Global 16-bit Development System
Speedbase Manual
Version 8.1

TABLE OF CONTENTS

APPENDICES

Appendix Description Page Number

0. Foreword

This manual describes the Speedbase Development System, a high-level programming language and associated utilities for use by developers of commercial application products.

Chapter 1 introduces Speedbase, explaining the capabilities and underlying concepts of the language. Chapter 2 provides a summary of the facilities of the Speedbase database management system. For a full account, see the Speedbase Presentation Manager Manual. Subsequent chapters then treat each aspect of the language in detail. The utilities are documented in the appendices.

In order to gain a fast appraisal of Speedbase, the following chapters are recommended:

Speedbase Development Language Overview	Chapter 1
Speedbase Database Manager	Chapter 2
Window Division	Chapter 6

The Speedbase Development System is distributed with a sample application system, described in Appendix C. Following a review of the above chapters, we recommend that you study both the source and running frames of this application. The remaining chapters in this manual may then be used for reference purposes.

1. Speedbase Development Language Overview

1.1 Capabilities

The Speedbase Development Language is a high level compiled programming language which allows extremely rapid development of commercial data processing systems. It consists of a number of very high level constructs, which are used to specify interaction between stored data and the user. To make maximum use of the available facilities, this data will normally be stored in a Speedbase database, which provides full relational control and retrieval facilities.

Speedbase has dramatically expanded the functionality of traditional development languages such as Cobol, while maintaining upward compatibility. Speedbase encompasses the functionality of both Assembler and Cobol, and statements from these languages may be interspersed with Speedbase high level facilities.

As a programming language, Speedbase has been aimed at the data processing professional and is not regarded as an end-user product. It has been designed to facilitate rapid development of complex business systems, the design of which would be far beyond the capabilities of most computer end-users. Speedbase may, of course, be used to develop relatively simple applications, but its main strength lies in its ability to control and integrate large volumes of diverse business data.

This data control function is performed by a run-time environment known as the Speedbase Presentation Manager, a network structured dbms integrated with a sophisticated windowing system which has been designed and optimised specifically for use on mini and microcomputer systems. All frames written using the Speedbase Development Language are automatically interfaced to the Speedbase Presentation Manager which is present whenever Speedbase frames are executed.

Speedbase provides the following capabilities:

- High level window constructs are used to define operator dialogue and provide a powerful, consistent and simple end-user interface. This includes features such as window overlays, POP-UPs, dynamic function key facilities, split-screen scrolling, partially scrolled records and automatic window sequencing;

- Automatic enquiry mode facilities allow data records to be retrieved by any of up to sixteen indexes for each record type. These facilities operate within all application frames using the database;

- High level Print Format construct defines report layouts. This construct defines the print layout and automatically generates the MOVE statements required to assemble the print line. Spooling, page throw conditions and printing restarts are also handled automatically;

- Instant access by each frame to up to four complete databases, each containing up to 36 separate record types (files) and up to 90 indexes. Database open and close procedures are automatically performed by Speedbase;

- Fast database accessing verbs including READ, FETCH and GET. Retrieval of data records may take place either directly or via each of up to 16 indexes associated with each record

type. FETCH FIRST, LAST, NEXT, and PRIOR are supported, allowing records to be retrieved in both ascending and descending index order;

- Speedbase supports three generations of language constructs in a single development environment. A single source frame can contain assembler instructions (2GL), Cobol instructions (3GL), and fourth generation (4GL) constructs such as windows;

- Fast, single pass compiler containing an integral linkage editor, which directly generates executable frames;

- Full Data Division implementation including Cobol FD constructs to allow interfacing to traditional indexed and relative sequential files. BASED variable declarations are also supported for system programming applications;

- Frames written using Speedbase are automatically multi-user, with record level locking. Speedbase applications may be implemented on either micro or mini computers providing simultaneous access to more than a hundred screens. Over sixty different computer hardware systems are currently supported, and both source and object frames **and** user data are 100% compatible across this entire range;

- Speedbase supports local area network (LAN) implementations, and allows databases to be dispersed across networks with up to twenty-six file servers, supporting hundreds of screens. Record level locking is supported across the entire network, thus allowing LAN implementations without frame modifications;

- Speedbase supports the storage of its database in a series of C-ISAM files in a Unix file structure. When stored in this form the database may be accessed by other Unix application programs written in languages such as Informix. A program produced by the Speedbase Development System may access databases in either the usual Global System Manager (GSM) format or in the Unix format, or both.

1.2 Basic Concepts

Speedbase application systems consist of programs known as **frames**. Each frame is an individually loadable object program, which will perform a particular task within the application, such as data entry and maintenance of a particular record type, or the production of a report. Each frame will usually require access to one or more Speedbase databases, and these are normally opened by the Speedbase menu program before execution.

Each Speedbase frame will usually consist of one or more **windows** in which data will be presented to, and accepted from the operator. Windows are used to present a particular view of the data stored within a Speedbase database, and provide enquiry, data-entry and maintenance facilities for this data. Complex frames are simply built up from a series of these windows, each window performing some aspect of the overall task.

For example, an order entry frame might consist of three windows. The first window would be used to select a particular customer, the second would allow data-entry of order header details such as delivery address, and the third window would allow entry of individual order detail lines. Each window would normally also provide enquiry and update facilities. More complex tasks can then be built up from even longer series of windows.

Windows are specified by using non-procedural code, and quite complex applications can be produced in this way. It is quite usual, however, to code procedural instructions to perform additional specialised processing, such as application specific validation routines. Entry-points are provided to allow this during many processing stages. It is even possible for procedural code entirely to "take over" control of the frame.

When procedural logic is required, it may be written using traditional Cobol statements. Speedbase also provides assembler statements, which may be used for highly technical programming tasks. All of these statements can be coded within the same source frame, and this gives the programmer an extremely flexible development environment.

Owing to the power of the Speedbase high level facilities, frames tend to be small. A typical Speedbase frame would take up less than two pages of code. A source program file therefore normally contains many individual frames, which are compiled together. The Speedbase compiler produces a single executable object file for each frame in the source program. These object frames can then be executed after any required databases have been opened.

The Speedbase compiler contains an integral linkage editor, which is used to link any required system routines into the object frames. It is usual, however, for these routines to reside in a **service module** which is automatically loaded when the frame is executed.

The service module contains most of the system routines normally used by frames, such as the routines used to perform database I/O and screen manipulation. A large fraction of the service module consists of a series of page-able monitor overlays which although permanently resident are outside the user memory space.

Most Speedbase frames will therefore easily fit into available memory. Particularly large frames can, however, be segmented using a feature called **dependent frames**. Dependent frames can automatically access all data items and I/O channels resident in higher levels of the overlay structure, and need only a single statement to be specified. This feature allows complex overlay structures to be established during the course of development, and needs little prior planning or design.

Speedbase also provides high-level facilities for the production of reports. Using the PF construct, complex reports can be produced with very little effort. The **PF construct** generates print-lines automatically, handles page throws, provides for spooling, and takes care of a variety of error conditions. This construct will normally be used for the development of more complicated reports, more straightforward listings being produced by **Speedquery**, the Speedbase Query System.

Speedquery provides a generalised screen-driven method to access the information stored on a Speedbase database. Speedquery allows end-users to view data in quite sophisticated ways, and to produce either on-line or printed output. This same facility can also be used by professional developers to produce pre-formatted enquiry and report frames.

The most striking feature of Speedbase applications is their consistent "look and feel". All frames operate using the same powerful conventions, and provide the same operator facilities. This means that once an operator has learned to use a single Speedbase frame, little effort is needed to understand the use of the rest of the application, or any other Speedbase application.

1.3 Introduction to the Language Structure

Each Speedbase frame is made up of six main sections:

Frame Header Area identifies the frame, and defines the order in which the frame will be executed.

Data Division is used to create database I/O channels, and to define other variables used by the frame. Reports are also defined here using the PF construct.

Window Division is used to define the screen formats by which the data will displayed, and accepted from the operator. It may contain many individual windows, which are called at various stages of processing. The windows are normally invoked automatically when the frame is run, and each window is executed in a predetermined sequence.

Procedure Division contains procedural instructions which may be used to define any task. When coded, the division completely takes over control of the frame, and may be used to execute windows under application frame control.

Load Division provides a convenient entry-point from which frame initialisation tasks may be performed.

Unload Division provides a convenient entry-point from which any frame termination tasks may be performed.

Of the above sections, only the frame header area is required and all other sections are therefore entirely optional.

When a frame is executed, the Load Division, if present, is executed first. This allows the programmer to introduce application code to perform any special initialisation tasks.

If a Window Division has been coded, this is normally executed next. This causes each window to be entered in turn. Within each window, the operator may perform operations such as record addition, maintenance and enquiries.

If, however, a Procedure Division has been coded this is executed instead of the Window Division. Any windows coded within the Window Division can then be executed directly under the control of the Procedure Division.

The Unload Division is executed on completion of either the Window or the Procedure Division. This division can be used to perform any termination task, such as completing a complex transaction update.

The Frame Header and Data Division are therefore non-procedural. The Window Division is partly procedural, in that it allows instructions to be coded within a special section of each window called the Routines Section. This section is used to perform specialised tasks such as field validation, and non-standard updates. The three remaining divisions (Procedure, Load and Unload Divisions) are wholly procedural.

A typical frame will normally contain simply the Frame Header Area and Window Divisions. The header area would be used to create I/O channels for any record types referenced in the database dictionary, and all other processing would be controlled within the Window Division. Speedbase frames usually contain few procedural instructions and quite powerful frames can be produced using no procedural code at all.

Where traditional procedural statements are needed, they can be introduced into the source program at many points. Normally Cobol-like statements would be used but, if necessary, assembler code can also be used. This ability to "mix and match" traditional code with fourth generation constructs provides an extremely flexible development environment. Speedbase frames are therefore used in a wide variety of environments, ranging from system software applications to the most complex of commercial systems.

2. Speedbase Database Manager

2.1 Introduction

Before effective use can be made of the Speedbase Development Language, an in-depth understanding is required of the Speedbase Database Manager. Many functions of commercial application systems can be handled automatically by designing systems to take advantage of the new facilities offered by Speedbase. This chapter should therefore be considered essential reading for all potential designers of systems to be written using Speedbase.

2.2 Background

Before addressing detailed issues, it is worthwhile examining the role of commercial data processing systems generally. Most organisations, irrespective of size or ultimate purpose, are heavily involved in the exchange and manipulation of information. As the size and complexity of organisations has grown, so has the need to process accurately an increasing volume of data.

The difficulty of processing manually such large volumes of data has led to the introduction of computerised information systems. And as organisations have introduced and benefited from such systems, increased competitiveness has placed their rivals under pressure to follow suit. This process has been so effective that few commercial organisations are now viable without heavy reliance on computerised data processing systems.

The demand for further improvements in information systems has, however, by no means diminished. Only a few years ago, systems based on batch processing, providing weekly or even daily turn-around, were considered quite adequate. Today, developers are under intense pressure to produce systems that not only provide instantaneous access to up-to-the-minute data, but also permit that data to be viewed in a variety of different ways, at a variety of differing levels. To further complicate matters, the race is also on to integrate many different types of information into this schema, to provide a complete model of the organisation to which the data belongs.

It is not impossible to provide these kinds of facilities using traditional computing techniques such as indexed and relative sequential file access methods. The resulting systems are, however, notoriously difficult and therefore costly to produce, and almost impossible to modify in any fundamental way.

At this point consider Speedbase. The essential purpose of any database management system is to provide a framework in which diverse information can be rapidly collected, modified and retrieved in an organised and integrated manner. Speedbase achieves this purpose by concentrating on the natural relation-ships that exist between data.

For example, consider the simple order entry system in Figure 2.2a. This example contains three data relationships between four data records.

The customer record and order header record are related in the sense that one customer may have a number of outstanding orders. Conversely, a group of orders exists that belong to a given customer. This one-to-many relationship is referred to as a master/servant relationship (i.e. the customer record is a master to a group of order header records).

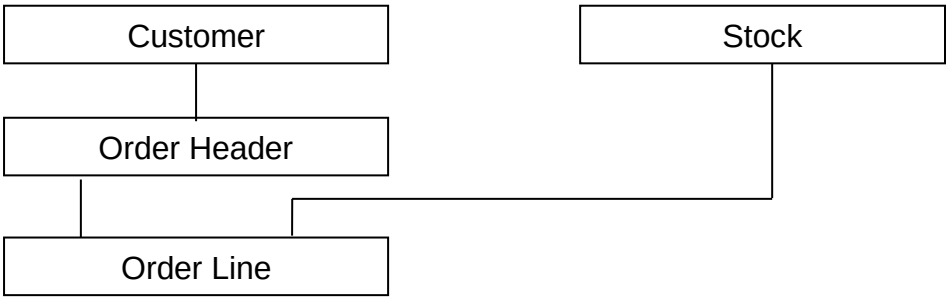


Figure 2.2a - Simple Order Entry System - Record Relationships

A similar master/servant relationship also exists between the order header and order line records (i.e. an order is composed of a number of line items). This relationship also exists between the stock and order line records.

There is nothing new about these kinds of data relationships, they pre-date the invention of the computer by thousands of years. In traditional DP systems these relationships are implied by processing rules (i.e. procedural code). An implementation using traditional methods would probably call for the creation of four separate files, each containing one record type. Copious quantities of code would then be produced to:

- Open and close the files, checking that the versions of each file are correct and dealing with exception conditions.

- Ensure that orders can only be placed on pre-existing customers (i.e. that order lines refer to existing orders and stock items).

- Ensure that stock and customer records cannot be deleted while orders are still active, etc., etc.

Most of this code would need to reside in each program to guarantee data integrity. A database management system permits of a different approach. Once the relationships between various records in a database have been defined, the responsibility for maintaining this kind of data integrity is completely assumed by Speedbase. This removes considerable functionality from application frames, making them easier to create and modify.

It is a mistake, however, to regard Speedbase as merely a new technique for simplifying application programs to reduce their size. The emphasis is on treating an organisation's data as a common, highly-integrated resource. Simply using Speedbase to automate individual applications may be extremely useful, but it is unlikely to maximize the potential of the database approach.

2.3 Speedbase DBMS Overview

The Speedbase database management system controls the storage, retrieval and modification of data in an unlimited number of databases. It forms part of the Speedbase Presentation Manager and is permanently resident in each user partition requiring database access. The Speedbase Presentation Manager performs four major functions:

2.3.1 Relational Integrity

Speedbase ensures that the integrity between related records is maintained by suppressing invalid I/O requests. Roll-ups of numeric data between related records are automatically performed using the GVA (Group Value Accumulator) / GVF (Group Value Field) facilities as described in section 2.6.3.

2.3.2 Index Management

Each record type may have up to sixteen separate indexes. As records are added, deleted or modified these indexes are dynamically maintained by Speedbase. All indexes are bi-directional and may therefore be read in ascending and descending order.

2.3.3 I/O Support

Speedbase supports ten main I/O verbs:

WRITE	Add a record to the database
REWRITE	Modify an existing record
DELETE	Logically remove a record from the database
FETCH/READ	Randomly retrieve record via any index
FETCH/READ NEXT	Retrieve next record sequentially via index
FETCH/READ PRIOR	Retrieve prior record sequentially via index
FETCH/READ FIRST	Retrieve first of group of records via index
FETCH/READ LAST	Retrieve last of group of records via index
GET	Relative (direct) record retrieval
UNLOCK	Relinquish record lock

The FETCH and READ verbs differ only in their treatment of related records. The FETCH verb will retrieve only the target record, whereas the READ verb will retrieve the target record with all of its associated master records. Variations of these verbs are also provided to retrieve a specific servant record set (e.g. all invoices for a specific customer).

2.3.4 Utility Support

Utilities are provided to support database backup and restore, rebuild and re-organisation, conversion, status and size estimation functions. These utilities are documented in the Speedbase Presentation Manager Manual.

2.4 The Database Life-Cycle

A Speedbase database is initially defined using the Speedbase dictionary maintenance utility $SDM, documented in Appendix F. The dictionary stores information about the contents of each data record, the relationships data records may have to each other, and the indexes associated with each record.

Using this utility, a database may be defined to contain up to thirty-six separate data record types of varying lengths. Each record may have up to sixteen separate indexes associated with it, up to a maximum of ninety indexes per database. In addition, each record may act as a servant record to up to sixteen other master records, allowing a comprehensive network of relationships to be implemented.

Once the dictionary has been produced, a new, empty database may be created using the Speedbase database generation utility. This establishes the database on one or more devices, and creates internal database linkages. Few practical limitations apply to the size of the created

database. A small test database may be created occupying as little as 100 Kbytes, while its production version might easily exceed 150 Mbytes.

Before records may be written to, or read from the database, it must be opened, and this is normally done by a menu program. A menu program can open up to four databases before transferring control to application frames. Since the databases will already be open when each application frame is run, there are usually no file open/close overheads incurred when frames are loaded. This leads to exceptionally fast application loading, and improved response for the user.

When a database dictionary is initially created, it is assigned a generation number. Whenever changes take place to the database design, the dictionary is assigned a new generation number, which the compiler includes in all frames requiring access to the data-base. This is checked against the generation number of the database opened at run-time, which ensures that frames match the database in use.

A new generation number is automatically assigned by the Speedbase dictionary maintenance utility when any data record layout is modified, added or deleted, any index is modified, added or deleted, any change takes place to relationships between records within the database, or any GVA/GVF relationship is added, changed or deleted.

The Speedbase generation utility uses the new dictionary either to create a new empty database, or to convert an existing database to the new format. It is important to note that when this is done, all frames accessing the new database must be recompiled before they can be executed.

2.5 Indexing Capabilities

Each record may be indexed by zero to sixteen separate indexes consisting of an optional primary, unique index and up to fifteen secondary indexes. Each index in turn may be composed of up to eight separate fields which do not have to be contiguously located within the data record.

As records are added to the database, Speedbase automatically maintains these index structures. It also incrementally re-organises indexes as the need arises, thus avoiding overflow chains and corresponding performance degradation. A Speedbase database therefore does not require frequent re-organisation.

Primary index keys are regarded by Speedbase as uniquely identifying a given record. The optional primary index key is used to establish and maintain record relationships, and may therefore not be modified. Any attempt to modify a record's primary index key during a rewrite will cause the offending frame to be aborted. Secondary indexes, however, may be modified, and always allow duplicate entries.

Index keys may contain computational fields, but when used must **not** be negative. The key values are ordered in strict ASCII collating sequence, and the introduction of negative computational numbers will therefore not provide the expected index order. Index keys may not start with a high-values byte (i.e. #FF).

All indexes maintained by Speedbase are bi-directional (i.e. may be read forward or backwards) using the FETCH NEXT and FETCH PRIOR statements.

2.6 Record Relationships - Masters and Servants

Speedbase allows master/servant relationships to be defined between record types within a single database. Relationships between records may be established within a network structure (i.e. each servant record may be linked to up to sixteen master record types). The following example shows how the various record types in a simple order entry system might be related:

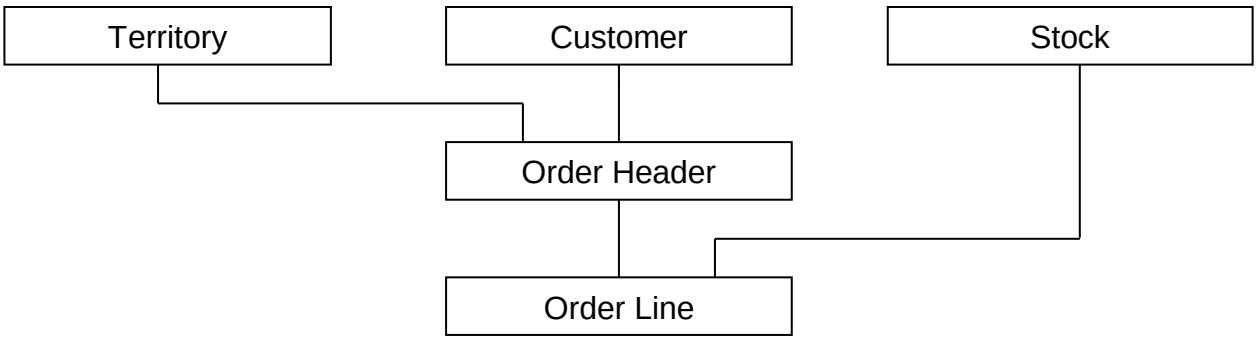


Figure 2.6a - Master/Servant Record Relationships

In the above example, each customer may have a number of orders, thus indicating a master/servant relationship between these records.

A similar relationship also exists between the territory record and the order header record. Each order header record may in turn have a number of order line records. The order header record therefore acts as a **servant** to the territory and customer records, and as a **master** to the order line record. The servant records relating to a specific master (e.g. order lines within an order header) are called a **servant set**.

The masters relating to a particular servant may be accessed implicitly using the READ statement, or explicitly using FETCH or GET statements. The servant set relating to a specific master are retrieved by a FETCH statement using an appropriate primary or secondary index.

When a servant record is linked to a master, the fields making up the master's primary index must also be stored on the servant. These fields, as stored on the servant record, are called the **master access key**. The master access key defines which **particular** master record the servant belongs to. This same key is also used by the Speedbase rebuild utility to relink records during total database reorganisation.

The relationship between a master and servant record is **not** fixed. Servant records may be relinked to new masters as the need arises, and this is achieved by simply amending the master access key**.** Speedbase then automatically unlinks the record from the old master and links it to the new one. Defining relationships between records has the following effects.

2.6.1 Linkages

Speedbase establishes pointer fields on the servant record which point directly to each of its master records. This linkage is direct and allows for extremely fast access to these master records. Using the READ verb, a servant record may then be read with all of its master records in one statement. This ease of access often avoids the need for duplication of data at the servant record level.

This feature is particularly useful during serial processing of records (e.g. during report printing). Rather than writing procedural code to fetch related records at key breaks, processing may proceed at the lowest level servant record, and corresponding master records are automatically retrieved.

2.6.2 Integrity

Where a relationship is specified, Speedbase ensures that master records are always correctly linked to associated servants. A servant record simply cannot be written to the database without the prior existence of appropriate master records. Similarly, a master record cannot be deleted until all of its servant groups are empty. These checks ensure that application frame errors are trapped and corrected at development time.

2.6.3 GVF/GVA Processing

Numeric fields residing on servant records can automatically be accumulated into one or more corresponding fields on associated master records. The numeric field stored on the servant record is known as a GVF, Group Value Field. This field is added automatically to one or more GVAs, Group Value Accumulators, when a WRITE, REWRITE or DELETE operation takes place. GVFs are normal computational numeric fields stored on the servant record. They may be accessed and modified as required.

Speedbase automatically maintains GVA fields to ensure that they equal the sum of the corresponding GVFs. During a REWRITE instruction, GVA fields are modified to reflect any changes in the value of GVFs.

GVF/GVA processing is also performed when a record is relinked to a new master during a REWRITE. In this instance the value of the GVF field is transferred to the new master at the same time as new linkages are established. Speedbase will also automatically handle the instance where a record is not only relinked to a new master, but GVF values have also been changed.

GVF/GVA processing is handled by Speedbase at a system level and is therefore extremely quick. The construct has very wide applicability and may be used to replace explicit procedural updates during data entry and updates, which could normally only be performed by batch processes.

Note that GVA fields are held in a special systems area of the record, see section 2.9.4, whereas GVF fields are part of the user area of the record. A field may not therefore be both a GVF and a GVA.

In the above example, GVF/GVAs could be usefully employed to:

- total units on order (order line GVF to stock record GVA)
- total value of order (order line GVF to order header GVA)
- total orders outstanding (order header GVF to customer GVA).

GVA values are automatically re-established by the Speedbase rebuild utility. This means that complex application recovery routines do not have to be written for each new system. Where possible it is therefore advantageous to use GVF/GVA constructs in preference to explicit updates.

Care must be taken when designing a database that the GVA values of master records cannot overflow even when large numbers of servants are added. A GVA field will overflow when the computational size of the field is exceeded. Since this is usually several times larger than the displayed size of the field, a considerable margin for error is implicitly provided. If, despite this, a GVA overflow does occur, the database will need to be rebuilt or restored.

2.7 Multi-User Access - Locking

A Speedbase database may be updated simultaneously by many users. Integrity of the data stored is ensured by the use of locks at record level. Speedbase allows records to be locked in two ways, exclusively and update-protected. These locks are automatically provided whenever a record is retrieved from the database.

Exclusive locks A record can be exclusively locked by only one frame at a time. Exclusively locking a record gives the frame the ability to rewrite or delete the locked record. Indeed, if an attempt is made to rewrite or delete a record that is not exclusively locked, the offending frame will be terminated with a stop code. In the remainder of this manual, the term "lock" designates this type of full, exclusive lock.

Placing a full lock on a record therefore indicates the **intention of the frame to update it**. Since two frames updating a record simultaneously would cause unpredictable results, Speedbase stops other frames from retrieving the record for update purposes. A lock therefore confers **update rights** to the locking frame. Conversely, it removes the ability of other frames to gain update rights while the record remains locked.

Update-Protection A record may be update-protected by several frames at the same time, and for this reason this lock is sometimes referred to as a non-exclusive lock. Placing update-protection on a record does **not** provide update rights to the locking frame, but simply stops any other frame from updating the record. Once protected, the record cannot therefore be re-written or deleted by **any** frame. This type of lock is referred to as a protect lock in the remainder of this manual.

Records should therefore only be **locked** if they are to be updated or deleted. If the only intention is to stop anyone else from updating the record, then the record should be **protected**. This has some implications when processing record structures, which is discussed in the next section.

2.7.1 Locking Record Structures

Some consideration must now be given to processing records that form part of a structure (i.e. are related). This may best be illustrated by use of an example:

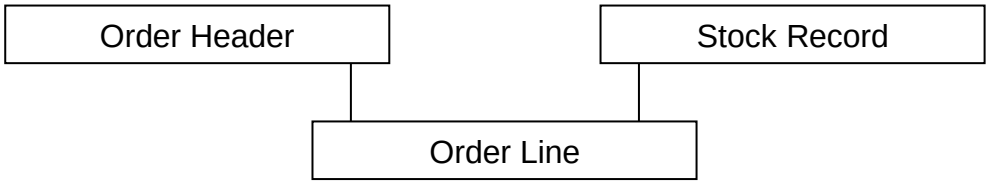


Figure 2.7a - Master/Servant Relationships and Locking

In the above example a master/servant relationship exists between the order header and stock records, and the order line record. Whenever an order line record is added to the database, it is therefore linked to both an appropriate order header and stock record. Which **particular** order and stock records it is to be linked to is defined by the master access keys present on the order line record.

When a new order line record is added to the database, it is therefore necessary to ensure that its associated masters exist. In this example, the order line must relate to an existing order header and be for a product that exists in a stock record. The only way to ensure this in a multi-user environment is to FETCH and LOCK or PROTECT the appropriate records before issuing a WRITE instruction. Simply performing a look-up, without a lock, leaves the way open for another concurrently executing frame to delete the required master record before the WRITE instruction is completed.

To overcome this potential problem, the record must therefore be locked or protected. If the master record will not be explicitly re-written then protection will suffice. This has the advantage of allowing other partitions to add servants to this master at the same time, which is not possible if a full lock is used.

Following successful addition of a servant, the linked master records cannot be deleted. Speedbase maintains a count of the number of servant records linked to each master, and while this count is non-zero, the record is considered to be active. The same considerations apply during a REWRITE. If a master access key is modified prior to a REWRITE instruction, then it is essential that the new master record is locked or protected.

Thus, the golden rule is that master records about to be linked to a servant by WRITE or REWRITE instructions must be locked or protected. This ensures that these records cannot be deleted by another frame during the update process, and avoids the possibility of deadlock occurring. If, during a WRITE or REWRITE operation, Speedbase detects that a required master record has not been locked or protected by the application frame, then a recovery will be attempted.

During this recovery, Speedbase will attempt to retrieve the required master records from the database and lock them so that the update may continue. If this can be done, the update will be concluded successfully, following which the offending frame will be terminated with a stop code.

If this process fails, either because the required master record does not exist, or is permanently locked, then the operation will be terminated unsuccessfully, and the frame terminated with a stop code. The database will be **corrupted** in this event, necessitating a restoration or rebuild. It is important to note that although writing or re-writing a record will normally cause that record to become unlocked, the lock status of any associated master records will be unaffected by the operation.

As discussed, master records about to be linked to a servant record must be locked or protected at the time of the WRITE or REWRITE instruction. No such requirement exists, however, when a servant record is deleted. Unlinking a master record as a result of a delete, or re-write when a master access key is changed, does not require the master record to be locked in any way.

Exclusively locking a given master record will therefore prevent other partitions from **adding** further servant records to its group, **but will not prevent deletions** of records from that group. Given that these records may have GVF options, this means that GVAs residing on a master record may be modified by Speedbase, even while locked or update-protected.

The system area of a data record, which contains all GVA fields, is therefore in effect treated as a separate record and managed entirely under the control of Speedbase. This strategy has been implemented to reduce the number of concurrent locks required during typical update sequences, and thus optimises access to the database during extensive multi-user operations.

2.7.2 Record Retrieval Locking Options

Whenever a record is retrieved it is usually fully locked so that other partitions are unable to update it. This avoids the possibility of two users simultaneously reading, modifying and updating a record, which would cause the modifications made by one of the users to be lost.

It is therefore necessary to specify whether a full lock or just protection is required when a data record is retrieved, and this is achieved by means of the NOLOCK, PROTECT and RETRY clauses. These clauses may be specified on each of the record retrieval verbs (i.e. READ, FETCH and GET) and are described as follows.

NOLOCK instructs Speedbase that the target record is not to be locked or protected. The record is therefore retrieved regardless of its lock status. If it is retrieved using the NOLOCK option, it cannot later be rewritten.

PROTECT indicates that update-protection is to be placed on the record. If successful, the record will be protected following retrieval, which ensures that it cannot be re-written or deleted by any other frame.

RETRY instructs Speedbase to retry retrieval following a locked record condition. It is coded:

RETRY *n*

where *n* is a numeric integer in the range -1 to 127, which indicates the number of retries to attempt in the event of a locked record condition.

If neither the NOLOCK nor the PROTECT clause is specified, then the record will be retrieved with a full, exclusive lock thus conferring update rights to the retrieving frame.

A locked record condition can arise in only two ways, when a protect lock is requested and the record is already fully locked, and when a full lock is requested while the record is protected **or** fully locked. In this event, Speedbase will wait and retry the operation a number of times as specified by the RETRY clause.

If *n* is positive, Speedbase retries the retrieval operation that number of times. On each retry Speedbase will display a LOCK message, including the name of the locked record, on the base-line of the screen. Each retry will take up to approximately two seconds, depending on system loading. When the number of retries specified has been exhausted, an exception condition will be returned which must be trapped by the frame using an ON EXCEPTION statement.

If the number of retries coded is -1, this indicates to Speedbase that no exception logic has been coded to deal with the condition. RETRY -1 therefore results in an indefinite number of retries which will continue until the operation is successfully completed, or the frame is aborted. RETRY -1 must therefore be used with great care, since its indiscriminate use can lead to frame deadlocks. If neither the RETRY nor NOLOCK clauses are coded the retrieval instruction defaults to RETRY 8.

The NOLOCK, PROTECT and RETRY clauses thus allow a variety of conditions to be dealt with. If the target record will not be re-written, the NOLOCK clause should normally be used. This avoids a frame unnecessarily suspending other partitions requiring the same record. Most reporting frames would normally retrieve records using the NOLOCK option.

If either a full or protect-lock is required but no code has been written to handle the possible record locked exception, RETRY -1 should normally be coded. Under most other conditions, the default RETRY 8 should be appropriate.

2.8 Database Access Statements

This section introduces the database access statements used to perform I/O transfers to and from the database. This section is concerned with the underlying processing of these statements, rather than the precise syntax, which is covered in Chapter 7.

2.8.1 The I/O Channel

Before any input or output operations can be performed for any record, an I/O channel must first be established within the application frame. This channel consists of a record control block, which is similar in concept to a Cobol FD, and a data record area. This data record area is used to perform all transfers of data to and from the database.

I/O channels are established by the ACCESS statement, described in Chapter 4. The access statement is a compiler directive, which causes it to compile an I/O channel into the frame for each designated record type. Using this statement, it is also possible to create multiple I/O channels for the same record type, and this is also discussed in Chapter 4.

The fields within the record area established by an access statement may be referenced by instructions, like the MOVE statement, just like any other data items. The record area is always pre-initialised by the compiler, and will contain binary zeros, decimal zeros or spaces depending on the types of its component fields.

2.8.2 Concept of the Current Position

Speedbase maintains information on the status of each I/O channel within the frame. As data records are written and retrieved from the database, Speedbase establishes a current record position for each record type. This record position is used to determine which record will be retrieved during sequential processing such as FETCH NEXT or PRIOR.

When a frame is first loaded from disk, all the I/O channels are established at an imaginary point one record before the beginning of the data record area. An initial READ NEXT instruction will therefore retrieve the first record in sequence of the specified index. This operation advances the current record position. The current record establishes a position within each of the record's indexes. This position will clearly differ depending on which index is being used. For example, customer number 1, with a name of "Zettech" would presumably be first when viewed through the customer number index, but somewhere near the end of the name index.

All I/O operations re-establish the current position in each of the indexes associated with the data record. It is therefore possible to read a series of records via one index, switching to another mid-stream. The current record position is not affected by unlocking the data record.

2.8.3 Database I/O Operations

Speedbase supports ten basic I/O operations:

DELETE	Deletes currently locked record
REWRITE	Rewrites the currently locked record
WRITE	Writes a new record to the database
READ/FETCH	Retrieves a specific record corresponding to specified index key value
READ/FETCH FIRST	Retrieves the first record with key value equal to or greater than that specified
READ/FETCH NEXT	Retrieves the next record in the sequence of he specified index
READ/FETCH LAST	Retrieves the last record with key value equal to or less than that specified
READ/FETCH PRIOR	Retrieves the preceding record in the sequence of the specified index
GET	Retrieves a record using its Relative Record Position (RRP)
UNLOCK	Release lock from one or all data records

Of the above statements, the UNLOCK statement is not a true I/O operation since no data transfer actually takes place. All other operations affect the current record position as described in section 2.8.2.

The **DELETE** statement allows a previously retrieved and locked record to be removed from the database. This process entails the dismantling of indexes referencing the record, after which the data record is returned to a list of free records. This free record "slot" may then be re-used during subsequent WRITE statements.

If the record being deleted has master records, linkages to these records will also be dismantled. This causes the master record's servant group count field (SGC) to be decremented. Any GVF fields will also be subtracted from the corresponding master record's GVA fields.

The **REWRITE** statement allows a modified record to be re-written to the database. A record that is to be re-written must previously have been retrieved, and must be exclusively locked at the time the instruction is executed. The processing that takes place for this instruction depends on just what modifications have been made to the record. If any indexes, other than the primary

index, have been changed, the pre-existing index entries are removed and new entries built to reference the record.

If any master access keys have been changed, the record is unlinked from the prior masters and linked to the new masters as specified by this key. Such a change may also involve dealing with any GVF/GVA relationships between the re-written record and its masters.

The **WRITE** statement adds a new record to the database. This process includes a check to see if the record's optional primary index already exists on the database, and if this is the case, an exception condition will be returned. Processing continues by allocating the next available free data record slot from a "free list". If none is available, an exception condition will again be returned.

Indexes are then established for the record. Following this, linkages are created to any master records associated with the written record, and GVF fields are added to their corresponding GVA fields.

Before the data record is finally written to the database, the system area part of the record is filled with binary zeros, #00. This ensures that all GVA fields have a zero value when the record is initially written. Any values previously existing in these fields will therefore be lost by this operation.

The above three statements can therefore all cause processing to take place of masters associated with the target record. Such processing may be required to simply update the servant group count (SGC) field on the master's data record, or may be more complex GVF/GVA updates. It is important to note that all the I/O required to process each master record actually takes place within the record area in the appropriate I/O channel. If the DELETE, REWRITE or WRITE statements are to be used, an I/O channel must therefore be established for each of the target record's masters.

The DELETE and WRITE statements always require I/O on all the target record's masters. The REWRITE statement will require I/O on master records if any master access key has been modified or any GVF has been changed. To perform this processing, the system area of each master record is read, updated as necessary and re-written to the database. This process takes place **within the record area** of the appropriate I/O channel for each master, and therefore updates any system area data previously stored there.

The **READ** and **FETCH** statements are used to retrieve records from the database via a specified index. These statements operate identically except for the processing of any associated master record. The FETCH statement will retrieve only its target record, whereas the READ statement will retrieve its target record **and any master records** for which I/O channels have been established.

The master records retrieved by the READ statement are always returned **unlocked** and the instruction will release any previously existing locks on those I/O channels. The main use of the statement is therefore within read-only functions such as reporting, where locks are not normally required.

The READ and FETCH statements may be used for sequential or direct indexed access to the database. READ/FETCH FIRST/LAST statements are used to retrieve the first or last record corresponding to a specified key value. The READ/FETCH NEXT or READ/FETCH PRIOR

instructions may then be used to retrieve records sequentially from the current record position. Records may be retrieved in ascending or descending index key order.

The **GET** statement is used to facilitate direct access to the database using a Relative Record Number (RRN). The RRN specifies the relative position a record occupies in the data record area, and access using this number is therefore direct, without the need for intermediate index look-ups. While the GET statement therefore operates very quickly, it is not significantly faster than sequential processing using the FETCH NEXT statement.

The **UNLOCK** statement is used to unlock a data record which has previously been locked by a successful READ, FETCH or GET statement. The statement is used to relinquish the locked status of a record, allowing it to be updated by another partition. The statement can be used to unlock a particular I/O channel, or can be used to relinquish **all locks** currently in force within the frame.

2.8.4 Retrieving Records within Structures

This section discusses the retrieval of records which form part of a structure (i.e. are related). It explains how the database access verbs can be used to get from one record type to others. Retrieving a given record can be accomplished by means of the GET or FETCH instructions. The GET instruction allows the target record to be retrieved when its RRN position is known. The FETCH instruction is used when one of the record's index key values is known, facilitating access via any of the record's indexes.

Whenever a data record is written to the database it is linked to any master record types declared when the dictionary was created. Links are created on the written record which point directly to the record's masters. When the READ statement is used to retrieve a record, these links are used to retrieve the master records at the same time as the target record. There are three ways of accessing the master records for a given servant:

- Using the READ statement, the target record may be retrieved at the same time as any associated masters. The READ statement first performs a FETCH on the target record, and then retrieves the associated master records using direct links. The retrieved master records will always be unlocked following this operation.

- The primary key value of each master record is always stored on its servant records. Therefore, if the servant record is first retrieved, each master record may then be retrieved using a FETCH instruction via the master's primary index.

- The processing performed by the READ statement may also be implemented explicitly. Once the target servant record has been retrieved, GET statements may be coded using the links stored on the servant. The data names allocated to these links are explained in Section 2.9.4.

Retrieving records at a **higher** relational level in the database is therefore achieved either via the master record's primary index, or directly using the links stored on the servant record.

Retrieving records at a **lower** relational level in the database is always done by use of indexes. If, for example, you need to retrieve all the invoices for a given customer, the invoice record must have an index starting with the customer number, and invoice records are READ or FETCHed by reference to this index.

Another way of stating this is that the master access key must also be the most significant portion of at least one index on the servant record. This index can be any of the record's primary or secondary indexes.

The FETCH and READ statements have been extended to simplify retrieval of servant record groups using indexes. This is achieved by use of the KEY clause within the READ and FETCH statements. The key passed using this clause may be shortened to select only a given servant group. This **short key** facility is explained further in Chapter 7.

2.8.5 I/O Error Handling

Most of the above statements can fail with exception conditions. Where the execution of a statement can result in an exception, this must be trapped in the frame code using an ON EXCEPTION, or ON NO EXCEPTION statement. The various exception conditions which can arise may then be distinguished by testing the system variable $$COND. For convenience all possible exception conditions generated by the database manager have been listed in Table 2.8a below:

Exception Condition	**$$COND**
A data record is found permanently locked during the execution of any READ, FETCH or GET statement. The record has been retrieved and placed in the data record area, but is not locked.	1
Requested record key not found during a READ or FETCH operation. If the instruction is READ/FETCH FIRST, NEXT, LAST or PRIOR, the next record in the sequence of the selected index is returned. Otherwise no action takes place.	2
A READ/FETCH FIRST, NEXT, LAST or PRIOR failed because the requested record key did not exist. The next record in sequence was then found to be permanently locked. This is a combination of both conditions 1 and 2 occurring simultaneously.	3
End or start of file condition has occurred during a READ/FETCH, FIRST, NEXT, LAST or PRIOR. At end of file, the data record area is filled with high values (#FF's). At start of file the data record area is filled with low values (#00's).	4
The DELETE verb has attempted to delete an active record (i.e. a record which is linked to a group of servants). No processing has taken place.	5
A duplicate primary index key would result from execution of the WRITE verb. No processing has taken place.	6
Data area full condition. No free data records are available to honour a WRITE	7

request (i.e. a "file full" condition).	
The GET verb specified a relative record number which is deleted.	8

Table 2.8a - Database Manager Exception Conditions

2.9 Global Database Structure

Unless your computer is running under the Unix operating system, your Speedbase database consists of three, four or five files, stored on disk. If you are running Unix you have the option to store your Speedbase data in the Unix file structure. A Global Speedbase database (i.e. one stored in the Global System Manager (GSM) file structure rather than a Unix file structure) consists of a data dictionary file, a main index file and one, two, or three datafiles. The names of the files are as follows, where *xxxxx* is the database name:

File-id	Description
DI*xxxxx*	Data Dictionary
DB*xxxxx*	Main Index File
DB*xxxxx*1	Data-file 1
DB*xxxxx*2	Data-file 2 (optional)
DB*xxxxx*3	Data-file 3 (optional)

Table 2.9 - Global format Speedbase Database Files

2.9.1 Database Dictionary

The database dictionary contains a description of the fields stored on each record, complete with details of associated indexes and relationships with other records in the database. The dictionary is created by the Speedbase dictionary maintenance utility, see Appendix F.

The five-character database name may vary for different copies of the same database. The database dictionary also contains a database **ID**, which is compiled into frames during compilation. At run-time, this ID, as compiled into the frames, is checked against the ID of the database to ensure that the correct database has been opened, and an error is reported if these IDs do not match. The database dictionary must be located on the same unit as the main index file.

2.9.2 Main Index File

This file contains index data for all records stored within the database. It is created when the database is initially established using the Speedbase generation utility.

2.9.3 Datafiles 1, 2 and 3

These files contain the data records stored within the database. Each file contains one or more contiguous record areas for each record type which are allocated at database creation time. A record area must be wholly contained within a single datafile (i.e. may not span volumes). Only datafile 1 necessarily exists, since all thirty-six possible record types may be accommodated within it. Datafiles 2 and 3 would normally be used to distribute the database over several physical devices, either for performance or space availability reasons. All three datafiles may occupy separate physical volumes to provide optimised access performance.

2.9.4 Data Record Structure

A data record area may contain space for up to eight million data records. Speedbase maintains a list of free data records within this area, which are allocated when new records are written to the database. Records are returned to this free list when deleted, and are eventually re-used.

Data records are normally accessed via an index using READ or FETCH instructions. Direct access is, however, possible using the record's Relative Record Number (RRN). When a record is initially added to the database, it is written to the next free data record "slot" which is identified by the RRN. The record remains in this position until it is deleted, or the database is reorganised using the data-concentration option of the Speedbase generation utility.

In some particularly performance-critical applications it may be desirable to access records by RRN rather than an index. Care should be taken however since the RRN specifies a record location rather than a particular record. In the instance where a record is deleted and the resulting free "slot" re-used by a subsequent WRITE, access by RRN will, of course, retrieve the new record. It should also be noted that RRNs change whenever the Speedbase database is restored or conversion utilities are used.

READ/FETCH NEXT/PRIOR instructions, using an index, will usually provide faster performance. The reasons for this are beyond the scope of this manual but confirm that RRN accessing would not be used in typical commercial applications. Each data record is composed of two distinct parts, the user data area and system area. The user data area contains all fields specified, up to but excluding the first GVA field. The system area contains all GVA fields. Figure 2.9a shows the complete data record layout:

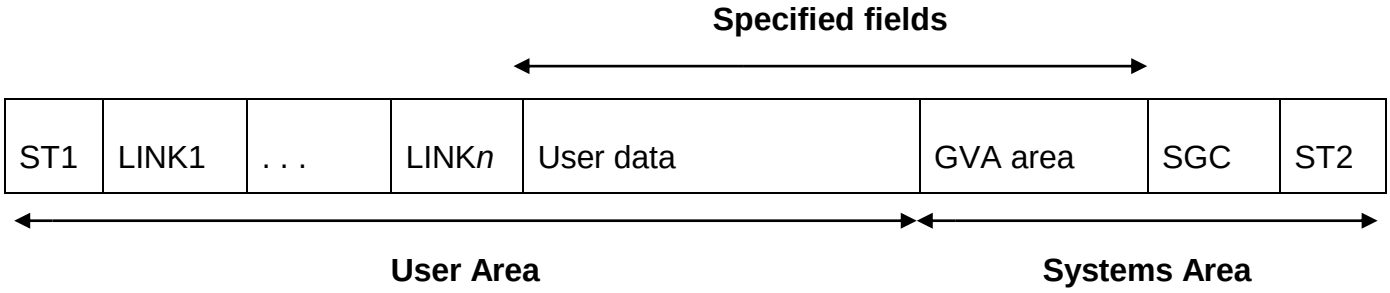


Figure 2.9a - Data Record Structure

2.9.4.1 ST1: Status Code 1 ($*rt*ST1)

All data records begin with a PIC 9(2) COMP status code. This is either positive to indicate the record is in use, zero to indicate the record has never been used, or negative to indicate the record was once in use, but has since been deleted and returned to the list of free records. The status code holds the number of the minor backup cycle during which the record was last modified, and is used to control incremental back-ups. The data-name of this field is $*rt*ST1, where *rt* is the two-character record ID specified in the ACCESS statement. This field may be examined by application frames but **must not be modified**.

2.9.4.2 Link*n*: Link to Masters ($*rt*LNK(*n*))

The Relative Record Number (RRN) of each linked master record is stored on the servant data record. One 9(6) COMP (3 byte) RRN exists on the data record for each master record. A record associated with three masters will therefore contain three 9(6) COMP RRN links, meaning a storage overhead of nine bytes per record. Which master each link relates to is specified when the dictionary is created using the Speedbase dictionary maintenance utility.

These fields may be accessed using the name $*rt*LNK(*n*) where *rt* is the record ID of the servant record and n is the master number as declared using the Speedbase dictionary maintenance utility. These links may be used to provide direct access to master records using the GET verb.

RRN accessing using these direct links has been implemented primarily for system programming purposes and is included in this manual for the sake of completeness only. The implementation of the READ verb, which retrieves a data record inclusive of all master records, performs the same function as a FETCH on the servant record followed by GETs (with NOLOCK) on each of the required masters. Furthermore, the READ verb executes faster than a series of individual GETs.

If, despite this, your application would benefit from using RRN accessing using these linkages, **do not** under any circumstances attempt to modify these links. Speedbase does not check to see if you have, and doing so results in database corruption.

2.9.4.3 Specified Fields

User-specified fields, as defined using the Speedbase dictionary maintenance utility, follow any master links. The size of this part of the record area is determined directly from the picture clauses assigned to each field. An 01 group level field is automatically generated by the compiler which includes all these data items, including GVA's. The name of this group field is the same as the record ID. This field allows redefinitions of the specified fields from within the Data Division, should this be necessary.

2.9.4.4 SGC: Sub Group Count ($*rt*SGC)

This 9(6) COMP field stores a count of the attached servant records (i.e. the number of records to which it acts as master). When this count is non-zero, the record may not be deleted. Any attempt to do so will cause an exception condition to be returned. This count cannot be modified by an application frame.

2.9.4.5 ST2: Status Code 2 ($*rt*ST2)

This PIC 9(2) COMP status code indicates the status of the system part of the data record. The code is either zero to indicate no changes have ever taken place to the system area, or positive to indicate the back-up cycle during which the system area was last updated. Status code value is undefined if the record is deleted (i.e. ST1 is negative).

2.10 Unix C-ISAM Database Structure

If your Speedbase database is stored in a Unix file system it has a special file structure. In this case the Speedbase database consists of a data dictionary file and a schema file, both stored in the usual Global System Manager (GSM) file structure, and in addition, a special index file and several datafiles and index files stored in the Unix file structure. The names of the files are as follows, where *xxxxx* is the database name and *rt*1 to *rtn* are up to thirty-six record types:

File-id	Description	File format
Di*xxxxx*	Data Dictionary	Global
DB*xxxxx*	Schema File	Global
DB*xxxxx*	Special Index file	Unix
DB*xxxxxrt*1.dat	C-ISAM data file for *rt*1	Unix
DB*xxxxxrt*1.idx	C-ISAM index file for *rt*1	Unix

DB*xxxxxrt*2.dat	C-ISAM data file for *rt*2	Unix
DB*xxxxxrt*2.idx	C-ISAM index file for *rt*2	Unix
DB*xxxxxrtn*.dat	C-ISAM data file for *rtn*	Unix
DB*xxxxxrtn*.idx	C-ISAM index file for *rtn*	Unix

Table 2.10a - Unix Speedbase Database Files

The data dictionary is created by the application developer using the dictionary utility of the Speedbase Development System. The data dictionary is stored in the GSM file structure and is identical with that for a Global Speedbase database as described in section 2.9.1.

The schema file is stored in the GSM file structure and is used by the database manager to retain details of the C-ISAM data and index file parameters. It is not accessible to application programs. The special index file is stored in the Unix file structure and is used by the database manager to retain the details required to provide the advanced indexing facilities of Speedbase.

Each database record type is stored in its own C-ISAM datafile. These datafiles are accessible to other Unix software although you must not update (i.e. write to these files) as to do so would interfere with the operation of Speedbase itself. The structure of the data within the datafile is identical with that described in section 2.9.4 for a Global Speedbase database datafile.

Associated with each C-ISAM datafile is a C-ISAM index file. These files are provided for the use of Unix programmers who wish to write software to read the datafiles in various special ways. For example, you might wish to extract name and address information from a customer record in order to do a mailshot using a Unix word-processing product. In introducing C-ISAM indexes you must ensure that you do not interfere with the operation of the Speedbase database manager. For example, you must allow duplicate index entries because these are a common feature of a Speedbase database.

2.11 Restrictions

2.11.1 File Sizes

All file sizes are calculated in 512 byte (0.5 Kbyte) blocks. The main index file may contain up to four million index blocks, giving a maximum file size of 2000 Mbyte. The minimum size of the main index file must be sufficient to accommodate one block for each index managed by the system, and one further block for a control area. A database containing twenty indexes therefore requires a minimum area of twenty-one blocks (i.e. 10.5 Kbyte).

A database may consist of one, two or three further files which contain data records. Data records are located in a contiguous space within these files, known as areas. An area may contain space for up to eight million data records, and each data file is subject to a maximum size of 2^31 - 1 bytes (i.e. the largest number containable in a 9(9) COMP field). This figure again exceeds 2000 Mbyte.

There is, however, a restriction on the placement of areas within a given data file. Speedbase requires that each record area **starts** within the first 32 Mbyte of the data file. The actual size of the record area is then limited only by the maximum file size of 2000 Mbyte.

Since three data files may be generated for a given database, this allows 96 Mbyte (3x32 Mbyte) of data area to be allocated as required. If particularly large record areas are required,

these can be placed at the end of each data file to comply with this restriction. This approach therefore allows three virtually unlimited record areas to be allocated with a total of 96 Mbyte of smaller areas, and is therefore unlikely to cause difficulties in practical implementations.

2.11.2 Record Types/Sizes

A database may contain thirty-six separate record types, each of varying lengths. The absolute maximum record size that may be allocated is 32 Kbyte, but overall frame size limitations would indicate a smaller practical limit. Record sizes exceeding 4 Kbyte are not recommended unless you can be certain this will not cause frame size problems within the application.

The Speedbase rebuild and generation utilities are affected by record sizes, and if very large record sizes are allocated, may not run on target machines with a limited user partition size. This is further detailed in the Speedbase Presentation Manager Manual.

Speedbase allows data records to span physical sectors, and makes no attempt to block data records so that they are exact multiples or divisors of the target disk sector length. If you know the physical sector size of the target computers, some small performance improvement can be achieved by blocking data records. This is done simply by introducing a filler into the data record's definition to expand it to the appropriate length. Note that allowance must be made for the system fields appended to the record by Speedbase - see Section 2.9.

The performance gains achieved by blocking data records are, however, very limited and not normally noticeable in practical implementations. Measured against this must be the increased, wasted storage space and therefore the probability of increased disk head movement during indexed random accessing.

2.11.3 Key Extract Area Limitations

Whenever Speedbase retrieves a data record, it copies certain significant fields from the target record into a special system area. This Key Extract (KE) Area is used to assemble both index and master access keys from their component fields. The KE area is also used to copy the numeric value of each GVF on the data record.

When the data record is subsequently re-written, Speedbase compares the KE area with the record area to be re-written and takes appropriate action. If, for instance, an index key has been changed, Speedbase will delete the existing index entry and create a new one. If a GVF value has been modified, Speedbase will adjust the corresponding GVAs on the master records. If a master access key has been modified, the master record will be re-linked as appropriate.

The actual size of the KE area for a given record is therefore determined by adding the lengths of each index key, master access key and GVF. The following example shows a calculation for a record with three indexes, two linked masters and five GVFs. The index key sizes of the record are 16, 24 and 35 bytes. The primary indexes of the master records are 15 and 10 bytes long. The record has five GVFs, each 9(6,2) COMP (i.e. four bytes each) see Figure 2.11a:

Index, Key, GVF	Segments	Bytes
Primary Index	3	16
Secondary Index 1	2	24
Secondary Index 2	2	35
Master Access Key 1	2	15
Master Access Key 2	1	10

5 GVFs (e.g. each 9(6,2) COMP)	5	20
Totals	15	120

Table 2.11a - Example Key Extract Area

This example record therefore requires a KE area of 120 bytes. During the compilation process, this space will be allocated automatically, as each I/O channel is created.

The maximum size of the KE area is limited in three respects:

Maximum Length of the KE area is limited to 256 bytes.

Maximum Number of Segments is limited to 64. For the purposes of this calculation, each GVF is regarded as a segment. Where a field has multiple GVF relations, this is counted as a single segment. In the example, 15 segments are defined.

Field Placement All index, master access and GVF fields must be amongst the first 125 fields on the data record as defined using the Speedbase dictionary maintenance utility.

2.11.4 Indexation

Each data record may have up to sixteen indexes, up to a total of ninety indexes for all records residing in a database. If, for example, the database contains thirty record types, each may have three indexes. Each index may be composed of between one and eight (inclusive) fields known as index key segments. The total length of each index key must be between one and forty-seven bytes inclusive.

2.11.5 Relations

Each data record may act as a servant to zero to sixteen (inclusive) master records. The number of servant record types that may be linked to a given master record is not restricted.

2.11.6 GVF Fields

GVF/GVA relationships can only exist between related records. The number of GVFs that may be declared for one record is limited to sixty-four as well as overall KE area limitations.

2.11.7 GVA Fields

GVA fields must be declared together at the end of the data record. These fields are included in a special part of the data record known as the system area. The GVA part of the system area may not exceed 127 bytes. This is calculated by simply summing the lengths of each GVA declared for a given record. For example, this would allow a record to contain 25 PIC 9(9,2) GVAs.

2.11.8 Multi-Database Access

Up to four database dictionaries can be specified when compiling a frame, and this therefore allows any one frame access to a maximum of four databases. A menu program is also restricted to opening up to four databases simultaneously.

2.12 Performance Hints

There is one guaranteed way to bring a Speedbase database to its knees and this is achieved by introducing large numbers duplicate index key values. **Under no circumstances** should a database be designed in which any index could contain numerous index keys that are identical. By numerous we mean any number over a hundred, with even fewer if the index keys themselves are longer than say twenty bytes.

When lists of identical keys build up in the index structure, the lack of differentiation between the keys means that higher level indexes cannot be properly built. When this occurs, sections of the index start to behave similarly to overflow chains, displaying similar degradation characteristics. In extreme circumstances, when thousands of duplicate keys exist in an index, write, re-write and delete performance can drop from the millisecond to the minute timescale.

To avoid this situation, simply extend any index that **might** generate substantial numbers of duplicates with some variable that will provide greater key discrimination. For example, it might be useful to index a stock record by say, responsible department. Rather than leaving it there, append the stock number to this index. This will have a significant effect on performance and make the index more useful.

Another important consideration is to ensure that a partial rebuild is performed from time to time to rebuild and optimise the indexes. This is particularly important when the database is large and/or volatile. We recommend that an index rebuild should be performed on a monthly basis, more often if degradation becomes apparent.

It is also important to understand the effect of creating records with multiple indexes, especially when significant numbers of records are being stored. For example, in a record type with 10,000 records each index will typically require four or five random I/O operations to write. So if you design such a record type with sixteen indexes, it is going to take at least seventy-five random I/O operations to write, and this is not going to be instantaneous in a multi-user situation. It is also often forgotten that it will take exactly as many I/O operations to delete the record as to write it.

The effect of linking a large number of masters to a record type is not as pronounced, but should also be considered. Each linked master typically adds two I/O operations to each write and delete operation, but this does not vary with file record volumes.

Taking the above points into consideration during database design will help you to avoid the basic pit-falls, and should give you a high-performance database system.

3. Language Structure

This chapter deals with the structure of Speedbase frames and is arranged in four sections. Section 3.1 describes the elements, such as character sets and symbols, that make up the Speedbase Development Language. Section 3.2 describes how source code lines should be presented, and Section 3.3 describes the physical layout of the various divisions that may be coded within a frame. Section 3.4 describes the processing associated with these divisions and the order in which they are executed.

3.1 Language Elements

3.1.1 Character Set

The character set is an ASCII 8-bit code with the top bit set to zero, see Table 3.1a:

Classification	Members
Digits	ASCII 0 to 9
Letters	ASCII A to Z, a to z, $
Alphanumerics	ASCII 0 to 9, A to Z, a to z, $, -

Table 3.1a - The Character Set

Blank is represented by *b* in this document. Other **special characters** will be introduced later.

3.1.2 Symbols

A symbol must start with a letter and be followed by any number of alphanumeric characters. Normally the first six characters of each symbol must be unique throughout the frame, the compiler ignoring the seventh and subsequent characters, apart from listing them. Symbols are used for data names, section names, paragraph names, entry names, frame names and file names. The letter $ should not be used in a symbol **created** by the application frame since it is employed in symbols used by GSM software. In addition there are eight **reserved words** in Speedbase and these should not be used as symbols. They are:

DEPENDING
FILLER
HIGH-VALUES
LOW-VALUES
NEXT
SPACE
SPACES
USING

Other language words such as IF, PIC and NOT can be used as symbols, although it is recommended that they be avoided in the interests of clarity.

3.1.3 Character Strings

A character string may be made up of any combination of the graphic ASCII characters (i.e. those with a numeric equivalent in the decimal range 32 to 126) When coded, the string appears as:

"*character string*"

For example:

"HELLO WORLD"

is a character string containing the eleven characters:

H E L L O *b* W O R L D

The compiler assumes that frames have been entered on computers with industry-standard tab settings at character positions 9, 17, 25... etc., and any tab characters are replaced with the appropriate number of blanks.

Note that in some countries the ASCII lower case character codes are used for different alphabets, for example Greek or Cyrillic.

3.1.4 Integers

Integers must be in the range -32768 to +32767. The plus sign is optional when an integer is coded.

3.1.5 Numeric Strings

Numeric strings are character strings consisting of:

- optional leading blanks, followed by ...
- an optional + or - sign, followed by ...
- 1 to 15 digits, which may be omitted if the decimal point is present. These in turn are followed by ...
- an optional decimal point which, if present, must be followed by between 1 and 7 digits followed by ...
- optional trailing blanks.

The total number of digits must not exceed 18. Examples of valid numeric strings are:

-3
3.14159
+1246
-0.120
.7

The strings:

3.
3.1*b*4159
+-12

.7-
+*b*9

are **not** valid numeric strings. Programmers familiar with Cobol should note that in Speedbase a string of ASCII blanks is **not** a numeric string, and such a string cannot therefore be used as a display numeric zero.

3.1.6 Standard Numeric Strings

When a number is converted to character form, either for output or for storage in a display numeric variable, it assumes the format of a **standard** numeric string. In such a string:

- leading zeros will always be replaced by blanks (except in the units position)
- the sign will be omitted if positive
- at least one digit will always precede the decimal point
- there will be no trailing blanks
- if the number is defined as fractional, a decimal point and the number of decimal places specified will be printed, even if the value is an integer.

For example, if a field is defined as signed with two digits before the point and two after, then:

3	becomes	*bb*3.00
+02.13	becomes	*bb*2.13
.1	becomes	*bb*0.10
-.2	becomes	*b*-0.20
-21.43	becomes	-21.43

3.1.7 Hexadecimal Strings

A hexadecimal string is coded as a # sign followed by pairs of ASCII "digits" in the ranges 0-9, A-F inclusive. The number of digits in the string must be even, since each digit pair makes up a single byte. For example:

#07

is a single byte string, representing the ASCII bell character, and:

#FFFF

is a two-byte string, with each bit set to 1.

3.1.8 Standard Date Strings

Date items are always stored within the frame as three-byte computational numbers (i.e. 9(6) COMP). When a date item is displayed or accepted from the operator, the format DD/MM/YY is used. The internal representation of this date is established by the calculation:

(YY * 10,000) + (MM * 100) + (DD * 1)

3.2 Source Code Layout

3.2.1 Comments

Two types of comments may be coded, help text and frame code comments. Frame code comments are preceded by an asterisk and may be coded on a separate line or following any language statement. These comments are simply ignored by the compiler.

Help text is introduced by a backslash character \ which must be the first significant character on the code line. Help text must be coded immediately after the WINDOW statement and is treated differently from normal comments in that the text following the backslash character is saved as part of the frame file. This text is displayed as a help window when the user keys <HLP><HLP>.

3.2.1 Statement Format

A Speedbase statement, including comments, consists of a single line of up to 72 characters. Statements may not be continued onto the next line, neither may more than one statement appear on a line. The individual constituents of a statement (e.g. language words, variables, strings and comments) must be separated from each other by one or more blanks or tabs but apart from this consideration the spacing within a line is unimportant. However, the following conventions result in a tidy, readable listing:

Paragraph names and the following should start in column one:

PF
01
77
FD
DATA DIVISION
PROCEDURE DIVISION
LOAD DIVISION
UNLOAD DIVISION
ROUTINES DIVISION
FRAME
ENDFRAME
WINDOW
ENDWINDOW
FORMAT
ENDFORMAT
ACCESS
ENTRY
SECTION

Other statements should begin in column nine, except:

- Statement within the window and PF constructs;
- The level numbers (02 to 49) used in group data definitions. These should be suitably indented to make the data structure clear;
- The VALUE statement, which should be coded underneath the preceding PIC statement. The PIC statement itself should begin in column 33;

- Each TO statement which should be aligned with the TO of its GO TO DEPENDING ON statement;

- Statements within a conditional or iterative structure. These should be indented an additional four spaces for each level of nesting, to highlight the frame structure;

- Comments should start in column 41 in the procedure division, and 49 in the data division, except "across the page" comments which should start in column 1.

3.2.3 The Page Statement

The PAGE statement causes the compiler to skip to the head of the next page on the frame listing. It is coded:

PAGE "*title*"

The statement is coded in the source file on a new line, on its own. When the optional *title* is coded, this will be printed on the current and subsequent pages.

3.2.4 The Copy statement

Copybooks may be included within source frames by the use of the COPY statement:

COPY *name* [SUBSTITUTING "*text*"] [SUPPRESS]

where *name* is the one or two-character alphanumeric name of the copybook to be copied into the frame. The SUBSTITUTING clause allows a parameter string in the copybook to be replaced by the characters specified by *text*. A parameter string is specified within the copybook by coding a string of one or more "&" characters. When the book is copied, these characters are then replaced by the specified text.

The library from which the specified book is to be copied is specified at compilation time. Up to three copy libraries can be processed by the Speedbase compiler. When the same copybook name is present in more than one library, it is copied from the first library in which it is present. The copied book may itself contain further COPY statements, which may be nested to a maximum of seven levels.

3.3 Frame Structure

Speedbase frames are entered using the Speedbase editor $SDE, or another suitable text editor. Each source file may contain 1 to 99 individual units known as **frames**. When the source file is compiled each results in an executable object frame which runs under the control of the Speedbase Presentation Manager. Each is introduced by the FRAME statement and ends with the ENDFRAME statement. The last statement in the source file must be ENDSOURCE. All other Speedbase language statements are optional.

