TPF Database Facility

Database Administration

Release 1

SH31-0175-09

TPF Database Facility

Database Administration

Release 1

SH31-0175-09

Note!
Before using this information and the product it supports, be sure to read the general information under “Notices”.

Tenth Edition (October 2002)

This is a major revision of, and obsoletes, SH31-0175-08.

This edition applies to Version 1 Release 1 Modification Level 3 of IBM Transaction Processing Facility Database Facility, program number 5706-196, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

IBM welcomes your comments. Address your comments to:

IBM Corporation
TPF Systems Information Development
Mail Station P923
2455 South Road
Poughkeepsie, NY 12601-5400
USA

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

Tables

About This Book

This book will help you:

- Plan and design a database
- Define DSECT macros
- Define DBDEF macros
- Know how DSECT and DBDEF macro statements affect the way TPFDF macros, functions, and utilities work.

In this book, abbreviations are often used instead of spelled-out terms. Every term is spelled out at first mention followed by the all-caps abbreviation enclosed in parentheses; for example, structured programming macro (SPM). Abbreviations are defined again at various intervals throughout the book. In addition, the majority of abbreviations and their definitions are listed in the master glossary in *TPFDF Glossary*.

Before You Begin

Before using this book, see *TPFDF General Information* for an overall understanding of the TPFDF product.

Who Should Read This Book

This book is intended for database administrators, application programmers, and system programmers who are currently working with Transaction Processing Facility (TPF) Version 4 Release 1 (or a subsequent release), or Airline Control System Version 2 (ALCS V2) systems.

How This Book Is Organized

The body of this book is divided into three parts. The first part is a tutorial for planning and designing a database. The second part gives the parameter descriptions to code the DSECT and DBDEF macros. The third part gives examples and more detailed information about the use of the DSECT and DBDEF macros.

Conventions Used in the TPFDF Library

The TPFDF library uses the following conventions:

Typography	Examples of Usage
italic	Used for important words and phrases. For example: A *database* is a collection of data. Used to represent variable information. For example: Enter **ZUDFC DISPLAY ID-***fileid*, where *fileid* is the file identifier (ID) of the file for which you want statistics.
bold	Used to represent keywords. For example: Enter **ZUDFC HELP** to obtain help information for the ZUDFC command.

Typography	Examples of Usage

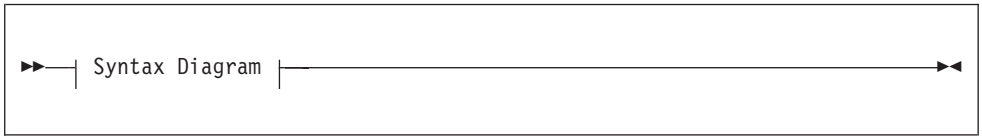
monospaced	Used for messages and information that displays on a screen. For example:
	`PROCESSING COMPLETED`
	Used for C language functions. For example:
	`dfcls`
	Used for examples. For example:
	`ZUDFC DISPLAY ID-J5`

bold italic	Used for emphasis. For example: You ***must*** type this command exactly as shown.
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord=*option*

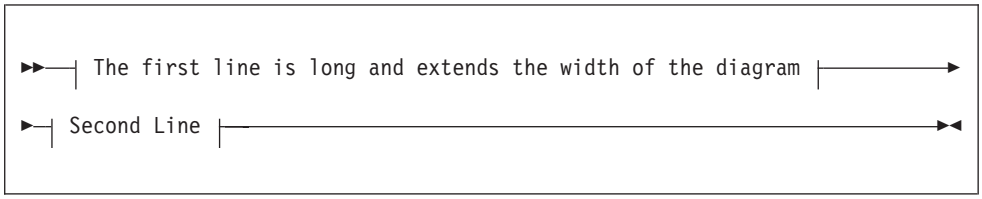
How to Read the Syntax Diagrams

This section describes how to read the syntax diagrams (informally called *railroad tracks*) used in this book.

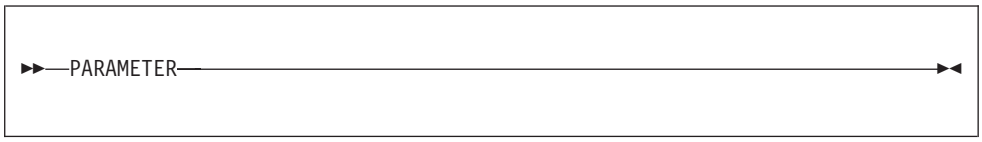
- Read the diagrams from left-to-right, top-to-bottom, following the main path line. Each diagram begins on the left with double arrowheads and ends on the right with 2 arrowheads facing each other.



- If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.

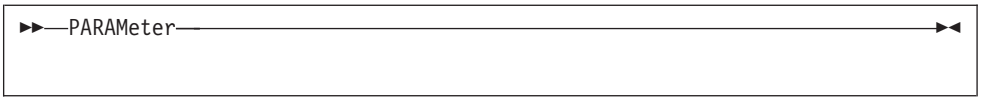


- A word in all uppercase is a parameter that you must spell ***exactly*** as shown.



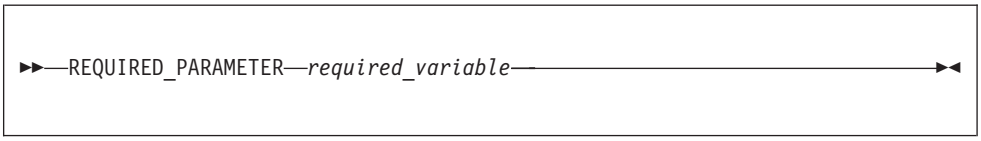
- If you can abbreviate a parameter, the optional part of the parameter is shown in lowercase. (You must type the text that is shown in uppercase. You can type none, one, or more of the letters that are shown in lowercase.)

  Note: Some TPF commands are case-sensitive and contain parameters that must be entered exactly as shown. This information is noted in the description of the appropriate commands.

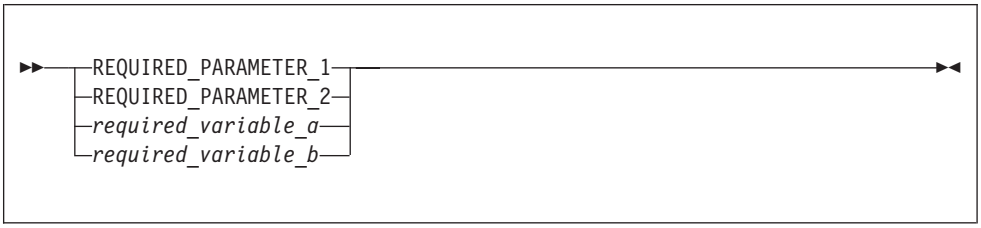


- A word in all lowercase italics is a *variable*. Where you see a variable in the syntax, you must replace it with one of its allowable names or values, as defined in the text.

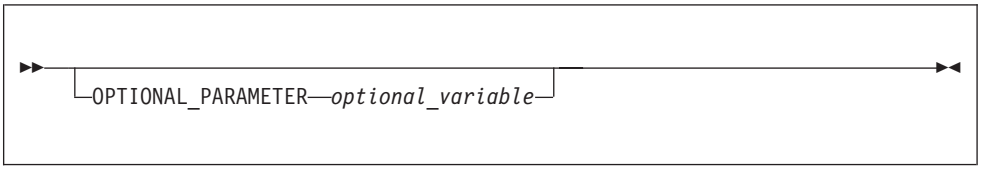
- Required parameters and variables are shown on the main path line. You must code required parameters and variables.



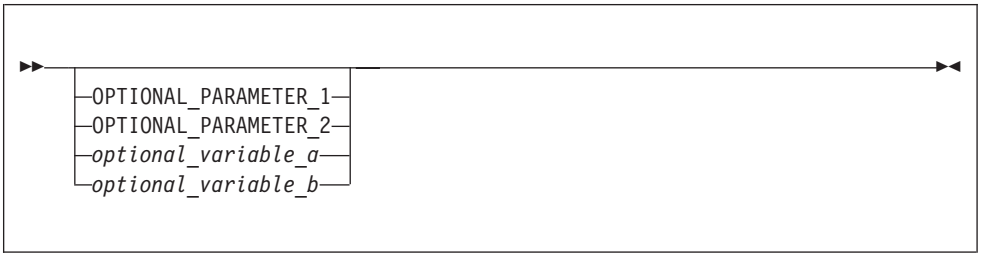
- If there is more than one mutually exclusive required parameter or variable to choose from, they are stacked vertically.



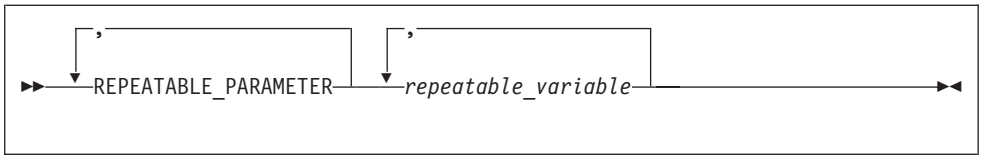
- Optional parameters and variables are shown below the main path line. You can choose not to code optional parameters and variables.



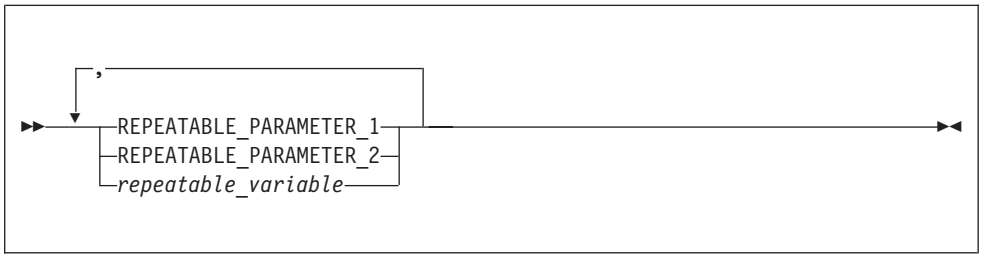
- If there is more than one mutually exclusive optional parameter or variable to choose from, they are stacked vertically below the main path line.



- An arrow returning to the left above a parameter or variable on the main path line means that the parameter or variable can be repeated. The comma (,) means that each parameter or variable must be separated from the next parameter or variable by a comma.

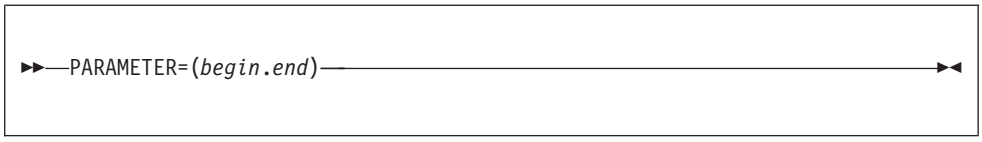


- An arrow returning to the left above a group of parameters or variables means that more than one can be selected, or a single one can be repeated.



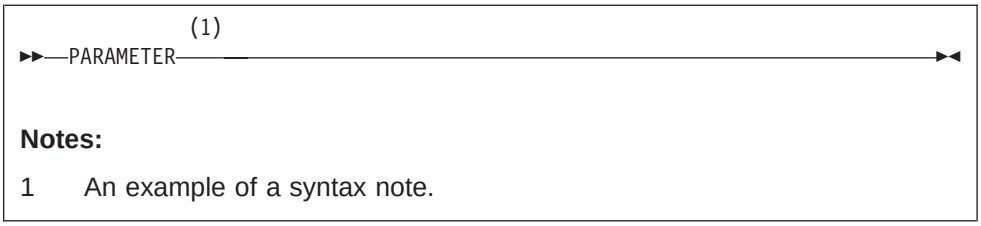
- If a diagram shows a blank space, you must code the blank space as part of the syntax. In the following example, you must code **PARAMETER** *variable*.

- If a diagram shows a character that is not alphanumeric (such as commas, parentheses, periods, and equal signs), you must code the character as part of the syntax. In the following example, you must code **PARAMETER=(***begin***.***end***)**.

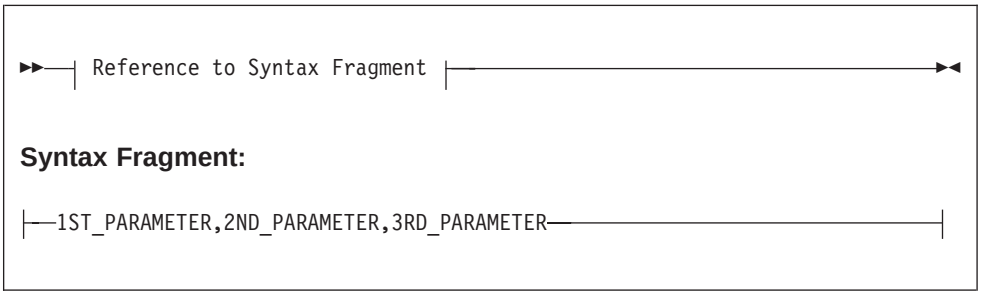


- Default parameters and values are shown above the main path line. The TPF system uses the default if you omit the parameter or value entirely.

- References to syntax notes are shown as numbers enclosed in parentheses above the line. Do not code the parentheses or the number.



- Some diagrams contain *syntax fragments*, which serve to break up diagrams that are too long, too complex, or too repetitious. Syntax fragment names are in mixed case and are shown in the diagram and in the heading of the fragment. The fragment is placed below the main diagram.



Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

IBM TPF Database Facility (TPFDF) Books

- *TPFDF General Information*, GH31-0177
- *TPFDF Installation and Customization*, GH31-0178
- *TPFDF Programming Concepts and Reference*, SH31-0179

IBM Transaction Processing Facility (TPF) Books

- *TPF System Generation*, SH31-0171.

Online Information

- *TPFDF Commands*
- *TPFDF Glossary*
- *TPFDF Messages (System Error, Online, Offline)*
- *TPFDF Utilities*.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, use one of the methods that follow. Make sure you include the title and number of the book, the version of your product and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send your comments electronically, do either of the following:
  - Go to http://www.ibm.com/tpf/pubs/tpfpubs.htm.

    There you will find a link to a feedback page where you can enter and submit comments.
  - Send your comments by e-mail to tpfqa@us.ibm.com
- If you prefer to send your comments by mail, address your comments to:

  IBM Corporation
  TPF Systems Information Development
  Mail Station P923
  2455 South Road
  Poughkeepsie, NY 12601-5400
  USA
- If you prefer to send your comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 +9788

Part 1. Tutorial for Planning and Designing a Database

Organizing a Database

The following describes the process of data normalization. Normalization is a method of logical data organization that minimizes data redundancy and maximizes data independence.

Data normalization is commonly used in the design of relational databases. Although a TPFDF database is hierarchical, it can still benefit from data normalization in the first stage of its design. In later stages, you will need to move away from rigid normalization to achieve the performance standards necessary for a high-volume transaction processing system. These stages are described in Optimizing the Database Design and Mapping Tables to TPFDF Files.

Normalization reduces the data to a minimal form, providing a clearer path for subsequent stages in the design process. The normalization process is always preceded by an analysis of the data to be stored. For now, it is assumed that you have already determined the following:

- The entities for which data will be stored
- The attributes (data fields) that will be stored for each entity.

Note: The mapping of tables to physical files is discussed in "Mapping Tables to TPFDF Files" on page 25.

In the following, you can assume that each table will map to a physical TPFDF file, and each row in a table to a TPFDF logical record (LREC).

TPFDF databases have the following characteristics:

- Many users
- A large number of files
- Different means of accessing files
- A high transaction rate.

Consider these characteristics as you work through the design of your TPFDF database.

Normalization

The purpose of normalization is to create tables (also known as relations) for all the entities on which the system holds data. Through the process of normalization, the structure of a table is progressively refined through first, second, and finally third normal form. The following guidelines may be of some help as you develop your tables:

First normal form
: Every attribute in the table must have only one value for each row. Each row must be independent of all the others.

Second normal form
: Every attribute in the table must be either directly or indirectly dependent on the primary key.

Third normal form
: Every attribute in the table must be directly dependent on the primary key. There must be no unnecessary duplication of data in the table.

Note: The terms *primary key* and *dependency* are important in data normalization.

Primary Key

In the context of data normalization, the purpose of the primary key is to identify a unique row in a table. The value of the primary key is always unique.

Each table can have only one primary key, though the primary key itself may be made from more than one attribute.

Unless otherwise stated, the attributes making up the primary key are shown in bold type in the tables.

Note: The primary key that identifies a table in a database is not the same as the TPFDF LREC ID (also known as a primary key).

The concept of the primary key is explained in more detail with the example tables provided later in this information.

Dependency

Dependency refers to the relationship of an attribute with the primary key of a table.

To say that an attribute is dependent on the primary key means that, given a particular value for the primary key, there is only one corresponding value for that attribute.

An attribute may be either directly or indirectly dependent on the primary key.

A directly dependent attribute depends on the value of the primary key itself. An indirectly dependent attribute depends on the value of another attribute, which is itself dependent on the primary key. This indirect dependency is sometimes called *transitive dependency*.

The concept of dependency is further explained with the example tables provided later in this information. See especially “Second Normal Form” on page 5.

Business Application

Assume that you want to keep passenger and flight information on the database.

To do this, you could have a single table containing rows holding all the necessary information resulting from your data analysis. A table of this kind is shown in Table 1.

Table 1. Flight Table (Nonnormalized Form)

Date	**Flight number**	Start	Destination	Aircraft type	**Seat number**	**Seat class**	Passenger number	Passenger name	Passenger address	Passenger facts
Da1	***Fl1***	St1	De1	At1	***Se1*** ***Se2*** ***Se3*** ***Se4***	***Cl2*** ***Cl2*** ***Cl2*** ***Cl3***	Pn1 Pn2 Pn3 Pn4	Na1 Na2 Na3 Na4	Ad1 Ad2 Ad3 Ad4	Ft1 Ft2
Da1	***Fl2***	St2	De2	At1	***Se5***	***Cl2***	Pn3	Na3	Ad3	

This table is not in normalized form and has several undesirable features. For example:

- The size of each row is not fixed. As the number of booked seats increases, the number of attribute values in the row increases. The indeterminate size of this row would create difficulties if the table were mapped to a physical file.
- Some of the data (for instance, the name and address of customer Na3) is repeated unnecessarily. The same data is likely to be repeated in other tables as well. In practice, duplicated data is often inconsistent. For example, a customer's name and address could be recorded slightly differently at different times.

  Restructuring the table into first normal form eliminates the first of these problems. You will need to progress to a higher level of normalization to eliminate the second.

First Normal Form

To get a table into first normal form, remove multiple occurrences of attribute values from the same row by creating a new row for each value. This ensures that every attribute in the table has only one value for each row.

Table 2 is in first normal form and contains the same data as the nonnormalized table.

Table 2. Flight Table (Nonnormalized Form)

Date	**Flight number**	Start	Destination	Aircraft type	**Seat number**	**Seat class**	Passenger number	Passenger name	Passenger address	Passenger facts
Da1	***Fl1***	St1	De1	At1	***Se1*** ***Se2*** ***Se3*** ***Se4***	***Cl2*** ***Cl2*** ***Cl2*** ***Cl3***	Pn1 Pn2 Pn3 Pn4	Na1 Na2 Na3 Na4	Ad1 Ad2 Ad3 Ad4	Ft1 Ft2
Da1	***Fl2***	St2	De2	At1	***Se5***	***Cl2***	Pn3	Na3	Ad3	

Second Normal Form

To get a table into *second normal form*, remove all attributes that are not dependent, either directly or indirectly, on the value of the primary key and put them into other tables.

In Table 2, the attribute *destination* directly depends on the primary key. Given the value of the primary key, especially the date and flight number, you can see that the aircraft will be flying to a particular destination. Without the primary key value, it would not make sense to speak of a flight destination at all.

In contrast, the *seat class* attribute is only indirectly dependent on the primary key. It really depends on the *aircraft type* and *seat number* attributes. *Aircraft type* is not part of the primary key, though it is itself dependent on the primary key. Because *seat class* is dependent on *aircraft type*, it is said to be *indirectly dependent* on the primary key.

Removing Independent Attributes

Looking back to the flight table in first normal form (Table 2), you can see that it contains some attributes that are not dependent, either directly or indirectly, on the primary key. The primary key in this table is the combination of the *date*, *flight number*, and *seat number* attributes.

Even if the flight table's primary key had no value at all, the following attributes would still be meaningful:
- Passenger name
- Passenger address

- Passenger facts.

Because they are not dependent on the primary key, you can remove these attributes from the flight table and put them into a separate table of their own.

Table 3 shows an example of such a table. Note that the primary key for this table is *passenger number*. This has been included in the new table because its value is always unique. For example, there may be more than one passenger named Smith, or more than one vegetarian passenger, but there can never be more than one passenger number Pn1.

Because it is unique, *passenger number* becomes the primary key of the new table. It also remains in the flight table because it is dependent on the primary key.

Table 3 is already in second normal form because every attribute in the table is either directly or indirectly dependent on the primary key.

Table 3. Passenger Table

Passenger number	Passenger name	Passenger address	Passenger facts
Pn1	Na1	Ad1	Ft1
Pn2	Na2	Ad2	Ft2
Pn3	Na3	Ad3	
Pn4	Na4	Ad4	

Now that you have removed the attributes that are independent on the primary key and have put them in the passenger table (Table 3), the flight table is in second normal form, as you can see in Table 4. The primary key is still the combination of the *date, flight number,* and *seat number* attributes that are shown in bold type.

Table 4. Flight Table (Second Normal Form)

Date	**Flight number**	Start	Destination	Aircraft type	**Seat number**	Seat class	Passenger number
Da1	***Fl1***	St1	De1	At1	***Se1***	Cl2	Pn1
Da1	***Fl1***	St1	De1	At1	***Se2***	Cl2	Pn2
Da1	***Fl1***	St1	De1	At1	***Se3***	Cl2	Pn3
Da1	***Fl1***	St1	De1	At1	***Se4***	Cl3	Pn4
Da1	***Fl2***	St2	De2	At1	***Se5***	Cl2	Pn3

Third Normal Form

The flight table (Table 4) has been improved by being put into second normal form, but it still contains some indirect dependencies, and also some duplicate data.

For example, the *seat class* attribute depends on *aircraft type* and *seat number*. It does not depend directly on the other two attributes of the primary key. Because of this, you can remove *seat class* and record it in a separate table.

The new table will also include the *aircraft type* and *seat number* attributes. Together, they form the primary key of the new table, which is shown in Table 5 on page 7.

Table 5. Aircraft Table

Aircraft type	**Seat number**	Seat class
At1	***Se1***	Cl2
At1	***Se2***	Cl2
At1	***Se3***	Cl2
At1	***Se4***	Cl3
At1	***Se5***	Cl2

The primary key here is the combination of *aircraft type* and *seat number*.

The indirect dependency has now been removed from the flight table, but with the table in second normal form, there is still some duplication, as you can see in Table 6.

Table 6. Flight Table (Second Normal Form)

Date	**Flight number**	Start	Destination	Aircraft type	**Seat number**	Passenger number
Da1	***Fl1***	St1	De1	At1	***Se1***	Pn1
Da1	***Fl1***	St1	De1	At1	***Se2***	Pn2
Da1	***Fl1***	St1	De1	At1	***Se3***	Pn3
Da1	***Fl1***	St1	De1	At1	***Se4***	Pn4
Da1	***Fl2***	St2	De2	At1	***Se5***	Pn3

Because the values for *seat number* and *passenger number* must be unique for each row, the table must contain a separate row for each of these values. To accommodate this, the other values in the table must be duplicated. For example, flight number Fl1 is repeated four times when only once would be enough.

Because *seat number* is already held in the aircraft table (Table 5) and *passenger number* is held in the passenger table (Table 3 on page 6), and there is no indirect dependency through either of these attributes, you can remove them both from the flight table.

Because you need a table to record the passengers who are on any particular flight, you could place these removed attributes in a seat table. This table would show the seats booked on the flight and the passenger number for the person booked on each seat. Table 7 is an example of this kind of table:

Table 7. Seat Table

Date	**Flight number**	**Seat number**	Passenger number
Da1	***Fl1***	***Se1***	Pn1
Da1	***Fl1***	***Se2***	Pn2
Da1	***Fl1***	***Se3***	Pn3
Da1	***Fl1***	***Se4***	Pn4
Da1	***Fl2***	***Se5***	Pn3

The primary key in this table is the combination of *date, flight number*, and *seat number*.

The flight table is now in third normal form. Each of its attributes is directly dependent on the primary key and there is no unnecessary duplication of data in the table.

As you can see in Table 8, the value for *date* is the same in both rows of the table. This duplication is necessary because there will certainly be more than one flight each day.

Table 8. Flight Table (Third Normal Form)

Date	**Flight number**	Start	Destination	Aircraft type
Da1	***Fl1***	St1	De1	At1
Da1	***Fl2***	St2	De2	At1

The *passenger number* attribute has been removed from the flight table along with *seat number* and *seat class* because *passenger number* was dependent on *seat number* and, without the seat number, no passenger number can be assigned.

The primary key in the flight table is now the combination of the *date* and *flight number* attributes.

Resulting Tables

You now have four complete and related tables (Table 9–Table 12) in third normal form:

- Flight table
- Passenger table
- Aircraft table
- Seat table.

Table 9. Flight Table (Third Normal Form)

Date	**Flight number**	Start	Destination	Aircraft type
Da1	***Fl1***	St1	De1	At1
Da1	***Fl2***	St2	De2	At1

Table 10. Passenger Table

Passenger number	Passenger name	Passenger address	Passenger facts
Pn1	Na1	Ad1	Ft1
Pn2	Na2	Ad2	Ft2
Pn3	Na3	Ad3	
Pn4	Na4	Ad4	

Table 11. Aircraft Table

Aircraft type	**Seat number**	Seat class
At1	***Se1***	Cl2
At1	***Se2***	Cl2
At1	***Se3***	Cl2
At1	***Se4***	Cl3
At1	***Se5***	Cl2

Table 12. Seat Table

Date	**Flight number**	**Seat number**	Passenger number
Da1	***Fl1***	***Se1***	Pn1
Da1	***Fl1***	***Se2***	Pn2
Da1	***Fl1***	***Se3***	Pn3
Da1	***Fl1***	***Se4***	Pn4
Da1	***Fl2***	***Se5***	Pn3

Duplicating Data across Tables

Looking back at the four normalized tables (Table 9–Table 12), you may notice that there is some duplication of data across them. For example, the *passenger number* attribute is held in both the passenger table and the seat table. *Seat number* is held in the seat table and the aircraft table, and there are further duplications as well.

This duplication occurs in normalized tables because they are generally being developed to be used in a relational database. Relational systems use these common attributes as links between one table and the next. The links provide paths around the database so that every table can access the data held in every other table. This means that, apart from the linking attributes, no data needs to be duplicated in the database.

In the example hierarchical database discussed in this publication, links of this kind are not needed because the tables are joined by pointers. Pointers provide a means of linking one table or file with another. Use the DBIDX and DBDIX TPFDF macros to maintain pointers.

Optimizing the Database Design

As you have seen in “Normalization” on page 3, the process of normalization has imposed a preliminary order on your data. Unnecessary duplications have been removed and the data is clearly set out in a readily comprehensible form.

However, tables that are rigidly normalized do not always produce the best results from a performance point of view. Retrieval speeds can often be improved when the same data is held in different tables.

Figure 1 shows the current structure of the database.

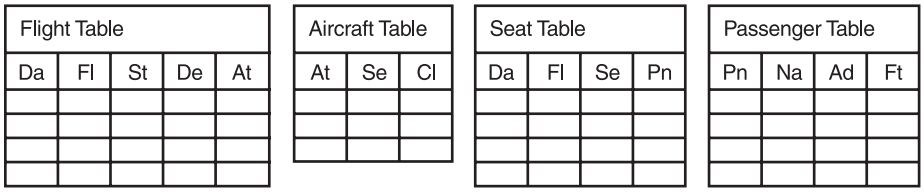


Figure 1. Four Normalized Tables

Duplicating Data to Improve Performance

Because duplicating data can improve retrieval speeds, you may choose to reintroduce some data duplication in the tables you have been working on.

Note: Data should be duplicated for performance reasons only. Data duplication can result in extra updating, which in turn requires more I/O processing. You must ensure that the extra I/O processing for updating does not outweigh the I/O processing saved through the duplication itself. Because I/O processing can represent a major part of the lifetime of a transaction lifetime, it is important to minimize I/O processing.

Data duplication can also lead to inconsistent data being held. However, some data fields are more suitable for duplication than others. There is less risk of data inconsistency if you duplicate fields whose values do not change often. For example, a passenger’s name is unlikely to change whereas seat availability changes with every seat sold.

You must balance these factors against the potential improvement in retrieval speeds.

In the examples that follow, you can see how selected duplication of data can improve the performance of your database.

Assessing the Normalized Tables

In this section, whether the existing design performs adequately in a realistic setting is assessed. The design is changed wherever necessary to ensure a good real-time performance level.

Note: When developing your design, you must also consider frequency of operation. A 10% performance gain in an operation performed several times per second is better overall than a 50% gain in a daily operation.

The following list shows six common requirements for an airline reservation system:

- Checking seat availability
- Booking a passenger on a flight
- Displaying all passengers booked on a flight
- Displaying all flights booked for a passenger
- Canceling a passenger's booking on one flight or more
- Displaying the configuration of an aircraft.

