio(table, query, maxrec, net, sta, lddate,
   x     len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*8  net(maxrec)
character*6  sta(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_arrival(table, query, maxrec, sta, time, arid,
   x     jdate, stassid, chanid, chan, iphase, stype, deltim, azimuth,
   x     delaz, slow, delslo, ema, rect, amp, per, logat, clip, fm, snr,
   x     qual, auth, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
real*8 time
integer*4 arid
integer*4 jdate
integer*4 stassid
integer*4 chanid
character*8  chan(maxrec)
character*8  iphase(maxrec)
character*1  stype(maxrec)
real*4 deltim
real*4 azimuth
real*4 delaz
real*4 slow
real*4 delslo
real*4 ema
real*4 rect
real*4 amp
real*4 per
real*4 logat
character*1  clip(maxrec)
character*2  fm(maxrec)
real*4 snr
```

```
character*1  qual(maxrec)
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_assoc(table, query, maxrec, arid, orid,
   x     sta, phase, belief, delta, seaz, esaz, timeres, timedef,
   x     azres, azdef, slores, slodef, emares, wgt, vmodel, commid,
   x     lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 arid
integer*4 orid
character*6  sta(maxrec)
character*8  phase(maxrec)
real*4 belief
real*4 delta
real*4 seaz
real*4 esaz
real*4 timeres
character*1  timedef(maxrec)
real*4 azres
character*1  azdef(maxrec)
real*4 slores
character*1  slodef(maxrec)
real*4 emares
real*4 wgt
character*15  vmodel(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_event(table, query, maxrec, evid, evname,
   x     prefor, auth, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 evid
character*15  evname(maxrec)
integer*4 prefor
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query
```

```
integer function get_gregion(table, query, maxrec, grn, grname,
   x     lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 grn
character*40  grname(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_instrument(table, query, maxrec, inid, insname,
   x     instype, band, digital, samprate, ncalib, ncalper, dir,
   x     dfile, rsptype, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 inid
character*50  insname(maxrec)
character*6  instype(maxrec)
character*1  band(maxrec)[1]
character*1  digital(maxrec)
real*4 samprate
real*4 ncalib
real*4 ncalper
character*64  dir(maxrec)
character*32  dfile(maxrec)
character*6  rsptype(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_netmag(table, query, maxrec, magid, net,
   x     orid, evid, magtype, nsta, magnitude, uncertainty, auth,
   x     commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 magid
character*8  net(maxrec)
integer*4 orid
integer*4 evid
character*6  magtype(maxrec)
integer*4 nsta
real*4 magnitude
real*4 uncertainty
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query
```

```
integer function get_network(table, query, maxrec, net, netname,
   x     nettype, auth, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*8  net(maxrec)
character*80  netname(maxrec)
character*4  nettype(maxrec)
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_origerr(table, query, maxrec, orid, sxx, syy,
   x     szz, stt, sxy, sxz, syz, stx, sty, stz, sdobs, smajax, sminax,
   x     strike, sdepth, stime, conf, commid, lddate, len_f_table,
   x     len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 orid
real*4 sxx
real*4 syy
real*4 szz
real*4 stt
real*4 sxy
real*4 sxz
real*4 syz
real*4 stx
real*4 sty
real*4 stz
real*4 sdobs
real*4 smajax
real*4 sminax
real*4 strike
real*4 sdepth
real*8 stime
real*4 conf
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_origin(table, query, maxrec, lat, lon, depth,
   x     time, orid, evid, jdate, nass, ndef, ndp, grn, srn, etype, depdp,
   x     dtype, mb, mbid, ms, msid, ml, mlid, algorithm, auth, commid,
   x     lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
real*4 lat
real*4 lon
```

```
real*4 depth
real*8 time
integer*4 orid
integer*4 evid
integer*4 jdate
integer*4 nass
integer*4 ndef
integer*4 ndp
integer*4 grn
integer*4 srn
character*7  etype(maxrec)
real*4 depdp
character*1  dtype(maxrec)
real*4 mb
integer*4 mbid
real*4 ms
integer*4 msid
real*4 ml
integer*4 mlid
character*15  algorithm(maxrec)
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_remark(table, query, maxrec, commid, lineno,
   x     remark, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 commid
integer*4 lineno
character*80  remark(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_sensor(table, query, maxrec, sta, chan, time,
   x     endtime, inid, chanid, jdate, calratio, calper, tshift,
   x     instant, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
character*8  chan(maxrec)
real*8 time
real*8 endtime
integer*4 inid
integer*4 chanid
integer*4 jdate
real*4 calratio
real*4 calper
```

```
real*4 tshift
character*1  instant(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_site(table, query, maxrec, sta, ondate,
   x     offdate, lat, lon, elev, staname, statype, refsta, dnorth,
   x     deast, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
integer*4 ondate
integer*4 offdate
real*4 lat
real*4 lon
real*4 elev
character*50  staname(maxrec)
character*4  statype(maxrec)
character*6  refsta(maxrec)
real*4 dnorth
real*4 deast
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_sitechan(table, query, maxrec, sta, chan, ondate,
   x     chanid, offdate, ctype, edepth, hang, vang, descrip, lddate,
   x     len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
character*8  chan(maxrec)
integer*4 ondate
integer*4 chanid
integer*4 offdate
character*4  ctype(maxrec)
real*4 edepth
real*4 hang
real*4 vang
character*50  descrip(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_sregion(table, query, maxrec, srn, srname,
   x     lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
```

```
integer*4 srn
character*40  srname(maxrec)
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_stamag(table, query, maxrec, magid, sta,
   x     arid, orid, evid, phase, magtype, magnitude, uncertainty,
   x     auth, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 magid
character*6  sta(maxrec)
integer*4 arid
integer*4 orid
integer*4 evid
character*8  phase(maxrec)
character*6  magtype(maxrec)
real*4 magnitude
real*4 uncertainty
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_stassoc(table, query, maxrec, stassid, sta,
   x     etype, location, dist, azimuth, lat, lon, depth, time, imb,
   x     ims, iml, auth, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
integer*4 stassid
character*6  sta(maxrec)
character*7  etype(maxrec)
character*32  location(maxrec)
real*4 dist
real*4 azimuth
real*4 lat
real*4 lon
real*4 depth
real*8 time
real*4 imb
real*4 ims
real*4 iml
character*15  auth(maxrec)
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query
```