3.3.1 Speedbase Frame Skeleton

The main divisions of a Speedbase source frame file are:

FRAME *frame-id*
[Frame Header Area]

[DATA DIVISION]
[WINDOW DIVISION]
[PROCEDURE DIVISION]
[LOAD DIVISION]
[UNLOAD DIVISION]

ENDFRAME
[.....
More frames
.....]
ENDSOURCE

3.3.1.1 Frame Header Area
The Frame Header Area contains options which control various aspects of the frame's execution. Examples of this are the SEQUENCE statement, which specifies the order in which individual frames are to be executed.

3.3.1.2 DATA DIVISION
The Data Division is used to define data-items used by the frame. For example, print layouts may be constructed using the print format (PF) construct.

3.3.1.3 WINDOW DIVISION
The Window Division is used to define interaction with the operator and may contain one or more window constructs. These define the fields to be displayed and accepted from the operator, and may also contain procedural routines to perform specialised processing within each window. The Window Division is normally executed automatically when the frame is run.

3.3.1.4 PROCEDURE DIVISION
The Procedure Division takes control of the frame after initialisation. It contains instructions which define the operation of the frame, and may be used, for instance, to sequence the order of windows displayed. It suppresses automatic invocation of the Window Division, allowing the programmer to assume full control over the operation of the frame.

3.3.1.5 LOAD DIVISION
The Load Division is used to perform initialisation tasks (e.g. the opening of files). Control is transferred to the first statement of the Load Division when the frame is run.

3.3.1.6 UNLOAD DIVISION
The Unload Division is used to perform close-down tasks (e.g. the closing of files). The Unload Division is executed on termination of the frame.

3.4 Control Structure
Frames are executed under the control of the Speedbase Presentation Manager. A menu opens the required databases prior to execution and performs authorisation checking. Thereafter, frames may either return control to the menu, or may transfer control to other frames.

The Speedbase Presentation Manager creates a data area, called $BASYS, in the high memory region of the user partition known as the user stack. $BASYS contains the system variables used by Speedbase frames and the special frame loader which controls the initialisation of frames when they are executed. $BASYS may also be loaded explicitly by a call to the routine

B$LOD, see Section 8.2. Once $BASYS is loaded, it remains on the user stack until the end of the user's session.

Once $BASYS has been invoked, Speedbase frames may be executed directly from the GSM READY prompt, or from any menu. Most frames, however, require access to one or more databases which must be opened before any I/O activity can take place. While it is possible for a frame to open the required databases itself, databases are normally opened **before** frame execution.

Menus provide a convenient way of executing Speedbase frames, ensuring that $BASYS is present, and automatically open up to four databases before executing a selected frame. Once a database has been opened, it remains open until control is returned to GSM. When a series of frames is executed the databases therefore remain open until the last frame completes. Because this eliminates file-open overheads, Speedbase frames can be loaded very quickly.

Once a frame has been loaded, execution proceeds under the supervision of a system program known as the **frame controller** which performs certain initialisation tasks, then causes the various elements of the frame to be executed in the following sequence:

Service Module — containing system routines required by the frame is loaded if not yet present.

Screen Initialisation — The screen is optionally cleared, and the optional screen header displayed.

Load Division — The optional Load Division is then executed. This allows the application programmer to specify further initialisation tasks, such as the opening of FDs.

Main Processing — If a Procedure Division has been coded, it is now executed. Windows coded within the Window Division may be explicitly invoked by procedural statements within the Procedure Division. Otherwise, if no Procedure Division has been coded, the Window Division is automatically executed and the first window invoked.

Unload Division — Following completion of main processing, the optional Unload Division is executed. This allows the programmer to specify termination tasks such as closing FDs.

Frame Termination — The frame controller then performs the termination tasks. If a print file was opened during execution of the frame (by invoking the PF construct), it is closed. Any remaining locks outstanding on the databases are relinquished. Control is then passed to another frame, or a STOP RUN is executed.

If a STOP RUN statement is executed by the frame controller on termination, any open databases are automatically closed and control is returned to GSM. If the frame was executed from a menu program, GSM will normally cause this menu to be re-displayed.

A frame may complete normally, or an exception condition may occur during processing. Exception conditions may be reported by the Load, Procedure and Unload Divisions by executing an EXIT statement with any exception number at the highest level of control (e.g. EXIT WITH 1). If no Procedure Division is coded, the Window Division is executed instead, and this division can also return exception conditions.

Normal completion of a frame occurs when all the divisions complete without reporting an exception, and this is called a **forward exit**. An exception condition reported by any division always causes the frame to be terminated, and this is called a **backward exit** from the frame. The processing that takes place during a backward exit depends on which division reported the exception:

- If an exception is reported by the **Load Division**, the frame termination tasks described above are immediately performed. It should be noted that the Window, Procedure, and Unload Divisions **are not** executed under these circumstances.

- If an exception is reported by the **Window or Procedure Divisions**, the Unload Division is immediately executed. The frame termination tasks are then performed.

- If an exception is reported by the **Unload Division**, only the normal frame termination tasks are performed.

It is possible to cause immediate termination of the frame by executing a STOP RUN instruction at any time. This causes all further processing to be cancelled, and an immediate exit to GSM takes place. Although Speedbase will ensure that the database is properly closed under these circumstances, other termination tasks **will not take place**. The STOP RUN is therefore not recommended as a normal method of terminating a frame, and should be used as a last resort only.

3.4.1 Creating a Chain of Frames

When a frame terminates, control is normally returned to GSM, which in turn usually re-invokes the initiating menu. It is possible, however, to specify that another frame is to be executed instead. On termination, this causes the specified frame to be loaded and executed just as if the operator had returned to a menu, and explicitly run the frame from it.

This can be useful in a number of instances. For example, a batch run may consist of a number of separate frames which must be run in a particular sequence. Equally, a complex data-entry task may be too big to fit into memory, and may need to be segmented into separate frames.

The order in which a series of frames is to be executed is specified using the sequence statement. This statement is coded in the frame header area, and allows two frame IDs to be specified. For example, coding:

SEQUENCE *frame1, frame2*

causes *frame1* to be executed on abnormal completion (i.e. backward exit) and *frame2* to be executed following normal completion (i.e. forward exit). Using this statement, therefore, chains of frames can be built up, each frame calling another in turn.

The sequence statement therefore normally specifies the names of the actual frames to be loaded on termination. It is possible, however, to simply return control to a menu, and this is achieved by coding the special frame-id EXIT. For example coding:

SEQUENCE EXIT, *frame2*

causes an exit to the menu to take place on abnormal completion, and causes *frame2* to be executed on successful completion. If the sequence statement is omitted, an exit takes place following both successful and unsuccessful completion.

When control is passed to another frame using the sequence statement, the databases remain open, but any record locks outstanding are released. The incoming frame will overwrite the memory area occupied by the preceding frame so that data processed by that frame is not accessible to the next frame. The transfer of data between frames is achieved by use of overlayed structures, as described in detail in Section 3.5.

3.5 Managing Overlay Structures

Because of memory constraints, it is sometimes necessary to segment a single function into a number of frames, allowing it to fit into available memory. For example, consider an order entry function which, to overcome this, has been split into three components. Frame one might deal with the creation of order header information, frame two with entry and update of order lines, and frame three with creating factory orders.

There are two ways in which these frames could be organised. As described above, the simplest solution is for each frame to call the next, using the sequence statement. Once the first frame has completed successfully, the second is invoked, and so on.

This approach is quite simple, but has the limitation that no information can be passed. It is also important to note that since each frame will physically occupy the same memory area, the I/O channels are lost as each successive frame is loaded. This means, for example, that the second frame would have to re-read the order header record from the database to continue processing.

3.5.1 Dependent Frames

A more elegant solution involves the creation of a root frame on which the three order entry frames will all be dependent. This root frame contains the I/O channels and other common information to be shared by the dependent frames. The dependent frames are then compiled so as to occupy the memory area immediately following the root frame, so that executing the dependent frames would not cause the root to be overwritten.

Dependent frames are implemented using a variation of the frame statement. For example, coding:

FRAME *frame2* DEPENDENT ON *frame1*

specifies that the current frame, *frame2*, is dependent on *frame1*, which is therefore the root frame. When *frame2* is compiled, the compiler checks the memory region used by *frame1* which therefore requires this frame to be online during compilation.

The memory area of the dependent frame automatically follows on from the root frame. All the symbols defined and accessed within the root frame are automatically global to the dependent

frame, meaning that all variables and I/O channels in the root frame are directly accessible. Processing then commences by executing the root frame. The root frame in turn loads and executes each dependent frame using the EXEC statement.

The EXEC statement causes the specified frame to be loaded and executed. On termination of the dependent frame, its sequence statement, if any, is executed. If the statement has been omitted, control is returned to the statement immediately following the EXEC in the root frame. Otherwise, the dependent frame passes control to the frame-id specified in the sequence statement of the dependent frame, causing the next dependent frame to be loaded and executed.

Returning to the above example, let us assume that the order entry function has now been split into four frames, a root frame, an order header frame, a detail line frame, and a factory order frame. The root frame would contain the I/O channels and other variables that are to be shared between the dependent frames. When the root frame is executed, it would then invoke the first dependent frame using the EXEC statement.

The processing that then follows depends on whether any of the dependent frames have a sequence statement. If this statement has been omitted, control will be returned to the root frame as each dependent frame terminates, and the root frame would therefore need to EXEC each dependent frame in turn.

By using the sequence statement, it is possible for the dependent frames simply to pass control to each other. Control is then returned to the root frame when the entire sequence of dependent frames has completed. For example, consider the following sequence statements coded for each of the three dependent frames:

```
FRAME2 (Order Header)   SEQUENCE EXIT, FRAME3
FRAME3 (Order Line)     SEQUENCE FRAME2, FRAME4
FRAME4 (Factory Order)  SEQUENCE FRAME3, EXIT
```

Control will be returned to the statement following the EXEC command in the root frame in two ways, as a backward exit from FRAME2, or a forward exit from FRAME4. Note that the exit from a dependent frame **never** passes back an exception, irrespective of whether a forward or backward exit actually took place. If it is important for the root frame to distinguish between these, the dependent frame could set a switch in the data division of the root frame before returning control.

This process is shown diagrammatically in Figure 3.5a below:

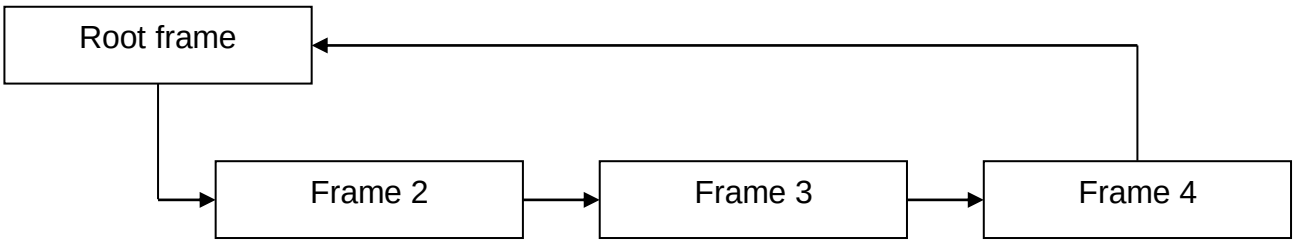


Figure 3.5a Executing a Chain of Dependent Frames

It is therefore important to note that control is not necessarily returned from the frame that was initially EXECed. Dependent frames can be implemented on multiple levels. For example, a dependent frame may itself have further dependent frames and so on. It is important, however, that each frame uses the sequence statement to invoke only frames at the same level in the overlay structure, and that the EXEC statement is used only to call overlays at the next level. This avoids inadvertently leaving memory areas uninitialised.

It is also important to note that recompilation of the root frame **will require recompilation of all its dependent frames**. This is because recompilation of the root may change its size (and therefore the location of variables and entry-points), meaning that the dependent frames will have been compiled using the wrong addresses. It is therefore a good idea to combine the root and dependent frames in a single source file, so that they will always be compiled together as a single unit.

If either of the above two rules is broken, the frame manager will terminate the loaded frame with a stop code.

3.5.2 Controlling Locks in Overlayed Structures

As described earlier in this chapter, locks in force on database records are automatically released when a frame terminates. It should be noted, however, that **only locks on local I/O channels are released** in this way. A local I/O channel is one that is declared using an ACCESS statement in the Data Division of the terminating frame. I/O channels declared in other frames (e.g. a root frame) will not be affected.

In overlayed structures, I/O operations within a dependent frame may take place on I/O channels resident in a root frame. On termination of the dependent frame, these locks **are retained**, and only those locks in force on I/O channels within the terminating frame are released. The locks retained by the root frame will be released when it, in turn, terminates.

This can be put to good effect when designing overlay structures. By placing I/O channels at the appropriate level, I/O channels and their locks may either be released or retained automatically as processing continues. It should be noted, however, that the UNLOCK statement will release locks for both local I/O channels **and** I/O channels defined higher in the structure.

4. The Frame Header

The Speedbase Frame Header consists of a FRAME statement, which introduces the new frame, followed by a number of optional statements. It is coded:

FRAME *frame-id* [DEPENDENT ON *frame-id1*] ["*frame-title*"]
[SEQUENCE *back-frame*, *forward-frame*]
[NOCLEAR] [NOHEADER | SHORTHEADER]
[ACCESS [*dbid*:] *rtid* [*rt2*.... *rtn*]]
[CONTROLLING FRAME]
[SWAP-FILE [*nnnnn*]]

4.1 The FRAME Statement

The FRAME statement introduces the frame and assigns it a name. It is coded:

FRAME *frame-id* [DEPENDENT ON *frame-id1*] ["*frame-title*"]

The *frame-id* specifies the name by which this frame will be known, and is identical to the file-id of the generated object code file. The *frame-id* must start with an alphabetic character, and may be followed by up to five alphanumeric characters. When the compiler detects the frame statement, it opens a new code file on the assigned object unit as specified by *frame-id*. If a file of the same name already exists, **it is immediately deleted**.

The DEPENDENT ON clause is used to indicate that this frame is an overlay which is dependent on *frame-id1*, which must have been compiled using the CONTROLLING FRAME statement. This causes the frame to be compiled at a start address immediately following the root frame. All variables, I/O channels, sections and entry-points defined in the root frame automatically become global to the current compilation. This clause is therefore used for the construction of frame overlays, which is discussed in detail in Section 3.5.

When coded, the optional *frame-title* is displayed as part of the frame header line on the first line of the screen. The *frame-title* is also displayed by the GSM $LIB utility when frames are packaged into program libraries.

4.2 The SEQUENCE Statement

The SEQUENCE Statement specifies the next frame to be loaded following termination. It is coded:

SEQUENCE *back-frame, forward-frame*

where *back-frame* and *forward-frame* are the frame-ids of frames to be executed following unsuccessful or successful completion respectively. The frame designated by *back-frame* will be loaded and executed if the current frame completes unsuccessfully. The frame designated by *forward-frame* will be loaded and executed if the current frame completes successfully.

The frame IDs specified in the sequence statement are not checked by the compiler, since the referenced frames may not necessarily be included in the current compilation. If the frame cannot be found on the program residence unit ($P) at run-time, the offending frame will be terminated with a stop code.

It is important to note that the frame IDs specified do not have to be frames produced by the Speedbase compiler. Any loadable program can be invoked by the sequence statement, and this includes GSM system utilities, programs written using Global Cobol, and job management streams. It should also be noted that the STOP RUN statement ignores the frame IDs specified in the sequence statement, and immediately terminates the frame, returning control to GSM.

When the sequence statement passes control to a new frame, any databases open **will remain open**, although any active locks outstanding will be released. It should be noted that FDs open on termination will also remain open, and any locks explicitly established on these files by use of the LOCK statement will also remain in force. The onus is therefore on the application programmer to ensure that any FDs used are properly closed.

A frame is regarded as having completed successfully if no exception is returned by the divisions coded within the frame. Unsuccessful completion is indicated by an exception being passed back by any of the Window, Procedure, Load or Unload divisions. An exception may be passed back from the Procedure, Load or Unload Divisions by coding EXIT WITH 1 at the highest level of control.

When no Procedure Division has been coded, the Window Division is automatically executed. This causes the first window coded within the division to be invoked. Under these circumstances, an exception condition can be passed back by the operator keying <ABO> at any field within the window. An exception condition can also be returned by application code terminating the window. This is explained in more detail in Section 6.5 of this manual.

The special frame-id "EXIT" may be coded for either the back or forward frame-ids. When this is done, further frames are **not** loaded on termination, and the frame simply performs an unconditional EXIT at its highest level of control. This would normally cause an exit to the menu to be executed. This process also automatically closes any databases opened by the partition.

If the sequence statement has been omitted, an exit to GSM will take place as described above, irrespective of whether normal or abnormal completion occurred. Omitting the statement therefore has the same effect as coding:

```
SEQUENCE EXIT, EXIT
```

It is possible to modify the frame IDs coded in the sequence statement at run-time, and two system variables have been provided for this purpose. The two PIC X(6) variables $BKFR and $FWFR respectively contain the back-frame and forward-frame IDs coded in the in the sequence statement. Either variable may be amended at run-time to a new frame ID, or to the keyword "EXIT" using the MOVE statement. See Section 9.8

4.3 Frame Header Options

Three options may be coded within the frame header to control initialisation and termination processing. These are coded:

```
[NOCLEAR] [NOHEADER | SHORTHEADER]
```

4.3.1 The NOCLEAR Option

The NOCLEAR option suppresses the clearing of the screen during frame initialisation. This allows a screen to be built up progressively by a number of frames executed in sequence. It should be noted that the clear operation is only suppressed by this statement if it is operating in

formatted mode. If scrolled mode processing is in force, the screen is cleared regardless of this option.

4.3.2 The NOHEADER Option

The NOHEADER option suppresses the display of a screen header on the first line of the screen. Normally, a header line is displayed which consists of the following fields:

Frame ID	as coded in the FRAME statement
Frame Mode	mode in which the frame is being executed
Frame Title	title as coded in the FRAME statement
Operator ID	operator's initials as keyed during sign-on
Date	system date, $$DATE

4.3.3 The SHORTHEADER Option

The SHORTHEADER option causes a truncated header to be displayed on the first line of the screen. This header consists only of the Frame ID and the Frame Mode.

4.4 The ACCESS Statement

The ACCESS statement creates an I/O channel for one or more record types residing on a Speedbase database. It is coded:

ACCESS [*dbid*:] *rtid* [*rt3* ...*rtn*]

or:

ACCESS [*dbid*:] *rtid* SUBSTITUTING "*rt2*"

where *dbid* is the ID of the dictionary as specified during compilation, and *rtid* is the name of the record type for which an I/O channel is required. The first form allows I/O channels to be created for a number of records. The second form is used to create an I/O channel for a record type using a different record ID, specified by *rt2.*

The statement creates a record area in a similar manner to the Cobol COPY construct. Details of the required record layout are extracted from the database dictionary as specified by *dbid*. When *dbid* is not coded, the dictionaries are searched in the same order as specified to the compiler. The first record found with the specified record-id is then loaded.

By using the SUBSTITUTING clause, an I/O channel can be created using a record-id that differs from the default record id in the dictionary. For example, the statements:

```
ACCESS DB1:CU
ACCESS DB1:CU SUBSTITUTING "C2"
```

would cause two I/O channels to be created for record type CU, with the second named C2. Each field defined within the record type, and all system variables associated with the record would also start with the substituted record-id, C2. This allows two customer records to be processed independently within the same frame. The use of further access statements for record type CU would permit as many customer records as required to be processed simultaneously.

If an accessed record type is to be written, rewritten or deleted, it is essential that all master records linked to it are also declared in an access statement. This is because the system area

of these master records may need to be updated during database I/O processing. It should be noted that where multiple I/O channels are established for the same record type, any system area updates resulting from servant I/Os will take place in the first declared I/O channel.

4.5 The CONTROLLING FRAME Statement

The CONTROLLING FRAME statement must be coded if the frame being coded will act as a controlling frame. The instruction causes the Speedbase compiler to save the frame's symbol table which is required during the subsequent compilation of frames that are dependent upon it. If the statement is omitted, a compilation error will result during the compilation of subsequent dependent frames.

4.6 The SWAP-FILE Statement

The screen image buffer associated with a pop-up window may be incorporated in the application frame or may optionally be stored in a swap file on disk. The storage of pop-up buffers on disk can reduced significantly the storage requirements of the frame. To make use of this option the SWAP-FILE statement is coded in the frame header. This keyword causes the Speedbase compiler to allocate space for pop-ups within the disk swap file rather than in the application frame for all pop-ups in the frame and any frame dependant on it, whether or not these dependant frames are part of the current compilation.

A particular pop-up window may be excluded from the disk swap file by use of the $OPT NSW compiler option, see section 6.6. Note that pop-ups wider than 85 characters are always excluded from the disk swap file. When the root frame is executed the Speedbase database manager allocates a disk file of size 32kb on logical unit BAW. This logical unit should be assigned, usually in a menu, before the frame is executed.

The swap file is named BAW*xxxxn* where *xxxx* is the operator-id and *n* is the operator's partition number. The swap file is deleted automatically on frame termination. The size of the swap file may be coded explicitly in the header:

SWAP-FILE *nnnnn*

where *nnnnn* is the size of the swap file in bytes, in the range 1 to 65535 inclusive. You are recommended to use only the default size of 32767 bytes or the size 65535 bytes in order to minimize the possibility of disk fragmentation.

5. The Data Division

The optional Data Division is used to define data items accessed later within the frame. As described in Chapter 6, data items may also be implicitly declared by use of Window construct. The Data Division may contain the following:

Data Definitions	Definitions of working storage variables used for calculation or other purposes. BASED items, used to implement parameterised subroutine CALLs, may also be defined.
Print Format (PF) constructs	Used to define printed report layouts.
File Definitions	FD constructs to allow access to traditional index-sequential, relative-sequential and text files.

5.1 Data Division Structure

The optional Data Division is introduced by the header:

```
DATA DIVISION
```

The end of the Data Division is indicated by the start of one of the following Procedural Divisions or the end of the frame:

```
WINDOW DIVISION
PROCEDURE DIVISION
LOAD DIVISION
UNLOAD DIVISION
ENDFRAME
```

Data declarations, FD and PF definitions may be coded within the Data Division in any order. The Global Cobol LINKAGE SECTION header statement is not supported by the Speedbase Development Language. Linkage Section items may be declared anywhere within the Data Division by use of the BASED clause.

5.2 Data Definitions

The Speedbase compiler provides the usual Cobol data definition facilities. The language supports level 77 elementary items, as well as level 01 group items which can themselves be subdivided into elementary items, or as many as 19 levels of subgroup.

5.2.1 Defining Level 77 Elementary Items

Level 77 elementary items, which are not subdivided, may be defined in the data division by coding:

77 *data-name* [REDEFINES *name-1*][OCCURS *n*] *picture clause* [BASED *name-2*]

The *data-name* must be a symbol, you **cannot** use the reserved word FILLER in its place. If an OCCURS clause is present the quantity *n* must be an unsigned positive integer. The optional REDEFINES clause allows you to redefine a previously declared item whose data-name you specify as *name-1*. An item with a REDEFINES clause is known as a **redefinition**. The optional

BASED clause enables you to declare a special type of item known as a **based area** whose location is determined from the contents of the pointer whose data-name is *name-2*. The statement establishes one or more elementary items whose attributes are determined by the picture clause. When the OCCURS clause is omitted, a single item is set up, when present, space is allocated for a table of *n* such items.

5.2.2 Defining Group Items

Group items, which are subdivided, are introduced by a level 01 data item, followed by any number of subordinate items, level 02 to level 49. Groups may be defined anywhere within the data division. The level 01 item is defined by coding:

01 *data-name* [REDEFINES *name-1*] [OCCURS *n*] [BASED *name-2*]

The quantities *data-name*, *name-1* and *name-2* should be supplied as symbols, as necessary. If it is not required to refer to the group explicitly the reserved word FILLER may be coded for the data-name. If an OCCURS clause is present the quantity *n* must be an unsigned positive integer. The optional OCCURS clause allows you to set up a table, each entry of which has the format of the data area described by the group. If the clause is omitted just a single occurrence of the group will be established. When the clause is present space is allocated for a table of *n* such groups.

Following the level 01 definition, the remaining subordinate items are declared by statements of the form:

level-number data-name [OCCURS n] [*picture clause*]

The *level-number* must be two digits in the range 02 to 49 inclusive. The *data-name* should normally be a symbol, although if it is not required to refer to the item explicitly you may supply the reserved word FILLER instead. If the OCCURS clause is present *n* must be an unsigned positive integer.

If the *picture clause* is omitted, the item forms a **subgroup** containing all the following items up to, but not including, the next item with an equal or lower level number. If the definition does not contain an OCCURS clause, a single occurrence of the subgroup will be established. Where the clause is present space is allocated for a table of *n* such subgroups.

If the picture clause is coded then the item is **elementary** and is treated in exactly the same way as a level 77 elementary item. If the definition of a group or subgroup contains an OCCURS clause then it is termed a **repeating group**. No subordinate definition within a repeating group may itself contain an OCCURS clause. This means that any tables defined by repeating groups are one-dimensional only.

5.2.3 Example

```
01          AREA
  03        A                 PIC X       * Elementary item
  03        B     OCCURS 20   PIC X       * Table of 20 elementary items
  03        C                             * Subgroup of AREA
    05      C-1               PIC X       * Elementary item in C
    05      C-2                           * Subgroup in C
      07    C-2-1             PIC X       * Elementary item in C-2
      07    C-2-2 OCCURS 5    PIC X       * Table of 5 elementary items in C-2
  03        D     OCCURS 10               * Repeating group of AREA
```

```
   05      D-1                PIC X       * Elementary item in D
   05      D-2                            * Subgroup in D
     07    D-2-1              PIC X       * Elementary item in D-2
     07    D-2-2              PIC X       * Elementary item in D-2
03         E                              * Subgroup of AREA
   05      E-1                PIC X       * Elementary item in E
   05      E-2   OCCURS 8                 * Repeating group in E
     07    E-2-1              PIC X       * Elementary item in E-2
     07    E-2-2              PIC X       * Elementary item in E-2
```

Figure 5.2.3a - Group Data Definition Example

Figure 5.2.3a shows a level 01 group named AREA, with the level numbers of subordinate data items suitably indented so that the structure of the data is readily apparent. The lines with a picture clause (e.g. PIC X) all define elementary items, or in the case of B and C-2-2, a single entry of a table of such items. Note how a table, C-2-2, can itself be part of a subgroup. It could not of course be part of a repeating group, because no item within such a group can itself contain an OCCURS clause.

The example shows how subgroups and repeating groups can themselves contain subgroups. Subgroup C contains subgroup C-2, and repeating group D contains subgroup D-2. A subgroup can contain repeating groups (e.g. E contains E-2). The only combination which is **not** possible is for a repeating group to contain another repeating group, because of the OCCURS clause limitation.

5.3 Picture Clauses

The picture clause has the general format:

PIC *type*[(*qualifier*)] [COMP]

where *type* indicates the type of item being declared, *qualifier* its precision or length and the COMP phrase applies to computational items only.

5.3.1 Character Pictures

The picture clause for a character item is:

PIC X(*length*)

where *length* is the number of characters required. If the length is 1, PIC X may be coded.

5.3.2 Display Numeric and Computational Pictures

The display numeric and computational variable picture clauses are written in one of the formats:

PIC 9(*p*) [COMP]

PIC S9(*p*) [COMP]

PIC 9(*p,q*) [COMP]

PIC S9(*p,q*) [COMP]

where *p* is the number of digits before the decimal point, in the range 1 to 15, and *q* is the number after the decimal point in the range 1 to 7. The sum *p*+*q* must not be greater than 18. If S is coded the variable is signed. If *p* is 1, PIC 9 or PIC S9 may be coded. If COMP is coded the variable is computational, otherwise it is display numeric. A computational variable occupies 1 to 8 bytes, the actual number being a function of *p*+*q* as tabulated in Table 5.3.2a:

p+*q*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Size	1	1	2	2	3	3	4	4	4	5	5	6	6	6	7	7	8	8

Table 5.3.2a - Size of Computational Variables

Because arithmetic working is in binary, computational items are themselves binary and are capable of containing numbers greater than the size implied by *p*. Also, they can always hold negative numbers even if their picture clause states that they are unsigned. The actual range of values that a computational variable can assume is termed its **capacity**. It is capacity, rather than the format specified in the picture clause, which determines when overflow occurs during arithmetic operations and moves to computational fields.

Note, however, that if you attempt to DISPLAY a computational item containing a value which does not agree with its picture clause, overflow will occur. Similarly, the picture information is used in validating computational input obtained using the ACCEPT operation so it is impossible to input a computational value which does not agree with the receiving field's picture clause.

5.3.3 Pointer Pictures

A pointer is a data item, two bytes long, in which the location of a data item or program statement can be stored. The value of a pointer must be between 0 and 65535 (64 Kbytes - 1). It is represented by true binary notation. The low address byte of a pointer is the most significant and its senior bit is not interpreted as a sign bit but is considered to represent the 32 Kbyte unit position. To indicate that a data item is a pointer, code the picture clause:

```
PIC PTR
```

5.3.4 Date Pictures

The picture clause for an item which will contain a date is coded as follows:

```
PIC D
```

or:

```
PIC DATE
```

This causes the compiler to generate a PIC 9(6) COMP item.

5.4 Value Clauses

A VALUE clause can be used to initialise an elementary item defined in **working storage**, providing that item is not part of a repeating group. The formats of the VALUE clause are as follows:

```
VALUE     "character string"          * Format 1 (see below)
VALUE     #hexadecimal string         * Format 2 (see below)
VALUE     numeric string              * Format 3 (see below)
VALUE     ZERO                        * Format 4 (see below)
```

VALUE	LOW-VALUES	* Format 5 (see below)
VALUE	HIGH-VALUES	* Format 6 (see below)
VALUE	SPACE or SPACES	* Format 7 (see below)

These formats are described in detail in the following sections.

5.4.1 Value Clauses for Character Items

Elementary character items may be initialised using formats 1, 2 and 4 to 7. Formats 1 and 2 only initialise the number of bytes specified by the string, whereas formats 4 to 7 initialise the whole of the item to ASCII zeros, LOW VALUES, HIGH VALUES, or ASCII blanks respectively. Several VALUE clauses may be specified following a data definition, in which case the values are concatenated. For example:

```
77    A     PIC   X(10)
            VALUE "ABC"
            VALUE "XYZ"
            VALUE SPACES
```

causes the first 6 bytes of A to be set to ABCXYZ and the remaining 4 bytes to be set to blanks. Note that the coding of VALUE SPACES in this example is unnecessary since uninitialised rightmost bytes of character items are set to blanks by default. If the character variable being initialised contains an OCCURS clause it is treated as a single long character string during VALUE clause processing.

5.4.2 Value Clauses for Display Numeric Items

For elementary display numeric items formats 1, 2 and 4 to 6 are valid. Format 1 converts the string specified to a standard numeric string and initialise the item to this value. Format 2 initialises the item to the value specified, left justified and padded with LOW-VALUES if necessary. Formats 4 to 6 initialise every byte of the item to ASCII zero, LOW-VALUES, or HIGH-VALUES respectively. Note that if the data definition contains an OCCURS clause a separate VALUE clause must be coded to initialise each occurrence. The first VALUE clause sets up occurrence 1, the second occurrence 2, and so on.

5.4.3 Value Clauses for Computational Items

For elementary computational items formats 2 to 6 are valid. If format 2 is used the hexadecimal string specified must establish every byte of the item and no more. Format 4 initialises every byte of the item to binary zeros. Format 5 initialises each byte to binary zeros and format 6 initialises each bit to binary ones. Note that if the data definition contains an OCCURS clause a separate VALUE clause must be coded to initialise each occurrence. The first VALUE clause sets up occurrence 1, the second occurrence 2, and so on.

5.4.4 Value Clauses for Date Items

A format 2 or 3 VALUE clause can be used to give a date field a specified numeric or hexadecimal value. A format 1 VALUE clause, where the character string is a valid long or short date in *dd/mm/yyyy* format, will cause the date field to be initialised to the internal format representation of the date specified.

5.4.5 Bytes Not Initialised by Value Clauses

All bytes not themselves set up by VALUE clauses, are initialised to binary zeros **if** the elementary item containing the bytes belongs to a repeating group. If the item does not, then the bytes are set up according to the data type established by its picture clause:

- Uninitialised bytes within character items are set to ASCII blanks
- Uninitialised display numeric items are set to ASCII zero - note that this is an invalid value unless the item is an unsigned integer
- Uninitialised computational items are set to binary zeros
- Pointer items are set to LOW-VALUES

Data items appearing within a redefinition do not cause any initialisation to take place.

5.5 Redefinitions

A redefinition is a level 01 group or level 77 elementary item (e.g. B) which redefines the storage occupied by another data item (e.g. A). A redefinition is introduced using the REDEFINES clause in the data definition:

```
        01      B       REDEFINES A [OCCURS n]
          03    etc.
          03    etc.
or:
        77      B       REDEFINES A [OCCURS n] PIC etc.
```

The data item being redefined (e.g. A) may itself have been declared as level 77, level 01, or indeed as any of the subordinate levels from 02 to 49. It may also be the filename or map-name labeling a file or map definition. However, it must have been defined **previously** in the data division, or be one of the in-built system variables described in Chapter 9.

The size of B in bytes should be no greater than that of A. If this rule is broken, the compiler flags the first subordinate item of B occupying storage outside that allocated to A with a warning message. If either B or A is a data definition with an OCCURS clause, then the size of the item for the purposes of this comparison is considered to be the length in bytes of the total table defined by the OCCURS clause.

5.6 The Print Format (PF) Construct

The PF construct is used to define printed report layouts. A frame will normally contain a number of PF constructs, each of which defines one or more print lines which are printed together in the report. The construct defines both the **source** of each data item to be printed, and the **destination** of that item within the print line. During execution, the construct automatically causes these source fields to be moved to their appropriate positions in the print lines. These print lines are then written to the printer as normal.

The PF construct provides the following functionality:

Print File is automatically opened when the construct is first executed. Output may be directed to one of a number of printers or may be spooled to any random access device. This print file is automatically closed when the frame finally terminates.

Print Line Construction The construct automatically causes the construction of print lines at run time. This process causes source fields to be moved to their assigned locations within these lines. Any field conversion from internal to display formats also takes place automatically.

Page Throws automatically take place when required. This causes any page trailer and header lines to be printed. No special logic is required within the frame to deal with these conditions.

Automatic Restarts are provided, allowing any report to be re-printed starting from a given page. The system also deals with any printer hardware problems, allowing re-assignment of the output device during printing.

The PF construct is executed by means of the PRINT verb, which may be coded in any of the frame's Procedural Divisions. Reports may be printed on blank listing paper, or on pre-printed forms. Special stationery may be selected by using the MOUNT verb, which is also used to specify non-standard form lengths. The construct may be used to produce reports of virtually any complexity. Even troublesome printouts such as combined Remittance Advice and Cheques can be handled with remarkable ease.

5.6.1 Basic Principles

5.6.1.1 Print Segments

The first task in coding a report print frame is to analyse the desired output. The purpose of this analysis is to determine which lines are always printed together (i.e. in sequence). These groups of lines are known as **print segments**, each segment being coded as a single PF construct. This is perhaps best illustrated by reference to a simple example report layout:

```
Sample Report          Customer Listing Date   01/05/89    Page: 1

Cust No     Customer Name                  Suburb            Balance

XXXXXXX     XXXXXXXXXXXXXXXXXX             XXXXXXXXXX        99999.99

Report Total                                                 99999.99
```

The above example consists of two header lines, a single detail line, and a report total line which is printed at the end of the report. Since the two header lines are always printed together, this constitutes a single report segment. The detail line and report total lines are printed independently of each other and therefore constitute separate print segments. This example therefore consists of three print segments and requires three PF constructs to be coded.

5.6.1.2 Fixed and Floating Segments

The location in which a segment is to be printed within a page may be fixed or may float over an area of the page. Returning to the above example, the header lines making up the report will always be printed in a fixed position (e.g. starting at line one of each page). The detail and total lines are printed in a floating area, which covers the rest of the page.

This area or position in which a particular print segment may be printed is coded as part of the PF construct. Whenever a print operation takes place, the current line position on the page is checked. If this current line position is within the defined area, then all is well, and the segment is simply printed. If the current position is before the start of the print area, the paper is advanced until it is within the print area. If the current line position is beyond the area specified for the print segment, a page throw automatically takes place.

5.6.1.3 Header and Trailer Print Segments

Several print segments may be related, and this relationship is coded as part of the PF construct. This relationship specifies the sequence in which PF constructs will follow each other. Two relationships may be specified by the HEADER and TRAILER options.

These options are invoked when a page advance needs to take place. Whenever printing a segment would cause an overflow of its designated print area, a page advance will occur. Before actually skipping to the next page, any trailer segments specified will first be printed. A page advance then occurs, after which any header segments are printed. Returning to the above example, the header lines segment will therefore be printed automatically whenever printing a detail or total line segment would cause a page throw to occur.

5.6.2 General Format

The general format of the PF construct is as follows:

```
        PF rt [HEADER r1] [TRAILER r2]
        [START FROM l1 [TO l2]]
or..
        [START AT l3]
        .......
        detail Lines
        .......
        ENDFORMAT
```

5.6.3 The PF Statement

The PF statement introduces the Print Format construct and assigns it a record-id, shown above as *rt*, which must be composed of an alphabetic character followed by any alphanumeric character. This ID is later used to differentiate between references to existing variables and variable declarations within the PF construct detail lines. It should therefore be unique (i.e. not referenced by an ACCESS or RF Statement).

The HEADER option allows a header PF to be selected. This header will be printed whenever a page throw is required. The record-id *r1* specifies the ID of the header PF construct to be used. This header PF construct must be coded previously within the current frame's Data Division.

The TRAILER option allows a trailer PF to be selected. This format will be printed whenever a page throw is required. The trailer PF will be printed **before** a page advance takes place. The PF construct describing the trailer must have been coded previously within the current frame's Data Division.

5.6.4 The START Statement

The optional START statement specifies the area of the page where the PF may be printed. This area is identified by line number, shown as *l1* through *l3* above. The first line of each page

is always counted as line one. Line numbers must be coded as numeric literals in the range 1 to 99 inclusive.

The line numbers specified by the START statement indicate the start and end line numbers **from which** the PF may be printed. For example, an end line number of 50 means that a PF print may take place at a point up to line 50. If the PF has three print lines, this means that lines 50, 51 and 52 would be used for printing.

The statement has two formats. The **START FROM** format allows the lower line number limit to be specified. The optional TO clause may then be coded to specify the end line number of the area. When coded, this end line number may not be less than the start line number. If the TO clause is omitted, the end line number will be calculated at run time as follows:

page length - length in lines, current segment - 6

if no trailer PF is coded, or:

trailer start line number - length in lines, current segment

if a trailer PF is coded. If the latter calculation leads to an invalid line number the prior calculation will be performed, as if no trailer PF had been coded.

The **START AT** format of the start statement simply specifies the same line number for both start and end lines. The statement:

START AT 1

is therefore functionally equivalent to the statement:

START FROM 1 TO 1

If the START statement is omitted, START FROM 1 is assumed.

5.6.5 PF Detail Lines

The optional START statement is followed by a sequence of detail lines. Each detail line may consist of zero or more **text-items** which are simply moved into the print line when the PF construct is executed. These may be followed by an optional **data-item** which may be a field definition, or a reference to a previously declared field.

The general format of a text-item is:

n1 n2 "*text to be printed*"

The general format of a data-item is:

n1 n2 name [*picture clause*] [ADD (*name-2*)][FMT "*options*"]

where *n1* and *n2* above represent the line and column number co-ordinates where the item is to be printed, *name* is the data-name of a declared or referenced field, *picture clause* specifies the format of the field as it is printed, the ADD option specifies automatic addition to numeric field

name-2 and FMT "*options*" causes the data-item to be formatted for printing according to Table 5.6a.

5.6.5.1 Line and Column Numbers

The line number *n1* specifies the line number **relative to the start of the print segment** on which the item is to be printed. The first line number is always counted as 1. The line number must be in the range 1 to 99 inclusive.

The column number *n2* specifies the position at which the leftmost character of the item is to be printed. This column number must be in the range 1 to 132 inclusive. The last column number at which data may be printed is column 132. If this right margin is exceeded, the compiler will generate an error message.

Each time a new line number is declared, the Speedbase compiler automatically generates a print line within working storage. For example, if a print segment contains two lines, two 132 character print lines will be created. The **spacing** between the lines is not significant in the allocation of print lines. If data is to be printed only on lines 1, 3 and 5, three print lines will be allocated. The compiler "remembers" that a line advance is necessary before printing lines 3 and 5. When this PF construct is printed, five lines will therefore be output to the printer. The maximum number of print lines that may be declared for a single PF construct is sixteen, exclusive of line spacing.

The **order** in which the line numbers is **declared** is significant. The compiler requires that they must be declared in line number order. It is therefore not permissible to begin by coding say the fifth line of the print segment, and then to code earlier lines.

5.6.5.2 The Data-Name

The data-name is always a variable name. This variable name may be either a **reference** to a variable which has been declared elsewhere, or may be a **declaration** for a variable stored within the print line. If the data-name coded, *name*, begins with the two-character record ID coded in the PF statement, then it is treated as a declaration. Otherwise it is treated as a reference to an existing variable.

Referencing an existing field causes the compiler to treat that field as a **source**. The compiler then generates code to cause this source field to be moved to the **target** location in the print line. The target location in the print line is defined by the line and column number coded for the item.

5.6.5.3 The Picture Clause

When a referenced field is coded, the field's picture clause is optional. If the picture clause is omitted, then it defaults to the picture clause of the referenced item. If a picture clause is coded, then this represents the picture clause of the item **as it will appear in the print line**. The picture clause must be of an appropriate type, so that a valid move instruction may be generated. Valid combinations are documented as part of the MOVE statement in Chapter 7.

Any field name beginning with the record-id coded in the PF statement is treated as a field declaration within the print line. The variable name coded must be unique to the frame and may be used as part of MOVE and other procedural statements to address the print line directly. Declared variables always require a picture clause which determines the length and other attributes of the field and are used to allow fields to be calculated or formatted in a special way prior to printing. Procedure Division code is written to perform this function, which then moves

the field directly into the print line prior to issuing the PRINT statement. A declared field can be the target of an arithmetic statement, and indeed can be used like any other working storage field.

5.6.5.4 The Add Option

The ADD option causes the field being printed to be added to an accumulator as part of the execution of the PF. The option is used for the calculation of report sub-totals and totals. Since ADD is an arithmetic operation its source variable must be numeric computational. Since declared variables are always display items, this means that it can only be used as an option for referenced fields. The target of the ADD option, shown as *name-2* above, must also be a numeric computational field. This field must previously have been declared, normally as a working storage item within the Data Division.

Whenever a PF construct containing an ADD option is executed, the value of the source field is added to the option's target. This target can then later be used as the source of a PF construct used to print a total line. Only one ADD option is permitted for each data item. If multiple levels of totaling are required, the PF construct printing a sub-total may itself contain an ADD option to calculate report totals. This process can be repeated indefinitely to calculate any number of totaling levels.

5.6.5.5 The FMT Option

Option	Condition	Print Formatting	As input	As printed
B	*field* = 0	**Blank** when zero	00.00	
C	*field* < 0	Trailing **CR**	-12.34	12.34CR
<		Enclosed in **()**		(12.34)
-		Trailing **-**		12.34-
D	*field* > 0	Trailing **DR**	+56.78	56.78DR
>		Enclosed in **()**		(56.78)
+		Trailing **+**		56.78+
,	None	**Comma** insertion	90123	90,123
$	None	Leading **dollar**	45.67	$45.67
0		**Zero** fill	8.90	0008.90
*		**Asterisk** fill	1.23	***1.23
L	Date only	**Long** date	DD/MM/YYYY	DD/MM/YYYY

Table 5.6a - The Print Formatting Options

FMT "*options*" is used to cause special formatting of the printed output. Table 5.6a lists the five groups of options. You may specify no more than one option from each group in any set of options. Examples of invalid options would include:

```
BC<+,$
->+*
BCD,0*
```

Examples of valid sets of options and the resultant print formatting are as follows:

Options	Field as Input	Field as Printed
BCD,	-678.90	678.90CR
	0.00	
	+1234.56	1,234.56DR
B<,	+789.01	789.01
	-7890.12	(7,890.12)
B-+$,	+345.67	$345.67+
	$890.12	$890.12-
-+0	+345.67	00345.67+
	-8901.23	08901.23-
CD*	+456.78	**456.78DR
	-901.23	**901.23CR

Table 5.6b - Examples of Print Formatting

5.6.6 Example Report

```
FRAME REPORT "CUSTOMER LISTING"
SEQUENCE MENU MENU
ACCESS CU                           * Create I/O channel to CUST record
DATA DIVISION
77 T-RTOT   PIC 9(6,2) COMP         * Working-storage field for report total
*
PF H1                               * Print format for 2 header lines
START AT 1
01 01 "Sample Report Customer Listing"
01 37 "Date:"           01 42 $$DATE
01 51 "Page:"           01 56 $PGNO
03 01 "Cust No    Customer Name"
03 38 "Suburb"
03 53 "Balance"
ENDFORMAT
*
PF D1 HEADER H1                     * Print format for detail line
START FROM 5                        *
01 01 CUSTNO                        * Customer number
01 11 CUNAME                        * Customer name
01 38 CUSUBB                        * Customer suburb
01 51 CUBAL ADD(T-RTOT)             * Customer balance
ENDFORMAT
*
PF T1 HEADER H1
02 01 "Report Total" 02 51 T-RTOT
ENDFORMAT
*
PROCEDURE DIVISION
*
DO
      FETCH NEXT CUIDX NOLOCK
      ON EXCEPTION FINISH
      PRINT D1
ENDDO
PRINT T1
EXIT
ENDFRAME
```

Figure 5.6c Sample Report-Printing Frame

```
Sample Report          Customer Listing Date   01/05/89    Page: 1

Cust No     Customer Name                  Suburb           Balance

XXXXXXX     XXXXXXXXXXXXXXXXXX             XXXXXXXXXX       99999.99

Report Total                                                99999.99
```

The frame listed in Figure 5.6c is all that is required to print the report example from section 5.6.2, repeated above. The frame includes three PF constructs each defining a print segment of the example report. PF H1 defines the header lines of the report, PF D1 the detail line and PF T1 the total line.

In PF H1 the statement START AT 1 means that this PF will only ever be printed starting at the first line of the page. This is followed by a number of text items which lay out the fixed text of the heading lines. Two systems variables, $$DATE and $PGNO will also be printed. These variables contain the system date and current page number respectively. Whenever a page throw is required, the variable $PGNO is automatically incremented.

PF D1 contains the layout of the detail line and consists of references to fields on the customer record, CU, so picture clauses are not therefore required. The field CUBAL is followed by the ADD option to accumulate the total balance of all customers in the field T-RTOT.

PF T1 contains the layout of the total line, which consists only of the text "Report Total" and the balance of all customers, accumulated in T-RTOT. The first line number declared for this PF is line two, meaning that a one-line advance will take place before the PF is printed. The same principle can also be used to achieve double line spacing.

Both the detail and total line print formats have defined PF H1 as their header. Whenever printing D1 or T1 would cause a page throw, header H1 will therefore be printed first. This will also take place when the first PRINT statement occurs, thus forcing out the header PF on to the first page of the report.

The Procedure Division contains all the code necessary to "drive" the frame. It starts by establishing a DO loop. The FETCH NEXT statement then reads each customer record sequentially. The PRINT statement causes this record to be printed and the total balance accumulated into the variable T-RTOT.

An end-of-file condition ultimately occurs and is trapped by the following ON EXCEPTION statement. The FINISH verb in this statement transfers control to the end of the DO loop, where the total line is printed. The EXIT statement returns control to the Speedbase Presentation Manager, which re-loads the menu program.

5.6.7 System Variables used for Printing

The following system variables relate to PF handling:

Name	Explanation
$PRUN	Printer unit ID. This PIC X(3) variable specifies the printer unit which is to be used to output reports generated by the PF construct. This ID is initially set up to the unit "$PR", but may be changed to some other random access device unit ID. If changed, this unit ID remains in force until the end of the current session.
$PGNO	Current page number. The page number of the page currently being printed on is stored in this 9(4) COMP variable. The page number is automatically set to zero when the printer is opened in response to the first PRINT statement in the frame. It is automatically incremented whenever a page throw takes place. $PGNO may be referenced in a PF construct to print the current page number on each page. It must NOT be modified by the application frame.
$LINO	Current line number within the page. This 9(4) COMP variable indicates the number of lines that have so far been printed within the current page. It is set to zero when the printer is initially opened, and reset as page throws occur. It is automatically incremented as lines are output to the printer. It must NOT be modified by the application frame.
$RSPG	Restart page number. This 9(4) COMP variable indicates the first page number from which re-printing should commence. The variable normally contains zero to indicate that printing should not be suppressed. The variable is reset to zero whenever the printer is closed. A restart option may be provided within a print frame by setting it to the page number from which re-printing should commence.
$PHLT	Printer halt suppress flag. This 9 COMP variable is used to control the print interrupt feature. When printing normally commences, the following message is displayed: `Type <Ctrl G> to halt printing` During a print run, the operator may then interrupt printing by keying <Ctrl G>. Printing may then be restarted, directed to another device or suppressed. If suppressed, the program will normally continue processing, but without producing a report. Moving -1 to $PHLT also allows the operator to suppress printing, but in this event the frame will be terminated by issuing a STOP RUN instruction. Moving the value 1 to $PHLT suppresses the interrupt feature. It should be noted that any changes made to $PHLT remain in force until the end of the session.

5.7 The File Definition (FD) Construct

The Global Cobol File Definition (FD) statement is supported by the Speedbase compiler to permit access to traditional ISAM, RSAM and TFAM files. These facilities would not normally be used in an Speedbase application, as the Speedbase database access method and PF construct provide enhanced functionality.

FD constructs have been provided solely to allow interfaces to existing GSM file-based applications to be implemented. The syntax and operation of FD constructs is described in detail in the Global Cobol Language Manual.

Please note that the ORGANISATION Statement required in the COBOL implementation for the text file access method (TFAM) is not required by the Speedbase compiler, and should not be coded.

6. The Window Division

This chapter describes the structure and operation of the Window Division. Section 6.1 provides a brief overview of window functionality, section 6.2 defines the structure of windows. Section 6.3 describes the layout of windows and the facilities provided for the operator. Section 6.4 explains how windows are controlled, describing the processing that takes place when a window is invoked. The Routines Section and the processing of database records are described in sections 6.5 and 6.6. Section 6.7 covers the syntax of the window construct in detail, followed by section 6.8 in which special programming facilities and considerations are discussed. Appendix C provides examples of the facilities described in this chapter.

A window defines the screen format by which a particular record type may be presented to, and accepted from the operator. The processing involved in this is handled by a window manager, which normally allows operations such as record addition, selection, maintenance, deletion and enquiries to take place. The window manager provides the following facilities:

Multiple Windows A screen may contain any number of windows. These may co-exist on the screen, overlay each other, or pop-up during processing.

Window Sequencing The sequence in which windows are invoked is either automatic or specified in the application program.

Scrolled Windows A window may be fully, partially or non-scrolled. A scrolled window may display up to twenty records. Scrolled and non-scrolled windows may co-exist on the same screen.

Enquiry Facilities Records may be retrieved via any index, and may be paged in both ascending and descending order.

Processing Routines Routines Section entry-points allow procedural code to be performed during window execution.

Optional Fields All fields within a window are optional, and can be suppressed at run-time in the Routines Section.

Colour Support All windows automatically support colour and monochrome facilities on appropriate terminals.

Re-entrant Processing The window manager allows windows to be invoked from a window already executing.

Function Key Support Keyboard operations are carried out using function keys. Using the <HLP> key, a menu of currently enabled functions is automatically displayed, allowing the operator to select an appropriate function.

Dependent Windows Windows can be restricted to operate only on a given set of records, (e.g. invoices for a given customer).

Automatic Validation Input fields are automatically validated for type and converted to the appropriate format.

6.1 Window Division Structure

The optional Window Division is introduced by the statement WINDOW DIVISION and is followed by one or more window constructs. The Window Division is terminated by the start of the next division, or by the ENDFRAME statement. A skeleton of the Window Division is shown below:

```
[WINDOW DIVISION]

[$OPT NSW]
WINDOW id [USING rt [DEPENDENT ON rt2 ]]
[window-options]
window-body
[ROUTINES SECTION]

ENDWINDOW

...more windows...
```

Each window defines fields that will be displayed and accepted for a particular record type known as the **target record**. This target record type must be stored on a Speedbase database. The window will then normally allow such operations as addition, maintenance, deletion and enquiry to take place on this target record type.

Each window may also contain a **Routines Section**, in which additional, application-specific procedures may be coded. The Routines Section contains a number of entry-points, which are called during processing of the window. These entry-points allow, for example, additional validation to be performed on accepted fields, and may be used to perform additional updates on completion of each transaction.

A window may occupy all or part of the screen. An example of a typical window, as it appears on the screen, is shown in Figure 6a. A screen may be constructed from many such windows, and these windows may overlay each other during the course of processing.

When a frame is run, the first window is normally executed automatically. However, if a Procedure Division has been coded, it is executed instead and windows are then executed under its control. Procedural statements provide control over window display, clearing and data entry functions.

6.2 Window Formats and Operator Facilities

6.2.1 Window Formats

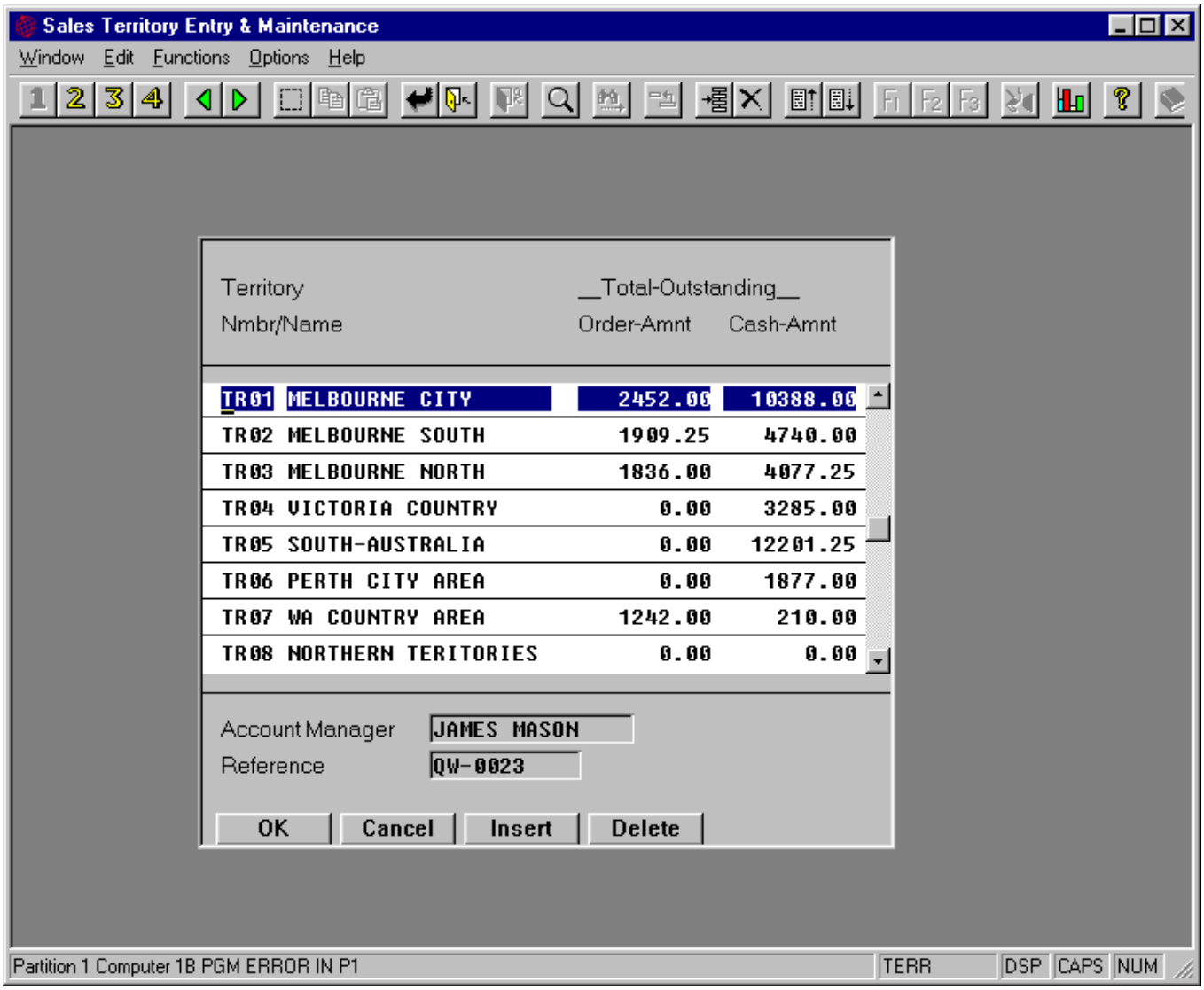


Figure 6.2a - A Typical Speedbase Window

Figure 6.2a shows a typical Speedbase window as it appears on the screen. This window contains two quite separate areas, known as the **scrolled** and **non-scrolled** areas. In the example, the **scrolled area** is located in the top portion of the window and currently contains eight records, TR01 to TR08.

The scrolled area of the window therefore contains a number of "slots" in which a record may be entered or displayed. Each one of these is called a Record Display Area (RDA). Thus the example window has eight RDAs, each of which is currently occupied by a record. The cursor is always positioned on one of these RDAs, and when this contains a record, it is called the **current record**.

Given the limited size of screens, it is often not possible to display a complete record in the scrolled portion of the screen. To overcome this, windows can also contain a **non-scrolled** area. This area is used to display those fields from the current record which could not be

accommodated in the scrolled region. In the example, purchase and sales history fields are displayed for the current record, TR01.

The non-scrolled area is optional and, when used, does not need to occupy any particular position on the screen. Equally the scrolled area is optional, may be placed anywhere on the screen and may be as large as desired, each record using as many lines as required. When scrolling is not needed, windows may also be laid out in traditional vertical form, so that only a single record is shown at a time.

6.2.2 Operator Facilities

The operator controls a window using the functions shown in Table 6.2b. These functions are invoked using the terminal's function or control keys, which are set up using the Speedbase customisation utility, $BACUS. Twenty functions have been provided to allow the operator to perform a variety of tasks:

Mnemonic	Description	$FUNC
RET	Accept current field, select record	0
UF1	User Function 1	1
UF2	User Function 2	2
UF3	User Function 3	3
NXT	Go to next window	4
PGE	Forward-page window	5
BPG	Back-page window	6
UP	Back one record (Uparrow)	7
DWN	Forward one record (Downarrow)	8
SKP	Skip fields to next tab-stop	9
ABO	Abort program	10
BCK	Terminate window (Back to prior)	11
CLR	Clear the window	12
DTE	Delete current record	13
HME	Cursor home	14
BFL	Back one field	15
ENQ	Enquiry mode - select index	16
INS	Insert record	17
UDL	Undelete record	18
MOV	Move record	19

Table 6.2b - Window Operations (Keystrokes)

The result of using each of these functions is as follows:

<RET> The RETURN key is used to complete entry of a field, or to accept the default displayed. RETURN is also used to select an existing record in the window for further processing, such as maintenance or deletion.

<UF1> The <UF1>, <UF2> and <UF3> keys are used to invoke special program-dependent functions coded within the application frame. For example, <UF1>

might be used to display a pop-up stock enquiry window from within an order-line entry window.

<UF2> The <UF1>, <UF2> and <UF3> keys are used to invoke special program-dependent functions coded within the application frame.

<UF3> The <UF1>, <UF2> and <UF3> keys are used to invoke special program-dependent functions coded within the application frame.

<NXT> The NEXT key indicates completion of the current window and causes the next window in sequence to be executed. It is normally used to indicate successful completion from a repeating window.

<PGE> The PAGE key causes the next page of records to be displayed in ascending key value within the current window. If the window is empty, the first page of records is displayed.

<BPG> The BACK-PAGE key causes the preceding page of records to be displayed in the current window. If the window is empty, the last page of records is displayed.

<UP > The UPARROW key moves the cursor from the current record to the record display area (RDA) immediately above it. If used at the first RDA the window is scrolled down one record.

<DWN> The DOWNARROW key moves the cursor from the current record to the immediately following RDA. If used at the last RDA the window is scrolled up one record.

<SKP> The SKIP key skips over fields, up to the end of the record or the next tab-stop field. All skipped fields are processed as if the operator had keyed <RET>.

<ABO> The ABORT key is used to abort the frame.

<BCK> The BACK key is used to terminate the current window, and usually transfers control back to the preceding window in the frame. If there is no preceding window in the frame, it is usually terminated as if <ABO> had been keyed.

<CLR> The CLEAR key clears data from the current window.

<DTE> The DELETE key causes the current record to be deleted, and removed from the window. The window is normally scrolled into the position previously occupied by the deleted record, so that there are no "gaps" in the display.

<HME> The HOME key is used to position the cursor at the first record displayed in the window. If already at the first record, the cursor is positioned at the last record displayed.

<BFL> The BACKFIELD key is used to step back to the preceding field in a record.

<ENQ> The ENQUIRE key causes the window to be cleared, and enquiry mode to be entered. The cursor is positioned on the first field of the currently selected index. If

<ENQ> is keyed again, the next available index is selected, to change the order in which records will be displayed. Use of the <PGE> or <BPG> keys displays the first or last page of records in the order of the selected index. The operator may also specify a starting position for this display by entering part or all the index key.

<INS> The INSERT key is used to insert a record before the currently displayed record. The window is normally scrolled apart to create a blank RDA, after which the operator may enter the new record. Note that on re-display the inserted record will be displayed in its index order, which will not necessarily place it in the same position relative to other records.

<UDL> The UNDELETE function is used to re-create the record deleted by the last delete operation. The record is displayed at the current record position.

<MOV> The MOVE function is used to change the order in which records are displayed, where an auto-sequence index is in use.

The type of processing taking place determines which functions may be used. For example, the delete function has no meaning when a record is being added. In order to select any of the currently available functions, an experienced operator will normally use the appropriate function key. The less experienced operator may key <HLP> to make use of the help facility.

When <HLP> is keyed, a menu of the currently available functions is displayed together with the associated key-top names from the operator's keyboard. The operator may then choose an appropriate function from this menu. If <HLP> is keyed again, the help text stored in the frame is displayed as a help window. Any function-keys that may be described in the help window are labeled with the appropriate key-top name. For example, a help-text phrase mentioning the <NXT> function could have the key-top "End" in place of <NXT>, so that the operator may see exactly which key to use on the keyboard in use at the time.

6.2.3 Enquiry Facilities

In order for the enquiry facilities to operate, the target record type must have at least one index. Most records have more than one index by which they may be retrieved (e.g. a customer record could be indexed by customer number as well as name). All enquiries operate using one of these indexes, and the index currently in use is known as the current index.

The current index specifies the order of displayed records when the window is paged. If, for example, the current index is the customer name index, then records would be displayed in name order. When a window is first entered, one of the record's indexes is designated as the current index. This is often the primary index, but another may be specified within the window construct. The operator may, however, select a different index by using the <ENQ> key, which clears the window and causes enquiry mode to be entered. The cursor is then placed on the first field of the current index, which allows the operator to see which index is in use. Keying <ENQ> again simply selects the next index available on the record.

The operator may then use the paging operations <PGE> or <BPG> to display the first or last page of records via the selected index. Alternatively, the operator may key in all or part of the index followed by <PGE> or <BPG>, and in this case only records following the requested index keys will be displayed. For example, keying "B" <PAGE> would display all customers who have a name starting with an ASCII value of "B" or greater (e.g. BURNS, COLLINS). Keying <RET>

instead of <PGE> causes the window manager to search for an **exact match only**, and if not found, to output an error message.

Once one or more records have been displayed on the screen, the operator may use the <UP > and <DWN> keys to move the cursor onto another record within the window. Alternatively, keying <PGE> or <BPG> displays the next or prior page of records from that last displayed.

These facilities provide a mechanism for selecting a record from the database. Once the cursor has been positioned on the required record, the operator may key <RET> to select it for maintenance, or may key <DTE> to cause the record to be deleted. These operations are explained in more detail later in this chapter.

The operator may key the index values to be searched for only when the record display area is blank, and this is normally achieved by keying <ENQ>. The <ENQ> key clears all data from the window, thus conforming to this rule. The enquiry facilities therefore operate in different ways depending on whether or not a record is displayed in the current record display area.