The four tables in Figure 1 on page 11 show that all this information can be extracted from them. However, if you consider the tables carefully, you can see that some of these operations would involve substantial I/O processing. For example, to display all flights booked for a passenger, you would need to read through the entire seat table for each flight. Because of this, it is more sensible to reintroduce some duplication of data in the tables. This data must be carefully selected. There should be no arbitrary data duplication.

In the following pages, each of the common requirements for an airline reservation system is analyzed.

Checking Seat Availability

Before booking a seat, check whether there is a seat of the required class available on the flight specified. To check seat availability using these tables:

1. Read the flight table to find the aircraft type (*At*).
2. Read the aircraft table to find a seat number (*Se*) in the required class.
3. Read the *passenger number* field (*Pn*) in the seat table to find whether that seat has already been booked.
4. If that seat number is already booked, read the aircraft table again to find the next seat in the required class.
5. Repeat steps 3 and 4 until an available seat is found, or until all seats in the required class are found to be booked.

This read process is shown in Figure 2 on page 13. You can see the seat table and the aircraft table will probably have to be read many times, which would involve a significant amount of I/O processing. Performance would be much improved if some changes were made to these two tables.

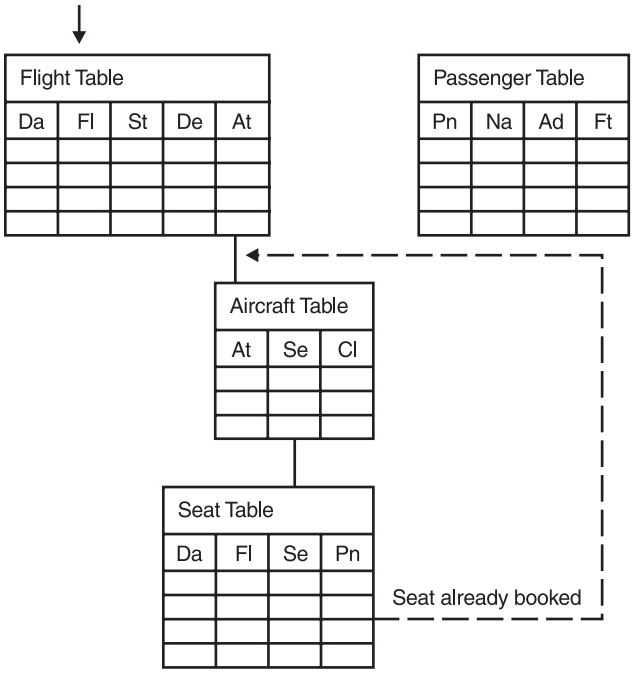


Figure 2. Read Process for Checking Seat Availability Before Optimization

Figure 3 shows how the tables have been changed.

In the aircraft table, the *seat number* (*Se*) attribute has been changed to *seat range* (*Sr*). The table is now more compact because you do not have to store every seat on a separate row.

The *class* (*Cl*) attribute has been duplicated in the seat table. *Availability* (*Av*), in the flight table, is a new attribute.

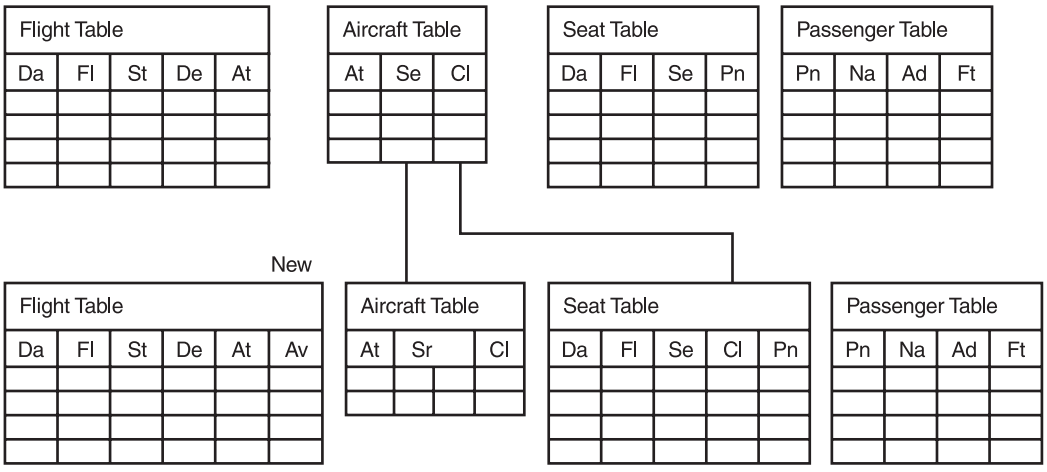


Figure 3. Altering the Tables to Improve Availability Checking

After improving these tables, the revised seat table (shown in Table 13 on page 14) contains all the data needed to check seat availability. Every flight is recorded there, and you can see at once whether a seat has been booked or not.

Table 13. Seat Table (Revised)

Date	**Flight number**	**Seat number**	Class	Passenger number
Da1	***Fl1***	***Se1***	Cl2	Pn1
Da1	***Fl1***	***Se2***	Cl2	Pn2
Da1	***Fl1***	***Se3***	Cl2	Pn3
Da1	***Fl1***	***Se4***	Cl3	Pn4

Booking a Passenger on a Flight

Note: When working through the following example, refer to Figure 3 on page 13.

Now that you have checked seat availability on a particular flight, you can make a booking for the passenger as follows:

1. Add the new details to the passenger table.
2. Add the new details to the seat table.
3. Update availability information in the flight table.

Displaying Passengers Booked on a Flight

To display all the passengers booked on a particular flight:

1. Read the seat table to find every passenger number for a particular flight.
2. Read the passenger table to find the corresponding passenger name for each passenger number.

Figure 4 shows this read process.

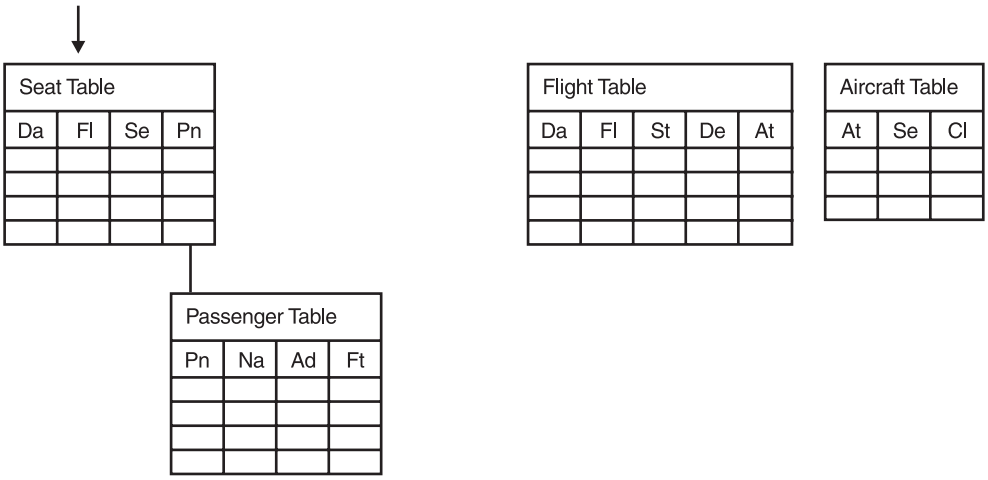


Figure 4. Read Process for Displaying Passengers Booked on a Flight

You can eliminate reading the passenger table if the *passenger name* attribute is duplicated in the seat table. Figure 5 on page 15 shows this duplication.

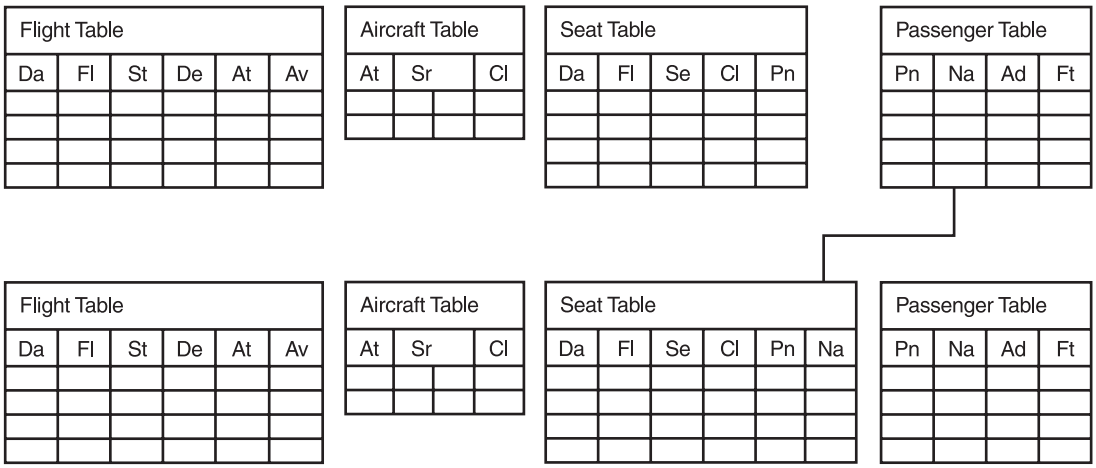


Figure 5. Duplicating Names to Display Passengers Booked on a Flight

After this duplication, the new seat table (Table 14) contains all the data needed to display passengers booked on a flight. You do not need to refer to any other table.

Table 14. Seat Table (Updated)

Date	**Flight number**	**Seat number**	Class	Passenger number	Passenger name
Da1	***Fl1***	***Se1***	Cl2	Pn1	Na1
Da1	***Fl1***	***Se2***	Cl2	Pn2	Na2
Da1	***Fl1***	***Se3***	Cl2	Pn3	Na3
Da1	***Fl1***	***Se4***	Cl3	Pn4	Pn4

Displaying All Flights Booked for a Passenger

To display all the flights booked for a particular passenger, read the seat table (Table 14) for each flight, checking for a match between each flight and your passenger.

You can avoid this I/O intensive search of the seat table if you add *flight* and *date* attributes to the passenger table.

Figure 6 on page 16 shows how *date* (*Da*) and *flight* (*Fl*) have been duplicated in the passenger table.

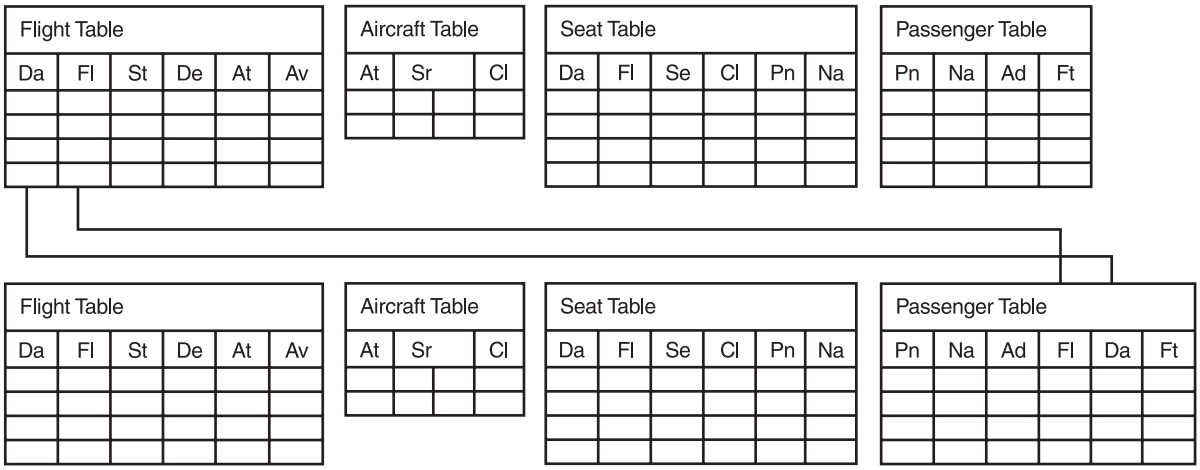


Figure 6. Duplicating Flights and Dates to Improve Flight Display

The revised passenger table (Table 15) shows all the flights booked for each passenger. You do not need to refer to any other table.

Table 15. Passenger Table (Revised)

Passenger number	Passenger name	Passenger address	Flight	Date	Passenger facts
Pn1	Na1	Ad1	Fl1 Fl2 Fl3	Da1	Ft1
Pn2	Na2	Ad2	Fl1	Da1	Ft2
Pn3	Na3	Ad3	Fl2	Da1	
Pn4	Na4	Ad4	Fl1	Da1	

Displaying an Aircraft Configuration

The revised aircraft configuration shows how many seats each aircraft holds in each class. In the original aircraft table (see Figure 1 on page 11), every seat in every aircraft is listed in a separate row.

Now that the *seat number* attribute has been changed to *seat range*, configurations for different aircraft in the same table are shown in the revised aircraft table (Table 16). You can now quickly display the number of seats in each class for each aircraft.

Table 16. Aircraft Table (Revised)

Aircraft type	**Seat range**	Seat class
A1	***Se1-Se12***	Cl2
A1	***Se13-Se120***	Cl3
A2	***Se1-Se4***	Cl1
A2	***Se13-Se23***	Cl2
A2	***Se28-Se70***	Cl2
A3	***Se1-Se23***	Cl2
A3	***Se26-Se40***	Cl2

Canceling Passenger Bookings

To cancel a passenger's booking:

1. Read the seat table to search for the passenger number and name. Delete these if found.
2. After deleting the number and name, continue reading the seat table to find any more occurrences of that number and name. If found, delete these as well.
3. When no more occurrences are found, delete the appropriate details from the passenger table.
4. Finally, delete the appropriate details from the flight table.

Figure 7 shows the process for canceling passenger bookings.

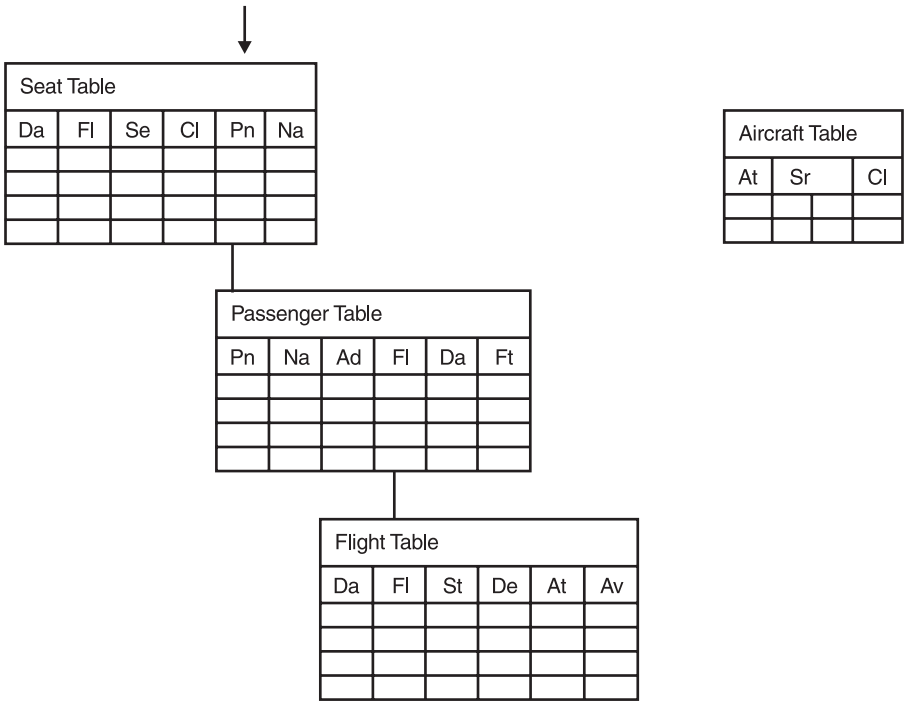


Figure 7. Read Process for Canceling Passenger Bookings

Looking at the process outlined in Figure 7, you can see that it would require substantial I/O processing to read through these three tables to check every flight for every day against a particular passenger number. The *date* and *flight* data duplicated in the revised passenger table (see Figure 6 on page 16) has, in fact, increased updating times because you must now update the passenger table as well as the flight and seat tables.

However, the performance benefits gained for the previous five queries outweigh the losses incurred in this query. Overall, the optimization has improved the performance of the database.

Improving Access to the Data

The duplication of some data in the tables has considerably improved the performance of the database. However, there are still some areas where poor access is slowing down the retrieval of data. You will need to improve these access paths before the database can achieve the performance level required of a real-time system.

The following list shows four common requirements that the database must be able to meet quickly:

- Display passenger information by passenger name or number
- Access flight information
- Access passengers from the seat table
- Access aircraft configurations from the flight table.

The revised tables, as shown in Figure 8, do not yet provide easy access paths for these requirements.

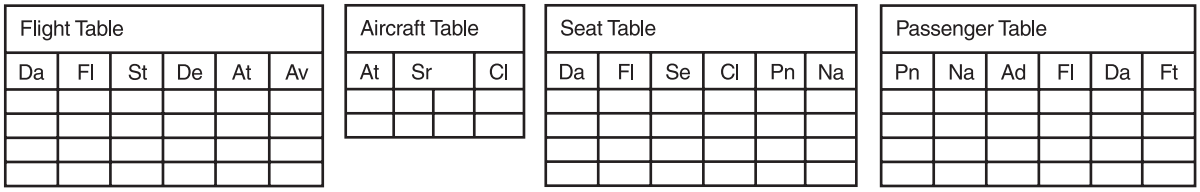


Figure 8. Four Revised Tables

The following pages explain each of the previously outlined requirements and a method of improving access times. See Figure 8 when working through these requirements.

Displaying Passenger Information by Name or Number

Once you have input a passenger name or number, you need to be able to access the passenger table directly. At present, there is no direct access from either of these inputs to the passenger table.

However, the addition of *pointers* would improve access times significantly. Because pointers provide a means of linking one table or file with another, you can use them to achieve direct access to tables. In the TPFDF product, you can use the DBIDX and DBDIX macros to maintain pointers between files. Files containing pointers are index files.

Table 17 and Table 18 directly access the passenger table.

Table 17. Passenger Name Table

Passenger name	**Pointer to passenger table**
Na1	Pointer1
Na2	Pointer2
Na3	Pointer3
Na4	Pointer4

Table 18. Passenger Number Table

Passenger number	Pointer to passenger table
Pn1	Pointer1
Pn2	Pointer2
Pn3	Pointer3
Pn4	Pointer4

Figure 9 shows the two pointer tables (passenger number and passenger name) pointing to the passenger table.

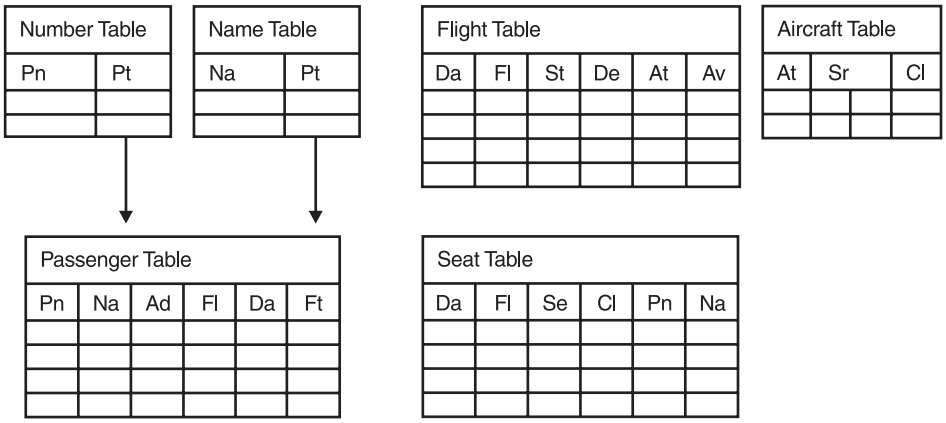


Figure 9. Adding Pointer Tables to Improve Access to the Passenger Table

Accessing Flight Information

Flight information is held in the flight table and the seat table (see Figure 8 on page 18). Between them, these two tables contain all the information you need to know about a flight.

However, at present there is no direct way of accessing the seat table from the flight table. You can overcome this difficulty if you include a pointer in the flight table.

Figure 10 shows how the pointer has created a direct access path from the flight table to the seat table.

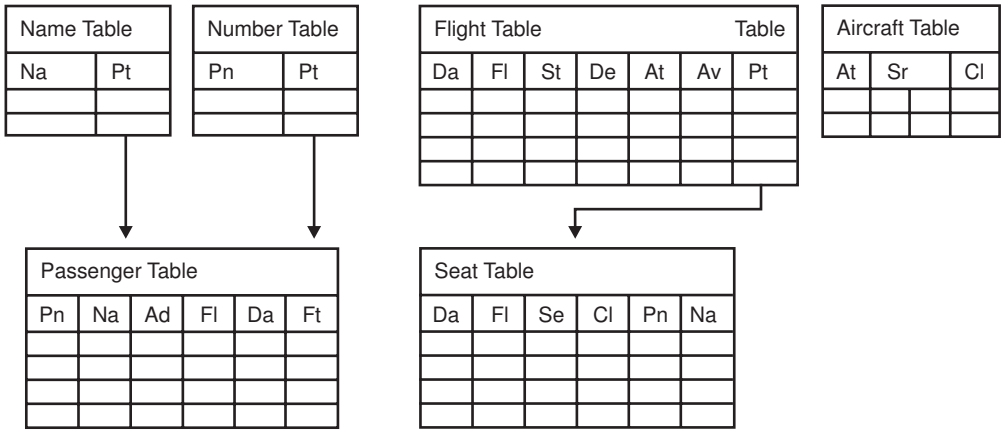


Figure 10. Adding a Pointer to Improve Access between the Flight and Seat Tables

Now that the flight table and the seat table are linked by the pointer, you can remove the resulting data duplication (*date* and *flight*). Because the TPFDF product reads the flight table before reading the seat table, you should remove the duplicated data from the seat table.

Accessing Passengers from the Seat Table

There is still no direct access from the seat table to the passenger table (see Figure 10).

An added pointer in the seat table provides the access path you need.

Figure 11 shows how the added pointer in the seat table creates a direct access path to the passenger table. (Note that the duplicated attributes, *date* and *flight*, have been removed from the seat table, as discussed in “Accessing Flight Information” on page 19.)

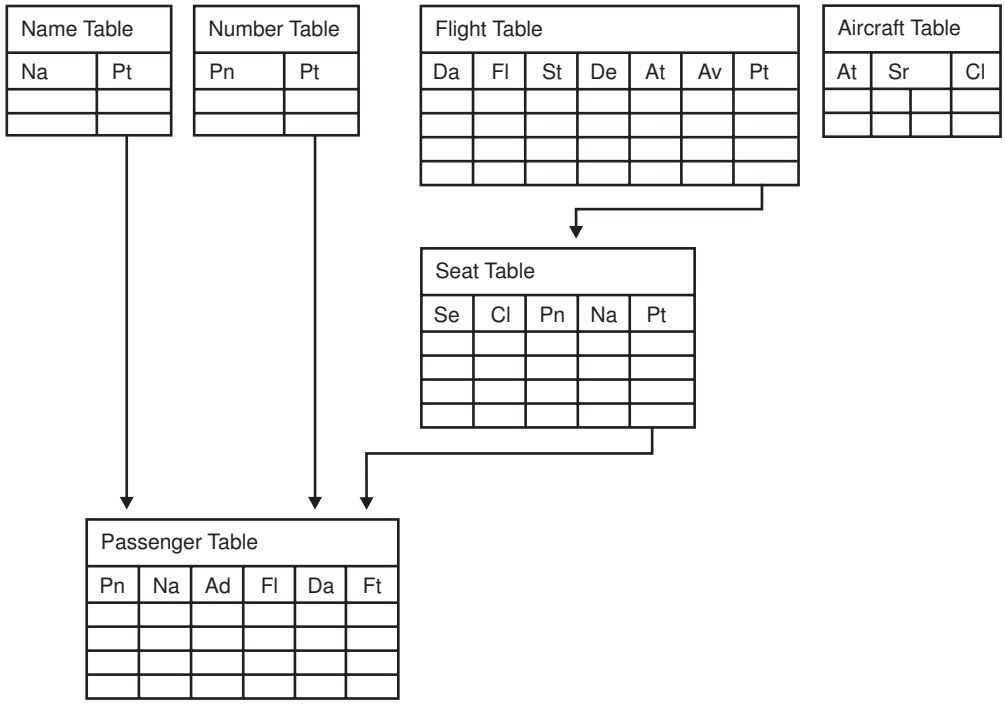


Figure 11. Adding a Pointer to Improve Access between the Seat and Passenger Tables

Accessing Aircraft Configurations from the Flight Table

There is still no direct access from the flight table to the aircraft table (see Figure 11). You can overcome this difficulty by including an additional pointer in the flight table.

Figure 12 shows how the second pointer in the flight table creates a direct access path to the aircraft table.

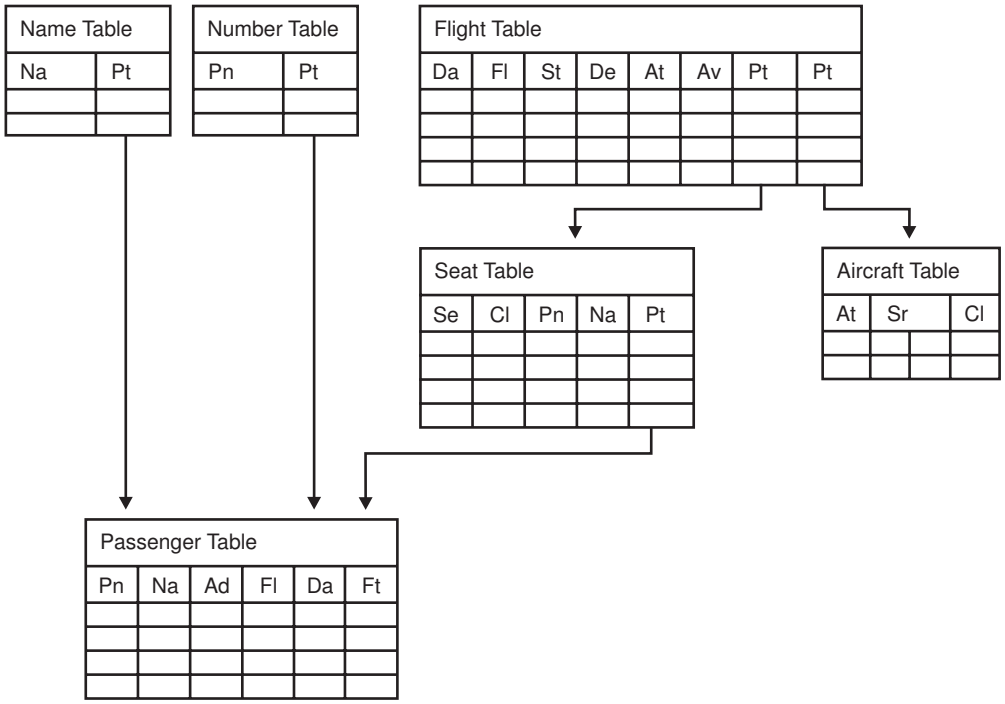


Figure 12. Adding a Pointer to Improve Access between the Flight and Aircraft Tables

Final Database Design Structure

The four original tables (see Figure 1 on page 11) have now been optimized to be used in a real-time database. Two new tables have been added to improve data access by using pointers. These six logical tables are now ready to be mapped to physical TPFDF files.

Figure 13 shows the database tables arranged hierarchically. The tables are shown in their relative sizes. Some attributes (for example, *passenger name*) are longer than others (for example, *date*).

The two pointer tables (passenger name and passenger number) that are at the top left of the diagram contain no detail data. Mapping Tables to TPFDF Files shows how the tables are mapped to TPFDF index files.

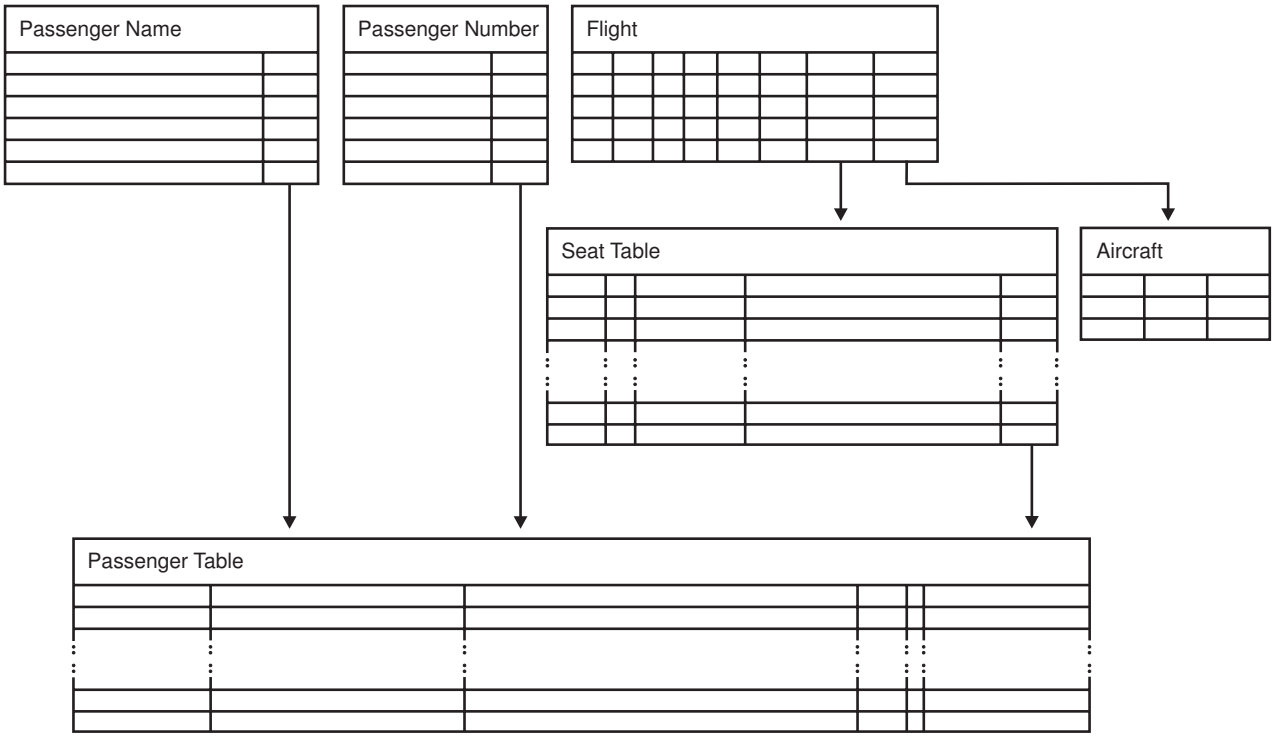


Figure 13. Final Tables Showing the Database Structure

Mapping Tables to TPFDF Files

The following describes how to map the logical tables developed in the previous chapter to physical TPFDF files.