```
integer function get_wfdisc(table, query, maxrec, sta, chan, time,
   x     wfid, chanid, jdate, endtime, nsamp, samprate, calib, calper,
   x     instype, segtype, datatype, clip, dir, dfile, foff, commid,
   x     lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
character*8  chan(maxrec)
real*8 time
integer*4 wfid
integer*4 chanid
integer*4 jdate
real*8 endtime
integer*4 nsamp
real*4 samprate
real*4 calib
real*4 calper
character*6  instype(maxrec)
character*1  segtype(maxrec)
character*2  datatype(maxrec)
character*1  clip(maxrec)
character*64  dir(maxrec)
character*32  dfile(maxrec)
integer*4 foff
integer*4 commid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_wftag(table, query, maxrec, tagname, tagid,
   x     wfid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*8  tagname(maxrec)
integer*4 tagid
integer*4 wfid
character*17  lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

integer function get_wftape(table, query, maxrec, sta, chan, time,
   x     wfid, chanid, jdate, endtime, nsamp, samprate, calib, calper,
   x     instype, segtype, datatype, clip, dir, dfile, volname,
   x     tapefile, tapeblock, commid, lddate, len_f_table, len_f_query)
character *80 table
character *80 query
integer*4 maxrec
character*6  sta(maxrec)
character*8  chan(maxrec)
real*8 time
integer*4 wfid
```

integer*4 chanid
integer*4 jdate
real*8 endtime
integer*4 nsamp
real*4 samprate
real*4 calib
real*4 calper
character*6 instype(maxrec)
character*1 segtype(maxrec)
character*2 datatype(maxrec)
character*1 clip(maxrec)
character*64 dir(maxrec)
character*32 dfile(maxrec)
character*6 volname(maxrec)
integer*4 tapefile
integer*4 tapeblock
integer*4 commid
character*17 lddate(maxrec)
integer*4 len_f_table
integer*4 len_f_query

DESCRIPTION

These functions provide centralized access to the core relations of the Center for Seismic Studies Database.

Include Files

The *dbsql{cf}.h* header files contain application specific database codes such as BADDATA and UNIXERR. The *dbora{cf}.h* header files contain database specific (ORACLE) codes such as DEADLOCK and NOTFOUND. *db_na.h* contains definitions for minimum, maximum, and NA values for database attributes that are checked prior to input into the database.

C programs which use any *libdb30.a* insert and fetch routines must include the C structure declaration for the table and use that defined structure for passing data to and from the routine. Include files for each table in the database follow a specific convention. The **db_** *relation*. **h** include files, such as *db_wfdisc.h*, contain the C structure declarations for a given table. Names of structure elements are the same as the attribute name in the database table. Character fields are one greater than the database definition to allow for the NULL terminator. For example, the *sta* attribute is a VARCHAR(6) in the database and the structure element is declared to be *char sta[7]*.

Other include files are provided for C and FORTRAN development for convenience. The *O_* **RELATION.** *H* files, such as *O_WFDISC.H*, contain simple variable C declarations for ORACLE. The *OA_* **RELATION.** *H* files, such as *OA_WFDISC.H*, contain array[50] variable C declarations for ORACLE. Each variable is named *table_attribute*. Character fields are the same size as the attribute in the database, for example *char wfdisc_sta[6]*. Any code which interacts directly with the database will find these include files helpful.

The *f_* **relation** *.h* files, such as *f_wfdisc.h*, contain variable declarations for FORTRAN.

The *libdb30_defs.h* file contains function prototypes for libdb30 functions.

Database Communications

dbopen establishes a connection with the database using the database identifying string in *uid*. For example:

```
/* Sample C database op
char    userid[80];
int     ierr;
ierr=dbopen(userid);
if(ierr < DBNOERROR)
{       fprintf(stderr, "Error opening database0);
        exit(ierr);
}

c
c Sample FORTRAN database open
c
        character*80    userid
        integer         ierr
        ierr = dbopen(80,userid)
        if( ierr .lt. DBNOER ) then
                write(GSTDOUT,*)'Error attaching to database'
                goto 9000
        end if
```

dbclose closes the database connection:

```
/* C Sample to close database */
        ierr=dbclose();

c ---FORTRAN sample to close database---
        ierr = dbclose()
```

dbwhoami returns the name of the active ORACLE account.

dbcancel terminates the current query. The integer status it returns is not reliable at this time, so the calling application should ignore a non zero status. The cancelled query itself will return return DB_CANCEL (ORA-01013, "user requested cancel of current operation"). *dbcancel* calls an OCI function, so any application that implements it must link in $(ORACLE_HOME)/rdbms/lib/libocic.a.

Transaction Management

dbcommit and *dbrollback* provide commands for managing transactions:

```
/* C Transaction Management calls */
        ierr = dbcommit();
        ierr = dbrollback();

c FORTRAN Transaction Management calls
        ierr = dbcommit()
        ierr = dbrollback()
```

A database transaction is a statement, or statements, treated as an atomic unit. A transaction is also called a work group. A logical unit of work begins when the first SQL statement is executed and ends when a COMMIT or ROLLBACK statement is executed. A work unit also ends when a data definition statement, such as CREATE TABLE or DROP INDEX, is executed. *dbcommit* explicitely ends the current SQL work group by committing all pending changes. *dbrollback* ends the current SQL work group by rolling back all pending changes.

Counters

Please see *dbgetcounter.3*.

String Manipulation

Two functions convert between NULL-terminated C strings and the blanked padded arrays required by the database. *cstr_to_pad* copies the contents of NULL-terminated C *string* to *array* and then blank pads *array* to the size of *array_length*. If *string* is larger than *array* only the number of characters which will fit will be copied (in other words, the original string is truncated when it is copied to *array*). *If the string is truncated,* Given the following includes and declarations:

```
#include "db_arrival.h"
EXEC SQL INCLUDE O_ARRIVAL.H;
struct arrival arrival_tuple;
if(cstr_to_pad(arrival_sta, arrival_tuple.sta, sizeof(arrival_sta)) == TRUNCATION)
        fprintf(stderr, "Warning: arrival.sta value truncated0);
```

pad_to_cstr goes the opposite direction. It copies the data of a blank padded character array to a NULL-terminated C string:

```
pad_to_cstr(arrival_tuple.sta,arrival_sta,sizeof(arrival_tuple.sta),sizeof(arrival_sta));
```

Error Handling

Please see dberror.3.

Insert Routines

Please see array_insert.3.

Fetch Routines

Please see array_fetch.3.

SEE ALSO

dbgetcounter.3
dberror.3
array_insert.3
array_fetch.3
Center for Seismic Studies Version 3 Database: Schema Reference Manual

NAME

Database Routines:

apma_Aadd, ceppks_Aadd, cpdisc_Aadd, db_count, db_delete, db_delete_array, db_exec_sql, detection_Aadd, evchar_Aadd, eventid_Aadd, fkdisc_Aadd, fsdisc_Aadd, sbsnr_Aadd, get_apma, get_ceppks, get_cpdisc, get_ceppks, get_detction, get_evchar, get_eventid, get_fkdisc, get_fsdisc, get_intarray, get_locregion, get_mag_coefs, get_merstat, get_mine, get_sbsnr, get_script, get_scriptloc, get_seisgrid, get_smatch, get_smatchvar, get_spvar, get_timestamp, locregion_Aadd, merstat_Aadd, sbsnr_Aadd, script_Aadd, scriptloc_Aadd, smatch_Aadd, smatchvar_Aadd, sp_timeadd, spvar_Aadd, sqltrace, update_sptime, update_timestamp

Retained for backward compatibility:

apmaadd, detadd, fkdadd, fsdadd, locadd, sbsnradd,

Calls from fortran applications:

apma_aadd_, dbcount_, dbdeletearray_, detection_aadd_, eventid_aadd_, fkdisc_aadd_, fsdisc_aadd_, get_apma_, get_detection_, get_eventid_, sbsnr_aadd_, get_mine_, get_sbsnr_, get_script_, get_scriptloc_, get_seisgrid_, get_smatch_, get_smatchvar_, get_spvar_, sp_timeadd_, sbsnr_aadd_, script_aadd_, scriptloc_aadd_, smatch_aadd_, smatchvar_aadd_, spvar_aadd_ update_sptm_, update_tmstamp_, get_timestamp_

Retained for backward compatibility:

apmaadd_, detadd_, fkdadd_, fsdadd_, sbsnradd_

SYNOPSIS

Include Files:

```
#include "dborac.h"
#include "dbsqlc.h"
#include "dbimsc.h"
#include "libdbims_defs.h"
```

Fortran Include Files:

```
include '../../dboraf.h'
include '../../dbsqlf.h'
```

Insert Routines:

```
#include "db_apma.h"
int
apma_Aadd(table, ptr, recnum, qa_flag)          /* adds an array of records to an apma structured relation. */
char    *table;                  /* (i) dynamic name of table */
struct apma *ptr;                /* (i) beginning address of records to insert */
int     recnum;                  /* (i) number of structures to insert */
int     qa_flag;                 /* (i) if flag TRUE, data checking enabled */

int
ceppks_Aadd(table, ptr, recnum, qa_flag)        /*adds an array of records to a ceppks structured relation */
char    *table;                  /* (i) dynamic name of table */
struct ceppks *ptr;              /* (i) beginning address of records to insert */
int     recnum;                  /* (i) number of structures to insert */
int     qa_flag;                 /* (i) if flag TRUE, data checking enabled */
```

```
int
cpdisc_Aadd(table, ptr, recnum, qa_flag) /* adds an array of records to a cpdisc structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct cpdisc *ptr;             /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

#include "db_detection.h"
int
detection_Aadd(table, ptr, recnum, qa_flag)     /* adds an array of records to a detection structured relation.
char    *table;                 /* (i) dynamic name of table */
struct detection *ptr;          /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

int
evchar_Aadd(table, ptr, recnum, qa_flag)/* adds an array of records to a evchar structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct evchar *ptr;             /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

int
eventid_Aadd(table, ptr, recnum, qa_flag)       /* adds an array of records to a eventid structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct eventid *ptr;            /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

#include "db_fkdisc.h"
int
fkdisc_Aadd(table, ptr, recnum, qa_flag) /* adds an array of records to a fkdisc structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct fkdisc *ptr;             /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

#include "db_fsdisc.h"
int
fsdisc_Aadd(table, ptr, recnum, qa_flag) /* adds an array of records to a fsdisc structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct fsdisc *ptr;             /* (i) beginning address of records to insert */
int     recnum;                 /* (i) number of structures to insert */
int     qa_flag;                /* (i) if flag TRUE, data checking enabled */

#include "db_locregion.h"
int
locregion_Aadd(table, ptr, recnum, qa_flag)     /* adds an array of records to a
locregion structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct locregion *ptr;          /* (i) beginning address of records to insert */
```

```
int    recnum;                  /* (i) number of structures to insert */
int    qa_flag;                 /* (i) if flag TRUE, data checking enabled */

#include "db_merstat.h"
int
merstat_Aadd(table, ptr, recnum, qa_flag)        /* adds an array of records to
a merstat structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct merstat *ptr;            /* (i) beginning address of records to insert */
int    recnum;                  /* (i) number of structures to insert */
int    qa_flag;                 /* (i) if flag is TRUE, data checking enabled *
/

#include "db_sbsnr.h"
int
sbsnr_Aadd(table, ptr, recnum, qa_flag)          /* adds an array of records to
a sbsnr structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct sbsnr *ptr;              /* (i) beginning address of records to insert */
int    recnum;                  /* (i) number of sbsnr structures to insert */
int    qa_flag;                 /* (i) if flag is TRUE, data checking enabled */

#include "db_script.h"
int
script_Aadd(table, ptr, recnum, qa_flag)         /* adds an array of records to
a script structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct script *ptr;             /* (i) beginning address of records to insert */
int    recnum;                  /* (i) number of script structures to insert */
int    qa_flag;                 /* (i) if flag is TRUE, data checking enabled */

#include "db_scriptloc.h"
int
scriptloc_Aadd(table, ptr, recnum, qa_flag)              /* adds an array of records to
a scriptloc structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct scriptloc *ptr;          /* (i) beginning address of records to insert */
int    recnum;                  /* (i) number of scriptloc structures to insert */
int    qa_flag;                 /* (i) if flag is TRUE, data checking enabled */

#include "db_smatch.h"
int
smatch_Aadd(table, ptr, recnum, qa_flag)                 /* adds an array of records to
a smatch structured relation. */
char    *table;                 /* (i) dynamic name of table */
struct smatch *ptr;             /* (i) beginning address of records to insert */
int    recnum;                  /* (i) number of smatch structures to insert */
int    qa_flag;                 /* (i) if flag is TRUE, data checking enabled */
```

```
#include "db_smatchvar.h"
int
smatchvar_Aadd(table, ptr, recnum, qa_flag)          /* adds an array of records to
a smatchvar structured relation. */
char    *table;                  /* (i) dynamic name of table */
struct smatchvar *ptr;           /* (i) beginning address of records to insert */
int     recnum;                  /* (i) number of smatchvar structures to insert */
int     qa_flag;                 /* (i) if flag is TRUE, data checking enabled */

#include "db_spvar.h"
int
spvar_Aadd(table, ptr, recnum, qa_flag)          /* adds an array of records to
a spvar structured relation. */
char    *table;                  /* (i) dynamic name of table */
struct spvar *ptr;               /* (i) beginning address of records to insert */
int     recnum;                  /* (i) number of spvar structures to insert */
int     qa_flag;                 /* (i) if flag is TRUE, data checking enabled */

int
sp_timeadd( sta, time )          /* adds a record to the sigpro_time relation*/
char    *sta;
double *time;
```

Update Routines:

```
int
update_sptime( sta, time )                /* updates the sigpro_time relation*/
char     *sta;                            /* (i) where sta = this value */
double   *time;                           /* (i) set time to this value */

int
update_timestamp(time, where_clause)      /* updates the timestamp relation*/
double  time;                             /* (i) set time to this value */
char     *where_clause;                   /* (i) where procname = 'x' or where procclass = 'x' */
```

Database retrieval Routines:

```
#include "db_apma.h"
int
get_apma(table_name, where_clause, apma_tuples, maxrec) /* retreive from apma structured table */
char            *table_name;          /* (i) dynamic name of table */
char            *where_clause;        /* (i) search criteria for table */
struct apma     **apma_tuples;        /* (o) address of records retrieved */
int             maxrec;               /* (i) max number of records to fetch */

#include "db_cpdisc.h"
int
get_cpdisc(table_name, where_clause, cpdisc_tuples, maxrec)      /* retreive from cpdisc structured table */
char    *table_name;                 /* (i) dynamic name of table      */
char    *where_clause;               /* (i) search criteria for table  */
struct cpdisc **cpdisc_tuples;       /* (o) address of records retrieved */
```

```
int     maxrec;                     /* (i) max records to fetch        */

#include "db_ceppks.h"
int
get_ceppks(table_name, where_clause, ceppks_tuples, maxrec)     /* retreive from ceppks structured table */
char    *table_name;                /* (i) dynamic name of table       */
char    *where_clause;              /* (i) search criteria for table   */
struct ceppks **ceppks_tuples;      /* (o) address of records retrieved */
int     maxrec;                     /* (i) max records to fetch        */

#include "db_detection.h"
int
get_detection(table_name, where_clause, det_tuples, maxrec) /* retreive from detection structured table */
char            *table_name;        /* (i) dynamic name of table */
char            *where_clause;      /* (i) search criteria for table */
struct detection **det_tuples;      /* (o) address of records retrieved */
int     maxrec;                     /* (i) max number of records to fetch */

#include "db_evchar.h"
int
get_evchar(table_name, where_clause, evchar_tuples, maxrec)
char    *table_name;                /* (i) dynamic name of table       */
char    *where_clause;              /* (i) search criteria for table   */
struct evchar **evchar_tuples;      /* (o) address of records retrieved */
int     maxrec;                     /* (i) max records to fetch        */

#include "db_eventid.h"
int
get_eventid(table_name, where_clause, eventid_tuples, maxrec)
char    *table_name;                /* (i) dynamic name of table       */
char    *where_clause;              /* (i) search criteria for table   */
struct eventid **eventid_tuples;    /* (o) address of records retrieved */
int     maxrec;                     /* (i) max records to fetch        */

int
get_fkdisc(table_name, where_clause, fkdisc_tuples, maxrec)     /* retreive from fkdisc structured table */
char            *table_name;        /* (i) dynamic name of table */
char            *where_clause;      /* (i) search criteria for table */
struct  fkdisc **fkdisc_tuples;     /* (o) address of return list of records */
int             maxrec;             /* (i) max number of records to fetch */

int
get_intarray(select, int_array, maxrec)         /* retrieve an array of integers (e.g. id's) */
char    *select;                    /* (i) search criteria for table   */
int     **int_array;                /* (o) address of records retrieved */
int     maxrec;                     /* (i) max records to fetch        */

#include "db_locregion.h"
int
get_locregion(table_name, where_clause, locregi_tuples, maxrec)         /* retreive from locregion structured
char    *table_name;                /* (i) dynamic name of table       */
```

```
char    *where_clause;                  /* (i) search criteria for table    */
struct locregion **locregi_tuples;      /* (o) address of records retrie
ved */
int     maxrec;                         /* (i) max records to fetch         */

#include "db_mag_coefs.h"
int
get_mag_coefs(table_name, where_clause, mag_coefs_tuples, maxrec)       /* retreive from mag_coefs structured
char            *table_name;            /* (i) dynamic name of table */
char            *where_clause;          /* (i) search criteria for table */
struct   mag_coefs **mag_coefs_tuples;          /* (o) address of return list of records */
int             maxrec;                 /* (i) max number of records to fetch */

#include "db_merstat.h"
int
get_merstat(table_name, where_clause, merstat_tuples, maxrec)    /* retreive from merstat structured table */
char    *table_name;                    /* (i) dynamic name of table        */
char    *where_clause;                  /* (i) search criteria for table    */
struct merstat **merstat_tuples; /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch         */

#include "db_mine.h"
int
get_mine(table_name, where_clause, mine_tuples, maxrec)          /* retreive from mine structured table */
char    *table_name;                    /* (i) dynamic name of table        */
char    *where_clause;                  /* (i) search criteria for table    */
struct mine **mine_tuples;       /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch         */

int
get_sbsnr(table_name, where_clause, sbsnr_tuples, maxrec)        /* retreive from sbsnr structured table */
char            *table_name;            /* (i) dynamic name of table */
char            *where_clause;          /* (i) search criteria for table */
struct sbsnr    **sbsnr_tuples;         /* (o) address of records retrieved */
int             maxrec;                 /* (i) max records to fetch */

#include "db_script.h"
int
get_script(table_name, where_clause, script_tuples, maxrec)       /* retreive from script structured table */
char    *table_name;                    /* (i) dynamic name of table        */
char    *where_clause;                  /* (i) search criteria for table    */
struct script **script_tuples;   /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch         */

#include "db_scriptloc.h"
int
get_scriptloc(table_name, where_clause, scriptloc_tuples, maxrec) /* retreive from scriptloc structured table */
char    *table_name;                    /* (i) dynamic name of table        */
char    *where_clause;                  /* (i) search criteria for table    */
struct scriptloc **scriptloc_tuples;    /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch         */
```

```
#include "db_seisgrid.h"
int
get_seisgrid(table_name, where_clause, seisgrid_tuples, maxrec)    /* retreive from seisgrid structured table */
char    *table_name;                    /* (i) dynamic name of table       */
char    *where_clause;                  /* (i) search criteria for table   */
struct seisgrid **seisgrid_tuples; /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch        */

#include "db_smatch.h"
int
get_smatch(table_name, where_clause, smatch_tuples, maxrec)     /* retreive from smatch structured table */
char    *table_name;                    /* (i) dynamic name of table       */
char    *where_clause;                  /* (i) search criteria for table   */
struct smatch **smatch_tuples;  /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch        */

#include "db_smatchvar.h"
int
get_smatchvar(table_name, where_clause, smatchvar_tuples, maxrec)       /* retreive from smatchvar structure
char    *table_name;                    /* (i) dynamic name of table       */
char    *where_clause;                  /* (i) search criteria for table   */
struct smatchvar **smatchvar_tuples;    /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch        */

#include "db_spvar.h"
int
get_spvar(table_name, where_clause, spvar_tuples, maxrec)       /* retreive from spvar structured table */
char    *table_name;                    /* (i) dynamic name of table       */
char    *where_clause;                  /* (i) search criteria for table   */
struct spvar **spvar_tuples;    /* (o) address of records retrieved */
int     maxrec;                         /* (i) max records to fetch        */

double
get_timestamp(query)                    /* select time from any table */
char    *query;                         /* (i) "select time from table whereclause" */
```

Count and Delete Routines:

```
int
db_delete(table_name, where_clause)             /* delete rows from any table */
char            *table_name;            /* (i) dynamic name of table */
char            *where_clause;          /* (i) search criteria for table */

int
db_count(table, key, where_clause)          /* count the number of records in a result set */
char    *table;
char    *key;
char    *where_clause;
```

Miscellaneous Routines:

```
int
sqltrace(mode)                          /* turn SQL_TRACE on/off */
int     mode;
```

Calls from fortran applications:

```
      integer function apma_aadd(table, irecnum, iqa_flag,
     &        phase, arid, freq, snr, ampp, amps,
     &       amplr, rect, plans, planlr, hvratp, hvrat, hmxmn,
     &       inang3, seazp, seazs, seazlr, inang1, ptime, stime,
     &       auth, apmarid, commid, lddate, len_table )
      character*80 table
      integer*4 irecnum
      integer*4 iqa_flag
      character*8  phase(irecnum)
      integer*4    arid(irecnum)
      real*4       freq(irecnum)
      real*4       snr(irecnum)
      real*4       ampp(irecnum)
      real*4       amps(irecnum)
      real*4       amplr(irecnum)
      real*4       rect(irecnum)
      real*4       plans(irecnum)
      real*4       planlr(irecnum)
      real*4       hvratp(irecnum)
      real*4       hvrat(irecnum)
      real*4       hmxmn(irecnum)
      real*4       inang3(irecnum)
      real*4       seazp(irecnum)
      real*4       seazs(irecnum)
      real*4       seazlr(irecnum)
      real*4       inang1(irecnum)
      real*8       ptime(irecnum)
      real*8       stime(irecnum)
      character*15 auth(irecnum)
      integer*4    apmarid(irecnum)
      integer*4    commid(irecnum)
      character*17  lddate(irecnum)
      integer*4 len_table

      integer function dbcount( table, key, query, len_table,
     &                          len_key, len_query )
      character*80 table
      character*80 key
      character*80 query
      integer*4 len_table
      integer*4 len_key
      integer*4 len_query

      integer function dbdeletearray( table, whereclause, array, num,
     &                          table_f_len, where_f_len )
      character*80 table
      character*80 whereclause
      integer*4 array()
```

```
integer*4 num
integer*4 table_f_len
integer*4 where_f_len

integer function detection_aadd( table, irecnum, iqa_flag,
&                                arid, time, sta, chan, bmtyp, sproid,
&                          cfreq, seaz, delaz, slow, delslo, snr, stav,
&                          fstat, deltim, bandw, fkqual, commid,
&                                lddate, len_table )
character*80 table
integer*4 irecnum
integer*4 iqa_flag
integer*4    arid(irecnum)
real*8       time(irecnum)
character*6  sta(irecnum)
character*8  chan(irecnum)
character*4  bmtyp(irecnum)
integer*4    sproid(irecnum)
real*4       cfreq(irecnum)
real*4       seaz(irecnum)
real*4       delaz(irecnum)
real*4       slow(irecnum)
real*4       delslo(irecnum)
real*4       snr(irecnum)

integer function fkdisc_aadd( table, irecnum, iqa_flag,
&        time, tlen, sta, fktyp, arid,
&        maxkx, maxsx, nx, maxky, maxsy, ny, cfreq, bandw,
&        commid, fkrid, fkid, datsw, foff, dir, dfile,
&        lddate, len_table)
character*80 table
integer*4 irecnum
integer*4 iqa_flag
real*8       time(irecnum)
real*4       tlen(irecnum)
character*6  sta(irecnum)
character*4  fktyp(irecnum)
integer*4    arid(irecnum)
real*4       maxkx(irecnum)
real*4       maxsx(irecnum)
integer*4    nx(irecnum)
real*4       maxky(irecnum)
real*4       maxsy(irecnum)
integer*4    ny(irecnum)
real*4       cfreq(irecnum)
real*4       bandw(irecnum)
integer*4    commid(irecnum)
integer*4    fkrid(irecnum)
integer*4    fkid(irecnum)
integer*4    datsw(irecnum)
integer*4    foff(irecnum)
character*64 dir(irecnum)
character*32 dfile(irecnum)
```

```
character*17  lddate(irecnum)
integer*4 len_table

integer function fsdisc_aadd( table, irecnum, iqa_flag,
&        time, tlen, sta, fstyp, arid,
&        maxf, nf, chanid, wfid, commid, fsrid, fsid, datsw,
&        foff, dir, dfile, lddate, len_table)
character*80 table
integer*4 irecnum
integer*4 iqa_flag
real*8       time(irecnum)
real*4       tlen(irecnum)
character*6  sta(irecnum)
character*4  fstyp(irecnum)
integer*4    arid(irecnum)
real*4       maxf(irecnum)
integer*4    nf(irecnum)
integer*4    chanid(irecnum)
integer*4    wfid(irecnum)
integer*4    commid(irecnum)
integer*4    fsrid(irecnum)
integer*4    fsid(irecnum)
integer*4    datsw(irecnum)
integer*4    foff(irecnum)
character*64 dir(irecnum)
character*32 dfile(irecnum)
character*17  lddate(irecnum)
integer*4 len_table

integer function sbsnr_aadd( table, irecnum, iqa_flag,
&              arid, sta, chan, stav, ltav, lddate, len_table)
character*80 table
integer*4 irecnum
integer*4 iqa_flag
integer*4    arid(irecnum)
character*6  sta(irecnum)
character*8  chan(irecnum)
real*4       stav(irecnum)
real*4       ltav(irecnum)
character*17  lddate(irecnum)
integer*4 len_table

integer function sp_timeadd( sta, time)
character*6  sta
real*8       time

integer function update_sptm( sta, time)
character*6  sta
real*8       time
```

```
integer function updatetimestamp( time, query)
character*80      query
real*8            time
integer*4         len_tquery