When there is a current record, the operator may page forward or back as previously described, or may use the <UP > and <DWN> keys to select another record within the window. Alternatively the operator can use the <RET> or <DEL> keys to select the current record for maintenance or deletion.

When there is no current record, the operator may key in all or part of the required index key, and then retrieve records corresponding to this entered key value. Again, the <UP > and <DWN> keys may also be used to move to another record display area within the window.

The <PGE>, <BPG>, <UP > and <DWN> keys may be used at any time, even when record addition or maintenance is in progress. When used, these keys cause the record maintenance or addition to be completed, just as if the operator had keyed <SKP>, where after the operation specified by the key takes place.

6.3 Control Structure

When a frame is first executed, the first window is usually invoked by the frame controller. If a Procedure Division is coded it is executed instead and has full control of the further processing of windows.

6.3.1 Basic Functions

All detailed operations of a window are controlled by a system routine called the **Window Manager**. The window manager looks after low-level functions such as cursor positioning, field display and accept, database I/O operations and record locking. Only three high-level statements are therefore necessary to control the operation of windows at run-time. These are:

DISPLAY Displays the window form (i.e. fixed text and/or data).

CLEAR Clears the entire screen, a specific window, or clears only the data from a specific window.

ENTER Transfers control to the window, which may then perform its various tasks, such as enquiry and maintenance.

6.3.1.1 DISPLAY

This statement is used to display either the window form, and/or to display data (i.e. a record) within this form. The window manager keeps track of the status of all windows within a frame. When a frame is first entered, clearly none of the window forms are as yet displayed. The display of the window form normally takes place automatically when it is entered, or displayed.

This normally means that windows are activated (i.e. the window form displayed) as the need arises. It is possible, however, to over-ride this by explicitly activating numerous windows using the display verb with the TEXT option. When a window is activated, a box is displayed around the area occupied by it and the various text items are displayed within this area. If the window is a POP-UP, the image under the window is saved prior to this display. This allows the screen to be reset to its original state when the POP-UP window is later cleared.

6.3.1.2 CLEAR

This statement has three variants. Normally the clear statement clears all windows from the screen (i.e. both the window form and any displayed data). The clear statement may be used, however, to clear a specific window, which must be active when the instruction is executed. Clearing a specific window, also known as window removal, usually causes the area underneath it to be reset to the screen background colour. However, if the cleared window is a POP-UP, then the image that existed underneath the window before it was activated is redisplayed, causing the screen to be reset to its original state.

6.3.1.3 ENTER

This statement causes a window to be executed. When a window is entered, the operator is normally able to add, maintain or delete a number of records. Enquiries may also be performed in order to select a record from the database for maintenance or deletion.

During this processing, routines coded in the Routines Section may be called to perform various tasks such as field validation. Provided these routines do not detect errors, the record will then be written, or rewritten, to the database. This normally concludes processing and control is returned to the statement immediately following the ENTER statement. Alternatively, the operator may abort the window by keying <BCK> or <ABO>, which is regarded as an abnormal exit, and causes an exception condition to be returned.

6.3.2 Window Processing Modes

Windows operate in a number of modes during the different stages of processing. These modes are listed in Table 6.3a below:

Mode	Description	Function	$MODE
ENQ	Enquiry	Initiate an enquiry	1
DSP	Display	Display existing record	2
MNT	Maintenance	Modify existing record	3
DEL	Deletion	Delete existing record	4
EDT	Edit	Create new record from existing	5
ADD	Addition	Add new record	6
INS	Insertion	Insert new record	7

Table 6.3a - Window Processing Modes

6.3.2.1 ENQ and DSP Modes

ENQ and DSP Enquiry and display modes operate together. Enquiry mode operates when the operator initiates an enquiry, and display mode is used to display the results of this enquiry. During enquiry mode, the operator may key in all or part of the record's index fields as previously described in Section 6.2.3. Then, keying <RET> or <PGE> or <BPG> initiates a search for records containing the required index keys. If any records are found, display mode is activated and the retrieved records are displayed on the screen. While in display mode, the operator may move the cursor between displayed records, or display follow-on pages of records, using the <PGE> and <BPG> keys.

6.3.2.2 MNT Mode

MNT Maintenance mode allows the user to maintain the currently displayed record. The record is first retrieved and displayed and is usually selected by keying <RET> with the cursor positioned on it. In this mode, the operator may amend all non-protected fields on the record, on completion of which the record is re-written to the database.

6.3.2.3 DEL Mode

DEL Delete mode operates when the user requests the deletion of an existing record. The operator normally instigates a deletion by placing the cursor on the required record, and then keying <DEL>. As for MNT mode, this record will first have been retrieved using enquiry and display modes. Upon deletion, the record is simply deleted from the database, and the resulting "gap" in the window is removed by scrolling the following records.

6.3.2.4 ADD and INS Modes

ADD and INS Add and insert modes are both used to add a new record to the database. The only difference between these modes is that add mode is used to add a record at the end of the displayed list of records, whereas insert mode is used to insert a new record before another record in the window. The operator enters add mode by placing the cursor on a blank record or by keying <CLR>, and enters insert mode by keying <INS>. Other than these minor differences, processing is identical, and these modes are only distinguished to allow the application programmer to perform any additional processing required for record insertion. This is further discussed in section 6.7.

If the record to be added has a primary index, a check will usually take place to ensure that the record does not already exist. If it does, the operator will be prompted:

```
Record Key already exists; Enquire? :
```

Keying Y to this prompt causes the record to be retrieved and displayed just as if the operator had used the enquiry functions to achieve this. In add and insert modes, the operator may enter all non-protected fields in the record, on completion of which the record is written to the database.

6.3.2.5 EDT Mode

EDT Edit mode is used only for windows that do not access the database, for example a window that is used to accept parameters for a print program. These windows can operate only in add and edit modes. The difference between these modes is that add mode initialises data-items as each field in the record display area is processed, whereas edit mode does not. Add mode therefore always starts with an empty Record Display Area, whereas edit mode starts with a displayed RDA. Edit mode therefore acts similarly to record maintenance in maintenance

mode. Edit mode is selected by simply displaying the various fields in the record using the display verb, after which the window is entered in the usual way.

6.3.2.6 Summary of Modes

These functions are essentially all that ever occurs in an on-line commercial application system. Records are initially created, retrieved many times, sometimes being modified, and are then finally deleted when no longer relevant. Complex on-line processes are simply built up from a number of such functions.

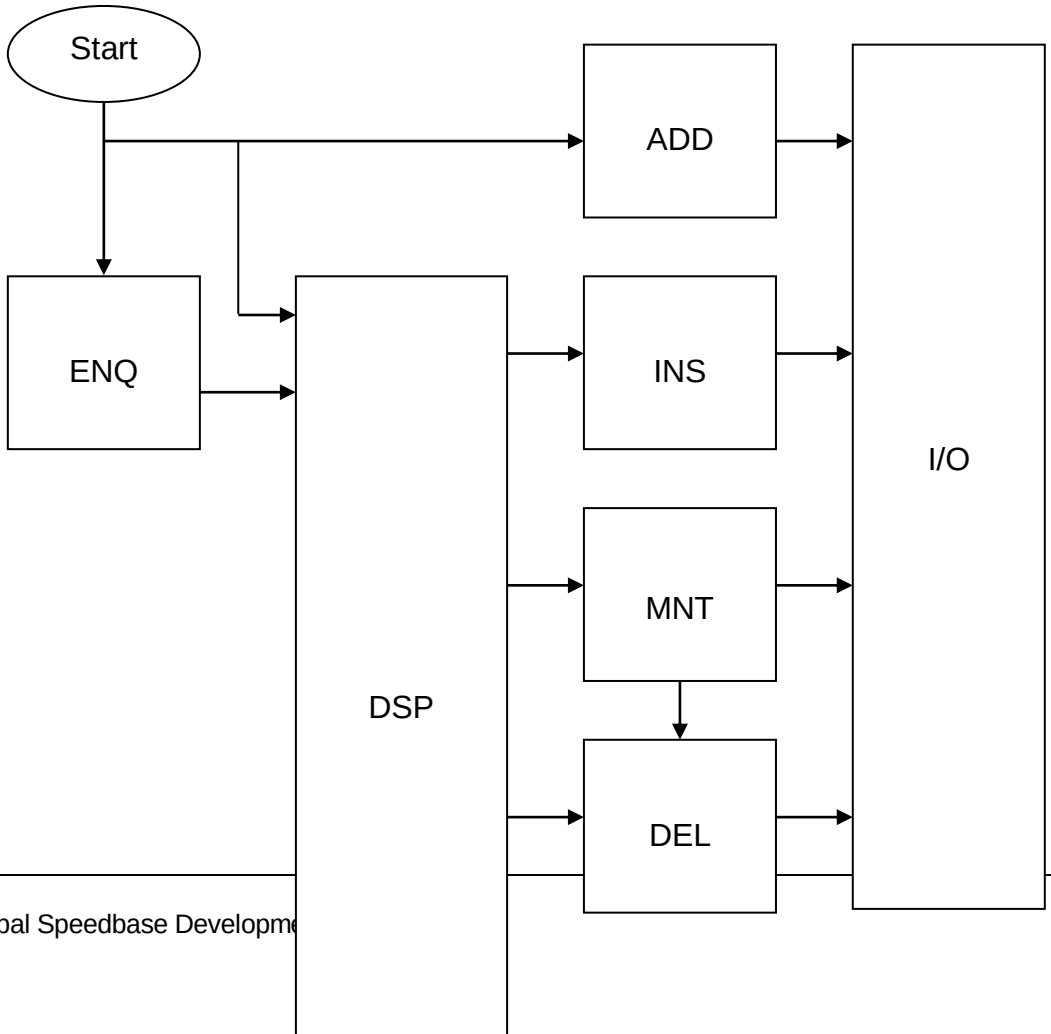
Consider, for example, an order-entry application. The first task is to determine which existing customer the order belongs to. This is achieved by a window using enquire and display modes on the customer record. This is then followed by a window which either identifies an existing order-header record (ENQ and DSP modes again), or allows a new one to be created (ADD mode).

Individual detail lines are then processed by a further window where new order lines are created (ADD mode). Existing order lines might also be reviewed (ENQ & DSP modes) and possibly maintained (MNT mode).

Specialised windows may be constructed by disabling certain modes. By disabling the ENQ mode, the operator is restricted to data-entry of new records. Disabling ADD mode means that only existing records can be processed. Equally, the window may be made to perform enquiries only.

6.3.3 The Processing Cycle

The operations that take place during the processing of a single record are collectively called the window processing cycle, which is outlined in Figure 6.3b:



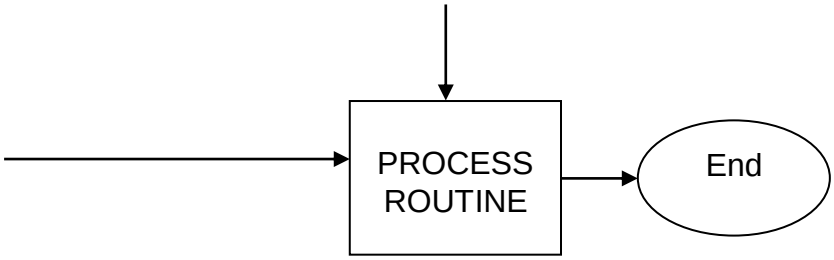


Figure 6.3b - The Window Processing Cycle

When a window is entered, either ADD or DSP modes will normally be activated. If the current Record Display Area (RDA) is not in use, ADD mode will be activated. If the RDA is in use (i.e. a record is currently displayed) DSP mode is activated. Direct entry into ENQ mode will only take place if the RDA is empty and ADD mode has been suppressed.

On entry into the window, an initial mode is therefore selected automatically by the window manager. The operator may then use the <UP > and <DWN> keys to move between record display areas, or use the <PGE> and <BPG> keys to display follow-on pages. Other functions such as <CLR> may also be used. After some initial cursor-key activity, the operator can only do one of the following things:

- Abort the window using the <ABO> or <BCK> keys.
- Move the cursor to a blank RDA to add a new record (ADD mode).
- Place the cursor on a record in the window and key:

  <RET> to select the record to maintain it (MNT mode)
  <DTE> to delete the record (DEL mode)
  <INS> to insert a record at the current RDA (INS mode).

Ignoring EDT mode for the moment, the action starts once ADD, INS, MNT or DEL mode is entered. The processing steps involved in these modes are discussed below:

ADD and INS modes The operator may enter each unprotected field in turn, accepting or overiding defaults as required. Once the last field has been entered, the record is automatically written to the database.

MNT mode The operator may amend any unprotected fields on the record, after which the record is rewritten to the database.

DEL mode The record is automatically deleted from the database and removed from the screen.

After any of the above I/O operations, the optional Process Routine is executed. Several other entry-points are provided in addition to the process routine within the optional routines section. These entry-points allow the application programmer to add application-specific code which is executed during the window processing cycle. A detailed description of these may be found in section 6.4. The Process Routine has been included in this section because it is particularly

useful. It is called at the end of the processing cycle and may be used to perform additional tasks, such as writing an entry to an audit log.

After the optional process routine has been executed, the window will normally exit, returning successful completion. Instead of exiting, it is possible to pass control to further windows, and this is discussed in the following section.

6.3.4 Creating Chains of Windows

Each window in the Window Division may be executed individually under the control of the Procedure Division. Chains of windows can be constructed, however, in much the same way as chains of frames. Each window may contain a sequence statement which specifies the next window to execute following successful or unsuccessful completion. For example, coding:

```
SEQUENCE W1, W3
```

will cause control to be transferred to window W1 on unsuccessful completion (backward exit), and otherwise to window W3 (forward exit). The keyword "EXIT" may also be coded in this statement, and this causes control to be returned. Omitting the sequence statement is the same as coding:

```
SEQUENCE EXIT, EXIT
```

which causes control to be returned under all circumstances.

By using the sequence statement, several windows may therefore be executed before control is finally returned. Any of the windows in the sequence may actually return control, provided the sequence statement permits this. If it is important to know **which** window actually returned control, this can be achieved by setting a flag within the routines section.

A chain of windows will therefore complete successfully or unsuccessfully, and this will cause control to be returned. If the window was invoked using the ENTER statement within the Procedure Division, control will be returned to the statement immediately following it. If unsuccessful completion occurred, this will be indicated by an exception condition, which may be trapped by an ON EXCEPTION statement within the Procedure Division. If no Procedure Division was coded, control will be returned to the frame manager.

6.3.5 Controlling Window Exits

It is important to note that the operator may abort a window at any time using the <BCK> or <ABO> keys, which causes unsuccessful completion to be returned. The <BCK> key causes the backward exit specified in the sequence statement to be taken, and this is normally used to transfer control back to the preceding window. If enabled, the <ABO> key ignores this backward exit, immediately terminating window processing. If no back-window-id is coded the treatment of <BCK> and <ABO> is identical.

Complex processes usually consist of a number of windows each performing a stage in the overall task. The conditions under which windows may follow each other on successful completion can often be important. For example, consider an order entry frame composed of three windows, the first used to maintain order header information, the second used to amend item lines, and the third used to enter optional delivery schedules.

In this example, the item-line and delivery schedule windows would need to repeat in order to allow multiple lines to be entered. Furthermore, it is necessary to ensure that an order header has been established before any item lines can be processed. This control is achieved using the REPEAT option.

When the REPEAT option is **not** used, successful completion will take place after a record has been added, inserted, selected, maintained or deleted. Following deletion the status of the I/O channel is undefined. For all other functions, the processed record is still present in the I/O channel, but is unlocked. The REPEAT option is used to control both when a forward exit may take place, and the lock status of the I/O channel at that point. It has three variants:

REPEAT (with no other clauses) causes the window processing cycle to be repeated indefinitely, and no forward exit is therefore possible.

REPEAT UNTIL NEXT causes the window to be repeated until the operator keys <NXT>. When this clause is coded, the operator can key <NXT> at any time during window processing, and this means that the status of the I/O channel will be undefined.

REPEAT UNTIL CURRENT RECORD specifies that a current record is required before a forward exit may take place. This option is often used to select a record for further processing, such as selecting a customer prior to entering an order. This clause has two effects, it stops a forward exit from taking place following deletion, and it causes the record lock to be retained.

Returning to the above example, the REPEAT UNTIL CURRENT RECORD option would be used in the header window to ensure that an order header record is returned locked on successful completion. The REPEAT UNTIL NEXT option would be used by the item-line window to allow a transfer to the third window at the operator's discretion.

6.4 The Routines Section

A ROUTINES SECTION may be coded for each window containing a number of entry-points called during the processing cycle. Any valid procedural instruction, as documented in Chapter 7, may be coded within the section, including window management statements such as ENTER and DISPLAY. It is therefore permissible to enter or display other windows **from within the Routines Section, providing these are not currently executing.** The entry-points provided by the routines section are divided into two types, entries called during field level processing, and entries called during record level processing, see Table 6.4a:

Routine	Description	Function	Level
B-*name*	Before field	Suppress optional fields	Field
V-*name*	Validate field	Perform extra validation	Field
D-*name*	Default routine	Default field contents	Field
R-FETCH	Record fetch	Reject retrieved records	Record
R-SELECT	Record select	Suppress selection	Record
R-DELETE	Record delete	Suppress deletion	Record
R-WRITE	Record write	Suppress record write	Record
R-REWRITE	Record rewrite	Suppress rewrite	Record
R-PROCESS	Record process	Extended processing	Record
R-TERM	Record terminate	Release locks	Record

Table 6.4a - Routines Section Entry-Points

Each of the above routines may return an exception to the window manager using the EXIT statement. EXIT WITH 1 has a general "incorrect - do not proceed" meaning. Other exit conditions are also used, and these are further explained below.

6.4.1 The Before Routine

The Before Routine is called immediately before the field is accepted or displayed. Returning control with exit condition 1 (i.e. performing EXIT WITH 1) causes the field to be suppressed. When suppressed, the area on the screen normally occupied by the field is cleared, and no further processing takes place for it. Note that Before routines are not called in ENQ mode processing. Returning control with exit condition 2 causes the field accept operation to be suppressed as if the PRO option were coded.

6.4.2 The Default Routine

The Default Routine is called before a field is processed and allows a default to be provided. Note that the routine is only called during ADD and INS modes, since fields are regarded as pre-initialised during MNT mode. The default is simply MOVE'd into the field and the operator may accept or change it. Before moving a default value into the field, the default routine should check that the field has not already been initialised. Uninitialised fields will be set to spaces if character fields and zero for computational and display numeric fields. It is also important to note that this routine is **not** called in Maintenance mode.

[Earlier versions of this manual included the phrase: It is also important to note that this routine is called during record maintenance. In maintenance mode, most if not all of the fields will already contain correct values by virtue of having been read from disk. The routine should therefore ensure that new defaults are only provided when necessary. If no default is provided by the routine, exit condition 1 should be returned.]

6.4.3 The Validation Routine

The Validation Routine is called immediately after a field has been accepted, and may be used to perform additional validation such as range checks. Returning exception 1 indicates that the field is invalid, and causes it to be re-input. Note that the validation routine is called even if it is not possible to ACCEPT the field (e.g. a protected field).

6.4.4 The Fetch Routine

The Fetch Routine is called whenever the window manager fetches a record from the database. The routine may be used to create derived fields before the record is displayed. Where special display formats are required, the routine may also be used to convert fields from database to external formats. It is called immediately after retrieval of the target record, but before any other processing has taken place.

The routine can also be used to suppress the retrieval of certain records, such as suppressing inactive customers. This is achieved by returning exit condition 1, and causes the window manager to proceed as if the record did not exist. This allows a selection of records to be displayed during enquiry operations.

6.4.5 The Select Routine

The Select Routine is called when the operator attempts to select a record, usually to enter MNT mode. The record will already have been fetched and is displayed. The routine may be used to stop the operator from selecting the record, and this is achieved by returning exit condition 1. For example, this might be useful to stop the operator from attempting to amend an invoiced order.

6.4.6 The Delete Routine

The Delete Routine is called after the operator has keyed <DEL> to delete the current record, but before the deletion is performed by the window manager. The routine may reject the deletion request by returning exception condition 1. For example, this may be required to stop the deletion of an invoice before it has appeared on a statement.

The routine may also be used to remove unwanted servant records. For example, when the operator requests the deletion of an order header, the routine might automatically delete all servant order lines thus allowing the deletion to succeed.

6.4.7 The Write Routine

The Write Routine is called immediately before a new record is written to the database during ADD or INS mode. The routine may be used to complete fields on the record, before it is actually written (e.g. calculate the extended value of a line item). Returning exit condition 1 returns the operator to the last input field on the record, and suppresses the write operation.

6.4.8 The Rewrite Routine

The Rewrite Routine is called immediately before an existing record is re-written to the database during MNT mode. This routine is otherwise identical to the write routine.

6.4.9 The Process Routine

The Process Routine is called **after** processing has been completed (i.e. on completion of ADD, INS, EDT, MNT or DEL modes). It may be used to perform additional updates, such as writing details of the transaction to an audit log. The system variable $MODE may be examined by the process routine to determine in which mode the record was processed if this is important, see Section 6.3.2.

When the routine is called, the record that has just been processed by the window manager is still contained within the I/O channel. The fields of the record may therefore be examined by the routine. It is important to note, however, that the I/O operation (i.e. write, rewrite or delete) will already have taken place. In the case of deletion, this means that the record no longer exists on the database at the time that the process routine is called.

Following ADD, INS, and MNT modes, the current record in the I/O channel will normally be locked, to ensure that it is not modified or deleted while the process routine executes. For ADD, INS and MNT modes, the record will be locked exclusively. For SEL mode the record will be protected (locked non-exclusively) unless the LOCK or NOLOCK options are in force.

Returning Exit condition 1 (i.e. performing an EXIT WITH 1) from the process routine returns unsuccessful completion of the window, and causes the back-action to be taken as defined by the window's optional Sequence statement. It should be noted that this has no effect on the processing of the last transaction, which will already have been written to the database.

Returning Exit condition 2 causes window processing to be terminated just as if the operator had keyed a series of <BCK> keys to terminate the current series of windows as defined by the window's sequence statement. Where no sequence statement has been coded, the effect of this is identical to returning exception condition 1.

6.4.10 The Terminate Routine

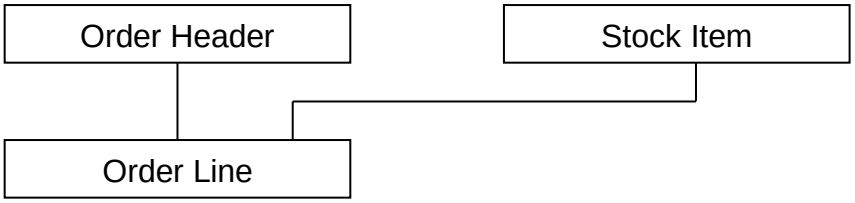
The Terminate Routine is called when the operator terminates record processing in MNT, EDT, ADD or INS modes by keying a function such as <BCK>, <HME>, <CLR>. It may be used, for example, to release locks established by earlier, now aborted, record processing, such as may have occurred within the window's validation routine. System variable $FUNC may be examined to determine the function used to terminate record processing. Note that the variable $MODE is not defined when the R-TERM routine is processed. Note that the R-TERM routine must not be used to unlock any window target record type.

6.5 Processing Database Records

This section discusses the operation of the window construct when used to process records with masters. The locking requirements of such records are discussed in section 6.5.1 with an explanation of the automatic retrieval that takes place in display mode. Section 6.5.2 describes the operation of dependent windows, used to process subsets of records.

6.5.1 Locking Master Records

As described in Chapter 2, special locking requirements exist when processing records that are linked to master records. For example, consider the following structure:



In the above structure, the Order Line record has two masters, the Order Header record and the Stock Record. When an order line record is added to the database, these masters must be locked before the I/O operation takes place. This ensures that a master record is not deleted by another user while the order line record is being added to the database.

Similar considerations also exist when maintaining a record. If for example, the stock number can be changed during maintenance, the **new** stock record must be locked before the order line is written to the database. Ensuring that these master records are correctly locked is the responsibility of the **application programmer**, and the simplest way to achieve this is to code FETCH statements in the validation routines of the appropriate fields.

If fields that are not stored on the target record are displayed (e.g. the display of the stock description from the stock record within the line item window) the application programmer must also ensure that the appropriate stock record is retrieved during processing. The simplest way of dealing with these "off-record" fields involves naming these record types as part of the window statement definition, and this is discussed in section 6.6.1.

Another method of fetching these additional records is to write a fetch routine in the routines section. The fetch routine will be called by the window manager each time a target record is

retrieved. Associated records can therefore be retrieved within this routine by coding an appropriate FETCH or GET statement.

This does, of course, have some bearing on performance. If it is necessary to retrieve, for example, eight associated records for each line within the window, then this will be much slower than just retrieving the displayed target record. When coding scrolled screens this can be particularly important, given that a number of records, and all their masters, will need to be fetched.

In order to improve paging performance within scrolled screens it is a good idea to place these fields within the non-scrolled area. These non-scrolled fields are only displayed once when the window is paged, in order to display the current record. The associated records then only need to be fetched once, improving performance considerably.

With this approach, a before routine should be coded for the appropriate off-record fields. This routine is called immediately before the field is displayed, and should therefore contain the fetch or get operation needed to retrieve the record. As record displays always take place in display mode, it is only necessary to perform these I/O operations in this mode.

6.5.2 Dependent Windows

A window generally allows access to **all** records of a particular type stored on the database. For example, a window processing invoice records will normally allow all invoices on file to be processed, subject only to the operation of the fetch routine within the routines section. It is quite a usual requirement however, for a window to process a subset of those records only, for example, allowing access **only** to invoices for a **particular** customer.

This requirement exists in many data-processing functions. For example, an invoice entry program composed of an invoice header window and a detail line window would have this requirement. When the invoice line window is entered, it should provide access **only** to the detail lines for the current invoice.

Dependent windows provide the facilities to achieve this effect. When the window construct is coded, dependency on a higher level record within the database is established using the WINDOW statement. For example:

```
WINDOW W1 USING IN DEPENDENT ON CU
```

This statement introduces the window W1, with a target record type IN (invoice record). The DEPENDENT ON clause restricts the operation of the window to a subset of records, in this case to those invoices that belong to record CU (customer record).

When the window is entered, an enquiry will return only invoice records that belong to the current customer record. The window manager does this by examining the I/O channel of the customer record, and thus determines **which particular** customer record is currently being processed. Only invoices that belong to this particular customer can then be retrieved or added.

Before entering window W1, a customer record must therefore be established, and this is done simply by reading the required customer record, using either the FETCH or READ statements. Alternatively, a preceding window may be used to do this (e.g. a selection window operating on customer records) which must be entered before the invoice window is executed.

Dependent windows can only operate if on the target record an index exists which starts with the same fields as the primary index of the "Dependent On" record type. For example, if the primary index of the customer record is the customer number, then an index must exist on the invoice record that also starts with this customer number. Such an index is needed to enable the efficient retrieval of invoices for the customer. If such an index does not exist, dependency cannot be used.

This implies that there is a master/servant relationship between the records, but there is no requirement for a formal linkage since all processing is performed using indexes. This feature operates simply by prefixing the indexes declared for the target record with the primary index of the "Dependent On" record type. The window manager then checks all retrieved records to ensure they conform to this prefix.

When processing dependent windows, the fields of the controlling key are moved automatically into the appropriate fields of the target record at the start of record processing. This saves the application programmer the task of initialising these fields before the record is written to the database.

6.6 Window Construct Syntax

The window construct is coded within the Data Division. Its general format is as follows:

```
[$OPT NSW]
WINDOW id [USING rt1 | rtidx [rta rtb... rtn] [DEPENDENT ON rt2 | (fielda, ... fieldn)]]
[\window help-text [$TR-key]]
[SEQUENCE id1 [Clear-opt], id2 [Clear-opt]]
[POP-UP | QREM]
[EDT] [ADD] [INS] [ENQ] [SEL] [MNT] [DEL] [UDL]
[REPEAT [UNTIL [NEXT | CURRENT RECORD]]]
[LOCK | NOLOCK]
[SCROLL n1 [BY n2] [SPLIT n3 OFFSET n4]]
[SBOX]
[LINE l1 [l2 ... l8]]
[BASE AT line col]
[ENABLE [NXT] [ABO]]
[DISABLE [SKP] [CLR] [HME]]
[AUTOPGE | AUTOBPG]
[LI CL Longname   LI CL Name   Pic Options...]

(Detail Lines)

[ROUTINES SECTION]

(Routines Section entry-points)

ENDWINDOW
```

The construct is introduced by the WINDOW statement and may optionally be preceded by the compiler directive $OPT NSW. If coded this directive causes the compiler to exclude the window from the disk swap file. You may wish to exclude a pop-up from the swap file, see section 4.6, if it is important for it to be displayed and removed in the shortest possible time. The WINDOW statement is followed by optional *window help-text*. This in turn is followed by a

number of **optional clauses** which specify a number of run-time parameters, such as in which modes the window may be operated. The **window-body** is then introduced by an optional header line (shown as LI CL Longname... above). This is in turn followed by the actual text and data fields to be displayed and accepted. The optional **Routines Section** may then be coded. This section contains additional procedural code which is executed during window processing. The ENDWINDOW statement indciates the end of the window construct, which may be followed by further window definitions, or the Procedure, Load or Unload Divisions.

6.6.1 The Window Statement

The WINDOW statement introduces the window construct and specifies its target record type. It is coded:

WINDOW id [USING rt1 | rtidx [rta rtb... rtn] [DEPENDENT ON rt2 | (fielda, ... fieldn)]]

where *id* is a unique two-character window-id by which the window is identified, *rt1* is the name of the target record, and *rta* to *rtd* are master records to be fetched with the target record *rt1*. *rt2* is the record-id of the record type upon which displays are dependent unless a field list, *fielda*, ... *fieldn* is defined. A default index, *rtidx* may be specified as the default index for use in enquiries by coding an index name instead of *rt1*.

The window-id is a two-character alphanumeric name which must start with an alphabetic character, and is used to reference the window during processing, such as when using the ENTER verb. It must therefore be unique amongst the windows declared within the frame, and should not be the same as any record-id referenced by the ACCESS statement.

The optional USING clause specifies that the record-id of the window's target record type is *rt1*, access to which must previously have been declared using the ACCESS statement. When used instead of *rt1*, the index name *rtidx* simultaneously defines the record-id and default current index by which records are to be retrieved using the <PGE> and <BPG> operations. If the record-id is coded, the first index defined for that record in the database dictionary will be used as the default index.

Up to four master records which are to be retrieved with the target record are specified by *rta* to *rtd*. Whenever the window manager fetches the target record, these master records will also be retrieved, in much the same way as the READ verb. This facility is used to display fields that are not stored on the target record (e.g. when displaying the stock description from the stock record within order line items). Note that records retrieved using this facility **will be unlocked**.

This facility can only be used for records that are directly linked to the target record type. If this is not the case, the same effect can be achieved by coding a fetch routine within the routines section. This routine must then perform the necessary fetch or get operations to cause the required records to be retrieved.

Omitting the USING clause indicates that the window is not associated with records residing on the database. The window will therefore operate in ADD and EDT modes only, and any necessary I/O operations will have to be explicitly coded within the frame.

The optional DEPENDENT ON clause has two different uses. If coded in conjunction with record type *rt2*, it restricts access to the group of records belonging to record *rt2*, which must be established before the window is executed. For example, it may be desirable to process only

those invoices that belong to a given customer. This is achieved by first reading, or creating that customer record, and then invoking the invoice window.

When the DEPENDENT ON clause is used with record-id *rt2*, only indexes that begin with the primary index of *rt2* will be used to retrieve the target record type. In other words, an index must start with the same fields as the primary index of the master record. Returning to the example, enquiries can use only those indexes that start with the customer number. This allows the window manager to get straight to the invoices belonging to the customer without having to scan through the entire file. If no such index exists this clause may not be used. If index *rtidx* is specified instead of *rt1*, it must also meet these criteria.

The DEPENDENT ON clause may instead be followed by a list of fields containing the significant portion of the index key upon which the operation of the window will be dependent. For example, coding:

```
WINDOW W1 USING IN DEPENDENT ON (Z-CUST)
```

makes the operation of this window dependent on the value contained in field Z-CUST, which must be set up before the window is entered.

When compiling this statement, the compiler searches for an index which starts with index segment named INCUST, which must have the same picture clause as the field Z-CUST. The important point here is that the last four characters of the field name serve to identify the corresponding index key segment as stored on the target record. These requirements are conceptually similar to the establishment of Master Access Keys, described in Appendix F.

Where a list of fields is coded to define dependent operation, each field must be defined in the same order as at least one index supported by the target record type. The picture clause of each field must match the picture clause of the corresponding index key segments, and the specified fields must be **located in contiguous memory locations**. For example, let us create an order line item window which shows only the order lines for a given order. Given that the order line record contains an index composed of:

Customer Number	OLCUST X(4)
Order Number	OLORDN X(5)
Stock Number	OLSTCK X(8)

we must specify a window which is dependant on **both** the customer and order number fields. This should be done by coding the following in the Data Division:

```
01    Z-KEY                  * Dependency key for order lines
  03  Z-CUST      PIC X(4)   * - Customer number
  03  Z-ORDN      PIC X(5)   * - Order number
```

The Window Statement would then be coded:

```
WINDOW W1 USING OL DEPENDENT ON (Z-CUST, Z-ORDN)
```

During compilation, the compiler identifies the appropriate index from the names of the specified field segments, and checks that the picture clauses are identical. At run time, it is then necessary to move the required customer and order numbers into Z-CUST and Z-ORDN respectively, prior to entering the window.

6.6.2 Optional Clauses

A number of optional clauses may be coded following the WINDOW statement. These are:

[*window help-text* [$TR-key]]
[SEQUENCE *id1* [*Clear-opt*], *id2* [*Clear-opt*]]
[POP-UP | QREM]
[EDT] [ADD] [INS] [ENQ] [SEL] [MNT] [DEL] [UDL]
[REPEAT [UNTIL [NEXT | CURRENT RECORD]]]
[LOCK | NOLOCK]
[SCROLL *n1* [BY *n2*] [SPLIT *n3* OFFSET *n4*]]
[SBOX]
[LINE *l1* [*l2* ... *l8*]]
[BASE AT *line col*]
[ENABLE [NXT] [ABO]]
[DISABLE [SKP] [CLR] [HME]]
[AUTOPGE | AUTOBPG]

6.6.2.1 Window Help-text

Window help-text lines have the backslash character \ as the first significant character on the line, must be contiguous, and follow immediately after the WINDOW statement. Key-top names may be embedded in the help-text by coding $TR-key where key is any of the function-key mnemonics listed in Table 6.3b. For example, to display the help-text "Key End for next window" you should code "\Key $TR-NXT for next window". The help window is displayed when <HLP> is keyed twice (i.e. <HLP><HLP>).

6.6.2.2 Sequence Clause

This clause defines the sequence in which the window is to be executed. It is coded:

SEQUENCE *id1* [*Clear-opt*], *id2* [*Clear-opt*]

where *id1* is the window-id to be entered on unsuccessful completion of the window, and *id2* is the window-id to be entered following successful completion of the window. The keyword EXIT may be coded for either *id1* and/or *id2*, and this causes the window manager to return control on completion, instead of executing a further window.

Clear-opt specifies a window clearing action to be taken on completion of the window and may be one of the following:

CLR Clear Screen. The screen is totally cleared, leaving only the screen header displayed.

CLW Clear Window. The window is removed from the screen. If the window is a POP-UP, the prior image is re-displayed. Otherwise the area occupied by the window is over-written with spaces in the window background attribute.

CLD Clear Data. Data displayed within the window is cleared. If the sequence statement is omitted, an exit will take place both on successful and unsuccessful completion, and no clearing action will take place.

6.6.2.3 Clearing controls

The POP-UP and QREM clauses are used to control the way a window is cleared from the screen. The POP-UP clause causes the screen image under the window to be saved when it is activated. When the window is cleared using the CLW option (see below) or the CLEAR *window* statement, this image is re-displayed, thus resetting the screen to its prior state.

The QREM clause is used to improve clearing performance. When a ordinary window (i.e. not a POP-UP) is removed using the CLW option or CLEAR *window* statement, the area underneath it is cleared by displaying spaces. This can take some time, especially if the window is large. The QREM clause causes the window to be removed using the clear-to-end-of-line facility, which operates much faster, but also has the effect of clearing the area to the right of the window. This option should therefore only be used when the area to the right of the window is otherwise unused.

6.6.2.4 Mode Enabling Clauses

This section describes the following optional processing modes:

EDT
ADD
INS
ENQ
SEL
MNT
DEL
UDL

If **none** of the above clauses is coded, defaults are allocated as follows. If the USING clause has been coded in the window statement (i.e. the window operates on a target record type) all clauses other than EDT and UDL are enabled. Otherwise only the ADD clause is enabled. These defaults are over-ridden by coding any of the above clauses which are described below:

6.6.2.4.1 The EDT Clause

The **EDT** clause enables edit mode, used only on windows that do not operate on a target record type and should not be used in scrolled windows. It allows the data fields processed by the window to be pre-initialised using the display verb and entry into the window then allows the operator to edit the displayed fields as a whole. This clause is typically used in windows which accept run-time parameters (e.g. those required in a print program). Coding EDT automatically enables ADD mode.

6.6.2.4.2 The ADD Clause

The **ADD** clause enables add mode, normally entered when the cursor is positioned on a blank record. This allows the operator to enter the fields on the record, on completion of which a new record is written to the database. Unless this mode is enabled addition of new records cannot take place, so only existing records can be processed by the window.

6.6.2.4.3 The INS Clause

The **INS** clause enables insert mode which allows a new record to be inserted before an existing record displayed in the window. When the operator keys <INS>, the window is scrolled apart to create a new blank record display area into which the new record can be inserted. Since the insert instruction requires existing records to be displayed, the INS clause also enables ENQ and DSP modes. ADD mode is also enabled by this instruction.

6.6.2.4.4 The ENQ Clause

The **ENQ** clause enables enquiry and display modes. Enquiry mode is activated whenever a database search is initiated, and is therefore required in order to display existing records within the window. This mode is also entered explicitly when <ENQ> is keyed. If the database search is successful, display mode is activated in order to display the retrieved records.

If the ENQ clause is not coded, the user will be unable to retrieve and display records from the database, and is therefore restricted simply to adding new records. Since enquiry mode requires a target record type, the window statement must contain the USING clause.

6.6.2.4.5 The MNT Clause

The **MNT** clause, when coded, the operator may key <RET> to select the current record for maintenance. Unprotected fields on the record may then be edited, on completion of which the record is re-written to the database. Since maintenance operates on existing records, ENQ and DSP modes are automatically enabled.

6.6.2.4.6 The SEL Clause

The **SEL** clause allows an existing record to be selected from the window. This clause is coded when a window is used for selection purposes only, an example of this being the selection of a customer prior to invoice entry. It is used instead of the MNT clause when no maintenance is to take place on the selected record. The window manager suppresses record editing and the re-write of the record that would otherwise take place. Since record selection operates on existing records, ENQ and DSP modes are automatically enabled. Unless the SEL clause is coded, the operator is able to position the cursor on a different record within the display, but is unable to select it.

6.6.2.4.7 The DEL Clause

The **DEL** clause allows the operator to select a record for deletion by placing the cursor on the required record, which is then deleted by keying <DTE>. If this clause is omitted, record deletion is disabled. Since deletion operates on existing records, ENQ and DSP modes are automatically enabled.

6.6.2.4.8 The UDL Clause

The **UDL** clause allows the operator to undelete the last deleted record by keying <UDL>. Coding this clause causes the compiler to create an area within the frame which will contain a copy of the last deleted record. Keying <UDL> activates ADD or INS mode, each field on the record being moved from the saved area instead of being accepted from the operator. The UDL clause automatically activates ADD, INS, ENQ, DSP and DEL modes.

6.6.2.4.9 Programming Notes

If the window does **not** have a target record type (i.e. the USING clause was not specified in the WINDOW statement) the window may only operate in ADD and EDT modes. Coding any of the clauses INS, ENQ, SEL, MNT, DEL, UDL will therefore result in an error during compilation.

The MNT and SEL clauses are mutually exclusive, and may not be coded for the same window. The SEL clause means that MNT mode is **not** to be entered following selection of a record. Coding SEL alone means that the window may only be used to do an enquiry and select a particular record. This option is often used to set up a controlling record for use by a subsequent dependent window (e.g. to select a given customer prior to processing that customer's invoices in a subsequent window).

If ENQ is the window's only valid mode, the processing cycle described earlier in this chapter can never complete, since the window simply stays "stuck" in enquiry mode. However, if the REPEAT UNTIL NEXT clause was coded, the operator will be able to use the <NXT> key to indicate successful completion. Otherwise, **successful completion cannot occur**.

6.6.2.5 Record Status Controlling Clauses

The following clauses control record lock status and window termination. They are coded:

```
REPEAT [UNTIL [NXT | CURRENT RECORD]]
LOCK | NOLOCK
```

6.6.2.5.1 The REPEAT option

The **REPEAT** option causes the window to loop until a terminating condition is reached. If the option is not coded, successful completion will be returned following a single processing cycle. When the REPEAT clause is coded on its own, the processing cycle will continue indefinitely, and no successful exit is therefore possible.

6.6.2.5.2 The REPEAT UNTIL NXT option

The **REPEAT UNTIL NXT** option enables the <NXT> key, and allows the operator to indicate completion of the window at any time.

6.6.2.5.3 The REPEAT UNTIL CURRENT RECORD option

The **REPEAT UNTIL CURRENT RECORD** option specifies that a current record is required before successful completion is permitted. When coded, this option ensures that a valid target record is contained in the I/O channel, and this record will be locked in accordance with the locking options discussed in the following section. This option must therefore be coded whenever the status of the I/O channel on successful completion is important.

Note that this option has no effect on the operation of the <ABO> and <BCK> keys, which may be used to force unsuccessful completion of the window at any time. The status of the I/O channel on **unsuccessful** completion of a window is therefore undefined under all circumstances.

6.6.2.5.4 The LOCK and NOLOCK option

The **LOCK** and **NOLOCK** options are used to specify the lock level required when a record is selected without further processing. When a record is added, deleted, or maintained a full (exclusive) lock is always placed on the record. The locking options are therefore only used with the SEL option described in section 6.6.2.4.

When the SEL option is coded, a selected record will normally be delete-protected (non-exclusively locked) so that it cannot be deleted by another concurrently executing frame. Coding the LOCK option causes a full lock to be placed on the record. Coding the NOLOCK option suppresses locking of the target record.

Note that the record lock, specified using the above options or as defaulted, is normally released on completion of the window. It will only be retained if REPEAT UNTIL CURRENT RECORD is coded. The lock status of the I/O channel following unsuccessful completion is undefined.

6.6.2.6 The SCROLL statement

The SCROLL statement is used to define a scrolled region within the window. It is coded:

SCROLL *n1* [BY *n2*] [SPLIT *n3* OFFSET *n4*]

where *n1* is the number of records in the scrolled area, each of *n2* contiguous lines. The number of columns in which the window is arranged for vertical split scrolling is defined by *n3* and the offset between these columns by *n4*.

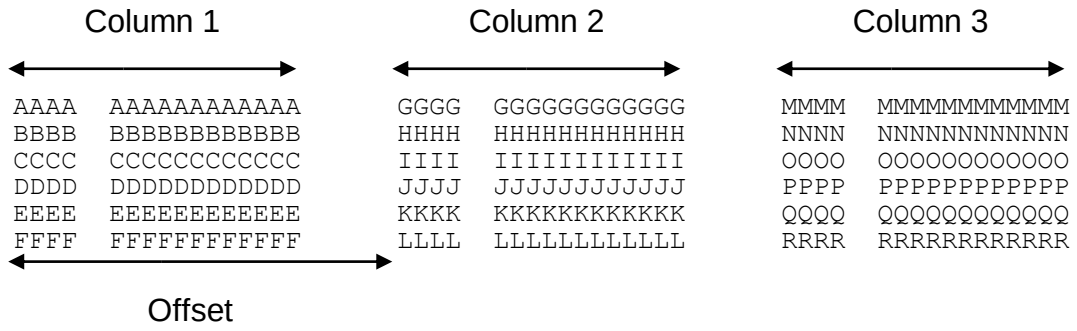
The scroll statement specifies that the **data fields** defined later within the window are to be scrolled, thus allowing multiple records to be displayed within the window at once. Note that **text fields are never scrolled**, and are therefore always displayed at the coded line and column positions.

The number of records to be displayed is defined by variable *n1*. Normally, each record will take up only one line within the display, but multi-line records can be specified by using the BY *n2* clause. Thus, if the scrolled region is to contain eight records, each using two physical lines on the screen, the following would be coded:

SCROLL 8 BY 2

The scrolled region would therefore occupy sixteen lines (8 x 2).

The Window Manager allows this scrolled region to be split vertically into several columns, and this is achieved by the SPLIT clause. The number of columns into which the scrolled region is to be split is specified by variable *n3*, and the vertical displacement between the two columns is specified in characters by variable *n4*. For example, consider a scrolled region split into three columns:



This scrolled region contains eighteen record display areas (areas AAAA to RRRR), each taking up one line. The region has been split into three columns, and the offset between the columns is twenty characters. This is coded:

SCROLL 18 BY 1 SPLIT 3 OFFSET 20

The BY 1 clause could be omitted, since this is the default.

The SCROLL statement applies to all the following data fields coded within the window, except those with the NSC option - see Section 6.6.4.2. It is normal for a window to contain both scrolled and non-scrolled fields, and these do not necessarily have to be coded in order. For example, the window may have a few scrolled fields, followed by a few non-scrolled fields, and ending again with scrolled fields.

6.6.2.6 The SBOX and LINE Statements

All windows are displayed with a box around the outer extremities of the displayed text and data items. The dimensions of the box are calculated by the Speedbase compiler so that the top and bottom lines of the box are immediately above and below the first and last used display lines respectively. The vertical lines of the box are normally displayed two characters before and after the first and last used column respectively. Fields should therefore start at, or after, character position three on the screen, and should not be placed higher than line two, or line three if a screen header is also displayed.

6.6.2.6.1 The SBOX Statement

The **SBOX** statement causes the box to be drawn immediately to the left and right of the first and last used column positions, thus creating a slightly narrower box than would normally be produced. This allows data and text to be displayed from column position two as opposed to column three. If used, the SBOX statement must be coded prior to any text or data item definitions.

6.6.2.6.2 The LINE Statement

The **LINE** statement specifies horizontal lines which will be joined to the vertical lines of the window's box. It is coded:

LINE *l1* [*l2* ... *l8*]

where *l1* to *l8* are up to eight line numbers at which lines are to be drawn.

6.6.2.8 The BASE AT Statement

The **BASE AT** statement specifies the window position offset. It is coded:

BASE AT *line col*

where *line col* is the line and column position of the top left-hand corner of the window (e.g. coding BASE AT 9 20 places a window 8 lines down and 19 characters towards the right of the screen).

6.6.2.9 The ENABLE Statement

The ENABLE statement is used to enable the <NXT> and <ABO> functions. Coding:

ENABLE ABO

has the effect of enabling the Abort function, which allows the operator to abort a chain of windows. This function avoids the user having to key <BCK> several times in order to terminate a series of windows.

Windows supporting MNT mode normally cause the operator to enter maintenance mode on selection, before the next window is entered. Coding:

ENABLE NXT

allows the operator to select the current record using the <NXT> key while in display mode. The window manager then automatically skips through all the fields on the record just as if the operator had keyed <RET> in response to each field. This feature is useful when processing master/servant windows, when maintenance is not necessarily carried out within the initiating window.

6.6.2.10 The DISABLE Statement

The DISABLE statement allows the <SKP>, <CLR> and <HME> options to be disabled. It is coded:

```
DISABLE [SKP] [CLR] [HME]
```

The statement simply removes these function keys during processing of the window. This is useful in simple windows such as menus, where functions such as <CLR> could cause confusion.

6.6.2.11 The Auto-page Statements

The AUTOPGE and AUTOBPG statements cause a page or back-page operation to occur when the window is initially entered, as if <PGE> or <BPG> had been keyed. These options are useful, for example, in enquiry windows, where it is often convenient to display the first or last page of records on entry. The options are mutually exclusive, and operate only when a clear (i.e. empty) window is entered.

6.6.3 The Window Body

The window body contains details of the text and fields that are to be displayed and accepted within the window, and may contain options that control its general layout. The format of the window body is as follows:

```
LI CL Longname LI CL Name Pic Options...

(Detail Lines)
```

The window body may be introduced by a comment line which may be helpful in laying out the source code. The compiler will ignore any line at the start of the window body that begins with the characters LI. This is then followed by the window's detail lines which are described in the following section.

6.6.4 Window Detail Lines

The detail lines define both the window's form (i.e. fixed text items with their boxes) and data items (i.e. variables that are to be displayed and accepted). A detail line consists of one or more text items, and/or one or more data-items. Text items specify fixed text which will be displayed at a fixed position on the screen. Data-items specify variable data which may be accepted and displayed on the screen in various positions.

6.6.4.1 Text Items

Text items are used to display fixed text on the screen at a given line and column position. The general form of a text-item is as follows:

```
line col "text"
```

where *line* is the line number (counting the top line as 1) at which the text is to be displayed, *col* is the column number at which the text is to be displayed, and *text* is the text to be displayed at that position, as offset by the BASE AT statement. For example, coding:

```
10 20 "Customer #"
```

would cause the characters **Customer #** to be displayed on the screen at line 10, column 20.

Fixed text items are normally displayed on the screen when the frame is executed. The basic screen form, composed of all the text items, boxes and lines of all the coded windows within the frame is displayed in this way, providing the operator with an "empty" screen. Text items are always displayed at the coded line and column positions, and are therefore unaffected by the SCROLL verb.

Line and column numbers must both be unsigned positive integer literals. Line numbers may be coded from 2 to 46 inclusive. Column numbers may be coded from 2 to 127 inclusive, so long as the item being displayed, as offset by the BASE AT statement, would not cause column 131 to be exceeded.

6.6.4.2 Data Items

A data-item specifies a variable that is to be displayed and/or accepted from the screen. The general form of a data item is:

line col name [*options*]

Where *line col* is the line and column number at which the item is to be displayed, as offset by the BASE AT statement, *name* is the variable name of the item to be displayed and/or accepted, and *options* consists of one or more option clauses which specify how the item should be processed.

6.6.4.2.1 Line and Column Numbers

Line and Column Numbers must be unsigned positive integer literals. Line numbers may be coded in the range of 2 to 46 inclusive. Column numbers may be coded in the range 2 to 127 inclusive, as long as the item displayed would not cause column 131 to be exceeded. It should be noted that the line and column position coded will be modified at run-time by the operation of the SCROLL verb, as documented earlier in this chapter, and offset by the BASE AT statement.

6.6.4.2.2 Variable Name

The Variable Name is a **reference** to an existing variable that has previously been declared within the Data Division or by an ACCESS statement.

6.6.4.2.3 Field Options

Field Options may be coded following the picture clause in order to affect the way the field is processed at run-time These are:

DIS	Display only field, the field is only displayed
PRO	Protected field, the field cannot be entered or amended
NOE	No-edit field, the field may not be amended
NUL	Null allowed (i.e. the field may be left blank)
TAB	Stops the cursor at this field after a <SKIP>
CHK	Perform duplicate record check after this field
UF1	Enable UF1 when this field is accepted
UF2	Enable UF2 when this field is accepted
UF3	Enable UF3 when this field is accepted
NSC	The field is non-scrolled in an otherwise scrolled window
CNV	Lower-case characters to be converted to upper-case
YN	Yes/No validation to be performed on field input

TXT	A text item specified in the form of a variable
RJF	Right-justify field
FMT	Specifies the display formatting to be applied to the field
HOT	Specifies field auto-inputs without need to key <RET>.
TTL	Display field using display attribute 5 (Titles).
ERR	Display field using display attribute 8 (Error message).
A12	Display field using attribute 12.
A13	Display field using attribute 13.
A14	Display field using attribute 14.

These options are discussed below.

6.6.4.2.3.1 DIS Option

DIS (Display) is similar to a protected field in that the operator will not be able to modify it. The display option is used, however, to indicate that the field contains valid data and should not be initialised during ADD mode. If despite this, a default routine has been coded, then this will nevertheless be called.

6.6.4.2.3.2 PRO Option

PRO (Protected) specifies that the field may not be modified by the operator under any circumstances. During ADD mode, the field will be initialised to binary zeros, spaces or "0". If a default routine has been coded, it will be called to initialise the field instead.

6.6.4.2.3.3 NOE Option

NOE (No Edit) prohibits the user from modifying the field in MNT and EDT modes, but allows normal access to the field during ADD mode. It is used to protect fields that may not be changed after the record has been written.

6.6.4.2.3.4 NUL Option

NUL (Null allowed) specifies that the field may be left blank, in the case of a character field, or zero, in the case of a numeric field. In the case of dates, this option allows a null date, " / / ", to be entered. If this option is not coded the window manager will re-input the field if any blank or zero values are entered, or defaults accepted by the operator.

6.6.4.2.3.5 TAB Option

TAB causes the cursor to stop at the current field whenever the <SKP> key is used, and therefore acts as a tab-stop. This allows fields to be arranged in blocks within the window, allowing the operator to skip from one block to the next. If the TAB option is not used in the window, then the <SKP> operation will automatically jump to the last field.

6.6.4.2.3.6 CHK Option

CHK causes a duplicate-record check to take place during processing of the associated field in ADD and INS modes. This option is used to check that the record being created does not already exist on the database. In effect, a duplicate key check takes place using the index key entered so far. If this key is found and ENQ mode is enabled, the baseline message:

Record key already exists - Enquire?

is displayed. The operator may then enquire on the duplicate record. This option should be coded **after** all primary index key fields have been entered. This option is particularly

recommended for maintenance programs, since it allows the operator a fast way of enquiring on records without having to use the ENQ key to switch between modes.

6.6.4.2.3.7 UF1 Option

UF1 enables the use of the <UF1> key when the associated field is accepted. If coded, a validation routine should be written within the routines section, which should examine system variable $FUNC to check whether this key had been entered, and if so, take the appropriate action.

6.6.4.2.3.8 UF2 Option

UF2 enables the use of the <UF2> key when the associated field is accepted.

6.6.4.2.3.9 UF3 Option

UF3 enables the use of the <UF3> key when the associated field is accepted.

6.6.4.2.3.10 NSC Option

NSC specifies that the field is not to be scrolled within a window coded using the SCROLL clause. If the SCROLL clause has not been specified, all fields are automatically non-scrolled, and there is therefore no point in coding this option. Option NSC is used to define those fields that will be displayed in the non-scrolled portion of the window, as described in section 6.2.1.

6.6.4.2.3.11 CNV Option

CNV specifies that any lower-case characters input are to be converted to upper-case before the field is validated, or any other processing is carried out.

6.6.4.2.3.12 YN Option

YN causes the field to be re-input unless it is Y or N.

6.6.4.2.3.13 TXT Option

TXT specifies that the field variable is to be displayed as text in the scrolled-text attribute unless the TXT option is immediately followed by the NSC option, in which case it is displayed in the non-scrolled-text attribute. This option is used where the text to be displayed varies under program control.

6.6.4.2.3.14 RJF Option

RJF causes the field to be accepted, stored and displayed in right-justified form.

6.6.4.2.3.15 FMT Option

FMT "*options*" specifies the output formatting to be applied to the field. Table 6.6a lists the five groups of *options*.

6.6.4.2.3.16 HOT Option

HOT Field entry terminates as if <RET> entered when last byte of the field is keyed. This saves having to key <RET> for the field.

Option	Condition	Output Formatting	As input	As output
B	*Field* = 0	**Blank** when zero	00.00	
C	*Field* < 0	Trailing **CR**	-12.34	12.34CR
<		Enclosed in **()**		(12.34)

-		Trailing **-**		12.34-
D	*Field* > 0	Trailing **DR**	+56.78	56.78DR
>		Enclosed in **()**		(56.78)
+		Trailing **+**		56.78+
,	None	**Comma** insertion	90123	90,123
$	None	Leading **dollar**	45.67	$45.67
0		**Zero** fill	8.90	0008.90
*		**Asterisk** fill	1.23	***1.23
L	Date only	**Long** date	DD/MM/YYYY	DD/MM/YYYY
8	Date only	Short date **8** byte input	DDMMYYYY	DD/MM/YYYY
6	Date only	Short date **6** byte input	DDMMYY	DD/MM/YY

Table 6.6a - The Output Formatting Options

You may specify no more than one option from each group in any set of options. Examples of invalid options would include:

```
BC<+,$
->+*
BCD,0*
```

Examples of valid sets of options and the resultant output formatting are as follows:

Options	Field as Input	Field as Output
BCD,	-678.90	678.90CR
	0.00	
	+1234.56	1,234.56DR
B<,	+789.01	789.01
	-7890.12	(7,890.12)
B-+$,	+345.67	$345.67+
	$890.12	$890.12-
-+0	+345.67	00345.67+
	-8901.23	08901.23-
CD*	+456.78	**456.78DR
	-901.23	**901.23CR

Table 6.6b - Examples of Output Formatting

6.6.4.3 Data Item Processing

The processing applied to each data item depends on the mode the window is operating in at the time, the field options in force, and the operation of the field level routines within the routines section. The usual order of events is as follows:

1 The before-routine is executed to suppress the field

2 The field is initialised, to spaces or low-values, if it is being processed for the first time in ADD or INS modes

3 The default routine is executed to allow a new value to be placed in the field

4 The field is displayed and accepted

5 The validation routine is executed

6 If the CHK option has been coded for the field, a duplicate key check takes place at this time

The field options cause some steps to be omitted. The **DIS** option disables field initialisation, step 2, and stops the field from being accepted in all modes other than ENQ. The **PRO** option prevents the field from being accepted in all modes other than ENQ. The **NOE** option prevents the field from being accepted in MNT and EDT modes. The **CHK** option causes step 6 to take place. The **UF1**, **UF2** and **UF3** options enable the corresponding keys if the field is accepted.

Consider the effect the various modes have on the processing of data items:

ENQ starts an enquiry and data entry of index key fields

DSP displays existing records from the database

MNT causes modification of an existing record

DEL causes deletion of an existing record

ADD causes creation of a new record

INS causes creation and insertion of a new record

EDT causes creation of a new record by editing an existing one

In **ENQ** mode the before-routine is not called, since index key fields may not be suppressed at run time. The current index fields are accepted irrespective of options NOE, DSP and PRO.

In **DSP** mode, fields within the record are display-only. The before-routine is called to suppress display of certain fields, the validation and default routines being ignored. If it is necessary to default certain fields prior to display, this should be done using a fetch routine in the routines section. The field options have no effect on processing during DSP mode.

In **MNT** mode, fields coded with NOE, DSP and PRO options are not accepted. It should be noted, however, that the default routine is called. This feature may be used to re-calculate a protected field, such as a line total, during the maintenance process. If a field is neither accepted nor re-defaulted using its default routine, then it will **not be redisplayed** during processing.

In **DEL** mode, no field-level processing takes place.

In **ADD** and **INS** modes, fields coded with DIS and PRO options are not accepted. Fields coded with the DIS option are also not cleared at step four. If the CHK option has been coded, a duplicate key check will take place at step six.

In **EDT** mode, fields coded with DIS, PRO and NOE options are not accepted.

6.6.5 The Routines Section

The optional Routines Section is coded immediately following the window body. It is introduced by the header:

```
ROUTINES SECTION
```

The routines section consists of a number of routines which are called during the various stages of window processing. Each routine is identified by a special label which determines when during the processing cycle it will be executed by the window manager.

Each routine may contain any of the procedural instructions described in Chapter 7, and on completion of processing must return control to the window manager by executing an EXIT instruction. The various routines that may be coded are described in detail in section 6.5.

6.7 Programming Notes

6.7.1 Insert Mode Utilisation

INS mode is provided for use with auto-sequenced indexes, described in Appendix F. It is of limited use with ordinary indexed records since the position at which a record is inserted within a set of records is always determined by the index order. If INS mode is enabled within a window using a normally-indexed record structure, the operator will still be able physically to insert a record within the window, but on redisplay this record will appear in its normal index order.

6.7.2 Memory Considerations

The code generated by the window construct is very compact, and the only major consideration in program design lies in the use of pop-up windows. When pop-ups are activated, the screen image under the window is saved for later re-display. The Speedbase compiler makes space within the frame memory area for this purpose, which therefore reduces the space left for other tasks.

The memory area set aside equals three times the number of characters occupied by the pop-up, including it's box. This will be of some concern when very large pop-ups are used. For example, a pop-up measuring 20 lines by 60 characters will reduce the remaining available memory area by 3600 bytes (20 x 60 x 3).

When memory space is tight, it is therefore a good idea to keep the dimensions of pop-ups as small as possible. This will also reduce the time taken physically to display and remove the pop-up from the screen.

When very large record types are used, the undelete (UDL) option may also cause memory problems. This option causes the Speedbase compiler to create an area into which the last deleted record is copied, which will therefore reduce available memory by the size of the target record type.

6.8 Example Order Entry Program

Appendix C describes an example order entry program which makes use of the window facilities described in this chapter. This sample program forms part of the demonstration program S.DEMO which is distributed with the Speedbase Presentation Manager and the database DBDEMON.

New users of Speedbase are encouraged to review this sample program, referring back to this chapter in order to gain an understanding of the principles of window operation and other features of the Speedbase Presentation Manager.

7. Procedural Statements

This chapter describes the procedural statements that may be coded in a Speedbase Development Language frame. The chapter has been organised into three main sections. Section 7.1 explains where procedural instructions may be coded, and explains the use of sections, labels and called entry-points. Procedural statements are then explained in detail within their various classifications in the subsequent sections.

7.1 Structure

The statements described in this chapter can be coded in four divisions within each frame. These are:

WINDOW DIVISION
PROCEDURE DIVISION
LOAD DIVISION
UNLOAD DIVISION

Within the window division, procedural statements are introduced by the routines section statement. Elsewhere within the frame, procedural statements are introduced by the procedure, load and unload division statements. If the window division is to be used it must be coded first. The procedure, load and unload divisions may be coded in any order.

Each of the divisions may contain any of the procedure statements supported by the Speedbase compiler, described later in this chapter. The divisions are simply regarded as entry-points which are called as execution of the frame proceeds. This is explained in Section 3.4. Subroutines coded in one division may be performed, called or jumped-to from code in any other division, and this allows common routines to be shared as required.

7.1.1 Sections and Paragraphs

A section is introduced by the statement:

SECTION *section-name*

where *section-name* is a symbol as defined in Section 3.2.2.

A paragraph name is established by coding any valid symbol immediately followed by a full stop. For example:

SYMBOL.

Paragraph names may either be coded on their own line or may precede any procedure statement. Only one paragraph name may be coded on a line. Once established, paragraph and section names may be used as the target of the PERFORM and GOTO statements.

Paragraph names are also used to identify routines within the routines section. These special paragraph names are prefixed by a single character and a hyphen (i.e. V- to indicate when, during window processing, they should be executed). It should be noted that these special paragraph names **cannot** be performed or jumped to. If it is necessary to call such a routine, a further section name or paragraph name should be coded.

7.1.2 Parameter-Passing Subroutine Calls

Parameters may be passed to subroutines by coding an ENTRY statement as the first statement of the routine:

ENTRY *entry-name* [USING A B ...]

where *entry-name* is a symbol as specified in Section 3.2.2. Each operand A, B, etc. in the optional USING clause above must be a BASED item. These items must be defined as either level 01 or level 77 within the data division. Items at an intermediate level (i.e. levels 02-49) cannot be used as parameters within this statement.

Up to seven operands may be coded for an entry statement. These operands are passed to the routine by means of the CALL statement as follows:

CALL *entry-name* [USING A B ...]

The number of parameters coded in the call must be the same as the number of parameters expected by the corresponding ENTRY statement.

7.2 Screen Management Statements

This section describes statements used to accept and display information on the screen. These statements are:

DISPLAY	Display a field or complete window
ERROR	Display an error message
ACCEPT	Accept a field
ATTRIBUTE	Set current display attributes
CLEAR	Clear a window or the entire screen
ENTER	Execute a window

7.2.1 The DISPLAY Statement

Three forms of the DISPLAY statement are available:

DISPLAY *name* [AT *line col* | SAMELINE] [FMT "*options*"]

DISPLAY *name* [LINE *line* COL *col* | SAMELINE] [FMT "*options*"]

DISPLAY WINDOW *window-id* [TEXT]

where *name* is a text literal or the name of the variable to be displayed, *line* and *col* are the line and column position on the screen at which the variable is to be displayed, FMT "*options*" defines the output formatting for the field and window-id is as defined in the WINDOW statement.