Before working through the following, make sure you are familiar with the concepts and terminology of TPFDF files. For more information about concepts and terminology of TPFDF files, see *TPFDF General Information*.

In the transfer from logical to physical data, some of the terms used will change. Table 19 shows the changes to be aware of.

Table 19. Comparative Terms for Tables and Files

Logical term (tables)	Physical term (files)
attribute	data field
attribute value	data field value
row	LREC
table	file

Before You Begin

The tables used for mapping are those developed in “Organizing a Database” on page 3 and “Optimizing the Database Design” on page 11. They are as follows:

- Passenger name table (index file)
- Passenger number table (index file)
- Aircraft table (detail file)
- Flight table (index file)
- Seat table (intermediate-level index)
- Passenger table (detail file).

Data Requirements

Table 20 shows the data requirements for the tables that are to be stored on the database. The requirements shown here are examples only. They are the result of the initial data analysis carried out before the start of the design process.

Table 20. Data Requirements

Data	Amount
Flights each day	100
Days stored	366
Passengers on each flight (min=50, max=300)	150 (average)
Flights for each passenger (max=20)	3 (average)
Flight classes	3
Aircraft types	50

Note: Because leap years must be accommodated, 366 days can be stored.

Data Field Lengths

Before mapping the tables to TPFDF files, you must estimate the amount of data for each file. To do this, assign a length to each data field (attribute).

Many data fields (for example, the flight number or the destination code) have a fixed length. However, you cannot always be precise when assigning a length to a data field. For example, you cannot be sure how long a passenger's name will be.

Because of this, always allow for future expansion when you are assigning data field lengths. Where possible, use variable length LRECs, and set the variable length portion of the LREC in the DSECT to zero. This makes it easier to expand the field length later.

Table 21 shows the length of each data field from the tables developed in the previous chapters.

Table 21. Data Field Lengths

Table	Data field	Length in bytes
Aircraft	aircraft type seat range seat class	4 8 1
Passenger name	passenger name pointer	25 5
Passenger number	passenger number pointer	8 5
Flight	date time flight number start destination aircraft type availability pointer1 pointer2	2 2 7 (airline=3, flight=4) 3 3 4 6 5
Seat	seat number seat class passenger number passenger name pointer	4 1 8 25 5
Passenger	passenger number passenger name passenger address flight information (flight, date, time, start, destination) passenger facts	8 25 50 17 4

Note: In the flight file, the *date* field has now been divided into two fields, *date* and *time*. This is done to provide a clearer display for the reservation agent.

Figure 14 on page 27 shows the number of LRECs required for each of these six files.

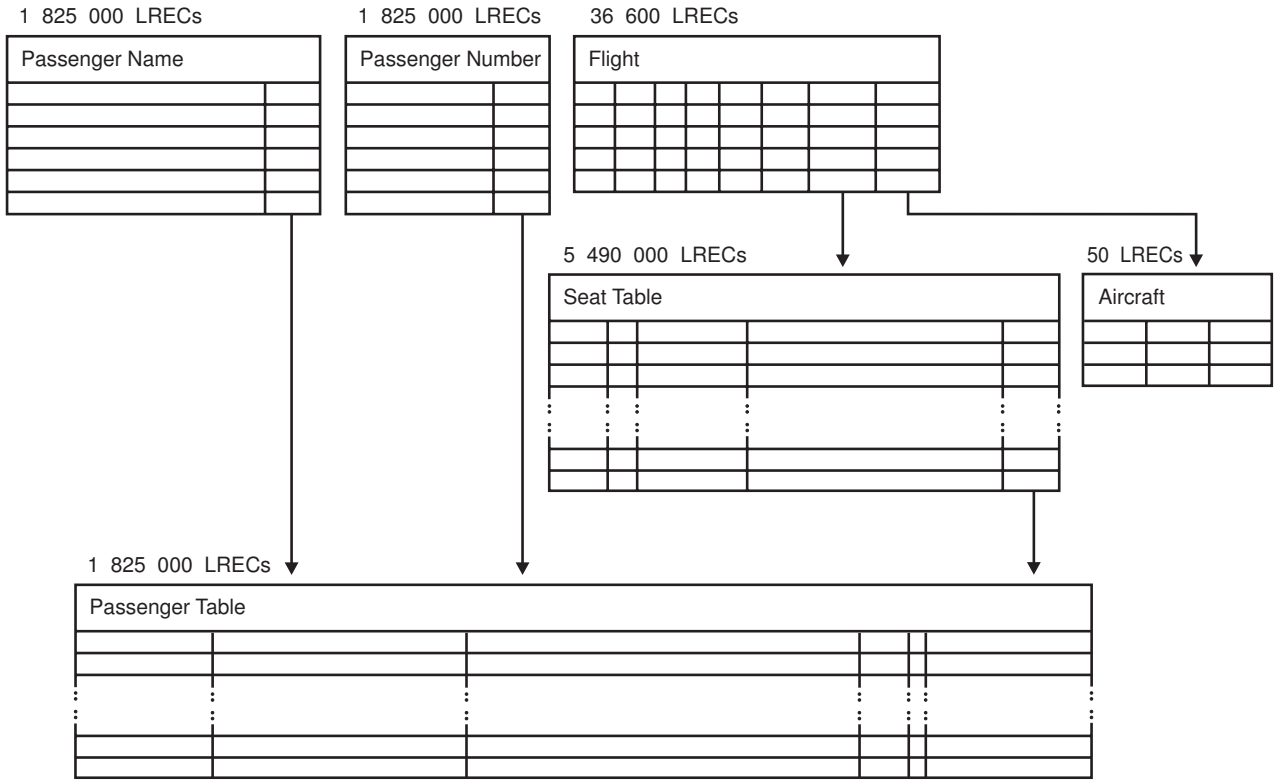


Figure 14. Number of LRECs Required for Each File

Passenger LRECs

Before mapping the tables to TPFDF files, you must calculate the number of passenger LRECs required overall. From the *Data Requirements* table (Table 20 on page 25), you can see that the following amounts are held:

- Flights each day = 100
- Days stored = 366
- Passengers on each flight (average) = 150
- Flights for each passenger (average) = 3

Calculate the number of required passenger LRECs as follows: *(flights each day x days stored x passengers on each flight) ÷ flights for each passenger*

The calculation is:

(100 x 366 x 150) ÷ 3 = 1.825 million

The calculation shows that the database must be able to accommodate 1.825 million passenger LRECs.

Calculating the Number of Subfiles Needed

The following points will help you determine how many subfiles you need to accommodate the data in your file:

- Calculate the number of bytes needed for each data field (for example, *passenger name*). Each character requires 1 byte.
- Calculate the total number of bytes needed for all the data fields in the file.

- Divide this by the number of bytes in your chosen block size.
- The resulting number is the number of subfiles you need.

Block Size

The TPF system offers four different block sizes. Table 22 shows the amount of user data allowed for each block (assuming you are using optional trailers):

Table 22. TPF Block Sizes

Block type	Bytes of user data
L0	127
L1	319
L2	993
L4	4033

The Airline Control System (ALCS) offers eight different block sizes as detailed in Table 23.

Table 23. ALCS Block Sizes

Block type	Bytes of user data
L0	127 (max.)
L1	319 (max.)
L2	993 (max.)
L3	4000 (min.)
L4	4033 (max.)
L5	32K (max.)
L6	32K (max.)
L7	32K (max.)
L8	32K (max.)

Though TPF block sizes are fixed, ALCS block sizes show the maximum number of bytes for each size. In practice, ALCS blocks can be any size you choose. For easier data transfer between TPF and ALCS, blocks L1, L2, and L4 are the same size in both systems.

Note: Block size affects performance. A large block size can reduce the amount of overflow blocks required but may waste DASD space.

Chaining

When deciding on the number of subfiles you need, you must also consider the number of overflow blocks (chains). A large number of blocks slows down data retrieval, hindering the performance of the database.

In general, you should have no more than 3 blocks in a subfile. However, if the file is not likely to be accessed frequently, you may be able to use more blocks.

Note: For files with a large number of data blocks, B^+Tree indexing will speed data retrieval and the performance of the database.

Overflow Blocks

Overflow blocks do not need be the same size as the prime block. Because of this, you can save DASD space by having small overflow blocks when the data overflow is slight.

"Database Design Hints and Tips" on page 161 discusses how to define the size of overflow blocks.

Mapping the Passenger Name File

Note: The example in this section is theoretical. It shows an even distribution of LRECs. In practice, an even distribution is unlikely to occur.

Each LREC in the passenger name file contains the fields shown in Table 24. Table 24 also shows the number of bytes in each field.

Table 24. LREC Fields for the Passenger Name File

Field	No. of bytes
size	2
key	1
passenger name	25
pointer to passenger file	5
Total	33

From this example, you can see that each LREC in the passenger name file contains 33 bytes.

Each LREC must be stored in a block. In this example, a block size of 4-K is used. A 4-K block can hold 4033 bytes of user data, assuming you use optional block trailers.

The following calculation shows how many LRECs from the passenger name file can be held in a 4-K block:

```
4033 ÷ 33 = 122 LRECs in
each block
```

The calculations made in "Passenger LRECs" on page 27 showed that the database must be able to accommodate 1.825 million passenger LRECs. Because of this, allow for 1.825 million LRECs in the passenger name file. Each 4-K block can hold 122 LRECs.

The calculation that follows shows how many blocks are needed to accommodate the 1.825 million LRECs. (The calculation assumes an even distribution of LRECs):

```
1.825 million ÷ 122 = 14959 blocks
```

Each TPFDF subfile contains one prime block and, if necessary, a number of overflow blocks. You can see that it would not be feasible to allocate all the passenger name LRECs to a single subfile because there would be 14 959 overflow blocks.

For performance reasons, distribute the 1.825 million LRECs over a number of subfiles. Because you are dealing with a passenger name file, distribute the LRECs by passenger name.

Distributing the Passenger Name LRECs

The TPFDF product provides predefined algorithms for distributing LRECs evenly across a range of ordinals. Some of these methods are designed for alphabetic data or alphanumeric data mapping. Others calculate an ordinal number using a hashing technique to distribute the LRECs. The method of distributing the LRECs is specified by an *algorithm number* in the &SW00RBV symbol of the file DSECT.

In TPFDF macros, you can use the ALG parameter to specify the location of an input string for an algorithm.

The TPFDF product provides the following three algorithms for distributing data by alphabetic characters:

- #TPFDB01
- #TPFDB02
- #TPFDB03.

#TPFDB01 distributes data by the first character of the algorithm string. #TPFDB02 uses the first 2 characters, and #TPFDB03 uses the first 3 characters.

Table 25 uses SMITH as the example passenger name. It shows the characters used by each algorithm. The table also shows the number of subfiles created by each algorithm and the number of blocks required in each subfile to hold the passenger name LRECs.

Table 25. Algorithms Using Alphabetic Characters

Algorithm	Characters used	No. of subfiles resulting	No. of blocks required
#TPFDB01	S	26	576
#TPFDB02	SM	676	23
#TPFDB03	SMI	17576	1

Note: Table 25 assumes a perfect distribution of LRECs over the subfiles. In practice, this is unlikely to occur. For example, names beginning with S are more common than names beginning with X. However, you could improve the distribution by basing the algorithm on a character that is not the first in the name; for example, the second consonant.

If you cannot get a good distribution using any of the predefined TPFDF algorithms, you can create a unique user-defined algorithm in the user exit UWBD. See page 79 for more information about creating user-defined algorithms.

In Table 25, you can see that algorithm #TPFDB01 would require chaining for 676 blocks. In a real-time system, this would create a substantial I/O overhead. Algorithm #TPFDB02 requires 23 blocks. It would clearly take less time to read only 23 blocks but the response would still be too slow.

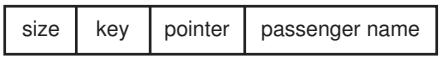
However, with a perfect distribution of LRECs, algorithm #TPFDB03 would require no overflow blocks at all. All the LRECs could be contained in a single block. Even with an actual (real world) distribution, #TPFDB03 would probably produce no more than 5 blocks.

File Structure

The passenger name file has a classic index structure which the TPFDF product can maintain easily. The passenger name is the *index key*. Each LREC contains a pointer to the passenger detail file. The TPFDF product maintains these indexes for you.

In TPFDF macros, you can use the ALG parameter to specify the location of an input string for an algorithm. In this example, the string passed with the ALG parameter is the passenger name. Algorithm #TPFDB03 uses this to access the subfiles.

The LREC structure for the passenger name file is as follows:



The *size* and *key* fields shown in this LREC are for the TPFDF product use.

Mapping the Passenger Number File

Each LREC in the passenger number file contains the fields shown in Table 26. Table 26 also shows the number of bytes in each field.

Table 26. LREC Fields for the Passenger Number File

Field	**No. of bytes**
size	2
key	1
passenger number	8
pointer to passenger file	5
Total	16

From this, you can see that each LREC in the passenger number file contains 16 bytes.

The following calculation shows how many LRECs from the passenger number file can be held in a 4-K block:

```
4033 ÷ 16 = 252 LRECs in
each block
```

The database must be able to hold 1.825 million passenger LRECs.

The calculation that follows shows how many blocks you would need to accommodate the 1.825 million passenger number LRECs. (The calculation assumes an even distribution of LRECs):

```
1.825 million ÷ 252 = 7243 blocks
```

Again, the number of overflow blocks required is too many for a real-time system. Choose an algorithm to distribute the LRECs over subfiles.

Distributing the Passenger Number LRECs

The algorithms used with the passenger name file cannot be used here because passenger numbers are not alphabetic. However, the TPFDF product provides a hashing algorithm that distributes LRECs numerically.

The hashing algorithm (#TPFDB09) distributes the LRECs evenly. They do not need to be stored in any particular order, because you will never need to access more than one at a time. For instance, you will not need to access a range of passenger numbers all at once.

Algorithm #TPFDB09 uses the algorithm string as a seed. It divides the result of the calculation by the number of subfiles in the file. The remainder is the ordinal number of the destination subfile.

Note: Because you need a remainder in every division process, choose an odd number (ideally a prime number) for the number of subfiles.

To ensure the least number of overflow blocks possible, choose the prime number nearest to 7243 (the number of blocks required for the passenger number file).

Ensuring a Good Distribution

Because the LRECs in this file are distributed by passenger number, you need to ensure that this field gives a good distribution over the subfiles.

The passenger number is the algorithm seed from which the pseudo-random number is calculated. Because many passenger numbers begin with zeros, the resulting distribution is not likely to be even. You can improve the distribution by manipulating the algorithm string before it is used.

For example, if the first 4 bytes of the passenger number are swapped with the second 4 bytes, the difficulty is overcome as you can see in Table 27.

Table 27. Manipulating the Algorithm String

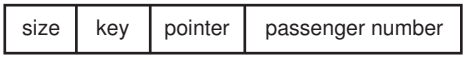
Passenger number	Algorithm string
00087628 00987653	76280008 76530098

Manipulating the algorithm string has given a good distribution of LRECs. If there is any overflow, it is likely to be small. Because of this, 1-K overflow blocks will probably be adequate. (You can change the size of overflow blocks by changing the value of the DBDEF ARS parameter and entering the ZUDFM OAP command.)

File Structure

The passenger number file has a classic index structure that the TPFDF product can maintain easily. Here the ALG string is the passenger number, which algorithm #TPFDB09 uses to access the subfiles.

The LREC structure for the passenger number file is as follows:



Mapping the Aircraft File

Each LREC in the aircraft file contains the fields shown in Table 28. Table 28 also shows the number of bytes in each field.

Table 28. LREC Fields for the Aircraft File

Field	No. of bytes
size	2
key	1
aircraft type	4
seat range	8
seat class	1
Total	16

Each LREC in the aircraft file therefore contains 16 bytes.

Because the database must be able to accommodate 50 aircraft types and three classes, you can calculate the required amounts of data as follows:

no. of aircraft types x no. of classes x LREC size = amount of data

The calculation is:

50 x 3 x 16 = 2400 bytes

This comparatively small number of bytes fits comfortably into a fixed block of size L4 (4095 bytes, including header and trailer). Because only one block is required, you can use a miscellaneous file.

Distributing the Aircraft LRECs

The aircraft file contains only one block of data. Because of this, all the data can be held in a single subfile, and you do not need to distribute the aircraft LRECs over many subfiles. Because you are using a single subfile, use the single-subfile algorithm (#TPFDB04) to distribute the LRECs in the file.

Note: The flight file (see Figure 14 on page 27) contains a pointer to the aircraft file. However, because the aircraft file contains only 1 block, the aircraft file address always locates the correct block. Because of this, you can remove the pointer from the flight file.

Allowing for Expansion

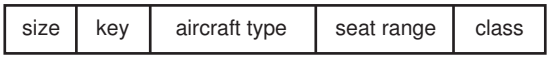
Although the aircraft file is small at the moment, it may well need to expand in the future. Therefore, consider now how a data overflow might be accommodated.

Because the aircraft LRECs are only 16 bytes long, there is already some room for expansion in the existing L4 (4095 bytes) block. Overflow blocks of L2 size (1055 bytes) should be adequate for future needs.

File Structure

When deciding the structure of the LRECs, consider how the TPFDF product will interrogate the file. The aircraft file is not an index file, so the LRECs do not need to contain pointers. The TPFDF product needs only the aircraft type, so the aircraft

LREC structure can be as follows:



Mapping the Flight File

Each LREC in the flight file contains the fields shown in Table 29. Table 29 also shows the number of bytes in each field.

Table 29. LREC Fields for the Flight File

Field	No. of bytes
size	2
key	1
date	2
flight number	7
time	2
start	3
destination	3
aircraft type	4
availability	6
pointer to seat file	5
Total	35

The pointer to the aircraft table has been removed from this file (see “Distributing the Aircraft LRECs” on page 33). Each LREC in the flight file now contains 35 bytes.

As shown previously in the *Data Requirements* table (Table 20 on page 25), the database must be able to accommodate 100 flights each day. You can calculate the amount of data as follows:

```
no. of
flights each day x LREC size = amount of data
```

The calculation is:

```
100 x 35 = 3500 bytes
```

Therefore, use 4-K blocks to store the LRECs. Because the database must hold records for 366 days, you need 366 blocks to store the flight information.

Distributing the Flight LRECs

The structure of the flight file is simple; there is one subfile for each day’s flights. One fast method of accessing the subfiles is to assign a relative number to each day of the year.

The application translates the day directly into an ordinal matching a subfile. For example, 1 Jan=1st, 2 Jan=2nd . . . . 31 Dec=366th. The ordinal serves as a value for the string passed with the ALG parameter. You can use algorithm #TPFDB05 to retrieve the ordinal directly.

For a more detailed explanation, see Figure 51 on page 134.

Note: Because leap years must be accommodated, you must define 366 blocks, although one of them will be redundant most of the time.

Allowing for Expansion

Although the size of the flight file is fixed at the moment, you may need to expand it in the future. Because of this, consider how a data overflow might be accommodated.

There is room for some expansion in the blocks already assigned, and 4-K blocks are likely to be large enough to accommodate an increased number of flights. If the number of flights increase greatly, increase the size of the overflow blocks by changing the value of the DBDEF ARS parameter and then using the ZUDFM OAP command. An increase in block size would mean a reduction in I/O processing, therefore maintaining good performance levels despite the increase in data.

File Structure

Because the flight file is an index file, its LRECs must contain pointers. The structure of the flight file LRECs is as follows:

Mapping the Seat File

Each LREC in the seat file contains the fields shown in Table 30. Table 30 also shows the number of bytes in each field.

Table 30. LREC Fields for the Seat File

Field	**No. of bytes**
size	2
key	1
seat number	4
seat class	1
passenger name	25
passenger number	8
pointer to passenger file	5
Total	46

From this table, you can see that each LREC in the seat file contains 46 bytes.

Because the average number of passengers on each flight is 150, you can calculate the amount of data as follows:

```
no. of passengers on each
flight x LREC size = amount of data
```

The calculation is:

```
150 x 46 = 6900 bytes
```

Some aircraft can carry as many as 300 passengers while others can carry only 50. Because data must be kept for all the different aircraft types, you must also calculate the maximum and minimum data requirements.

The calculations are as follows:

```
300 x 46 = 13800 bytes  (maximum)
 50 x 46 =  2300 bytes  (minimum)
```

Because the seat file is referenced from the flight file (Table 29 on page 34), you should create it as a pool file. As a pool file, it is allocated only as needed. If you create the seat file as a fixed file, it is permanently allocated and must be defined for every file. Therefore, you must consider the size of the file as well.

Distributing the Seat File LRECs

Because 4-K blocks hold only 4033 bytes of user data, some chaining is needed in this file. The chain lengths vary from 1 to 4. You need 4 blocks to accommodate the number of seats in the largest aircraft.

When you distribute the seat file LRECs, use algorithm #TPFDBFF to indicate that the seat file is an index file.

Mapping the Passenger File

Each LREC in the passenger file contains the fields shown in Table 31. This table also shows the number of bytes in each field.

Table 31. LREC Fields for the Passenger File

Field	No. of bytes
size	2
key	1
passenger number	8
passenger name	25
passenger address	*n* (10–50)
flight information	17
passenger facts	4
Total	107

From this table, you can see that each LREC in the passenger file contains from 57–97 bytes. (The passenger address field has a minimum of 10 bytes and a maximum of 50 bytes.)

Data requirements for the passenger file must be calculated twice. You need to know the requirements for the average number of flights (3) and for the maximum number of flights (20).

You can calculate the data requirements for the average number of flights as follows:

```
number + name + address + facts + (average no. of flights
x flight information) = amount of data
```

The calculation is:

```
8 + 25 + n + 4 + (3 x 17) = 88 + n bytes
```

The calculation for the maximum number of flights is as follows:

```
8 + 25 + n + 4 + (20 x 17) = 377 + n bytes
```

From these calculations, you can see that most passenger LRECs fit into a single block of L1 size (320 bytes). However, you need 1 overflow block for each LREC where passengers are taking 14 flights or more.

Spreading Data over Several LRECs

Some of the passenger LRECs contain too much data to fit into a single 381-byte block. Moreover, the data held in variable length LRECs is likely to vary in size.

Because the TPFDF product does not allow you to spread a single LREC over more than 1 block, you must spread the data over several LRECs. The alternative of using larger blocks for the passenger LRECs is wasteful because only a few of the LRECs need the larger size block.

Looking at the passenger file again (Table 32), you can see that the data can be split into five separate LRECs.

Table 32. Passenger File

Passenger number	**Passenger name**	**Passenger address**	**Flight**	**Date**	**Passenger facts**
Pn1	Na1	Ad1	Fl1 Fl2 Fl3	Da1 Da1 Da2	Ft1
Pn2	Na2	Ad2	Fl1	Da1	Ft1
Pn3	Na3	Ad3	Fl1	Da1	
Pn4	Na4	Ad4	Fl1	Da1	

Figure 15 shows the five new LRECs and the number of bytes held in each LREC.

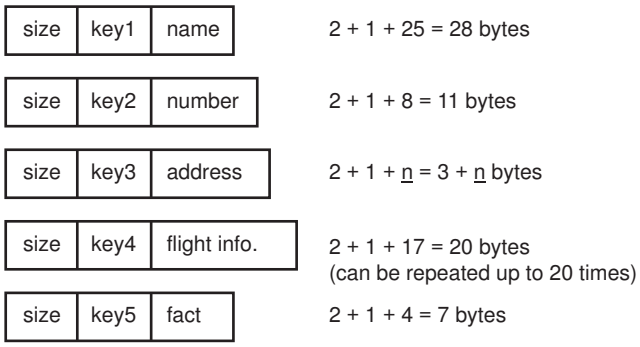


Figure 15. Spreading Data over Several LRECs

Because the data is now spread over several LRECs, the database must hold slightly more data. (Each LREC holds 3 bytes of identifying data.) However, the data is easier to manipulate in this form so that performance is improved overall.

Note: *Name* and *number* are kept as two separate LRECs in case the name field needs to be expanded in the future.

Coding the DSECT and DBDEF Macros

Each of the files mapped in the previous chapter requires a DSECT macro and a DBDEF macro. The following describes how to set the global symbols for each DSECT and gives examples of possible DBDEF statements for each file.

It is assumed that you already know how to create DSECT and DBDEF macros. If you are not sure how to do this, see the following:

- "Creating a DSECT Macro Definition" on page 69
- "Creating a DBDEF Macro Definition" on page 89.

Only the *backward* path is shown in the sample DBDEFs. The *forward* path (recoup information) is not shown.

Figure 16 shows the DSECT names, algorithms, and paths for each of the six mapped files.

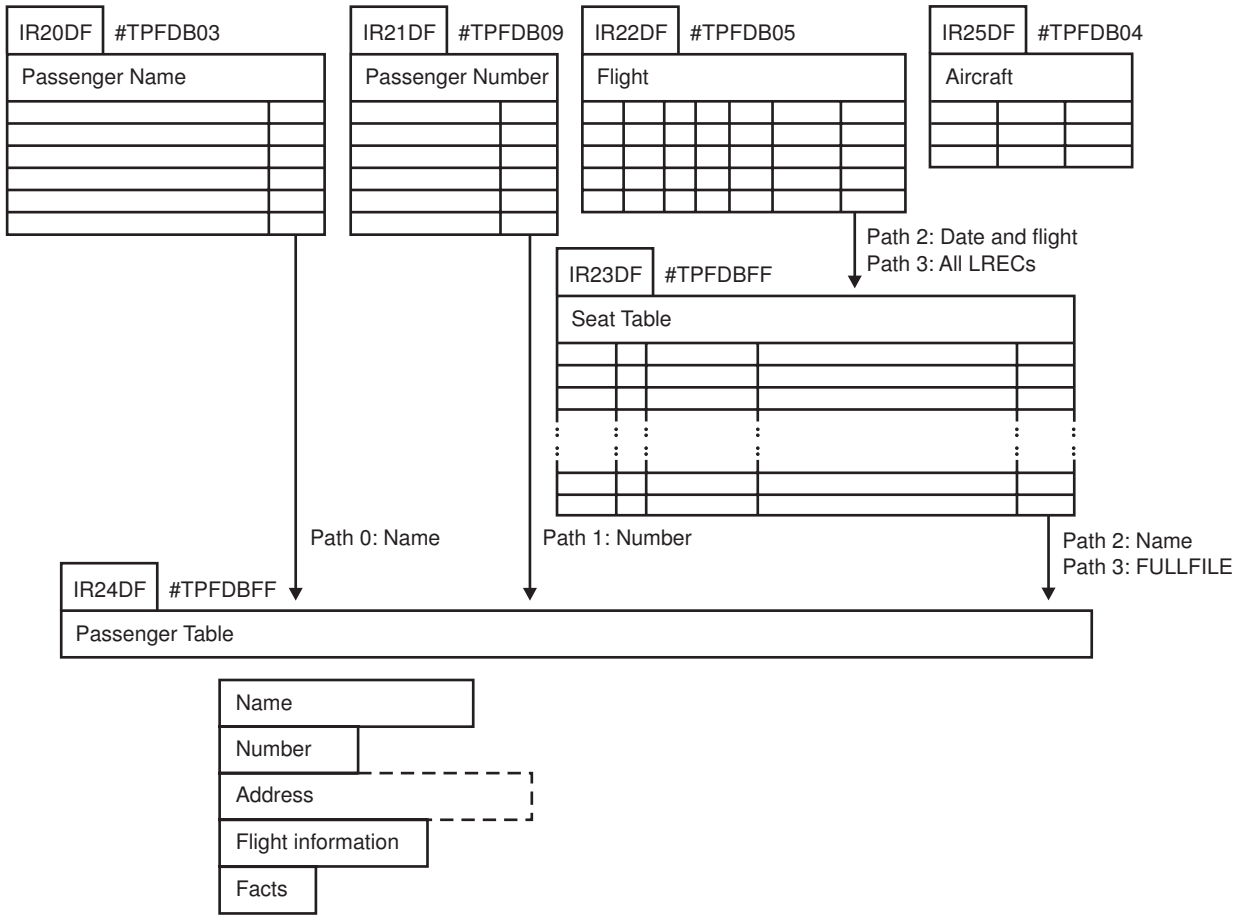


Figure 16. TPFDF Files: DSECT Names, Algorithms, and Paths

DSECT and DBDEF for the Passenger Name File

Macro IR20DF is the DSECT for the passenger name file. The following example shows the DSECT and the DBDEF for this index file.