integer function get_timestamp(query, len_tquery)
character *80     query
real*8            time
integer*4         len_tquery
```

DESCRIPTION

These functions provide centralized access to the application specific IMS relations of the Center for Seismic Studies Database Version 3 database. FORTRAN interfaces are provided for C routines when requested. This library depends on libdb30.a for error handling and value checking routines.

Include Files

dbsql{c,f}.h contains application specific database codes such as BADDATA and UNIXERR. *dbora{c,f}.h* contains database specific (ORACLE) codes such as DEADLOCK and ORACLE_ROWCOUNT. *dbimsc.h* contains Function declarations for libdbims.a, #defines for fetches and inserts and Structure definitions included for function prototyping.

Include files for each table in the database follow a specific convention.

*db_* **relation.** *h* include files, such as *db_wfdisc.h*, contain the C structure declarations for a given table. Character fields are one greater than the database definition to allow for the NULL terminator, as in *char wfdisc_sta[7]*.

*f_* **relation.** *h* include files, such as *f_wfdisc.h*, contain the fortran structure declarations for a given table.

The library routines include *O_* **RELATION.** *H* files, such as *O_WFDISC.H*. These contain simple variable declarations for ORACLE. Each structure element is named *table_attribute*. Character fields are the same size as the attribute in the database, for example *char wfdisc_sta[6]*. Queries that do not use the library routines but, interact directly with the database require this include.

The *f_* **relation** *.h* files, such as *f_apma.h*, contain variable declarations for FORTRAN. Character fields are the same size as the field in the database.

The *libdbims_defs.h* file contains function prototypes for libdbims routines.

Error Handling

See dberror.3

Insert Routines

The insert routines add records to the application specific ims 3.0 release database relations. The records are passed in as a pointer to a structure defined in the include files located in ../../include/ims3/db_*relation_name*.h Array_insert.3 has a detailed description of the array insert routines. The sp_timeadd is an exception with only two attributes no structure is used. The in_*relation*add and out_*relation*add routines act on synonyms of the core database relations. These add routines have been replaced with the array add routines in libdb30.a utilizing the dynamic table name feature. Please see array_insert.3 for additional information. Backward compatibility has been maintained for a transition period. The return value is less than DBNOERROR if an error occured. If a database error occurred, a sql error code is returned. BADDATA is returned if the passed in attributes are out of range. Error code and text can be retreived with *dberror_get* (see dberror.3). Upon successful operation, the oracle row count is returned.

The SigPro insert routines call robustness checking programs to verify the attributes are in the specified range before updating the database. The key attributes are checked for valid value or valid null when passing data to the database. See libdb30.3 for an example of using the robustness checking program.

If *lddate* is '\0' or '-' (NA), it is assigned at insert time from "now" in Greenwich Meantime. The insert routines will not overwrite a non-NA *lddate*. The *lddate* must conform to the following ORACLE date mask: 'YYYYMMDD HH24:MI:SS'. This is a 24 hour time clock. For example: 19901112 16:05:33 is November 12, 1990, at 4:05 pm and 33 seconds. The insert routines convert this string TO_DATE(:lddate,'YYYYMMDD HH24:MI:SS').

The attribute jdate is calculated by the insert routine based on time for relations detection, fkdisc, and fsdisc.

Update Routines

The update routines alter a record in the application specific ims 3.0 release database relations. Updating is different than inserting in that on update existing data is altered. Inserts add supplemental records. The attributes sta and proctime are passed in for update_sptime. The value of proctime assocated with the specified sta is set by the routine for the relation sigpro_time. The lddate is also assigned based on sysdate. The routine update_timestamp() updates the time attribute in the IMS timestamp relation for those tuples that statisfy the constraints in the where_clause string. For example if time = 621983454 and where_clause = "WHERE PROCNAME = 'ESAL' AND PROCCLASS = 'IMS'" the complete SQL update statement would look like: UPDATE TIMESTAMP SET TIME = 621983454 WHERE PROCNAME = 'ESAL' AND PROCCLASS = 'IMS'. This would result in the time attribute being altered. The lddate is not set.

The return value is less than DBNOERROR if an error occured. If a database error occurred, an sql error code is returned. BADDATA is returned if the passed in attributes are out of range. Error code and text can be retreived with *dberror_get* (see dberror.3). Upon successful operation, the oracle row count is returned.

Database retrieval Routines:

The dastatus routines have a very specific application in the comm directory.

The get_timestamp returns the (single) value of the time attribute from any table, especially the time_stamp table. Unlike the other get_*relation* routines The query must include full syntax for a select. For example: "select time from time_stamp where procname = 'esal'" The get_*relation* routines (excluding get_timestamp) select an array of records from the IMS specific 3.0 release database relations. The table name is determined at runtime and passed in *via* the *table* argument. For example, *get_detection* fetches from a table which contains the structure of the 3.0 *detection* table, but the name does not have to be *detection*. It could be a synonym such as *in_detection* or a table tagged with the name of the owner such as *esal.detection*. This table_name string should contain the actual table name and the correlation variable name which allows joins. The routines require that the correlation name used for the main select be the same as the table structure name. Any correlation name may be used for the join table or in sub-selects. If the table name is the same as the structure definition, no correlation name is required. No host variables may be in this string. For example,

"in_detection detection"
"detection, assoc a"
"my_detection detection, assoc asc"

If no correlation name is given with a one word table name, it will be added. No host variables may be in this string.

The query is flexible and can be personalized. The "where_clause" is a string containing the complete where clause for the SELECT. For example,

"WHERE time > 600000000 and time <= 600001000"
"WHERE sta = 'NRA0'"
"WHERE detection.sta = a.sta order by detection.orid"

No host variables may be in the string. Character attributes as where clause delimeters must be in single quotes(' '). If the keyword *where* is missing it will be prepended. If a trailing semicolon is detected, it will be stripped. All attributes are selected for each record that meets the where clause. If no where clause is specified (a NULL string) all records will be retrieved up to the limit integer *maxrec* (the last argument). The records are passed back as a pointer to an array of structures defined in the include files located in ../../include/db3/db_ *relation*. h. A place holder is passed in the argument list for this pointer. The memory for this array of structures is dynamically allocated. The pointer should be free'd by the calling routine when it is no longer needed. The integer maxrec is the limit number of records to fetch. If all the records in the table are desired, the maxrec should be set above the maximum number of records that might be in the table. A large request for records will only result in valid data being returned. For exmple if *maxrec* is set to 1000 and only 3 rows of data actually satisfy the query, three structures will be dynamically allocated and filled with the attributes from the three records. The return value from the database routine will be the integer three and the place holder will point to the three structures. Be aware that there could potentially be a large amount of valid data. There is no upper limit on the maxrec parameter.

The following example shows how to call *get_detection:*

```
#include "dbsqlc.h"
#include "dborac.h"
#include "db_detection.h"
int     ierr;                           /* return code from get_detection */
int     recnum = 55;                    /* maximum number of records to fetch */

struct  detection  *det_rec_ptr;        /* place holder for detection records, memory dynamically
                                         * allocated in get_detection */

int     error_code,                     /* dberror_get: database error code */
char    error_string[DB_ERRORMSG_SIZE+1];   /* dberror_get: database error text */
int     print_flag;                     /* dberror_get: status of print_flag */
int     warn_flag;                      /* dberror_get: status of warn_flag */

if (dbopen(database) < DBNOERROR)
{
        (void) dberror_get(&error_code, error_string, &print_flag, &warn_flag);
        fprintf(stderr, "Error detected: %d %s\n", error_code, error_string);
        exit(-1);
}
ierr=get_detection("esal.detection", "where sta = 'ARC' and time <= 6000010000",
                &det_rec_ptr, recnum);
if(ierr < DBNOERROR)
{
        (void) dberror_get(&error_code, error_string, &print_flag, &warn_flag);
        fprintf(stderr, "Error detected in get_detection: %d %s\n", error_code, error_string);
        dbclose();
}
        /*
```