The first two forms of the statement are used to display individual variables on the screen. The last form is used to display complete windows. The following discussion applies to the first two forms of the display statement.

7.2.1.1 Displaying Fields

The first two forms of the display statement cause the contents of the variable *name* to be displayed at the coded line and column positions. If no line or column position is coded, the display operation takes place at the baseline (i.e. the last line of the screen). The display operation will normally take place at the start of the baseline (i.e. at column 1) unless the SAMELINE option is coded. If the SAMELINE option is coded the display takes place after any immediately preceding baseline display or accept.

If the display statement is a baseline display (i.e. no line or column numbers are coded) the display takes place using display attribute 7, otherwise it takes place using the current display attribute in force, as documented in Section 7.2.4.

FMT "*options*" is used to format the field as displayed. Table 7.2a lists the five groups of options.

Option	Condition	Output Formatting	As input	As output
B	*field* = 0	**Blank** when zero	00.00	
C	*field* < 0	Trailing **CR**	-12.34	12.34CR
<		Enclosed in **()**		(12.34)
-		Trailing **-**		12.34-
D	*field* > 0	Trailing **DR**	+56.78	56.78DR
>		Enclosed in **()**		(56.78)
+		Trailing **+**		56.78+
,	None	**Comma** insertion	90123	90,123
$	None	Leading **dollar**	45.67	$45.67
0		**Zero** fill	8.90	0008.90
*		**Asterisk** fill	1.23	***1.23

Table 7.2a - The Output Formatting Options

You may specify no more than one option from each group in any set of options. Examples of invalid options would include:

```
BC<+,$
->+*
BCD,0*
```

Examples of valid sets of options and the resultant output formatting are as follows:

Options	Field as Input	Field as Displayed
BCD,	-678.90	678.90CR
	0.00	
	+1234.56	1,234.56DR
B<,	+789.01	789.01
	-7890.12	(7,890.12)
B-+$,	+345.67	$345.67+
	$890.12	$890.12-

-+0	+345.67	00345.67+
	-8901.23	08901.23-
CD*	+456.78	**456.78DR
	-901.23	**901.23CR

Table 7.2b - Examples of Output Formatting

7.2.1.2 Displaying Windows

The third form of the display statement is used either to display the screen form (i.e. fixed text) and/or variable data contained within a window. When the TEXT keyword is coded, only the fixed text will be displayed, and the window will become active. When the TEXT keyword is omitted, only the variable data within the window will normally be displayed. However, if the window has not yet been activated, the text portion of the window will first be displayed, thus activating it.

DISPLAY WINDOW *window-id* TEXT statement is often used to activate a number of windows at the start of a frame so that the complete, but so far empty screen is displayed prior to processing. When this is **not** done, windows are activated as required, for example when executed using the ENTER statement.

DISPLAY WINDOW *window-id* statement is used to display the data fields within a window. The various fields defined within the window must be initialised prior to the display operation. The most frequent use of the display statement without the TEXT option is to initialise a window prior to entering it in EDT mode, explained in Chapter 6, or to refresh a record that is currently displayed.

Note that this statement provides no control over scrolled windows. If it is used to display data into a scrolled window, this data will be displayed into whichever Record Display Area happens to be current. It is therefore not possible to use this statement to display successive RDAs within a window.

While this statement can be used to display data within a window associated with a target record type, care must be taken to ensure that the I/O channel of the target record type matches with the data being displayed. The simplest way to ensure this is **not** to use the statement in such windows for any purpose other than to refresh the currently displayed record.

The display statement may be used within the routines section, and thus cause re-entrant calls on the window manager. If however any attempt is made to display a window that is currently executing, such as when attempting to display a window from its own routines section, the frame will be aborted with a stop code.

7.2.2 The ERROR Statement

The ERROR statement is used to display an error. It is coded:

ERROR *message*

where *message* is a text literal or variable error message. When executed, the ERROR statement causes the console bell to sound, and the *message* is displayed on the baseline. This message is always displayed using display attribute 8.

7.2.3 The ACCEPT Statement

The ACCEPT statement is coded as follows:

ACCEPT *name* [AT *line col* | LINE *line* COL *col* | NEWLINE]
[*keys*] [CNV] [NUL] [YN] [RJF] [FMT "*options*"]

where *name* is the variable to be accepted, *line* and *col* are the screen position at which it is to be accepted, *keys* is a list of function keys enabled for the accept operation, CNV, NUL, YN and RJF are input options and FMT "*options*" specifies the formatting to be used when the accepted field is re-displayed. Where *line* or *col* are variables, these must be 9(4) COMP. If no line or column position is coded, the accept operates at the baseline immediately after any previous baseline displays, unless the NEWLINE clause is coded, in which case the baseline is cleared before the accept operation takes place.

If CNV is coded, lower-case characters are converted to upper-case before re-display. Unless NUL is coded a null reply causes the field to be re-input. If YN is coded the field is re-input unless the reply is Y or N. If RJF is coded, the field is displayed and stored in right justified form. The accept operation first displays the contents of the variable name at the requested position. The accept process may consist of the operator modifying or accepting the displayed default, or keying an entirely new field value. The accept is normally completed by keying <RET>, after which the new field value is returned in *name*. If FMT "*options*" has been coded the field is formatted according to the set of options specified before being re-displayed, see Table 7.2a. Function keys other than <RET> can be enabled by coding a list of *keys*, see Table 7.2c. Of the function keys, <RET> is always enabled in order to allow the completion of the accept operation. All the other functions are enabled by coding the appropriate mnemonic. For example, coding:

ACCEPT *field* AT 02 20 NXT BCK

causes the *field* to be accepted at line 2, column 20, and allows the use of the <NXT> and <BCK> keys to return an exception.

7.2.3.1 Successful Completion

Successful completion of the accept operation takes place when the operator enters, accepts or modifies the displayed default value and keys <RET>.

7.2.3.2 Exception Conditions

An Exception Condition is returned if the operator keys any of the enabled function keys. When any function keys are enabled, an ON EXCEPTION statement must be coded immediately following the accept statement. When the accept statement returns an exception, the system variable $FUNC will contain a value between 1 and 19, see Table 7.2c. If more than one function key is enabled, $FUNC should be examined to determine which was used.

If a function in the range 10 (Abort) to 19 (Move) is keyed, any input so far keyed by the operator is always discarded and the contents of the variable *name* is unchanged. The accepted variable can therefore only change if the accept operation is terminated with a function in the range 0 (Return) to 9 (Skip).

Mnemonic	**Normal Usage**	**$FUNC**

RET	Accept current field, select record	0
UF1	User Function 1	1
UF2	User Function 2	2
UF3	User Function 3	3
NXT	Go to next window	4
PGE	Forward-page window	5
BPG	Back-page window	6
UP	Back one record (Uparrow)	7
DWN	Forward one record (Downarrow)	8
SKP	Skip fields to next tab-stop	9
ABO	Abort program	10
BCK	Terminate window (Back to prior)	11
CLR	Clear the window	12
DTE	Delete current record	13
HME	Cursor home	14
BFL	Back one field	15
ENQ	Enquiry mode - select index	16
INS	Insert record	17
UDL	Undelete record	18
MOV	Move record	19

Table 7.2c - Function Key Mnemonics

7.2.3.1 Programming Notes

The accept operation makes use of a number of additional function keys which are used during field editing, such as character insertion and deletion. In addition, the user may use the <HLP> key during the accept operation, and this causes two types of help to be displayed.

The first help window contains a menu of the currently enabled functions, listed in the same order as in the accept statement. It is therefore useful to enable function keys in the order that places the most important functions first in this help menu.

The second help window appears if the users keys <HLP> again. This time the help message displayed is the one coded prior to the accept statement. When designing the frame, it is therefore important to pay attention to its layout, so that sensible help messages are displayed at each accept operation.

7.2.4 The ATTRIBUTE Statement

The ATTRIBUTE statement is used to select the current display attributes for use in field displays and accepts. It is coded:

ATTRIBUTE *n*

where *n* is a numeric literal or 9(4) COMP attribute type number. The attributes used in the display of fields and text items are those set up by the operator with the Speedbase customisation utility and listed in Table 7.2d.

Type	Description

1	Scrolled data fields in the current record
2	Scrolled data fields in other records (de-emphasised)
3	Text items within the scrolled area (emphasised)
4	Text items outside the scrolled area (de-emphasised)
5	Title line at top of screen (line 1)
6	Help text and function key menu
7	Base line messages
8	Error messages
9	Lines and boxes
10	Non-scrolled data fields
16	Currently accepted field
17	Screen background colour
18	Window background colour

Table 7.2d - Attribute Types

Table 7.2d describes the types of fields that may be displayed, and their associated type numbers. Using the Speedbase customisation utility, different display attributes are allocated to each field type. For example, field type 1, which is used to display scrolled fields within the current record, might be assigned the attributes bright blue on black. Therefore, if the statement ATTRIBUTE 1 is coded, the following fields are displayed bright blue on black. When an accept takes place the field is always re-displayed on completion using the current attributes set up in the previous attribute statement.

Thus the field types shown above represent a customised set of video facilities, which are to be used whenever fields of that type are displayed. The video facilities for a given field type are referred to as an attribute set which is identified using the associated field type or **attribute number**.

When a display takes place on the baseline (i.e. when no line or column number is coded) attribute 7 is always used. If a line and column are coded, the display takes place using the current attribute. Displays using line and column positions are referred to as **formatted displays** in the rest of this section.

When a frame is first entered, the current attribute is set to attribute 10, the attribute normally used to display non-scrolled data-fields. If no other action is taken, all formatted displays in the frame take place using this attribute. It is possible, however, to change the current attribute using the ATTRIBUTE statement. For example, executing the statement:

```
ATTRIBUTE 1
```

changes the current attribute to attribute 1, and causes all further formatted displays to be performed using this attribute.

Any of the attributes listed in Table 7.2d may be selected using the ATTRIBUTE statement, with the exception of attribute 16. This attribute is used by the accept operation in order to highlight the currently accepted field. It is important to note that the current attribute only controls the attribute in which a field will be re-displayed **after** an accept operation. **During** the accept operation the field will always be shown in attribute 16.

Note that setting the default attribute has no effect on the operation of the window manager which always "knows" what type of fields it is displaying, and therefore automatically selects the appropriate attribute.

7.2.5 The CLEAR Statement

The CLEAR statement is used to remove data from a particular window, to completely remove a window, or to completely clear the screen. It is coded:

CLEAR [WINDOW *window-id* [DATA]]

where *window-id* is the ID assigned by the window statement.

If the CLEAR statement is coded without a window-id, the entire screen is cleared, thus de-activating any currently displayed windows. If the screen contains a screen header, this will be re-displayed as part of the clear operation. Otherwise, the screen is entirely cleared of all data and text.

When the CLEAR statement is coded with a window-id, the specified window is removed from the screen. If the window is a POP-UP, the screen-image under the window at the time it was activated is re-displayed. Otherwise, the area under the window is cleared back to screen background attribute 18.

When CLEAR *window-id* DATA is coded, only the data items displayed within the specified window are cleared. After this instruction is executed only the fixed text portion of the window is displayed.

7.2.5.1 Programming Note

A CLEAR statement may be executed in the routines section, causing a re-entrant call on the window manager. If an attempt is made to clear a window that is currently executing, the frame will be terminated with a stop code. This error is most commonly made when using the CLEAR statement without a window-id, since this causes all windows to be cleared.

7.2.6 The ENTER Statement

The ENTER statement causes a window to be executed. It is coded:

ENTER WINDOW *window-id* [*line*:*col*]

where *window-id* is the ID assigned by the window statement. This normally allows the operator to add, select, maintain, or delete one or more records stored on a database. The functions actually available to the operator will depend on the options coded for the window. These options are described in detail in Chapter 6.

The optional *line* and *col* clause is used to define a new position on the screen for the window, and thus allows the window to be dynamically positioned. The variables define the top left-hand corner of the window, and may be coded as constants or as 9(4) comp variables. If either line or column number is zero, then the window will revert to its normal position.

7.2.6.1 Successful Completion

Depending on the options specified in the window construct, one or more records may have been processed, and the window will have been successfully completed. Control will be

returned on successful completion only when this is permitted by the window's sequence statement. This statement is described in section 6.3.4.

7.2.6.2 Exception Conditions

Unsuccessful Completion occurs when the window is aborted by the user keying <ABO> or <BCK> or by the window's process routine returning any exception. When the <ABO> function is keyed, or any exception other than 1 is returned by the process routine, control is immediately returned to the statement following the ENTER instruction. Otherwise the window specified in the sequence statement, if any, is entered next.

These conditions can be differentiated by testing the system variable $$COND. If $$COND=1, <BCK> was keyed, if $$COND=2, <ABO> was keyed. If an exception is passed back by the window's process routine, this exception number will also be passed back in $$COND. These exception conditions must be trapped by coding an ON EXCEPTION statement immediately following the ENTER verb.

7.3 Report Printing Statements

7.3.1 The PRINT Verb

The PRINT verb is used to print a PF construct. It is coded:

PRINT *pf* [NEWPAGE]

where *pf* is a record ID defined by a PF statement in the data division. Processing of the PRINT verb proceeds as follows:

If the verb is being used for the first time within the frame, a print file is opened. This process automatically causes any headers defined by the PF construct to be printed.

If the printer's current line position is before the start-line number coded for the PF, an appropriate paper advance takes place. If printing the PF would cause its end-line number to be exceeded, or the NEWPAGE option has been coded, a page advance will take place. This process involves printing any trailer PF constructs, followed by a page throw, which is in turn followed by the printing of any header PFs.

The print lines are then assembled as specified by the PF construct. This causes all text and data-items to be moved to the print lines, which will include conversions of computational fields to display format. If any ADD options were coded, these are also processed at this time. The assembled print lines are then output to the printer, after which they are cleared with spaces.

The PRINT statement can fail if an overflow condition takes place. This can occur when executing an ADD option, or when converting computational numbers to display format. In this event the program will be terminated with Program Check 11 - Overflow. This condition cannot be trapped within the application program. It is therefore essential that output fields are made sufficiently large to accommodate any printed variables.

7.3.2 The MOUNT Verb

The MOUNT verb is used to load non-standard stationery on the printer. It is coded:

MOUNT *description* LENGTH *n* USING *pf*

where *description* is that of the form to be mounted, *n* is the length of the form in print lines, and *pf* is the record-id which is to be used to produce a test alignment pattern.

The MOUNT verb should be executed before the first PRINT operation using the desired form takes place. The MOUNT verb causes a print file to be opened, and establishes a new form length. The output device specified may be a real printer or a print spooler.

In either case, the output from the program will eventually be printed. When printed, the operator is prompted as follows:

UNIT *nnn description*

where *nnn* is the unit number assigned to the physical printer on which the output is eventually printed, and *description* is as coded in the MOUNT verb. Once the operator has indicated that the appropriate forms are mounted, the alignment PF indicated by the MOUNT statement will be printed. This is then followed by the prompt:

UNIT *nnn* IS ALIGNMENT OK?

If the operator confirms this prompt, the mount statement will complete. Otherwise, the alignment pattern will be repeated, until the operator indicates correct alignment.

The rest of the report is then produced as normal. Stationery may again be changed by executing a further MOUNT verb. When printing of the report is complete the operator is automatically requested to replace normal stationery by the prompt:

UNIT *nnn* REPLACE STANDARD STATIONERY

The MOUNT verb can fail if the print unit is a real printer and the operator indicates inability to mount the required form. This exception condition can be trapped by coding an ON EXCEPTION statement immediately following the MOUNT verb.

7.3.3 Report Printing in Dependent Frames

As discussed earlier, the printer will be closed automatically on frame termination. Note, however, that the printer is always closed by the frame that opened it (i.e. if the printer is opened by the root frame, then it will be closed when the root frame terminates). If opened by a dependent frame, then it will be closed when that dependent frame terminates.

7.4 Database Access Statements

This section describes statements used to access the database:

WRITE	Adds a record to the database
REWRITE	Modifies an existing record
DELETE	Logically remove a record from the database
FETCH/READ	Randomly retrieve record via any index
FETCH/READ NEXT	Retrieve next record sequentially via index
FETCH/READ PRIOR	Retrieve prior record sequentially via index
FETCH/READ FIRST	Retrieve first of group of records via index
FETCH/READ LAST	Retrieve last of a group of records

GET	Relative (direct) record retrieval
UNLOCK	Relinquish record lock

7.4.1 The WRITE Verb

The WRITE verb causes a single record to be added to the database:

WRITE *rt* [LOCK]

where *rt* is the name of record type defined in an ACCESS statement. The statement causes the record stored in area *rt* to be added to the database. It changes the current record position so that the next FETCH NEXT/PRIOR instruction will retrieve the record immediately following or preceding the written record.

When *rt* is a servant record to one or more master records, these records must previously have been retrieved and must be locked when the WRITE instruction is executed.

If the LOCK clause is coded the record will be exclusively locked following successful completion. If the LOCK clause is omitted, the record will be unlocked following this operation.

7.4.1.1 Successful Completion

The system part of the data record stored in area *rt* will have been zeroed, and the new record added to the database. Indexes and linkages to any master records will have been established, and any GVA fields residing on the target record's masters will have been updated. The updates of these master records take place within the corresponding data record area within the application program, and therefore causes these GVA fields to be "refreshed". The lock status of these master records remains unchanged following successful completion.

7.4.1.2 Exception Conditions

If no free data records exist at the time the call is performed, an exception code will be returned. If writing the record would lead to a duplicate primary index key, an exception code will be returned. If the required master records are not locked at the time of the instruction, the frame is terminated with a stop code. If the operation causes the database to exhaust its free index block pool, the frame is terminated with a stop code. If any of the indexes to be created begins with a HIGH-VALUES byte (i.e. #FF) the frame is terminated with a stop code.

7.4.2 The REWRITE Verb

The REWRITE verb allows a previously retrieved and exclusively locked record to be re-written to the database:

REWRITE *rt* [LOCK]

where *rt* is the name of record type defined in an ACCESS statement. In order to REWRITE a data record, it must previously have been retrieved, and must be exclusively locked at the time the instruction is executed. The statement is used to update a record, and causes the record stored in area *rt* to be re-written to the database. If the LOCK clause is coded, the record will be exclusively locked, following successful completion, or unlocked following unsuccessful completion of the re-write. The nature of the update that takes place is dependent on the fields on the data record that were modified:

If any fields comprising secondary index keys were modified, the existing index entries referencing the data record are deleted, and new index entries established. The primary index key may not be modified by this instruction.

If any fields comprising master access keys were modified, the record is unlinked from the previously existing master records, and linked to the new masters. These new master records **must** be locked or protected at the time the verb is executed.

If any GVF field has been modified, the corresponding master record GVA fields will be similarly amended. This process will also occur if any master access key has been changed. This process takes place within each master record data record area as discussed in the previous section.

The REWRITE instruction only re-writes the user-data part of the record. Modification of GVA fields residing on the re-written record will therefore have no effect.

7.4.2.1 Successful Completion

The user part of the data record stored in area rt will have been re-written to the database. Indexes and linkages to any master records will have been amended as necessary, and any GVF field changes reflected in their target GVAs. The lock status of any master records remains unaffected following successful completion, whereas the target record will be unlocked unless the LOCK clause has been specified.

7.4.2.2 Exception Conditions

If an attempt has been made to modify the primary index key of the record, the frame will be terminated with a stop code. If the record to be re-written is not locked, the frame will be terminated with a stop code. If any new master records are not locked/protected at the time of the instruction, the frame will be terminated with a stop code (some later versions of Speedbase will generate an exception under these conditions). If index modification causes the database to exhaust its free index block pool, the frame will be terminated with a stop code. If any of the indexes to be created begin with a HIGH-VALUES byte (i.e. #FF) the frame will be terminated with a stop code.

7.4.3 The DELETE Verb

The DELETE verb logically removes a previously retrieved and locked record from the database:

DELETE *rt*

where *rt* is the name of record type defined in an ACCESS statement. In order to DELETE a data record, it must previously have been retrieved, and must be exclusively locked at the time the instruction is executed. The record to be deleted may not have an active servant group (i.e. it may not act as a master to any servant records).

7.4.3.1 Successful Completion

Deletion of a record causes all index entries referencing it to be removed, following which the record is unlinked from any associated master records. This process includes removal of GVF values from their corresponding GVA fields. The statement causes the data record to be returned to a free list of data record "slots". These free slots may then subsequently be re-used by the WRITE instruction.

7.4.3.2 Exception Conditions

If the record has an active servant record group, an exception condition will be returned, and no action will have taken place. If the record is not locked when the instruction is executed, the frame will be terminated with a stop code.

7.4.4 The READ and FETCH Verbs

The READ and FETCH verbs both retrieve data records via a specified index:

READ | FETCH [*option*] *idx-name* [KEY *key-value* | *idx-name2*] [*lock-option*]

where *option* is one of FIRST, NEXT, LAST or PRIOR. The index via which retrieval is to take place is determined by *idx-name*. Since index names are always unique, this also defines the target record type to be retrieved. An optional index *key-value* may be coded following the KEY clause. This specifies the required key value, or range of key-values to be returned by the verb. Alternatively, the KEY clause may include an index name, *idx-name2*, in which case the record is retrieved via index *idx-name* according to the key value of the last record accessed in the I/O channel specified by *idx-name2*. The optional *lock-option* is one of NOLOCK or PROTECT, and may also include the optional RETRY clause.

The FETCH verb always retrieves the target record only, whereas the READ verb will retrieve the target record with all master records for which an I/O channel has been established. The master records so retrieved are always returned unlocked. In all other respects these verbs are identical. The following discussions therefore apply equally to the READ and FETCH verbs.

The FETCH instruction, without an option, is used to retrieve a record with a particular index key. The required index key may be explicitly coded using the KEY clause, or may be placed into the appropriate fields within the data record. If the requested key is found, the record is retrieved. Otherwise an exception is returned and no further processing takes place.

The FETCH FIRST instruction retrieves the first record in the sequence of the requested index. When the KEY clause is coded, the first record equal to or greater than the requested key is returned. The retrieved record key is checked and an exception condition generated if this does not match the requested key value. If the KEY clause is not coded, the first record within the specified index is retrieved.

The FETCH LAST instruction retrieves the last record in the sequence of the requested index. When the KEY clause is coded, the last record equal or less than the requested key is returned. The verb then checks the retrieved record key, generating an exception condition if this does not match the requested key value. If the KEY clause is not coded, the last record within the specified index is retrieved.

The FETCH NEXT instruction retrieves the next record in the sequence of the specified index. It is therefore used to retrieve records sequentially by ascending key value. If the KEY clause is coded the retrieved record key is checked. If this key does not match the requested key, an exception condition is returned.

The FETCH PRIOR instruction retrieves the preceding record in the sequence of the specified index. It is used to retrieve records sequentially by descending index key value. If the KEY clause is coded the retrieved record key is checked. If this key does not match the requested key, an exception condition is returned.

The FETCH, FETCH FIRST and FETCH LAST instructions are therefore used to retrieve randomly a record via a selected index. The FETCH NEXT and FETCH prior instructions may then be used retrieve further records sequentially in the order of this index.

7.4.4.1 The Key Clause

The optional KEY clause may be used to specify a key value required from the operation. If the KEY clause is omitted from the FETCH verb, the required key will be assembled from the appropriate fields within the data record. No other operations (i.e. FETCH FIRST, LAST, NEXT and PRIOR) require a key value. If coded, however, the supplied key is matched against the returned key and if these differ an exception condition is returned.

When the KEY clause is used, the length of the specified key is significant if shorter than the actual key length of the index. If a short key is used in a FETCH instruction, it will be extended to the right with binary zeros, and the retrieval will then either succeed or fail on the basis of this extended key. In all other instructions, the key length as passed is used to determine if a corresponding record key has been found.

7.4.4.2 The NOLOCK, PROTECT and RETRY Clauses

These clauses are used to specify whether a record lock is required when the record is retrieved, and if so the action to take if the required record is already locked. These clauses are described in detail in section 2.7.

7.4.4.3 The FETCH Statement

The FETCH statement is used to retrieve a record with a specific index key value. This key value may either be explicitly coded using the KEY clause, or may be placed into the appropriate field within the data record area prior to execution. If a short key value is passed, it will be extended to the right with binary zeros prior to the lookup.

7.4.4.3.1 Successful Completion

If an index key value is found on the database corresponding to the requested key, the data record is read into its record area. This changes the current record position to the record so retrieved. In the event that more than one record exists with the required key value, the first record in sequence will be returned. The sequence of records containing duplicate keys is determined by their relative position (i.e. in RRN order).

If NOLOCK was coded, the record will be unlocked. If PROTECT was coded, the record will be delete protected. Otherwise the record will be exclusively locked.

7.4.4.3.2 Exception Conditions

If the requested index key value is not found, an exception condition is returned, and no processing takes place. In this event, the data record area as established prior to the instruction remains unchanged. If the target record was locked, and the NOLOCK option was not coded, a locked record exception is returned and the data is returned unlocked.

7.4.4.4 The FETCH FIRST and FETCH LAST Verbs

The FETCH FIRST/LAST verbs perform a random lookup to retrieve the first/last record in the sequence of the specified index. If no KEY clause is coded, the first/last record within the specified index is returned. If the KEY clause is coded, the first/last record with a corresponding key value is returned. The FETCH FIRST statement returns the first record with a key value **not less** than the requested key. The FETCH LAST statement returns the last record with a key value **not greater** than the requested key.

If the KEY clause is coded, the index key of the retrieved record is matched against the requested key. This match takes place on the basis of the actual length of the key specified in the KEY clause. If the retrieved key differs from the requested key an exception is returned. **Note that the operation will have completed successfully in all other respects.**

7.4.4.4.1 Successful Completion

A data record is retrieved as requested. If NOLOCK was coded the record is unlocked. If PROTECT is coded the record is delete protected. Otherwise the record is exclusively locked. If the KEY clause is coded, a record corresponding to the requested key value has been retrieved. The operation will have changed the current record position.

7.4.4.4.2 Exception Conditions

If the target record was locked, and no NOLOCK option is coded, a locked record exception is returned and the data record is returned unlocked. If the retrieved record did not correspond to the requested key value specified in the optional KEY clause, a record key not found exception is returned. Note that the above exceptions can occur simultaneously.

The FETCH FIRST/LAST verbs can experience an end/start-of-file exception condition, when no equal-or-greater/equal-or-smaller key exists. The data record area will be filled with LOW-VALUES, #00s, in a start-of-file condition and HIGH-VALUES, #FFs, in an end-of-file condition.

7.4.4.5 The FETCH NEXT and FETCH PRIOR Verbs

These verbs allow sequential retrieval of records in the sequence of the specified index. The FETCH NEXT instruction retrieves records in ascending index key order, whereas the FETCH PRIOR instruction retrieves records in descending index key order.

If the KEY clause is coded, the index key of the retrieved record is matched against the requested key. This match takes place on the basis of the actual length of the key specified in the KEY clause. If the retrieved key differs from the requested key an exception will be returned. Note that prior to version 8.1 the data record is returned locked (if so requested) following a key mismatch. In V8.1 and later the next/ prior record is always returned unlocked, irrespective of the requested lock.

7.4.4.5.1 Successful Completion

The next/preceding record in sequence will have been retrieved as requested. If NOLOCK is coded the record is unlocked. If PROTECT is coded it is delete protected. Otherwise the record is exclusively locked. If the KEY clause is coded, a record corresponding to the requested key value is retrieved. The operation will have changed the current record position.

7.4.4.5.2 Exception Conditions

If the target record was locked, and no NOLOCK option is coded, a locked record exception is returned and the data record is returned unlocked. If the retrieved record did not correspond to the requested key value specified in the optional KEY clause, a record key not found exception is returned. Note that the above exceptions can occur simultaneously.

The FETCH NEXT/PRIOR verbs can experience an end/start-of-file exception condition when an attempt is made to read past the last/first record. The data record area will be filled with LOW-VALUES, #00s, in a start-of-file condition, and HIGH-VALUES, #FFs, in an end-of-file condition.

7.4.4.6 Using the FETCH Verb to Retrieve Servant Groups

The short key facility makes retrieval of servant record groups very simple. This is perhaps best illustrated by an example:

```
SECTION U01-PROCESS-ONE-CUST
*
FETCH FIRST INCUS KEY CUCUSN NOLOCK       * Get 1st invoice
ON EXCEPTION EXIT WITH 1                  * None at all!
DO                                        * Process each invoice

Process each invoice here ...

      FETCH NEXT INCUS KEY CUCUSN NOLOCK  * Get next invoice
      ON EXCEPTION FINISH                 * Key break exception
ENDDO                                     * All invoices read
EXIT                                      * So just exit.
```

The FETCH FIRST statement retrieves the first invoice record. A short key has been passed which contains only the customer number, whereas the index is composed of two fields (i.e. the customer number and invoice number). The statement returns an exception condition if no invoice is found for the customer. Otherwise, the first invoice record for the customer will be returned.

A DO loop is then established in which the invoice is processed, which is followed by a FETCH NEXT to retrieve the next invoice for the customer. The KEY clause causes Speedbase to test the retrieved record. If the retrieved key does not match the supplied customer number, an exception is generated. This is trapped on the next line, causing a transfer out of the DO loop. A normal EXIT then takes place.

Similar principles can also be used to retrieve records in descending order. To achieve this, a FETCH LAST instruction would have been coded, followed by an iteration of FETCH PRIORs.

7.4.5 The GET Verb

The GET verb allows the direct (i.e. non-indexed) retrieval of a record:

GET *rt* [KEY *n*] [*lock-option*]

where *rt* is the name of a record type declared in an ACCESS statement, *n* is the relative record number to be retrieved, and *lock-option* is either NOLOCK or PROTECT.

The relative record number of the record to be retrieved may be specified by the KEY clause. If the KEY clause is not used, the record last accessed is retrieved. If the KEY clause is specified *n* must be a 9(6) COMP variable containing the RRN of the required record. The RRN specifies the relative record position of the record to be retrieved. The first record stored is 0, the second is 1 and so forth.

The *lock-option* shown above is optional but must be one of NOLOCK or PROTECT. The optional RETRY clause may also be coded. These clauses are used to specify whether a record lock is required when the record is retrieved, and if so the action to take if the required record is already locked. These clauses are described in detail in section 2.8.

7.4.5.1 Successful Completion

The data record residing at the RRN is retrieved. If the NOLOCK option is coded the record is unlocked. If the PROTECT option is coded the record is delete protected. Otherwise the record is exclusively locked.

7.4.5.2 Exception Conditions

If the target record was locked, and the NOLOCK option was not coded, a locked record exception is returned and the data record is returned unlocked. If the RRN specified is higher than the last used RRN on file, an end-of-file condition is returned. In this event the data record will be filled with HIGH-VALUES (#FFs). If the RRN specified a deleted record, an exception condition is returned. The data record is retrieved under these circumstances, but is not locked irrespective of the lock-option in force.

7.4.5.3 Programming Note

Speedbase keeps track of the highest RRN ever accessed for each record type stored in the database, and this is known as the logical end-of-file. If a GET instruction specifies any record past this point an end-of-file condition is returned, and the data record area is filled with HIGH-VALUES (#FFs).

If the RRN specified is a deleted record, a data-transfer is performed and an exception condition is returned. This facility has been provided for system programming purposes only. The contents of the data record are modified by the DELETE verb, and cannot therefore be used to perform recovery or other procedures.

7.4.6 The UNLOCK Verb

The UNLOCK verb is used to release a locked or delete protected record:

UNLOCK *rt*

where *rt* is the name of a record type declared in an ACCESS statement. The UNLOCK verb may be used to unlock a record previously locked or protected by a FETCH, READ or GET statement. The verb releases the lock of its target record so that updates may be performed on it by other concurrent partitions. If the verb is executed when no current lock is in force, no action takes place. **The verb cannot therefore result in an exception condition**.

Any pre-existing lock is usually automatically relinquished when any database I/O executes successfully, such as when executing a FETCH. It is not therefore necessary to code explicit UNLOCK statements before executing any of the database access verbs.

7.5 Arithmetic Statements

The arithmetic statements are listed in Table 7.5a:

Statement		Operation
ADD A TO B	[ROUNDED]	B + A → B
ADD A TO B GIVING C	[ROUNDED]	B + A → C
SUBTRACT A FROM B	[ROUNDED]	B - A → B
SUBTRACT A FROM B GIVING C	[ROUNDED]	B - A → C
MULTIPLY A BY B	[ROUNDED]	B x A → B

MULTIPLY A BY B GIVING C	[ROUNDED]	B x A → C
DIVIDE A INTO B	[ROUNDED]	B / A → B
DIVIDE A INTO B GIVING C	[ROUNDED]	B / A → C

Table 7.5a - Arithmetic Statements

In arithmetic statements, A may be a computational variable or literal. B may be a computational variable or, if the GIVING clause is present, a computational literal. C may be a computational or display numeric variable. No other combinations are valid.

7.5.1 Arithmetic Truncation and Rounding

If the result of an arithmetic operation contains more digits following the decimal point than are contained in the receiving variable, the extra digits are truncated. However, if the ROUNDED phrase was coded and the most significant digit thus truncated was 5, 6, 7, 8 or 9 then the least significant digit of the receiving variable is incremented, otherwise it remains unchanged.

7.5.2 Overflow

Any arithmetic statement will suffer overflow if the result exceeds the capacity of a receiving computational variable, or does not satisfy the picture of a receiving display numeric variable. Overflow will also take place if the capacity of internal fields used to hold intermediate results is exceeded.

You may check for overflow by coding the ON OVERFLOW statement as the **statement immediately** following the arithmetic statement. If no such ON OVERFLOW statement is coded and overflow occurs the program will be terminated with an error. If an arithmetic statement suffers overflow it is suppressed and the receiving variable remains unchanged.

7.5.3 Examples

Suppose A and B are declared as PIC 9(2,1) COMP and PIC 9(2,2) COMP respectively. Then, for the statement:

```
ADD 1 TO B GIVING A ROUNDED
```

if B = 3.42 we have 1 + 3.42 (i.e. 4.42) yielding A = 4.4 and if B = -4.75, we have 1 + -4.75 (i.e. -3.75) yielding A = -3.8

Consider the division of B by A, where A = 1.1 and B = 0.28:

```
DIVIDE A INTO B GIVING A ROUNDED
```

We have 0.28/1.1 (=0.254545...) yielding A = 0.3.

To summarise, a useful rule to remember is that truncation is towards zero and that rounding, when the digit involved is 5 or more, is away from zero.

7.6 The MOVE Statement

The MOVE statement is coded:

```
MOVE A TO B [C D...]
```

where A is a literal, figurative constant or variable and B, C, D..., etc. are variables. A maximum of seven variables may follow the word TO. Where more than one variable follows TO the content of A is moved, in turn, to each of these variables. In the simple move, MOVE A TO B, execution depends on the data type of each operand. Table 7.6a shows that, of the 25 types of move theoretically possible, 15 are supported.

MOVE to From	**PIC X**	**PIC 9 COMP**	**PIC 9**	**PIC PTR**	**PIC D**
PIC X	Yes	No	Yes	No	Yes
PIC 9 COMP	No	Yes	Yes	No	Yes
PIC 9	Yes	Yes	Yes	No	Yes
PIC PTR	No	No	No	Yes	No
PIC D	Yes	Yes	Yes	No	Yes

Table 7.6a - Valid Data Types for MOVE A TO B

7.6.1 Character to Character Move

The contents of A are moved to B, one byte at a time from left to right. The leftmost (low location) byte of A is copied to the leftmost byte of B, then the next byte of A is copied to B, and so on. If A is shorter than B, then B is padded with rightmost blanks. If B is shorter than A the MOVE stops once B has been filled. In this case **character truncation**, as distinct from the numeric truncation described in section 7.5.1, occurs.

In addition, A may be a figurative constant, and you may code:

```
MOVE HIGH-VALUES TO B                * Each byte set to #FF
MOVE LOW-VALUES TO B                 * Each byte set to #00
MOVE SPACE TO B                      * Each byte set to ASCII blank, #20
MOVE SPACES TO B                     * Each byte set to ASCII blank, #20
```

The A and B fields involved in a character to character move may overlap. Indeed, the following example shows how overlapping fields may be used to set every byte of a long field to a particular value, in this case ASCII E. This operation is called a **ripple move** and is useful for initialising large data items:

```
01    A
  03  A1    PIC X
  03  B     PIC X(999)

      MOVE "E" TO A1
      MOVE A TO B
```

A special form of the MOVE statement allows processing of partial fields:

```
MOVE f1(s1:l1) TO f2(s2:l2)
```

where: *f1* and *f2* are non-indexed PIC X or Group Item variables.
s1 and *s2* define the start position within *f1* and *f2*.
l1 and *l2* define the number of bytes to move from *s1* and *s2*.

The above allows you to move all or part of a field as defined by the start position and length. Start and Length may be coded as numeric literals or as computational fields. Note that start and length **MUST** both be => 1, or unpredictable results will occur.

7.6.2 Character to Display Numeric Move
This is treated like a character to character move as described in Section 7.6.1. You must be careful that the number of bytes in the B field is sufficient since the move can result in the loss of digits if character truncation takes place.

7.6.3 Character to Date Move
The date in standard form, "dd/mm/yy", "dd/mm/yyyy" (with or without the "/" separators), is moved to the PIC 9(6) COMP date format. If the date is invalid (e.g. 31/02/89) the move will result in an exception.

7.6.4 Computational to Computational Move
A is transferred to B, taking account of the precision of the two operands. If B is of lower precision than A the difference in precisions will result in arithmetic truncation. If B is of insufficient capacity to contain A, overflow will occur.

7.6.5 Computational to Display Numeric Move
A is converted to standard display numeric format according to the picture of B. If A is too large, or is negative when B is unsigned, overflow will occur.

7.6.6 Computational to Date Move
As for computational to computational move.

7.6.7 Display Numeric to Character Move
This is treated like a character to character move as described in Section 7.6.1. You must be careful that the number of bytes in B is sufficient or the move can result in the loss of digits if character truncation takes place.

7.6.8 Display Numeric to Computational Move
A is converted to binary according to its picture clause and the result is stored in B. Arithmetic truncation will take place if the precision of A is greater than that of B. If A is too large, or does not contain a valid numeric string conforming to the picture clause of A, overflow will occur. It will also take place if A is valid but B is of insufficient capacity to contain the result.

7.6.9 Display Numeric to Display Numeric Move
The numeric string A is converted to standard numeric string format in B. If A is too large, or does not contain a valid numeric string conforming to the picture clause of A, or is negative when B is unsigned, overflow will occur.

7.6.10 Display Numeric to Date Move
As for display numeric to computational move.

7.6.11 Pointer to Pointer Move
The contents of the two-byte pointer at A are transferred, unchanged, to the two-byte pointer at B.

7.6.12 Date to Character Move

Converts the internal 9(6) COMP date format to the standard date 8 byte date string "dd/mm/yy". In order to produce a long X(10) string in the form "dd/mm/yyyy", the MOVEL (Move Long) verb should be used (i.e. MOVEL COMP-DATE TO LONG-DATE).

7.6.13 Date to Computational Move
As for computational to computational move.

7.6.14 Date to Display Numeric Move
As for computational to display numeric move.

7.6.15 Date to Date Move
The date A is moved to the date B.

7.6.16 Overflow
Overflow can occur, for the reasons described above, in the following types of move operation:

- computational to computational
- computational to display numeric
- display numeric to display numeric
- display numeric to computational

You may check for overflow during a simple move, or the last operation of a compound move, by coding the ON OVERFLOW statement, see Section 7.7, immediately following the MOVE statement. If no such ON OVERFLOW statement is coded and overflow occurs the program will be terminated in error. This will also occur if ON OVERFLOW follows a compound move but the operation suffering overflow was not the last. If a move operation suffers overflow it is suppressed and the receiving variable remains unchanged.

7.7 Transfer of Control Statements

7.7.1 The GO TO Statement
GO TO unconditionally transfers control to a paragraph or section. It is coded:

```
GO TO A
```

where A is the name of a paragraph or section, or the name or a pointer set to address the first executable instruction of a paragraph or section.

7.7.2 The GO TO DEPENDING ON Statement
The GO TO DEPENDING ON statement provides a switch capability. It is coded:

```
GO TO DEPENDING ON data-name
      TO label-1
      TO label-2
      ....
      TO label-n
```

where *label-1*, *label-2* .... *label-n* are paragraph or section names. The *data-name* must be the name of a computational variable whose integral part, *i*, is in the range 1 to *n* when the statement is executed. In this case control is passed to *label-i* as if the statement:

```
GO TO label-i
```

had been executed. If *i* is greater than *n* the results will be unpredictable since there is no upper bound range checking.

7.7.3 The PERFORM Statement

The PERFORM statement passes control to a paragraph or section. It is coded:

```
PERFORM A
```

where A is the name of a paragraph or section, or the name of a pointer set to address the first executable instruction of a paragraph or section. The statements beginning at the indicated section or paragraph are executed until control is returned to the statement following the PERFORM by an EXIT statement.

7.7.4 The CALL Statement

The CALL statement passes control to an entry point identified by the entry-name appearing in an ENTRY statement. The ENTRY statement may reside either in the current compilation, or in a compilation to be linkage edited with it. It is coded:

```
CALL A [USING B C ...]
```

where A is an entry name, or the name of a pointer set to address the first executable instruction at the entry point. A maximum of 7 parameters may be passed in the optional USING clause.

If the CALL statement does not possess a USING clause then neither must the ENTRY statement. Otherwise the parameters in the two USING clauses involved must correspond one for one. Each operand of a CALL statement's USING clause may be a variable, literal, paragraph name, section name, or entry name. However, the figurative constants HIGH-VALUES, LOW-VALUES, SPACE and SPACES must **never** appear.

Each operand in the target ENTRY statement must be of the same type as the corresponding operand of the CALL statement, and must be defined as a based item. It may **not** be a subordinate (level 02-49) item, nor the redefinition of such an item.

When a **variable** is passed as a parameter the corresponding ENTRY operand is a level 77 item or level 01 group describing the storage area the variable occupies.

If an **integer literal** is passed the corresponding ENTRY operand should be a level 77 item or level 01 group describing a single PIC S9(4) COMP field. This will overlay the integer literal, and must therefore be read-only.

If a **character literal** is passed the corresponding ENTRY operand is a level 77 item or level 01 group describing the character string. This will overlay the character literal and must therefore be read-only.

When a **paragraph name, section name** or **entry name** is passed, the corresponding ENTRY operand should be a level 77 item or level 01 group describing a single PIC PTR field. This pointer will address the paragraph, section or entry point in question, and must remain read-only. The pointer form of the GO TO, PERFORM or CALL statement can then be used to invoke the passed paragraph, section or entry point from the called routine.

7.7.5 The EXIT Statement

The EXIT statement causes control to be returned to the statement following the last **outstanding** PERFORM or CALL statement. When a PERFORM or CALL is executed the address of the following statement is stored on an internal stack. The previous contents of the stack are "pushed down" so that a number of outstanding PERFORMs or CALLs can be nested. When an EXIT is executed the top item in the stack is used to determine the statement to which control is to be passed. This stack item is then made available for re-use and the stack contents are "popped up".

The EXIT statement is therefore the **dynamic** end of a sequence of code entered by a PERFORM or CALL. An explicit EXIT statement, rather than an implied exit at the end of a paragraph, section or group of sections, makes code easier to follow and facilitates structured programming. Note that an EXIT statement issued from the highest level of a program, causes control to be passed to the next frame as defined by the Frame Header SEQUENCE statement.

7.7.6 The STOP RUN Statement

The STOP RUN statement causes immediate termination of the frame. Although Speedbase ensures that any open databases are properly closed, other termination tasks, such as those specified by the Unload Division, will **not** take place. It should be noted that a print file opened using the MOUNT or PRINT statements will **not** be closed properly when a STOP RUN statement is executed and, if spooled to disk, will be lost. This statement should therefore be used with care, and would normally only be used following irrecoverable error conditions.

7.7.7 The Finish Statement

The FINISH statement can be coded anywhere within a DO loop, as described in Section 7.8.2. It has the effect of transferring control to the statement following the next ENDDO, thereby exiting from the loop in a clear and structured manner.

7.7.8 Prefixed Transfer of Control Statements

The transfer of control statements:

GO TO *label*
PERFORM *label*
EXIT
FINISH
STOP RUN

may be prefixed by IF *condition*, ON OVERFLOW, ON EXCEPTION, ON NO OVERFLOW and ON NO EXCEPTION. See Section 7.8.1. The result is to make execution of the transfer of control statement dependent on the condition defined by the prefixing clause. For example:

```
IF A ZERO GO TO LAB2
MOVE A TO B
ON OVERFLOW PERFORM AA223
DELETE GA
```

```
ON NO EXCEPTION EXIT
```

7.7.9 The EXEC Statement

The EXEC statement is used to load and execute a dependent frame. It is coded:

```
EXEC  A
```

where A is the name of a dependent frame, which must have been coded to be DEPENDENT ON the frame performing the EXEC statement. The statement causes the requested frame to be loaded into memory, where upon it is immediately executed. Control is returned to the statement following the EXEC when the dependent frame, or chain of dependent frames, terminates execution.

The EXEC Statement never returns an exception. In the event that the dependent frame cannot be loaded, for example because it could not be found, the frame is terminated with an EXIT code.

7.7.9.1 Programming Note

The EXEC statement may **only** be used to call dependent frames. It Is not possible to EXEC a Cobol program. Control may however be passed to such a program using the sequence statement in the header of a non-dependent frame (i.e. one in which the DEPENDENT ON clause is not used).

7.8 Conditional and Iterative Statements

7.8.1 Format of Conditional Structures

There are two basic formats for conditionals. Format 1 is:

IF *condition* | ON [NO] OVERFLOW | ON [NO] EXCEPTION
*
[OR *statements* | AND *statements*]
*
..... * *Statements to be executed if condition true (*group A*)*
*
[ELSE
*
..... * *Statements to be executed if condition false* (group B)]
*
END

Format 2 is:

IF *condition* | ON [NO] OVERFLOW | ON [NO] EXCEPTION
*
GO TO *label* | PERFORM *label* | EXIT | FINISH | STOP RUN

In format 1, if the statements in group A are not terminated by a GO TO, GO TO DEPENDING ON, EXIT or STOP RUN, when the ELSE statement is encountered control is passed to the statement following END. If the ELSE statement is missing there are no group B statements. In this case if the group A statements are not terminated by an unconditional transfer of control, when the END statement is met it is ignored and execution continues with the next statement.

Similarly, if the group B statements are not terminated by a GO TO, GO TO DEPENDING ON, EXIT or STOP RUN then, when their last statement has executed, control drops through the END statement and continues with the next statement.

The statements ELSE and END must be coded on new lines and cannot be combined with other statements.

Format 1 conditionals may be nested up to 32 times and contain iterative structures, see Section 7.8.2. An END statement terminates the most recent conditional statement which has not yet been matched by an END statement. An ELSE statement refers to the most recent conditional statement which has not yet been matched by an END statement.

Format 2 conditional statements may appear within format 1 conditionals. Format 2 statements always generate less code than the equivalent logic coded using Format 1.

7.8.2 Format of Iterative Structures

There are three formats for iterative structures (i.e. DO loops). Format 1:

```
DO
*
.....          * Statements to be executed
*
ENDDO
```

Format 2:

```
DO WHILE condition
*
[OR statements | AND statements]
*
.....          * Statements to be executed while the condition remains true
*
ENDDO
```

Format 3:

```
DO UNTIL condition
*
[OR statements | AND statements]
*
.....          * Statements to be executed until the condition becomes true
*
ENDDO
```

In Format 1 the enclosed statements are executed over and over until some transfer of control (such as GO TO or FINISH) causes an exit from the loop.

In Format 2 the enclosed statements are executed zero or more times, while the *condition* remains true. The *condition* is tested before the first iteration, and then before each subsequent iteration. As soon as it is not satisfied, the statement immediately following the ENDDO receives

control. It is possible therefore that the statements between DO and ENDDO may not be executed at all.

Format 3 is similar to Format 2 except that the enclosed statements are executed only as long as the *condition* remains false.

DO loops may be nested up to 16 times and may contain conditional structures. An ENDDO statement terminates the most recent DO statement which has not yet been matched by an ENDDO statement.

Condition Clause	**Equivalent**	**Restrictions**
A EQUAL B	A = B	A, B must either both be PIC X or PIC PTR or, if B is PIC 9 COMP then A may be PIC 9 or PIC 9 COMP. One but not both may be literal.
A NOT EQUAL B	A NOT = B	
A LESS B	A < B	
A NOT LESS B	A NOT < B	
A GREATER B	A > B	
A NOT GREATER B	A NOT > B	
A SPACES	A SPACE	A must be PIC X
A NOT SPACES	A NOT SPACE	
A HIGH-VALUES		
A NOT HIGH-VALUES		
A LOW-VALUES		
A NOT LOW-VALUES		
A ZERO		A must be PIC 9 COMP or PIC 9
A NOT ZERO		
A POSITIVE		
A NOT POSITIVE		
A NEGATIVE		
A NOT NEGATIVE		
A NUMERIC		A must be PIC 9
A NOT NUMERIC		

Table 7.8a - The Condition Clause

7.8.3 The Condition Clause

The condition clause that appears in IF and DO statements may assume any of the formats summarised in the left-hand column of Table 7.8a. The mathematical symbols = > < may be coded in the place of the words EQUAL, GREATER and LESS, respectively, and the figurative constants SPACE and SPACES are synonymous. The conditions are divided into four groups, depending on the restrictions which apply to the operand or operands.

Comparison of display numeric and computational items obeys the normal rules of arithmetic. The comparison of character variables and pointers takes place, byte by byte, from the left-most byte at the low address to the right-most byte at the high address. The bytes being compared are treated, for the purposes of the comparison, as 8-bit unsigned numbers. If two strings of unequal length are compared, the shorter will be considered to be extended to the right with ASCII blanks. A figurative constant is treated as a character string of exactly the same length as the variable with which it is being compared.

The condition:

A NUMERIC

is true only if the variable A contains a valid numeric string which is compatible with the picture clause of A.

The following three examples show the use of condition clauses in a Format 1 conditional, Format 2 conditional and iterative structure:

```
IF COUNT > 100                          * Format 1 conditional
      MOVE 0 TO COUNT
END
*
IF NAME SPACES GO TO ASKRTN          * Format 2 conditional
*
DO WHILE COUNT POSITIVE              * Iterative structure
      ADD -1 TO COUNT
      PERFORM CALC
ENDDO
```

7.8.4 Compound Conditions

Compound conditions may be established by coding groups of one or more OR statements or AND statements immediately following one of these eight **initial conditional** statements:

IF *condition*
ACCEPT... NULL
ON OVERFLOW
ON EXCEPTION
DO WHILE *condition*
ON NO OVERFLOW
ON NO EXCEPTION
DO UNTIL *condition*

The format of a compound condition is either:

initial conditional statement
OR *condition-1*
.....
OR *condition-n*

or:

initial conditional statement
AND *condition-1*
.....
AND *condition-n*

The first compound condition is true if **any** of the condition clauses it contains is satisfied, but the second form is only true if **all** of the constituent conditions hold. Once sufficient conditions have been evaluated to establish the result of the compound condition, no further conditions are evaluated. It is not allowable to mix AND and OR statements in the group following the initial conditional statement, and if you attempt to do so the compiler will flag any out-of-place statement in error.

Here are two examples using compound conditions:

```
IF COUNT > 100
OR COUNT NEGATIVE                    * Count zeroised if not
      MOVE 0 TO COUNT                * between 0 and 100
END
DO WHILE COUNT POSITIVE
AND NAME NOT SPACES
AND ERRFLAG ZERO
      ADD -1 TO COUNT
      PERFORM CALC
ENDDO
```

7.8.5 The ON OVERFLOW Statement

The ON OVERFLOW statement is used for checking for the overflow condition which may result following an arithmetic statement, or a MOVE, DISPLAY or EDIT statement. ON OVERFLOW must be coded as the statement immediately following the one generating the condition to be tested. If this is not done and an overflow condition arises the frame is terminated in error.

Note that the second of the two examples which follow shows the coding required if you simply wish to ignore an overflow condition:

```
ADD COUNT TO ACCUM                        * Set ACCUM negative
ON OVERFLOW                         * If any count not
OR COUNT NEGATIVE                   * positive and in range
      MOVE -1 TO ACCUM
END
ADD COUNT TO ACCUM                        * Ignore any
ON OVERFLOW                         * overflow condition
END
```

7.8.5 The ON EXCEPTION Statement

The ON EXCEPTION statement is used for checking for the exception condition that can be returned as the result of CALL'ing or PERFORM'ing certain types of routine, database processing or console I/O operations. ON EXCEPTION must be coded as the statement immediately following the one generating the condition to be tested. If this is not done and an exception condition arises the frame is terminated in error.

A CALL or PERFORM statement may not necessarily be liable to an exception condition. Whether this is the case or not depends on the routine invoked by the statement. If it inevitably returns control by means of an EXIT statement it will always return normal completion, and the invoking CALL or PERFORM statement will never suffer an exception. However, it is possible, and often very useful, to write routines which generate exception conditions to indicate when special circumstances have arisen.

When an exception occurs the system variable $$COND, the condition number, is set to a positive value. This is normally 1, except when the same statement can generate an exception for a variety of different reasons. In this case $$COND assumes values 1, 2... and so on, each of which distinguishes a different condition. System variable $$RES, the result code, may also be established when an exception occurs, to give further information about the cause of the exception.

If you need to process the condition number or the result code, handle $$COND and $$RES at the very beginning of the logic introduced by your ON EXCEPTION statement. Normally you should immediately save their values with MOVE statements, or branch on $$COND with a GO

TO DEPENDING ON, or code a sequence of IF statements. This is because the majority of Global Cobol statements cause the values in $$COND and $$RES to be destroyed by the time they return.

Note, in particular, that the statement:

```
DISPLAY $$COND
```

which you might be tempted to write for debugging purposes, should **not** in fact be used. If you code it by mistake as part of your exception handling logic it will not have the desired effect since $$COND will be reset before it comes to be displayed and all that will appear at the console will be zero. The correct technique is indicated by the first example below. The second shows the coding required if you simply wish to ignore an exception condition:

```
CALL EXRTN
ON EXCEPTION                              * If exception and
AND TESTFL POSITIVE                       * if test flag on
      MOVE $$COND TO Z-WORK             * computational work field
      DISPLAY "EXCEPTION CODE "             * displayed
      DISPLAY Z-WORK SAMELINE           * not $$COND
ELSE
      DISPLAY "NORMAL COMPLETION"
END
*
DELETE RT                                 * Delete record
ON EXCEPTION                              * ignore exceptions
END
```

7.8.6 The ON NO OVEFLOW and ON NO EXCEPTION Statements

These statements check for the reverse condition to those checked by the ON OVERFLOW and ON EXCEPTION statements. For example:

```
ON NO OVERFLOW
      statements to be executed when no overflow has occurred
END
```

They should be used in preference to the construct:

```
ON OVERFLOW
ELSE
      statements to be executed when no overflow has occurred
END
```

7.9 Table Handling

A table consists of a number, *n*, of fixed length entries occupying contiguous storage. Each entry is identified by its index, a number between 1 and *n*. The first entry has index 1, the second index 2 and so on. Tables are defined by repeating groups or elementary items with OCCURS clauses.

The table handling operations cause a rapid examination of the table to take place, a selected field from each entry being compared with a key, whose length and value you specify. When comparison takes place the key and the current entry are treated like character variables and compared byte by byte from left to right.

After the table handling operation completes you will either be returned the index of the entry which satisfied your request or, if the request was not satisfied, a table operation exception.

7.9.1 The SEARCH Statement

The SEARCH statement is used to identify the first entry of a table **equal** in value to a supplied key. It is coded:

```
SEARCH tc table key [entry-length]
```

where *tc* is the name of a **table control area** of the following format:

```
01    TC
  03  TCKEYL      PIC 9(2) COMP          * Supplied key length
  03  TCTERM      PIC X                  * Supplied terminator
  03  TCINDEX     PIC 9(4) COMP          * Returned index
```

You must set up the key length and terminator yourself, but GSM will return the index field.

The *table* parameter is a variable identifying the location at which the search is to begin, and *key* is a literal or variable containing the supplied key.

The fourth parameter, the *entry-length*, must be supplied if you are searching a repeating group, when the key length and entry length will be different. It must be a 9(4) COMP integer. If the parameter is omitted the entry length is assumed to be equal to the key length, as will normally be the case if you search a table of elementary items.

The key field is assumed to be located at the start of each table entry, and the search operation proceeds as follows:

- The first table entry is selected.
- If the first byte contains the terminator value, TCTERM, EXIT with 1 is returned and processing terminates.
- If the key field (i.e. the TCKEYL bytes at the start of the entry) is equal to the key supplied as the third parameter of the SEARCH statement, processing terminates normally.
- Otherwise the next entry is selected, using the fourth parameter or TCKEYL if it is omitted, and processing continues as at the second step above.

On termination the index of the entry last processed is stored in the TCINDEX field. This will either identify the first entry satisfying your request or, if an exception took place, it will identify a dummy entry starting with the terminator value. You must ensure that such an entry is placed immediately following the table if it is possible to search for a key which is not present, otherwise GSM will continue examining the memory following the table, with unpredictable results.

If there is a possibility that the key value is not present in the table the SEARCH statement should be immediately followed by an ON EXCEPTION statement to process this condition. If the ON EXCEPTION statement is not coded and the key is not present the frame is terminated in error.

7.9.2 The SCAN Statement

The SCAN statement is used to identify the first entry of a table whose value is **equal to or greater than** a supplied key. It is coded:

SCAN *tc table key* [*entry-length*]

where *tc*, *table*, *key* and the optional *entry-length* parameter are as defined in Section 7.9.1.

SCAN functions identically to SEARCH, except that the criterion for terminating the search normally is that the key field, the TCKEYL bytes at the start of each entry, should be equal to or greater than the key supplied by the third parameter. If the fields involved are character data items, the ASCII collating sequence determines the result of the comparison. If the fields are both non-negative computational items of the same precision the result is determined by the numeric value as you would expect. However, table scans involving negative or display numeric keys, or computational keys of different precision, should be avoided since the outcome is difficult to predict.

If it is possible that no key field in the table will satisfy the scan, you must delimit the table with a dummy entry starting with a byte containing the terminator value, and you should follow the SCAN statements involved with ON EXCEPTION statements to trap possible table operation exceptions.

7.10 The SUSPEND Statement

SUSPEND causes the program to be suspended for a period of time. It is coded:

SUSPEND [*seconds*]

The optional *seconds* parameter is the name of a PIC 9(4) COMP variable or integer literal containing the number of seconds for which the program is to be suspended. If the parameter is omitted, or the value supplied is less than 1, the program is simply suspended for a brief period, as if it had reached the end of its time-slice, allowing other processes to execute.

If characters are keyed at the console when a program is suspended, the suspend will be cancelled so that the program can process the input if necessary.

7.10.1 Programming Notes

The SUSPEND statement should be used when you know your program cannot proceed until some other user completes an activity. Since the only way of co-operating jobs communicating is by means of shared files, a typical use of SUSPEND by, say, a special purpose spooling program, might be as follows:

- Read the shared communications file to see if a report requires printing.
- If no report is available, execute a SUSPEND for 60 seconds, then repeat the first step.
- Otherwise, print the report, then update the communications file to indicate it has been printed.
- Repeat this process.

7.11 Global Cobol Support

The Global Cobol File Management Manual contains full information regarding the GSM Relative Sequential and Indexed Sequential Access methods, both of which are supported by the Speedbase compiler.

File sorting is also supported using the SORT, RELEASE and RETURN verbs which are documented in the Global Cobol Language Manual.

The EDIT verb used for display editing of numeric fields is also supported within the language syntax. This verb, which is provided for compatibility reasons, is documented in the Global Cobol Language Manual.

Use of the above facilities requires the presence of a licenced copy of the system library C.$MCOB during compilation, which must be specified as part of the compilation parameters. If these statements are used without the presence of this system library, an "UNDEFINED CALL" error messages will result during compilation.

8. Speedbase System Routines

Certain system routines distributed in the library C.$BALIB, listed in Table 8a, may be called directly, and are documented in this chapter. Note that the routines **must never** be called by frames operating in a Speedbase environment prior to V3.0.

Routine	Description	Parameters
B$CHK	$BASYS Presence Check	
B$LOD	Load Speedbase System Area	
B$OPN	Open Database	*name unit lock-flag*
B$FEX	Execute Frame	*frame*
B$STA	Return Database Status	*$rt area1 area2*
B$ST2	Return Extended Status	*$rt area3*
B$PRC	Close Print File	
B$CDB	Close Database	*name*
B$DSC	Clear Baseline	
B$XCL	Get Exclusive Access	*dbid*
B$XSH	Release Exclusive Access	*dbid*
B$WRJ	Right-justify Field	*window-id fld*
B$RBL	Database Re-index Routine	*dbid ...*

Table 8a - Speedbase System Routines

8.1 B$CHK - $BASYS Presence Check

This routine may be called to test whether the Speedbase system area has been loaded onto the user stack. The following statement:

```
CALL B$CHK
```

will execute successfully if $BASYS is present. An exception condition is returned if the system area has not yet been loaded. There is no point in calling this routine from an executing frame, since the system area must by definition already be present. It may be used by a Global Cobol program in preparation for executing a frame.

8.2 B$LOD - Load Speedbase System Area

This routine causes the Speedbase system area $BASYS to be loaded onto the user stack. The routine first checks to see if the systems area is present by calling B$CHK and, if it is, only the console screen is cleared. Otherwise $BASYS is placed on the user stack and is loaded with the appropriate customisation details from the T>*nnn* file, where *nnn* is the current terminal number.

There is no point in calling this routine from an executing frame, since the system area must by definition already be present. It may be used by a Global Cobol program in preparation to executing a frame.

The routine is invoked by the parameter-less call:

```
CALL B$LOD
```

An exception will be returned if the routine was unable to load $BASYS. This may occur because of I/O errors, insufficient room on the user stack, or because $BASYS could not be found.

8.3 B$OPN - Open Database

This routine is used to open a database. The call is of the form:

```
CALL B$OPN USING name unit lock-flag
```

where *name* is the PIC X(5) name of the database to be opened, without the preceding "DB", *unit* is the ID of the unit on which the main index file resides, and *lock-flag* is a 9 COMP variable with the value 0 to allow shared access or 1 to provide exclusive access to the database.

The routine causes the designated database to be opened, placing a database access block of at least 448 bytes on the user stack. The routine may be called several times to open a number of databases. There is no restriction on the number of databases that may be opened using this routine, subject only to memory restrictions.

Any databases opened may then be accessed by both the calling frame and/or subsequent frames. The only requirement is that the database must be open before the first I/O operation accessing it takes place. Databases remain open until specifically closed or a STOP RUN condition occurs, when **all** opened databases are automatically closed.

This routine may be called from a Global Cobol program preparatory to running a frame. If this is done it is **essential** that the Speedbase system area $BASYS is first loaded, and this **must** be checked using the B$CHK or B$LOD routines.

8.3.1 Exception Conditions

The open call may fail for a number of reasons, and the systems variable $$COND may be tested to discriminate between these as follows:

Exit code	Description
21	Database is already open
22	Database not found or invalid file type
23	Database data-file not found or invalid type
24	Database is in use
25	I/O error
26	Insufficient memory available on user stack

Table 8.3a - B$OPN Exception Codes

Condition 23, data-file not found, is a common error often caused by invalid unit assignment. The main index file contains the unit-id of each data-files which, if a logical unit-id, must be correctly assigned prior to the call.

8.4 B$FEX - Execute Frame

This routine is used to load and execute a frame from within the Global Cobol environment. The call:

CALL B$FEX USING *frame*

causes the *frame*, which must not be a dependent frame, to be loaded into memory and immediately executed. This routine acts similarly to the Global Cobol CHAIN verb, in that the incoming frame will overlay the memory region occupied by the calling frame, and control is therefore **never returned**.

Prior to executing the call, it is essential the Speedbase system area $BASYS is present on the user stack. This must be ensured by calling the B$CHK or B$LOD routines. B$FEX does not return exceptions, so in the event that the requested frame is not found, processing is terminated with an EXIT or STOP code.