DSECT

Figure 17 shows the DSECT used to define the passenger name file.

```
         MACRO
&LABEL   IR20DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*************************************************************************
*                                                                       *
*   IR20DF   PASSENGER NAME FILE (INDEX)                                *
*                            DATE: 11 APRIL 1991                        *
*                                                                       *
*************************************************************************
         GBLB  &IR20DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR20DF       '    DOC NAME
&DATE    SETC  '06FEB91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*********************************************************************
.*      DEFINITIONS FOR TPFDF                                         *
.*********************************************************************
&SW00WID SETC  '20'          FILE ID
&SW00WRS SETC  'L1'          BLOCK SIZE
&SW00RCT SETC  '#IR20DF'     FACE FILE TYPE
&SW00RBV SETC  '#TPFDB03'    FILE ALGORITHM
&SW00BOR SETC  '0'           BASE ORDINAL NUMBER
&SW00EOR SETC  '-1'          END ORDINAL NBR
&SW00ILV SETC  '0'           MAXIMUM INTERLEAVING FACTOR IF APPLIC
&SW00PTN SETC  '0'           NUMBER OF PARTITIONS
&SW01EO# SETC  '&SW00EOR'    RECOUP END ORDINAL
&SW02FIL SETC  'IR20DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*********************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR20DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                     *
*  DESCRIPTION OF IR20DF                                              *
*                                                                     *
*  1. DATA AREA NAME                                                  *
*                                                                     *
*     PASSENGER NAME FILE                                             *
*                                                                     *
*  2. MEMBER NAME                                                     *
*                                                                     *
*     IR20DF                                                          *
*                                                                     *
*  3. INVOCATION                                                      *
*                                                                     *
*     IR20DF REG=RGD,                                                 *
*            (SUFFIX=X),                                              *
*            (ORG=IR20HDR)                                            *
```

Figure 17. DSECT to Define the Passenger Name File (Part 1 of 3)

```
*  4. GENERAL CONTENTS AND USAGE                                    *
*                                                                   *
*  4.1. ROLE IN SYSTEM                                              *
*                                                                   *
*       THE PASSENGER NAME FILE CONTAINS INDEX POINTERS TO THE      *
*       PASSENGER FILE. THIS ALLOWS FAST ACCESS TO A PASSENGER      *
*       BY THE NAME.                                                *
*                                                                   *
*  4.2. DATA LAYOUT                                                 *
*                                                                   *
*       STANDARD TPFDF FILE HEADER                                  *
*                                                                   *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.  *
*                                                                   *
*       PRIMARY KEY     USAGE                                       *
*          80           PASSENGER NAME LOGICAL RECORD               *
*                                                                   *
*  4.3. PROGRAMMING ASPECTS                                         *
*                                                                   *
*  4.3.1. PROGRAMMING RESTRICTIONS                                  *
*                                                                   *
*         NONE.                                                     *
*                                                                   *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                          *
*                                                                   *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:             *
*         - PRIMARY KEY                                             *
*                                                                   *
*  5. STORAGE FACTORS                                               *
*                                                                   *
*  5.1. BLOCK SIZE                                                  *
*                                                                   *
*       DEFINED IN DBDEF.                                           *
*                                                                   *
*  5.2. FILE REQUIREMENTS                                           *
*                                                                   *
*       THE #TPFDB03 ALGORITHM REQUIRES 17576 FIXED FILES           *
*                                                                   *
*  5.3. ACCESSING SCHEME                                            *
*                                                                   *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND        *
*       CROSS RELATION OF LRECS ETC)                                *
*       THE FILE IS UP ORGANIZED. IT IS ACCESSED BY THE PATH=0      *
*                                                                   *
*       DEFINITION OF THE PASSENGER FILE.                           *
*                                                                   *
*  6. DATA CONTROL                                                  *
*                                                                   *
*  6.1. CHAINING AND OVERFLOW                                       *
*                                                                   *
*       STANDARD TPFDF CHAINING.                                    *
*                                                                   *
*  6.2. DATA FIELD ADDRESSING                                       *
*                                                                   *
*       OFFSET WITHIN STANDARD TPFDF LREC.                          *
*  7. IMPLEMENTATION REQUIREMENTS                                   *
*                                                                   *
*  8. REFERENCES                                                    *
*                                                                   *
*  9. COMMENTS                                                      *
*                                                                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 17. DSECT to Define the Passenger Name File (Part 2 of 3)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR20DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR20DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*********************************************************************
*        STANDARD TPFDF HEADER                                      *
*********************************************************************
IR20HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR20VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR20HDL&CG1 EQU IR20VAR&CG1-IR20HDR&CG1  HEADER-LENGTH UP TO IR20VAR
            ORG IR20HDR&CG1
IR20REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR20SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR20KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR20DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)             *
*********************************************************************
.*                   USE KEY #IR20K80 IF ONLY ONE KEY
.*                   #IR20K00-#IR20K0F ARE RESERVED FOR TPFDF
.*                   #IR20KF0-#IR20KFF ARE RESERVED FOR TPFDF
#IR20K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR20L80 EQU   IR20E80&CG1-IR20REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR20DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR20ORG&CG1 EQU *            START VARIABLE DATA PER LREC
.*
*********************************************************************
*        DESCRIPTION OF  F I R S T  LOGICAL RECORD TYPE             *
*********************************************************************
IR20FAD&CG1 DS  0AL4         F.A. OF POINTER USED BY DBDEF
IR20FA1&CG1 DS  AL4          F.A. OF POINTER TO DETAIL/LOWER LEVEL IN
IR20RCC&CG1 DS  0AL1         CHECK BYTE USED BY DBDEF
IR20RC1&CG1 DS  AL1          CHECK BYTE
IR20A80&CG1 DS  0CL25        KEYAREA
IR20PNM&CG1 DS  CL25         PASSENGER NAME
IR20E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
.*
         ORG IR20ORG&CG1
.*
.*********************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 17. DSECT to Define the Passenger Name File (Part 3 of 3)

DBDEF

Figure 18 shows the position of IR20DF in the file structure.

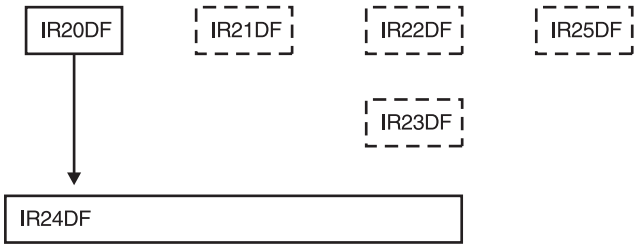


Figure 18. Position of IR20DF in the File Structure

```
DBDEF FILE=IR20DF,             -
      (ITK=#IR20K80,ID2=,      -
      INDEX=(IR24DF,0))
```

Note: IR20DF and IR21DF both contain references to IR24DF. Specify RCI processing for these files. (See Figure 53 on page 138 for more details of RCI processing.)

DSECT and DBDEF for the Passenger Number File

Macro IR21DF is the DSECT for the passenger number file. The following example shows the DSECT and the DBDEF for this index file.

DSECT

Figure 19 shows the DSECT used to define the passenger number file.

```
         MACRO
&LABEL   IR21DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*********************************************************************
*                                                                   *
*   IR21DF  PASSENGER NUMBER FILE (INDEX)                           *
*                           DATE:11APR91                            *
*                                                                   *
*********************************************************************

         GBLB  &IR21DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR21DF       '    DOC NAME
&DATE    SETC  '08FEB91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*******************************************************************
.       DEFINITIONS FOR TPFDF                                      *
.*******************************************************************
&SW00WID SETC  '21'          FILE ID
&SW00WRS SETC  'L4'          BLOCK SIZE
&SW00ARS SETC  'L2'          ALTERNATE BLOCK SIZE
&SW00RCT SETC  '#IR21DF'     FACE FILE TYPE
&SW00RBV SETC  '#TPFDB09'    FILE ALGORITHM
&SW00BOR SETC  '0'           BASE ORDINAL NUMBER
&SW00EOR SETC  '-1'          END ORDINAL NBR
&SW00ILV SETC  '0'           MAXIMUM INTERLEAVING FACTOR IF APPLIC
&SW00PTN SETC  '0'           NUMBER OF PARTITIONS
&SW01EO# SETC  '&SW00EOR'    RECOUP END ORDINAL
&SW02FIL SETC  'IR21DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*******************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR21DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*  DESCRIPTION OF IR21DF                                           *
*                                                                  *
*  1. DATA AREA NAME                                               *
*                                                                  *
*     PASSENGER NUMBER INDEX FILE                                  *
*                                                                  *
*  2. MEMBER NAME                                                  *
*                                                                  *
*     IR21DF                                                       *
*                                                                  *
```

Figure 19. DSECT to Define the Passenger Number File (Part 1 of 3)

```
*  3. INVOCATION                                                *
*                                                               *
*     IR21DF REG=RGD,                                           *
*           (SUFFIX=X),                                         *
*           (ORG=IR21HDR)                                       *
*                                                               *
*  4. GENERAL CONTENTS AND USAGE                                *
*                                                               *
*  4.1. ROLE IN SYSTEM                                          *
*                                                               *
*       THE IR21DF CONTAINS INDEX POINTERS TO THE PASSENGER FILE *
*       BASED ON THE UNIQUE PASSENGER NUMBER.                   *
*                                                               *
*  4.2. DATA LAYOUT                                             *
*                                                               *
*       STANDARD TPFDF FILE HEADER                              *
*                                                               *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.*
*                                                               *
*       PRIMARY KEY     USAGE                                   *
*          80           PASSENGER NUMBER LOGICAL RECORD         *
*                                                               *
*  4.3. PROGRAMMING ASPECTS                                     *
*                                                               *
*  4.3.1. PROGRAMMING RESTRICTIONS                              *
*                                                               *
*         NONE.                                                 *
*                                                               *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                      *
*                                                               *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:         *
*         - PRIMARY KEY                                         *
*                                                               *
*  5. STORAGE FACTORS                                           *
*                                                               *
*  5.1. BLOCK SIZE                                              *
*                                                               *
*       DEFINED IN DBDEF.                                       *
*                                                               *
*  5.2. FILE REQUIREMENTS                                       *
*                                                               *
*       THE NUMBER OF FILES ALLOCATED IS 7999.                  *
*                                                               *
*  5.3. ACCESSING SCHEME                                        *
*                                                               *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND    *
*       CROSS RELATION OF LRECS ETC)                            *
*       THE IR21DF IS ACCESSED BY PATH=1 OF THE PASSENGER FILE. *
*                                                               *
*  6. DATA CONTROL                                              *
*                                                               *
*  6.1. CHAINING AND OVERFLOW                                   *
*                                                               *
*       STANDARD TPFDF CHAINING.                                *
*                                                               *
*  6.2. DATA FIELD ADDRESSING                                   *
*                                                               *
*       OFFSET WITHIN STANDARD TPFDF LREC.                      *
*                                                               *
*  7. IMPLEMENTATION REQUIREMENTS                               *
*                                                               *
*  8. REFERENCES                                                *
*                                                               *
*  9. COMMENTS                                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 19. DSECT to Define the Passenger Number File (Part 2 of 3)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR21DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR21DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*************************************************************************
*        STANDARD TPFDF HEADER                                          *
*************************************************************************
IR21HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR21VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR21HDL&CG1 EQU IR21VAR&CG1-IR21HDR&CG1  HEADER-LENGTH UP TO IR21VAR
            ORG IR21HDR&CG1
IR21REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR21SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR21KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR21DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*************************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)                 *
*************************************************************************
.*                    USE KEY #IR21K80 IF ONLY ONE KEY
.*                    #IR21K00-#IR21K0F ARE RESERVED FOR TPFDF
.*                    #IR21KF0-#IR21KFF ARE RESERVED FOR TPFDF
#IR21K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR21L80 EQU   IR21E80&CG1-IR21REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR21DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR21ORG&CG1 EQU *            START VARIABLE DATA PER LREC
*************************************************************************
*        DESCRIPTION OF  FIRST  LOGICAL RECORD TYPE                     *
*************************************************************************
IR21FAD&CG1 DS  0AL4         F.A. OF POINTER USED BY DBDEF
IR21FA1&CG1 DS  AL4          F.A. OF POINTER TO DETAIL/LOWER LEVEL INDX
IR21RCC&CG1 DS  0AL1         CHECK BYTE USED BY DBDEF
IR21RC1&CG1 DS  AL1          CHECK BYTE
IR21A80&CG1 DS  0CL8         KEYAREA
IR21NBR&CG1 DS  CL8          UNIQUE PASSENGER NUMBER
IR21E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
.*
         ORG IR21ORG&CG1
.*
.*****************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 19. DSECT to Define the Passenger Number File (Part 3 of 3)

DBDEF

Figure 20 shows the position of IR21DF in the file structure.

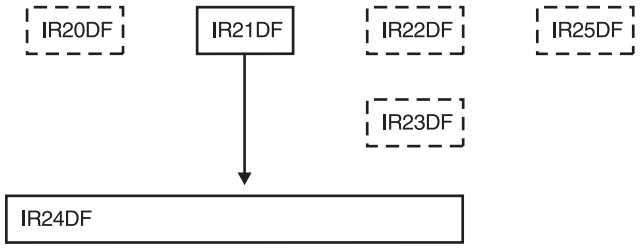


Figure 20. Position of IR21DF in the File Structure

```
DBDEF FILE=IR21DF,             -
      (ITK=#IR21K80,ID2=,      -
      INDEX=(IR24DF,0))
```

Note: IR20DF and IR21DF both contain references to IR24DF. Specify RCI processing for these files. (See Figure 53 on page 138 for more details of RCI processing.)

DSECT and DBDEF for the Flight File

Macro IR22DF is the DSECT for the flight file. The following example shows the DSECT and the DBDEF for this index file.

DSECT

Figure 21 shows the DSECT used to define the flight file.

```
         MACRO
&LABEL   IR22DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*********************************************************************
*                                                                   *
*   IR22DF      FLIGHT FILE (INDEX)                                 *
*                      DATE: 11APR91                                *
*                                                                   *
*********************************************************************
         GBLB  &IR22DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR22DF       '    DOC NAME
&DATE    SETC  '08FEB91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*********************************************************************
.*             DEFINITIONS FOR TPFDF                                 *
.*********************************************************************
&SW00WID SETC  '22'          FILE ID
&SW00WRS SETC  'L4'          BLOCK SIZE
&SW00RCT SETC  '#IR22DF'     FACE FILE TYPE
&SW00RBV SETC  '#TPFDB05'    FILE ALGORITHM
&SW00BOR SETC  '0'           BASE ORDINAL NUMBER
&SW00EOR SETC  '-1'          END ORDINAL NBR
&SW00ILV SETC  '0'           MAXIMUM INTERLEAVING FACTOR IF APPLIC
&SW00PTN SETC  '0'           NUMBER OF PARTITIONS
&SW01EO# SETC  '&SW00EOR'    RECOUP END ORDINAL
&SW02FIL SETC  'IR22DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*********************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR22DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
*  DESCRIPTION OF IR22DF                                            *
*                                                                   *
*  1. DATA AREA NAME                                                *
*                                                                   *
*     FLIGHT FILE                                                   *
*                                                                   *
*  2. MEMBER NAME                                                   *
*                                                                   *
*     IR22DF                                                        *
*                                                                   *
*  3. INVOCATION                                                    *
*                                                                   *
*     IR22DF REG=RGD,                                               *
*            (SUFFIX=X),                                            *
*            (ORG=IR22HDR)                                          *
*                                                                   *
```

Figure 21. DSECT to Define the Flight File (Part 1 of 3)

```
*  4. GENERAL CONTENTS AND USAGE                                    *
*                                                                   *
*  4.1. ROLE IN SYSTEM                                              *
*                                                                   *
*       THIS FILE CONTAINS ALL THE FLIGHTS FOR A PARTICULAR DAY     *
*       WITH POINTERS TO THE SEAT FILE.                             *
*                                                                   *
*  4.2. DATA LAYOUT                                                 *
*                                                                   *
*       STANDARD TPFDF FILE HEADER                                  *
*                                                                   *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.  *
*                                                                   *
*       PRIMARY KEY     USAGE                                       *
*          80           FLIGHT INFORMATION LOGICAL RECORD           *
*                                                                   *
*  4.3. PROGRAMMING ASPECTS                                         *
*                                                                   *
*  4.3.1. PROGRAMMING RESTRICTIONS                                  *
*                                                                   *
*         NONE.                                                     *
*                                                                   *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                          *
*                                                                   *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:             *
*         - PRIMARY KEY                                             *
*                                                                   *
*  5. STORAGE FACTORS                                               *
*                                                                   *
*  5.1. BLOCK SIZE                                                  *
*                                                                   *
*       DEFINED IN DBDEF.                                           *
*                                                                   *
*  5.2. FILE REQUIREMENTS                                           *
*                                                                   *
*       REQUIRES 366 FIXED FILES, 1 PER DAY                         *
*                                                                   *
*                                                                   *
*  5.3. ACCESSING SCHEME                                            *
*                                                                   *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND        *
*       CROSS RELATION OF LRECS ETC)                                *
*       IS ACCESSED BY PATH=3 AND PATH=4 OF THE PASSENGER FILE.     *
*                                                                   *
*  6. DATA CONTROL                                                  *
*                                                                   *
*  6.1. CHAINING AND OVERFLOW                                       *
*                                                                   *
*       STANDARD TPFDF CHAINING.                                    *
*                                                                   *
*  6.2. DATA FIELD ADDRESSING                                       *
*                                                                   *
*       OFFSET WITHIN STANDARD TPFDF LREC.                          *
*                                                                   *
*  7. IMPLEMENTATION REQUIREMENTS                                   *
*                                                                   *
*  8. REFERENCES                                                    *
*                                                                   *
*  9. COMMENTS                                                      *
                                                                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 21. DSECT to Define the Flight File (Part 2 of 3)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR22DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR22DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*********************************************************************
*        STANDARD TPFDF HEADER                                      *
*********************************************************************
IR22HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR22VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR22HDL&CG1 EQU IR22VAR&CG1-IR22HDR&CG1  HEADER-LENGTH UP TO IR22VAR
            ORG IR22HDR&CG1
IR22REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR22SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR22KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR22DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)             *
*********************************************************************
.*                           USE KEY #IR22K80 IF ONLY ONE KEY
.*                         #IR22K00-#IR22K0F ARE RESERVED FOR TPFDF
.*                    #IR22KF0-#IR22KFF ARE RESERVED FOR TPFDF
#IR22K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR22L80 EQU   IR22E80&CG1-IR22REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR22DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR22ORG&CG1 EQU *            START VARIABLE DATA PER LREC
.*
*********************************************************************
*        DESCRIPTION OF  F I R S T  LOGICAL RECORD TYPE             *
*********************************************************************
IR22FAD&CG1 DS  0AL4         F.A. OF POINTER USED BY DBDEF
IR22FA1&CG1 DS  AL4          F.A. OF POINTER TO DETAIL/LOWER LEVEL INDX
IR22RCC&CG1 DS  0AL1         CHECK BYTE USED BY DBDEF
IR22RC1&CG1 DS  AL1          CHECK BYTE
IR22A80&CG1 DS  0CL27        KEYAREA
IR22FLN&CG1 DS  CL7          FLIGHT NUMBER
IR22DAT&CG1 DS  XL2          DATE OF FLIGHT
IR22TIM&CG1 DS  XL2          TIME OF FLIGHT
IR22BRD&CG1 DS  CL3          BOARDING POINT
IR22DES&CG1 DS  CL3          DESTINATION
IR22ACT&CG1 DS  CL4          AIRCRAFT TYPE
IR22AVL&CG1 DS  XL6          AVAILABILITY COUNTS FOR THE 3 CLASSES
IR22E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
.*
         ORG IR22ORG&CG1
.*
.*****************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 21. DSECT to Define the Flight File (Part 3 of 3)

DBDEF

Figure 22 shows the position of IR22DF in the file structure.

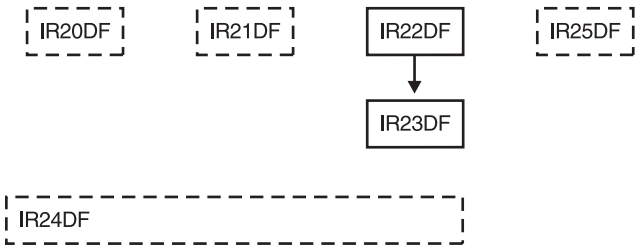


Figure 22. Position of IR22DF in the File Structure

```
DBDEF FILE=IR22DF,             -
      (ITK=#IR22K80,ID2=,      -
      INDEX=(IR23DF,0))
```

DSECT and DBDEF for the Seat File

Macro IR23DF is the DSECT for the seat file. The following example shows the DSECT and the DBDEF for this index file.

DSECT

Figure 23 shows the DSECT used to define the seat file.

```
         MACRO
&LABEL   IR23DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*********************************************************************
*                                                                   *
*   IR23DF      SEAT FILE (INDEX)                                   *
*                    DATE: 11APR91                                  *
*                                                                   *
*********************************************************************
         GBLB  &IR23DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR23DF       '    DOC NAME
&DATE    SETC  '08FEB91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*********************************************************************
.*      DEFINITIONS FOR TPFDF                                        *
.*********************************************************************
&SW00WID SETC  '23'          FILE ID
&SW00WRS SETC  'L4'          BLOCK SIZE
&SW00RBV SETC  '#TPFDBFF'    FILE ALGORITHM
&SW02FIL SETC  'IR23DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*********************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR23DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
*  DESCRIPTION OF IR23DF                                            *
*                                                                   *
*  1. DATA AREA NAME                                                *
*                                                                   *
*     SEAT FILE                                                     *
*                                                                   *
*  2. MEMBER NAME                                                   *
*                                                                   *
*     IR23DF                                                        *
*                                                                   *
*  3. INVOCATION                                                    *
*                                                                   *
*     IR23DF REG=RGD,                                               *
*            (SUFFIX=X),                                            *
*            (ORG=IR23HDR)                                          *
*                                                                   *
```

Figure 23. DSECT to Define the Seat File (Part 1 of 3)

```
*  4. GENERAL CONTENTS AND USAGE                                      *
*                                                                     *
*  4.1. ROLE IN SYSTEM                                                *
*                                                                     *
*       CONTAINS THE PASSENGER NAMES, NUMBERS, CLASSES AND SEAT       *
*       NUMBERS FOR A PARTICULAR FLIGHT.                              *
*                                                                     *
*  4.2. DATA LAYOUT                                                   *
*                                                                     *
*       STANDARD TPFDF FILE HEADER                                    *
*                                                                     *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.    *
*                                                                     *
*       PRIMARY KEY     USAGE                                         *
*          80           SEAT ALLOCATION LOGICAL RECORD                *
*                                                                     *
*  4.3. PROGRAMMING ASPECTS                                           *
*                                                                     *
*  4.3.1. PROGRAMMING RESTRICTIONS                                    *
*                                                                     *
*         NONE.                                                       *
*                                                                     *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                            *
*                                                                     *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:               *
*         - PRIMARY KEY                                               *
*                                                                     *
*  5. STORAGE FACTORS                                                 *
*                                                                     *
*  5.1. BLOCK SIZE                                                    *
*                                                                     *
*       DEFINED IN DBDEF.                                             *
*                                                                     *
*  5.2. FILE REQUIREMENTS                                             *
*                                                                     *
*       POOL FILES (VARIES WITH NUMBER OF FLIGHTS AND AIRCRAFT TYPE)  *
*                                                                     *
*  5.3. ACCESSING SCHEME                                              *
*                                                                     *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND          *
*       CROSS RELATION OF LRECS ETC)                                  *
*       THIS FILE IS ACCESSED BY PATH=3 AND PATH=4 METHODS OF FILE    *
*       IR24DF (PASSENGER FILE).                                      *
*                                                                     *
*  6. DATA CONTROL                                                    *
*                                                                     *
*  6.1. CHAINING AND OVERFLOW                                         *
*                                                                     *
*       STANDARD TPFDF CHAINING.                                      *
*                                                                     *
*  6.2. DATA FIELD ADDRESSING                                         *
*                                                                     *
*       OFFSET WITHIN STANDARD TPFDF LREC.                            *
*                                                                     *
*  7. IMPLEMENTATION REQUIREMENTS                                     *
*                                                                     *
*  8. REFERENCES                                                      *
*                                                                     *
*  9. COMMENTS                                                        *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 23. DSECT to Define the Seat File (Part 2 of 3)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR23DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR23DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*********************************************************************
*        STANDARD TPFDF HEADER                                      *
*********************************************************************
IR23HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR23VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR23HDL&CG1 EQU IR23VAR&CG1-IR23HDR&CG1  HEADER-LENGTH UP TO IR23VAR
            ORG IR23HDR&CG1
IR23REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR23SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR23KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR23DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)             *
*********************************************************************
.*                   USE KEY #IR23K80 IF ONLY ONE KEY
.*                   #IR23K00-#IR23K0F ARE RESERVED FOR TPFDF
.*                   #IR23KF0-#IR23KFF ARE RESERVED FOR TPFDF
#IR23K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR23L80 EQU   IR23E80&CG1-IR23REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR23DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR23ORG&CG1 EQU *            START VARIABLE DATA PER LREC
.*
*********************************************************************
*        DESCRIPTION OF  FIRST LOGICAL RECORD TYPE                  *
*********************************************************************
IR23FAD&CG1 DS  0AL4         INDEX FILE ADDRESS
IR23FA1&CG1 DS  AL4
IR23RCC&CG1 DS  0AL1         INDEX RECORD CODE CHECK
IR23RC1&CG1 DS  AL1
IR23A80&CG1 DS  0CL38        USER DEFINED AREA
IR23PNA&CG1 DS  CL25         PASSENGER NAME
IR23PNN&CG1 DS  CL8          PASSENGER NUMBER
IR23SNB&CG1 DS  XL2          SEAT NUMBER
IR23SCL&CG1 DS  CL1          SEAT CLASS
IR23E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
.*
         ORG IR23ORG&CG1
*********************************************************************
*        ALGORITHM DESCRIPTION                                      *
*********************************************************************
         ORG IR23REC&CG1
IR23@2BEG&CG1 EQU *             PATH 2 DESCRIPTION
IR23@2FLN&CG1 DS  CL7
IR23@2END&CG1 EQU *
         ORG IR23REC&CG1
IR23@3BEG&CG1 EQU *             PATH 3 DESCRIPTION
IR23@3END&CG1 EQU *
.*******************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 23. DSECT to Define the Seat File (Part 3 of 3)

DBDEF

Figure 24 shows the position of IR23DF in the file structure.

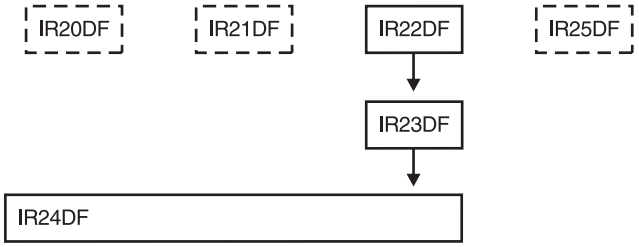


Figure 24. Position of IR23DF in the File Structure

```
DBDEF FILE=IR23DF,                                          -
      (ITK=#IR22K80,ID2=,                                   -
      INDEX=(IR24DF,0)),                                    -
      (IID=IR22DF,PTH=2,IKY=80,IPA=7,ILA=2,IPK=0,ILK=9,-
      KEY1=(PKY=#IR22K80,UP),                               -
      KEY2=(R=IR22FL,S=0,UP)),                              -
      (IID=IR22DF,PTH=3,IKY=80,                             -
      KEY1=(PKY=#IR22K80,UP),                               -
      KEY2=(R=IR22FLN,S=0,UP))
```

DSECT and DBDEF for the Passenger File

Macro IR24DF is the DSECT for the passenger file. The following example shows the DSECT and the DBDEF for this detail file.