```
        * use the records as required
        */

    free(det_rec_ptr);          /* calling application is responsible for freeing memory
                                 * allocated in get_detection. */
    dbclose();
```

If a database error occurred, the sqlca.sqlcode is returned. UNIXERR is returned if the pointer to an argument is NULL. Error code and text can be retreived with *dberror_get* (see dberror.3). Upon successful operation, the oracle row count (number of records retrieved) is returned.

The timestamp and sigpro_time Routines

get_timestamp, *update_timestamp*, *sp_timeadd*, *update_sptime* and fortran interface routines *get_timestamp_*, *update_tmstamp_*, *sp_timeadd_*, *update_sptm_*, are used to insert, fetch and update data in the sigpro_time and timestamp relations. These are atypical relations used by the communication code to assign a timestamp to an occurance. They are used to signal processing completion and to track data transfers from NORSAR to Washington. Each sucessful transfer updates sigpro_time or timestamp thereby providing a timestamp as to how far processing for a given station has proceeded. The sigpro_time relation routines are specific to the Sigpro process. The timestamp relation and associated routines are generic and will ultimately replace sigpro_time and thresh_time. The get_timestamp routine has a dynamic query passed in and can select time from any relation.

Delete Routine

The delete routine deletes all rows in "table_name" that meet the "where_clause" criteria. If the where_clause contains a NULL string "", all rows in the table are deleted. The calling application is responsible for transaction management. The delete will not be visible untill a commit has been performed. The db_delete routine will wait if a parse lock is present on the table_name.

This code fragment illustrates a sample call to db_delete:

```
    dberror_init(TRUE, TRUE);

    if (dbopen("dataset/passord") < DBNOERROR)
    {
        (void) dberror_get(&error_code, error_string, &print_flag, &warn_flag);
        fprintf(stderr, "Error detected: %d %s\n", error_code, error_string);
        exit(-1);
    }
    /* delete rows from temp_assoc where the sta = 'AR0' */

    ret= (db_delete("temp_assoc","where sta='AR0'"));

    if (ret <= DBNOERROR)          /* assumes 0 rows deleted is an application error */
    {
        (void) dberror_get(&error_code, error_string, &print_flag, &warn_flag);
        fprintf(stderr, "Error detected in db_delete: %d %s\n", error_code, error_string);
        dbrollback();
        dbclose();
        exit(-1);
    }
    dbcommit();
    dbclose();
```

If a database error occurred, the sqlca.sqlcode is returned. If no database error occurred, the oracle rowcount is returned. A return code of 0 indicates no rows met the where clause criteria so none were deleted. This maybe an error depending on the application. UNIXERR is returned if the pointer to an argument is NULL. Error code and text can be retreived with *dberror_get* (see dberror.3).

Miscellaneous Routines

If the ORACLE instance has enabled TIMED_STATISTICS but not set SQL_TRACE = TRUE for the ORACLE instance, *sqltrace* supports turning trace on a session by session basis for PRO*C applications. If set to TRUE, a trace file will be generated in the location specified in the ORACLE startup parameter USER_DUMP_DEST. For more information about SQL_TRACE, see your Database Administrator.

FORTRAN INTERFACES

The routines which interface to Fortran usually have the same name as the non-interface routine, with an underscore appended and all lower case. (.e.g sbsnr_aadd_ is the interface to sbsnr_Aadd). For some routines with long names (more than 12 characters) the interface routine called by Fortran has a slightly different name to insure unique names within the first 12 characters. The Fortran compiler automatically adds the underscore when the code is compiled. The Fortran call to the routine does not need to show the underscore (See sample below). The interface routines will have a different list of parameters than the non-interface routine since Fortran does not use structures. The number of meaningful characters in the table name array is also passed to the routine.

Sample Fortran code

```
c
c     Sample FORTRAN insert into wfdisc table
c
      program wfdisc_Aadd_example
c
      implicit    undefined (a-z)
      character*20 table
      integer*4    ierr, print_flag, warn_flag, dberrno
      character*70 error_txt
      integer*4    i, numrec, nwfdisc, len_table, qa_flag
c
      include '/nmrd/dev/include/db3/dbsqlf.h'
      include '/nmrd/dev/include/db3/dboraf.h'
c
c     ------ declare array variables to hold wfdisc attributes ---
c
      parameter    (numrec = 50)
      character*64 dir(numrec)
      character*32 dfile(numrec)
      character*17  lddatc(numrec)
      character*8 chan(numrec)
      character*6 instype(numrec),sta(numrec)
      character*2 datatype(numrec)
      character*1 segtype(numrec),clip(numrec)
      integer*4 wfid(numrec),chanid(numrec),jdate(numrec)
      integer*4 nsamp(numrec),foff(numrec),commid(numrec)
      real*4 samprate(numrec),calib(numrec),calper(numrec)
      real*8 time(numrec),endtime(numrec)
```

```
c
c     ------------declare functions------------------
c
      integer*4 wfdisc_Aadd, dbopen, dbclose, dbcommit, dbrollback
c
c     --------- turn debugging on, print errors and warnings ------
c
      print_flag = 1
      warn_flag = 1
      call dberror_init( print_flag, warn_flag)
c
c     ---- FORTRAN strings are blank padded to the declared size.----
c     ---- The interface routine needs to know how many characters --
c     ---- are meaningful. ----------------------------------------
c
      table = 'wfdisc'
      len_table = 6
c
c     --- blank pad character strings to initialize
c
      do 1000 i=1,numrec
        call pad_to_blank(len(sta(i)),sta(i))
        call pad_to_blank(len(chan(i)),chan(i))
        call pad_to_blank(len(instype(i)),instype(i))
        call pad_to_blank(len(segtype(i)),segtype(i))
        call pad_to_blank(len(datatype(i)),datatype(i))
        call pad_to_blank(len(clip(i)),clip(i))
        call pad_to_blank(len(dir(i)),dir(i))
        call pad_to_blank(len(dfile(i)),dfile(i))
        call pad_to_blank(len(lddate(i)),lddate(i))
 1000 continue
c
c     ---- fill the arrays with valid data ---------
c
      do 1200 i=1,numrec
        time(i) = 632476541.0 + i
        wfid(i) = 1 + i

        chanid(i) = 2 + i
c       ------- jdate calculated by the insert routine ------
c       ------- endtime calculated by the insert routine ----
        nsamp(i) = 300 + i
        samprate(i) = 40.0 + i
        calib(i) = 5.0 + i
        calper(i) = 6.0 + i
        foff(i) = 7 + i
        commid(i) = 9 + i
        sta(i) = 'NRA0'
        chan(i) = 'ib'
        instype(i) = 'SRO'
        segtype(i) = 's'
        datatype(i) = 's4'
        clip(i) = 'c'
```