8.5 B$STA - Return Database Status

This routine returns database status information for a specified record type. The call is of the form:

CALL B$STA USING $*rt area1* [*area2*]

where $*rt* identifies the record for which status information is required and *area1*, *area2* are areas into which this information is returned. $*rt* is an internal name for the I/O channel, where *rt* is the name of the record-type as specified in the ACCESS statement. **It is essential that the record ID is prefixed by the $ symbol**. *Area1* returns information that is specific to the requested record type. Its format is as follows:

```
01    A1                          * Layout of Rec Status Block
  03  A1NAME     PIC X(6)         * Record Name
  03  A1RLEN     PIC 9(4) COMP    * Record Length
  03  A1RRN      PIC 9(6) COMP    * Record no (RRN) of last I/O
  03  A1LOCK     PIC S9 COMP      * I/O Channel Lock Status
  03  A1SIZE     PIC 9(6) COMP    * Size of extend in records
  03  A1FREE     PIC 9(6) COMP    * Number of free records
```

The field A1RRN contains the relative record number of the last I/O operation. The field A1LOCK contains the current lock status of the I/O channel. This is either 0 (no lock), 1 (Protect Lock) or 2 (Exclusive Lock). Field A1SIZE shows the total number of records of the requested type that may be stored in the database and A1FREE how many of these are free for use by subsequent WRITE operations.

If the optional second parameter *area2* is passed, more general information relating to the database as a whole is also returned. The format of *area2* is as follows:

```
01    A2                        * Layout of DB Status Block
  03  A2DBID    PIC X(5)        * Database ID
  03  A2DGEN    PIC 9(4) COMP   * Database Generation No
  03  A2NRCS    PIC 9(2) COMP   * Num Rec-types stored on DB
  03  A2BACY    PIC X           * Backup Cycle ID
  03  A2BASR    PIC 9(2) COMP   * Incremental Backup Serial No
  03  A2SIZE    PIC 9(6) COMP   * Size of Index Block Pool
  03  A2FREE    PIC 9(6) COMP   * No of free IDBs remaining
```

Field A2DBID and A2DGEN return the database-id and generation number. A2NRCS returns the number of different record types stored within the database. Fields A2BACY and A2BASR return the current backup cycle-id and incremental backup serial number respectively. The use of these fields is described in the Speedbase Presentation Manager User Manual.

The fields A2SIZE and A2FREE relate to the common index pool which contains index data for all record types stored within the database. The field A2SIZE contains the size of the index pool in Index Blocks (IDBs), and A2FREE numbers the unused IDBs.

Note that the database for which information is being requested must be open at the time of the call. If this is not the case the routine returns an EXIT 25550.

8.6 B$ST2 - Return Extended Database Status

This routine returns extended database status information for a specified record type. The call is of the form:

```
CALL B$ST2 USING $rt area3
```

where $*rt* identifies the record and hence the database for which extended status information is required and *area3* is an area into which this information is returned. $*rt* is an internal name for the I/O channel, where *rt* is the name of the record-type as specified in the ACCESS statement. **It is essential that the record ID is prefixed by the $ symbol**. The format of *area3* is as follows:

```
01    A3                        * Layout of DB Status Block
  03  A3DBID    PIC X(5)        * Database ID
  03  A3DGEN    PIC 9(4) COMP   * Database Generation No
  03  A3NRCS    PIC 9(2) COMP   * Num Rec-types stored on DB
  03  A3BACY    PIC X           * Backup Cycle ID
  03  A3BASR    PIC 9(2) COMP   * Incremental Backup Serial No
  03  A3SIZE    PIC 9(6) COMP   * Size of Index Block Pool
  03  A3FREE    PIC 9(6) COMP   * No of free IDBs remaining
```

```
03   A3DBNM     PIC X(5)          * Name of Database
03   A3DBUN     PIC X(3)          * Unit id of Main Index File
03   FILLER          PIC X(9)          * Reserved
```

Fields A3DBNM and A3DNUN allow you to determine the name and unit of the currently open database which therefore allows you to close and subsequently re-open it using the B$OPN call. These fields are additional to those included in Area2 of the B$STA routine.

8.7 B$PRC - Close Print File

This routine causes the print-file opened by a prior PRINT or MOUNT statement to be closed. It is coded:

```
CALL B$PRC
```

The printer is normally closed automatically when the frame terminates. This call allows the printer to be closed earlier where this is advantageous.

8.8 B$CDB - Close Database

This routine causes a database opened in the current session to be closed. It is coded:

```
CALL B$CDB USING db-name
```

The database *db-name*, which must be open at the time of the call, is closed. All locks outstanding prior to the call are released. If the USING clause is omitted, all open databases are closed.

8.9 B$DSC - Clear Baseline

This routine enables you to clear the baseline. The parameter-less call:

```
CALL B$DSC
```

causes any characters displayed on the baseline using the ACCEPT or DISPLAY verbs to be removed from the screen.

8.10 BXCL, BXSH - Get, Release Exclusive Access

These routines allow you to gain or release exclusive access to a database, which must be open at the time of the call. Code:

```
CALL B$XCL USING dbid
```

to gain exclusive access to the database *dbid*. The call causes the database to be re-opened in exclusive mode. In the event that exclusive access cannot be granted because the database is in use in another partition, exception condition EXIT 25524 is reported and should be trapped by use of an ON EXCEPTION statement.

To relinquish exclusive control of database *dbid*, code:

```
CALL B$XSH USING dbid
```

which causes the database to be re-opened in shared mode. Note that the database *dbid* must be open when either of these routines is called. If this is not the case STOP 25582 results.

8.11 B$WRJ - Right-justify Field

This routine causes a field accepted within a window to be right-justified. This is useful when processing fields which may be numeric or character, depending on customisation. Coding:

CALL B$WRJ USING *window-id fld*

causes the character field *fld* to be accepted as a right-justified field during processing of the window *window-id*. Make the call before the window is entered (e.g. in the Load Division). Right-justification of the field stays in force until the frame terminates. Several field in the window made be dynamically specified as right-justified by multiple calls of the routine.

8.12 B$RBL Database Re-index Facility

This routine allows the indexes of a single or all record types to be rebuilt. The rebuilding process is functionally identical to a partial database rebuild as performed by the database rebuild utility $BARBL. This system routine may be run from time to time to re-organise indexes following heavy processing such as mass deletions or insertions in order to optimise performance. The routine has the advantage that only a selected record type need be re-built, thus potentially saving unnecessary re-building time of other record types.

The call is of the form:

CALL B$RBL USING *dbid rcid opr col row*

where:

dbid is the PIC X(5) Database ID of the database to be rebuilt, which must be open with exclusive access prior to the call.

rcid is the PIC X(2) ID of the record type to be rebuilt. If this parameter contains spaces then the indexes of all record types will be rebuilt.

opr is the PIC X operator response flag, which if set to "Y" will request a pause before continuing. Otherwise this flag should be set to "N".

col is an optional PIC 9(2) COMP variable containing the column number at which the status window is to be displayed. The column number must be in the range 1 to 34 inclusive. If this parameter is not provided, the default is column 17. Note that this parameter, if coded, **MUST** be coded as a variable. A numeric literal must not be coded.

row is an optional PIC 9(2) COMP variable containing the row number at which the status window is to be displayed. The row number must be in the range 1 to 18 inclusive. If this parameter is not provided, the default is row 18. Note that this parameter, if coded, **MUST** be coded as a variable. A numeric literal must not be coded.

Following the call, the status window will be displayed and the indexes of the designated record type(s) are rebuilt. On successful completion control is returned without exception.

8.12.1 Programming Notes

The rebuilding process is memory intensive, and requires quite substantial amounts of memory to perform quickly. We recommend that the routine is called from frames that contain only a minimal amount of additional code. For this reason, we recommend that database opening and closing and so forth is performed in a prior frame or overlay.

8.12.2 Exception conditions

The following exception conditions may be returned:

Exit code	Description
1	Specified database was not open
2	Specified database was open non-exclusively
3	Database Dictionary was not found or in use
4	Database Dictionary file was of invalid file type
5	Database Dictionary file was corrupt
6	Database and Dictionary generations are different
7	Database is corrupt and cannot be rebuilt
8	Record ID supplied was invalid
9	Record ID supplied has no indexes
11	Insufficient memory to start re-indexation
12	Key Errors during re-index; database is corrupt
13	Unix C-ISAM channel cannot be opened

Table 8.12a - B$OPN Exception Codes

Any serious error detected during the re-building process will cause the routine to terminate with a stop code. Note that this will leave the database in a partially rebuilt state, and a full re-build should be performed before further processing using the database rebuild utility $BARBL.

9. Speedbase System Variables

This chapter describes the system variables supported by the Speedbase compiler and listed in Table 9a. Unless specifically stated to the contrary, system variables **must not be modified**, and must therefore be treated as read-only items by an application frame.

Variable	Description	Picture clause
$PRUN	Printer unit-id	X(3)
$PGNO	Current page number	9(4) COMP
$LINO	Current line number	9(4) COMP
$RSPG	Restart page number	9(4) COMP
$PHLT	Printer halt suppress flag	9 COMP
$FUNC	Accepted function number	9(2) COMP
$MODE	Current window operating mode	9 COMP
$FWFR	Forward frame-id to load	X(6)
$BKFR	Backward frame-id to load	X(6)
$FNTX0	Keytop name, <RET> key	X(7)
$FNTX()	Keytop name, functions 1-19	X(7)
$FNBY0	Function-key value, <RET> function	X
$FNBY()	Function-key value, functions 1-19	X

Table 9a - Speedbase System Variables

9.1 $PRUN - Printer Unit-id PIC X(3)

This variable specifies the printer unit which is to be used to output reports generated by the PF construct. This unit-id is initially set up to the unit "$PR". It may be changed to any valid random access or printer device unit number or logical-id (e.g. 220, 500, FLS, PR1). If changed, this unit-id remains in force until the end of the current session.

9.2 $PGNO - Current Page Number PIC 9(4) COMP

The page number of the page currently being printed on is stored in this variable. The page number is automatically set to one when the printer is opened in response to the first PRINT statement in the frame. It is automatically incremented whenever a page throw takes place. $PGNO may be referenced in a PF construct to print the current page number on each page.

9.3 $LINO - Current Line Number PIC 9(4) COMP

This variable indicates the number of lines that have so far been printed within the current page. It is set to zero when the printer is initially opened, and reset as page throws occur. It is automatically incremented as lines are output to the printer.

9.4 $RSPG - Restart Page Number PIC 9(4) COMP

This variable indicates the first page number from which re-printing should commence. The variable normally contains zero to indicate that printing should not be suppressed. The variable

is reset to zero whenever the printer is closed. A restart option may be provided within a print frame by setting it to the page number from which re-printing should commence.

9.5 $PHLT - Printer Halt Suppress Flag PIC 9 COMP

This variable is used to control the print interrupt feature. When printing normally commences, the following baseline message is displayed:

Type <Ctrl G> to halt printing

During a print run, the operator may then interrupt printing by keying <Ctrl G>. Printing may then be restarted, directed to another device or suppressed. If suppressed, the frame will normally continue processing, but without producing a report. Moving -1 to $PHLT also allows the operator to suppress printing, but in this event the frame will be terminated by issuing a STOP RUN instruction. Moving the value 1 to $PHLT suppresses the interrupt feature. Note that any changes made to $PHLT remain in force until the end of the session.

9.6 $FUNC - Accepted Function Number PIC 9(2) COMP

Following an accept operation, this variable returns the Speedbase function number of the key used. The possible function numbers are shown in Table 9.6a below:

Mnemonic	Description	$FUNC
RET	Accept current field, select record	0
UF1	User Function 1	1
UF2	User Function 2	2
UF3	User Function 3	3
NXT	Go to next window	4
PGE	Forward-page window	5
BPG	Back-page window	6
UP	Back one record (Uparrow)	7
DWN	Forward one record (Downarrow)	8
SKP	Skip fields to next tab-stop	9
ABO	Abort program	10
BCK	Terminate window (Back to prior)	11
CLR	Clear the window	12
DTE	Delete current record	13
HME	Cursor home	14
BFL	Back one field	15
ENQ	Enquiry mode - select index	16
INS	Insert record	17
UDL	Undelete record	18
MOV	Move record	19

Table 9.6a - Returned $FUNC Values

9.7 $MODE - Current Window Operating Mode PIC 9 COMP

The variable $MODE allows the current window processing mode to be discerned. A window may operate in up to seven modes, listed in Table 9.7a below:

Mode	Description	Function	$MODE
ENQ	Enquiry	Initiate an enquiry	1
DSP	Display	Display existing record	2
MNT	Maintenance	Modify existing record	3
DEL	Deletion	Delete existing record	4
EDT	Edit	Create new record from existing	5
ADD	Addition	Add new record	6
INS	Insertion	Insert new record	7

Table 9.7a - Window Processing Modes

A full description of window processing modes may be found in Section 6.3.2.

9.8 $FWFR and $BKFR - Frame-id to Load PIC X(6)

These system variables specify the frame-id to be loaded following successful and unsuccessful completion of the current frame respectively.

These variables are normally initialised by use of the frame header's SEQUENCE statement, but may be modified by simply moving a new frame-id into the appropriate variable at run-time.

If the variable contains spaces, **no** frame is loaded on completion. Please note that this differs from the Sequence statement, where the keyword EXIT must be coded to achieve this.

9.9 $FNTX0 and $FNTX() - Keytop Names PIC X(7)

These variables contain the keytop names allocated to each of the Speedbase functions by the $BACUS customisation program. $FNTX0 corresponds to the keytop name used for <RET>, Speedbase function zero, and $FNTX is an array containing the keytop names for Speedbase functions 1-19, <UF1> through <MOV>. The entry number within the array corresponds to the Speedbase Function number $FUNC as described in Table 9.6a above.

The variables contain the keytop names as customised using $BACUS for the current terminal, and can be useful within operator prompts.

9.10 $FNBY0 and $FNBY() - Function-key Values PIC X

These variables contain the function-key values generated by the current terminal for each of the Speedbase functions. $FNBY0 contains the code generated for the <RET> function, on most terminals this is #0D. $FNBY is an array containing the key values generated for Speedbase functions 1-19, <UF1> through <MOV>. The entry number within the array corresponds to the Speedbase function number $FUNC as listed in Table 9.6a.

These variables may be used in combination with the TYP$ system routine. The TYP$ routine allows you to place characters into the console's type-ahead buffer, which are then input during subsequent accept operations. The format of the call is as follows:

CALL TYP$ USING *area length*

where *area* identifies the characters to be placed in the type-ahead buffer, and *length* is a PIC 9(4) COMP variable or numeric literal specifying the length of *area*.

It should be noted that TYP$ always places characters at the **front** of the type-ahead buffer, from which characters are **first** retrieved by accept operations. If multiple calls on TYP$ are required, you must therefore make these in **reverse order**.

The function key byte values may be used during window processing as well as normal accept operations. For example, the following code segment would display a page of territory records starting from TR10 in the demonstration system frame TERR:

```
CALL TYP$ USING $FNBY(5) 1          * 1 <PGE> Display next page
CALL TYP$ USING "TR10" 4            * 2 Key value "TR10"
CALL TYP$ USING $FNBY(16) 1         * 3 <ENQ> Enter ENQ Mode
ENTER WINDOW W1
```

Remembering that the type-ahead buffer must be filled in the reverse order to the operations required, line 3 causes the <ENQ> function to be placed in the type-ahead buffer, and forces the window into enquiry mode. Line 2 supplies the key value "TR10", and line 1 requests a page operation from this key value. The window is then entered in the normal way.

9.10.1 Programming notes

The type-ahead buffer is limited to the size (normally) defined in the configuration file, and you must not therefore attempt to save more characters than this. If you do, TYP$ will return exception condition 3.

The type-ahead buffer may already contain characters keyed in by the operator. If this does not leave enough room for your characters to be stored, TYP$ will return exception 2. In this event, you could execute the BELL verb, which clears the type-ahead buffer as well as sounding the console bell. You must then perform the TYP$ calls again.

Appendix A - The Speedbase Compiler ($SDL)

The Speedbase Development Language Compiler generates executable frames in a single-pass compilation process. It processes a source file which may contain one or more individual frames, producing executable frame files as its main output. Figure Aa provides a diagrammatic view of the main input and output files processed by the compiler. Of these, only the frame source and generated object frames necessarily exist, all other inputs and outputs being optional.

The compiler may reference the records defined in up to four database dictionaries, details of which are compiled into executable frames. The database dictionaries are created and maintained using the Speedbase dictionary maintenance utility, see Appendix F. Each frame may access any of the records defined in the dictionaries using the ACCESS statement, which causes record layouts and other details to be copied into the application frame. This is a similar concept to a Cobol copy library.

The Speedbase compiler contains an integral linkage editor which is invoked during compilation if the need arises. Most frames will normally make use of a service module, which contains most of the frequently used system routines. This module is loaded automatically whenever a frame is executed, and therefore avoids linking system routines into each application frame.

When system routines not resident in the service module are used, the compiler automatically links them into the frame. These routines may be system routines distributed in the Speedbase routine library C.$BALIB, or any other compilation files or libraries created using the Global Cobol compiler.

When a dependent frame is compiled, the controlling frame is used as an input during the compilation process. The various field and I/O channel definitions are transferred to the dependent frame, which allows it to access these items as if declared locally. The compiler may also make use of up to four copybook libraries, which are accessed using the COPY statement.

The compiler can produce any number of object frames in a single invocation, limited only by the size and number of free directory entries of the specified object device. A practical limitation in the order of 50 to 99 frames per compilation therefore usually exists.

The created object frames are executed under the control of the Speedbase Presentation Manager. The Presentation Manager contains routines which control the transfer of information between the frame and screen or printer, such as those used to control video attributes. A frame cannot be executed directly, but must run using a loader program, see Section 8.4, or a menu. The menu opens the databases required, and manages service module loading prior to running a frame.

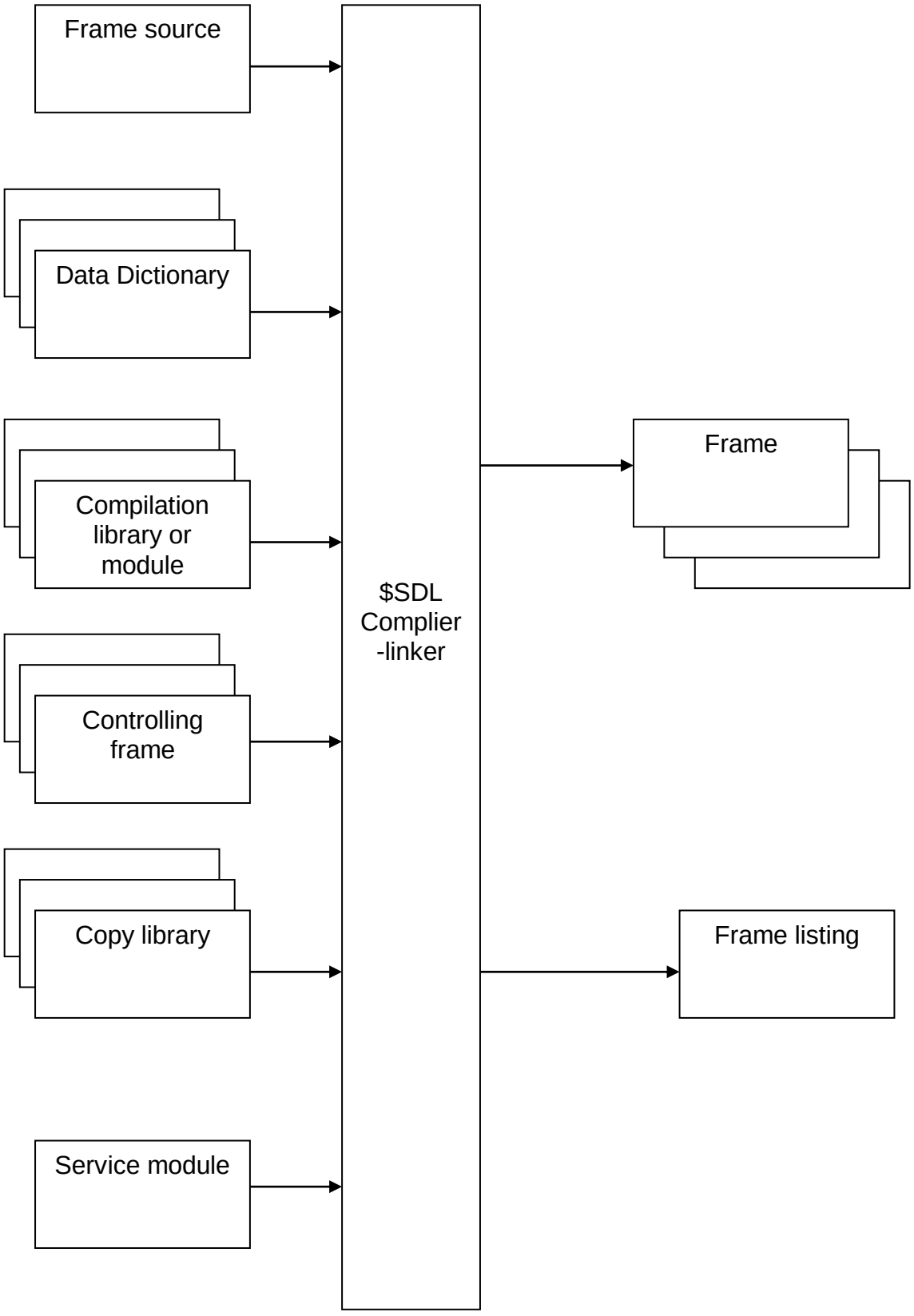


Figure Aa The SDL Compiler I/O Structure

A.1 Compiler Dialogue

The Speedbase compiler is invoked by using a menu entry set up for the purpose or by keying $SDL at a menu or GSM READY prompt. The following is an example of dialogue which may be used to compile the sample application listed in Appendix C:

```
GSM READY:$SDL
$A3 SOURCE:V3DEMO UNIT:S
$A3 OBJECT UNIT:$P  SIZE:60000
$A3 DICTIONARY 1:DEMON UNIT:FLS GENERATION 6
$A3 DICTIONARY 2:<CR>
$A3 LISTING UNIT:$PR
$A3 COMPILATION OPTION:<CR>
```

The above dialogue specifies that the source file S.V3DEMO residing on logical device S is to be compiled. Object programs are directed to the logical device $P, the program residence unit, and are restricted to a maximum size of 60,000 bytes. The dictionary to be used in this compilation is then specified. This dictionary, located on logical device FLS, is then found by the compiler, which displays its generation number in confirmation.

No further dictionaries are required for this compilation, and this is indicated by entering <CR> in response to the second dictionary prompt. A compilation listing is required, and this has been directed to the logical device $PR. No further compilation options are specified.

Compilation of the frames in the source file S.V3DEMO then commences. As compilation of each new frame begins, the following message is displayed on the screen:

```
COMPILING frame-id .......
```

where *frame-id* is as specified in each FRAME statement. If a file with this frame-id already exists on the specified object unit, this message is appended by the following text:

```
OLD VERSION DELETED
```

Some care must therefore be taken that existing files are not inadvertently deleted. In most installations it is wise to reserve a specific unit to act as object unit for Speedbase compilations.

The compiler therefore produces a number of individual object frame files on the designated unit. It is good practice to combine these files into a program library immediately following compilation. This is achieved using the $LIB Librarian utility which is described in detail in the Global Cobol User Manual. The remainder of this section describes each of the Speedbase compiler prompts in detail.

A.1.1 The Source Prompt

The SOURCE prompt is used to specify the file and unit-id of the source program file to be compiled. The file specified must be a text file and must begin with an S. prefix. This prefix is automatically assumed and should not therefore be entered. The unit-id specified may be any logical or physical random access device.

A.1.2 The Object Unit and Size Prompts

The OBJECT UNIT prompt allows the specification of the object unit to which the generated executable programs are to be directed. This unit may be any logical or physical random access device. This is followed by the SIZE prompt. This prompt allows the specification of the

maximum size of each object file. This maximum must be large enough to contain the largest single object program in the compilation. If <CR> is entered in response to this prompt, a default of 60,000 bytes will be used and the object file truncated on completion of its compilation.

A.1.3 The Dictionary and Unit Prompts

The DICTIONARY prompt requests the name of up eight dictionaries to be referenced during the compilation. The dictionary name is limited to five alphanumeric characters, the prefix DI is assumed and should not therefore be entered.

This is followed by the UNIT prompt to allow the specification of the unit on which the dictionary resides or is to be created. This unit-id may be any logical or physical random access device. The generation number of the specified dictionary is then displayed.

It is possible that a dictionary may be invalid, owing to serious errors during dictionary maintenance. If this is the case the compiler will display the following message:

```
$A3 INVALID DICTIONARY
```

In this event, the dictionary will have to be corrected before any further attempt is made to compile frames using it.

A.1.4 The Listing Unit Prompt

The LISTING UNIT prompt allows the specification of the unit-id to which the program listing is to be directed. The unit-id may be a logical or physical device, being a physical printer or random access device used for spooling. If the unit-id entered is a printer (i.e. has a unit number in the range 500 to 599) this device is simply opened.

If the unit-id entered is a random access device, the user is prompted for the file size, in bytes, to be allocated. The size entered should be large enough to contain the entire listing for all programs within the source file. If no size is entered, the largest possible spool file will be created. This file will be truncated on completion of the compilation run.

The listing file created is of text-file format and can therefore be examined using the $INSPECT utility. This listing file has a file name "L.*file*" where file is the name of the source file as entered in response to the SOURCE prompt. If <CR> is entered in response to the LISTING UNIT prompt, the default printer unit $PR will be selected. The production of the listing may be suppressed altogether by entering <CTRL A> for this prompt.

A.1.5 The Compilation Option Prompt

The following compilation options may be specified:

ST	Prints a symbol table for each program on the listing
BL	Prints a binary listing showing generated object code
COP	Copies selected member into target library
NCX	Causes copybook content not to be listed
NSL	Suppresses print of source program lines
LNK	Used to specify compilation files for linkage editing
LM	Specifies production of a linkage edit map
HIGH	Specifies highest memory address that may be used
BASE	Specifies an explicit base address
SD	Enables Symbolic Debugging of the program using $DEBUG.

HT Specifies off-line storage of help text.

These options are further explained below:

A.1.5.1 The ST option
The ST option causes a symbol table to be printed on the program listing. The symbol table is printed at the end of each compiled program. The table shows the symbol names declared within the compilation with the appropriate addresses. This information is often useful during program debugging.

A.1.5.2 The BL option
The BL option causes full details of generated object code to be printed on the compilation listing. This object code is shown in hexadecimal form next to the corresponding source program instructions. The principal use of this option is to allow patches to be produced for application programs. If you have no intention of doing this, the option is best avoided since it produces a somewhat cluttered listing.

Object code is always printed on the right hand side of the listing. In some circumstances the generated code may not be printed exactly next to the source code line to which it belongs. For example, in compound conditional statements where some code cannot be generated until the entire construct has been processed.

The addresses for the targets of forward jumps and calls within the program are also not known at the time these instructions are processed, and the listing file is printed. The addresses printed in this case refer to the code address of the previous reference to the so far undeclared entry. Similar considerations also apply to the use of HIGH-VALUES and LOW-VALUES figurative constants, the location and length of which are not known until the entire program is compiled.

A.1.5.3 The COP option
The COP option allows up to three copy libraries to be specified using the following dialogue:

```
$A3 COMPILATION OPTION:COP
$A3 COPY LIBRARY:name UNIT:unit-id
$A3 COPY LIBRARY:<CR>
```

where *name* is the name of the copy library and *unit-id* is the unit where the copy library resides. The *unit-id* defaults to the unit-id of the program source file if <CR> is keyed. This dialogue allows up to six copy libraries to be specified. As each is entered, there is a pause as the copy library is opened and indexed.

A.1.5.4 The NCX option
The NCX option causes copybook content not to be listed within the frame listing.

A.1.5.5 The NSL option
The NSL option suppresses printing of source program lines to the compilation listing. Only source lines on which errors occurred are printed.

A.1.5.6 The LNK option
The LNK option is used to specify up to four compilation files or libraries to be used during by the link phase of the Speedbase compiler. Note that the specified compilation files and/or libraries are only linked into each frame if actually referenced using the CALL statement. During

compilation, the Speedbase compiler keeps track of calls made on system library or other external routines, and these are automatically linked into the program during the last phase of compilation of each frame.

This facility will normally be used to link system subroutines, such as those required by the EDIT and SORT verbs. It is, however, also possible for the developer to produce his own subroutines or subroutine libraries using Global Cobol. The LNK option dialogue is shown below:

```
$A3 COMPILATION OPTION:LNK
$A3 COMPILATION MODULE:module-name UNIT:unit-id
$A3 COMPILATION MODULE:<CR>
```

where *module-name* is the name of a compilation module or library, entered without the C. prefix, and *unit-id* is the unit on which the module or library resides. If <CR> is entered in response to the UNIT prompt, the unit-id will be defaulted to the object-unit previously entered.

A.1.5.7 The LM option

The LM option causes a link map to be printed at the end of each frame listing. This link map shows the names of all modules linked into the object frame, with the addresses at which each module was loaded.

A.1.5.8 The HIGH option

The HIGH option is used to control the use of memory. When you develop application software products it is important to ensure that they will fit in the available user area on the user's computer. You might, for example, decide to limit your frames to the use of a 50 Kbyte user area. To do so, use the HIGH option with an address of C800. The HIGH option dialogue is as follows:

```
$A3 COMPILATION OPTION:HIGH  HIGH ADDRESS LIMIT #:high
```

where *high* is the first unused address, in hexadecimal (e.g. C800).

If you compile a frame which uses a higher address than you have specified with the HIGH option, the ENDFRAME statement of the offending frame is marked with compiler error 291. Note that in the compilation listing the ENDFRAME statement always includes the address of the first free memory location following the frame. In this case the address will be greater than you have specified in the HIGH option.

A.1.5.9 The BASE option

The BASE option allows an explicit base address to be specified. This is then used as a program base for all frames in the compilation source. If it conflicts with either a specified controlling frame or services module it is reset to the default and an error message is displayed.

A.1.5.10 The SD option

The SD option Causes symbolic debug tables to be compiled into the program thus allowing you to use the symbolic debugging facilities of $DEBUG when testing your program.

A.1.5.11 The HT option

The HT option Causes help text to be stored offline. Offline help text is stored within the program file but in such a manner that no program address space is used, thus making more space available for application code. Note that programs compiled with option HT must be run

with Speedbase V8.1 or later. If run on earlier versions, the message " No help Available " will be displayed in response to the <HLP> function.

Appendix B - Compiler Error and Warning Messages

This appendix describes the error and warning messages produced by the Speedbase compiler. All the messages listed may be printed as warnings or errors, depending on the context in which the error occurred. The Speedbase compiler will attempt to produce an executable code file **irrespective** of the seriousness of the errors detected during compilation, to avoid unnecessary re-compilations during the development process.

Compilations containing warnings only are generally correctly executable, but should be corrected as a matter of good practice. Programs compiled with errors **must** be corrected and re-compiled before any serious attempt is made at system testing.

Most error messages indicate the exact location at which the error was detected by means of an up-arrow character. This character points to the current source code segment being compiled at the time of error detection and should be used as a guide only.

1 STATEMENT NOT RECOGNISED

A program statement is invalid (e.g. spelt incorrectly) or may not be used within current program division (e.g. PERFORM in Data Division).

2 OPENING BRACKET "(" EXPECTED

An opening bracket is missing from a statement.

3 CLOSING BRACKET ")" EXPECTED

A matching closing bracket is missing.

4 UNSIGNED INTEGER EXPECTED

An unsigned integer value is either missing or invalid.

5 "ENDSOURCE" STATEMENT EXPECTED

The ENDSOURCE statement has not been coded as the last statement in the program source file.

6 END OF LINE NOT SEEN

The code line is not terminated by an end-of-line or a comment. The rest of the line is ignored by the compiler.

7 LISTING FILE SPACE EXHAUSTED

The listing file used to produce the compilation listing is full, or a non-recoverable error has occurred on the printer. Further printing is suppressed, but the compilation will otherwise be correctly concluded.

8 INVALID CHARACTER

The source program file contains an invalid character, such as lower case letters outside of quotes, or an invalid end-of-line terminator.

9 TRAILING QUOTE MISSING

A trailing quote (") is missing from an embedded string literal. The rest of the source line is taken to be part of the literal string.

10 SYMBOL TABLE PAGING ERROR

An irrecoverable I/O error has taken place during a symbol table swapping operation and the compilation is aborted. Correct hardware problem before re-compiling.

11 INVALID SYTAB OPERATION REQUESTED

An invalid symbol table operation has been requested. Compiler internal error.

12 SYMBOL TABLE SPACE EXHAUSTED

The symbol table is full. This condition will arise if any single program requires more than approximately 3750 symbols (e.g. data names, sections or labels, etc.) to compile. The only recourse is to reduce the size of your program.

13 OBJECT OR WORK FILE IS FULL

This error may arise if the object file size specified as a compilation parameter was too small. In this instance you should increase the size of the file.

14 LEVEL NUMBER EXPECTED

A Global Cobol data item level number in the range of 01 to 49 or 77 was expected but not found.

15 VALID ONLY FOR LEVELS 01 AND 77

REDEFINES or BASED may only be coded for data items at level 01 or 77.

16 DATA-NAME EXPECTED

A data-name is expected within a Window construct. This name must be at least 3 characters long.

17 INVALID SYMBOL

An invalid symbol has been coded or omitted altogether. Symbols must begin with a $ or a character in the range "A" to "Z".

18 LEVEL NUMBER IS INVALID

A Global Cobol data item level number has been detected but is not in the range 01 - 49 or 77.

19 DUPLICATE NAME DECLARATION

A symbol name has been declared twice within the program, or a variable has been defined with the same name as a systems variable.

20 UNDECLARED DATA-NAME

The data-name specified as the target of a redefinition or base is undeclared.

21 "PIC" EXPECTED

The clause "PIC" is missing from, or spelt incorrectly in, a Global Cobol data item definition.

22 9, S9, X, D, OR PTR EXPECTED

An invalid picture clause has been coded or altogether omitted. Picture clauses must start with 9, S9, X, D or "PTR".

23 "BASED" INVALID WITH A REDEFINITION

A data item declaration cannot contain both the REDEFINES and BASED clauses.

24 BASE MUST BE A PIC PTR ITEM

The symbol specified as a base pointer in a data item declaration has not been given the picture clause "PTR".

26 "PIC" INVALID FOR LEVEL 01

A level 01 data item declaration may not contain a picture clause.

27 INVALID PICTURE CLAUSE

The picture clause is present, but has been incorrectly coded. For example, PIC S99 instead of PIC S9(2).

28 ARRAY MUST BE SINGLE DIMENSION

An attempt has been made to declare a two dimensional array, by coding an OCCURS clause for an item which already belongs to an occurring group. This is not permitted.

29 ZERO LENGTH GROUP

A group data item does not contain any elementary data items, or only contains items which themselves are of 0 length.

30 REDEFINITION IS TOO LONG

A redefinition is longer than the group it redefines. Valid code will be generated, but this is likely to be a programming error.

31 QUALIFIER ILLEGAL FOR PTR OR D

PIC, PTR or PIC D item is followed by an opening bracket "(".

32 COMMA (,) EXPECTED

A comma is missing from a statement.

33 "FRAME" OR "PROGRAM" EXPECTED

The first statement for all programs must be the FRAME or PROGRAM statement.

34 FRAME ID EXPECTED

A Frame ID has not been found after the FRAME or PROGRAM statement. Alternatively the ID is an invalid symbol name.

36 "ENDFRAME" EXPECTED

An ENDFRAME or ENDPROG statement has not been coded as the last instruction in the current program.

39 INVALID LINE OR COL NUMBER

The line or column number coded in a WINDOW statement would result in the display of characters beyond permitted limits.

41 UNDEFINED RECORD ID

When this error is produced when processing the ACCESS statement, the coded record ID has not been found in the dictionary. When produced while processing the WINDOW statement, the record ID has not been earlier defined by an ACCESS statement.

45 DICTIONARY DRA/RCB LOAD FAILURE

A dictionary ACCESS'ed during compilation is corrupt. Try recreating the dictionary and then re-compile. If the error does not disappear, an internal compiler error is indicated.

48 COMPILER INTERNAL ERROR

An internal error has arisen within the compiler causing it to abort.

51 INVALID RECORD ID

The record ID is either shorter than 2 characters or does not start with an alphabetic character.

64 OPTION NOT RECOGNISED

A Window construct option has been spelt incorrectly.

67 INCONSISTENT OPTION USAGE

An option is inconsistent with the items picture clause, or a number of options have been combined illegally. For example, a protected field coded with option NUL.

68 "DIVISION" EXPECTED

The keyword DIVISION has been omitted or spelt incorrectly.

69 RECORD TYPE EXPECTED

A record type code has been omitted or is invalid.

72 DUPLICATE ACCESS FOR RECORD TYPE

The same record type has been coded twice in an ACCESS statement.

83 "ENDFORMAT" EXPECTED

The ENDFORMAT statement has been omitted.

85 INCONSISTENT PICTURE CLAUSE

A picture clause has been coded for a data item in the Window construct. This picture clause is not the same as that of the referenced field. Either change it so it is correct, or omit it.

100 STATEMENT OUT OF CONTEXT

A statement has been misplaced. For example, Data Division statement within the Procedure Division.

103 COMPILER INTERNAL ERROR

An internal compiler malfunction has been detected and the compilation will normally be aborted. Keep copies of the compilation source code and dictionaries (if any) and contact your support team.

106 COMPILATION ABORTED

The compilation has been aborted due to a serious error. This message will be preceded by another explaining the cause of the problem.

107 FEATURE NOT SUPPORTED

The statement or option coded is not supported in this version of the compiler.

108 NEGATIVE WRNXT/OBJ LEN REQUEST

Internal compiler error. Refer to error 103.

111 ILLEGAL OPCODE/OPERAND/QUAL

An illegal operation code, operand or qualifier has been detected. Internal compiler error, refer to error 103. Compiler versions 2.0 and earlier only.

112 OPERAND EXPECTED

A symbol or literal string was expected in the source program but not found.

113 INVALID NUMERIC STRING

A numeric literal has been coded incorrectly or omitted altogether.

114 HEX STRING EXPECTED

A hexadecimal string literal was expected, but not found.

115 HEX STRING MUST BE EVEN

A hexadecimal string has been found but has an odd number of characters. The last character will be ignored to make the string even.

116 INVALID HEX STRING

A hexadecimal string is invalid in that it contains characters outside the range 0 - 9, A - F.

117 OPERAND IS NOT INDEXED

An attempt has been made to code an index for a data item which is not part of an occurring group, or is not an occurring item.

118 INDEX EXPECTED, 1 ASSUMED

An index has been omitted from a data item which is part of an occurring group or is an occurring item. An index of 1 will be assumed.

119 COMP VARIABLE OR INTEGER EXPECTED

A numeric variable or integer was expected but not found.

120 OPERAND CAN NOT BE INDEXED

An attempt has been made to index an item with a variable which is itself indexed. This is illegal.

121 UNDEFINED SYMBOL

A coded symbol is undefined.

122 INVALID NUMERIC STRING

A numeric string literal has been coded incorrectly.

123 "TO" EXPECTED

Self-explanatory.

124 INVALID OPERAND TYPE

A symbol of an incorrect type required for the specified operation has been coded. For example, moving an entry name to a data-item, or a numeric field to a pointer item.

125 "GIVING" EXPECTED

Self-explanatory.

126 "ROUNDED" EXPECTED

Self-explanatory.

127 COMP OR DISP NUMERIC VARIABLE REQUIRED

A non-numeric variable or literal has been coded as the target of an arithmetic statement.

128 INVALID ARITHMETIC CONSTRUCT

An arithmetic statement has not been coded with a correct link Word (i.e. "FROM", "TO", "INTO" or "BY"). Alternatively, an incorrect link word has been used for the particular statement. For example, ADD A FROM B.

129 32 "IF" LEVELS EXCEEDED

IF statements may not be nested to a greater depth than 32. This limit has been exceeded.

130 NO MATCHING "IF"

An END statement has been found which does not correspond to a preceding IF statement. Too many END statements have been coded.

131 "EXCEPTION" OR "OVERFLOW" REQUIRED

The ON statement must be followed by "OVERFLOW" or "EXCEPTION".

132 16 "DO" LEVELS EXCEEDED

"DO" statements may be nested to a maximum of 16 levels. This limit has been exceeded.

133 NO MATCHING "DO"/"ENDDO"

An ENDDO or FINISH statement has been found which does not correspond to a matching "DO" statement. The FINISH statement may only be executed from within a DO loop.

134 DO "UNTIL" OR "WHILE" EXPECTED

If any program code follows the DO statement, then this must either be "UNTIL" or "WHILE". Probably misspelling.

135 "AND"/"OR" MAY NOT BE MIXED

A compound IF or DO condition must be composed only of OR or AND relations. The compiler has detected the use of both relations.

136 INVALID RELATIONAL OPERATOR

A relational operator is one of:

> "=", "<", ">", "LESS", "EQUAL", "GREATER", "SPACE", "SPACES", "HIGH-VALUES", "LOW-VALUES", "ZERO", "POSITIVE", "NEGATIVE" or "NUMERIC".

137 COND OPERANDS BOTH LITERALS

Both operands in an expression are literals, which of course means that the statement is not really conditional. The statement will therefore always or never be "true".

138 INVALID CONDITIONAL ACTION

Valid actions that may be coded following an IF statement are:

> GO TO, PERFORM, FINISH, EXIT or STOP.

The DEPENDING clause may only be used in an unconditional GO TO statement.

139 "ON" EXPECTED

The word ON was expected but not found.

140 "DEPENDING" VALID WITH GOTO ONLY

The DEPENDING clause may only be used within the GOTO statement. It has been incorrectly used with the PERFORM statement.

141 "WITH" EXPECTED

Self-explanatory.

142 "WITH" OR "RUN" EXPECTED

Self-explanatory.

143 INCONSISTENT CALL/GOTO/PERFORM

A symbol has been the target of a GOTO, PERFORM and CALL instruction. Since the GOTO and PERFORM statements require a label or section name, whereas the CALL statement requires an ENTRY name, this is inconsistent. One of the preceding calls must be incorrect.

145 "USING" EXPECTED

Self-explanatory.

146 SYSTEM GLOBAL USED IN COMPILE

A symbol has been defined in the program which contains a "$" character. This clashes with a symbol that the compiler requires for internal purposes. The compilation will therefore fail.

147 "INTO" EXPECTED

Self-explanatory.

148 EDIT TARGET > 30 BYTES

The target of an EDIT statement exceeds 30 bytes. This is not permitted.

149 MAX EDIT SIZE IS 9(12,6)

The source operand in an EDIT statement has a picture clause larger than PIC 9(12,6) and cannot therefore be processed.

150 9(4) COMP OPERAND EXPECTED

A statement has been coded which requires that the indicated operand has a PIC 9(4) COMP picture clause. For example, EXIT with name.

151 IF LEVEL NOT ZERO

A section or entry statement was found while an IF statement was still outstanding. One or more END statements have presumably been omitted.

152 DO LEVEL NOT ZERO

A section or entry statement was found while a DO statement was still outstanding. One or more ENDDO statements have presumably been omitted.

153 LVL 01/77 BASED ITEM REQUIRED

Operands as targets of the ENTRY USING clause must be BASED and must be level 01 or level 77 data items.

154 "PROCEDURE DIVISION" ASSUMED

The PROCEDURE DIVISION Header Statement has been omitted.

155 DUPLICATE DIVISION DECLARATION

The same Division Header Statement (e.g. PROCEDURE DIVISION) has been coded more than once within the current program.

156 UNDECLARED ENTRY/SECTION/LABEL

The target of a GOTO, PERFORM, or CALL statement is not defined in the program. The offending symbol name is displayed in the error message.

157 "LINE"/"COL" EXPECTED

A DISPLAY or ACCEPT statement has omitted or incorrectly spelt the key word LINE or COL.

158 DISPLAY ITEM EXCEEDS 127 BYTES

The target of a DISPLAY or ACCEPT statement is longer than 127 bytes. This is not permitted.

159 "LINE" EXPECTED IN DISPLAY

The key word "LINE" is missing from DISPLAY or ACCEPT statement.

160 "COL" EXPECTED IN DISPLAY

The key word "COL" is missing from DISPLAY or ACCEPT statement.

161 UNDEFINED RECORD TYPE CODE

The record ID for a WRITE, REWRITE, DELETE or GET statement is not defined in an ACCESS or RF statement.

162 RECORD TYPE CODE EXPECTED

The target coded for WRITE, REWRITE, DELETE or GET statement is not a valid record ID code, but some other symbol.

163 ALL MASTER RECORDS REQUIRED FOR UPDATE

A WRITE, REWRITE or DELETE statement has been coded. All master records associated with this record must previously have been referenced in an ACCESS statement.

164 "KEY" EXPECTED

Characters were found after the record ID in a GET statement, but this was not the "KEY" clause. If an RRN is coded, it must be preceded by this key word.

165 KEY MUST BE 9(6) COMP

The key supplied as part of a GET statement must be 9(6) COMP.

166 "NOLOCK", "PROTECT" OR "RETRY" REQUIRED

Characters were found at the end of a READ, FETCH or GET statement, but these were not the NOLOCK or RETRY options.

167 RETRY RANGE IS -1 TO 127

The retry option requires an argument in the range of -1 to 127. This value **must** be a numeric literal.

168 DATABASE INDEX OR FD NAME EXPECTED

The READ and FETCH statement must be followed by an FD name or an index name associated with an record declared by an ACCESS statement. All other I/O statements require a record ID or FD name to be coded.

169 PF NAME EXPECTED

The PRINT statement must be followed by a PF construct record ID. The ID was either not coded or not present in the current program.

170 POINT "AT" EXPECTED

Self-explanatory.

171 BASE "ON" OR "AT" EXPECTED

The BASE verb requires the key-word ON or AT. This was not coded for the instruction.

172 UNDECLARED DATA-ITEM

A routine has been defined in the Routines Section for a variable which has not been declared within the window.

173 DUPLICATE ROUTINE DECLARATION

Two or more routines of the same type have been coded for the same field. This is not permitted.

176 "LENGTH" EXPECTED

The LENGTH key word has been omitted or spelt incorrectly within the MOUNT statement.

177 MAX 16 PRINT LINES PER PF

The maximum number of print lines that may be created by a PF construct is 16. This limitation has been exceeded.

178 PF LINES DEFINED OUT OF SEQUENCE

Print lines must be **declared** in order within a PF construct. See Chapter 5 for further details.

179 ITEM EXCEEDS COLUMN 132

A data or text item has been coded with a column number that would cause it to be printed past column 132. This is not permitted.

180 PF RECORD ID EXPECTED

The record ID specified in a PF statement HEADER or TRAILER clause is not a previously defined PF record ID. These options can only refer to PF constructs coded previously within the same Data Division.

181 START "AT" OR "FROM" EXPECTED

The PF construct START verb requires the key word AT or FROM. This has been omitted or spelt incorrectly.

182 END LINE # PRECEDES START LINE

The end line number in a PF construct START statements precedes the start line number. This is not permitted.

183 NO PRINT LINES IN PF CONSTRUCT

A PF construct has been defined which contains no print lines. This will produce unpredictable results.

184 INSTRUCTION BUFFER OVERFLOW

The instruction being compiled is so complex that the compiler's instruction buffer has overflowed. No code will be generated for the instruction. This error should not occur and indicates a compiler internal error. Contact your support team.

185 "FORMAT" EXPECTED

The key word FORMAT has been spelt incorrectly or omitted from the EDIT statement.

186 "ORGANISATION" EXPECTED

The ORGANISATION clause has been spelt incorrectly or omitted from the FD statement.

187 FD ITEM TOO LARGE/LONG

A parameter coded for an FD construct is either too long or has too great a value. For example, a record length exceeding 32767 bytes.

188 INVALID FD ITEM

A parameter coded for an FD construct is invalid or omitted. For example, coding "ABC" as record length.

189 INVALID ORGANISATION

The ORGANISATION IS statement in the FD was not followed by INDEXED-SEQUENTIAL, RELATIVE-SEQUENTIAL or UNDEFINED.

190 "ASSIGN TO UNIT" EXPECTED

Self-explanatory.

191 "FILE" EXPECTED

The FILE clause has been incorrectly spelt or omitted from the FD construct ASSIGN statement.

192 INVALID FD OPTION

An FD option has been coded which is invalid for the access method. For example, a KEY statement with an UNDEFINED access method.

193 "KEY IS" EXPECTED

The key word IS has been omitted or spelt incorrectly in an FD construct.

194 "LENGTH IS" EXPECTED

The key word IS has been omitted or spelt incorrectly in an FD construct.

195 "SIZE IS" EXPECTED

The key word IS has been omitted or spelt incorrectly in an FD construct.

196 DUPLICATE FD OPTION

The same option has been specified twice within an FD. For example, two OPTION ERROR statements.

197 "ERROR", "RESET", OR "IGNORE" REQUIRED

The FD OPTION statement is not followed by the key word ERROR or RESET or IGNORE.

198 "ON ERROR" EXPECTED

The FD ON statement is not followed by the key word ERROR.

199 INVALID ACTION FOR ORGANISATION

A FETCH statement or a READ PRIOR, READ FIRST or READ LAST statement has been coded to access an FD. This is invalid.

200 (RE)WRITE FD "FROM" EXPECTED

A WRITE or REWRITE instruction has had the FROM key word spelt incorrectly or omitted.

201 LOCK FD "WAIT" EXPECTED

A LOCK statement contains characters after the area specification which is not the WAIT option. Probably spelt incorrectly.

202 FD NAME EXPECTED

An FD name defined in the Data Division was expected, but not found.

203 OPEN "OLD", "NEW", OR "SHARED" EXPECTED

An OPEN statement does not contain one of the options OLD, NEW or SHARED.

204 "TRUNCATE" OR "DELETE" EXPECTED

A CLOSE statement has characters at the end of the instruction which is not one of TRUNCATE or DELETE.

205 BASED ITEM INVALID IN WINDOW

BASED variables, which includes most systems variables, may not be referenced within a window.

206 INVALID/ILLEGAL VALUE ASSIGNMENT

A value assignment made for an item in the Data Division is invalid or illegal. For example, attempting to add the value "ABC" to a numeric field.

207 VALUE ASSIGNMENT IS TOO LONG

A value assignment made to a character variable is longer than that character variable. Any excess characters will be ignored.

208 COPYBOOK NOT FOUND

The book ID coded for a copy statement was not found in the copy libraries specified in the initial compiler dialogue.

209 TOO MANY NESTED COPY STATEMENTS

A copy book may itself copy further copy books and so forth up to a maximum of eight nested copies.

210 ".END" WITHOUT COPYBOOK

A .END statement has been found within the program source.

211 COPYBOOK NAME EXPECTED

The copybook name is missing from the COPY statement.

212 INVALID SUBSTITUTION VALUE

The target of the SUBSTITUTING clause in the COPY statement is invalid. The name must be enclosed in quotes and no longer than eight characters.

213 OPERAND TOO SHORT

The region code operand in the LOCK or UNLOCK statement is smaller than four bytes.

214 RECURSIVE CALLS ARE ILLEGAL

A CALL statement has passed the called entry-point as a parameter. This is illegal.

219 TOO MANY SYSTEM ROUTINES REQUIRED

The frame requires more than 127 compilation modules to be linked into the current frame.

220 TOO MANY GLOBAL SYMBOLS IN LINK

The frame requires more than 512 global symbols, which has exceeded the compiler global table capacity.

221 DUPLICATE GLOBAL SYMBOL DETECTED

A global symbol is defined in two or more places within the current frame, and/or linked compilation modules.

222 COMPILATION LIBRARY IS CORRUPT

One or more of the compilation files specified in the initial compiler dialogue is corrupt, and must be replaced prior to re-compilation.

223 COMPILER INTERNAL LINKING ERROR

An internal error has arisen during the linkage edit phase of compilation. Contact your support team.

224 PROGRAM SIZE EXCEEDS 32K

The current frame exceeds the 32K compiler limitation. The program must be reduced in size or segmented into an overlay structure.

225 EQ/GT/GE/LT/LE/NE/8/16/EX OR NX EXPECTED

A conditional intermediate code instruction, such as $JUMP, has been coded with an invalid or missing condition.

226 OPERAND IS WRONG FORMAT OR LENGTH

The indicated operand has a picture clause that is inconsistent with the coded statement, for example coding a 9(6) COMP variable in an EXIT WITH statement.

227 UNSUPPORTED INTERMEDIATE CODE INSTRUCTION

The coded intermediate code instruction is invalid or unsupported by the compiler (e.g. $CC 31).

228 INVALID SVC CALL

The System Service number coded with the SVC statement is invalid.

229 KEYWORD "MAP" EXPECTED

This keyword has been spelt incorrectly or omitted.

230 INVALID DISPLAY MAPPING FUNCTION

The coded function is not recognised, or is invalid with other options coded in the statement.

231 TOO MANY ACCESS STATEMENTS

More than 64 I/O channels have been declared using the ACCESS statement. The program must be segmented in order to allow access to more than 64 record types.

232 NO DICTIONARY OPEN

The dictionary ID coded in the ACCESS statement has not been opened during initial compiler dialogue.

233 "DEPENDENT ON" EXPECTED

One or both of the above key-words has been spelt incorrectly or omitted.

234 DUPLICATE WINDOW DIVISION STATEMENT

The WINDOW DIVISION statement has been coded twice within the current frame.

235 "ENDWINDOW" EXPECTED

The ENDWINDOW statement has been spelt incorrectly or omitted

236 WINDOW ID EXPECTED

The window ID has been omitted or miscoded in the Window Statement. This ID must be 2 characters long.

237 "USING" EXPECTED

The USING clause has been omitted from the Window Statement.

238 RECORD OR INDEX NAME EXPECTED

A record ID or index name must be coded following the USING clause in the Window Statement.

239 UNDECLARED RECORD OR INDEX NAME

The ID coded following the USING clause in a Window Statement is not a record type code or an index name. The ID has either been miscoded, or the record type has not been coded in an ACCESS statement.

240 "DEPENDENT ON" EXPECTED

One or both of the above key-words is spelt incorrectly or omitted.

241 UNDECLARED MASTER RECORD ID

One of the master record types coded in the window statement is undefined. It has probably not been coded in an ACCESS statement.

242 RECORD IS NOT A LINKED MASTER

The coded master record type is not directly linked from the target record type, and therefore may not be used.

243 CONTROLLING RECORD ID EXPECTED

The ID coded following the DEPENDENT ON Clause is not a valid record ID.

244 UNDECLARED CONTROLLING RECORD ID

The controlling record ID coded following the DEPENDENT ON Clause Is undefined. It has probably been omitted from the ACCESS statement.

245 RECORD HAS NO PRIMARY INDEX

The controlling record type coded following the DEPENDENT ON clause has no primary index, and may not therefore be used.

246 NO INDEX MATCHES CONTROLLING RECORD

The target index does not have an index which starts with the same fields as the controlling record's primary index key.

247 DEFAULT IDX CANNOT BE USED

The default index specified in the USING clause does not start with the same fields as the controlling record's primary index key, and cannot therefore be selected.

248 "POP-UP" INVALID WITH "QREM"

These two options are mutually exclusive.

249 CAN'T SEQUENCE TO SAME WINDOW

The target of the Window construct's SEQUENCE statement must differ from the current window ID. It is not possible for a window to go forwards or back to itself.

250 UNTIL "NXT" OR "CURRENT REC" EXPECTED

The repeat clause has been coded with an invalid option. One of the above two options must be used.

251 NOLOCK INVALID WITH UPDATES

The NOLOCK option has been specified in a window that is to perform updates. The mode-enabling clauses should be coded before the unlock statement.

252 OPTION REQUIRES TARGET RECORD

The option coded requires a target record type (as specified by the USING clause). For example, the ENQ, enable enquiries clause requires that the window has a target record type on which to enquire.

253 SPLIT *n* OFFSET *n* EXPECTED

One of the above keywords has been omitted or spelt incorrectly.

254 INVALID WINDOW DIMENSIONS

The line or column number coded, when combined with the effect of the data-item or text-item length, the BASE statement and the SCROLL statement, has exceeded the permitted window dimension. The permitted dimensions are columns 2 to 127 inclusive, and lines 2 to 46 inclusive.

255 BASE "AT" EXPECTED

The keyword "AT" has been omitted from the Window Construct BASE statement.

256 OPTION INVALID WITH TARGET RECORD

The indicated option is invalid for windows coded with the Window Statement's USING clause.

257 RECORD ACCESSED WITH SUBSTITUTION

The master record type coded after the USING clause in the Window Statement has a substituted name (i.e. was declared in an ACCESS statement coded with the SUBSTITUTING clause). This is not permitted.

258 SEL AND MNT ARE MUTUALLY EXCLUSIVE

Self-explanatory.

259 MUST BE CODED BEFORE LOCKING OPTION

The mode enabling clauses, ADD, ENQ, etc., must be coded before the LOCK and NOLOCK window options.

260 MORE THAN 127 FIELDS IN WINDOW

More than 127 data items have been coded for a given window, thus exceeding this compiler limitation. The window should be broken into a number of smaller windows.

261 NO DATA ITEMS IN WINDOW

The window contains no data-items, and can therefore only be displayed. Unpredictable result will occur if the window is entered.

262 NO INDEX AVAILABLE IN WINDOW

The target record type has no index by which it can be retrieved.

263 DEFAULT INDEX FIELDS MISSING

A default index was coded in the window construct's USING clause, but none of the index fields are coded as data-items.

264 TEXT ITEMS EXCEED 2K BYTES

Text items coded for the current window exceed the compiler buffer limitation of 2K. The amount of text displayed in the window must be reduced.

265 UNDECLARED WINDOW ID

A window ID coded in a SEQUENCE, ENTER CLEAR or DISPLAY statement is undefined.

266 ROUTINES SECTION MISPLACED

The Routines Section has been coded outside of the Window Division.

267 "SECTION" EXPECTED

The keyword "SECTION" is required.

268 INVALID ROUTINE NAME

An invalid routine name has been coded in the Routines Section.

269 "WINDOW" EXPECTED

The keyword "WINDOW" has been omitted from the CLEAR, DISPLAY or ENTER statement.

270 INVALID ACCEPT STATEMENT OPTION

An invalid accept statement option has been coded.

273 HELP COMMENTS ARE MISPLACED

Help text comments (i.e. help text preceded by a "\") may only be coded between the WINDOW Statement and the first Window Construct Detail Line.

274 HELP COMMENTS MAY NOT BE BROKEN

Only one group of help text comments may be coded for each window, which may not be interrupted by any intervening statements, including comment lines.

275 CONTROLLING "FRAME" EXPECTED

The key-word FRAME has been omitted or spelt incorrectly.

276 OBJECT FILE IS FULL

This limitation has been exceeded while writing the symbol table of a controlling frame to the frame's object file.

277 CONTROLLING FRAME NOT FOUND OR IN USE

The controlling frame specified within the FRAME statement was not found or in use by another partition.

278 PROGRAM IS NOT A CONTROLLING FRAME

The frame specified as a controlling frame was not compiled using the CONTROLLING FRAME statement.

279 CONTROLLING FRAME IS CORRUPT

The symbol table stored in the controlling frame could not be retrieved because of corruption. It should be recompiled.

280 DEPENDENT FRAMES SVCS MODULE DIFFERS

The controlling frame has been compiled using a different version of the compiler. It should be recompiled.

281 DISABLE "SKP", "CLR", "HME" EXPECTED

The Window Construct's DISABLE clause has not been followed by one of the above function mnemonics.

282 ENABLE "ABO" OR "NXT" EXPECTED

The window construct's ENABLE clause has not been followed by one of the above function mnemonics.

283 DEP FRAME'S DICTIONARIES DIFFER

The dictionary(s) used to compile the controlling frame differ, or have been specified in a different order, from the dictionary(s) now being used to compile the dependent frame

284 FIELDS MUST BE CONTIGUOUS

The field list specified as a controlling key within a dependent window is not located in contiguous memory. Furthermore, the fields must be defined in the same order within the Data Division as coded within the field list.

285 INVALID DISPLAY FORMAT

The display formats coded in a FMT clause are either invalid, or incorrectly combined.

286 VALID ONLY WITH NUMERIC FIELDS

The FMT option may only be used to re-format numeric fields.

287 INVALID WITH CHANGED PICTURE

The FMT option may not be used when a picture clause has been coded which differs from the variable's defined picture clause.

288 "TXT" ONLY VALID WITH PIC X

The TXT Window option may only be coded for PIC X variables.

289 BASE ADDRESS CORRUPTS SVCS MODULE

The explicit base address entered at compile time would conflict with memory space occupied by the services module. The base address has been increased to avoid corruption of this module.

290 BASE ADDRESS CORRUPTS CONTROLLING FRAME

The explicit base address entered at compile time would conflict with memory space occupied by the controlling frame. The base address has been increased to avoid corruption.

291 HIGH ADDRESS LIMIT EXCEEDED

The high address limit entered at compile time has been exceeded by the frame.

292 PRV8 REQUIRES DEPENDENT FRAME

Compiler option Privilege level 8 can only be assigned to Dependent Frames.

293 INVALID FOR DEPENDENT FRAME

The SWAP-FILE instruction may only be coded in non-dependent frames. When coded in the root frame of an overlay structure this instruction automatically applies to all dependent frames and it may not therefore be coded in them.

294 INVALID SWAP-FILE SIZE

A swap-file size has been coded that is not a valid numeric, or is not in the range 1 to 65535.

295 SWAP-FILE SIZE EXCEEDED

The indicated pop-up would exceed the allocated size of the swap-file and the pop-up's screen image will be stored in memory instead. To overcome this you should increase the size of the swap file. This message will also result if you attempt to store a pop-up wider than 85 bytes in the swap-file.

296 OPT "NSW" EXPECTED

The NSW clause has been omitted or spelt incorrectly.

297 UNBALANCED DO/IF STRUCTURE

The compiler has detected a DO.. ENDDO structure which is partly subject to an IF ... END Structure. The IF ... END structure must either wholly enclose an DO ... ENDDO structure or vice-versa.

298 COLON (:) EXPECTED

A colon is missing in the indicated position.

299 INVALID FOR LOCAL PF VARIABLE

The FMT and ADD options may only be used with referenced fields.

300 SOURCE FOR ADD MUST BE COMP

The source field for the ADD option must be a COMP variable. Note that the COMP specification should only be on the original declaration of the variable not in the PF construct.