DSECT

Figure 25 shows the DSECT used to define the passenger file.

```
         MACRO
&LABEL   IR24DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*********************************************************************
*                                                                   *
*   IR24DF      PASSENGER FILE                                      *
*                         DATE: 11APR91                             *
*                                                                   *
*********************************************************************
         GBLB  &IR24DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR24DF    '    DOC NAME
&DATE    SETC  '11APR91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*****************************************************************
.*      DEFINITIONS FOR TPFDF                                     *
.*****************************************************************
&SW00WID SETC  '24'          FILE ID
&SW00WRS SETC  'L1'          BLOCK SIZE
&SW00ARS SETC  'L1'          ALTERNATE BLOCK SIZE
&SW00RBV SETC  '#TPFDBFF'    FILE ALGORITHM
&SW02FIL SETC  'IR24DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*****************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR24DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
*  DESCRIPTION OF IR24DF                                            *
*                                                                   *
*  1. DATA AREA NAME                                                *
*                                                                   *
*     PASSENGER FILE                                                *
*                                                                   *
*  2. MEMBER NAME                                                   *
*                                                                   *
*     IR24DF                                                        *
*                                                                   *
*  3. INVOCATION                                                    *
*                                                                   *
*     IR24DF REG=RGD,                                               *
*            (SUFFIX=X),                                            *
*            (ORG=IR24HDR)                                          *
*                                                                   *
*  4. GENERAL CONTENTS AND USAGE                                    *
*                                                                   *
*  4.1. ROLE IN SYSTEM                                              *
*                                                                   *
*       THIS FILE CONTAINS ALL PASSENGER RELATED INFORMATION.       *
```

Figure 25. DSECT to Define the Passenger File (Part 1 of 4)

```
*                                                                     *
*  4.2. DATA LAYOUT                                                   *
*                                                                     *
*       STANDARD TPFDF FILE HEADER                                    *
*                                                                     *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.    *
*                                                                     *
*       PRIMARY KEY     USAGE                                         *
*          70           NAME LOGICAL RECORD                           *
*          80           PASSENGER NUMBER                              *
*          90           ADDRESS                                       *
*          A0           FLIGHT INFORMATION                            *
*          B0           FACTS                                         *
*                                                                     *
*  4.3. PROGRAMMING ASPECTS                                           *
*                                                                     *
*  4.3.1. PROGRAMMING RESTRICTIONS                                    *
*                                                                     *
*         NONE.                                                       *
*                                                                     *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                            *
*                                                                     *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:               *
*         - PRIMARY KEY                                               *
*                                                                     *
*  5. STORAGE FACTORS                                                 *
*                                                                     *
*  5.1. BLOCK SIZE                                                    *
*                                                                     *
*       DEFINED IN DBDEF.                                             *
*                                                                     *
*  5.2. FILE REQUIREMENTS                                             *
*                                                                     *
*       POOL FILE (NUMBER OF FILES VARIES)                            *
*                                                                     *
*  5.3. ACCESSING SCHEME                                              *
*                                                                     *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND          *
*       CROSS RELATION OF LRECS ETC)                                  *
*       THE ACCESS PATHS 2,3 ARE USED TO RETRIEVE THE PASSENGER       *
*       FILE BY FLIGHT NUMBER AND DATE, PATH=0 BY PASSENGER NAME      *
*       AND PATH=1 BY PASSENGER NUMBER.                               *
*                                                                     *
*  6. DATA CONTROL                                                    *
*                                                                     *
*  6.1. CHAINING AND OVERFLOW                                         *
*                                                                     *
*       STANDARD TPFDF CHAINING.                                      *
*                                                                     *
*  6.2. DATA FIELD ADDRESSING                                         *
*                                                                     *
*       OFFSET WITHIN STANDARD TPFDF LREC.                            *
*                                                                     *
*  7. IMPLEMENTATION REQUIREMENTS                                     *
*                                                                     *
*  8. REFERENCES                                                      *
*                                                                     *
*  9. COMMENTS                                                        *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 25. DSECT to Define the Passenger File (Part 2 of 4)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR24DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR24DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*********************************************************************
*        STANDARD TPFDF HEADER                                      *
*********************************************************************
IR24HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR24VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR24HDL&CG1 EQU IR24VAR&CG1-IR24HDR&CG1  HEADER-LENGTH UP TO IR24VAR
            ORG IR24HDR&CG1
IR24REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR24SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR24KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR24DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)             *
*********************************************************************
.*                    USE KEY #IR24K80 IF ONLY ONE KEY
.*                    #IR24K00-#IR24K0F ARE RESERVED FOR TPFDF
.*                    #IR24KF0-#IR24KFF ARE RESERVED FOR TPFDF
#IR24K70 EQU   X'70'         LOGICAL RECORD KEY X'70'
#IR24K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR24K90 EQU   X'90'         LOGICAL RECORD KEY X'90'
#IR24KA0 EQU   X'A0'         LOGICAL RECORD KEY X'A0'
#IR24KB0 EQU   X'B0'         LOGICAL RECORD KEY X'B0'
#IR24L70 EQU   IR24E70&CG1-IR24REC&CG1  LENGTH OF LOGICAL RECORD X'70'
#IR24L80 EQU   IR24E80&CG1-IR24REC&CG1  LENGTH OF LOGICAL RECORD X'80'
#IR24L90 EQU   IR24E90&CG1-IR24REC&CG1  LENGTH OF LOGICAL RECORD X'90'
#IR24LA0 EQU   IR24EA0&CG1-IR24REC&CG1  LENGTH OF LOGICAL RECORD X'A0'
#IR24LB0 EQU   IR24EB0&CG1-IR24REC&CG1  LENGTH OF LOGICAL RECORD X'B0'
&IR24DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR24ORG&CG1 EQU *            START VARIABLE DATA PER LREC
.*
*********************************************************************
*        PASSENGER NAME LOGICAL RECORD                              *
*********************************************************************
IR24NAM&CG1 DS  CL25         PASSENGER NAME
IR24E70&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'70'
.*
         ORG IR24ORG&CG1
*********************************************************************
*        PASSENGER NUMBER LOGICAL RECORD                            *
*********************************************************************
IR24NUM&CG1 DS  CL8          PASSENGER NUMBER
IR24E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
.*
         ORG IR24ORG&CG1
*********************************************************************
*        ADDRESS LOGICAL RECORD                                     *
*********************************************************************
IR24ADR&CG1 DS  CL50         PASSENGER ADDRESS
IR24E90&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'90'
.*
         ORG IR24ORG&CG1
```

Figure 25. DSECT to Define the Passenger File (Part 3 of 4)

```
*********************************************************************
*        FLIGHT INFORMATION LOGICAL RECORD                          *
*********************************************************************
IR24FLI&CG1 DS  0CL17        FLIGHT INFORMATION
IR24FLT&CG1 DS  CL7          FLIGHT NUMBER
IR24DAT&CG1 DS  XL2          DATE
IR24TIM&CG1 DS  XL2          TIME
IR24ORI&CG1 DS  CL3          ORIGIN (START)
IR24DES&CG1 DS  CL3          DESTINATION
IR24EA0&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'A0'
.*
         ORG IR24ORG&CG1
*********************************************************************
*        FACTS LOGICAL RECORD                                       *
*********************************************************************
IR24FAC&CG1 DS  CL4          FLIGHT INFORMATION
IR24EB0&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'B0'
.*
         ORG IR24ORG&CG1
*********************************************************************
*        ALGORITHM DESCRIPTION                                      *
*********************************************************************
         ORG IR24REC&CG1
IR24@0BEG&CG1 EQU *             PATH 0 DESCRIPTION
IR24@0NAM&CG1 DS  CL25          PASSENGER NAME
IR24@0END&CG1 EQU *
         ORG IR24REC&CG1
IR24@1BEG&CG1 EQU *             PATH 1 DESCRIPTION
IR24@1NUM&CG1 DS  CL8           PASSENGER NUMBER
IR24@1END&CG1 EQU *
         ORG IR24REC&CG1
IR24@2BEG&CG1 EQU *             PATH 2 DESCRIPTION
IR24@2DAY&CG1 DS  XL4           DAY
IR24@2FLN&CG1 DS  CL7           FLIGHT NUMBER
IR24@2PAN&CG1 DS  CL25          PASSENGER NAME
IR24@2END&CG1 EQU *
         ORG IR24REC&CG1
IR24@3BEG&CG1 EQU *             PATH 3 DESCRIPTION
IR24@3END&CG1 EQU *
.*************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 25. DSECT to Define the Passenger File (Part 4 of 4)

DBDEF

Figure 26, Figure 27, and Figure 28 show the index key definitions for paths 0, 1, and 2 respectively.

Index Keys for Each Path

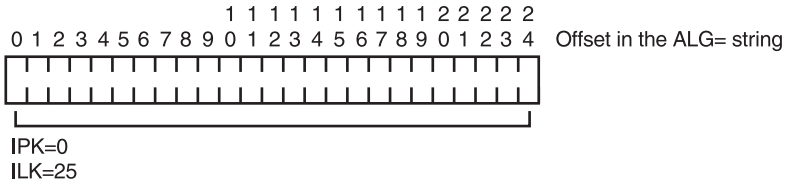


Figure 26. Index Key Definitions for Path 0, IR20DF to IR24DF

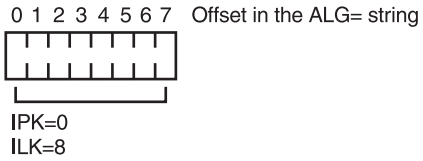


Figure 27. Index Key Definitions for Path 1, IR21DF to IR24DF

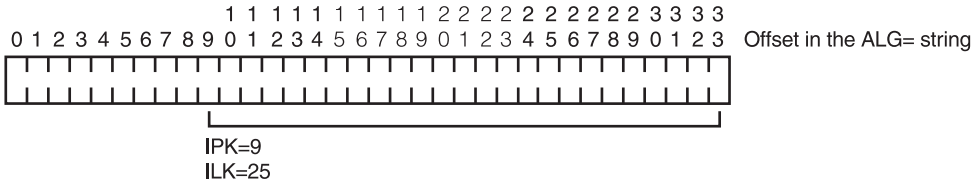


Figure 28. Index Key Definitions for Path 2, IR23DF to IR24DF

Figure 29 shows the position of IR24DF in the file structure.

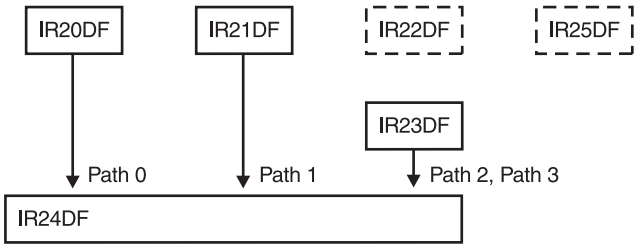


Figure 29. Position of IR24DF in the File Structure

```
DBDEF FILE=IR24DF,                        Passenger File         -
      (PKY=#IR24K70,                                             -
      KEY1=(PKY=#IR24K70,UP),                                    -
      KEY2=(R=IR24NAM,UP)),               Passenger Name         -
      (PKY=#IR24K80,                                             -
      KEY1=(PKY=#IR24K80,UP),                                    -
      KEY2=(R=IR24NUM,UP)),               Passenger Number       -
      (PKY=#IR24K90,                                             -
      KEY1=(PKY=#IR24K90,UP)),                                   -
      (PKY=#IR24KA0,                                             -
      KEY1=(PKY=#IR24KA0,UP),                                    -
      KEY2=(R=IR24DAT,UP),                Date                   -
      KEY3=(R=IR24TIM,UP),                Time                   -
      KEY4=(R=IR24FLT,UP)),               Flight                 -
      (PKY=#IR24KB0,                                             -
      KEY1=(PKY=#IR24KB0,UP)),                                   -
      (IID=IR20DF,PTH=0,IKY=80,IPA=0,ILA=1,IPK=0,ILK=25,         -
      KEY1=(PKY=#IR20K80,UP),                                    -
      KEY2=(R=IR20PNM,S=0,UP)),           Passenger Name         -
```

```
(IID=IR21DF,PTH=1,IKY=80,IPA=0,ILA=8,IPK=0,ILK=8,        -
KEY1=(PKY=#IR21K80,UP),                                  -
KEY2=(R=IR21NBR,S=0,UP)),          Passenger Number      -
(IID=IR23DF,PTH=2,IKY=80,IPA=0,ILA=0,IPK=9,ILK=25,       -
KEY1=(PKY=#IR23K80,UP),                                  -
KEY2=(R=IR23PNA,S=7,UP)),          Passenger Name        -
(IID=IR23DF,PTH=3,IKY=80,                                -
KEY1=(PKY=#IR23K80,UP))            All Passengers
```

DSECT and DBDEF for the Aircraft File

Macro IR25DF is the DSECT for the aircraft file. The following example shows the DSECT and the DBDEF for this index file.

DSECT

Figure 30 shows the DSECT used to define the aircraft file.

```
         MACRO
&LABEL   IR25DF &REG=,&SUFFIX=,&ORG=,&ACPDB=
.*
*********************************************************************
*                                                                   *
*   IR25DF      AIRCRAFT FILE                                       *
*                        DATE: 14JUL90                              *
*                                                                   *
*********************************************************************
         GBLB  &IR25DF1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  'IR25DF       '    DOC NAME
&DATE    SETC  '08FEB91'     UPDATE DATE
&VERS    SETC  '00'          VERSION NUMBER
.*********************************************************************
.*      DEFINITIONS FOR TPFDF                                        *
.*********************************************************************
&SW00WID SETC  '25'          FILE ID
&SW00WRS SETC  'L2'          BLOCK SIZE
&SW00ARS SETC  'L2'          ALTERNATE BLOCK SIZE
&SW00RCT SETC  '#MISC4'      FACE FILE TYPE
&SW00BOR SETC  '#IR25DFF'    BASE ORDINAL NUMBER
&SW00EOR SETC  '#IR25DFL'    END ORDINAL NBR
&SW00RBV SETC  '#TPFDB04'    RBV TYPE
&SW01EO# SETC  '&SW00EOR'    RECOUP END ORDINAL
&SW02FIL SETC  'IR25DF'      FILE DSECT NAME
&SW00OP1 SETC  '00000000'    OPT BYTE1
&SW00OP2 SETC  '00000110'    OPT BYTE2
&SW00OP3 SETC  '00000000'    OPT BYTE3
&SW00TQK SETC  '15'          HIGHEST TLREC
.*********************************************************************
         COPY  DBCOD         COPY DSECT DEFINITION FUNCTIONS
         AIF   ('&IR25DF1' EQ '1').NOT1ST
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                   *
*  DESCRIPTION OF IR25DF                                            *
*                                                                   *
*  1. DATA AREA NAME                                                *
*                                                                   *
*     AIRCRAFT FILE                                                 *
*                                                                   *
*  2. MEMBER NAME                                                   *
*                                                                   *
*     IR25DF                                                        *
*                                                                   *
*  3. INVOCATION                                                    *
*                                                                   *
*     IR25DF REG=RGD,                                               *
*            (SUFFIX=X),                                            *
*            (ORG=IR25HDR)                                          *
*                                                                   *
```

Figure 30. DSECT to Define the Aircraft File (Part 1 of 3)

```
*  4. GENERAL CONTENTS AND USAGE                                      *
*                                                                     *
*  4.1. ROLE IN SYSTEM                                                *
*                                                                     *
*       CONTAINS CONFIGURATION INFORMATION FOR THE VARIOUS AIRCRAFT *
*       TYPES.                                                        *
*                                                                     *
*  4.2. DATA LAYOUT                                                   *
*                                                                     *
*       STANDARD TPFDF FILE HEADER                                    *
*                                                                     *
*       ABV.: CREATOR (C), USERS (U), AND PURGER (P) OF EACH LREC.  *
*                                                                     *
*       PRIMARY KEY     USAGE                                         *
*          80           AIRCRAFT TYPE LOGICAL RECORD                  *
*                                                                     *
*  4.3. PROGRAMMING ASPECTS                                           *
*                                                                     *
*  4.3.1. PROGRAMMING RESTRICTIONS                                    *
*                                                                     *
*         NONE.                                                       *
*                                                                     *
*  4.3.2. PROGRAMMING TECHNIQUES AND USAGE                            *
*                                                                     *
*         STANDARD TPFDF LREC LOCATION TECHNIQUE USING:               *
*         - PRIMARY KEY                                               *
*                                                                     *
*  5. STORAGE FACTORS                                                 *
*                                                                     *
*  5.1. BLOCK SIZE                                                    *
*                                                                     *
*       DEFINED IN DBDEF.                                             *
*                                                                     *
*  5.2. FILE REQUIREMENTS                                             *
*                                                                     *
*       1 #MISCELLANEOUS FILE                                         *
*                                                                     *
*  5.3. ACCESSING SCHEME                                              *
*                                                                     *
*       (DESCRIBE ALGORITHM, PATHS, UP/DOWN ORGANIZATION AND          *
*       CROSS RELATION OF LRECS ETC)                                  *
*       THERE IS NO SPECIAL ACCESS METHOD REQUIRED. THERE IS ONLY     *
*       ONE FILE.                                                     *
*                                                                     *
*  6. DATA CONTROL                                                    *
*                                                                     *
*  6.1. CHAINING AND OVERFLOW                                         *
*                                                                     *
*       STANDARD TPFDF CHAINING.                                      *
*                                                                     *
*  6.2. DATA FIELD ADDRESSING                                         *
*                                                                     *
*       OFFSET WITHIN STANDARD TPFDF LREC.                            *
*                                                                     *
*  7. IMPLEMENTATION REQUIREMENTS                                     *
*                                                                     *
*  8. REFERENCES                                                      *
*                                                                     *
*  9. COMMENTS                                                        *
*                                                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure 30. DSECT to Define the Aircraft File (Part 2 of 3)

```
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#IR25DFS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#IR25DFI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
*********************************************************************
*        STANDARD TPFDF HEADER                                      *
*********************************************************************
IR25HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
IR25VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR25HDL&CG1 EQU IR25VAR&CG1-IR25HDR&CG1  HEADER-LENGTH UP TO SAM2VAR
            ORG IR25HDR&CG1
IR25REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
IR25SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR25KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR25DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)             *
*********************************************************************
.*                    USE KEY #IR25K80 IF ONLY ONE KEY
.*                    #IR25K00-#IR25K0F ARE RESERVED FOR TPFDF
.*                    #IR25KF0-#IR25KFF ARE RESERVED FOR TPFDF
#IR25K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR25L80 EQU   IR25E80&CG1-IR25REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR25DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR25ORG&CG1 EQU *            START OF LOGICAL RECORD DESCRIPTION
.*
*********************************************************************
*        DESCRIPTION OF  F I R S T  LOGICAL RECORD TYPE             *
*********************************************************************
IR25ACT&CG1 DS  CL4          AIRCRAFT TYPE
IR25STR&CG1 DS  0XL4         SEAT RANGE
IR25SOR&CG1 DS  XL2          START OF RANGE
IR25EOR&CG1 DS  XL2          END OF RANGE
IR25CLA&CG1 DS  CL1          CLASS INFORMATION
IR25E80&CG1 EQU *            END OF LOGICAL RECORD X'80'
.*
.*********************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 30. DSECT to Define the Aircraft File (Part 3 of 3)

DBDEF

Figure 31 shows the position of IR25DF in the file structure.

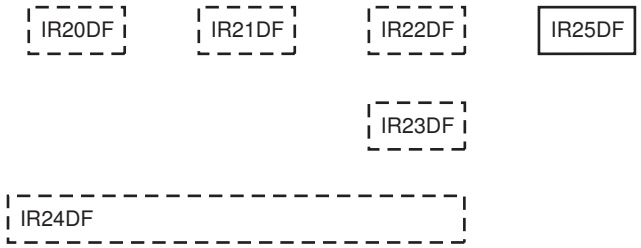


Figure 31. Position of IR25DF in the File Structure

```
DBDEF FILE=IR25DF,             -
      (PKY=#IR25K80,           -
      KEY1=(PKY=#IR25K80,UP),  -
      KEY2=(R=IR25CLA,UP))
```

Part 2. Creating the DSECT and DBDEF Macros

Creating a DSECT Macro Definition

You must create a DSECT macro definition to describe each file that you are going to access with an application program. Some sample DSECT macro definitions that can help you in this task are included as part of the TPFDF product. Copy these DSECTs and modify one or more of the copies to create DSECTs of your own.

Sample DSECT Macros Supplied with the TPFDF Product

Table 33 lists the sample DSECT macros that are provided with the TPFDF product.

Table 33. Sample DSECT Macros

	Fixed-Length LRECs or Variable-Length LRECs	**Extended LRECs**	**Extended LRECs with Unique Key Support**
R-type B+Tree index file	SAMTSR		
R-type fixed file	SAM1SR	SAM6SR	SAMESR
R-type pool file	SAM2SR	SAM7SR	SAMFSR
W-type file	SAM3SR	SAM8SR	
T-type file	SAMCSR		
R-type top-level index file (fixed)	SAM4SR	SAMISR	SAMHSR
R-type lower-level indexed file (pool)	SAM5SR	SAMKSR	SAMGSR

File Names

The TPFDF product uses the DSECT name to identify a file in application programs. Figure 32 shows a convention for the 6-character file (and DSECT) name that is based on the International Passenger Airline Reservation System (IPARS) standard. While you do not have to use the IPARS standard, the TPFDF product DSECT names must be 6 characters long and each character must adhere to any rules stated in the descriptions that follow.
Each character is defined as follows:

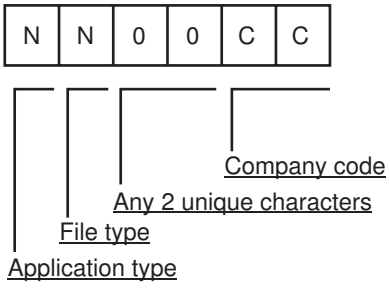


Figure 32. Syntax of a DSECT Macro File Name

1st The first character can be any alphabetic character that defines the application type. If you are using the International Passenger Airline Reservation System (IPARS) standard, the first character identifies one of the following application types:

A Accounting

C Cargo

F Fare quote/ticketing

G General functions

M Message switching

O Operation

Q Communication middleware

R Passenger reservation

S System software

W Departure control system

X, Y, and Z
Reserved for future use.

Optionally, you can set the first character of the DSECT macro name to any alphanumeric character and use the DBDEF APL parameter to specify the type of application. See the APL parameter description on page 122.

2nd The second character is alphabetic. It identifies the type of file:

P P-type files are customer-format files that do not have to follow the same standards as standard-format files (R, W, and T). Because P-type files do not have to follow one standard format, you must include more information when coding TPFDF macros, functions, and utilities than with standard-format files. P-type files do not contain logical records (LRECs); an entire block acts as an LREC.

R Real-time files. This means all types of application data files (whether stored in fixed or in pool prime blocks), excluding W-type files.

W Work file that only lasts the lifetime of the entry control block (ECB).

T Temporary LREC stored in a W-type file.

Optionally, you can set the second character of the DSECT macro name to any alphanumeric character and use global set symbol &SW00TYP to specify the file category. See the &SW00TYP description on page 82.

3rd and 4th The third and fourth characters can be any combination of alphanumeric characters that you choose to identify the file uniquely. You can set these to the file ID (if this consists of 2 alphanumeric characters).

5th and 6th The fifth and sixth characters can be any combination of alphanumeric characters that you choose to identify the file. If you are using the IPARS standard, the fifth and sixth characters represent a 2-character company code.

Modifying the Sample DSECT Macros

This section describes how to copy and modify one of the sample DSECTs to create a DSECT to match your own requirements.

Modifying the Beginning DSECT Macro Statements

Figure 33 shows the beginning macro statements in the sample code for SAM3SR. The beginning statements, before the **Definitions for TPFDF** section, are always required. Do not change or delete them, except to:

- Change the name of the DSECT macro according to naming conventions
- Modify the &NAM, &DATE, and &VERS symbols as necessary.

```
         MACRO                     LIBR NAME = SAM3SR09
&LABEL   SAM3SR &REG=,&SUFFIX=,&ORG=,&ACPDB=
*************************************************************************
*                                                                       *
*   $TITLE$     SAMPLE DSECT FOR TPFDF W-TYPE BASIC                     *
*   CREATED EX: $CREATE$     DATE: $CDATE$                              *
*                                                                       *
*                                                                       *
*  CHANCE HISTORY  DATE      DESCRIPTION                                *
*                                                                       *
*************************************************************************
         GBLB  &SAM3SR1      1ST TIME CALLED SWITCH
         COPY  DBGBL         COPY TPFDF GLOBAL DEFINITIONS
         COPY  DBLCL         COPY TPFDF LOCAL DEFINITIONS
&NAM     SETC  '             '    DOC NAME
&DATE    SETC  '17DEC85'     UPDATE DATE
&VERS    SETC  '03'          VERSION NUMBER
```

Figure 33. Instructions Always Required at the Start of a DSECT Macro Definition

Assigning Values to Global Set Symbols

As shown in Figure 34, after the **Definitions for TPFDF** section in the SAM3SR sample code, there are a series of instructions that assign values to several global set symbols. Assign values to each global set symbol that you need to define. For example:

```
&SW00WID SETC               FILE ID
&SW00WRS SETC               BLOCK SIZE
&SW00ARS SETC               ALTERNATE BLOCK SIZE
&SW02FIL SETC  'SAM3SR'     FILE DSECT NAME
&SW00OP1 SETC  '00000000'   OPT BYTE1
&SW00OP2 SETC  '00000110'   OPT BYTE2
&SW00OP3 SETC  '00000000'   OPT BYTE3
&SW00TQK SETC  '15'         HIGHEST TLREC
```

Figure 34. Instructions to Assign Values to Global Set Symbols

The global set symbols are described here in alphabetic order.

Notes:

1. Do not assign a value to a particular global set symbol if it is not required. For example, do not assign values to &SW00BOR, &SW00EOR, and &SW00RCT for pool files.
2. Do not change any values except &SW00WID and &SW00WRS in the B+Tree index file sample, SAMTSR.
3. You can override some of the global set symbol values assigned in a DSECT macro by using a corresponding parameter in the DBDEF instructions or with a TPFDF macro or function.

&SW00ARS: Specify the Block Size of Overflow Blocks

&SW00ARS is optional. Use it to specify the size of any overflow blocks that are chained to the prime blocks of the file; for example:

```
&SW00ARS SETC  'L4'
```

Because overflow blocks are always pool blocks, &SW00ARS is the only definition of the overflow block size and does not have to correspond with any other value defined elsewhere (see the description for &SW00OP1 bit 5 on page 75).

If you do not specify &SW00ARS, the TPFDF product assigns a value equal to the value set for prime blocks, &SW00WRS.

For add current processing, when the number of chains (NOC) is greater than 0, &SW00ARS must be equal to the value defined for &SW00WRS.

For B+Tree index files, &SW00ARS is ignored. &SW00WRS supplies the block size of the nodes.

&SW00BOR: Specify the Begin Ordinal

&SW00BOR is required for fixed or miscellaneous files. Use &SW00BOR to specify the beginning ordinal for the file.

- If this is a fixed file, always set the base ordinal number to zero.