```
      dir(i) = '/data/pipeline/1990262/'
      dfile(i) = 'NRS.66003.02.w'
c     ------ '-' lddate will allow insert routine to assign ---
      lddate(i) = '-'
 1200 continue
c
c
c     ------- open database --------
c
      ierr = dbopen('scott/tiger',80)
      if ( ierr .ne. DBNOERROR ) then
        call dberror_get( dberrno, error_txt, print_flag, warn_flag )
        write(0,*) 'dbopen error:', dberrno, error_txt
c       ---------exit on error-------
        goto 9000
      endif
c
c     ----- turn data checking on ---------
      qa_flag = 1
c
      nwfdisc = wfdisc_Aadd(table, numrec, qa_flag, sta, chan,
     &     time, wfid, chanid, jdate, endtime, nsamp, samprate,
     &     calib, calper, instype, segtype, datatype,
     &     clip, dir, dfile, foff, commid, lddate,
     &     len_table )
c
c     ----- error check: 1 or more rows to have been inserted -----
c
      if ( nwfdisc .lt. DBNOERROR ) then
        call dberror_get( dberrno, error_txt, print_flag, warn_flag )
        write(0,*) 'wfdisc_Aadd error:', dberrno, error_txt
c
c       --------- do not commit on error -----------
        ierr = dbrollback()
c
c       --------- exit on error -------
        goto 9000
      endif
c
      if ( nwfdisc .ne. numrec ) then
        write(0,*) 'tried to insert', numrec,
     &          ' actually inserted', nwfdisc
      endif
      write(0,*) nwfdisc, 'rows inserted'
c
c     ------ commit the insert, this must be done or data is lost ---
c
      ierr = dbcommit()
c
c     --------- close database ---------
c
      ierr = dbclose()
c
```

```
c       ------- skip over error exit -----
        go to 9990
 9000   write(0,*) 'Error -> Exit'
 9990   stop
        end
        subroutine pad_to_blank (isize,charvar)
c
        character*(*) charvar
c
        do 1000 i=1,isize
           charvar(i:i) = ' '
 1000   continue
 9990   return
        end
```

NOTE

The arguments shown for the fortran interface routines are the arguments that should appear in the Fortran call to the routine. The argument list in the interface routine itself includes an additional item for each character string in the list. These additional arguments are integer values containing the length of the character string. As with the unseen underscore added to the name, these arguments are added by the f77 compiler. For example:

The interface routine sbsnr_aadd_.c - is actually coded as follows:

```
/*
 * Copyright 1990 Science Applications International Corporation
 *
 *
 * FILENAME
 *     sbsnr_aadd_
 *
 * DESCRIPTION
 *     Routine to interface fortran code to sbsnr_Aadd
 *     and insert an array of records into an sbsnr structured table.
 *
 * ARGUMENTS
 *     Arguments are the same as for sbsnr_Aadd with the addition
 *     of structure elements as individual arguments, string length of
 *     table and size of character array attributes.
 *
 * RETURN VALUE
 *     sbsnr_Aadd return value is passed on.
 *
 * CALLED BY
 *     Fortran application code
 *
 * SEE ALSO
 *     libdb30.3
 *     libdbims.3
 *     array_insert.3
 *     dberror.3
 *
 * AUTHOR
 *     generated by machine
```

```
 *      See:    Mari Mortell
 *
 * SCCSId:      @(#)sbsnr_aadd_.c       1.1 3/4/91 Copyright 1990 Science Applicatons International Corporation
 *
 */

#include "db_sbsnr.h"

int
sbsnr_aadd_(table, recnum, qa_flag, arid, sta, chan,
        stav, ltav, lddate, len_f_table, len_table, len_sta,
        len_chan, len_lddate )
char    *table;
int     *recnum;
int     *qa_flag;
long    *arid;
char    sta[][6];
char    chan[][8];
float   *stav;
float   *ltav;
char    lddate[][17];
int     *len_f_table;
int     len_table;
int     len_sta;
int     len_chan;
int     len_lddate;
```

FILES

../../../libsrc/libdb30 Must follow in link path.

SEE ALSO

libdb30.3
array_insert.3
array_fetch.3
dberror.3

NOTES

The routines apmaadd, detadd, fkdadd, fsdadd and sbsnradd are being replaced with apma_Aadd, detection_Aadd, fkdisc_Aadd, fsdisc_Aadd and sbsnr_Aadd. The routines apmaadd_, detadd_, fkdadd_, fsdadd_ and sbsnradd_ are being replaced with apma_aadd_, detection_aadd_, fkdisc_aadd_, fsdisc_aadd_ and sbsnr_aadd_. locadd is being obseleted. Backwards compatibility is being maintained for an interim period. The sigpro_time routines will be replaced with the timestamp routines.

The *in_arrivaladd, in_assocadd, in_origerradd, in_originadd, out_arrivaladd, out_assocadd, out_netmagadd, out_origerradd out_stamagadd, out_originadd* and their respective interfaces to fortran act on synonyms of the core database relations. These add routines have been replaced with the array add routines in libdb30. These synonym names are passed into the dynamic table name. This also results in the new data checking procedure being utilized. Currently, these files simply pass arguments to the respective array add routine. See *libdb30/array_insert.3*

WARNINGS

AUTHORS

Jean Anderson, Mari Mortell, Donna Williams, Karen Garcia

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 1993 April 12	3. REPORT TYPE AND DATES COVERED Special Technical 11/27/91-3/21/93

4. TITLE AND SUBTITLE
The IMS Software Integration Platform

5. FUNDING NUMBERS
MDA972-92-C-0026

6. AUTHOR(S)
J.W. Given, W.K. Fox, J. Wang, T.C. Bache

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Science Applications International Corporation
10260 Campus Pt. Drive
San Diego, CA 92121

8. PERFORMING ORGANIZATION REPORT NUMBER
SAIC-93/1069

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 N. Fairfax Drive, #717
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release;
Distributed Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The *Software Integration Platform* (*SIP*) supports automated and interactive distributed processing for the Intelligent Monitoring System (*IMS*). The *SIP* addresses the practical problem of integrating existing software and data archives into a processing system distributed on a network of UNIX workstations. It consists of software components that are widely applicable to other scientific data-processing operations. The *SIP* is divided into two subsystems. The Data Management System (DMS) manages shared data stored in archives distributed over a wide-area network. The Distributed Applications Control System (DACS) handles inter-process communication (IPC) and process control for user applications distributed over a local-area network. The data archives managed by the DMS consist of commercial relational database management systems (RDBMS) supplemented by UNIX file systems. User applications access data stored in the archives through the

14. SUBJECT TERMS
Distributed processing, Inter-process communication, Software integration, Data Management, Relational Database

15. NUMBER OF PAGES
79

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	NONE

Data Management System Interface (DMSI). The DMSI allows global access to data independent of a specific physical archive. Because *IMS* requires the capabilities provided by commercial RDBMS products, the DMSI includes extensive support for these products. The DACS divides the IPC and process-control services between two applications. The CommAgent provides message routing and queueing. The DACS Manager interprets the IPC and monitors local-area network resources to decide how and when to start processes. Working together, these applications isolate user applications from network-specific details. The messaging facilities of the DACS enable the distributed system to exploit concurrency and pipelining to expedite data processing. The Process Manager is a general application developed for the DACS that manages the processing of data through complex configurable sequences of user applications. All components of the *SIP* exploit commercially available software products and anticipate current trends in software-product development.