Appendix C - Sample Application

```
FRAME TERR "Sales Territory Entry & Maintenance" **********************
**                                                                      **
**************************************************************************
*
ACCESS TR
*
WINDOW W1 USING TR ****************************************************
**                                                                      **
**    S A L E S T E R R I T O R Y W I N D O W                            **
**                                                                      **
**************************************************************************
*
\This window is used for the entry and maintenance of sales territory
\records. Information may be displayed in order of territory number,
\territory name or account manager. Also displayed is a reference
\and, in protected fields, the outstanding orders and cash.
*
REPEAT
AUTOPGE
*
LINE 4 13
BASE AT 5 16
SCROLL      8 BY 1 SPLIT 1 OFFSET 1
02 02 "Territory"
03 02 "Nmbr/Name"
            05 02 TRTRNO      X(4)  CHK NOE
            05 07 TRNAME      X(20)
*
02 29 "__Total-Outstanding__"
03 29 "Order-Amnt"
            05 29 TROSOR      S9(6,2) C PRO
03 40 " Cash-Amnt"
            05 40 TROSCH      S9(6,2) C PRO
14 02 "Account Manager" NSC
            14 18 TRACMG      X(15) NSC
15 02 "Reference" NSC
            15 18 TRREFN      X(11) NSC
*
ENDWINDOW
ENDFRAME
PAGE "CUST CUST MAINTENANCE"
FRAME CUST "Customer Entry & Maintenance" ****************************
**                                                                      **
**************************************************************************
*
ACCESS CU
*
WINDOW W1 USING CU ****************************************************
**                                                                      **
**          C U S T O M E R W I N D O W                                 **
**                                                                      **
**************************************************************************
*
\This window is used for entry and maintenance of customer records.
\Information may be displayed in the order of customer number, name
\or contact. Also displayed is the customer address, telephone
\number and, in protected fields, the outstanding orders and cash.
*
REPEAT
AUTOPGE
LINE 4 13
BASE AT 5 15
SCROLL      8 BY 1 SPLIT 1 OFFSET 1
02 02 "Customer"
03 02 "Number/Name"
            05 02 CUCSNO      X(6)  CHK NOE
            05 09 CUNAME      X(20)
02 31 "__Total-Outstanding__"
```

```
03 31 "Order-Amnt"
            05 31 CUOSOR      S9(6,2) C PRO
03 42 " Cash-Amnt"
            05 42 CUOSCH      S9(6,2) C PRO
14 02 "Address" NSC
            14 10 CUADD1      X(20) NSC
            15 10 CUADD2      X(20) NSC
14 32 "Contact" NSC
            14 40 CUCONT      X(12) NSC
15 32 "Phone"     NSC
            15 40 CUPHON      X(12) NSC
*
ENDWINDOW
ENDFRAME
PAGE "STCK STOCK MAINTENANCE"
FRAME STCK "Stock Entry & Maintenance " ******************************
**                                                                    **
************************************************************************
*
ACCESS ST
*
WINDOW W1 USING ST
*
\This window is used for the entry and maintenance of stock records.
\Records may also be retrieved, in three different sequences:
\
\ stock number
\ stock description
\ primary supplier number.
\
\Also displayed is the retail price, quantity in-hand and reserved,
\stock levels 1,2,3 and, in protected fields, the outstanding
\quantities from sales and purchase orders.
*
AUTOPGE REPEAT
LINE 4 13
BASE AT 4 7
SCROLL      8 BY 1 SPLIT 1 OFFSET 1
*
02 02 "Stock"
03 02 "Number"
            05 02 STSTNO      X(6)  CHK NOE
            05 10 STDESC      X(20)
02 31 "Primary"
03 31 "Supplier"
            05 32 STSUP1      X(6)
02 43 "Retail"
03 43 "Price"
            05 40 STRCRP      9(6,2) C
02 51 "_____ Units ____"
03 51 "On-hand"
            05 51 STOHND      S9(6) C
03 60 "reservd"
            05 60 STRSVD      S9(6) C
14 02 "Outstanding Orders....."
15 02 "Purchase Orders" NSC
            15 18 STPORD      S9(6) C NSC PRO
16 02 "Sales Orders" NSC
            16 18 STSORD      S9(6) C NSC PRO
14 32 "Bin-No"
            15 32 STBINO      X(6)  NSC
14 44 "Price Level-1"
            14 58 STLVL1      9(6,2) C NSC
15 50 "Level-2" NSC
            15 58 STLVL2      9(6,2) C NSC
16 50 "Level-3" NSC
            16 58 STLVL3      9(6,2) C NSC
*
03 10 "Description"
ENDWINDOW
ENDFRAME
PAGE "INVC INVOICE ENTRY"
```

```
FRAME INVC "Sales Invoice Entry & Maintenance" ************************
**                                                                    **
**********************************************************************
*
ACCESS CU TR IN
*
WINDOW W1 USING IN CU TR ***************************************************
**                                                                        **
**    S A L E S I N V O I C E W I N D O W                                  **
**                                                                        **
**************************************************************************
*
\This window is used for the entry and maintenance of sales invoices.
\Records may be displayed in four different sequences:
\
\Invoice number, Invoice-Date, Customer number, and Territory number.
\
\When a new invoice record is created, you must enter the appropriate
\customer and territory numbers. If you are not sure of these numbers
\you may key $TR-UF1 at the customer number or territory number field
\to display a pop-up enquiry window.
*
REPEAT
LINE 4 13
BASE AT 4 11
SCROLL      8 BY 1 SPLIT 1 OFFSET 1
02 02 "Customer"
03 03 "Number"
            05 03 INCSNO      X(6)  UF1 NOE
15 02 "Customer's Name :" NSC
            15 21 CUNAME      X(20) NSC DIS
02 12 "Invoice"
03 12 "Number"
            05 12 ININVC      X(6)  CHK NOE
03 22 "Date"
            05 20 INDATE      D
03 29 "Territory"
            05 31 INTRNO      X(4)  UF1
16 02 "Territory's Name :" NSC
            16 21 TRNAME      X(20) NSC DIS
03 40 "GL-Code"
            05 40 INGLCD      X(6)
02 51 "Invoice"
03 52 "Amount"
            05 49 INIAMT      S9(5,2) C NOE
14 02 "Invoice Narrative:" NSC
            14 21 INDESC      X(28) NSC NUL
*
PAGE
ROUTINES SECTION ***********************************************************
**                                                                        **
**************************************************************************
*
V-INCSNO.                                 * VALIDATE CUSTOMER NUMBER...
      IF $FUNC = 1                        * IF UF1
      ENTER WINDOW W2                     * - ENTER CUSTOMER POP-UP
      ON EXCEPTION EXIT WITH 1            * EXCEPTION MEANS <BCK> KEYED
      MOVE CUCSNO TO INCSNO         * ELSE SAVE SELECTED CUST #
      EXIT                                * ALL DONE
      END
      IF $MODE = 1 EXIT                   * NO VALIDATION IN ENQ MODE
      FETCH CUCSN KEY INCSNO PROTECT RETRY 3    * FETCH CUST REC & PROTECT IT
      ON NO EXCEPTION EXIT                      * IF THERE ALL DONE
      ERROR " Customer Locked or Not Found "    * ELSE AN ERROR
      EXIT WITH 1
*
V-INTRNO.                                 * VALIDATE TERRITORY NUMBER...
      IF $FUNC = 1                        * IF UF1
      ENTER WINDOW W3                     * - ENTER TERRITORY POP-UP
      ON EXCEPTION EXIT WITH 1            * EXCEPTION MEANS <BCK> KEYED
      MOVE TRTRNO TO INTRNO         * ELSE SAVE SELECTED TERR #
      EXIT                                * ALL DONE
```

```
      END
      IF $MODE = 1 EXIT             * NO VALIDATION IN ENQ MODE
      FETCH TRTRN KEY INTRNO PROTECT RETRY 3    * FETCH TERRITORY RECORD
      ON NO EXCEPTION EXIT                * IF THERE ALL DONE
      ERROR " Territory Locked or Not Found " * ELSE AN ERROR
      EXIT WITH 1
*
D-INDATE.                                 * DEFAULT INVOICE DATE
      MOVE $$DATE TO INDATE               * TO TODAYS DATE
      EXIT                                *
*
ENDWINDOW
*
PAGE
WINDOW W2 USING CUNAM ********** C U S T O M E R P O P - U P *******
*                     ********************************************
*
\This optional window is used to select a customer during invoice
\entry. You may display customer records using the usual enquiry
\facilities, and select one from the displayed list.
*
POP-UP SEL
AUTOPGE
SEQUENCE EXIT CLW, EXIT CLW
REPEAT UNTIL CURRENT RECORD
LINE 4 10
BASE AT 10 45
SCROLL      5 BY 1 SPLIT 1 OFFSET 1
*
02 02 "Customer"
            05 02 CUCSNO      X(6)
03 02 "Number/Name"
            05 09 CUNAME      X(20)
            11 02 CUADD1      X(20) NSC
            12 02 CUADD2      X(20) NSC
*
ENDWINDOW
*
*
WINDOW W3 USING TRNAM ********** T E R R I T O R Y P O P - U P ****
*                     *********************************************
*
\This optional window is used to select a territory during invoice
\entry. You may display territory records using the usual enquiry
\facilities, and select one from the displayed list.
*
POP-UP SEL
AUTOPGE
SEQUENCE EXIT CLW, EXIT CLW
REPEAT UNTIL CURRENT RECORD
*
LINE 4 10
BASE AT 10 45
SCROLL      5 BY 1 SPLIT 1 OFFSET 1
02 02 "Territory"
            05 02 TRTRNO      X(4)
03 02 "Number/Name"
            05 09 TRNAME      X(20)
11 02 "Account Manager" NSC
            12 02 TRACMG      X(15) NSC
11 18 "Reference" NSC
            12 18 TRREFN      X(11) NSC
*
ENDWINDOW
ENDFRAME
PAGE "CENQ CUSTOMER/INVOICE ENQUIRY"
FRAME CENQ "Customer Invoice Enquiry" ********************************
**                                                                    **
************************************************************************
*
ACCESS IN CU
*
```

```
WINDOW W1 USING CU ************************************************
**                                                                **
**    CUSTOMER SELECTION WINDOW..                                  **
**                                                                **
********************************************************************
*
\This window is used to select a customer using the usual enquiry
\facilities. Once a customer has been selected, the next window
\is used to display that customer's invoices.
*
SEL
SEQUENCE EXIT W2
REPEAT UNTIL CURRENT RECORD
*
LINE 3
BASE AT 4 6
02 02 "Customer Number"
            02 19 CUCSNO      X(6)
02 30 "Name"
            02 36 CUNAME      X(20)
04 02 "Contact"
            04 11 CUCONT      X(12)
05 02 "Address"
            05 11 CUADD1      X(20)
            06 11 CUADD2      X(20)
04 33 "Telephone"
            04 44 CUPHON      X(12)
05 33 "O/S-Order"
            05 46 CUOSOR      S9(6,2) C
06 33 "O/S-Cash"
            06 46 CUOSCH      S9(6,2) C
*
ENDWINDOW
PAGE
WINDOW W2 USING IN DEPENDENT ON CU **********************************
**                                                  **
**    INVOICE DISPLAY WINDOW..                            **
**                                                  **
********************************************************************
*
\This window is used to display invoices for the selected customer.
\Invoices are displayed in invoice number order.
*
SEQUENCE W1 CLW
REPEAT
ENQ POP-UP AUTOPGE
*
LINE 4
BASE AT 8 11
SCROLL 10 BY 1 SPLIT 1 OFFSET 1
02 02 "Invoice"
03 02 "Number"
            05 02 ININVC      X(6)
03 11 "Date"
            05 09 INDATE      D
03 18 "Description"
            05 18 INDESC      X(28)
03 46 "GL-Code"
            05 47 INGLCD      X(6)
03 57 "Amount"
            05 54 INIAMT      S9(5,2) C
*
ENDWINDOW
ENDFRAME
PAGE "ORDER ORDER ENTRY "
FRAME ORDER "Order Entry and Maintenance" ****************************
**                                                  **
************************************************************************
*
ACCESS CU TR OR OL ST
*
DATA DIVISION
```

```
*
77 STLVL REDEFINES STLVL1 OCCURS 3 PIC 9(6,2) C * TO INDEX PRICE LEVELS.
*
WINDOW W1 USING ORORD CU TR
*
\This frame is used for entry and maintenance of orders. The initial Order
\Header window may be used to select an existing order for modification or
\to create a new order. When a new order is created,  you must enter    the
\appropriate customer and territory numbers. If you are not sure of these
\numbers, you may key   $TR-UF1 at the customer Number or territory Number
\field to display an enquiry window.
\
\The initial Order Header window is then followed by the Order    Line window
\where lines belonging to the order may be created or displayed.
*
ADD MNT                         * ENABLE ADD,ENQ/SEL & MNT MODES
REPEAT UNTIL CURRENT RECORD          * FWD EXIT ONLY WITH LOCKED RECORD
ENABLE NXT                      * ALLOW <NXT> FOR QUICK SELECTION
LINE 3 6
BASE AT 3 3
02 02 "Order      Number"
            02 20 ORORDN      X(6)  CHK NOE
02 42 "Order Total"
            02 55 ORTOTL      9(6,2) C PRO
04 02 "Customer Number"
            04 20 ORCSNO      X(6)  UF1 NOE
            04 42 CUNAME      X(20) DIS
05 02 "Territory Number"
            05 20 ORTRNO      X(4)  UF1 NOE
            05 42 TRNAME      X(20) DIS
07 02 "Delivery Address"
            07 20 ORDAD1      X(20) TAB NUL
            08 20 ORDAD2      X(20) NUL
09 02 "Contact"
            09 20 ORCONT      X(12) NUL
10 02 "Telephone"
            10 20 ORPHON      X(12) NUL
07 42 "Order Date"
            07 56 ORORDT      D     NUL
08 42 "Required Date"
            08 56 ORRQDT      D
09 42 "Price Code"
            09 63 ORPRCD      9 C   NOE
10 42 "Notes"
            10 54 ORDLNO      X(10) NUL
*
PAGE
ROUTINES SECTION ************************************************************
**                                                                        **
**************************************************************************
*
V-ORCSNO.                       * VALIDATE CUSTOMER NUMBER...
      IF $FUNC = 1                   * IF UF1
      ENTER WINDOW W3           * - ENTER THE CUST POP-UP
      ON EXCEPTION EXIT WITH 1       * - EXCEPTION MEANS <BCK> KEYED
      MOVE CUCSNO TO ORCSNO     * - ELSE SAVE SELECTED CUSTOMER #
      EXIT                      * - ALL DONE.
      END
      IF $MODE = 1 EXIT         * NO VALIDATION IN ENQ MODE
      FETCH CUCSN KEY ORCSNO PROTECT RETRY 3 * FETCH CUST REC & PROTECT IT
      ON NO EXCEPTION EXIT           * IF THERE ALL DONE
      ERROR " Customer Locked or Not Found " * ELSE AN ERROR
      EXIT WITH 1
*
V-ORTRNO.                       * VALIDATE TERRITORY NUMBER...
      IF $FUNC = 1                   * IF UF1
      ENTER WINDOW W4           * - ENTER THE TERRITORY POP-UP
      ON EXCEPTION EXIT WITH 1       * - EXCEPTION MEANS <BCK> KEYED
      MOVE TRTRNO TO ORTRNO     * - ELSE SAVE SELECTED CUSTOMER #
      EXIT                      * - ALL DONE.
      END
      IF $MODE = 1 EXIT         * NO VALIDATION IN ENQ MODE
```

```
      FETCH TRTRN KEY ORTRNO PROTECT RETRY 3 * FETCH THE TERITORY RECORD
      ON NO EXCEPTION EXIT          * IF THERE ALL DONE
      ERROR " Territory Locked or Not Found "* ELSE AN ERROR
      EXIT WITH 1
*
D-ORORDT.                     * DEFAULT ORDER DATE
      MOVE $$DATE TO ORORDT         * TO TODAYS DATE
      EXIT
*
V-ORPRCD.                     * VALIDATE THE PRICE-CODE
      IF ORPRCD POSITIVE            * MUST BE > 0
      IF ORPRCD < 4 EXIT            * .. AND < 5
      END
      ERROR " Price Code must be in range 1 TO 3 "
      EXIT WITH 1
*
ENDWINDOW
PAGE
WINDOW W2 USING OL$SQ DEPENDANT ON OR ************************************
**                                                          **
**    O R D E R L I N E I T E M     W I N D O W . . .       **
**                                                          **
******************************************************************************
*
\This window is used to add line items to the current order, or to amend
\existing order lines.  When a new order line is added, the stock number
\of the order line must be entered. If you do not know the appropriate
\stock number key $TR-UF1 at the Stock Number field to display the stock
\selection window.
\
\Keying $TR-NXT at any stage completes the order, and another order may
\then be entered or amended.
*
POP-UP                              * THIS WINDOW POPS UP
SEQUENCE EXIT CLW, EXIT CLW         * EXIT TO PROCEDURE DIVISION
AUTOPGE                       * DISPLAY 1ST PAGE ON ENTRY
REPEAT UNTIL NXT              * LOOP TILL <NXT> KEYED.
*
LINE 4
BASE AT 8 8
SCROLL      8 BY 1 SPLIT 1 OFFSET 1
02 02 "Stock"
03 10 "Description"
            05 02 OLSTNO      X(6)  UF1 CHK NOE
03 02 "Number"
            05 10 OLDESC      X(20) DIS
02 33 "Date"
03 33 "Reqd"
            05 31 OLDTRQ      D
02 45 "Unit"
03 44 "Price"
            05 41 OLPRCE      9(5,2) C NOE
02 51 "Order"
03 52 "Qty"
            05 51 OLORQT      9(5) C NOE
02 62 "Line"
03 61 "Amount"
            05 58 OLLAMT      S9(5,2) C DIS
PAGE
ROUTINES SECTION ***********************************************************
**                                                                      **
**********************************************************************
*
V-OLSTNO.                     * VALIDATE STOCK #
      MOVE ORORDN TO OLORDN         * INITIALISE THE ORDER NUMBER
      MOVE ORTRNO TO OLTRNO         *     THE TERRITORY NUMBER
      MOVE ORCSNO TO OLCSNO         *     THE CUSTOMER NUMBER
      IF $FUNC = 1                  * IF 1 KEYED..
      ENTER WINDOW W5         * - ENTER THE STOCK POP-UP
      ON EXCEPTION EXIT WITH 1      * - EXCEPTION MEANS <BCK> KEYED
      MOVE STSTNO TO OLSTNO   * - ELSE SAVE STOCK#
      EXIT                    * - ALL DONE.
```

```
      END                        *
      IF $MODE = 1 EXIT          * SUPPRESS VALID'N IN ENQ MODE
      FETCH STSTN KEY OLSTNO PROTECT RETRY 3    * FETCH THE STOCK RECORD
      ON EXCEPTION                          * EXCEPTION MEANS...
      ERROR " Stock record Locked or Not Found " * ELSE AN ERROR
      EXIT WITH 1
      END
      MOVE STDESC TO OLDESC         * MOVE DESCRIPTION TO O/L RECORD
      EXIT
*
D-OLDTRQ.                       * DEFAULT DATE REQUIRED
      MOVE ORRQDT TO OLDTRQ         * SET LINE DATE TO HEADER DATE.
      EXIT
*
D-OLPRCE.                       * DEFAULT UNIT PRICE
      MOVE STLVL(ORPRCD) TO OLPRCE  * SELECT PRICE AS PER PRICE LEVEL
      EXIT                      * AND EXIT
*
V-OLORQT.                       * VALIDATE ORDER QUANTITY
      MULTIPLY OLORQT BY OLPRCE GIVING OLLAMT
      ON EXCEPTION                  * MAKE SURE NOT TOO BIG!!
      OR OLLAMT NOT POSITIVE        * VALIDATE OUTSTANDING QTY AND
      OR OLLAMT > 99999         * CALCULATE LINE VALUE.
      ERROR " Invalid Qty or Price "
      EXIT WITH 1               * EXIT WITH 1 INDICATES INVALID
      END                       *
      EXIT                      * NORMAL EXIT MEANS ALL OK.
*
R-WRITE.                        * INITIALISE PRIOR TO WRITE
      MOVE OLORQT TO OLOSQT         * O/S QTY = ORIGINAL QTY
      EXIT
*
ENDWINDOW
PAGE
*****************************************************************************
**                                                                         **
**    MISC POP-UPS....                                                     **
**                                                                         **
*****************************************************************************
*
WINDOW W3 USING CUNAM               * C U S T O M E R P O P - U P
*
\This optional window is used to select a customer during entry of the
\Order Header window. You can display customer records using the usual
\Enquiry facilities, and select one from the displayed list.
*
POP-UP SEL
SEQUENCE EXIT CLW, EXIT CLW
REPEAT UNTIL CURRENT RECORD
AUTOPGE
*
LINE 4 10
BASE AT 11 43
SCROLL      5 BY 1 SPLIT 1 OFFSET 1
02 02 "Customer"
            05 02 CUCSNO      X(6)
03 02 "Number/Name"
            05 09 CUNAME      X(20)
            11 02 CUADD1      X(20) NSC
            12 02 CUADD2      X(20) NSC
ENDWINDOW
*
WINDOW W4 USING TRNAM               * T E R R I T O R Y P O P - U P
*
\This optional window is used to select a Territory during entry of the
\Order Header window. You can display Territory records using the usual
\Enquiry facilities, and select one from the displayed list.
*
POP-UP SEL
SEQUENCE EXIT CLW, EXIT CLW
REPEAT UNTIL CURRENT RECORD
AUTOPGE
```

```
*
LINE 4 10
BASE AT 11 42
SCROLL      5 BY 1 SPLIT 1 OFFSET 1
02 02 "Territory"
            05 02 TRTRNO      X(4)
03 02 "Number/Name"
            05 09 TRNAME      X(20)
11 02 "Account Manager"
            12 02 TRACMG      X(15) NSC
11 18 "Reference"
            12 18 TRREFN      X(11) NSC
ENDWINDOW
PAGE
WINDOW W5 USING STDES               * S T O C K P O P - U P
*
\This optional window is used to select a Stock record during entry of
\order      line items. You can     display stock records using the usual
\Enquiry facilities, and select the required stock item from the list.
*
POP-UP SEL
SEQUENCE EXIT CLW, EXIT CLW
REPEAT UNTIL CURRENT RECORD
AUTOPGE
*
LINE 4 8
BASE AT 12 40
SCROLL      3 BY 1 SPLIT 1 OFFSET 1
02 02 "Stock"
03 02 "Number/Description"
            05 02 STSTNO      X(6)
            05 09 STDESC      X(20)
09 02 "Quantity on Hand" NSC
            09 22 STOHND      S9(6)C NSC
10 02 "On-hand/Reserved" NSC
            10 22 STRSVD      S9(6)C NSC
11 02 "Retail Price" NSC
            11 20 STRCRP      9(6,2)C NSC
*
ENDWINDOW
*
PROCEDURE DIVISION ************************************************
**                                                  **
**    ALL THE WINDOWS ARE CONTROLLED HERE!                  **
**                                                  **
**********************************************************************
*
AA-000. ENTER WINDOW W1             * ENTER ORDER HEADER WINDOW
      ON EXCEPTION EXIT WITH 1      * <BCK> EXIT..
*
      ENTER WINDOW W2         * ENTER LINE-ITEM WINDOW
      ON EXCEPTION                  * IGNORE <BCK>
      END                     *
      DISPLAY WINDOW W1       * REDISPLAY HEADER TO UPDATE
      GOTO AA-000             * THE ORDER TOTAL & RE-ENTER.
*
ENDFRAME
PAGE "REP01 STOCK STATUS REPORT"
FRAME REP01 "Stock Status/ Forward Orders Report" ******************
**                                                  **
**********************************************************************
*
ACCESS CU ST OL               * CUST/STOCK/ORDER LINE RECS.
*
DATA DIVISION
*
77 Z-ORQT         PIC S9(9) COMP    * TOTAL # ON ORDER
77 Z-OSQT         PIC S9(9) COMP    * TOTAL O/S QTY
77 Z-LAMT         PIC S9(9,2)COMP * TOTAL VALUE OF ORDERS
*
PF H1                         * PRINT FORMAT FOR HEADER
START AT 1                    * PRINT AT LINE 1 ONLY
```

```
*
01 01 "SPEEDBASE V3 DEMONSTRATION SYSTEM"
01 52 "STOCK STATUS - FORWARD ORDERS"
01 107 "DATE:"          01 113 $$DATE
01 123 "PAGE:"          01 128 $PGNO
03 10 "STOCK#     DESCRIPTION"
03 40 "REQ-DATE ORDER# CUST# CUSTOMER NAME"
03 86 "ORDER-QTY O/S-QTY      VALUE"
ENDFORMAT
*
PF D1 HEADER H1               * PRINT FORMAT FOR DETAIL LINE
START FROM 5                        * PRINT FROM LINE 5 ONWARDS
*
01 10 OLSTNO                        * STOCK#
01 19 OLDESC                        * DESCRIPTION
01 40 OLDTRQ                        * DATE REQ'D
01 19 OLDTRQ                        * DATE REQ'D
01 50 OLORDN                        * ORDER NUMBER
01 58 OLCSNO                        * CUSTOMER NUMBER
01 65 CUNAME                        * CUSTOMER NAME
01 90 OLORQT      ADD(Z-ORQT)       * ORDER QUANTITY
01 100 OLOSQT     ADD(Z-OSQT)       * OUTSTANDING QTY
01 109 OLLAMT     ADD(Z-LAMT)       * LINE TOTAL
ENDFORMAT
*
PF T1 HEADER H1               * FORMAT FOR TOTAL LINES
START FROM 5                        * PRINT FROM LINE 5 ONWARDS
*
01 40 "ON-HAND:-" 01 49 STOHND
02 40 "RESERVD:-" 02 49 STRSVD
01 58 "P-ORDRS:-" 01 67 STPORD
02 58 "S-ORDRS:-" 02 67 STSORD
02 87 "========"  01 87 Z-ORQT      S9(7)
02 97 "========"  01 97 Z-OSQT      S9(7)
02 107 "==========="    01 107 Z-LAMT     S9(7,2)
03 01 ""
ENDFORMAT
PAGE
WINDOW S1               * A C C E P T     O P T I O N S
*
\This frame produces the Stock Status Forward Orders report. It gives
\an example of the use of the PF construct in developing report pro-
\grams.
*
LINE 4
BASE AT 10 21
02 02 "First Stock# Required"
            02 32 S1STK1      X(6)  NUL
03 02 "Last Stock# Required"
            03 32 S1STK2      X(6)
05 02 "Restart from page No"
            05 32 S1RSPG      9(4) C NUL
*
ROUTINES SECTION
*
V-S1STK1.                     * VALIDATE FIRST STOCK#
      FETCH FIRST OLSTN KEY S1STK1 NOLOCK* SEE IF ANY AT ALL..
      ON EXCEPTION                  * IF STOCK # DOES NOT EXISTS
      AND $$COND = 4                * AND IT'S END OF FILE
      ERROR " Invalid Stock Number " * - MUST BE INVALID
      EXIT WITH 1             *
      END
      EXIT
*
D-S1STK2.                                       * DEFAULT SECOND STOCK#
      MOVE "ZZZZZZ" TO S1STK2                   * CAN'T BE HIGHER THAN THAT.
      EXIT
*
V-S1STK2.                                       * VALIDATE SECOND STOCK #
      IF S1STK2 NOT < OLSTNO EXIT               * OK IF >= TO 1ST ITEM FOUND
      ERROR " No Items to Print "               * ELSE NOTHING TO PRINT
      EXIT WITH 1
```

```
*
R-PROCESS.
*
      MOVE S1RSPG TO $RSPG                        * SET UP START FROM PAGE
      DO UNTIL OLSTNO > S1STK2                    * PRINT EACH STOCK#
      MOVE 0 TO Z-ORQT Z-OSQT Z-LAMT              * CLEAR TOTALS
      FETCH STSTN KEY OLSTNO NOLOCK       * FETCH THE STOCK RECORD
      DO                                          * NOW PROCESS EACH LINE
      FETCH CUCSN KEY OLCSNO NOLOCK       * FETCH CUST RECORD
      PRINT D1                                    * PRINT THE LINE ITEM
      FETCH NEXT OLSTN KEY OLSTNO NOLOCK  * GET NEXT LINE ITEM
      ON EXCEPTION FINISH                         * NO MORE FOR THIS ITEM
      ENDDO                                       * ALL ORDER LINES DONE
      PRINT T1                                    * SO PRINT THE TOTALS
      ENDDO                                       * DO THE NEXT STOCK#
      EXIT                                        * REPORT FINISHED.
*
ENDWINDOW
ENDFRAME
ENDSOURCE
```

Appendix D - EXIT and STOP Codes

This appendix documents the EXIT and STOP errors that may occur during the running of a Speedbase application. When the Speedbase Presentation Manager detects an error of this type, the EXIT or STOP code is displayed at the baseline. Please report these error messages to your software supplier.

D.1 EXIT Codes

This sections describes EXIT codes generated by the Speedbase Presentation Manager. EXIT codes occur when an exception condition arises within your Speedbase application program and the exception has not been trapped by an ON EXCEPTION or ON OVERFLOW statement. This causes the following error message to be displayed:

$91 TERMINATED - EXIT *nnnnn*

The number *nnnnn* is the exit code, the meanings of which are described later in this section. It should be noted that only exit conditions that are produced by the Speedbase Presentation Manager are documented here. Those produced by the GSM and various system routines are described in detail in the GSM Utilities Manual.

EXIT 25301 A window was terminated by the <BCK> function.

EXIT 25302 A window (or series of windows) was terminated by the use of the <ABO> function.

EXIT 253*nn* An exception condition was returned by a window Process Routine in order to terminate the window.

EXIT 254*nn* An accept operation was terminated using a function other than <RET>.

EXIT 25501 A record locked exception was returned following a GET, READ, or FETCH statement coded without the NOLOCK clause.

EXIT 25502 Requested record key not found. The index key specified in a READ or FETCH statement was not found. In sequential operations in V8.1 and later the next or prior record is returned unlocked.

EXIT 25503 Exception conditions 25501 and 25502 have occurred simultaneously using the READ or FETCH FIRST, NEXT, LAST or PRIOR statements.

EXIT 25504 An end-of-file or start-of-file condition has occurred when using a READ or FETCH FIRST, NEXT, LAST or PRIOR statement. This condition can also occur with the GET statement, when the requested RRN key is beyond the logical end-of-file of the target record type.

EXIT 25505 An attempt has been made to DELETE a record with an active (i.e. non-zero) servant record group.

EXIT 25506 The target of a WRITE statement contained a primary index key which already existed on the database.

EXIT 25507 A WRITE statement could not be completed because no free data records remain in the database for the target record type.

EXIT 25508 The RRN key coded for a GET statement specified a deleted record.

EXIT 25511 A date conversion from display to computational format using a MOVE statement failed because the display format date was invalid.

EXIT 25513 A MOUNT statement could not be fulfilled by the operator.

EXIT 25520 The routine B$CHK has detected that the Speedbase system area $BASYS is loaded.

EXIT 25521 An attempt was made to open the same database twice using database open routine B$OPN.

EXIT 25522 The database open routine B$OPN was unable to open the specified database either because it was not present on the specified unit, or the file-type was incorrect.

EXIT 25523 The data-file(s) belonging to a database could not be opened by the database open routine B$OPN. This problem is usually caused by incorrect unit assignment.

EXIT 25524 A database could not be opened because it was already exclusively opened by another partition.

EXIT 25525 An I/O error occurred during database open.

EXIT 25526 Insufficient memory on user stack to open database.

EXIT 25527 A call on B$OPN could not complete because the DB index file could not be opened within the UNIX directory, and the resulting error was not trapped.

EXIT 25528 A call on B$OPN could not complete because a C-ISAM channel could not be initialised, and the resulting error was not trapped. This may occur when insufficient memory is available.

EXIT 25529 A duplicate key condition has been detected within a C-ISAM file and the resulting exception was not trapped. The database should be rebuilt.

EXIT 25530 The system area $BASYS could not be loaded prior to executing a frame. This occurs if the program $BASYS is not found, an I/O error occurs, or if there is insufficient room on the user stack to load this module.

EXIT 25531 Invalid display formatting codes passed to qualifier definition routine B$QLN.

EXIT 25532 The display format codes passed to B$QLN specify positive or negative value highlighting (<>DC+-) although the target operand's format is un-signed.

EXIT 25542 An attempt has been made to run a frame compiled using the V4.0 development system within the V3.0 Speedbase Presentation Manager.

EXIT 25543 An attempt has been made to execute a frame which is not at the anticipated depth in an overlay structure. For example, this will happen when using the SEQUENCE statement to transfer control between frames at different levels in an overlay structure.

EXIT 25544 An attempt has been made to execute a dependent frame which was not compiled with the currently loaded controlling frame.

EXIT 25546 The system area $BASYS could not be loaded prior to executing a frame. This occurs if the program $BASYS is not found, an I/O error occurs, or if there is insufficient room on the user stack to load this module.

EXIT 25547 A frame load has failed because the required program was not found, or was too large to fit into the available memory. This exit condition will also result if an I/O error occurs during loading.

EXIT 25548 The field name coded in a call on the B$WRJ right justification routine was not found in the coded window-id.

EXIT 25549 A call on B$XCL failed because the routine could not provide exclusive access, and the resulting exception was not trapped.

EXIT 25550 A call on B$STA or B$ST2 failed because the required database was not open, and the resulting exception was not trapped.

EXIT 25551 XA$FAM exception condition caused by unusual UNIX file processing condition. $$RES holds corresponding result code. This error will not occur in application programs.

D.2 STOP Codes

This section describes STOP codes generated by the Speedbase Presentation Manager. STOP codes occur when a serious processing condition arises within your application program, from which recovery is not possible. This causes the following error message to be displayed:

$91 TERMINATED - STOP *nnnnn*

The number *nnnnn* is the stop code as listed and described in detail later in this section. Note that only stop codes produced by the Speedbase Presentation Manager are documented here. The GSM and various system routines also produce stop codes. These are described in detail in the GSM Utilities Manual.

Note that certain STOP codes can arise during error checking performed by the Speedbase Presentation Manager, and indicate that the database is corrupt. In this event the database should be restored or rebuilt before any further processing is performed.

STOP 25301 An I/O operation has been performed on the I/O channel of a window's target record type during execution of the window.

STOP 25302 An unsupported I/O operation has been attempted by the window processor. This error indicates a system software malfunction.

STOP 25303 A clear operation has been attempted on a window that has not yet been displayed.

STOP 25304 A DISPLAY WINDOW . . TEXT statement has been executed when the text of the window was already displayed.

STOP 25305 An attempt has been made to display a window using the DISPLAY WINDOW statement while the I/O channel of the window's target record type did not contain a record.

STOP 25306 A recursive call has been made on a window that is executing. For example, this error will occur if a window is cleared from the routines section while still executing as a result of a prior entry to the window.

STOP 25307 The index key length coded within a window does not match the key-length of the target record type. This error should not occur when processing records stored on the database.

STOP 25308 A CLEAR statement has been performed while windows are executing. This error can also occur if a dependent frame without the NOCLEAR option has been EXECed while windows are still active in the controlling frame.

STOP 25501 Database not open or generation number mismatch. This condition arises when a database required by the loaded program is not open, or the open data-base has a generation number different from that expected by the program. This error occurs if old, superseded versions of programs are run.

STOP 25502 Rewrite or delete without current record. An application program has attempted to REWRITE or DELETE a record which was not locked at the time the instruction was executed.

STOP 25503 I/O area outside current partition. A database access verb has specified a data record area which is not within the current program partition. Suspect program file corruption or a compiler malfunction.

STOP 25504 Primary index key modification. A REWRITE instruction has attempted to modify the target record's primary index key. This is illegal.

STOP 25505 Data record not locked. This stop code indicates an internal error within the Speedbase DBMS. The M$DW I/O routine has detected a data re-write without the presence of a record level lock.

STOP 25506 Master record not locked. A WRITE or REWRITE Instruction has taken place in which master records to be linked to the target record were not locked. The DBMS has recovered from this error and completed the I/O request correctly before terminating the program with this code.

STOP 25507 GET key is negative. The RRN specified in the GET verb's KEY clause was negative. This is invalid. This error can also arise if a GET statement is coded without KEY clause. This occurs when no I/O has so far taken place, or the last retrieval returned an end-of-file or start-of-file condition.

STOP 25513 Illegal line or col. The line or column number coded as a variable in a DISPLAY or ACCEPT... LINE statement is invalid. The number was either not positive or else exceeded the boundaries of the screen.

STOP 25514 PF structure exceeds 16 levels. An attempt has been made to execute a PF construct with more than 16 levels of headers or trailers using the PRINT statement. The PF statement may specify a PF as a header or trailer, and this PF may itself specify a further PF as a header or trailer. However, this is limited to a maximum of 16 levels of PFs.

STOP 25515 An internal error has been detected within the database rebuild utility, and it is likely that this is caused by an programming problem within this program. This problem could alternatively be caused by extremely serious corruption within the database or database dictionary files.

STOP 25530 An attempt has been made to run a frame directly, rather than using the $BA command.

STOP 25540 An unsupported ACCEPT or DISPLAY statement call has been made by a Global Cobol module linked into a Speedbase frame.

STOP 25541 An indexing error has occurred within a copy-library specified during a compilation using $SDL. The copy-library is probably corrupt.

STOP 25550 An internal system error has occurred. You should write down the details from the screen and note in detail what you were doing just before the stop code occurred. This information should be forwarded to your software supplier for analysis.

STOP 25551 Speedquery was unable to find the database dictionary (or it was in use) on the same volume as the database itself. Make sure that the dictionary is available to Speedquery.

STOP 25552 An irrecoverable I/O error occurred while reading the database dictionary. This probably means that the dictionary has become corrupted and should be restored from backup. Care should be taken to ensure that the correct version is recovered.

STOP 25553 Speedquery has exhausted the available memory during the execution of a critical part of the query and was unable to recover. You should simplify the query or increase the user partition size to obtain more memory.

STOP 25554 An irrecoverable I/O error has occurred while trying to read or write the query work file. This could be caused by a bad track on the $P subvolume. You should verify the volume before continuing.

STOP 25555 Speedquery was unable to create a work file large enough for the current query. You should allocate more file space on the current $P volume before restarting the query.

STOP 25556 Speedquery has detected a corrupt database index. You should perform a rebuild or a restore to recover the index file.

STOP 25557 Speedquery has detected that the database data files are corrupt. You should perform a rebuild or a restore to recover the database.

STOP 25558 Speedquery has detected that the saved query file requested is corrupt. You should perform a restore to recover the query from backup.

STOP 25559 Speedquery was unable to find the saved query requested.

STOP 25560 Incompatible saved query. Speedquery has detected that the database dictionary has been modified since the query was saved. You must re-develop the query.

STOP 25561 Speedquery was unable to open the print file. This could be caused by a system error, no room on the spool unit, or no room available on the $PR volume. Please check before continuing.

STOP 25562 An irrecoverable I/O error occurred while trying to write to the print file. This could be caused by a system corruption, lack of space on the spool unit, lack of space on the allocated $PR subvolume, or a hardware error..

STOP 25563 The DX file created to hold the database structure is too small. This is caused by an excessively complex database. Contact your software supplier with the details.

STOP 25564 A bad header was found in the dictionary. This may mean that the dictionary has somehow become corrupted and will have to be restored from backup. Care should be taken to ensure that the correct version is recovered.

STOP 25565 Speedquery was unable to read the dictionary. This may mean that the dictionary has somehow become corrupted and will have to be restored from backup. Care should be taken to ensure that the correct version is recovered. This error may also be caused by a bad disk track, in which case you should verify the dictionary.

STOP 25566 An irrecoverable I/O error occurred while opening or closing the DX file. This could be caused by a system corruption, a corrupt DX file, or a bad disk track. Verify the subvolume, then delete the existing DX file. When re-started, Speedquery automatically recreates the DX file.

STOP 25567 An irrecoverable I/O error occurred while reading or writing the DX file. This could be caused by a system corruption, a corrupt structure file, or a bad disk track. You should verify the subvolume then delete the existing DX file. When restarted Speedquery will recreate the DX file.

STOP 25568 Insufficient room for the DX file on the database subvolume. Speedquery needs 4.5 Kbytes of spare space to create the DX file. You should allocate the space then restart Speedquery.

STOP 25569 An I/O error occurred while trying to read the list of available files on the $P subvolume. This is usually caused by a corrupt directory. Please check and rectify before continuing.

STOP 25570 An I/O error occurred while reading or writing the sort work file on the SQW subvolume. Verify the subvolume before continuing.

STOP 25571 An internal system error has occurred while trying to perform a sort. You should write down the details from the screen and note in detail what you were doing just before the STOP code occurred. This information should be forwarded to your software supplier for analysis.

STOP 25580 Speedbase was unable to open the pop-up image swap file required by the last loaded frame. This will occur if there is insufficient free space on logical unit "BAW", or if an I/O error occurs during opening of the swap-file.

STOP 25581 An I/O error has occurred during swap-file processing. This may be due to a disk error.

STOP 25582 Using the BCDB, BXCL or B$XSH routines, an attempt was made to access a database which was not open at the time of the call.

STOP 25583 I/O error in B$XFAM. The UNIX error result code will have been displayed immediately before this stop code. This error will not occur in application programs.

STOP 25584 Unsupported I/O Request in XA$FAM. Internal Systems Software error.

STOP 25586 The next free data-record slot was permanently locked during a WRITE operation. The database should be restored or rebuilt before any further processing is performed.

STOP 25587 Find exceeds 16 IDX Levels. A random index search has exceeded 16 levels of index. The database should be restored or rebuilt before any further processing is performed.

STOP 25588 Dummy high not found in IDB scan. The HIGH-VALUES index terminator was not found during an index scan. The database should be restored or rebuilt before any further processing is performed.

STOP 25589 GVA over/underflow. A WRITE, REWRITE or DELETE statement has caused a GVA field to overflow on one of the associated Master records. GVA over-flow will occur if the computational capacity of the field is exceeded. The database should be restored or rebuilt before any further processing is performed.

STOP 25590 Link fails by permanent lock. A WRITE or REWRITE Instruction failed because one of the master records to be linked was not locked. The DBMS recovery procedure was then unable to recover from this error because the master record to be linked was permanently locked by another partition. The database should be restored or rebuilt before any further processing is performed.

STOP 25591 Link fails by non-existent master. A WRITE or REWRITE instruction failed because one of the master records to be linked was not locked. The DBMS recovery procedure was then unable to recover from this error because the master record specified by the master access key did not exist. The database should be restored or rebuilt before any further processing is performed.

STOP 25592 Index key not found in delete. During the DELETE verb processing, all index key entries referencing the record to be deleted are normally removed. The DELETE verb was unable to find one of these index keys during processing. See Note 1. The database should be restored or rebuilt before any further processing is performed.

STOP 25593 Illegal index key. A WRITE or REWRITE verb has attempted to create an index entry starting with a high-values byte (#FF). This is illegal. The database should be restored or rebuilt before any further processing is performed.

STOP 25594 Free IDB pool exhausted. A WRITE or REWRITE instruction required a free index block to complete an index key addition. The free index pool was however empty. The database should be restored or rebuilt before any further processing is performed.

STOP 25595 Link to unused or deleted IDB. An error has been detected where an index block is incorrectly linked to a deleted or unused IDB. The database should be restored or rebuilt before any further processing is performed.

STOP 25596 Irrecoverable I/O error has taken place during an IDB or data transfer. The database should be restored or rebuilt before any further processing is performed.

STOP 25597 Negative IDB or data record. The IDB/data transfer routine within the DBMS has detected a request for a negative record number. Internal error. The database should be restored or rebuilt before any further processing is performed.

STOP 25598 File condition. A file condition has arisen during a data transfer to or from disk, such as an attempt to transfer data to or from an area out-side the database file extents. The database should be restored or rebuilt before any further processing is performed.

STOP 25599 IDB and record key mismatch. An index entry has been read which does not match the index key held on the record as stored on the database. The database should be restored or rebuilt before any further processing is performed.

Appendix E - The Speedbase Editor

The Speedbase editor utility $SDE, described in this appendix has two main functions. The first is the traditional manipulation of text in a new or pre-existing text file. The other main function is a specialist one and concerns the creation and modification of the window displays used in all Speedbase applications. In other words, it is a text editor and a window editor.

$SDE is a **full screen** editor and allows you to add, delete, insert or modify text in any position on the screen. The cursor is moved around the screen using the cursor keys, up, down, left, right and the tab, back-tab and <RET> keys.

The editor allows the simultaneous manipulation of up to three text files, using the main text buffer and two hold buffers. It provides facilities for you to copy and move blocks of text from one buffer to another, and to delete blocks of text from a buffer. Buffers may be merged into one another and may be copied to an external text file. Text from an external text file may be merged into a buffer. You may swap to any buffer at any time, to inspect or edit its contents, then move on to another buffer, using these buffer facilities to perform powerful and complex editing functions.

The editor allows you to move forwards or backwards through the entire file without restriction and has no requirement, as some editors do, for you to exit and restart the edit session. The complexity associated with switching between modes (e.g. insert mode to overstrike mode) common to most editors, has been eliminated. $SDE is always in overstrike mode, with function keys used to insert characters, lines, or blocks of text anywhere in the file. The editor provides a full backup of the original input file on completion of an edit session to ensure that in the event of a system failure you may always return to the previous version of the text file.

Text searches are performed across the entire buffer from the current position, eliminating the need to move to the top of the file to start a full search. All text searches may be terminated on command, in the case of an incorrectly initiated search for example. The editor also provides a very powerful global replace function with single or multiple replacements, in silent or full display mode, with or without replacement validation. You may recover the last deleted line or block of text, using the undelete and void block functions. A line may be duplicated, split, appended to another line or centred on the screen.

This appendix is divided into seven sections. The introduction describes the major functionality, benefits and restrictions of the editor. Section E.2 describes how to run the utility and enter or exit its amendment phase. Section E.3 describes the direct commands of the editor in detail and Section E.4 the executable commands. Sections E.5 and E.6 describe window editing in detail and Section E.7 lists the error and warning messages which may appear during the amendment phase.

E.1 Editor Facilities

The editor makes extensive use of the keyboard's cursor and function keys. These are set up using the Speedbase Presentation Manager customisation utility. At any point in the operation of the editor you may see which keys are available to carry out editing commands by keying <HLP> to display the Speedbase help window. The commands recognised by the editor fall naturally into two main groups, known as direct and executable commands.

Direct commands are carried out directly a key is pressed. To see which keys are assigned to each direct command simply key <HLP> while editing. A standard Speedbase help window is displayed listing the commands available and the associated keytop names. Figure E.1a shows the screen that results from keying <HLP> while editing the sample program S.V3DEMO, including typical key-top names:

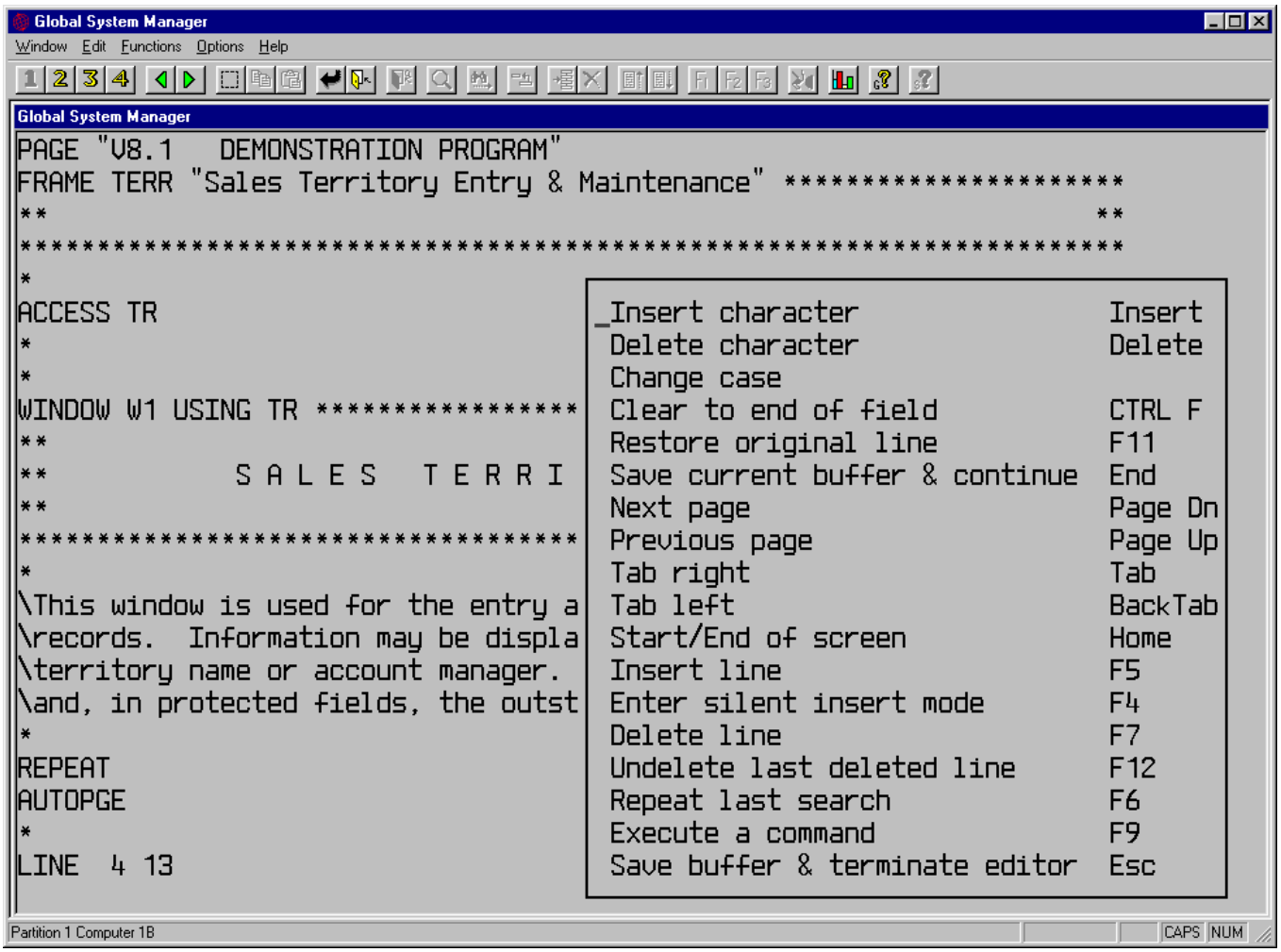


Figure E.1a - Direct Command Help Window

If you use the key labelled Execute a command, F9 in the above example, the editor enters executable command mode. The cursor moves to the baseline and the command prompt is displayed:

```
C:
```

You may reply with any of the executable commands listed in Table E.4a. A response of <RET> or <BCK> returns the cursor to its original screen position.

E.1.1 Buffers

The editor has three buffers for the manipulation of text. Buffer 0 is the main text buffer into which the input file is read on entry and must be the buffer in use at the end of the edit session. Otherwise the buffers are identical and you may use any of the direct or executable commands, except commands A and E, in any buffer. Commands allow you to move, delete, copy, save, read and merge text into buffers and external files.

E.1.2 Backup

The editor keeps a backup of the file you are editing on the same direct access volume as the input file. At the end of the editing session the new version of the text file is given the name of the input file which itself has the prefix of its file-id changed to B. For example, if you edit the text file S.V3DEMO, at the end of the edit session the modified file is named S.V3DEMO and the old version of it is named B.V3DEMO.

E.1.3 Automatic Insert

As you add text at the end of file, EOF for short, the editor extends the file automatically. This eliminates the need for you to insert new lines in order to extend the file.

E.1.4 Restrictions

The editor assumes you are using a display terminal of 24 lines by 79 characters. If it finds a line of greater than 79 characters it will automatically truncate it. Note that the Speedbase compiler compiles 79 character lines but that the Global Cobol compiler truncates all lines longer than 72 characters. The editor assumes that the terminal you are using has a TAP set up using the Speedbase customisation utility.

You may only increase the size of your text file by 32K characters per editing session. When you get to within 2% of this limit the editor issues the following warning:

```
Less than 2% available space - Please save
```

You should then use one of the instructions:

Save text	Performs the same function as command E, end edit, but you remain in the same edit window.
New file	As above but allows you to edit a new input file.
End edit	See Section E.3.18. and Section E.4.9.
Abandon	See Section E.4.5.

If you fail to take the appropriate action, the editor will terminate the edit session once the available space has been exhausted with the following message:

```
Edit output file exhausted
```

If you receive the above message during an important editing session, you could attempt to recover the editor work file. The editor work file is named as follows:

$SDE *nnb*

where *nn* is the number of the user partition in which you are running the editor and *b* is the buffer number. It occupies the same volume as the source file, and contains that part of the file which has been correctly updated. You may use the $INSPECT command to merge this with the remainder of the original source file, so that you can continue working from the point at which the file space became exhausted. You should be aware of the difficulty of recovering the work file and take heed of the warning message when issued.

E.2 Edit Phase

To run the editor use the menu entry you have set up or key $SDE At the option prompt. The initial screen is displayed and you enter the filename and unit:

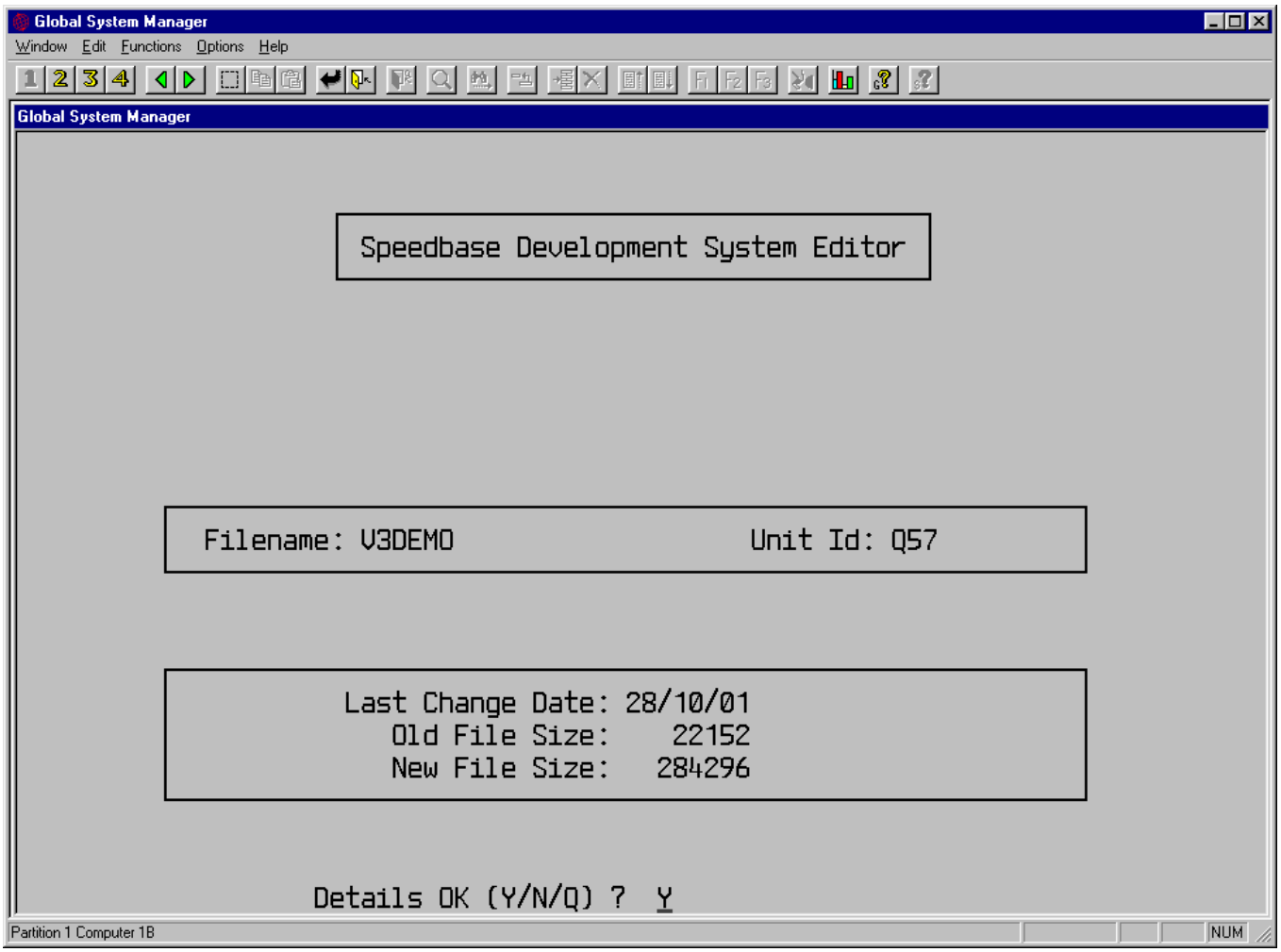


Figure E.2a - The Initial Editor Screen

In response to the filename prompt you should enter the name of the required input file. Unless you explicitly supply a prefix when you key the file-id, the editor assumes you are working with a program source file and appends the S prefix by default. If, for example you reply V3DEMO, the editor assumes you wish to create or modify source file S.V3DEMO. Note that you may not edit a file with a B prefix because this is reserved by the editor for backup versions of edited files.

You then reply to the Unit Id prompt with the unit holding the input file. Logical or physical unit-id's may be entered.

E.2.1 To Quit

To quit at this stage and return control to the monitor simply key <BCK> to either the Filename or Unit Id prompt.

E.2.2 Creating a New File

When the input file you specify does not exist you are prompted to create a new file or reply N to return to the filename prompt. If a new file is required, the editor creates it with a size of 32K

characters using the name specified, then enters the amendment phase. A blank screen is displayed with the cursor in the top left hand corner and the message "Reached EOF" displayed at the baseline. You may start entering text anywhere on the screen. Note that when the cursor is at EOF the editor automatically appends lines to the file as you enter text.

E.2.3 Updating an Existing File

When the file specified on the initial screen is found, is valid and is not in use, details are displayed of the old and new file size and the date the file was last changed. At the baseline the following message is displayed:

```
Details OK (Y/N/Q) ?
```

Reply Y or <RET> to accept the details and enter the amendment phase, N to go back to the filename prompt or Q to exit. If the details are accepted, the editor displays the first screen of text from the input file. If the file does not contain a full screen of text the "Reached EOF" message is displayed. The cursor is placed in the top left hand corner of the screen.

E.2.4 The Amendment Phase

After input of the required file details, the text screen is displayed. The first 23 lines of this screen are used as a window into the input file, displaying those lines that can currently be edited. The baseline, line 24, is a multiple purpose line used for the entry of editor commands and the display of prompts, information, warning and error messages.

At any moment, the only text lines that may be changed by editing instructions are those that appear on the current screen, referred to as a page. However the current screen may be moved forward or backward within the file to make other lines available for modification.

E.2.5 Quitting the Amendment Phase

Two special editing instructions cause the editor to quit the amendment phase. Command E, end edit, is used when you have satisfactorily completed an edit session. It causes the remainder of the input file, if any, to be copied to the output file, and the input file, if present, to be renamed with prefix B as the new backup file. Any previous backup file is deleted.

The output file is then given the same file-id as the original input file. For example, suppose you reply V3DEMO in response to the filename prompt in order to modify source file S.V3DEMO. Then, as a result of keying command E, the following happens:

- If a file named B.V3DEMO already exists on the volume containing S.V3DEMO it is deleted
- Then the existing S.V3DEMO is renamed B.V3DEMO
- Next, the new file containing the result of the edit session is itself renamed S.V3DEMO

Note that if on issuing this command there are active buffers (i.e. buffers that are in use) the editor issues the following warning:

```
Buffers active - Are you sure (Y/N) ?
```

If you reply Y to this prompt the editing session is terminated. Any other reply causes the editor to ignore the request and return to the amendment phase. Command A, abandon edit, may be used if you decide not to keep the changes made during the current edit session and wish to

retain the original input file. The editor does not delete the original backup file or rename the input file as it does for command E. You are prompted:

```
Are you sure (Y/N) ?
```

unless there are buffers active (i.e. in use) in which case you are prompted:

```
Buffers active - Are you sure (Y/N) ?
```

If you reply Y to these prompts the session is abandoned. Any other reply causes the editor to ignore the request and return to the amendment phase.

E.3 Direct Commands

The editor recognises two types of commands, direct and executable commands, see Tables E.3a and E.4a. Direct commands are executed using predefined function keys on the keyboard. The key-tops shown in Table E.3a are typical examples, yours will probably be different and are displayed when you key <HLP>.

Direct command	Example key-top	Appendix section
Insert character	Insert	E.3.1
Delete character	Delete	E.3.2
Change case	Control-C	E.3.3
Clear to end of line	Control-F	E.3.4
Restore original line	Control-R	E.3.5
Save current buffer & continue	End	E.3.6
Next page	Page Down	E.3.7
Previous page	Page Up	E.3.8
Tab right	Tab	E.3.9
Tab left	^Tab	E.3.10
Start/End of screen	Home	E.3.11
Insert line	F5	E.3.12
Enter silent insert mode	F4	E.3.13
Delete line	F7	E.3.14
Undelete last deleted line	Control-D	E.3.15
Repeat last search	F6	E.3.16
Execute a command	F9	E.3.17
Save buffer & terminate editor	Escape	E.3.18

Table E.3a - $SDE Direct Commands

E.3.1 Insert character

Inserts a blank space at the current cursor position, which may be anywhere on the screen. On insert, the text string to the right of the cursor is shifted across by one character without affecting the rest of the line. A text string is defined to be a string of text delimited by two spaces. Only the current line is affected.

E.3.2 Delete character

Deletes the character at the current cursor position.

E.3.3 Change case

Lower case a - z are changed to upper case A - Z and vice versa, at the current cursor position.

E.3.4 Clear to end of line

Clears the line from the current cursor position to the end of the line, leaving the cursor in the same position.

E.3.5 Restore original line

If having edited a line you would prefer to restore its original contents use this command before moving from the line or executing a command.

E.3.6 Save current buffer & continue

This command causes the contents of the current buffer to be saved on disk. Its use prevents the loss of editing work in the event of a computer failure.

E.3.7 Next page

Fetches the next screen of 23 lines and displays it. If the end of the input file is reached, the last page of the file is displayed with the message "Reached EOF" at the baseline.

E.3.8 Previous page

Fetches the previous screen and displays it. If the start of the input file is reached, the first page of the file is displayed and the message "Reached TOF" is displayed at the baseline.

E.3.9 Tab right

Move the cursor to the next tab position. Tab positions are at columns 1, 9, 17 ... etc. If the cursor reaches the end of line, a warning is sounded and the cursor left unchanged. If line wrap around has been switched on, using command J, the cursor is moved to column 1, the first column of the current line.

E.3.10 Tab left

Move the cursor to the previous tab position. If the cursor reaches the start of line, a warning is sounded and the cursor position left unchanged. If line wrap around has been switched on, using command J, the cursor will be moved to the last tab position of the current line.

E.3.11 Start/End of screen

Moves the cursor to column 79 of line 23 of the current screen or to column 1 of line 1 if already there. This command therefore causes the cursor to toggle between the bottom right-hand and top left-hand corners of the screen.

E.3.12 Insert line

Inserts a blank line before the current line and makes the inserted line the new current line. The new line may now be edited as normal.

E.3.13 Enter silent insert mode

The screen is cleared from the current line to the end of the screen so that you can insert text without the existing text scrolling down. To exit silent insert mode, key <ABO> and the following text is redisplayed.

E.3.14 Delete line

Deletes the current line. The rest of the screen is scrolled to take up the blank space.

E.3.15 Undelete last deleted line

Inserts before the current line a copy of the last line deleted. If the cursor has moved since the deletion, the line is inserted before the new current line. This command may therefore be used, in conjunction with the delete command, to move a line from one position to another or to make several copies of a line.

E.3.16 Repeat last search

Repeats the last search performed, starting at the current column plus one. If no previous search was performed by either command F, find string, or command R, replace string, a warning is sounded and the command ignored.

E.3.17 Execute a command

Causes the editor to enter command mode. The command prompt, C: is displayed at the baseline. Reply with one of the commands listed in Table E.4a or key <RET> to cancel the command prompt.

E.3.18 Save buffer & terminate editor

The contents of the buffer are saved on disk and the editor terminates. The initial editor screen is redisplayed as in Figure E.2a.

E.4 Executable Commands

Command	Description	Appendix section
!	Display ruler	E.4.1
+	Move forward *nnnnn* lines	E.4.2
-	Move backward *nnnnn* lines	E.4.3
.	Display information line	E.4.4
A	Abandon edit session	E.4.5
B	Swap to buffer *N*	E.4.6
C	Centre the current line	E.4.7
D	Delete block text	E.4.8
E	Save & end current edit	E.4.9
F	Find string	E.4.10
G	Go to the end of line	E.4.11
H	Copy text to buffer *N*	E.4.12
I	Enter silent insert mode	E.4.13
J	Switch wrap around on/off	E.4.14
K	Case insensitive Find String	E.4.31
L	Goto line *nnnnn*	E.4.15

M	Move text to buffer *N*	E.4.16
N	Save current file & fetch new	E.4.17
O	Output buffer *N*	E.4.18
P	Goto first (T) or last (B) page	E.4.19
Q	Enter window generation mode	E.4.20
R	Global replace	E.4.21
S	Display current buffer status	E.4.22
T	Split line into two	E.4.23
U	Define user keys F1, F2, F3	E.4.24
V	Void the last delete block	E.4.25
W	Duplicate previous line	E.4.26
X	Merge new file into text	E.4.27
Y	Rename current buffer file	E.4.28
Z	Flush buffer *N*	E.4.29
&	Append characters prior to cursor to previous line	E.4.30

Table E.4a - Executable Commands

To execute one of the commands listed above, key the direct command "Execute a command" and reply to the baseline prompt:

```
C:
```

with the appropriate command. Certain commands prompt for further details. Some require a number (e.g. command L; List Line) and others require a text string (e.g. command R; Global Replace). For multiple character responses, key <RET> to terminate the input, but for single character responses the editor automatically starts execution of the command as soon as it is entered (e.g. command J; Wrap Around on/off).

E.4.1 Command !, Display ruler

The ruler is displayed at the baseline to show column numbers.

E.4.2 Command +, Move forward *nnnnn* lines

The display window is moved forward *nnnnn* lines. For example, command +1 causes the display to move up one line so that the second line becomes the top line etc.

E.4.3 Command -, Move backward *nnnnn* lines

The display window is moved backward *nnnnn* lines. For example, command -1 causes the display to move down one line so that the top line becomes the second line etc.

E.4.4 Command ., Display information line

Displays on the baseline the current line number, the number of characters in the file and the space available for expansion, in characters.

E.4.5 Command A, Abandon edit session

Providing you are in buffer 0, the editor displays on the baseline the prompt:

```
Are you sure (Y/N) ?
```

unless there are buffers in use, in which case the prompt is:

```
Buffers active - Are you sure (Y/N) ?
```

In case you keyed command A in error. You must reply Y to complete the abandon operation. The editor terminates and the initial screen is displayed as in Figure E.2a. Note that none of the changes made in the editing session are saved when you abandon the session. If you reply with any other character the command is ignored and the editor returns to the window for further editing.

E.4.6 Command B, Swap to buffer *N*

Swaps the editing session to buffer *N*. You may then proceed to edit the text in this buffer. The buffer is displayed at the point at which you last left it.