  ```
  &SW00BOR SETC '0'
  ```

- If this is a miscellaneous file, specify the symbol (usually defined in SYSEQ) that defines where the miscellaneous file starts in the miscellaneous fixed file type.

  By convention, this symbol is "#", followed by the DSECT macro name, followed by "F":

  ```
  &SW00BOR SETC '#dsnameF'
  ```

&SW00EOR: Specify the End Ordinal

&SW00EOR is required if this is a fixed or miscellaneous file. Use &SW00EOR to specify the end ordinal number for the fixed file.

- If this is a nonpartitioned fixed file, specify −1 (minus 1):

  ```
  &SW00EOR SETC '-1'
  ```

  The TPFDF product resolves the correct value when the file is opened.

- If this is a miscellaneous file, specify the symbol (usually defined in SYSEQ), that defines where the miscellaneous file ends in the miscellaneous fixed file type. By convention, this symbol is "#", followed by the DSECT macro name, followed by "L":

  ```
  &SW00EOR SETC '#dsnameL'
  ```

- If this is a partitioned file, specify the number of prime blocks in each partition, not the total number of prime blocks in all partitions. For example, if there are 10 ordinals in each partition, specify:

  ```
  &SW00EOR SETC '10'
  ```

&SW01EO#: Specify the End Ordinal for TPFDF Recoup

&SW01EO# specifies the FACE-type end ordinal (for TPFDF recoup).

- If this is a nonpartitioned, noninterleaved fixed file, set as follows:

  ```
  SW01EO# SETC '-1'
  ```

- If the file is partitioned, set SW01EO# to −1 or the total number of prime blocks in all the partitions of the file.

- If the file is interleaved, set SW01EO# to −1 or the total number of prime blocks in all the interleaves of the file.
- If the file is miscellaneous, specify the symbol (usually defined in SYSEQ), which defines where the miscellaneous file ends in the miscellaneous fixed file type. By convention, this symbol is "#", followed by the DSECT macro name, followed by "L":

  ```
  &SW01EO# SETC '#dsnameL'
  ```

Notes:

1. If &SW01EO# is not specified in the DSECT or the DBDEF EO# parameter override, −1 is used for fixed file types.
2. If &SW01EO# is not specified in the DSECT or the DBDEF EO# parameter override, the &SW00BOR value or the DBDEF BOR parameter override, is used for miscellaneous files.
3. Miscellaneous file type prefixes are defined in the TPFDF product with the &MISTYPE(n) array in segment DBLCL. You can define as many as 20 prefixes (including those shipped with the TPFDF product). For more information about the DBLCL segment, see *TPFDF Installation and Customization*.

&SW02FIL: Specify the File Name

Set &SW002FIL to the DSECT macro name. For example:

```
&SW002FIL SETC 'dsname'
```

&SW00ILV: Specify the Number of Interleaves

&SW00ILV is optional. If you set this to a nonzero value, it implies that the file is interleaved. The number you assign defines the number of interleaves. For example:

```
&SW00ILV SETC '3'
```

&SW00NLR: Specify Number of Fixed-Length LRECs in Each Block

&SW00NLR is optional. Only assign a value to &SW00NLR when the file uses algorithm #TPFDB0D. With this algorithm, the file contains only prime blocks. All LRECs must be fixed-length. Set &SW00NLR to the maximum number of LRECs that can fit into each prime block of the file.

Notes:

1. The global set symbol &SW00NLR is not allowed for B+Tree data files.
2. Using the global set symbol &SW00NLR will result in an MNOTE being issued because the TPFDF product automatically calculates this value, and any value specified is ignored.

&SW00NOC: Specify the Number of Blocks to Use in Implementing Add Current Files

Only assign a value to &SW00NOC when bit 2 of &SW00OP1 is set on. Setting bit 2 of &SW00OP1 instructs the TPFDF product to limit the number of overflow blocks used when adding LRECs to the subfile. You can set the limit (using &SW00NOC) from zero to a maximum of 255 blocks. When this limit is reached in the subfile, the TPFDF product discards the contents of the oldest chain block and copies the LRECs from the first chain block to the prime block. The first (empty) chain block is moved to the last chain block with an initial next available byte (NAB) setting. A new LREC is added to the last (empty) chain block. For example:

```
&SW00NOC SETA 5
```

instructs the TPFDF product to use the prime block and 5 overflow blocks.

If global set symbol &SW00NOC is set to zero, the TPFDF product uses only the prime block of the subfile.

Note: The &SW00NOC parameter is not allowed for B+Tree data files.

&SW00OP1: Specify Processing Options

&SW00OP1 contains 8 bits that specify TPFDF processing options.

Notes:

1. In the TPFDF product, bit 0 is the most significant and bit 7 is the least significant.
2. Bit fields labeled 'reserved' have no meaning in the system. Do not set them.

Bit 0: Implement Full Backward Chaining

Setting bit 0 of &SW00OP1 indicates that you want full backward chaining.

Backward chaining causes extra I/O overhead each time an overflow block is inserted or deleted. Backward chaining is required if the DBDEF DELEMPTY parameter is coded as YES. Backward chaining is recommended if an application program uses the DBRED macro with the BACKWARD parameter or the `dfred` function with the DFRED_BACKWARD option. If backward chaining is not defined, and an attempt is made to read the file backwards, the action taken by the TPFDF product is determined by the &DB013A symbol in the DBLCL macro.

Note: The value of &SW00OP1 bit 0 has no meaning for files that use add current processing (&SW00OP1 bit 2 set to 1) with no chains (&SW00NOC equal to 0).

See *TPFDF Programming Concepts and Reference* for more information about reading backward. See *TPFDF Installation and Customization* for more information about the DBLCL macro.

Bit 1: Implement Automatic Chain Correction

If you set bit 1 of &SW00OP1, the TPFDF product performs automatic chain correction if it detects a broken chain when you use a macro or function to access an LREC.

If you have set this indicator and the TPFDF product detects an incorrect forward chain pointer, it truncates the chain. Data may be lost in these circumstances, so set bit 1 of &SW00OP1 only for files where missing data is not critical.

Bit 2: Implement Add Current Files (Discarding Old Chain Blocks)

Setting bit 2 of &SW00OP1 indicates that you want to limit the number of overflow blocks used when adding LRECs.

When this limit is reached, the TPFDF product discards the contents of the oldest chain block and copies the LRECs from the first chain block to the prime block. The empty first chain block is moved to the last chain block with an initial next available byte (NAB) setting. You can set the limit using the &SW00NOC global set symbol from zero to 255.

If you set the &SW00NOC global set symbol to zero, the TPFDF product restricts processing to the prime block only. This feature is useful if you want to keep a limited amount of current data; for example, a log of the last series of macros issued by the program.

Notes:

1. Setting bit 2 of &SW00OP1 is not allowed with B+Tree data files.
2. Do not set bit 2 of &SW00OP1 for a file that has index LRECs in a basic indexing structure.

Bit 3: Implement Pushdown Chaining

If you set bit 3 of &SW00OP1, this specifies *pushdown chaining*. This works as follows:

1. When an application program uses the DBADD macro to add a new LREC, the TPFDF product stores the LREC starting at the next available byte in the prime block if there is room.
2. If the prime block does not have enough room left for the new LREC, the TPFDF product gets a new overflow block. It copies the contents of the prime block into the new overflow block, initializes the prime block, and puts the next LREC into the start of the prime block which is now empty.

Notes:

1. If you use this feature, the file cannot be UP or DOWN organized.
2. Setting bit 3 of &SW00OP1 is not allowed with B+Tree data files.

Bit 4: Checkpoint after Each DBUKY Macro

If you set bit 4 of &SW00OP1, the TPFDF product performs a checkpoint automatically whenever the application program uses the DBUKY macro. A checkpoint after each DBUKY macro call is often useful because application programs typically call DBUKY as the first step of a sequence of related updates to a file.

Bit 5: Specify Variable Sizes of Overflow Block

If bit 5 of &SW00OP1 is not set, the TPFDF product uses the value you set in parameter &SW00ARS to determine the size of all overflow blocks.

If you set bit 5 of &SW00OP1, the TPFDF product uses the size you have specified for prime blocks (&SW00WRS) for the first overflow block and the size you have specified for overflow blocks (&SW00ARS) for all other overflow blocks.

You must have specified the size of overflow blocks (&SW00ARS) to be greater than the size of prime blocks (&SW00WRS) for this bit setting to be effective.

Bit 6: Check Whether Blocks Should Be Packed

If you set bit 6 of &SW00OP1, the TPFDF product always checks for the packing criteria specified by &SW00PIN in every block that has been modified, regardless of whether the DBDEL macro or `dfdel` function has been used to delete LRECs. For example, the DBREP macro or `dfrep` function may have been used to replace an LREC with a smaller LREC.

Notes:

1. This bit setting creates processing overhead.
2. Setting bit 6 of &SW00OP1 is not allowed with B+Tree data files.

Bit 7: Maintain a File Sequence Update Counter

If you set bit 7 of &SW00OP1, the TPFDF product maintain a file sequence update counter in the prime block of the file.

Use this option with the DBRST command or `dfrst` function and sequence parameters.

&SW00OP2: Specify Processing Options

&SW00OP2 contains 8 bits that specify TPFDF processing options.

Notes:

1. In the TPFDF product, bit 0 is the most significant and bit 7 is the least significant.
2. Bit fields labeled 'reserved' have no meaning in the system. Do not set them.

Bit 0: Validate Next Available Byte (NAB)

When this bit is on, the TPFDF product validates the next available byte (NAB) every time it files a block.

Note: Using this feature creates processing overhead.

Bit 1: New Pool Option When a File Is Packed

By default, when the TPFDF product packs a file (when you close a file or use the ZUDFM OAP command, or a ZFCRU command with the pack function), it packs the file into the existing blocks, releasing any overflow blocks that are no longer required.

If you set bit 1 of &SW00OP2, new overflow blocks are used and old overflow blocks are released when a pack operation is completed successfully.

Bit 2: New Pool Option When a File Is Restored

By default, when the TPFDF product restores a file, it restores the file into existing blocks releasing any overflow blocks that are no longer required.

If you set bit 2 of &SW00OP2, new overflow blocks are used and old overflow blocks are released when a restore operation completes successfully.

Note: A B^+Tree index is always rebuilt using new pool records regardless of the bit setting.

Bit 3: New Pool Option When a File Is Loaded from Tape

By default, when the TPFDF product loads a file from tape (or for ALCS users, from sequential file), it loads the file into the existing blocks releasing any overflow blocks that are no longer required.

If you set bit 3 of &SW00OP2, new overflow blocks are used and old overflow blocks are released when a tape load operation completes successfully.

Bit 4: Retrieve Prime Blocks

Bit 4 should normally be set to zero. When bit 4 is set to zero and a hold is specified on a TPFDF call (for example, with the HOLD parameter on the DBOPN macro or the DFRED_INDEX_HOLD value on the `dfred` function), the TPFDF product issues the FIWHC macro to find and hold the prime block. Because open file processing does not perform an I/O, the FIWHC macro is processed on the subsequent TPFDF call that accesses the subfile.

When bit 4 is set to 1, the TPFDF product issues the FINWC macro to find the prime block regardless of whether a hold is specified or not, and the prime block is not held.

Note: Setting bit 4 to 1 allows two ECBs to update a file at the same time.

Bit 5: Retrieve Overflow Blocks

Bit 5 should normally be set to 1. When bit 5 is set to zero and a hold is specified on a TPFDF call (for example, with the HOLD parameter on the DBOPN macro or DFRED_INDEX_HOLD value on the `dfred` function), the TPFDF product issues the FIWHC macro to find and hold the overflow blocks. Because open file processing does not perform an I/O, the FIWHC macro is processed on the subsequent TPFDF call that accesses the subfile.

When bit 5 is set to 1, the TPFDF product issues the FINWC macro to find the overflow blocks regardless of whether a hold is specified or not, and the overflow blocks are not held.

Bit 6: Issue an OPR-DB010C System Error if the File is Modified without HOLD

Setting bit 6 to 1 causes the TPFDF product to issue an OPR-DB010C dump and return to the application program when a file that was opened without the HOLD parameter is modified and then closed. This prevents two ECBs from updating a file at the same time.

Note: If a file is opened with the HOLD parameter and &SW00OP2 bit 4 is set to 1, setting bit 6 to 1 will not prevent two ECBs from updating the file at the same time.

Bit 7: Reserved

This bit is reserved for IBM use.

&SW00OP3: Specify Processing Options

&SW00OP3 contains 8 bits that specify TPFDF processing options.

Notes:

1. In the TPFDF product, bit 0 is the most significant and bit 7 is the least significant.
2. Bit fields labeled 'reserved' have no meaning in the system. Do not set them.

Bit 0: LRECs Are of Extended Type

If you set bit 0 of &SW00OP3, this indicates to the TPFDF product that all LRECs in the file are extended LRECs.

Bit 1: Reserved

This bit is reserved for IBM use.

Bit 2: Reserved

This bit is reserved for IBM use.

Bit 3: File Is an Indexed Fixed File

You must set bit 3 of &SW00OP3 if the file you are specifying is a fixed file and it has one or more index files referencing it. See “Simple Indexing” on page 136.

Bit 4: Checkpoint When an Index LREC Is Added or Deleted

When the TPFDF product updates index LRECs, the LRECs are updated in main storage only. The TPFDF product only writes the index file to disk when the detail file is closed.

If you set bit 4 of &SW00OP3, the TPFDF product performs a checkpoint automatically whenever an index LREC is added or deleted from an index file.

Bit 5: Implement B+Tree Indexing
Set bit 5 of &SW00OP3 for files using B+Tree indexing. See "B+Tree Indexing" on page 149.

Note: Before you can implement B+Tree indexing in an ALCS environment, enable C language support. See *TPFDF Installation and Customization* for more information.

Bit 6: Default DETAC Mode
When this bit is set, each time you open a subfile, the TPFDF product opens it in DETAC mode (as if you had used the DETAC parameter with the DBOPN macro or `dfopn` function).

Bit 7: Unique Key Feature
Set this bit if you want to allow application programs to generate unique keys using the DBUKY macro or `dfuky` function.

&SW00PIN: Specify the Packing Threshold in a Block
When you close a subfile after deleting LRECs, the TPFDF product packs the subfile if the number of bytes used for LRECs in any block falls below the percentage (as indicated by &SW00PIN) of total bytes available for LRECs and the optional block trailer. If any block falls below the packing threshold, the subfile is packed to the level defined by the PLI value in the DBDEF. Always specify a PLI value greater than the &SW00PIN value; otherwise, the file is always below the packing threshold and is packed continuously.

For example:

```
&SW00PIN SETC '60'
```

For non-B+Tree files, the default setting is 75% if either &SW00WRS or &SW00ARS is set to L4. Otherwise the default is 50%. If the PLI parameter is not specified in the DBDEF, the blocks are packed to 100%.

Note: B+Tree files have the following considerations:

- For B+Tree index files, the &SW00PIN value (defined in the DBDEF statement of the data file) applies to node blocks. If a node block drops below the specified percentage, the node block is balanced or combined with an adjacent node. The default value, which is the maximum for a B+Tree index file, is 50%.
- For B+Tree index files, the &SW00PIN value should be set to 0 in the DSECT or by using the PIN parameter on the DBDEF macro so node blocks are not accidentally packed by a utility.
- B+Tree data files are only packed using a utility or when the file is closed, there are no nodes in the B+Tree structure, and there are overflow blocks in the data file.

&SW00PTN: Specify the Number of Partitions
&SW00PTN is required for partitioned files. It specifies the number of partitions. For example:

```
&SW00PTN SETC '4'
```

&SW00RBV: Specify the Addressing Algorithm
&SW00RBV defines the algorithm you want to use with this file.

If the file is a detail pool file, set it to "#TPFDBFF" as follows:

```
&SW00RBV SETC '#TPFDBFF'
```

Direct Translation Algorithms

There are eight TPFDF algorithms that distribute LRECs to subfiles using a direct relation between the algorithm argument and the subfile (ordinal) number.

Using Your Own Distribution Method

If you want to allocate LRECs over subfiles yourself, there are two TPFDF algorithms that let you use your own distribution method.

Hashing Algorithms

Hashing algorithms distribute LRECs across any number of subfiles in a file. To get an even distribution:

- Use a prime number for the number of subfiles.
- Use a wide a range of numbers or characters in the algorithm argument.

Algorithm for Files Requiring No Overflow

The TPFDF product provides one algorithm, #TPFDB0D, which distributes LRECs across prime blocks only. Use this algorithm if you want to avoid chaining.

Because the TPFDF product does not have to search through a chain of overflow blocks, this algorithm provides one of the fastest ways of retrieving data, but only works with fixed-length LRECs. You access the data in this type of file using the LRECNBR parameter.

Single-Subfile Algorithm

The TPFDF product provides the #TPFDB04 algorithm for files that have a single subfile.

Algorithm for Basic and B+Tree Index Support

The TPFDF product provides the #TPFDBFF algorithm for basic and B+Tree index support.

Create Your Own Algorithm

To create your own (user-defined) algorithm, do the following:

- Specify an equate value in the range 256–511 in the ACPDBE macro
- Specify the user-defined algorithm in user exit UWBD
- Set the &SW00RBV symbol.

Note: See Table 34 for a list of the previously mentioned algorithms.

Table 34. Algorithms

Direct Translation	
#TPFDB01	Distributes LRECs across subfiles according to the first alphabetic character in the algorithm argument. It requires 26 subfiles; one for each letter of the alphabet.
#TPFDB02	Distributes LRECs among subfiles according to the first 2 alphabetic characters in the algorithm argument. It requires 676 subfiles, one for each combination of 2 alphabetic characters: `AA, AB, AC,...ML, MM, MN,...ZX, ZY, ZZ`
#TPFDB03	Distributes LRECs across subfiles according to the first 3 alphabetic characters in the algorithm argument. It requires 17 576 subfiles, one for each combination of 3 alphabetic characters: `AAA, AAB, AAC,...MOL, MOM, MON,...ZZX, ZZY, ZZZ`
#TPFDB06	Distributes LRECs across subfiles according to the first alphabetic or numeric character in the algorithm argument. It requires 36 subfiles, one for each letter of the alphabet and one for each numeric character (0–9).

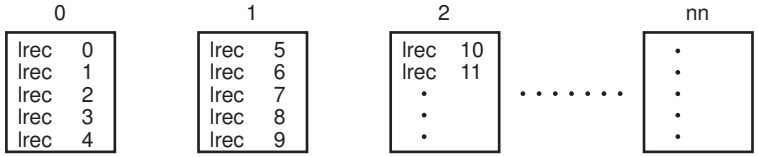
Table 34. Algorithms (continued)

#TPFDB07	Distributes LRECs across subfiles according to the first 2 alphabetic or numeric characters in the algorithm argument. It requires 1296 subfiles, one for each combination of 2 alphabetic or numeric characters: `AA,...A0,...A9,...ZA,...Z0,...Z9,...12,...5M,...99`
#TPFDB08	Distributes LRECs across subfiles according to the first 3 alphabetic or numeric characters in the algorithm argument. It requires 46 656 subfiles, one for each combination of 3 alphabetic or numeric characters: `AAA,...AAZ,...AA0,...AA9,...A00,...A09,...A99,...M56,...Z90,...123,...4T6,...999`

#TPFDB0A	Distributes LRECs among subfiles according to the first alphabetic, numeric, or special character in the algorithm argument. It requires 43 subfiles, one for each letter of the alphabet, one for each numeric character, and one for each of the following special characters:
	`. (X'4B')`
	`$ (X'5B')`
	`* (X'5C')`
	`- (X'60')`
	`/ (X'61')`
	`# (X'7B')`
	`@ (X'7C').`
	
	(The characters represented by these hexadecimal values can be different in some countries, so the TPFDF product uses the hexadecimal value not the character value (for example X'5B' not $).

#TPFDB0B	Distributes LRECs among subfiles according to the first 2 alphabetic, numeric, or special characters in the algorithm argument. The special characters are as listed above for #TPFDB0A. It requires 1849 subfiles, one for each possible combination of 2 alphabetic, numeric, or special characters: `..,.$,...AA,...BB,...B5,...B9,...#A,....M,...*Z,...$3,.../6,...99`
Your Own Distribution Method	
#TPFDB05	Use #TPFDB05 if the ordinal number that you specify will be held in a 4-byte field.
#TPFDB0C	Use #TPFDB0C if the ordinal number that you specify will be held in a 2-byte field. **Note:** These two algorithms use absolute ordinal values, not ordinals relative to &SW00BOR.
Hashing	
#TPFDB09	This hashing algorithm uses the first 8 bytes of data in the algorithm argument to determine in which subfile the LREC should be placed. The algorithm argument can contain any hexadecimal values. The TPFDF product divides the first half of the 8-byte string by the number of subfiles, then divides the second half by the number of subfiles. The two remainders are added together and then divided again by the number of subfiles. The remainder is the subfile to be retrieved. This produces an even distribution of LRECs among the subfiles.
#TPFDB0F	This algorithm is for partitioned files. It uses a 10-byte algorithm argument. The TPFDF product processes the first 8 bytes in the algorithm argument in the same way as #TPFDB09. It uses the next 2 bytes in the algorithm argument to distribute LRECs between partitions on a similar basis.
#TPFDB10	This hashing algorithm uses the first 8 bytes of data in the algorithm argument to determine in which subfile the LREC should be placed. The calculated ordinal is the remainder of an 8-byte algorithm argument divided by a 4-byte number of subfiles. The algorithm argument can contain any hexadecimal value. The result is achieved as follows: The TPFDF product divides the first 4 bytes of the 8-byte string by the number of subfiles. The second 4 bytes of the string are appended to the end of the remainder of the first division, and the entire resulting number is divided by 4 times the number of subfiles. The remainder of this division is then divided by the number of subfiles. The remainder is the subfile to be retrieved. This produces an even distribution of LRECs among the subfiles.
Files Requiring No Overflow	

Table 34. Algorithms (continued)

#TPFDB0D	#TPFDB0D is suitable for applications that process data sequentially by number or in circumstances where no pool storage is available. #TPFDB0D is not allowed for B+Tree data files. #TPFDB0D uses 4 bytes of data from the LREC number to determine where the LREC should be stored. It divides the 4-byte value by the number of LRECs in each block. The result is the relative ordinal number of the subfile in which the LREC is placed. The remainder gives the LREC number in the block. An example of such a file is shown in the following:  You do not need to specify how many LRECs are in each block when you use this algorithm. The TPFDF product calculates the number of LRECs in each block by dividing the length of the fixed-length LRECs into the block size.
Single Subfile	
#TPFDB04	Use this algorithm with files consisting of a single fixed subfile. #TPFDB04 does not perform any calculations to obtain a block ordinal number because there is only 1 prime block.
Basic and B+Tree Index Support	
#TPFDBFF	For basic indexing, this algorithm is used to access detail files. For B+Tree indexing, node files are defined using this algorithm.

&SW00RCT: Specify the Record Type for Fixed Files

&SW00RCT is required for fixed or miscellaneous files. Use &SW00RCT to specify the file's record type.

```
&SW00RCT SETC '#IDFCS'
```

&SW00REF: Specify the Work Area Reference Name

&SW00REF is valid for T-type files only. Use it to specify the reference name of the W-type file that contains the T-type file. The default is GW01SR.

```
&SW00REF SETC 'dsname'
```

Where *dsname* is the W-type file reference name.

&SW00SKE: Specify Block Index Support

To implement block index support, set &SW00SKE equal to the length in bytes of the key fields to be removed from the first LREC of each overflow block. The TPFDF product stores this in a TLREC (LREC ID = X'02'). The length you specify must include the *zzzz*KEY field (the LREC ID). For example, if the last field in the key is *zzzzlabel*, you would specify:

```
&SW00SKE SETC 'zzzzlabel-zzzzKEY+L''zzzzlabel'
```

Notes:

1. Make &SW00SKE large enough to hold any field used as a key field by any application program that might use this file.
2. If the file contains several different formats of LRECs, each identified by a separate LREC ID, set &SW00SKE equal to the size of the largest key that needs to be used.
3. Do not use block index support with add current files (these are indicated by bit 2 of &SW00OP1 being set).
4. &SW00SKE is not allowed for B+Tree data files.
5. To set up a block indexed index file, set the length of &SW00SKE from the LREC ID to the last key field; for example:

```
&SW00SKE SETC  'GR25E80-GR25KEY' EXTRACT FOR BLOCK INDEX
GR25REC&CG1 DS  0CL1
GR25SIZ&CG1 DS  H
GR25KEY&CG1 DS  XL1
GR25FAD&CG1 DS  CL4
GR25RCC&CG1 DS  XL1
GR25A80&CG1 DS  0XL6
GR25FID&CG1 DS  XL2
GR25FVN&CG1 DS  XL1
GR25IND&CG1 DS  XL1
GR25SSU&CG1 DS  XL2
GR25E80&CG1 EQU *
```

&SW00TQK: Specify the Highest Technical LREC (TLREC) ID that the TPFDF Product Can Use

You can use &SW00TQK to set the value of the highest technical LREC ID that the TPFDF product can use for this file. For block indexed files, it must be set to a value greater than 2. For B+Tree data files, it must be set to a value greater than 4. For B+Tree index files, it must be set to a value less than 3.

```
&SW00TQK SETC '15'
```

Note: Technical LREC IDs 1–15 are reserved for IBM. Except for B+Tree index files, set TQK=15 to avoid conflicts with the TPFDF product.

&SW00TYP: Specify the File Type

&SW00TYP is optional. If you omit it, the TPFDF product takes the file type from the second character of the DSECT macro name. These characters are listed in “File Names” on page 69.

You can set &SW00TYP equal to "R", "W", "T", or "P". B+Tree data files must be R-type files, whether specified or taken from the second character of the DSECT macro name.

If you are defining a DSECT macro to access a P-type file, set &SW00TYP equal to "P", as follows:

```
&SW00TYP SETC 'P'
```

&SW00WID: Specify the File ID

&SW00WID is always required. It specifies the *file ID*, which is a 2-byte value in the standard header of every block. Use a unique file ID for every DSECT macro. The range of valid file IDs is from #TPFDBID to X'FFFF' (where the value of #TPFDBID is defined in ACPDBE). For example:

```
&SW00WID SETC X'C000'
```

See *TPFDF Installation and Customization* for more information about #TPFDBID values.

&SW00WRS: Specify the Size of Prime Blocks

&SW00WRS is required. Set it to the size of the prime blocks used in the file. For example, to use a 1055-byte (L2) prime block, set &SW00WRS as follows:

```
&SW00WRS SETC  'L2'
```

File Description

The **Description of SAM3SR** section of the SAM3SR sample DSECT allows you to document your DSECT macro. Do not change the instructions that immediately follow the **Description of SAM3SR** section except to rename the DSECT to your new DSECT name. See Figure 35 on page 83.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*  DESCRIPTION OF SAM3SR                                                    *
*                                                                           *
*                                                                           *
*  A standard prolog follows that contains the following headings           *
*  which allow you to document information about the DSECT                  *
*                                                                           *
*                                                                           *
*  1. DATA AREA NAME                                                        *
*  2. MEMBER NAME                                                           *
*  3. INVOCATION                                                            *
*  4. GENERAL CONTENTS AND USE                                              *
*  5. STORAGE FACTORS                                                       *
*  6. DATA CONTROL                                                          *
*  7. IMPLEMENTATION REQUIREMENTS                                           *
*  8. REFERENCES                                                            *
*  9. COMMENTS                                                              *
*                                                                           *
*                                                                           *
*                                                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         EJECT
         AIF   ('&SW00WRS' EQ '').CHECKID
#SAM3SRS EQU   &SW00WRS      BLOCK SIZE
.CHECKID AIF   ('&SW00WID' EQ '').NOT1ST
#SAM3SRI EQU   C'&SW00WID'   FILE ID
.NOT1ST  ANOP
```

Figure 35. Instructions Always Required after Setting the Global Symbols in a DSECT Macro

Note: The #SAM3SRI equate in Figure 35 is set for a file ID that is in character format. If the file ID is in hexadecimal format, change the equate to:

```
#SAM3SRI EQU   X'&SW00WID'   FILE ID
```

Block Header

Do not change the instructions that define the block header unless you want to put extra information in the extended header in each prime block (see “Optimizing the Database Design” on page 11). You can do this after the standard TPFDF header field.

Figure 36 shows a block header for a DSECT that uses fixed-length or variable-length LRECs.

```
***************************************************************************
*        STANDARD TPFDF HEADER                                            *
***************************************************************************
SAM3HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDF HEADER
SAM3VAR&CG1 EQU *            START OF VARIABLE USER-AREA
SAM3HDL&CG1 EQU SAM3VAR&CG1-SAM3HDR&CG1  HEADER-LENGTH UP TO SAM3VAR
```

Figure 36. Instructions to Define the File Header in a DSECT Macro

Defining the LREC Size and LREC ID Fields

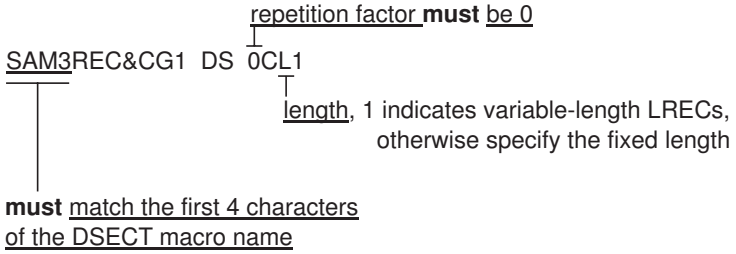
Figure 37 on page 84 shows an example of the instructions needed to define the logical record (LREC) size and the LREC ID fields for a file.

```
           ORG SAM3HDR&CG1
SAM3REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE SIZE)
SAM3SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
SAM3KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&SAM3SR1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
```

Figure 37. Instructions to Define the LREC Size and LREC ID Fields

The instructions are explained as follows:

SAM3REC&CG1
indicates the size of the LRECs for a file using fixed-length LRECs or it indicates that the file is is using variable-length LRECs.



SAM3SIZ&CG1
creates a 2-byte field that is used in each LREC to hold the actual length of the LREC (used for variable-length LRECs only). Make sure that you include the length of the **SIZ** field (2 bytes) and the **KEY** field (1 byte) when calculating the total length of the LREC.

SAM3KEY&CG1
creates a 1-byte field that contains the LREC ID. Its meaning and use are the same for fixed-length and variable-length LRECs.

Note: Because the SAM3**REC** definition occurs only once in a DSECT, you can define fixed-length or variable-length LRECs in a file, but not both.

Defining Different LREC IDs in the Same File

Every LREC contains an LREC ID, or primary key, that identifies it in the file. More than one LREC in a file can have the same ID. (It is possible for all LRECs in a file to have the same ID.) Different LREC IDs are used in a file to differentiate between different types of LREC.

Figure 38 shows how you can define two LREC IDs, X'80' and X'90'. The two equates are used later in the DSECT to define the two individual LREC IDs.

```
*********************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)
*********************************************************************
#SAM3K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#SAM3K90 EQU   X'90'         LOGICAL RECORD KEY X'90'
```

Figure 38. Defining Two Different LREC Types in a DSECT

Note: Do not use LREC IDs X'00'–X'0F' and X'F0'–X'FF'. LREC ID X'00' cannot be used and the other LREC IDs are reserved by the TPFDF product.

DSECT Instructions for Defining User Fields in LRECs

Define user fields by using DSECT instructions similar to those shown in Figure 39. Figure 39 defines a **member number** field that contains 10 characters of data and a **surname** field that contains 20 characters of data.

```
SAM3ORG&CG1 EQU *            START OF LOGICAL RECORD DESCRIPTION  *****

************************************************************************
*        DESCRIPTION OF  F I R S T  LOGICAL RECORD TYPE                *
************************************************************************
SAM3A80&CG1 DS  0CL30        KEYAREA
SAM3NUM&CG1 DS  CL10         member number
SAM3NAM&CG1 DS  CL20         surname
SAM3E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
```

Figure 39. Defining User Fields in a DSECT

Note: For index files, the TPFDF product determines how much data to move into an index LREC when a DBIDX macro or `dfidx` function is issued by calculating the number of bytes between the beginning of the key area and the end of the key area. In Figure 39, this would be 30 bytes (SAM3E80 minus SAM3A80).

Algorithm DSECT Statements

The algorithm area size must be defined for each detail file and any intermediate-index files, but is not required for top-level index files or files that are not part of a basic indexing structure. The size of the algorithm string is normally calculated using information in the DSECT of the detail or intermediate-index file. The reserved labels `dsnd@nBEG&CG1` and `dsnd@nEND&CG1` are used to indicate the beginning and end of the algorithm string.

Notes:

1. *dsnd* indicates the first 4 characters of the DSECT macro name of a detail or intermediate-index file.
2. *n* identifies the path number to which the definition applies.

Figure 40 shows these labels used in the SAM5SR sample DSECT.

```
************************************************************************
*        ALGORITHM DESCRIPTION                                         *
************************************************************************
         ORG SAM5REC&CG1
SAM5@0BEG&CG1 EQU *            PATH 0 DESCRIPTION
SAM5@0...&CG1 DS
SAM5@0...&CG1 DS
SAM5@0...&CG1 DS
SAM5@0END&CG1 EQU *
```

Figure 40. DSECT Code to Define the Algorithm String Size

Notes:

1. **@0** indicates that this applies to path zero (0).
2. The labels between the BEG and the END labels can be used to define parts of the algorithm string.

Ending DSECT Statements

See Figure 41 for the instructions required at the end of each sample DSECT macro. Do not change them.

```
.*
.*********************************************************************
         AIF   (&BG1).MACEXIT          GO IF INTERNAL USAGE
&SYSECT  CSECT
         AIF   ('&REG' EQ '').MACEXIT  GO IF REG= NOT SPECIFIED
.GEUSING ANOP                          GENERATE USING
         USING &DSN,&REG
.MACEXIT ANOP
         SPACE 1
         MEND
```

Figure 41. Instructions Always Required at the End of a DSECT Macro

Creating C Structures for Files with Existing DSECT Definitions

To allow a C program to access a file, create a C structure that reflects the LREC definitions in an existing DSECT. Figure 42 on page 87 shows an example of the LREC definitions for the IR71DF file and a C structure that allows a C program to access the file.

```
*************************************************************************
*                    IR71DF DSECT Definitions                           *
*************************************************************************
*************************************************************************
*        STANDARD TPFDB HEADER                                          *
*************************************************************************
IR71HDR&CG1 DS  CL16         STANDARD FILE HEADER
            DS  CL10         STANDARD TPFDB HEADER
IR71VAR&CG1 EQU *            START OF VARIABLE USER-AREA
IR71HDL&CG1 EQU IR71VAR&CG1-IR71HDR&CG1  HEADER-LENGTH UP TO IR
            ORG IR71HDR&CG1
IR71REC&CG1 DS  0CL1         1ST RECORD START (1=VARIABLE,ELSE
IR71SIZ&CG1 DS  H            SIZE OF LOGICAL RECORD
IR71KEY&CG1 DS  X            LOGICAL RECORD IDENTIFIER
         AIF   ('&IR71DF1' EQ '1').KEYEQ  GO IF NOT FIRST ISSUE
*************************************************************************
*        EQUATE OF LOGICAL RECORD KEYS (KEY AND LENGTH)                 *
*************************************************************************
#IR71K80 EQU   X'80'         LOGICAL RECORD KEY X'80'
#IR71L80 EQU  IR71E80&CG1-IR71REC&CG1  LENGTH OF LOGICAL RECORD X'80'
&IR71DF1 SETB  (1)           INDICATE 1ST TIME THROUGH
.KEYEQ   ANOP
IR71ORG&CG1 EQU *            START OF LOGICAL RECORD DESCRIPTION
*************************************************************************
*        DESCRIPTION OF  F I R S T  LOGICAL RECORD TYPE                 *
*************************************************************************
IR71A80&CG1 DS  0CL25        KEYAREA
IR71AK&CG1  DS  CL25         "A" KEY
IR71AL&CG1  DS  CL1          ALGORITHM BYTE
IR71TXT&CG1 DS  0CL25        FIXED TEXT
IR71DAT&CG1 DS  CL23         FIXED DATA
IR71RCN&CG1 DS  CL2          SEQUENTIAL RECORD NUMBER
IR71E80&CG1 EQU *            END OF LOGICAL RECORD WITH KEY = X'80'
         ORG IR71ORG&CG1
```

```
/*************************************************************************/
/*              IR71DF File C Structure Definitions                      */
/*************************************************************************/
_Packed struct ir71df
{
  short   ir71siz;                       /* Size of the record      */
  dft_pky ir71key;                       /* Primary key             */
  char    ir71ak[25];                    /* "A" key                 */
  char    ir71al;                        /* Algorithm byte          */
  char    ir71dat[23];                   /* Fixed data              */
  short   ir71rcn;                       /* Record sequence number  */
};
/*************************************************************************/
/*                  Miscellaneous Equates                                */
/*************************************************************************/
#define _IR71DFI  "\xB0\x71"
#define _IR71K80  0x80
#define _IR71L80  (sizeof(struct ir71df))
```

Figure 42. C Structure for a File with Existing DSECT Definitions

Creating a DBDEF Macro Definition

The DBDEF macro (along with the DSECT macro) defines the characteristics of a file; for example, whether the file uses an algorithm, indexed files, B+Tree indexing, or a combination of attributes. Segments UF2A–UF89 are reserved for customer DBDEF macro statements. To use the DBDEF macro statements that you code in these segments, include ENTRC statements in case 1 of the DFUEX segment. For example, if you have DBDEF macro statements coded in UF2A and UF2F, include the following statements in DFUEX:

```
ENTRC UF2A              USER SPECIFIC DB DEFINITIONS FOR UF2A
BR    R7                RETURN TO ROUTINE
ENTRC UF2F              USER SPECIFIC DB DEFINITIONS FOR UF2F
BR    R7                RETURN TO ROUTINE
```

Notes:

1. Reassemble UWA0 and UWA1 to use the changes made to DFUEX.
2. Except for segments UF90–UF9Z, if a file ID is used more than once on DBDEF statements, UWA1 sends a DB0131 system error and ignores the second file ID.
3. If a file ID is used twice and the second DBDEF statement is in segment UF90–UF9Z, the second DBDEF statement overrides the first.

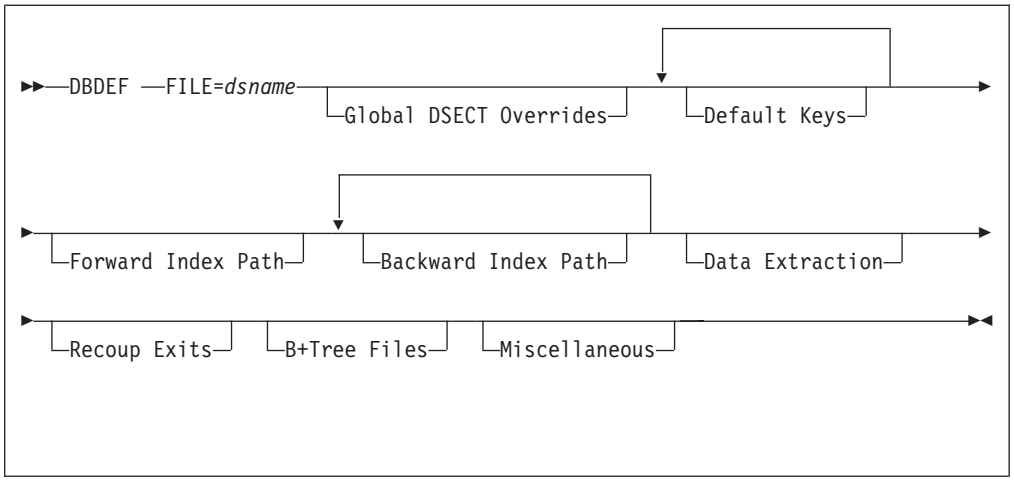
It is not necessary to specify data level independence (DLI) with the DBDEF macro. The TPFDF product preserves all data levels holding blocks before a macro or function call. See *TPFDF Programming Concepts and Reference* for more information about DLI.

DBDEF Macro Parameter Syntax

This section describes DBDEF macro statements according to the following functions:

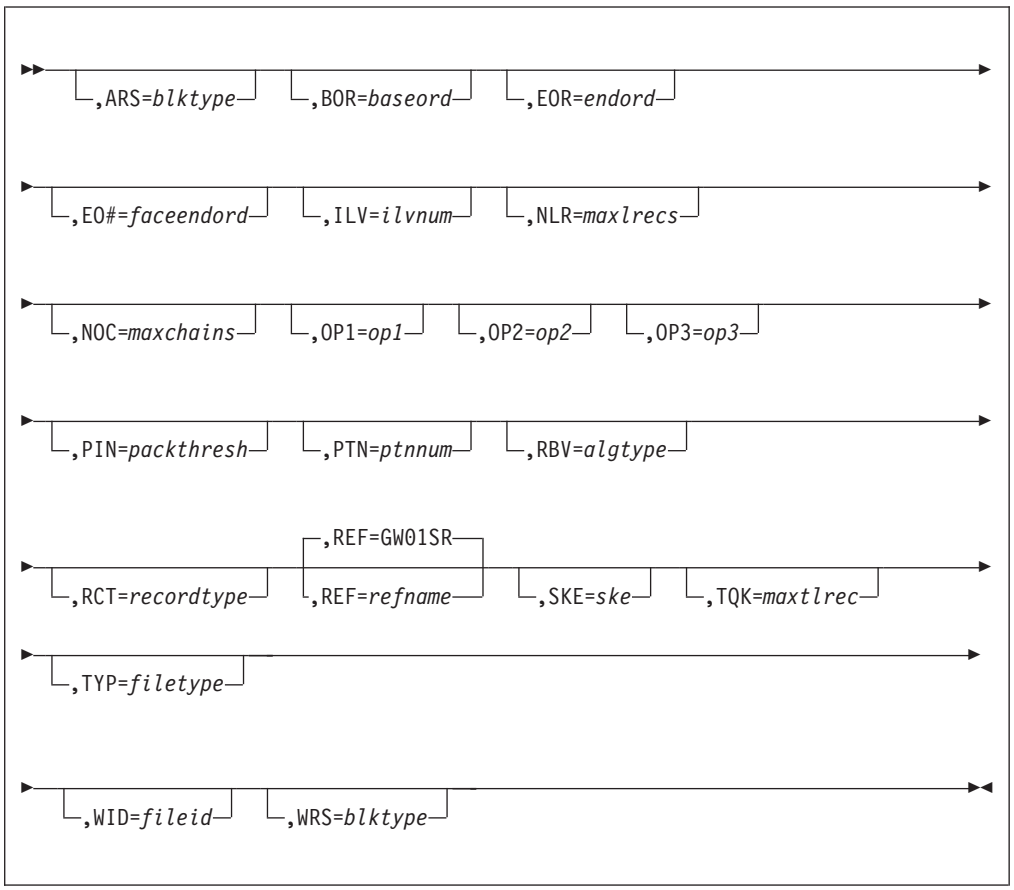
- Global DSECT overrides
- Default keys
- Indexing
- Data extraction
- Exits from TPFDF recoup to user code
- B+Tree files
- Miscellaneous.

The description of each function provides a syntax (railroad track) diagram for the macro and a description of each parameter and variable. See “How to Read the Syntax Diagrams” on page xiv for more information about syntax diagrams.



Global DSECT Override Parameters

These parameters override the equivalent SW00*var* statements found in the DSECT macro, where *var* equals the 3-digit global parameter (for example, ARS. See “Assigning Values to Global Set Symbols” on page 71 for more information about these parameters.



ARS=*blktype*

specifies the overflow block size for the referenced file, where *blktype* is one of the following block types:

L0 specifies a 128-byte block size.

L1 specifies a 381-byte block size.

L2 specifies a 1055-byte block size.

L3 specifies a 4000-byte block size. This block type is available only in an ALCS environment.

L4 specifies a 4095-byte block size.

L5 specifies a user-defined size. This block type is available only in an ALCS environment.

L6 specifies a user-defined size. This block type is available only in an ALCS environment.

L7 specifies a user-defined size. This block type is available only in an ALCS environment.

L8 specifies a user-defined size. This block type is available only in an ALCS environment.

For example:

```
ARS=L4
```

BOR=*baseord*

specifies the base relative ordinal for the file, where the value of *baseord* is as follows:

- If this is a fixed file, set *baseord* to zero.
- If this is a miscellaneous file, *baseord* is a symbol (usually defined in SYSEQ) that defines where the miscellaneous file starts in the miscellaneous fixed file type.

  By convention, this symbol is "#", followed by the DSECT macro name, followed by "F":

  ```
  BOR=#dsnameF
  ```

EOR=*endord*

specifies the end ordinal for the file, where the value of *endord* is as follows:

- If this is a nonpartitioned fixed file, set *endord* to −1 (minus 1). The TPFDF product resolves the correct value when the file is opened.
- If this is a miscellaneous file, *endord* is a symbol (usually defined in SYSEQ), that defines where the miscellaneous file ends in the miscellaneous fixed file type. By convention, this symbol is "#", followed by the DSECT macro name, followed by "L":

  ```
  EOR=#dsnameL
  ```

- If this is a partitioned file, specify the number of prime blocks in each partition, not the total number of prime blocks in all partitions. For example, if there are 10 ordinals in each partition, specify:

  ```
  EOR=10
  ```

Note: For more information about partitioning and interleaving, see “Partitioning and Interleaving” on page 157.

EO#=*faceendord*

specifies the FACE-type end ordinal (for TPFDF recoup), where the value of *faceendord* is as follows:

- If this is a nonpartitioned, noninterleaved, fixed file, set *faceendord* equal to −1.

- If the file is partitioned, set *faceendord* to −1 or the total number of prime blocks in all the partitions of the file.
- If the file is interleaved, set *faceendord* to −1 or the total number of prime blocks in all the interleaves of the file.
- If the file is miscellaneous, set *faceendord* equal to the symbol (usually defined in SYSEQ) that defines where the miscellaneous file ends in the miscellaneous fixed file type. By convention, this symbol is "#", followed by the DSECT macro name, followed by "L":

  ```
  EO#=#dsnameL
  ```

Notes:

1. If &SW01EO# is not specified in the DSECT or the DBDEF EO# parameter override, −1 is used for fixed file types.
2. If &SW01EO# is not specified in the DSECT or the DBDEF EO# parameter override, the &SW00BOR value or the DBDEF BOR parameter override, is used for miscellaneous files.
3. Miscellaneous file type prefixes are defined in the TPFDF product with the &MISTYPE(n) array in segment DBLCL. You can define as many as 20 prefixes (including those shipped with the TPFDF product). For more information about the DBLCL segment, see *TPFDF Installation and Customization*.
4. For more information about partitioning and interleaving, see “Partitioning and Interleaving” on page 157.

ILV=*ilvnum*
specifies the interleaving number, where *ilvnum* is a nonnegative decimal number. If *ilvnum* is a nonzero value, it implies that the file is interleaved. The number you assign defines the number of interleaves. For example:

```
ILV=3
```

Note: For more information about interleaving, see “Interleaves” on page 159.

NLR=*maxlrecs*
specifies the number of LRECs that can fit into each prime block of a file, where *maxlrecs* is a nonnegative decimal number. Only assign a value to &SW00NLR when the file uses algorithm #TPFDB0D. With this algorithm, the file contains only prime blocks. All LRECs must be fixed-length. Set NLR to the maximum number of LRECs that can fit into each prime block of the file.

Notes:

1. The NLR parameter is not allowed for B^+Tree data files.
2. Using the NLR parameter will result in an MNOTE being issued because the TPFDF product calculates this value, and any value specified is ignored.

NOC=*maxchains*
specifies the number of overflow blocks to use in implementing add current files, where *maxchains* is a nonnegative decimal number. Only assign a value to NOC when bit 2 of &SW00OP1 is set on. Setting bit 2 of &SW00OP1 instructs the TPFDF product to limit the number of overflow blocks to use when adding LRECs to the subfile. You can set the limit (using NOC) from zero to a maximum of 255 blocks. When this limit is reached in the subfile, the TPFDF product discards the contents of the oldest chain block and copies the LRECs from the first chain block to the prime block. The first (empty) chain block is moved to the last chain block with an initial next available byte (NAB) setting. A new LREC is added to the last (empty) chain block. For example:

```
NOC=5
```

instructs the TPFDF product to use the prime block and 5 overflow blocks.

If NOC is set to zero, the TPFDF product uses only the prime block of the subfile.

Note: The NOC parameter is not allowed for B+Tree data files.

OP1=*op1*
specifies TPFDF processing options, where *op1* is a bit setting that overrides the &SWOOOP1 bit setting. For example,

```
OP1=10000000
```

sets OP1 bit 0 to indicate the file uses backward chaining. See “Assigning Values to Global Set Symbols” on page 71 for more information about &SW00OP1 processing options.

OP2=*op2*
specifies TPFDF processing options, where *op2* is a bit setting that overrides the &SWOOOP2 bit setting. For example,

```
OP2=00010000
```

sets OP2 bit 3 so the file uses new overflow blocks and releases old overflow blocks when a tape load operation completes successfully. See “Assigning Values to Global Set Symbols” on page 71 for more information about &SW00OP2 processing options.

OP3=*op3*
specifies TPFDF processing options, where *op3* is a bit setting that overrides the &SWOOOP3 bit setting. For example,

```
OP2=00000100
```

sets OP3 bit 5 to indicate that the file uses B+Tree indexing. See “Assigning Values to Global Set Symbols” on page 71 for more information about &SW00OP3 processing options.

PIN=*packthresh*
specifies the packing threshold, where *packthresh* is a decimal number from 0–100. For example:

```
PIN=60
```

PTN=*ptnnum*
specifies the number of partitions, where *ptnnum* is a nonnegative decimal number. If *ptnnum* is a nonzero value, it implies that the file is partitioned. The number you assign defines the number of partitions. For example:

```
PTN=4
```

Note: For more information about partitioning, see “Partitions” on page 157.

RBV=*algtype*
specifies the algorithm you want to use with this file, where *algtype* is one of the following:

- #TPFDB01
- #TPFDB02
- #TPFDB03
- #TPFDB04
- #TPFDB05
- #TPFDB06
- #TPFDB07

- #TPFDB08
- #TPFDB09
- #TPFDB10
- #TPFDB0A
- #TPFDB0B
- #TPFDB0C.

For example:

```
RBV=#TPFDB0C
```

You can also create a unique user-defined algorithm and specify it as *algtype* See the &SW00RBV description on page 78 for more information about specifying an algorithm.

RCT=*recordtype*
specifies the record type for fixed or miscellaneous files, where *recordtype* is a defined record type. For example:

```
RCT=#USREC
```

SKE=*ske*
specifies the search key extract for block index support, where *ske* equals the length, in bytes, of the key fields to be removed from the first LREC of each overflow block. The TPFDF product stores this in a TLREC (LREC ID = X'02'). The length you specify must include the primary key (the LREC ID). For example, if the last field in the key is GR21SRLAST, you would specify:

```
SKE='GR21SRLAST-GR21SRKEY+L'GR21SRLAST'
```

Notes:

1. Make SKE large enough to hold any field used as a key field by any application program that might use this file.
2. If the file contains several different formats of LRECs, each identified by a separate LREC ID, set SKE equal to the size of the largest key that needs to be used.
3. Do not use block index support with add current files (these are indicated by bit 2 of &SW00OP1 being set).
4. SKE is not allowed for B+Tree data files.

TQK=*maxtlrec*
specifies the highest technical LREC ID that the TPFDF product can use for this file, where *maxtlrec* is a decimal number from 1–15. For block indexed files, *maxtlrec* must be set to a value greater than 2. For B+Tree data files, *maxtlrec* must be set to a value greater than 4. For B+Tree index files, it must be set to a value less than 3. For example:

```
TQK=2
```

Note: Technical LREC IDs 1–15 are reserved for IBM. Except for B+Tree index files, set TQK=15 to avoid conflicts with the TPFDF product.

TYP=*filetype*
specifies the file type, where *filetype* is R, W, T, or P. B+Tree data files must be R-type files. For example:

```
TYP=R
```

Note: See “File Names” on page 69 for more information about file types.

WID=*fileid*
specifies the file identifier, where *fileid* is a 2-character alphanumeric value (for

example, WID=AB), or a 4-character hexadecimal value (for example, WID=B075). Use a unique file ID for every file.

WRS=*blktype*
specifies the size of the prime block, where *blktype* is one of the following block types:

L0 specifies a 128-byte block size.

L1 specifies a 381-byte block size.

L2 specifies a 1055-byte block size.

L3 specifies a 4000-byte block size. This block type is available only in an ALCS environment.

L4 specifies a 4095-byte block size.

L5 specifies a user-defined size. This block type is available only in an ALCS environment.

L6 specifies a user-defined size. This block type is available only in an ALCS environment.

L7 specifies a user-defined size. This block type is available only in an ALCS environment.

L8 specifies a user-defined size. This block type is available only in an ALCS environment.

For example:

```
WRS=L4
```

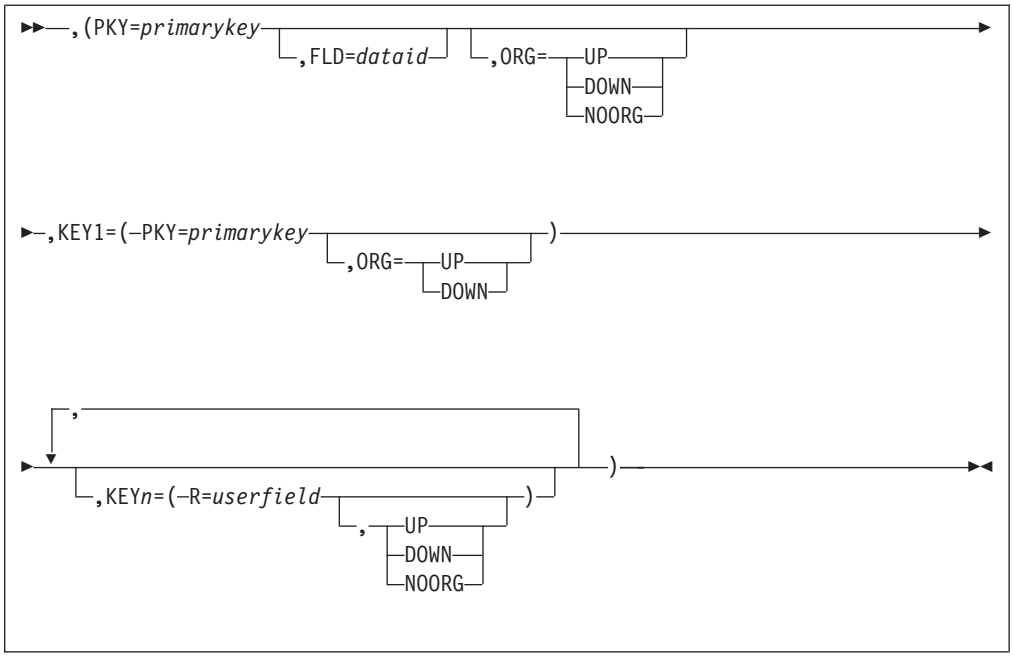
Default Key Parameters

Default keys are used by the TPFDF product to maintain file organization when adding LRECs to a subfile. They can also be used by application programs when reading LRECs from a subfile. Default keys are optional in a DBDEF macro for all TPFDF files except B+Tree data files. Default keys are required for B+Tree data files because they are used by the TPFDF product to organize the B+Tree index.

If a default key is defined for one LREC ID, all LREC IDs in that file must have default keys defined. If a file has the default keys subtable defined and the application tries to add an LREC of an LREC ID that is not defined, an OPR-DB0117 system error is issued and the ECB exits.

Default keys are only appropriate for files that have UP or DOWN organization and should not be defined for:

- P-type files
- T-type files
- Add current files
- Index files (including B+Tree index files)
- Pushdown chaining files.



PKY=*primarykey*

specifies the characteristics of the primary key (LREC ID) of the LREC, where *primarykey* is one of the following:

- An equate that represents the primary key (for example, PKY=#GR00K80)
- An explicit term that represents the primary key (for example, X'80').

Read-only default keys must be in the range X'01' to X'0F' and can be used only for read operations. Read-only default keys can be used to read any combination of keys in an LREC. Therefore, you do not have to read LRECs using the same default keys that you used to create the file. See *TPFDF Programming Concepts and Reference* for more information about using default keys to read LRECs.

Figure 43 on page 97 shows a DBDEF that has two default keys defined. The first key, which has an LREC ID of X'80', is the default key that defines the layout and organization of the file that is created when records are added to the file. This key can also be used for read operations.

The second key has a primary key of X'06', which identifies it as a read-only default key. Primary key X'06' does not affect the layout or organization of the file.

```
* FILE ID X'B073'
         DBDEF FILE=IR73DF,                                          *
               (PKY=#IR73K80,ORG=UP,         READ/ADD DEFAULT KEY    *
               KEY1=(PKY=#IR73K80),                                  *
               KEY2=(R=IR73NAM),                                     *
               KEY3=(R=IR73CTY),                                     *
               KEY4=(R=IR73SAL)),                                    *
               (PKY=#IR73K06,ORG=UP,         READ-ONLY DEFAULT KEY   *
               KEY1=(PKY=#IR73K06),                                  *
               KEY2=(R=IR73NAM),                                     *
               KEY3=(R=IR73CTY))                                     *
```

Figure 43. Read-Only Default Keys in the DBDEF

Note: Do not use LREC IDs X'00'–X'0F' and X'F0'–X'FF'. LREC ID X'00' cannot be used and the other LREC IDs are reserved by the TPFDF product.

FLD=*dataid*
specifies the 2-byte data identifier (secondary key) of an extended LREC, where *dataid* is one of the following:

- An equate that represents the 2-byte data identifier (for example, FLD=#GR00D1000)
- An explicit term that represents the 2-byte data identifier (for example, X'1000').

Notes:

1. The *dataid* value must resolve to a value from X'0000'–X'FFFF'.
2. The DID=*dataid* value in the detail or intermediate-index file must be the same as the FLD=*dataid* value in this index file.

ORG
specifies the default organization for all fields for the specified primary key (PKY) in an LREC. If you specify an organization for KEY1, that organization overrides the ORG value and becomes the default. If you specify an organization for any other KEY*n* value, that organization overrides any previous defaults for that field. If you specify a NOORG value for any key, all keys that follow must be NOORG. For example, you cannot specify the following:

```
KEY1=(PKY=#IR73DF,NOORG),
KEY2=(R=IR73NAM,UP)
```

UP
arranges the fields in ascending order.

DOWN
arranges the fields in descending order.

NOORG
does not arrange the fields in any order.

KEY*n*
specifies the organization of the specified key, where *n* is a number from 2–6.

Note: Keys must be specified sequentially; that is, if KEY1 and KEY4 are specified, you must specify KEY2 and KEY3.

R=*userfield*
specifies the name of a user field, where *userfield* was previously defined in the DSECT macro. For example,

```
KEY2=(R=GR00NAM,UP)
```

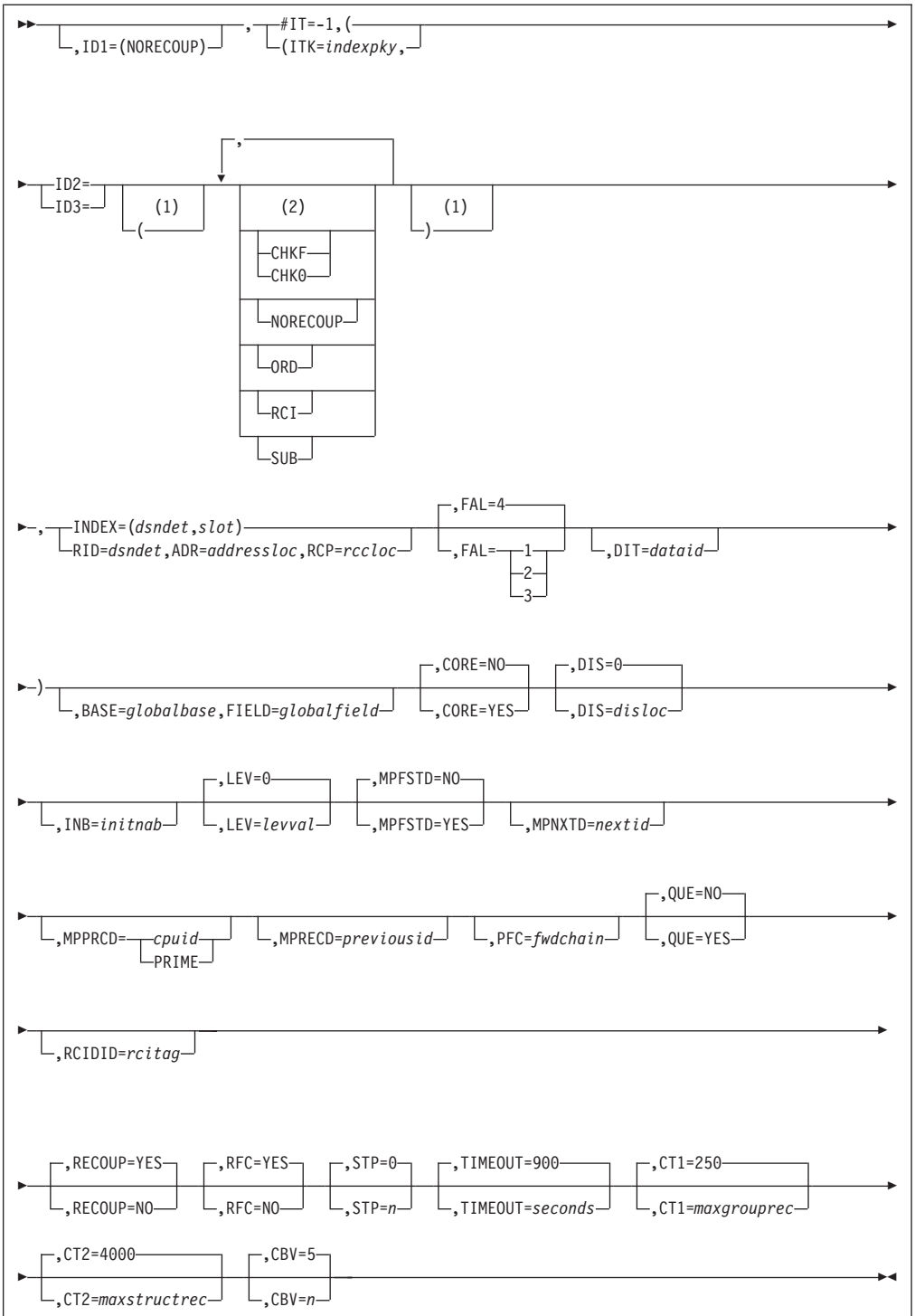
Basic Index Parameters

The following must be coded before index parameters are used in the DBDEF macro:

- The DSECT macro to define the index file, which includes:
  - Definitions of the fields in the index LRECs
  - Global symbols that specify how the index file is contained in physical records, including &SW00RBV to specify the algorithm used to access the top level of the index file.
- The DSECT macro to define the indexed file, which includes:
  - &SW00RBV set to "#TPFDBFF" for an indexed file
  - Bit 3 of &SW00OP3 set to `1` for an indexed fixed file or `0` for an indexed pool file
  - The statements that define the fields in the ALG string associated with each indexing path (see Figure 40 on page 85).