E.4.7 Command C, Centre the current line

The current line is moved to line 12 of the screen and redisplayed with the surrounding lines. If the editor is unable to centre the line, due to being near TOF or EOF, it attempts to centre the line as close as possible and displays the message:

```
Reached EOF
```

or:

```
Reached TOF
```

E.4.8 Command D, Delete block text

The following message is displayed at the baseline:

```
Delete Block mode
```

Move the cursor to the start line and key <UF1>, usually F1. The start line number is displayed at the baseline. Move the cursor to the last line to be deleted and key <UF1> again. The end line number is displayed, the text block deleted and the baseline message "Delete Block - Finished" is displayed. If the marked end line precedes the start line the baseline message "End line before start line - Delete Block - Aborted" is displayed and the command ignored. The delete block operation may be abandoned by keying <BCK> before marking the end line, and its effect reversed by use of command V, Void the last delete block.

E.4.9 Command E, Save & End current edit

This command may only be used while in buffer 0, the main text buffer. The contents of the buffer are saved on disk and the session is terminated. If other buffers are in use, the prompt:

```
Buffers active - Are you sure (Y/N) ?
```

is displayed in case you keyed command E in error. Reply Y to complete the end edit operation. If you reply with any other character the command is ignored and the editor returns to the window for further editing.

At the end of the session the input source file, if any, becomes the backup file, and the new file becomes the current source file. The initial screen is displayed as shown in Figure E.2a.

E.4.10 Command F, Find string

You are prompted for the required search string, terminated with <RET>. The length of the string is restricted to a maximum of 50 characters. A null response to the string prompt causes

the editor to repeat a search for the last entered string. If no previous search has been performed, the command is ignored.

The baseline message "Searching" is displayed and the search commences at the current cursor position. The search cycles through the entire file. On reaching EOF it continues from the beginning of the file until reaching the original start, before deciding that the string cannot be found.

If the string is found on the current screen, the cursor is moved to the start of the string. If found on a new page, a new screen is displayed with the line containing the string centred, and the cursor placed at the start of the string. If the string is not found, the baseline message:

```
String not found
```

is displayed and editing recommences at the original cursor position. The search may be terminated prematurely by keying any key.

E.4.11 Command G, Go to the end of line

The cursor moves to the last non-blank character on the line.

E.4.12 Command H, Copy text to buffer *N*

Reply with the required buffer number, 0, 1 or 2, to the baseline prompt. The default is buffer 1. Mark the start line by moving the cursor to it and keying <UF1>, usually F1. The line number of the start line is displayed at the baseline. Move the cursor to the required end line and key <UF1> again. The line number of the end line is displayed, the block of text is copied to the buffer specified and the baseline message "Hold Block - Finished" is displayed. If the marked end line precedes the start line the baseline message "End line before start line - Hold Block - Aborted" is displayed and the command ignored. If the buffer already contains text, the editor displays the baseline message:

```
Buffer in use - Text will be appended
```

and any text copied is appended to the existing text in the output buffer. If you key <BCK> before marking the end line, the copy text operation is abandoned and the baseline message "Hold - aborted" is displayed. If you attempt to copy text to the current buffer the baseline message "Invalid - Currently in buffer *N*" is displayed and the command ignored.

E.4.13 Command I, Enter silent insert mode

The screen is cleared from and including the current line to the end of the screen. You may commence editing at the start of the cleared area. The use of this command avoids the necessity manually to insert blank lines for text to be inserted. To terminate silent insert mode, key <ABO> and the screen is redisplayed from the current line onwards.

E.4.14 Command J, Switch wrap around ON/OFF

Allows the cursor to move past column 79, and move to the left of column 1. Entry of this command toggles the switch on/off. The default on entry to the editor is wrap around off and the cursor is stopped from moving right of column 79, and left of column 1. When on, moving to the right of column 79 places the cursor at the start of the next line and moving to the left of column 1 places the cursor at the end of the previous line.

E.4.15 Command L, Goto line *nnnnn*

Reply with the required line number to the baseline prompt. If the required line is found on the current screen, the cursor is moved to that line. If found on a new screen, the new screen is displayed with the required line centred. If EOF is reached before finding the required line, the last page of the file is displayed, and the baseline message "Reached EOF" is displayed. The cursor is then placed on the last line of the file.

E.4.16 Command M, Move text to buffer *N*

Reply with the required buffer number, 0, 1 or 2, to the baseline prompt. The default is buffer 1. Mark the start line by moving the cursor to it and keying <UF1>, usually F1. The line number of the start line is displayed at the baseline. Move the cursor to the required end line and key <UF1> again. The line number of the end line is displayed, the block of text is moved to the buffer specified and the baseline message "Move Block - Finished" is displayed. If the marked end line precedes the start line the baseline message "End line before start line - Move Block - Aborted" is displayed and the command ignored. If the buffer already contains text, the editor displays the baseline message:

```
Buffer in use - Text will be appended
```

and any text copied is appended to the existing text in the output buffer. On completion of the move the screen is redisplayed with the block removed, with the cursor on the line before the block. If you key <BCK> before marking the end line, the move text operation is abandoned and the baseline message "Move - aborted" is displayed. If you attempt to move text to the current buffer the baseline message "Invalid - Currently in buffer *N*" is displayed and the command ignored.

E.4.17 Command N, Save current file & fetch new

The baseline message:

```
Saving buffer
```

is displayed. Once the file has been saved, reply to the baseline prompt:

```
New filename:
```

with the name of the new file. If it does not exist on the disk volume holding the input file the editor prompts you to create it. The editor then displays the first page of the file which may itself be edited.

E.4.18 Command O, Output buffer *N*

Reply with the required buffer number, 0, 1 or 2, to the baseline prompt. The default is buffer 1. The contents of the required buffer are inserted before the current line. The screen is redisplayed to show the new contents. If the buffer is empty, the baseline message "Buffer empty" is displayed, and the command ignored. If you attempt to output text from the current buffer the baseline message "Invalid - Currently in buffer *N*" is displayed and the command ignored.

E.4.19 Command P, Go to first or last page

Reply to the baseline prompt with T for the top page, or B for the bottom page.

E.4.20 Command Q, Enter window generation mode

Used to create and modify the windows used extensively in Speedbase applications. If the cursor is on the WINDOW statement line when command Q is keyed, a representation of the window defined by the WINDOW construct is displayed. If the WINDOW construct contains syntax errors the editor places the cursor on the line in error and displays a baseline error message - see Appendix B for details. If the cursor is on any other line you are prompted for details of the new window - see Section E.6.

E.4.21 Command R, Global replace

Reply to the baseline prompt with the search string. Then reply to the prompt "Wth:" with the replacement string, terminated by <RET>. The length of the string is restricted to a maximum of 50 characters. The baseline prompt:

```
All Occurrences (Y/N)
```

is displayed next. If you reply N, only the first occurrence is replaced. Otherwise the editor displays the baseline prompt:

```
Display On (Y/N)
```

If you reply N, the editor replaces all occurrences of the search string found without displaying them as they are replaced. Otherwise the editor displays the baseline prompt:

```
Validate (Y/N)
```

If you reply N, the editor replaces all occurrences of the search string found. Otherwise the editor pauses at each occurrence found, and displays the baseline prompt "Validate (Y/N/Q)". Reply Q to terminate the replace command. Reply N to skip the occurrence found and search for the next occurrence. Reply Y to replace the text string with the replacement string and search for the next occurrence.

E.4.22 Command S, Display current buffer status

The baseline display shows the name and unit-id of the file being edited and the number, 0, 1 or 2, of the current buffer in use.

E.4.23 Command T, Split line into two

All characters from the current cursor position to the end of the line are moved to a new line before the next line.

E.4.24 Command U, Define user keys, <UF1>, <UF2>, <UF3>

The three user function keys, <UF1>, <UF2>, <UF3>, may be assigned to any executable command. For example, to assign the key <UF1>, usually F1, to command Q, reply 1 and then Q to the baseline prompts. Note that key assignments remain in force only until the end of the editing session. They apply to all buffers.

E.4.25 Command V, Void the last delete block

The last text block deleted is inserted before the current line. It may be restored any number of times, anywhere in the buffer, not necessarily in its original position. If no text block has been deleted during this edit session the editor displays the baseline message "Buffer empty" and the command is ignored.

E.4.26 Command W, Duplicate previous line

A copy of the line preceding the current line is inserted before it. The duplicated line becomes the current line.

E.4.27 Command X, Merge new file into text

Reply to the baseline prompts:

```
Merge filename:  Unit Id:
```

with the filename and unit-id of the file holding the text to be merged. The editor reads the entire contents of the merge file and inserts the text before the current line. The screen is then redisplayed with the cursor at the first line of the merged text. If the size of the merge file exceeds certain size constraints, the baseline message:

```
No room to read file
```

is displayed and the command is ignored.

E.4.28 Command Y, Rename current buffer file

Reply to the baseline prompt:

```
New filename:
```

with the filename you wish to use for the output file of the editing session. The original input file to the editor remains unchanged on exit from the editing session instead of being renamed as the backup file.

E.4.29 Command Z - Flush buffer *N*

Reply to the baseline prompt with the number, 0, 1 or 2, of the buffer you wish to flush. Reply to the prompt:

```
Flush buffer N - Are you sure (Y/N) ?
```

with Y to cause the buffer to be flushed, or N to abort the command. On completion the editor displays the message "Buffer flushed" and clears the screen if it is the current buffer. If the buffer is already empty the editor displays the baseline message "Buffer empty" and the command is ignored.

E.4.30 Command &, Append characters prior to cursor to previous line

The characters to the left of the current cursor position are moved to the end of the previous line. Note that characters to the right of column 79 after the move are lost.

E.4.31 Command K, Case insensitive find operation

This operation allows you to perform a case insensitive search. The operation is identical to the Find command (See section E.4.10), excepting that upper and lower case characters are treated identically during the search.

E.5 Regenerating a Window

This section of the appendix deals with the modification of a window described by a WINDOW construct in your source file. If you key command Q when the cursor is on a WINDOW statement line, the editor enters window generation mode and displays a representation, or template, of the window based on the statements in the window construct. For example, if you

key command Q when the cursor is on the line WINDOW W1 of FRAME CUST of the file S.V3DEMO, a template of the customer window of the sample application is displayed:

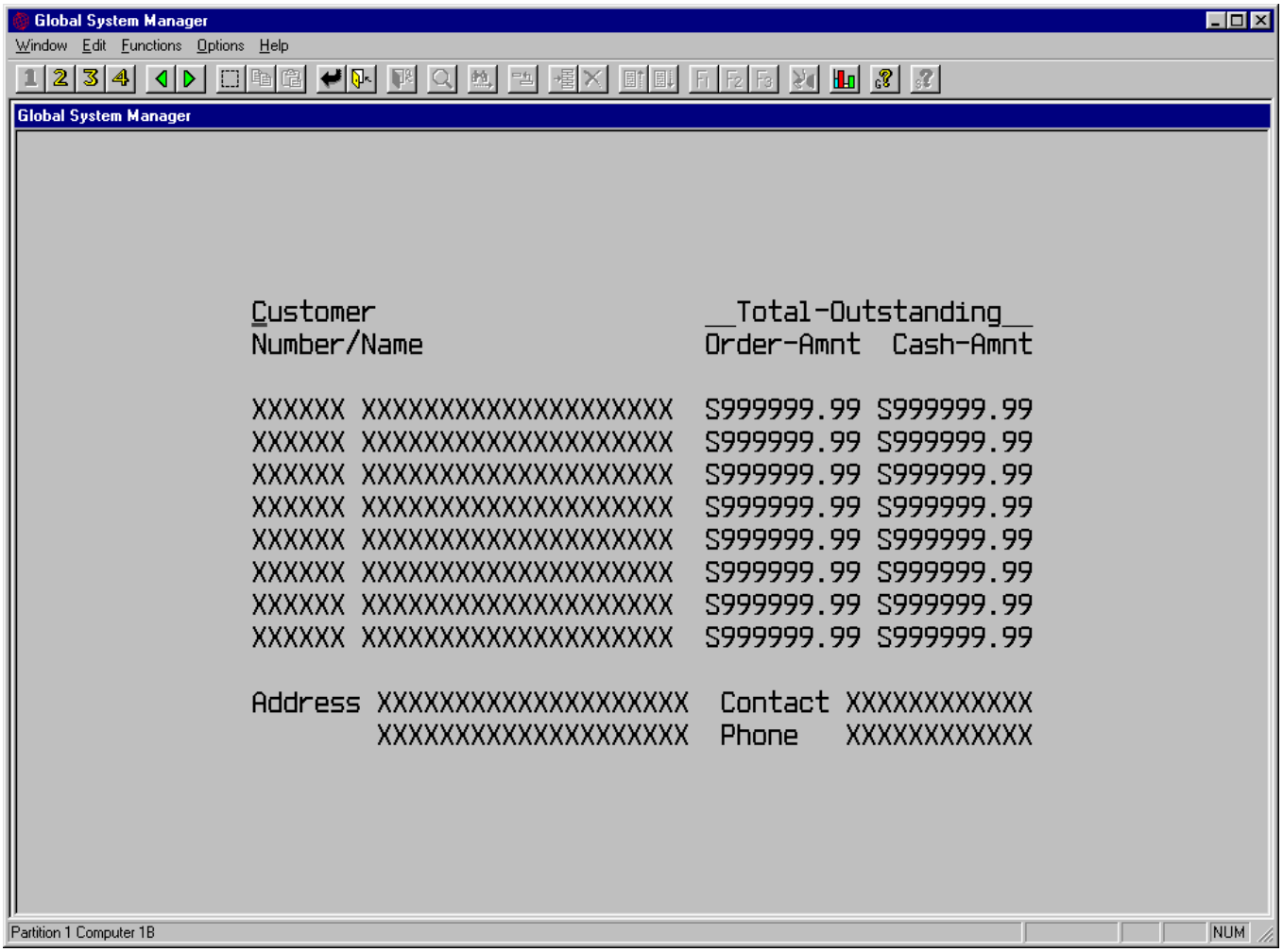


Figure E.5a - The Customer Window Template

To see what functions are available, key <HLP> to display the usual Speedbase help window. Table E.5b lists them and gives an example key-top for each. The editor keeps track of each field in the template separately. If you move a field so that it partially or completely overlaps another, the editor retains information about both. If the fields are subsequently moved so that they do not overlap, they are correctly displayed. It is helpful to refresh the display in these circumstances, which you can do by using the command "Draw boxes/lines" for example.

Note that the first scrolled line in the template of a scrolled window is highlighted. The subsequent scrolled lines are for display purposes only and cannot be manipulated. If the WINDOW construct contains syntax errors the editor places the cursor on the line in error and displays a baseline error message - see Appendix B for details. Syntax errors must be corrected before the window template can be displayed.

Window mode command	**Example key-top**	**Appendix section**
Start/End of screen	Home	E.5.1

Move to next field/text item	Tab	E.5.2
Move to previous field/text item	^Tab	E.5.3
Shift current item right	Insert	E.5.4
Shift current item left	Delete	E.5.5
Insert line	F5	E.5.6
Delete line	F7	E.5.7
Move current item	Control-D	E.5.8
Delete current item	F4	E.5.9
Modify/Enter text items	F2	E.5.10
Draw boxes/lines	F1	E.5.11
Modify scroll details	F3	E.5.12
Generate window source	End	E.5.13
Move entire window	Page Down	E.5.14
Select more fields	Escape	E.5.15
Insert selected fields	F6	E.5.16
Abandon window & return to editor	F9	E.5.17

Table E.5b - Window Mode Commands

E.5.1 Start/End of screen

Moves the cursor to column 1 of line 1 of the current screen or to column 79 of line 23 if already there. This command therefore causes the cursor to toggle between the bottom right-hand and top left-hand corners of the screen.

E.5.2 Move to next field/text item

The cursor moves to the beginning of the next field. The editor displays a baseline message showing the line and column position of the field, "TEXT" if it is a text field, the field name, picture and "SCR" if a scrolled data field and "NSC" if a non- scrolled data field.

E.5.3 Move to previous field/text item

The cursor moves to the beginning of the previous field. The editor displays a baseline message showing the line and column position of the field, "TEXT" if it is a text field, the field name, picture and "SCR" if a scrolled data field and "NSC" if a non-scrolled data field.

E.5.4 Shift current item right

The text or data item at or following the current cursor position is shifted one place to the right. Note that the position of the item is defined by its first character. If this command is keyed with the cursor positioned for example at the second character of an item, the next item is moved. After the move the cursor is positioned on the first character of the moved item.

E.5.5 Shift current item left

The text or data item at or following the current cursor position is shifted one place to the left. Note that the position of the item is defined by its first character. If this command is keyed with the cursor positioned for example at the second character of an item, the next item is moved. After the move the cursor is positioned on the first character of the moved item.

E.5.6 Insert line

Inserts a blank line at the current line. If the cursor is on or above the first line of the window template the entire window moves down one line. If the insert would move any part of the

window off the screen the baseline message "Outside window boundary" is displayed and the command ignored. Note that the editor reserves a blank line above and below the window template for the box lines.

E.5.7 Delete line

Deletes the current line. If the line contains a text or data field the editor displays the baseline message "Non blank line - Can't delete" and the command is ignored.

E.5.8 Move current item

The editor displays a baseline message showing the line and column position of the field, "TEXT" if it is a text field, the field name, picture and "SCR" if a scrolled data field and "NSC" if a non-scrolled data field. Move the cursor to the new field position and key "Move current item" again. Reply to the baseline prompt:

```
Scrolled or Non-scrolled ?
```

with S for a scrolled item or N for a non-scrolled item. The default is S for a previously scrolled field and N for a previously non-scrolled field.

E.5.9 Delete current item

Deletes the item beginning at the current cursor position. If no item begins at the cursor position the baseline message "No item at this location" is displayed and the command is ignored.

E.5.10 Modify/Enter text items

If this command is keyed with the cursor placed anywhere within a text item, the editor highlights the item and extends it to the beginning of the next field. It displays a baseline message showing the line and column position of the field and "TEXT". Enter the new text and terminate it by keying <RET>.

If this command is keyed with the cursor on an unoccupied character position the editor highlights the available space, up to sixty characters. Enter the text and terminate it by keying <RET>.

Having entered the new or modified text, reply to the baseline prompt:

```
Scrolled or Non-scrolled ?
```

with S for a scrolled item or N for a non-scrolled item. The default is S for a previously scrolled field and N for a previously non-scrolled field. If you attempt to enter or modify text at a position occupied by a data item the editor displays the baseline message "No room for text field" and the command is ignored.

E.5.11 Draw boxes/lines

The window template is redisplayed with a box and the appropriate window background. Unless the template is positioned at the edge of the screen the editor leaves a one-character border between the template and the box. If there is insufficient space for this border it generates a window with the SBOX statement.

Reply to the baseline prompt:

```
Line:
```

with the line number of a line to be drawn in the box. Key <RET> to the default of line zero when you wish to return to the template display. Note that the top line of the screen is line 1 and that the editor calculates the correct line number for the LINE statement taking into account the box position. To erase a line, overtype its line number with zero.

E.5.12 Modify scroll details

The window template is replaced by the prompt:

```
Generate scrolled window:
```

If you reply Y, the editor displays the prompts:

```
Scroll   by   spilt   offset
```

The defaults for these parameters are the existing values in the template. Having replied to these prompts, or having replied N to the scrolled window prompt, the editor redisplays the template using the new parameters.

E.5.13 Generate window source

The editor generates the appropriate WINDOW construct source code from the window template as modified. The current text buffer is redisplayed with the new WINDOW construct in place of the old and the new WINDOW statement line centred on the screen.

E.5.14 Move entire window

The baseline message "Move cursor to top left corner for window - key <BASE>" is displayed. Move the cursor and key "Move entire window" again. The window template is redisplayed at the new position. If the move would place any of the window off the screen the baseline message "Window will not fit on screen - Aborted" is displayed and the command is ignored.

E.5.15 Select more fields

The window template is replaced by the prompt:

```
Dictionary:      Unit:
```

Reply with the name and unit-id of the Speedbase data dictionary from which you wish to select more fields for the window. The editor displays a window listing the records in the dictionary from which fields may be selected. The process of selecting fields is described in detail in section E.6.

E.5.16 Insert selected fields

The editor displays a baseline message "Insert - " then the name of the first field available and its description. Reply Y to insert the field, or N to display the next available. The fields available for insertion must previously have been selected using the command "Select more fields". Reply to the baseline prompt:

```
Scrolled on Non-scrolled:
```

with S for a scrolled item or N for a non-scrolled item. The default is S if the field was selected as a scrolled field and N if it was selected as a non-scrolled field. The field is inserted at the position occupied by the cursor when the command was keyed. If this would place part of the

field outside the screen area the baseline message "Outside window boundary" is displayed and the command is ignored. If no fields are available the baseline message "No more selected fields" is displayed and the command ignored.

E.5.17 Abandon window & return to editor

The current edit buffer is redisplayed with the cursor located where it was when you keyed command Q. No window source code has been generated and the text is as it was when you keyed command Q.

E.6 Generating New Windows

This section of the appendix deals with the creation of a new window and the addition to your source file of its WINDOW construct. The use of command Q puts the editor into window generation mode. If you do so when the cursor is not on a WINDOW statement, the editor displays the prompt:

```
Generate scrolled window:
```

If you reply Y, the editor displays the prompts:

```
Scroll   by   spilt   offset
```

The default for the number of records is 8 and for the latter three parameters is 1. Having replied to these prompts, or having replied N to the scrolled window prompt, the editor prompts for the name and unit of the dictionary from which you wish to extract details.

E.6.1 Selecting a Record

The editor then displays details of the records available:

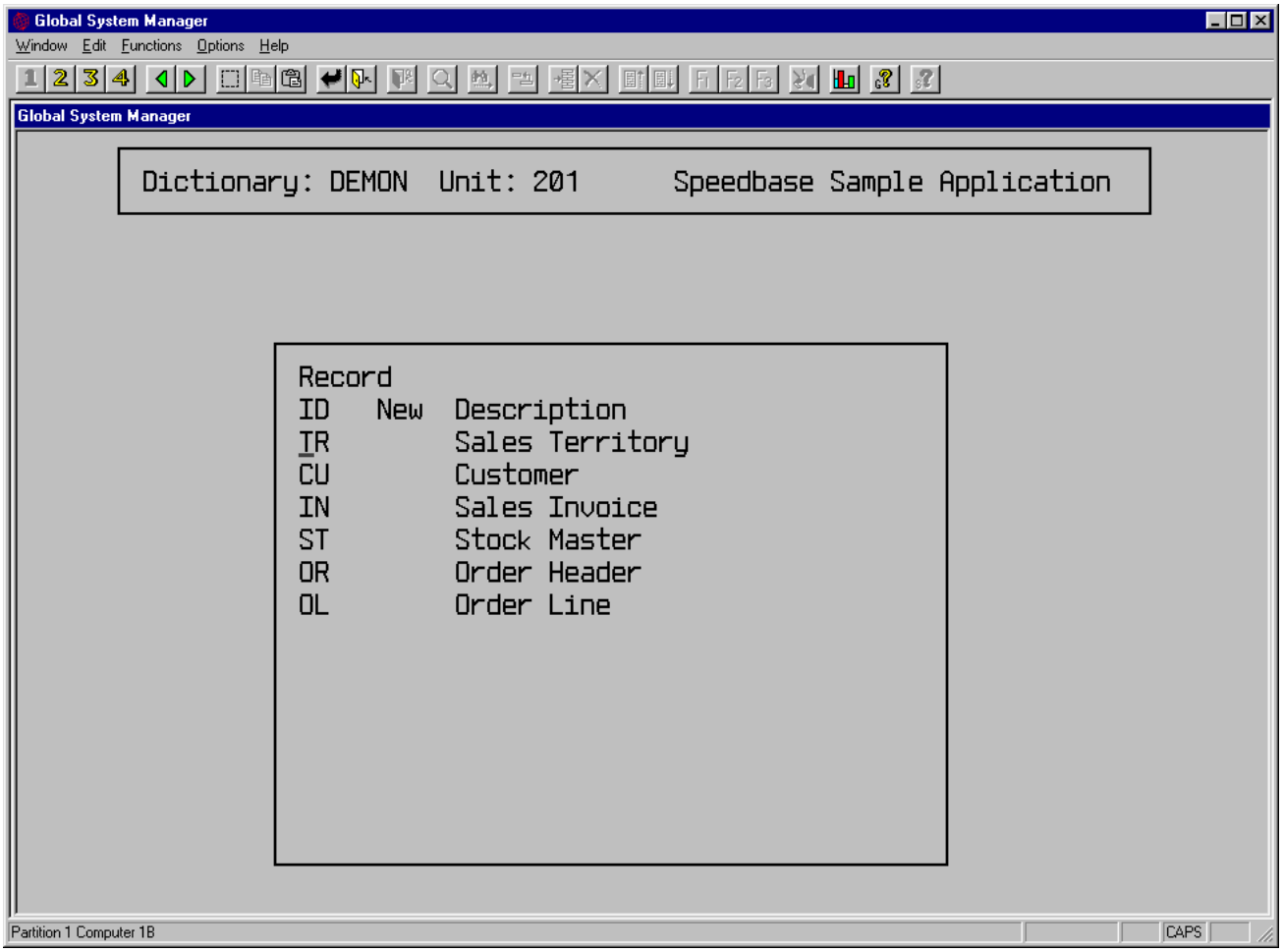


Figure E.6.1a - Record Selection Window

If, for example, you select record CU, and give it the new ID C1, the list of fields available for selection is displayed:

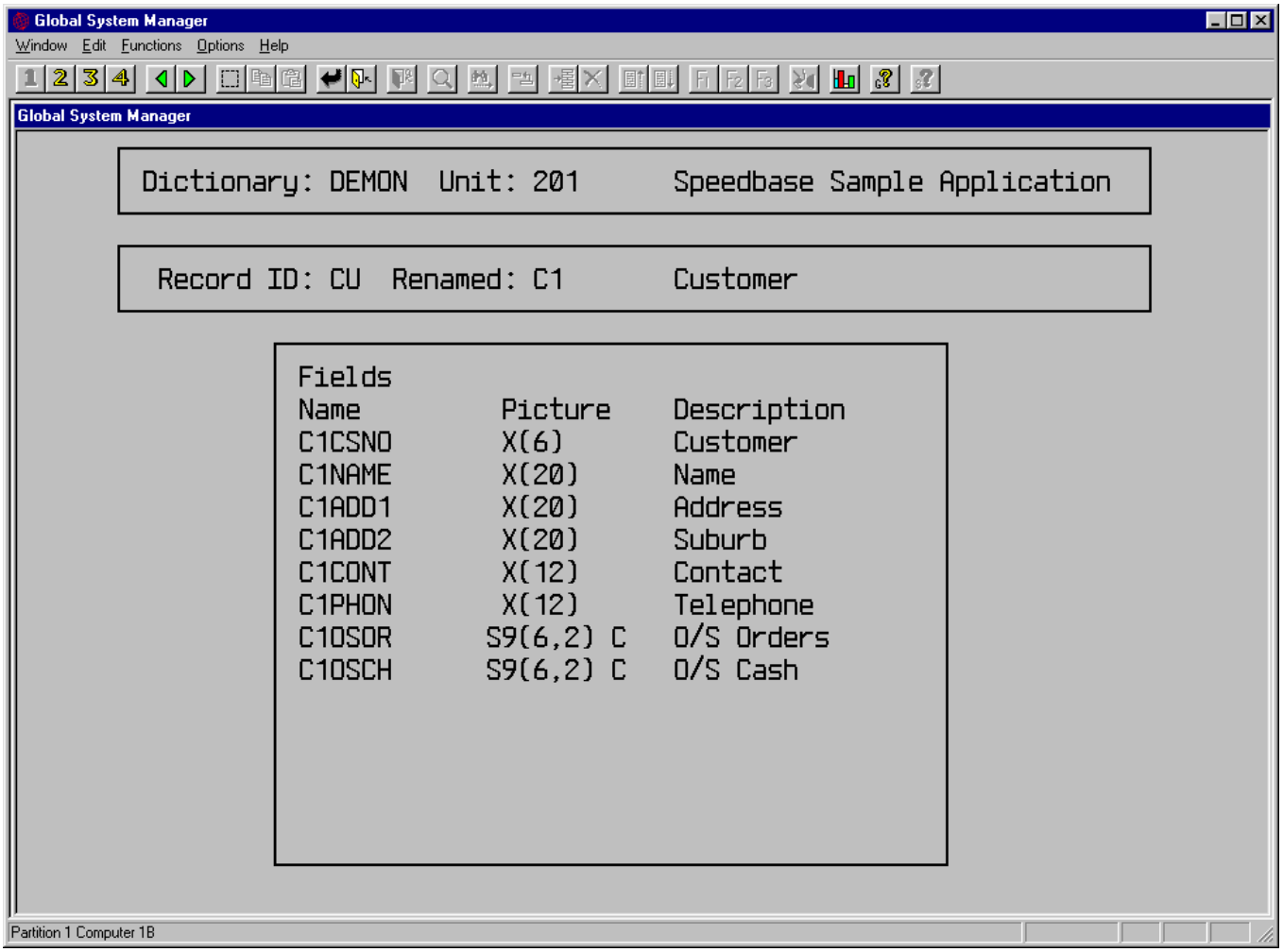


Figure E.6.1b - Field Selection Window

Key <RET> to select a field for inclusion in the window. If the window is scrolled, reply S or N to the baseline prompt:

```
Place in scrolled or nonscrolled area (S/N) ?
```

The editor places * to indicate your selection. Note that the order in which you select fields from this list is the order they will appear in the WINDOW construct source code when the editor generates it. Using the cursor keys you may select fields in any order. When you have done so, key <NXT>. The window template is then displayed and may be modified as described in section E.5. If you key <NXT> before selecting any fields a default template is generated using all the available fields. When you are satisfied with your window template, key <NXT> and the editor creates the appropriate source code and inserts it in the current buffer at the point at which you keyed command Q.

If, when prompted for a dictionary name, either when first setting up the window, or when selecting more fields for inclusion in the window, you key <RET> to the dictionary name prompt, you get an opportunity to define local fields. The editor displays a window into which you enter the field name, picture clause and description. These fields may be inserted in the window in the usual way.

E.7 Error and Warning Messages

The error and warning messages described in this section are displayed at the baseline at various stages in the editing session. They are erased on the next cursor movement.

Edit output file exhausted

There is no space in the buffer work file on disk to save the last change made. The editor terminates automatically. You may attempt to recover the work file, see Section E.1.4.

End line before start line - Aborted

The end line specified in a command D, H, or M precedes the start line, which is invalid.

File already exists

When saving a buffer, this message informs you that the file already exists on the work volume. Enter a new filename.

File in use or Invalid type - Reenter

The file entered is currently being accessed by another user, or it is not a valid text file. Check and reenter a valid filename.

File not found or in use

The file required was not found on the requested volume, or is being accessed by another user. Check and reenter.

Invalid key
Invalid command
Invalid function

The key, command or function used is invalid. Key <HLP> to display a Speedbase help window listing the valid responses.

Invalid picture clause

See Section 5.3 for valid picture clauses.

Invalid scroll dimensions

The window will not fit within the screen dimensions with the scroll statement parameters as specified.

Less than 2% available space - Please save

This warns that the main output file is short on available space. You still have at least 600 characters expansion left but you should save the current edit session and start again as soon as possible.

No item at this location

There is no item to move at this cursor position. Move the cursor to the beginning of the required field and try again.

No more selected fields

Either no fields have been selected for insertion, or all those selected have already been inserted.

No room for text field

There is no room to specify a text item at the current cursor location.

Non-blank line - Can't delete

The editor only deletes blank lines. To delete individual fields, use the delete field function.

Outside window boundary

Inserting a line at the current cursor location would cause part of the window to move off the screen, which is invalid.

Return to Buffer 0

You may not end or abandon the edit session when in buffers 1 or 2.

No room for workfile - Press Return
Not enough space in directory

The editor does not have enough space in the current volume to create the workfile. It requires space enough for the current input file size plus 32K. Check before continuing.

No room to read file
No room to output buffer

The editor has discovered that the available space in the buffer is insufficient to continue. Use command . to check available space.

Unable to determine end of window

The window construct must be terminated by an ENDWINDOW, ROUTINES SECTION or ENDFRAME statement.

Unable to fit in fixed area

The current field does not fit in the available non-scrolled area. Try placing it in the scrolled area.

Unable to fit in scrolled area

The current field does not fit in the available scrolled area. Try placing it in the non-scrolled area.

Unable to open dictionary

The editor has failed to open the specified dictionary file.

Unable to read dictionary

The dictionary file DI*xxxxx* specified is unreadable.

Unable to read work file

Suspect system corruption.

Window will not fit on screen

The window dimensions are outside the current screen range.

Work file in use - Try again later

The workfile the editor is trying to create already exists and is still in use. Check and correct before continuing.

Appendix F - Dictionary Maintenance Utility

The information about the structure of a Speedbase database is stored in a file called a data dictionary. For example, the Speedbase V3.0 sample application database has a dictionary file called DIDEMON. The dictionary is used by the Speedbase Presentation Manager utility programs to control their access to your database. The dictionary is also used in the compilation of your Speedbase application frames, see Appendix A.

Before you can compile your application you must therefore generate a dictionary and to do this you use the dictionary maintenance utility $SDM, as described in this appendix. The structure of the dictionary itself is unsuitable for the purposes of manipulation by a utility program, so for this purpose the dictionary information is maintained in a structure called a meta-dictionary. Once you have used the $SDM utility to establish the database structure you require, the meta-dictionary is used to generate the dictionary itself, see Section F.5.

This paragraph should probably be ignored if you are reading this appendix for the first time. The meta-dictionary is actually a special Speedbase database, and the dictionary maintenance utility $SDM is itself therefore a Speedbase application. It therefore has its own dictionary, DI$DICT, which is supplied with the Speedbase Development System, which you will use to create your meta-dictionaries, see Section F.7. If you use $SDM to examine the special DI$DICT dictionary you will see that it itself was generated using $SDM, from its own meta-dictionary DI$dict.

F.1 Running the Utility

To run the dictionary maintenance utility, use the menu entry you have set up, or key $SDM at the option prompt. Reply to the prompt:

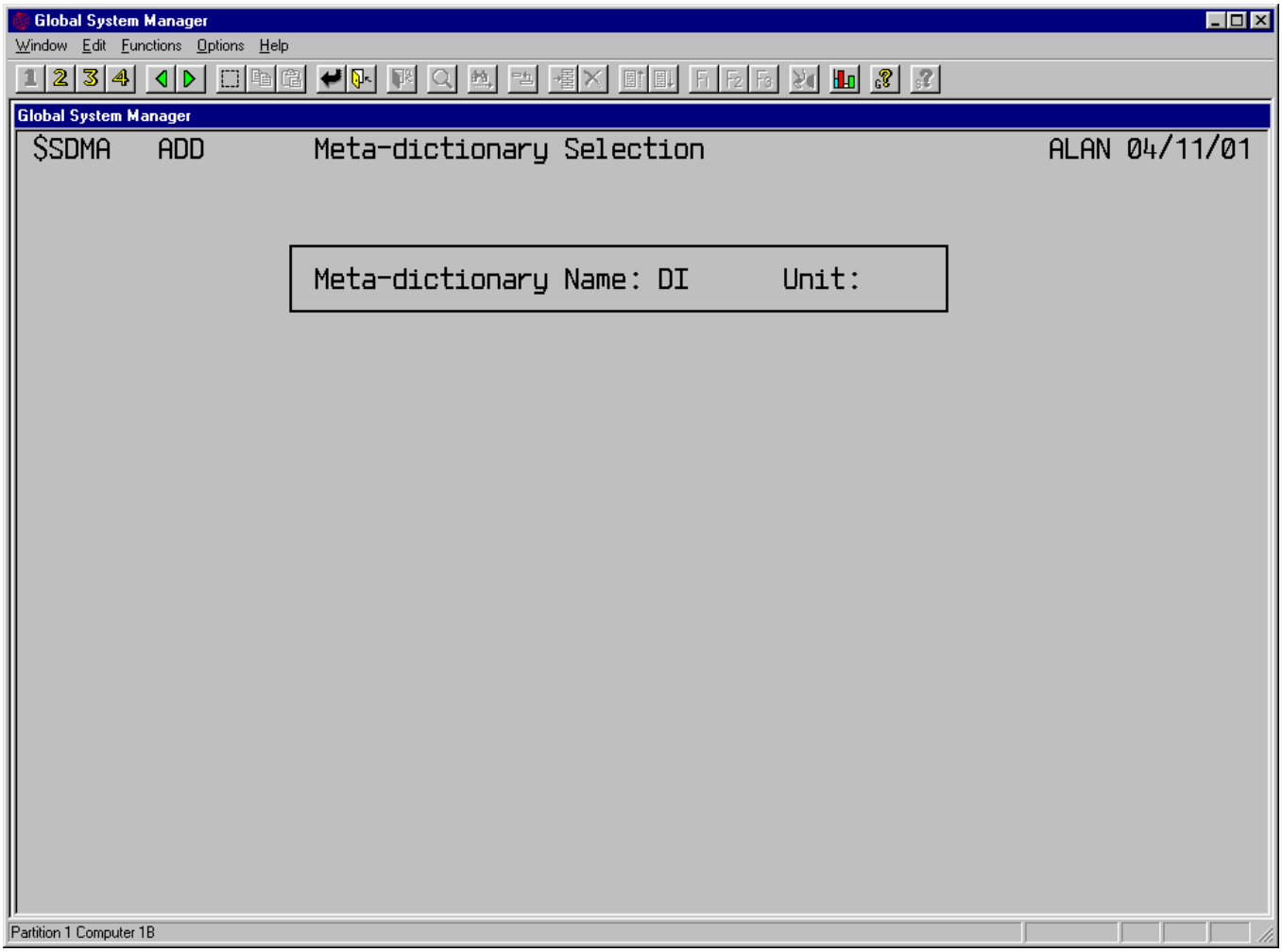


Figure F.1a - The Meta-dictionary Prompt

with the name and unit-id of the meta-dictionary you wish to maintain. Note that you must create the meta-dictionary before running $SDM, by using the generation utility as described in Section F.7. We suggest you adopt the convention that the name of the meta-dictionary defining the dictionary DI*XXXXX* is DI*xxxxx*. For example, the meta-dictionary for the sample application DIDEMON should be called DIdemon. The following sections describe how to create a new meta-dictionary and how to amend an existing one.

F.2 Establishing a New Meta-dictionary

A new, empty meta-dictionary is created by using the generation utility as described in Section F.7. Run $SDM and reply to the meta-dictionary prompts with the name and unit-id you used in creating it. For example, if you have created an empty meta-dictionary called DItest on unit FLS in order to define the details of a new Speedbase dictionary DITEST, the utility will display a menu window as follows:

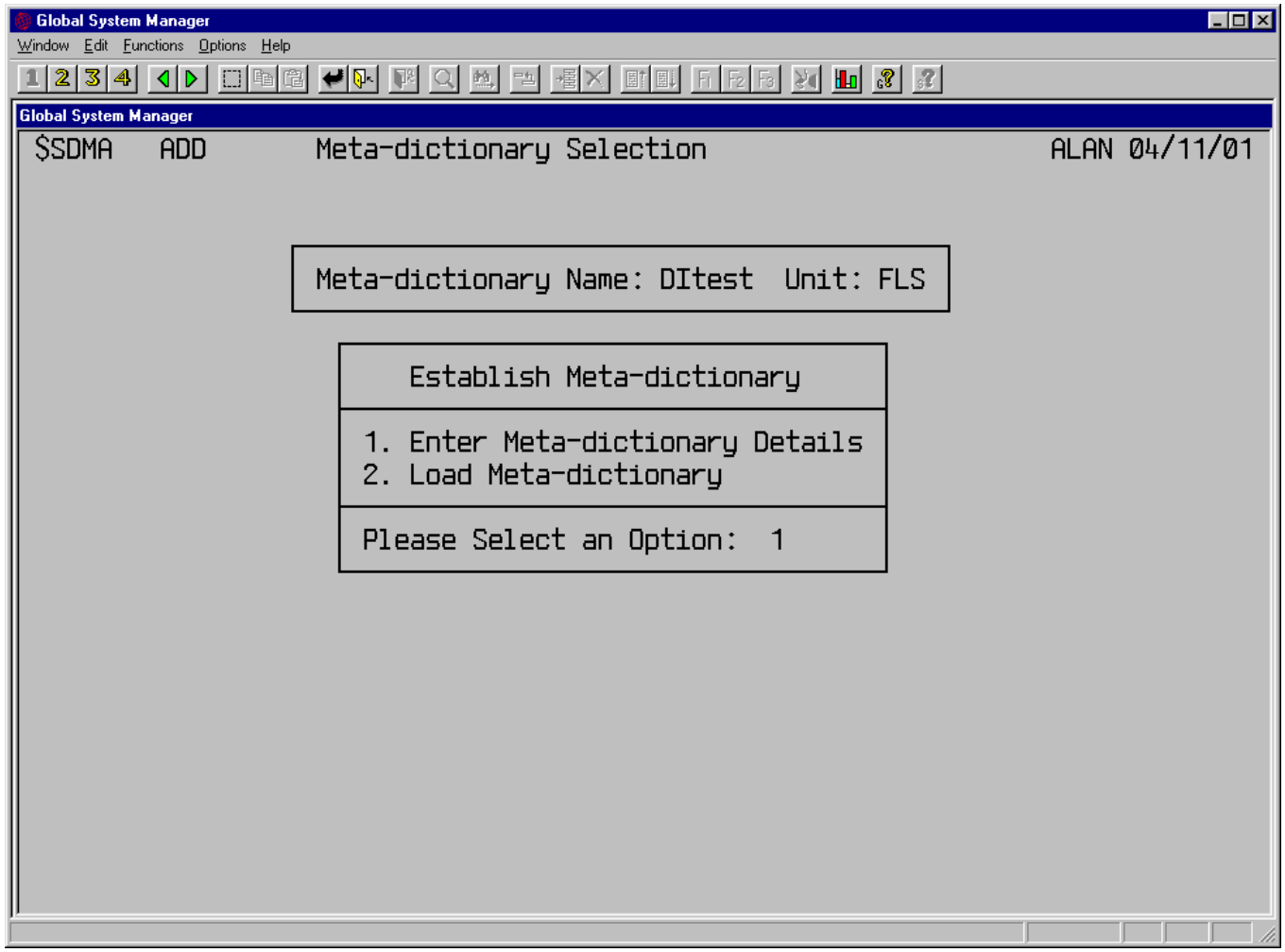


Figure F.2a - Establishing the Meta-dictionary

If you wish to enter the details of the meta-dictionary reply 1, and the utility prompts for the dictionary name and title. Since you are defining the details of dictionary DITEST on FLS you should make these replies to the prompts, enter some text to describe the database and the screen should look as follows, in Figure F.2b.

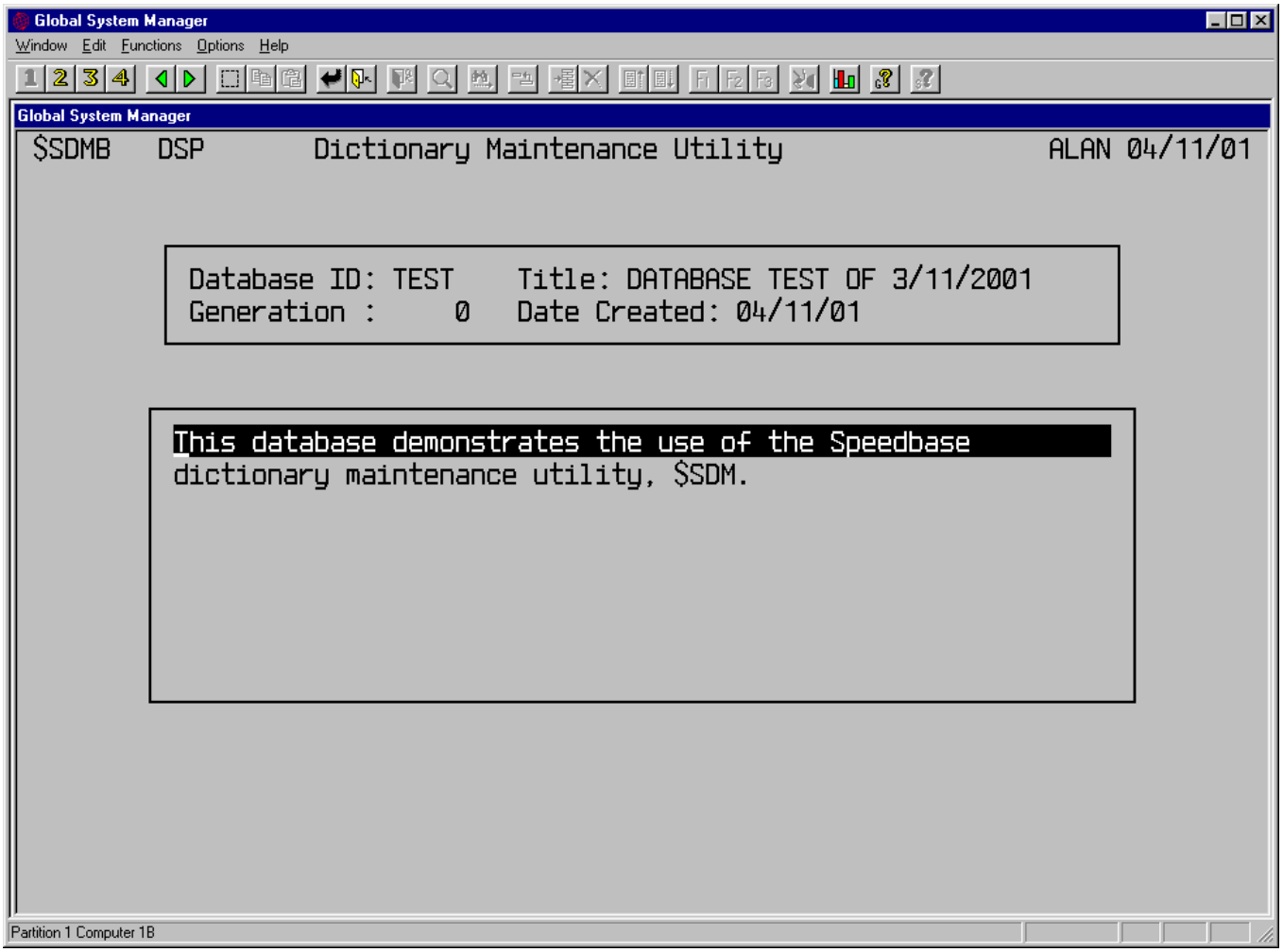


Figure F.2b - Entering the Database ID and Title

Key <NXT> to display the window listing the database record types. You then proceed as described in Section F.3, except that you will enter the various details rather than amending existing details.

If, instead of entering the details by hand, you wish to load the meta-dictionary with details from an existing dictionary, reply 2 at the establish meta-dictionary prompt. Enter the name and unit-id of the dictionary from which you wish to load details. For example, you could create the meta-dictionary DIdemon of the sample application by loading the dictionary DIDEMON provided. Having created an empty meta-dictionary for the purpose, see Section F.7, reply with the dictionary name DIDEMON and its unit- id to the prompts in Figure F.2c.

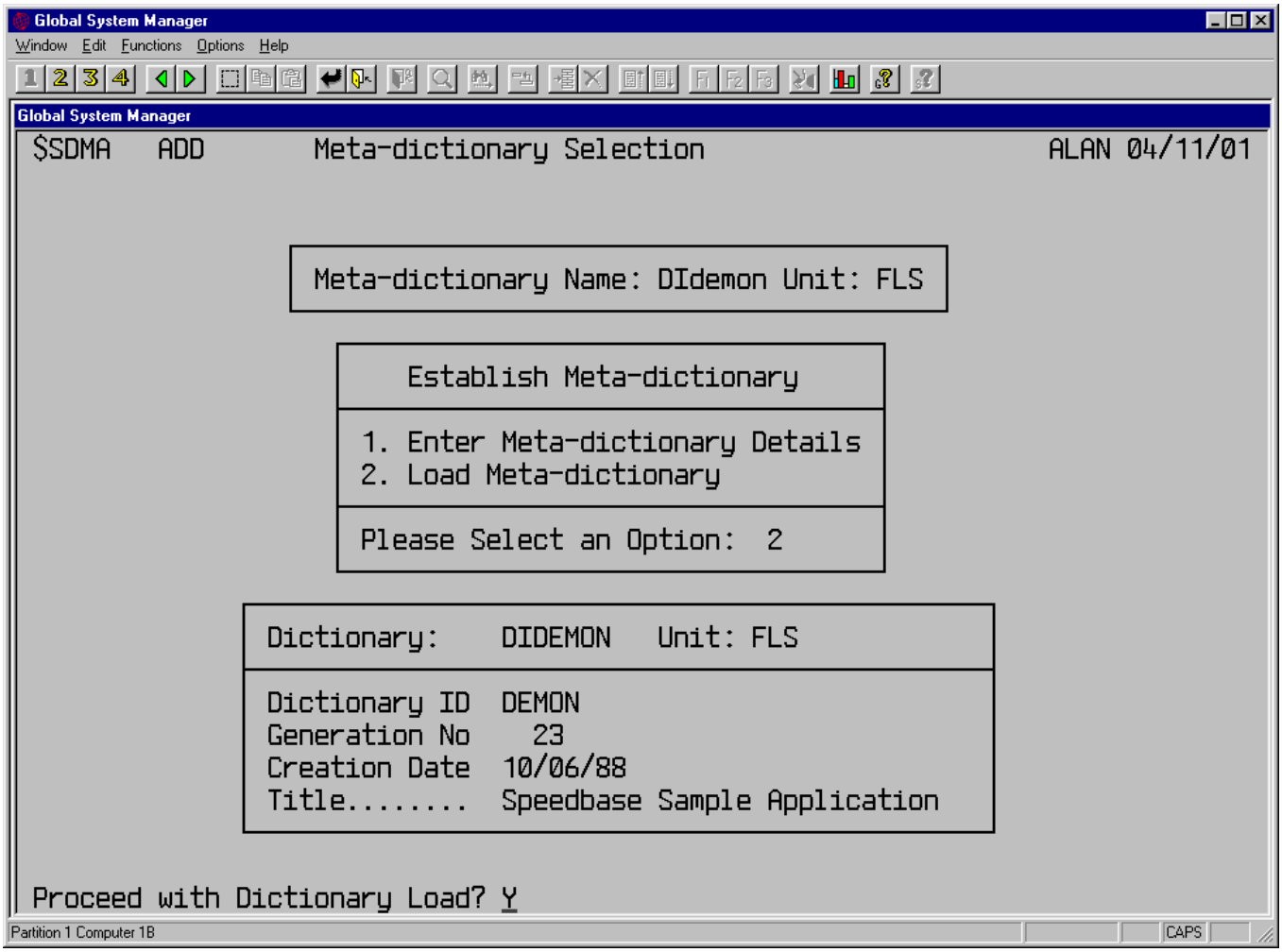


Figure F.2c - Loading the Meta-dictionary

Key <RET> to continue and load the Didemon meta-dictionary from the DIDEMON dictionary. Note that if you load a meta-dictionary from a dictionary produced prior to version three of Speedbase, some fields will be blank (e.g. the record description field). You may of course amend these blank fields. Once the meta-dictionary has been loaded in this way you may amend it, as described in Section F.3.

F.3 Amending the Meta-dictionary

Once you have created your meta-dictionary its details may be amended. The examples in this section make use of the sample application meta-dictionary, DIdemon. This is deliberately not provided and before proceeding you should create it, if you have not yet done so, using the steps documented in Section F.2.

To amend the sample meta-dictionary, run the dictionary maintenance utility, reply with DIdemon and its unit-id to the initial prompts, and the utility displays the options window:

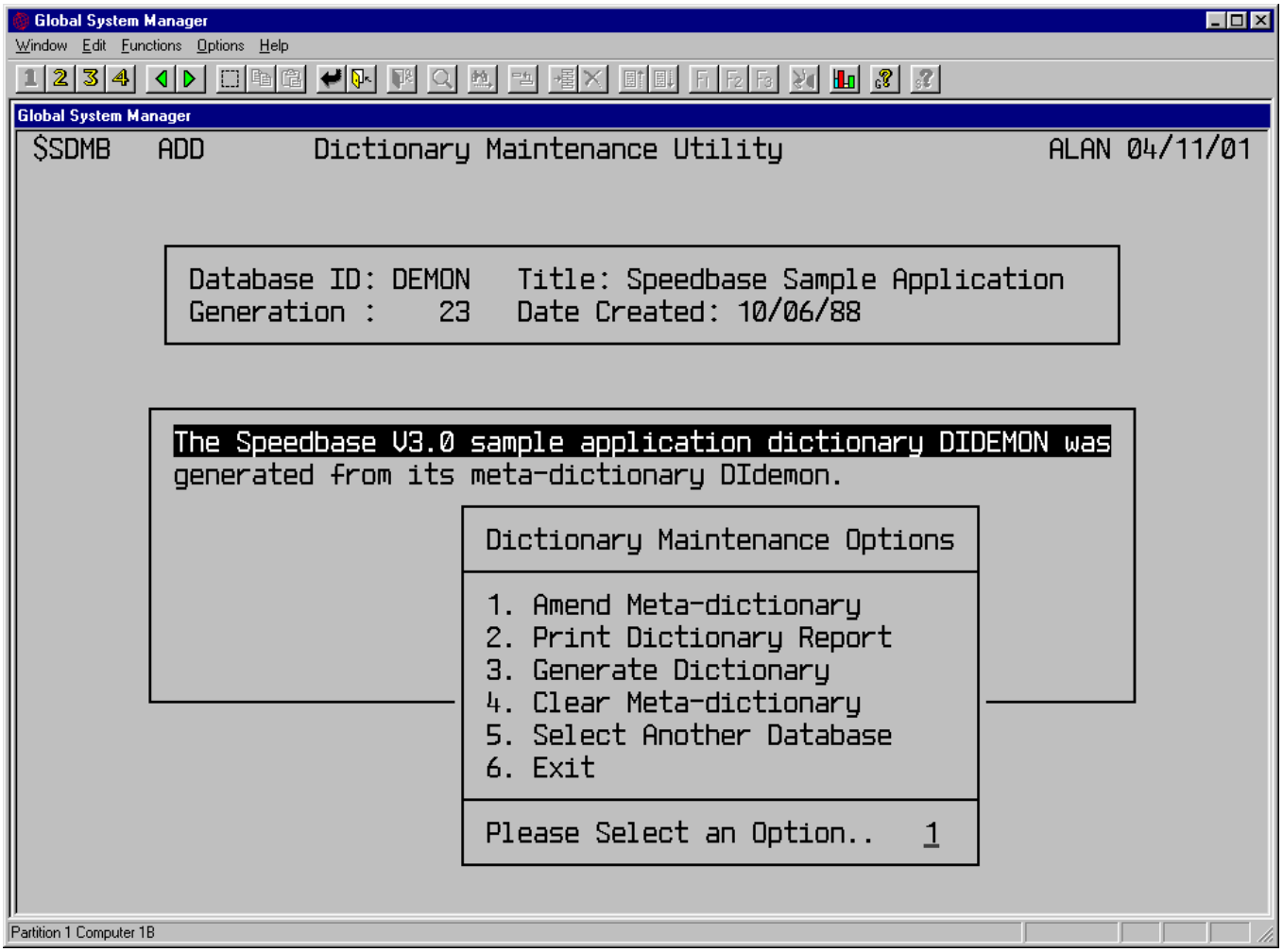


Figure F.3a - Amending the Sample Meta-dictionary

Enter <RET> to amend the sample meta-dictionary. You may amend the database ID and title, and the descriptive text. Key <NXT> and the utility displays a window listing the record types defined in the sample application, see Figure F.3b.

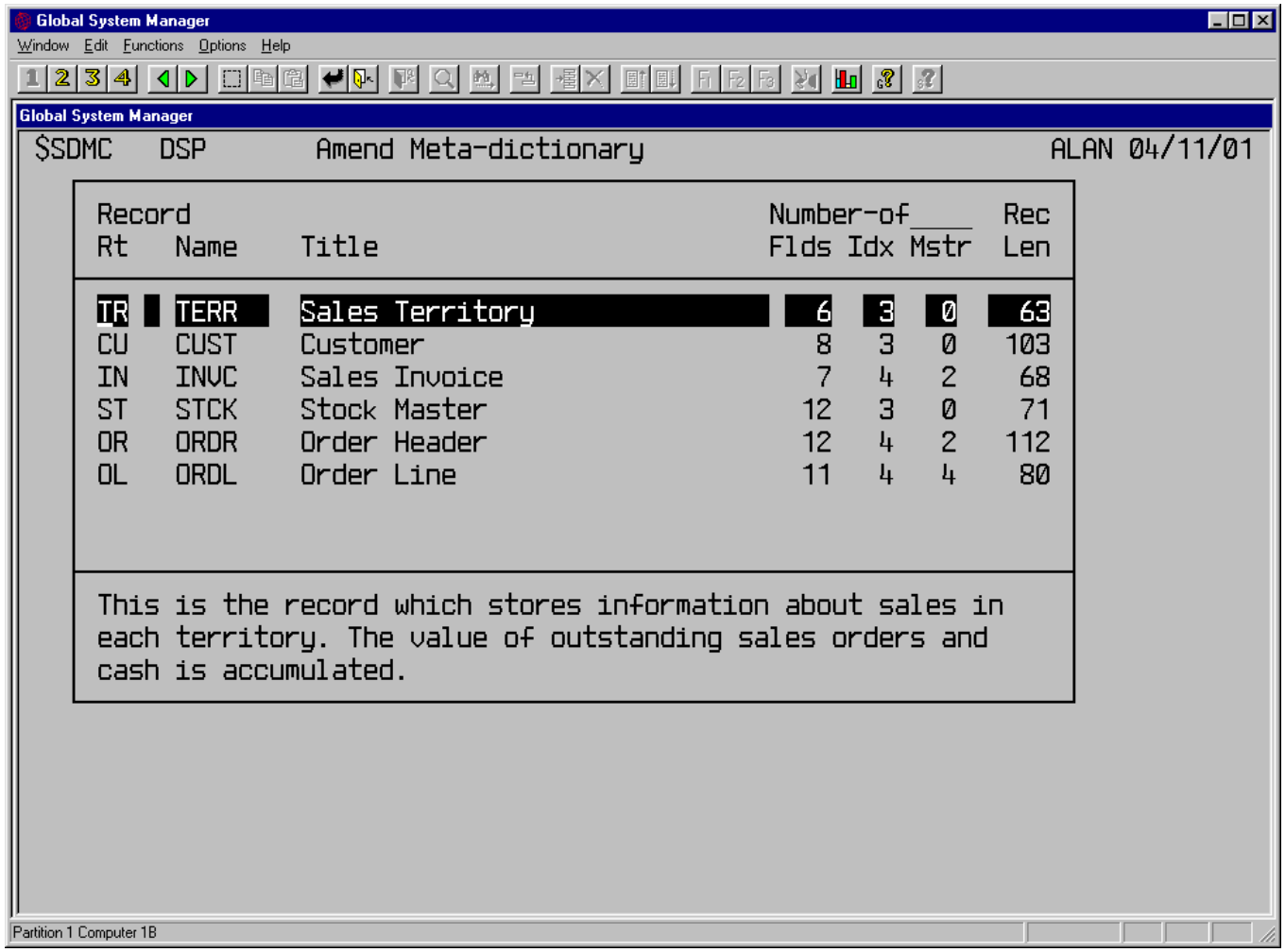


Figure F.3b - The Record-type Window

As with any Speedbase utility or application, you may key <HLP> to display a list of options and key <HLP> again to display the Speedbase help window. If you key <HLP> after displaying the record-type window the list of options is displayed:

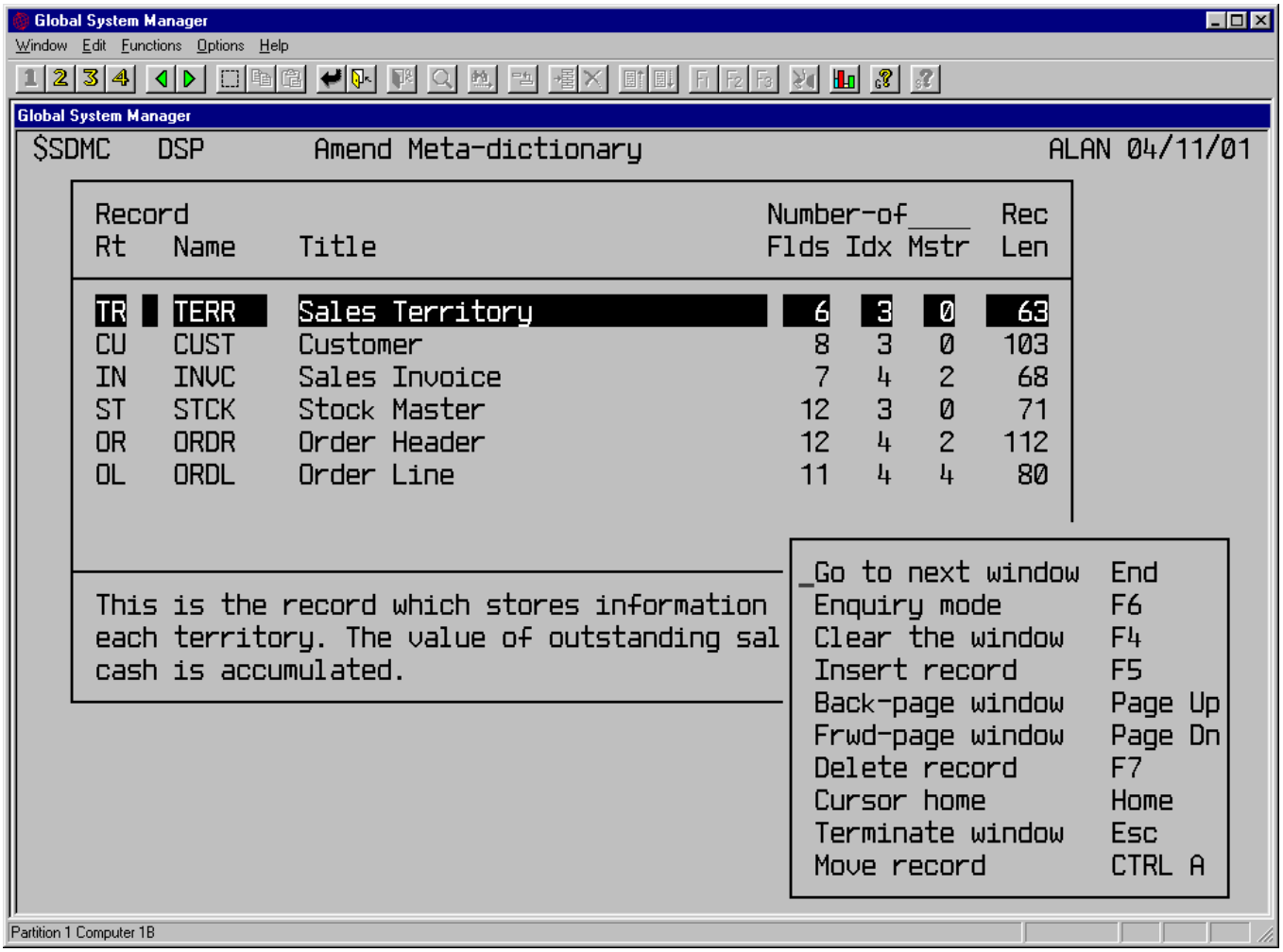


Figure F.3c - Help in the Record-type Window

Note the last option, move record. This is a special option which allows you to change the order in which the record-types are stored and displayed. It makes use of a specialised feature of Speedbase called an auto-sequence index, explained in Section F.8. If you make regular use of the dictionary maintenance utility, you should assign a key to the move record function, using the Speedbase Presentation Manager customisation utility.

You may now amend the record name, title and the description which is displayed in the non-scrolled area of the window. To see the effect of the move record function, place the cursor on the customer record and key <MOV>. Move the cursor to where you wish to insert the customer record, above the territory record for example, and key <INS> (i.e. insert record). If you should wish to abort the move after keying <MOV>, key <HLP>, move the cursor to the last line of the help window, "Abort move", and key <RET>. Note that the order in which the records are displayed is the order they are written to the dictionary, DIDEMON in this case, and therefore also the order in which they are processed by the Speedbase compiler.

Move the cursor to the order header record and key <NXT>.

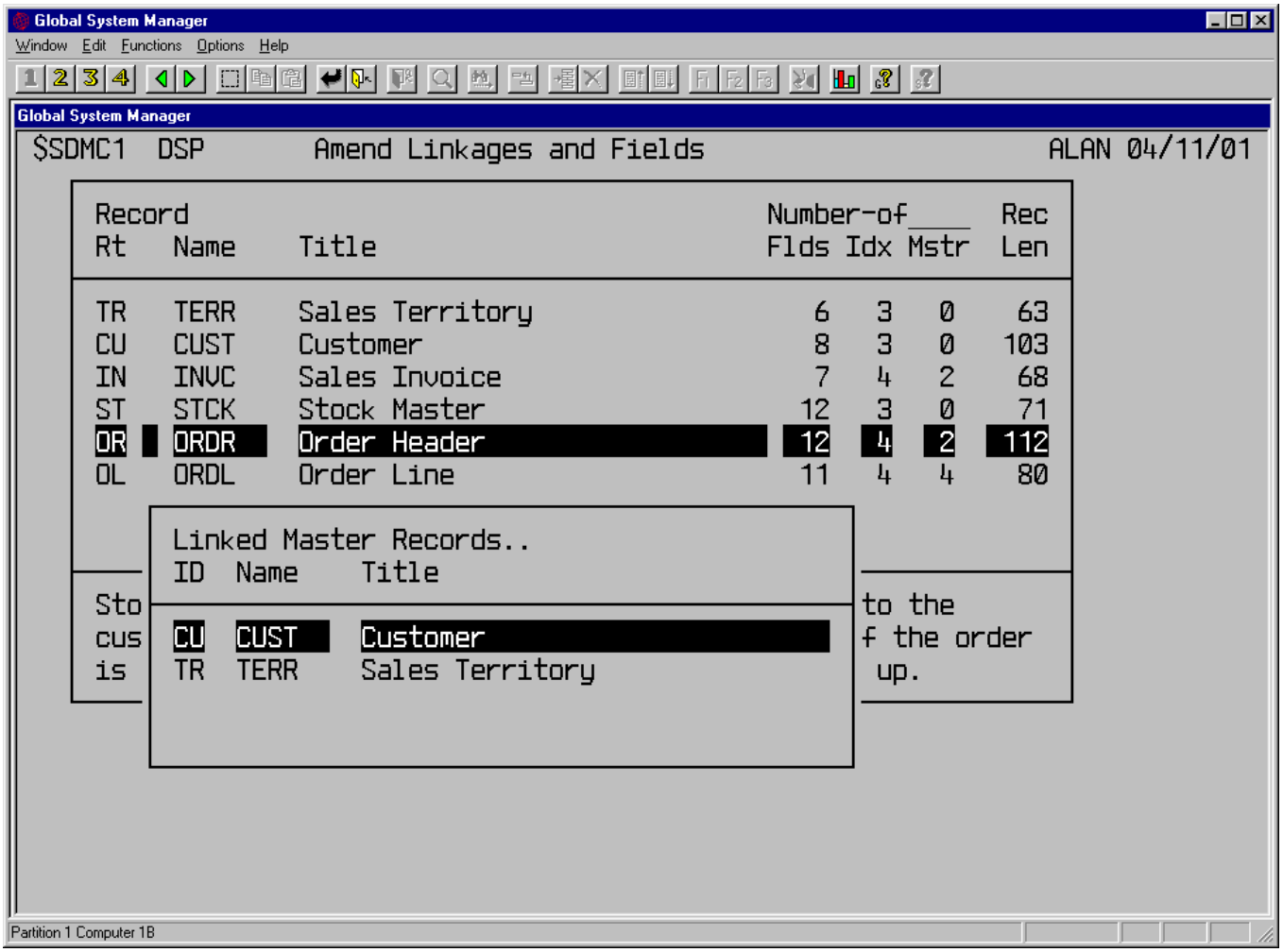


Figure F.3d - The Linked Masters Pop-up

This pop-up window shows which records act as masters to the current record (i.e. the customer and sales territory records in the case of the order header record). If you add a new master in this window and the appropriate master access key fields are not present on the record, when you key <NXT> to continue the utility displays the baseline prompt "Link fields are needed for these masters. Create them? Y". If you reply Y it adds the appropriate fields to the record. Otherwise you must delete the master relationship before continuing. Key <NXT> and the fields on the order header record are displayed.

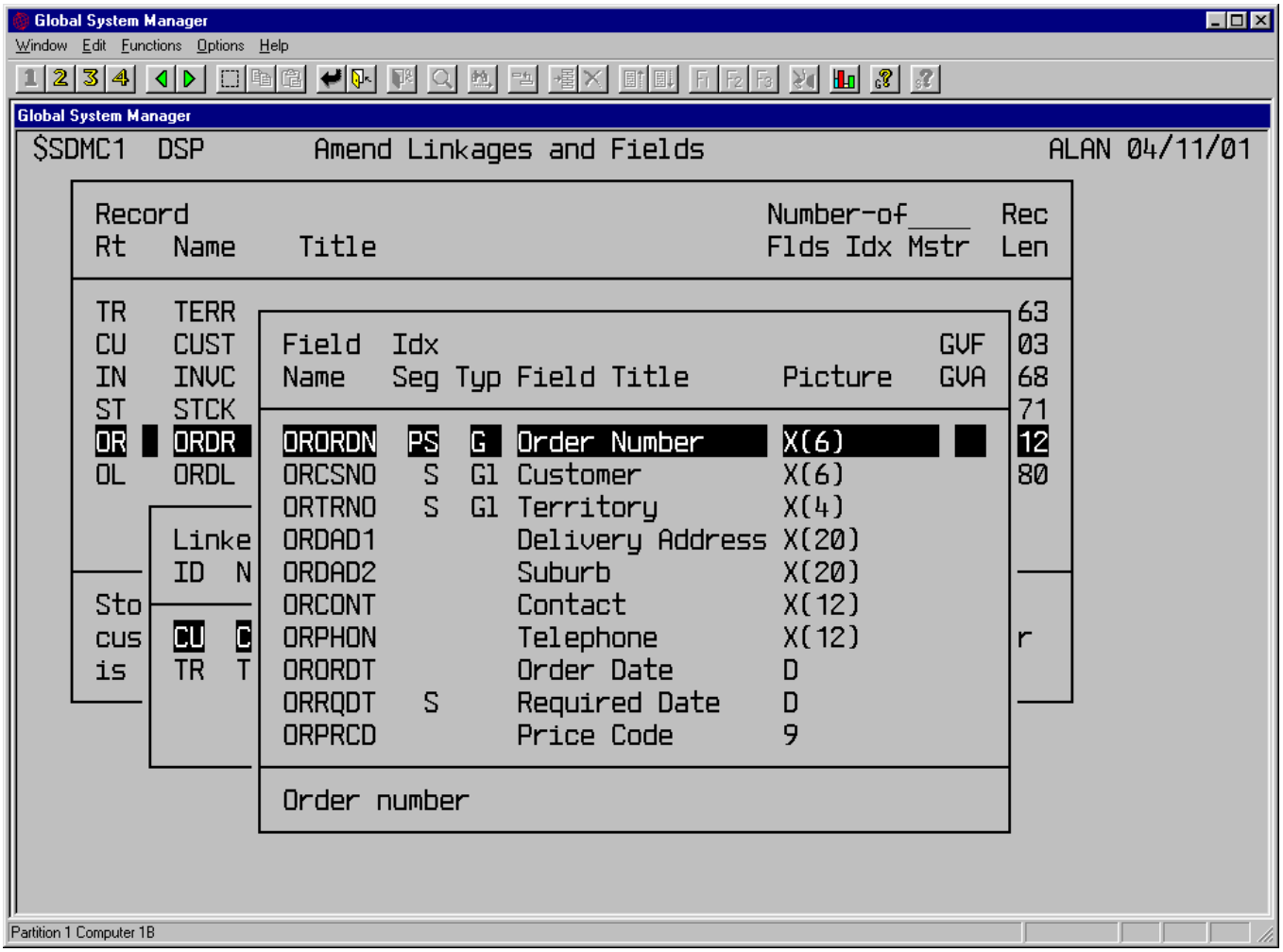


Figure F.3e - Field Details

The fields which make up the primary index of the record are marked P in the Idx Seg column (e.g. the order number field on the order header record). The order of the segments of the primary index is therefore defined by the order in which the fields themselves are listed. To change the order of the segments of the primary index you should make use of the move function to change the order of the fields themselves.

If the field is part of any secondary index it is marked S in the next column. Fields are often used in more than one record type and these are known as global fields. For example, the order number, customer and territory fields on the order header record are used on the order line, customer and sales territory records respectively. Global fields are marked G in the Typ (i.e. type column).

The parameters of a global field (i.e. its name, title and picture) are maintained across the meta-dictionary. Any change you make to one of these parameters is applied everywhere it is used. For example, if you change the title of the customer field on the order header record from "Customer" to "Customer number", the utility displays the baseline message "Warning: Field title will be changed DB-wide from Customer" before doing so.

Global fields provide the links between a record and its masters and comprise the master access key, see Section 2.6. If a field is part of any master access key it is marked Gl in the Typ column. On the order header record, for example, the customer field is the master access key

to the customer record and therefore provides a link to it. Move the cursor to the last field on the order header record, ORTOTL, the order amount field. Because this field is a GVA it is marked A in the GVF/GVA column. Move the cursor to the GVA column and key <UF1>. The GVA pop-up is displayed, see Figure F.3f.

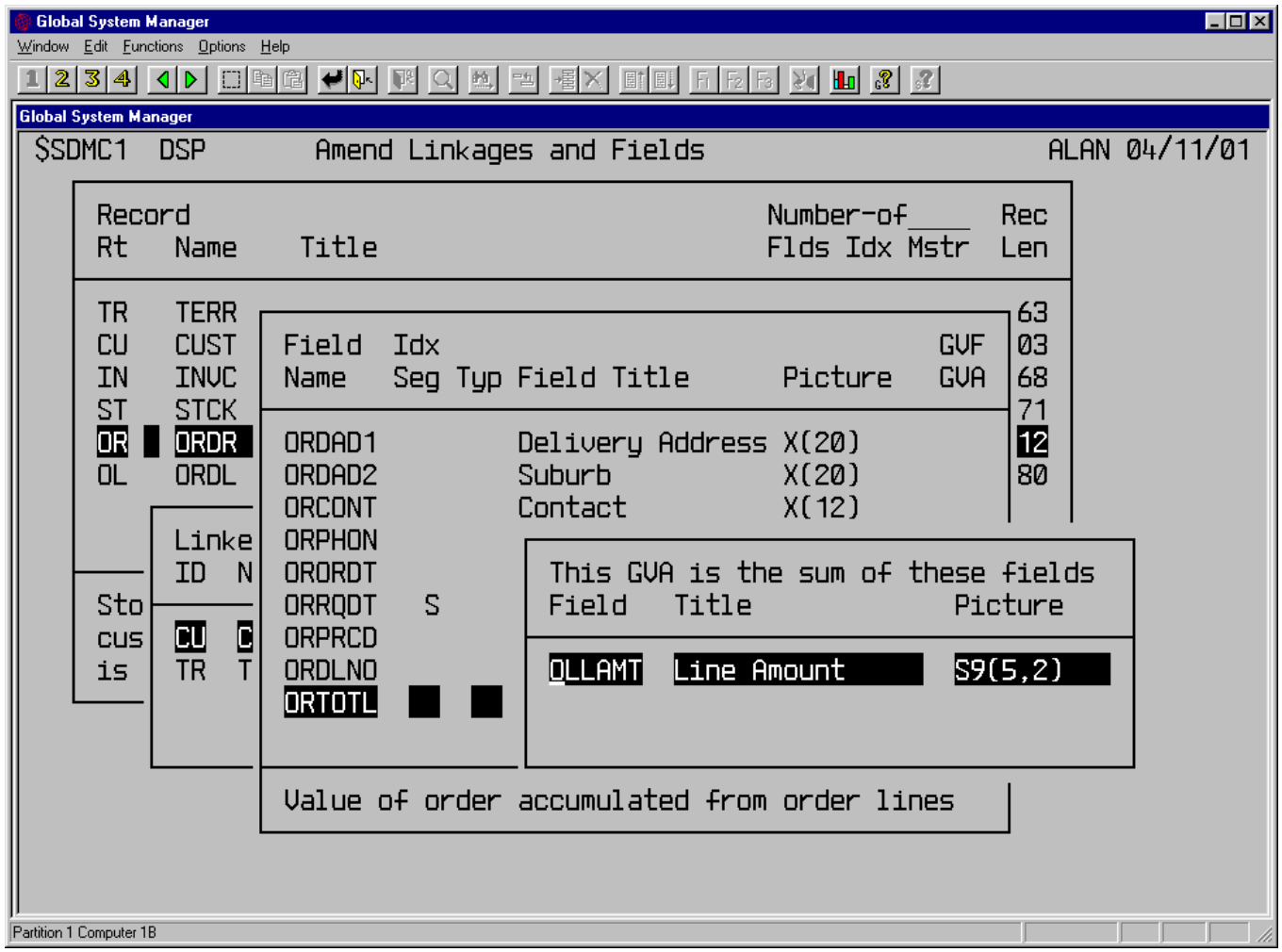


Figure F.3f - The GVA Pop-up

The GVA pop-up shows which fields make up the GVA, in this case the single field OLLAMT from the order line record. If you display the fields of the order line record you will see that the field OLLAMT is marked F in the GVF/GVA column. If you select the field, by placing the cursor on it and keying <RET>, then <SKP>, usually tab, the GVF pop-up is displayed:

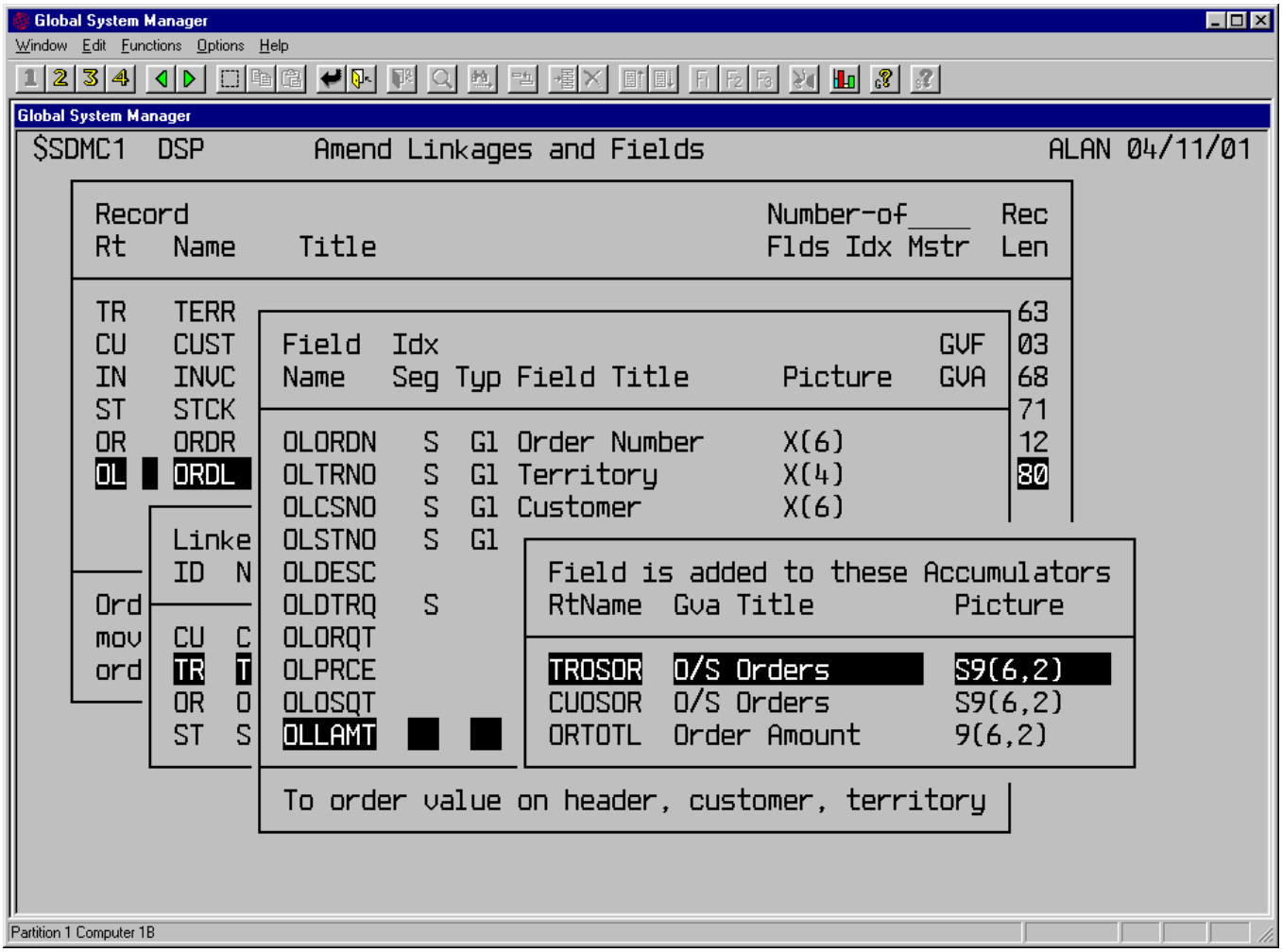


Figure F.3g - The GVF Pop-up

The GVF pop-up shows to which GVA fields the GVF field is added, in this case the order amount field and the outstanding orders field on both the territory and customer records. Key <NXT> to return to the field window and <NXT> to display the index pop-up:

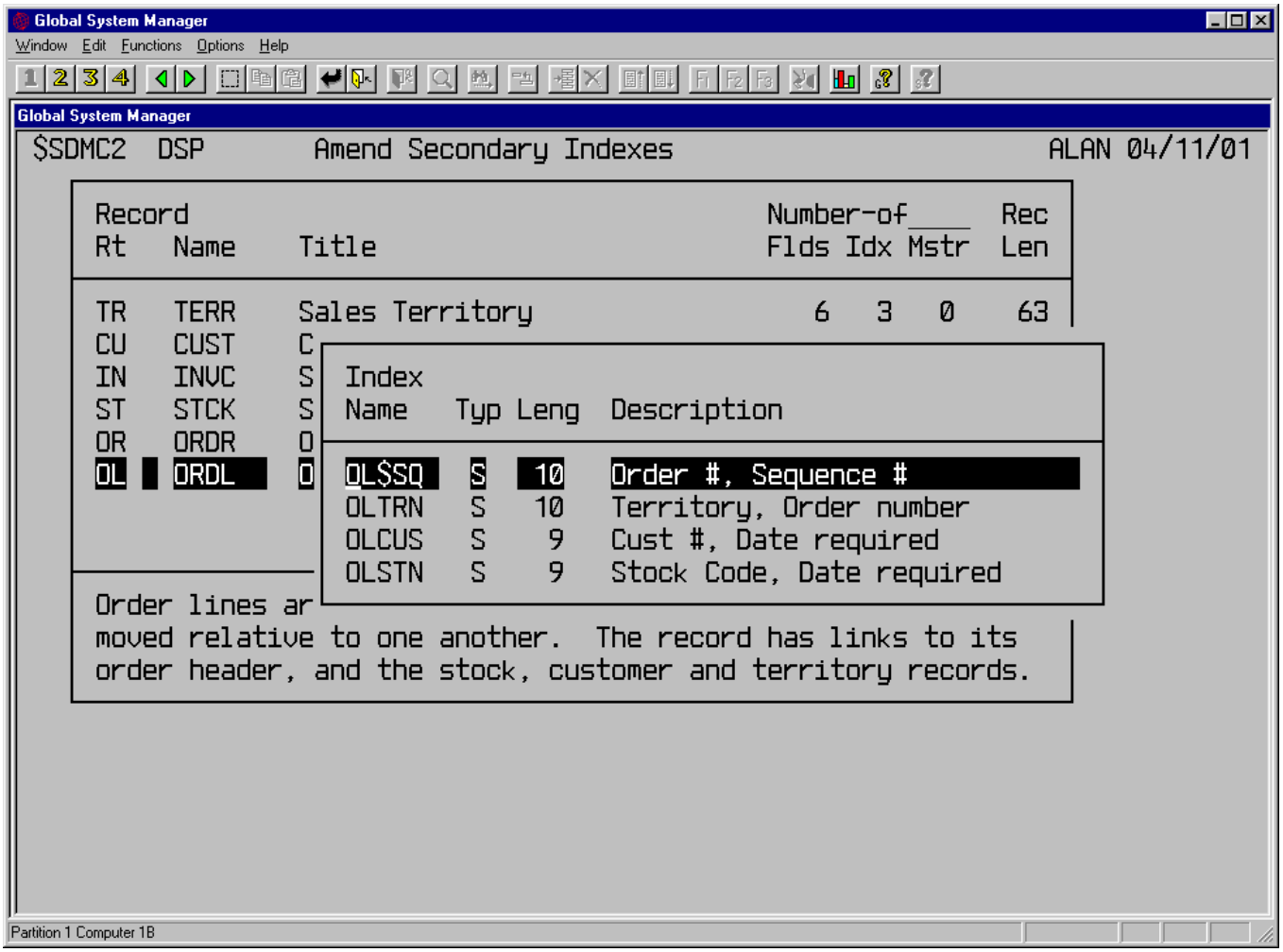


Figure F.3h - The Index Window

The index window lists the secondary indexes and the primary index, if any. Place the cursor on the first secondary index, OLTRN and key <RET> to select it. Key <NXT> to display the list of segments making up the index. When adding index segments you may key <UF1>, usually F1, when prompted for the field name. This displays a pop-up window showing the fields available. To select a field, place the cursor on it, key <RET> and its name is used automatically as the index segment. Note that the primary index segments are displayed in the field window instead, Figure F.3e.

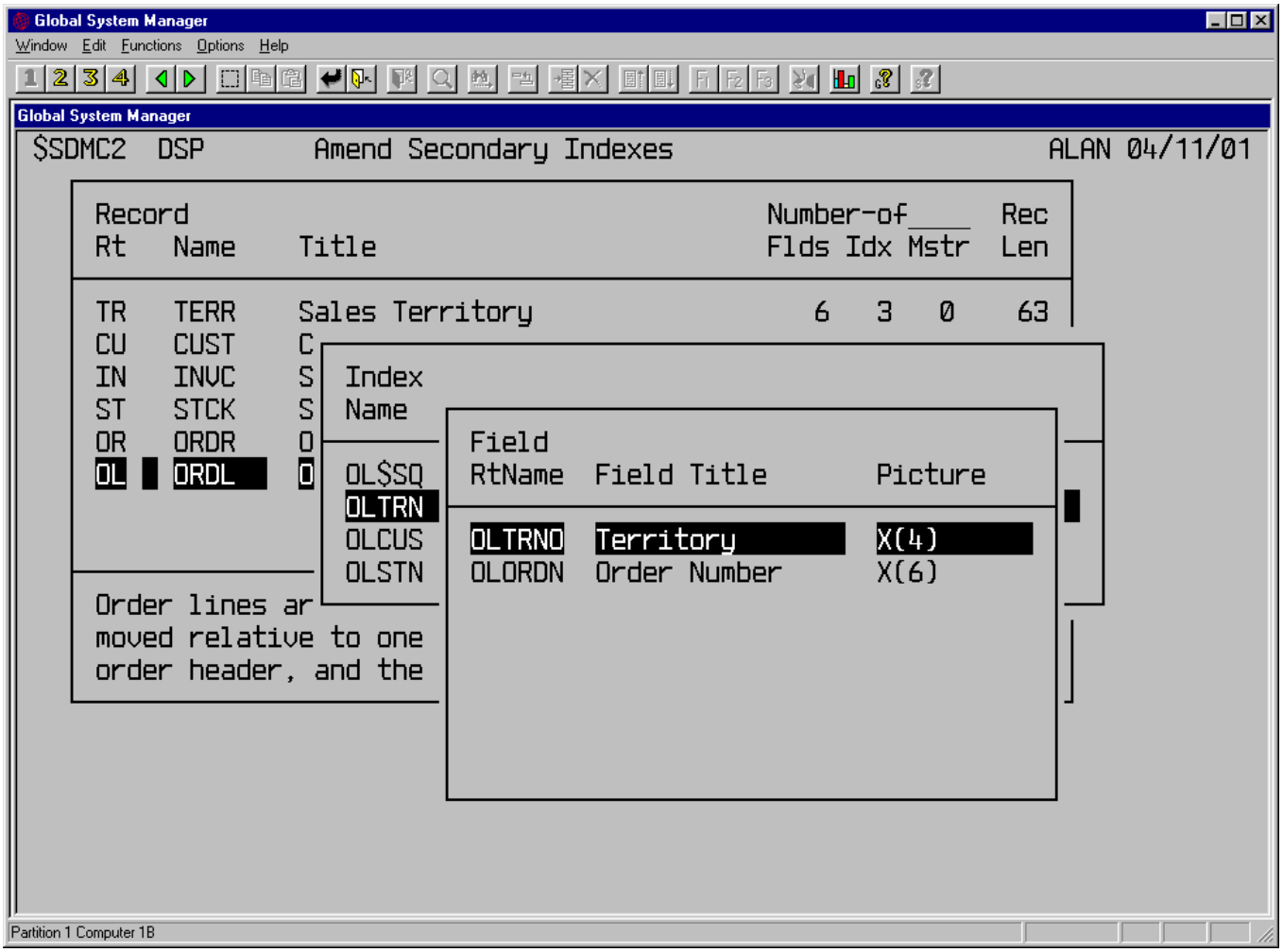


Figure F.3i - The Index Segment Pop-up

F.4 Printing the Dictionary Report

To print the dictionary report, run the dictionary maintenance utility, reply to the meta-dictionary name and unit prompts and key <NXT> to display the maintenance options:

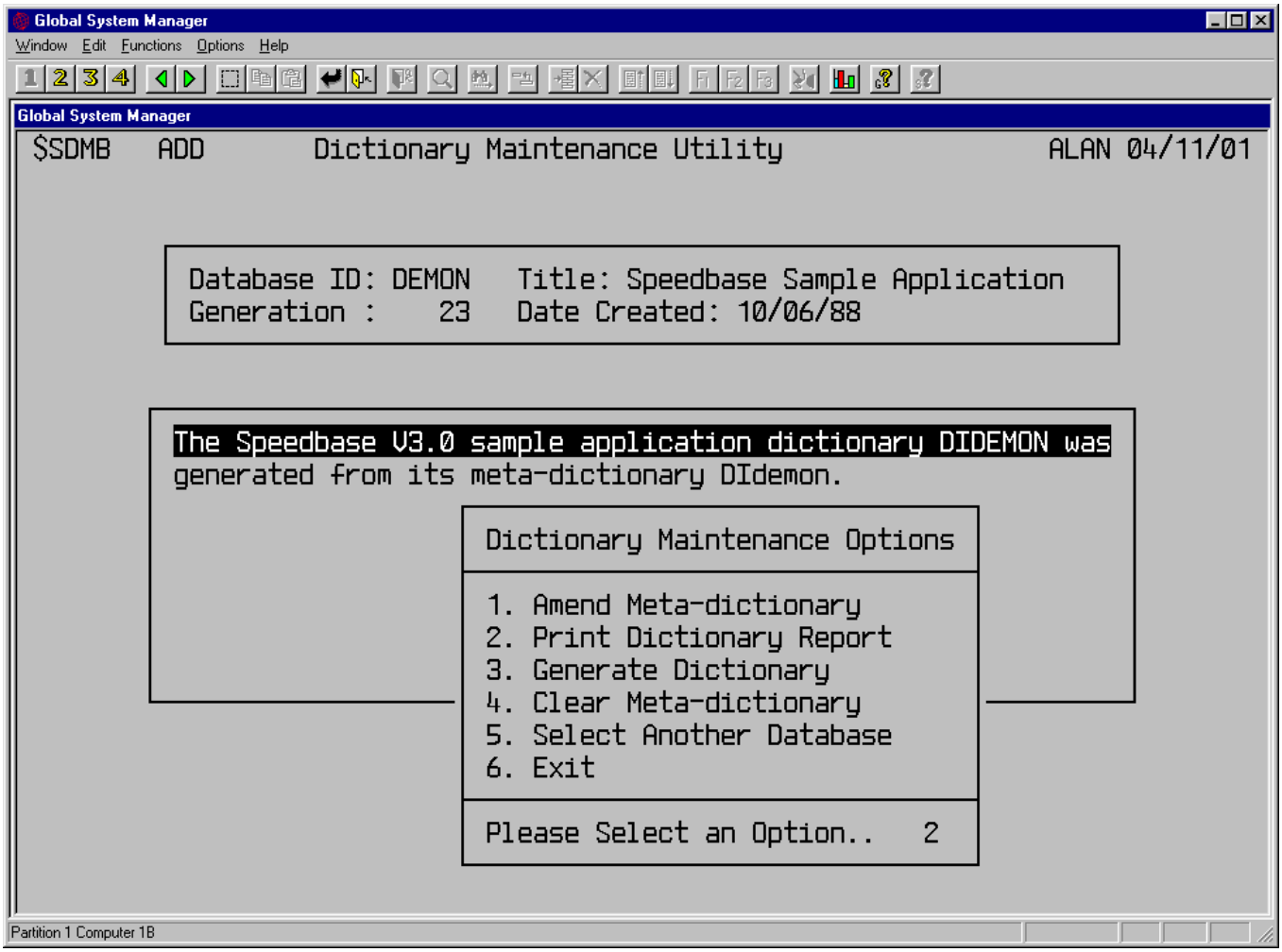


Figure F.4a - Dictionary Maintenance Options

When you reply 2 to print the dictionary report, the utility displays the confirmation window:

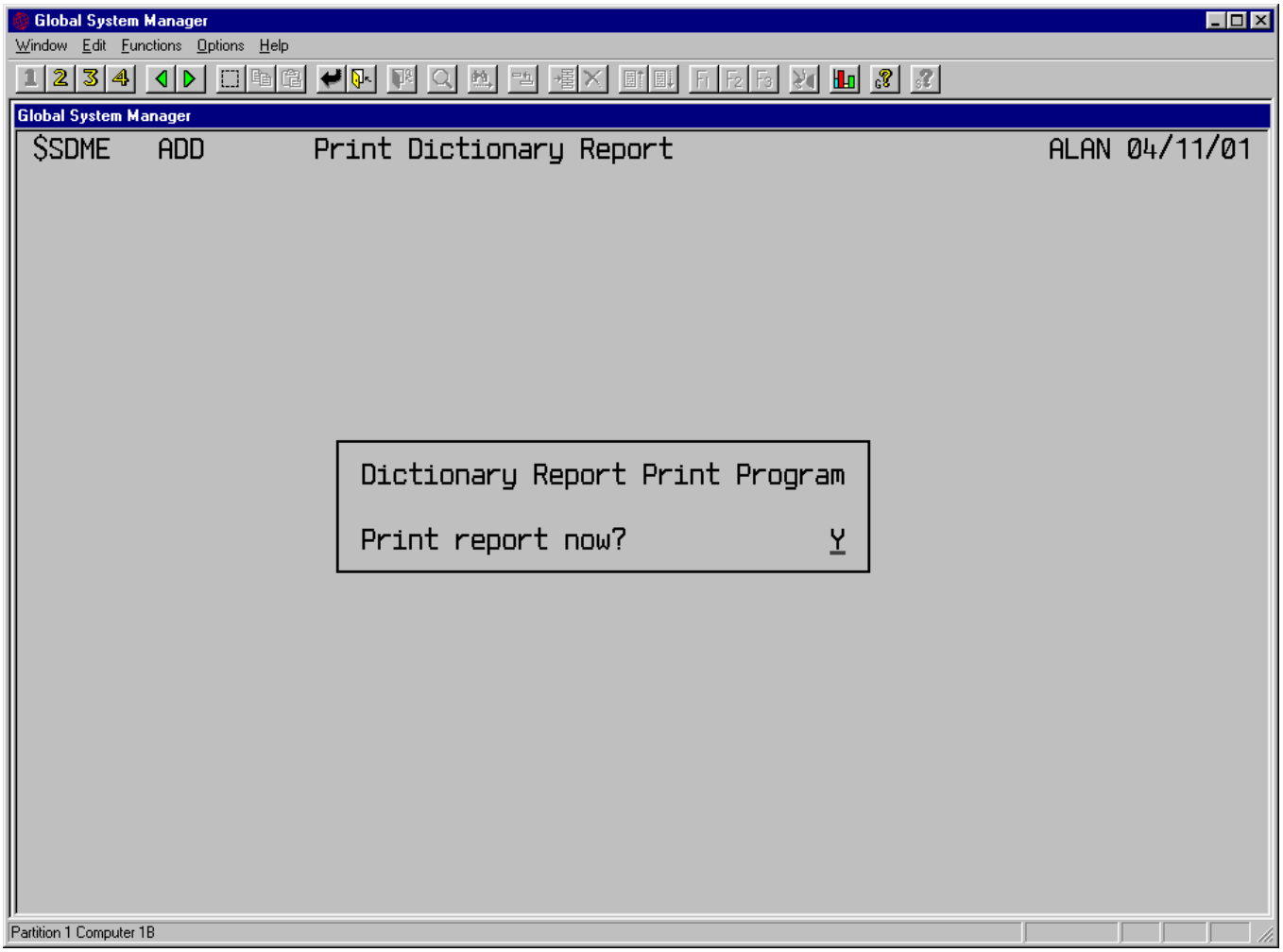


Figure F.4b - Confirming the Dictionary Report Print

Figure F.4c shows the beginning of the report on the sample meta-dictionary, DIdemon. The database description and list of record-types is printed on the first page. This is followed by information on each record-type in turn. In our example the first record-type is the territory record.

The indexes are listed with their descriptions and a list of their index segments. For the territory record the first index is the primary index TRTRN, with the single segment TRTRNO. Following the list of indexes is a list of master and servant records. These are identified by the index which provides access to them. For example, on the territory record, the invoice record is a servant and access to it is by the index INTRN, which has segments INTRNO and INDATE. In the case of a servant record it is possible that there is no index providing access to it, and this is indicated by an ID of the form *rt****, where *rt* is the servant record-type.

The index, master and servant list is followed by a list of the fields on the record, together with the field attributes (i.e. name, index segment, type, title, picture, GVF/GVA and description). For a GVF the names of its associated GVAs are printed after "To". For a GVA the names of its associated GVFs are printed after "Fm", short for from.

Once you reply to the confirmation prompt, the report is printed. It may be interrupted, and printed on a different printer or cancelled, in the normal way.

```
DICTIONARY ID: DEMON Speedbase Sample Application  GEN: 22 DATE:18/08/89 PAGE 1

================================================================================
                     R E C O R D    S U M M A R Y
================================================================================

--------------------------------------------------------------------------------
   Record  Record
     ID     Name    Record Title
--------------------------------------------------------------------------------
     TR    TERR     Sales Territory
     CU    CUST     Customer
     IN    INVC     Sales Invoice
     ST    STCK     Stock Master
     OR    ORDR     Order Header
     OL    ORDL     Order Line
--------------------------------------------------------------------------------

================================================================================
  T R  T E R R                          Sales Territory
================================================================================

  This is the record which stores information about sales in
  each territory. The value of outstanding sales orders and
  cash is accumulated.

--------------------------------------------------------------------------------
  Index  IDX-Typ  Index-Decription                                Index-Segments..
--------------------------------------------------------------------------------
  TRTRN  Primary  Primary index, territory code                   TRTRNO
  TRNAM  Secndry  Territory name (sales area)                     TRNAME
  TRACM  Secndry  Account manager                                 TRACMG

  INTRN  Servant  Territory code, Invoice date                    INTRNO/INDATE
  OLTRN  Servant  Territory, Order number                         OLTRNO/OLORDN
  ORTRN  Servant  Territory code, required date                   ORTRNO/ORRQDT

--------------------------------------------------------------------------------
  Field Idx Ty Field-Title..       Picture     GVF / GVA         Field-Description
--------------------------------------------------------------------------------
  TRTRNO P  G  Territory           X(4)                          Territory code
  TRNAME  S    Sales Area          X(20)                         Territory name
  TRACMG  S    Account Manager     X(15)                         Account manager
  TRREFN       Reference           X(11)                         Reference
  TROSOR       O/S Orders          S9(6,2)     Fm OLLAMT         Accumulated orders
  TROSCH       O/S Cash            S9(6,2)     Fm INIAMT         Accumulated cash
```

Figure F.4c - The Dictionary Report

F.5 Generating the Dictionary

Once you have completed the amendments to the meta-dictionary you may generate the dictionary itself. Run the utility, reply to the meta-dictionary name and unit prompts:

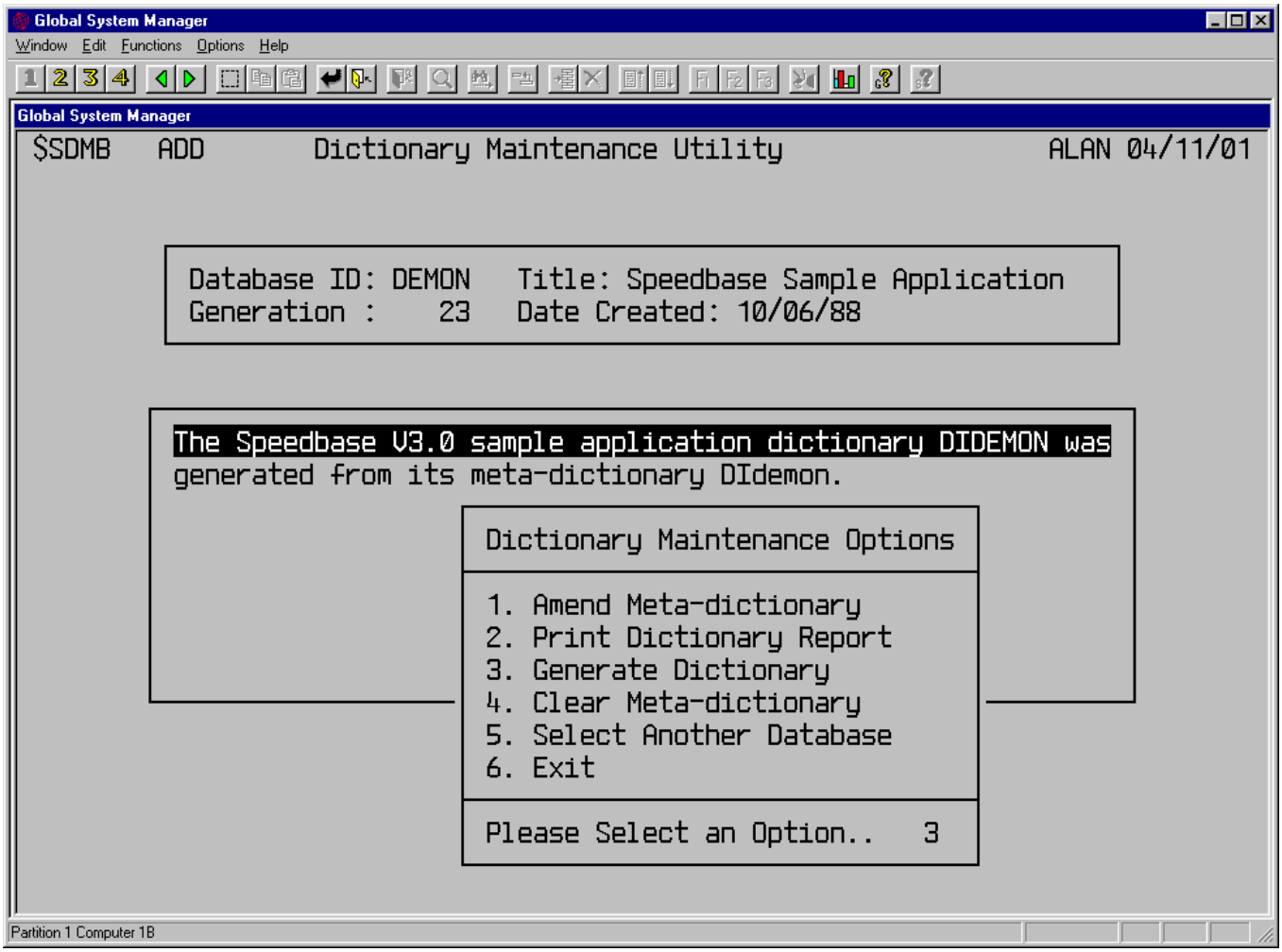


Figure F.5a - Dictionary Maintenance Options

Reply 3 and reply to the database name, unit and size prompts to generate the dictionary:

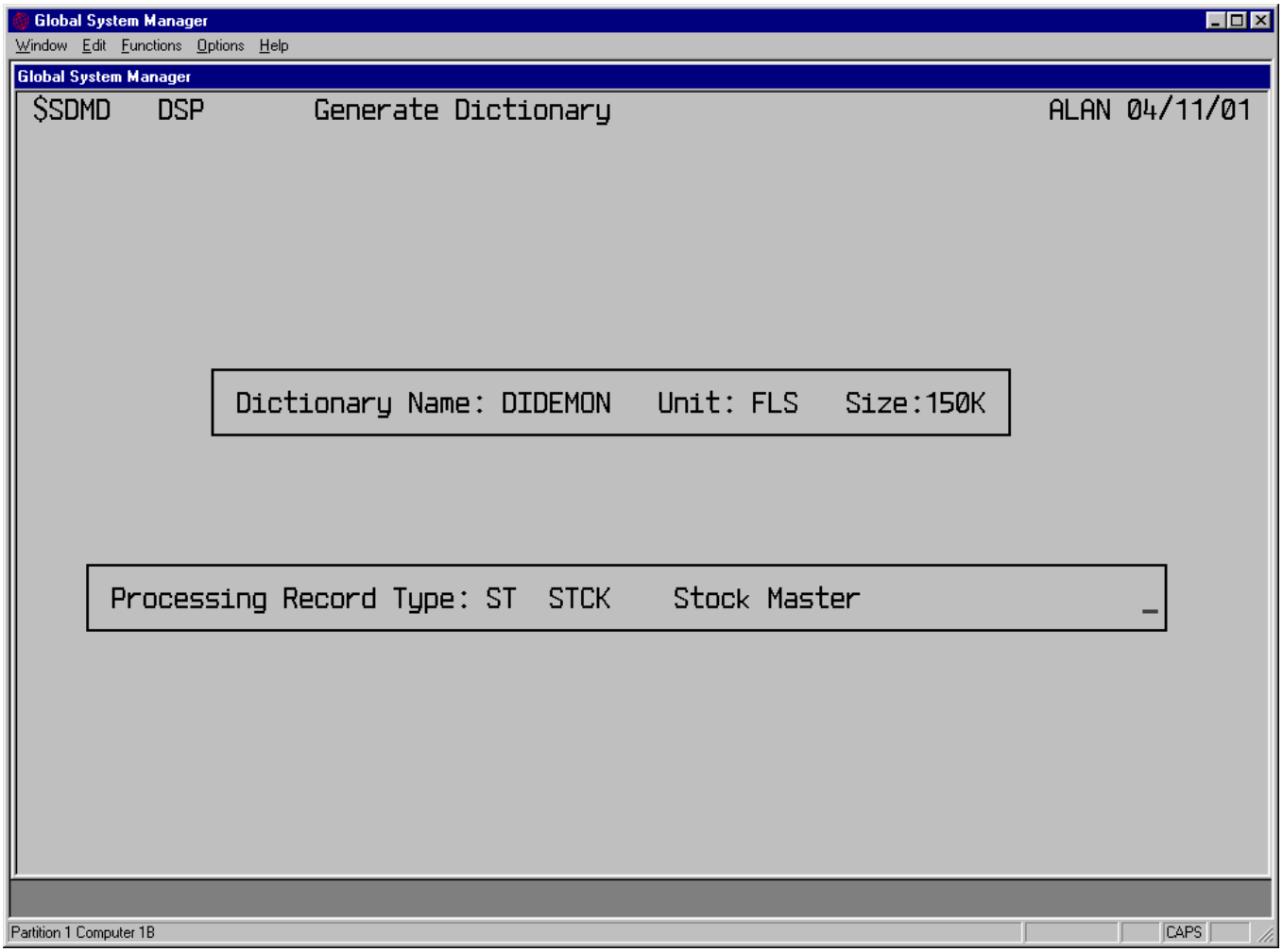


Figure F5b - Generating the Dictionary

The default size of the dictionary file is 150 Kbytes, which will be sufficient unless you have an unusually large number of text records. Note that the dictionary file is truncated when closed so that no disk space is wasted by using the default size. The utility displays the name and title of the record type it is processing. Note that processing is terminated if any of the errors listed in Section F.5.1 is detected. When the generation process is complete the utility displays the report print window.

F.5.1 Dictionary Generation Error Messages

The error messages described in this section are displayed at the baseline when encountered by the dictionary generation program. The generation process is terminated on detection of the first error, which you must correct before trying again.

Dictionary file is full or I/O error

> You have specified too small a dictionary file. Execute the generation option again using a larger dictionary file size. Note that the default is 150 Kbytes which is usually enough.

Field must be numeric - See field *field2*

> GVF/GVA *field2* must be numeric.

GVA fields not all placed at end of record

Move the GVA fields so that they form a contiguous set at the end of the list of fields.

GVF field declared after 125th field

You have more than 125 fields in the record. Move the GVF fields so that they occur before the 125th position in the list of fields.

Inconsistent global field definition - See fields *field1 field2*

The picture clause of global *field1* in the record being processed is different from that of the same global *field2* as defined elsewhere in the meta-dictionary.

Index key length exceeds 47 bytes

The fields which make up the segments of the index being processed exceed 47 bytes, which is invalid.

Index key is composed of more than 8 segments

The index being processed has more than 8 segments, which is invalid.

Index key segment occurs past 125th field in record

You have more than 125 fields in the record. Move the fields used as index segments so that they occur before the 125th position in the list of fields.

Invalid GVA field - also a GVF/index segment - See field *field2*

GVA *field2* may not also be a GVF field or be used as an index segment.

Invalid GVF relation - See field *field2*

The GVA/GVF field corresponding to GVF/GVA *field2* is either missing or has an inconsistent picture clause.

Invalid linkage - Masters must be declared before all servants

The record currently being processed is linked to a master that follows it in the meta-dictionary, which is invalid.

Invalid primary index key definition

Move the fields making up the segments of the primary key so that they form a contiguous set at the beginning of the field list.

Key extract area exceeds 256 bytes

The GVF, master access key and index key fields exceed 256 bytes, which is invalid. See Section 2.10.3.

Linked master is missing or has invalid primary index

Check that the masters linked to the record currently being processed are present and have a valid primary index.

Link field is missing or defined after 125th field

A master access key field is either missing or does not occur within the first 125 fields in the field list, which is invalid.

Meta-dictionary contains more than 36 record definitions

No Speedbase database may have more than 36 record types.

Meta-dictionary is corrupt - Please rebuild using $BARBL

Use the Speedbase Presentation Manager rebuild utility $BARBL to rebuild the meta-dictionary.

More than 90 indexes defined in the database

No Speedbase database may have more than 90 indexes.

No records specified in Meta-dictionary

You must define at least one record type.

Record definition contains no fields

Each record type must have defined at least one field.

Record has more than 16 indexes

No record type may have defined more than 16 indexes.

Record has more than 16 linked masters

No record may be linked to more than 16 masters.

Record has more than 32 GVF relations

A record may have no more than 32 GVF-GVA relations.

Record requires more than 64 index/MAK/GVF segments

The fields making up the segments of the key extract area must not exceed 64 in number. See Section 2.10.3.

Sequence field $SEQ may not also have GVF relations

Do not define a sequence field to be a GVF.

Sequence field $SEQ may only occur in 1 index on each record

The sequence field $SEQ may be used in only one index in each record type.

System area fields (GVAs) exceed max length of 127 bytes

The sum of the lengths of the GVA fields on the record being processed plus 4 bytes exceeds 127 bytes, which is invalid.

Zero length index key detected

One of the indexes on the record type being processed has a length of zero, which is invalid.

$SQ index requires $SEQ field [9(9) COMP] as last segment

You must specify the index $SQ so that its last segment is the field $SEQ.

F.6 Clearing the Meta-dictionary

If you wish to clear the meta-dictionary you have been using, as an alternative to creating a new one using the generation utility as described in Section F.7, run the utility, reply with the meta-dictionary name and unit and key <NXT>:

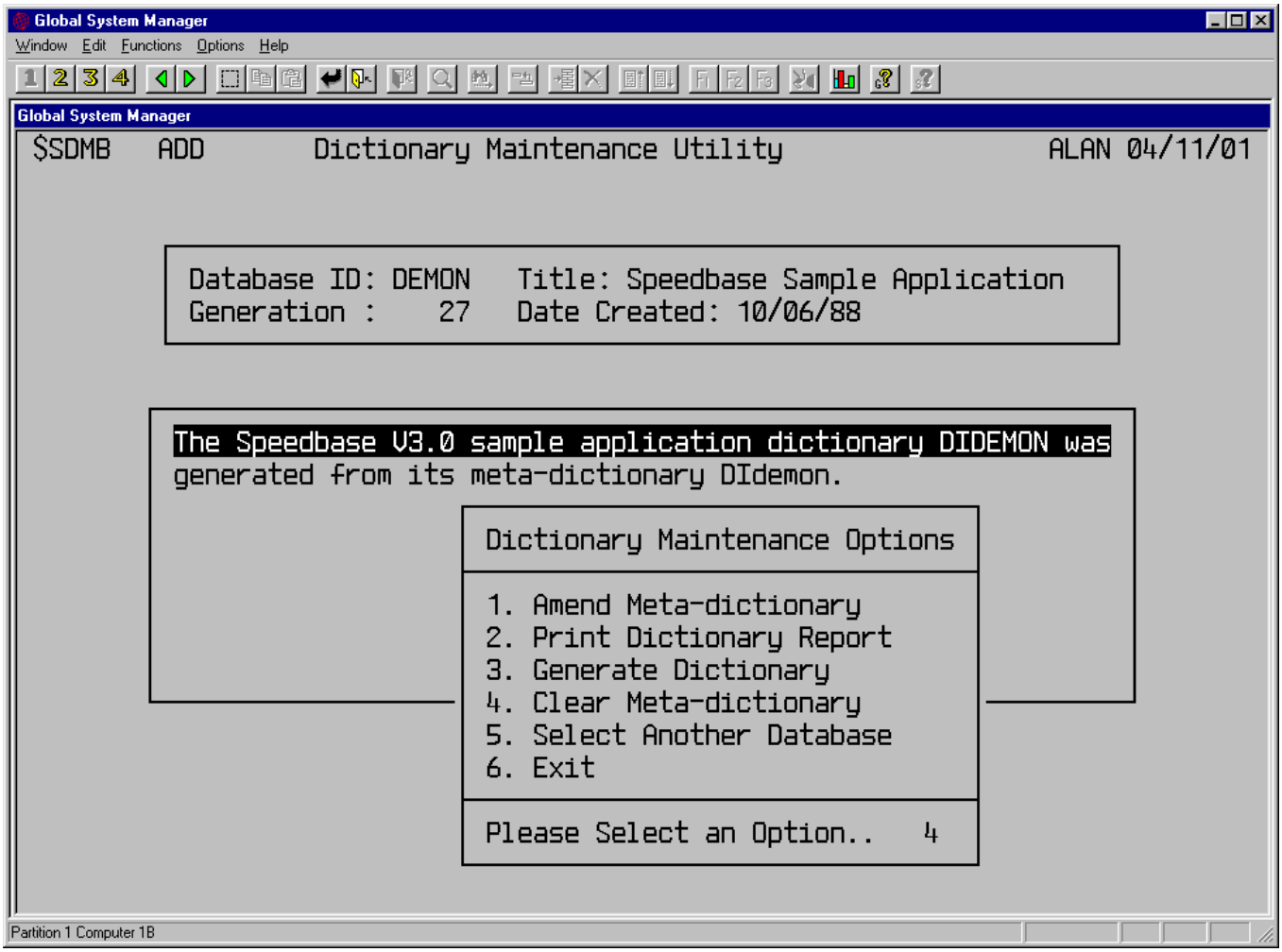


Figure F.6a - Dictionary Maintenance Options

Reply 4 to clear the meta-dictionary. Reply to the baseline prompt "Clear the Meta-dictionary - Are you sure? N" with Y to continue or N to return to the option prompt. If you continue, the utility displays the baseline message "Clearing Meta-dictionary, Please wait...".

When complete the utility displays the baseline prompt "Meta-dictionary cleared. ,". Key <RET> to continue. The utility displays the baseline message "Please now re-organise it using $BARBL". We recommend you do re-organise the meta-dictionary, using the Speedbase rebuild utility $BARBL, in order to obtain the best performance when the meta-dictionary is re-used.

F.7 Creating a Meta-dictionary

The meta-dictionary is stored in the form of a Speedbase database and you may therefore use any of the Speedbase Presentation Manager utilities with it. The meta-dictionary therefore has its own dictionary which is supplied with the Speedbase Development System. This dictionary has the special name DI$DICT.

To create a new, empty meta-dictionary, run the Speedbase Presentation Manager generation utility $BADGN and reply to the source prompt with the special dictionary name DI$DICT, and its unit:

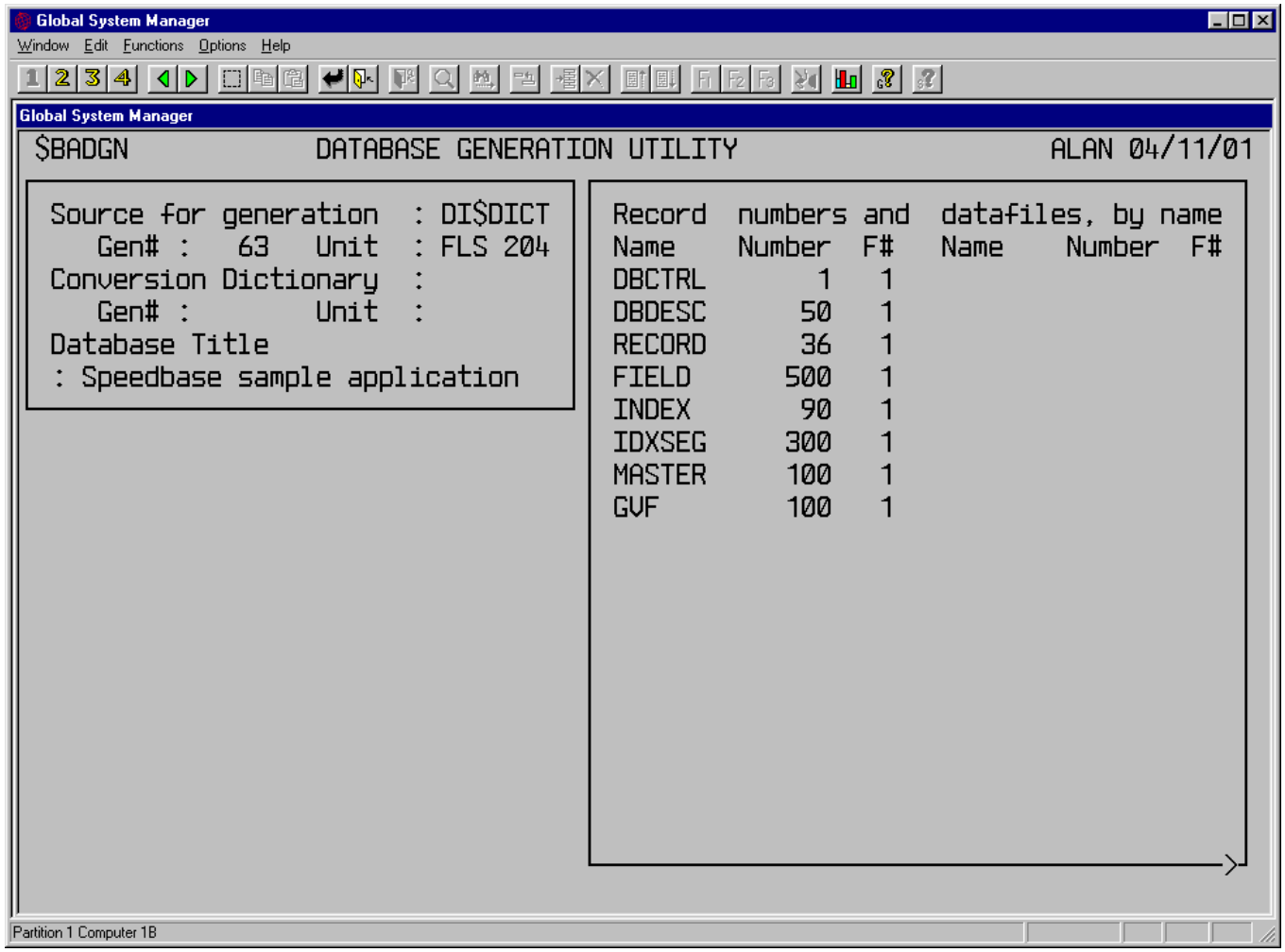


Figure F.7a - Creating the Meta-dictionary

The record numbers are initially set to the default of fifty and you should change them to those shown in Figure F.7a, or to other suitable numbers. When you have done so, key <NXT> and the file allocation window is displayed, as shown in Figure F.7b. Reply to the name prompt with the name you wish to use for the meta-dictionary.

For example, if you are creating the meta-dictionary for the sample application, as suggested in Section F.2, you should reply "demon", to create the meta-dictionary DIdemon. Once the generation process is complete the rebuild utility runs automatically to create the index file required. Once the rebuild is complete the meta-dictionary is ready for use.

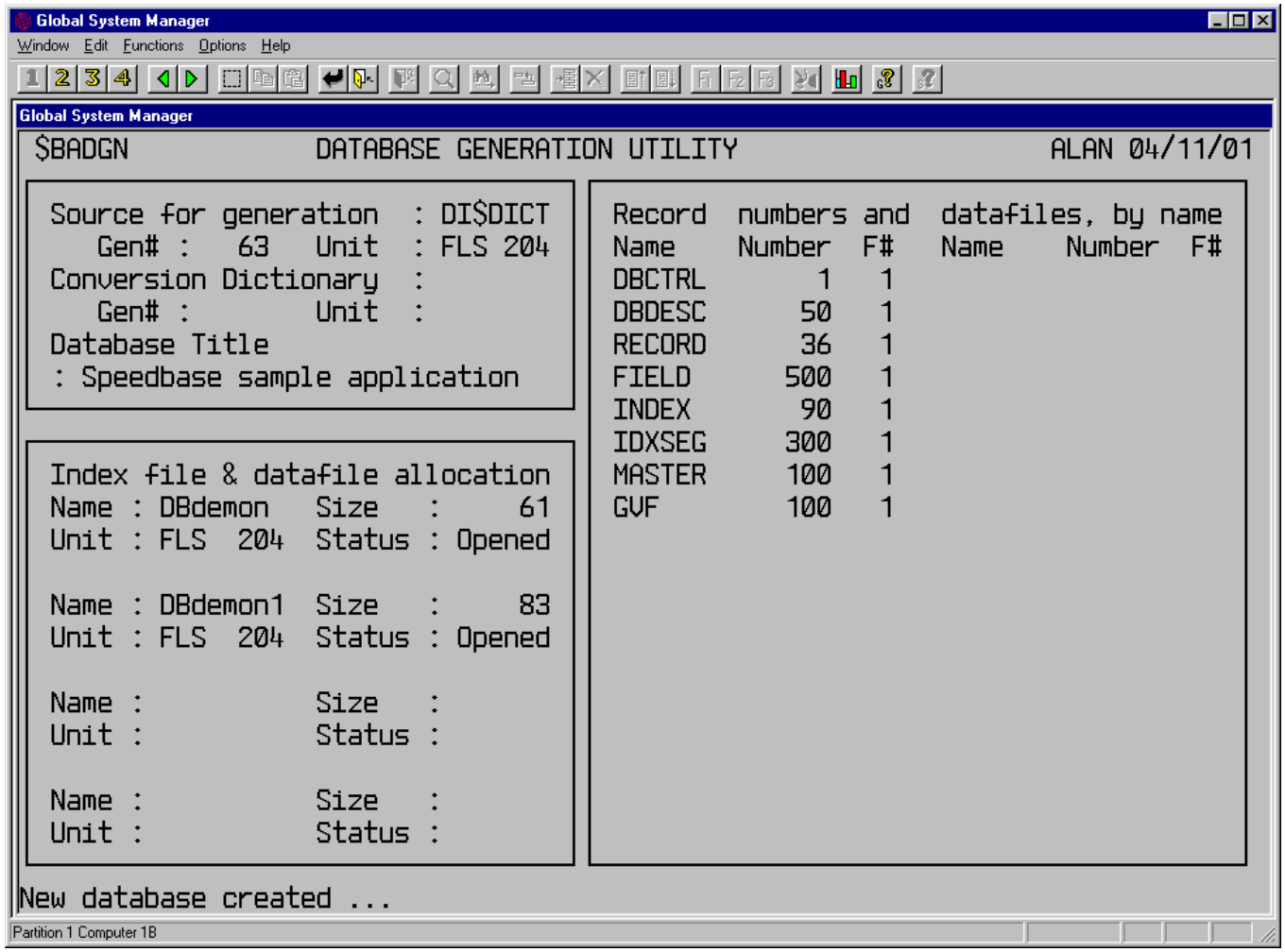


Figure F.7b - Allocating the Meta-dictionary Files

F.8 Auto-Sequence Indexes

There are circumstances in which the sequence of records in a database is important. In the meta-dictionary database, for example, the order of the record-type records is important. In the field record, the actual position of a field relative to other fields is important, if for example the field is a primary index segment. In an invoicing application the order of the detail lines within a particular invoice is likely to be of significance.

To ensure that records are kept in a particular order you can make use of a special index known as an auto-sequence index. This feature operates using a 9(9) COMP sequence number field, which is optionally incremented by the database manager when a new record is written. The sequence number field must reside on the data-record, and must therefore be declared when defining the database dictionary. The field name must be of the form:

rt$SEQ

where *rt* is the record-type. $SEQ must be appended to a special index which is also defined using the database dictionary. The index name assigned to this index must be:

rt$SQ

It is important to note that $SEQ must be the last segment defined for the index. Furthermore, $SEQ may only be used within this particular index, and may not be a GVF or GVA field. The index is otherwise a normal Speedbase index, and may therefore be used with the usual I/O operations. When a record is written to the database, the database manager examines the $SEQ field. If it is negative, it searches for the highest sequence number so far allocated. It adds 65536 (i.e. 2^16) to this sequence number before the record is written. **Important Note: $SEQ must be set negative prior to the I/O operation.**

If $SEQ contains any positive value, no special processing takes place, and the record is written unchanged. Note that $SEQ is also examined during REWRITE as well as WRITE operations. Setting $SEQ negative prior to a rewrite operation causes the next sequence number to be assigned as before. Note that the window manager maintains the rt$SEQ field automatically for I/O operations resulting from a window process.

A record may be inserted into a sequence by initialising $SEQ to an appropriate value prior to writing or re-writing it. This facility is made use of by the Speedbase Presentation Manager window manager move function, <MOV>. When you key <MOV>, to move a field in the meta-dictionary for example, the record to be moved is first removed from the window and may then be inserted in its new position, using the <INS> function. When you key <INS>, the window manager examines the sequence number of the records before and after the insertion point and allocates the moved record a sequence number between them. To do so it uses an algorithm which optimises the number of insertions that may be made after a particular point at the expense of the number that may be made before it.

F.8.1 Programming Note

Sequence numbers are allocated sequentially, and can run out. Where $SEQ is the only segment of the $SQ index, it will be possible to write only 32767 records to the database before the range of available sequence numbers is exhausted.

If the index contains segments before the $SEQ field, it will be possible to write 32767 records for each occurrence of these preceding segments. For example, if the index for a invoice line record is composed of both invoice number and $SEQ, it will be possible to write 32767 invoice line records on **each invoice**.

If a write or re-write operation would exceed the available range, then processing depends on whether $SQ is a primary or secondary index. If $SQ is the primary index, a duplicate primary key condition results, and the I/O operation will have been suppressed.

If $SQ is a secondary index, the record will have been written to the database re-using the highest possible sequence number (i.e. #7FFFFFFF) thus creating a duplicate key. No exception condition results, but the order of the records sequenced by these duplicate keys will be undefined.

If record deletions have occurred, "lost" sequence numbers can be reclaimed by providing an application program to do so, provided that $SQ is not a primary index. It is better, however, to design the application in such a way that the need for this does not arise.

Appendix G - Speedbase Memory Allocation

#0000 to #0500	Debug Area used by $DEBUG
#0500 to #0600	Index Key Extraction Area
#0600 - #3000	Services Module, $BASVC Frame Controller Basic Screen I/O and Help Window Manager Database Manager
#3000 onwards	Application Root Frame
	Application Dependent Frame(s)
	Free Memory
	User Stack
	Database Control Block(s) %BA*dbid*1 %Ba*dbidn*
High memory address	Speedbase System Area $BASYS

Figure Ga - Speedbase Memory Map