Forward Index Path Parameters

The definitions for the forward path are used by TPFDF recoup, the TPFDF capture/restore utility, information and statistics environment (CRUISE), and some TPFDF macros. Included with the forward index path parameters are descriptions of how the parameters work and some brief examples. For more complete examples of basic indexing, see “Basic Indexing” on page 135.



Notes:

1 If one parenthesis is specified, both parentheses must be specified. If you do not specify any subparameters with the ID2 or ID3 parameter, parentheses () are not required. For example:

```
ID2=,...
```

2 If CHKF or CHK0 is not specified, the record code check (RCC) of the indexed file is in the RCP field of the index LREC.

ID1=(NORECOUP)
inhibits chain chasing during recoup, but allows TPFDF CRUISE to use the recoup subtable (assuming RECOUP=YES was specified to generate the subtable); that is, when ID1=(NORECOUP) is specified in a file DBDEF statement, the ZRECP command does not recoup the file.

#IT=−1
specifies that all primary keys (for example, LREC IDs X'80', X'90', and X'A0') forward index the same file with the same TPFDF recoup and TPFDF CRUISE processing attributes (specified by the ID2 parameter).

ITK=*indexpky*
specifies the primary key (LREC ID) of this index file, where *indexpky* is one of the following:

- An equate that represents the primary key (for example, ITK=#GR00K80)
- An explicit term that represents the primary key (for example, X'80').

You need only one ITK parameter for each primary key. All records that follow with the same primary key (for example, all LREC IDs of X'80') forward index the same file with the same TPFDF recoup and TPFDF CRUISE processing attributes (specified by the ID2 parameter).

Notes:

1. The attributes specified with the ITK parameter remain in effect for a specified primary key until a new ITK parameter with the same primary key is specified.
2. The ITK parameter can forward index more than one file by entering address slot arguments with the INDEX parameter.

ID2
specifies options that affect TPFDF recoup and TPFDF CRUISE processing. ID2 is used to describe references held in items. The position of the references is defined only once because the same description applies to each item. ID2 references are defined using CBV=1, CBV=2, CBV=4, or CBV=5. If a block contains a mixture of ID2 and ID3 references, TPFDF recoup processes:
1. All the ID2= references (specified in items)
2. All the ID3= references (fixed position).

ID3
specifies options that affect TPFDF recoup and TPFDF CRUISE processing. ID3 is used to describe references held in fixed locations. Each reference is described separately because the reference is not held as a part of an item. If the block contains only ID3 references, CBV=3 is used to identify the structure. If a block contains a mixture of ID2 and ID3 references, they are defined using CBV=1, CBV=2, CBV=4, or CBV=5. In this case, TPFDF recoup processes:
1. All the ID2= references (specified in items)
2. All the ID3= references (fixed position).

CHKF
specifies that the record code check (RCC) of the indexed file is the same as the record code check of this index file.

Note: The RCP parameter must be set to −1, or omitted.

CHK0
specifies that TPFDF recoup should use a record code check (RCC) of X'00'.

Note: The RCP parameter must be set to −1, or omitted.

NORECOUP
allows the definition of the reference to be used by TPFDF CRUISE, but the reference is not chain chased by TPFDF recoup.

ORD
specifies that the reference is an ordinal number and not a file address.

RCI
specifies that this file contains a reference to a detail file and will use recoup chain-chasing indicator (RCI) processing to avoid unnecessary chain chasing.

RCI processing for a file is specified as follows:

- For each top-level index file in the structure, specify `RCIDID=rcitag`.
- For each file that contains a reference to a detail file, specify, ID2=(RCI),...).

SUB
indicates that there are subitems in items.

Notes:

1. Subitems can be defined only in customer-format files. See “Parameters for TPFDF Recoup and TPFDF CRUISE Processing for Customer-Format Files” on page 112.
2. SUB is not allowed with the ID3 parameter.
3. SUB is not allowed if CBV=5 is specified or used as a default.
4. Subitems must have a fixed size and start at a fixed location in the item.
5. Each subitem contains a reference.
6. Specify the number of subitems using the CNT parameter or specify the location of a count field using the CPT parameter.
7. Specify the size of the subitem count field using the SSZ parameter.

INDEX=(*dsndet*,*slot*)
identifies the file you are forward indexing, where *dsndet* is a 6-character file (and DSECT) name of the detail or intermediate-index file to which this index file is chained, and *slot* is the number of an index slot that contains a reference to the detail or intermediate-index file to which this index file is chained. This number must be the same as the index slot coded with the IFR parameter in the detail or intermediate-index file. See the IFR parameter in “Backward Index Path Parameters” on page 107.

Use INDEX if the RID, ADR, and RCP definitions are in the standard positions in the index LRECs. For example, specifying:

```
INDEX=(SAM5SR,0)
```

has the same effect as specifying:

```
RID=SAM5SR,ADR=SAM4FAD-SAM4REC,RCP=SAM4RCC-SAM4REC
```

RID=*dsndet*
identifies the file you are forward indexing, where *dsndet* is one of the following:

- A 2-character decimal record ID of the detail or intermediate-index file (for example, RID=10).
- A 4-character hexadecimal record ID of the detail or intermediate-index file (for example, RID=FDFD).
- A 5 or 6-character DSECT name of the detail or intermediate-index file (for example, RID=SAM5SR).
- The relative location of a field in the index file that contains the record ID of the detail or intermediate-index file (for example, RID=SAM4LAB-SAM4REC).

ADR=*addressloc*
specifies the relative position of the address field, where *addressloc* is expressed as the location of the field that contains the file address (FAD) minus the location of the start of the first record (REC). For example,

```
ADR=SAM4FAD-SAM4REC
```

RCP=*rccloc*
specifies the relative position of the record code check, where *rccloc* is expressed as the location of the field that contains the record code check (RCC) minus the location of the start of the first record (REC). For example,

```
RCP=SAM4RCC-SAM4REC
```

Note: When ID2=CHK0, ID2=CHKF, ID3=CHK0, or ID3=CHKF is specified, the RCP parameter must be set to −1, or omitted.

FAL
specifies the number of bytes used to hold the ordinal number when ID2=(ORD) or ID3=(ORD) is specified.

DIT=*dataid*
for extended LRECs, specifies the 2-byte data identifier (secondary key) of this index file, where *dataid* is one of the following:

- An equate that represents the 2-byte data identifier (for example, DIT=#GR00D1000)
- An explicit term that represents the 2-byte data identifier (for example, X'1000').

Notes:

1. The *dataid* value must resolve to a value from X'0000'–X'FFFF'.
2. The DID=*dataid* value in the detail or intermediate-index file must be the same as the DIT=*dataid* value in this index file. See the DID parameter in “Backward Index Path Parameters” on page 107.

BASE=*globalbase*
specifies the global base for recoup chain chasing, where *globalbase* is the name of a global area. BASE, FIELD, and STP=1 are required together to specify that a subfile is referenced from a global field, for example:

```
STP=1,         * Special monitor
BASE=GLOBZ,    * Global area
FIELD=@KRPCA0  * Global field
```

Note: BASE and BASECOD are not allowed together.

FIELD=*globalfield*
specifies the global field for recoup chain chasing, where *globalfield* is the name of a global field. BASE, FIELD, and STP=1 are required together to specify that a subfile is referenced from a global field, for example:

```
STP=1,         * Special monitor
BASE=GLOBZ,    * Global area
FIELD=@KRPCA0  * Global field
```

CORE
specifies whether the file is located in main storage or on DASD.

YES specifies that TPFDF recoup and TPFDF CRUISE will retrieve a core block and copy the contents of the main storage file into the core block and pass it to the monitor.

The reference defined by the BASE parameter and the FIELD parameter (or defined by the BASECOD parameter) is a reference to a block in main storage, and not a file address.

NO specifies that the file is located on DASD and uses normal TPFDF recoup and TPFDF CRUISE processing.

DIS=*disloc*
specifies the location of a displacement field, which contains a displacement to be added to the ADR value in any ID3 reference, where *disloc* is the label of an area, a register, or an immediate value. If this is a CNT-type file, the displacement is also added to the value specified by the PIT parameter.

INB=*initnab*
specifies the initial next available byte (NAB), where *initnab* is a numeric value.

Notes:

1. INB can only be specified for P-type files.
2. Although it cannot be specified for standard-format files (R-type, W-type, and T-type), the initial NAB value for standard-format files is X'1A' (the standard-format header is 26 bytes).

LEV=*levval*
specifies the ECB level control, where *levval* is a number in the range 0–7, which specifies the number of levels over which CRUISE will distribute available ECBs.

CRUISE does not recalculate the ECB distribution when the actual level count is less than the last recalculated level value. This may lead to less-effective ECB usage during chain chasing for some data structures. The use of LEV can help to prevent this.

When the LEV parameter value equals 0 (by default or if specified), the TPFDF product uses the following formula to distribute available ECBs over the levels (where *actuallevels* equals the actual number of levels at run time and *maxecbs* is the maximum number of ECBs you allow the TPFDF product to use):

```
for n = 0 to (actuallevels - 1)

   Level (actuallevels - n) = .8 * maxecbs * .5^n
```

For example, if LEV=0 was specified and *actuallevels* equals 4 and *maxecbs* equals 200, the TPFDF product distributes ECBs as follows:

```
Level 4  =  160    ***  Data level               ****
Level 3  =   80    ***  Intermediate-index level ****
Level 2  =   40    ***  Intermediate-index level ****
Level 1  =   20    ***  Top-index level          ****
```

This distribution assumes that each level gets progressively larger. If your data structure progresses differently, use the LEV parameter to change the distribution.

When *levval* is less than the actual levels in the structure (for example, if *levval* equals 3 and the actual levels in the structure at run time equal 5 and *maxecbs* equals 200), the TPFDF product uses the following formula to distribute available ECBs over the levels.

```
for n = 0 to (levval - 1)

   Level (levval- n) = .8 * maxecbs * .5^n
```

```
   Level 3  =  160    ***  Intermediate-index level ****
   Level 2  =   80    ***  Intermediate-index level ****
   Level 1  =   40    ***  Top-index level          ****
```

```
for n = 1 to (actuallevels - levval)

   Level (levval+ n) = 1.1^n * Level levval ECBs
```

```
   Level 4  =  1.1 * 160
   Level 5  =  1.1 * 1.1 * 160

   Level 4  =  176    ***  Intermediate-index level ****
   Level 5  =  194    ***  Data level               ****
```

MPFSTD
specifies the priority for chain chasing a record.

YES
specifies that this record and any other records with MPFSTD=YES specified must be chain chased before other records are chased.

NO
specifies that this record will be chain chased after all records with MPFSTD=YES specified.

MPNXTD=*nextid*
specifies the ID of the record for which chain chasing will start when chain chasing is completed for this record, where *nextid* is a 2-character alphanumeric value (for example, MPNXTD=AB), or a 4-character hexadecimal value (for example, MPNXTD=B075).

Note: If you specify the MPNXTD parameter, you must specify the MPPRCD parameter.

MPPRCD
specifies the ID of the processor that will chain chase this record. If the specified processor is not active, the primary processor chain chases the record at the end of recoup phase 1 processing.

cpuid
is a 1-character processor identifier.

PRIME
specifies that this record will be chain chased on the primary recoup processor.

MPRECD=*previousid*
specifies the ID of the record for which chain chasing must be completed before chain chasing can begin for this record, where *previousid* is a 2-character alphanumeric value (for example, MPRECD=AB), or a 4-character hexadecimal value (for example, MPRECD=B075).

PFC=*fwdchain*
specifies the position of forward chain, where *fwdchain* equals the address of the forward chain minus the address of the beginning of the file. For example:

```
PFC=dsniFCH-dsniBID
```

Notes:

1. PFC or CPF is necessary if FCH is present.
2. PFC is not allowed with CPF.
3. PFC=-1 is used for a no forward chain file.
4. The value can also be specified as an absolute displacement.

QUE
specifies whether the data structure is processed in recoup QUE-inhibited mode.

NO specifies that the file is not recoup QUE-inhibited; that is, the file address can pass references between subfiles during recoup.

If a file is accessed by passing its file address from one subfile to another, the file can be missed during recoup processing. Figure 44 shows how access to the "PR" file is controlled by passing its file address from one subfile to another (and setting the unused reference to X'0000'). If QUE=NO is specified, recoup is not QUE-inhibited and the "PR" file is not chain chased as explained in the following:

1. The subfile addressed by "AAC" is chain chased.
2. The subfile ("ZRH") has the file address of "PR".
3. The file address of "PR" is passed from "ZRH" to "AAC".
4. The subfile addressed by "ZRH" is then chain chased.

Note: The "PR" file is not chain chased because the file address "ZRH" no longer has the reference to "PR".

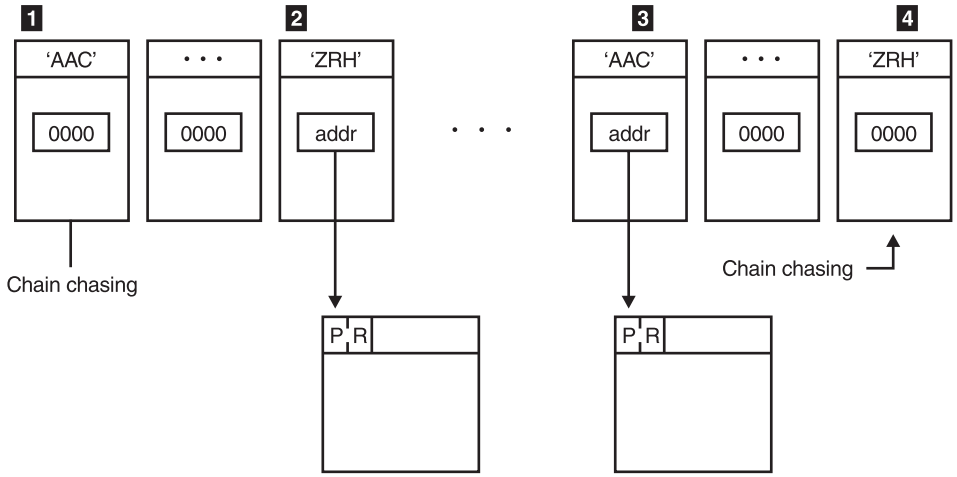


Figure 44. QUE=NO Parameter

YES specifies that the file is recoup QUE-inhibited; that is, the file address cannot pass references between subfiles during recoup. Recoup QUE-inhibited mode is specific to the airline `QUEUING` function and assumes the application environment is using the @BRCPQ global as the queue control indicator. While TPFDF recoup is performing the queuing phase, BIT0 of the @BRCPQ global is set on. Applications should not pass references while this bit is set. The bit is turned off at the end of the queuing function.

Note: QUE=YES is not valid for ALCS.

RCIDID=*rcitag*
specifies that this top-level index file will use recoup chain-chasing indicator (RCI) processing to avoid unnecessary chain chasing, where *rcitag* is a 4-character alphanumeric value (for example, RCIDID=AB12). The *rcitag* can be specified to identify top-level index files that are in the same structure.

RCI processing for a file is specified as follows:

- For each top-level index file in the structure, specify `RCIDID=`*`rcitag`*.
- For each file that contains a reference to a detail file, specify, `(ID2=(RCI),RID=....)`.

RECOUP
specifies whether the table used by the recoup utility and TPFDF CRUISE is built for this data structure

YES specifies that a recoup table is built for the data structure; therefore, the ZRECP commands and TPFDF CRUISE process this file.

NO specifies that a recoup table is not built for the data structure; therefore the ZRECP commands and TPFDF CRUISE do not process this file. RECOUP=NO can be specified for ALCS and non-TPFDF files that do not contain embedded references or forward chaining.

RFC
specifies whether TPFDF recoup verifies that the record code check (RCC) in an overflow block is the same as the RCC in the prime block.

YES specifies that TPFDF recoup checks the RCC of the prime block and overflow blocks.

NO specifies that TPFDF recoup does not check the RCC of the prime block and that overflow blocks are retrieved with an RCC of zero.

STP=*n*
specifies the type of chain chasing monitor, where *n* is one of the following:

0 causes TPFDF recoup and TPFDF CRUISE to chain chase all the ordinals defined in the FACE file type for this file, starting with the lowest ordinal number (BOR) and ending with the highest ordinal number (EO#).

1 specifies that references to this subfile are in a global location or core field.

2–50 Reserved for IBM use.

51–255
Reserved for customer-specific chain chasing monitors.

Note: BASE and BASECOD require STP=1.

TIMEOUT=*seconds*
specifies the time-out value for chain-chasing a file structure, where *seconds* is a number of seconds.

Timeouts can be related to timing problems caused by a faulty structure that causes loops in forward chain processing or by too many embedded references at the lower levels of a file structure. If there are too many embedded references, not enough ECBs are allocated to the lower levels and a timeout can occur. The LEV parameter can be used to adjust the number or ECBs that are allocated for each level during CRUISE processing.

CT1=*maxgrouprec*
specifies the maximum number of records read for an ALCS group, where *maxgrouprec* is a nonnegative number.

Note: The CT1 and CT2 parameters only affect ALCS recoup processing if the TPFDF sample ALCS recoup user exit code (ARD0, ARD1, and ARD2)

has been installed. For more information about installing the TPFDF sample ALCS recoup user exit code, see *TPFDF Installation and Customization*.

CT2=*maxstructrec*
specifies the maximum number of records read for an ALCS structure that contains all groups chained from any one record in an ALCS prime group, where *maxstructrec* is a nonnegative number.

Note: The CT1 and CT2 parameters only affect ALCS recoup processing if the TPFDF sample ALCS recoup user exit code (ARD0, ARD1, and ARD2) has been installed. For more information about installing the TPFDF sample ALCS recoup user exit code, see *TPFDF Installation and Customization*.

CBV=*n*
specifies the type of chain chasing monitor, where *n* is one of the following:

1 Fixed-length type items in NAB or ADD/DEL files.

2 CNT type file.

3 Fixed-position-only references.

4 Variable-length type items in NAB or ADD/DEL files.

5 Specifies that the file is a standard-format file with LRECs with embedded pointers and a logical record ID.

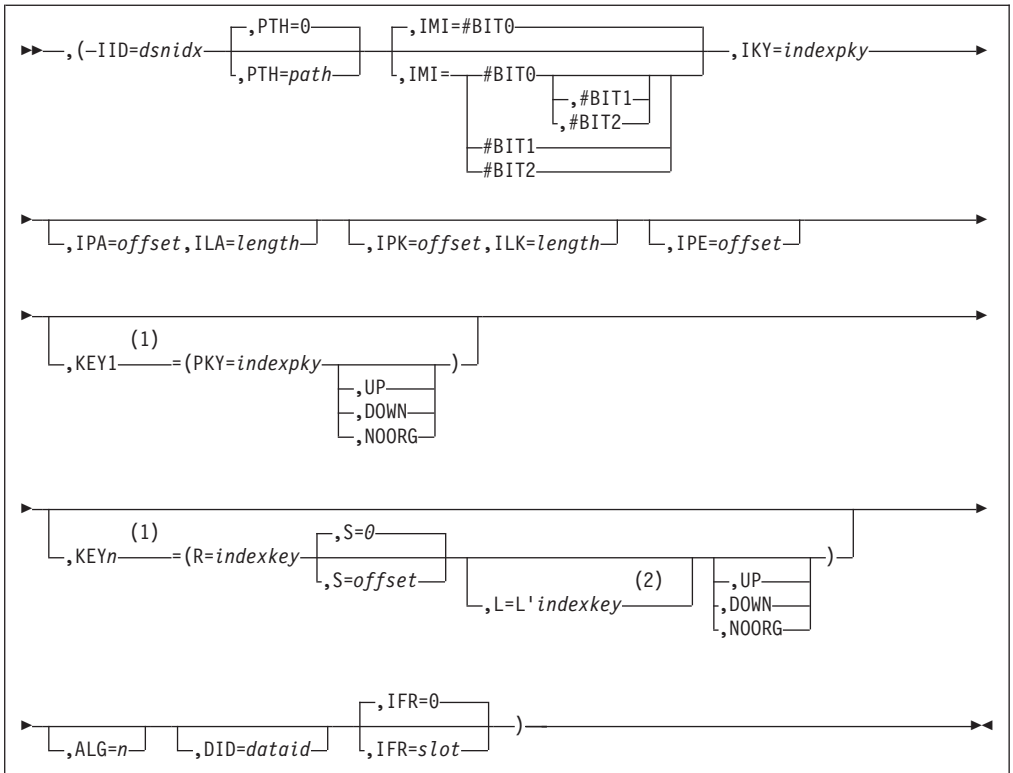
6–50 Reserved for IBM use.

>50 Reserved for customer-specific chain chasing monitors.

Note: If CBV does not equal 5 (for example, CBV=4), see “Parameters for TPFDF Recoup and TPFDF CRUISE Processing for Customer-Format Files” on page 112 to include parameters that specify the location of fields.

Backward Index Path Parameters

The definitions for the backward path are used to access the detail file. Included with the backward index path parameters are descriptions of how the parameters work and some brief examples. For more complete examples of basic indexing, see “Basic Indexing” on page 135.



Notes:

1 KEY parameters are only required when a search other than the default is required. If KEY parameters are not supplied, the default key fields are as follows:

```
KEY1=(PKY=#dsniK80,UP),
KEY2=(R=dsniA80,S=0,L=L'dsniA80,UP)
```

2 If you do not specify the L parameter, L defaults to the length of the R parameter.

IID=*dsnidx*

identifies the backward path of the file whose index LRECs point to this file, where *dsnidx* is a 6-character file (and DSECT) name of the index file to which this detail or intermediate-index file is chained.

Note: If the index LRECs are extended, the DID parameter is used to specify the secondary key.

PTH

specifies a path to an index file, where *path* is a nonnegative decimal number.

Note: All of the parameters that follow the PTH parameter define the characteristics of the specified path.

IMI

specifies one of the following:

#BIT0

specifies that a path is read only. Read-only paths cannot be used to add indexes. #BIT0 is a system equate that can be specified in binary form. For example:

```
IMI=10000000
```

#BIT1
specifies FULLFILE processing from the ordinal defined by BOR to the ordinal defined by EOR. #BIT1 is a system equate that can be specified in binary form. For example:

```
IMI=01000000
```

Note: If #BIT1 is specified, the addressing arguments (IPA and ILA) are ignored.

#BIT2
specifies FULLFILE processing from the ordinal located using the addressing argument defined by IPA and ILA parameters to the ordinal defined by EOR. #BIT2 is a system equate that can be specified in binary form. For example:

```
IMI=00100000
```

IKY=*indexpky*
specifies the primary key (LREC ID) of the index file to which this detail or intermediate-index file is chained, where *indexpky* is one of the following:

- An equate that represents the primary key (for example, IKY=#GR00K80)
- An explicit term that represents the primary key (for example, X'80').

Notes:

1. Set the IKY parameter value equal to the ITK parameter value of the index file to which this detail or intermediate-index file is chained. Do not use LREC IDs X'00'–X'0F' and X'F0'–X'FF'. LREC ID X'00' cannot be used and the other LREC IDs are reserved by the TPFDF product.

IPA=*offset***,ILA=L'***length*
specifies the offset into and length to use of a string passed with the ALG parameter on a DBOPN or other TPFDF macro, where *offset* and *length* are nonnegative decimal numbers. This string is used to extract, according to the IPA and ILA parameter, an addressing argument that is used to locate a prime block of an index file.

The following example shows how the IPA and ILA parameter settings defined in a detail file, IR24DF, are used to locate an argument string in an index file, IR25DF.

```
DBDEF FILE=IR24DF
(IID=IR25DF,
IPA=0,ILA=1,

DBOPN REF=IR24DF,ALG=C'SMITH'
```

In this example, when the DBOPN macro is processed, the TPFDF product accesses the **S** prime block in the first-level index file, IR25DF. The S prime block is determined by extracting, for a length of 1 bytes (ILA=1), the value found at offset 0 (IPA=0) into SMITH (ALG='SMITH').

IPK=*offset***,ILK=***length*
specifies the offset into and length to use of a string passed with the ALG parameter on a DBRED or other TPFDF macro, where *offset* and *length* are nonnegative decimal numbers. This string is used to extract, according to the IPK and ILK parameters, an index key that is used to locate an index record in an index file.

In the following example, when the DBRED macro is processed, the TPFDF product reads the **SM** prime block in the first-level index file, IR20DF.

```
DBDEF FILE=IR24DF
(IID=IR20DF,
IPA=0,ILA=2
IPK=0,ILK=5
KEY1=(PKY=#IR20K80,UP),   search keys for index
KEY2=(R=IR20NAM,DOWN))

DBRED REF=IR24DF,ALG=C'SMITH'
```

The SM prime block of the first-level index file (IR20DF) is determined by extracting, for a length of 2 bytes (ILA=2), the value found at offset 0 (IPA=0) into SMITH (ALG='SMITH'). The TPFDF product then reads the first index record in the IR20NAM field ((KEY2=(R=IR20NAM,DOWN)) that contains the name SMITH. SMITH is determined by extracting, for a length of 5 bytes (ILK=5) the value found at offset 0 (IPK=0) into **SMITH**. The TPFDF product then uses the SMITH index record to locate the SMITH detail subfile.

IPE=*offset*

specifies an offset into a string passed with the ALG parameter on a DBRED or other TPFDF macro, where *offset* is a nonnegative decimal number. IPE is used with the IPA and ILA parameter, to extract the addressing argument for the end ordinal used in FULLFILE processing. This end ordinal addressing argument starts at the first byte after the starting ordinal addressing argument for a length specified by the ILA parameter.

The following shows how the IPA, ILA, and IPE parameter settings are used by the DBRED macro to read all of the detail subfiles indexed by ordinal 0 ("A") to ordinal 4 ("E"). The #DO loop is executed until all of the detail subfiles have been read.

```
IPA=0,ILA=1
IPE=1

DBOPN REF=GR00SR,REG=R4
#DO INF
   DBRED FULLFILE REF=GR00SR,REG=R4,ALG=C'AE'
#DOEX TM,SW00RTN,#BITA,NZ
#ELOP
#EDO
```

Beginning ordinal 0 ("A") is determined by extracting, for a length of 1 byte (ILA=1), the value found at offset 0 (IPA=0) into AE (ALG=C'AE'). Ending ordinal 4 ("E") is determined by extracting, for a length of 1 byte (ILA=1), the value found at offset 1 (IPE=1) into AE (ALG=C'AE').

Note: The IPE parameter should not be used when the hashing algorithms (#TPFDB09, #TPFDB0F, and #TPFDB10) are used to address the top-level index file.

KEY

specifies the position of the index keys in a string passed with the ALG parameter on a DBRED or other TPFDF macro.

Note: Default keys are not valid for T-type files.

PKY

specifies the primary key of the index LREC.

Note: Do not use LREC IDs X'00'–X'0F' and X'F0'–X'FF'. LREC ID X'00' cannot be used and the other LREC IDs are reserved by the TPFDF product.

R=*indexkey*
specifies the position in the index LREC of the index key, where *indexkey* is field that contains the index key.

S=*offset*
specifies the offset in the algorithm string of the data to compare with the index key located by the R parameter, where *offset* is a nonnegative decimal number.

L=L'*indexkey*
specifies the length of the index key, where *indexkey* is field that contains the index key.

ALG=*n*
For indexed detail files, but not top-level index files, *n* specifies the size of the algorithm string for each index path, where *n* is a decimal value. If the algorithm string size is not defined in the DSECT (using the `&BEG` and `&END` statements), the ALG parameter must be used to specify the length of the algorithm string.

The following example defines an algorithm string as 4 bytes long:

```
(IID=IR23DF,PTH=1,IPA=0,ILA=0,IPK=0,ILK=4,ALG=4)
```

Notes:

1. The ALG parameter in the DBDEF macro does not have the same meaning as the ALG parameter in other TPFDF macros and functions.
2. The ALG parameter must be used in the DBDEF for any file that requires the area to be specified.
3. The ALG parameter cannot be used with a read-only path (`IMI=#BIT0`).

DID=*dataid*
for extended LRECs, specifies the 2-byte data identifier (secondary key) of the index file that refers to this detail or intermediate-index file, where *dataid* is one of the following:

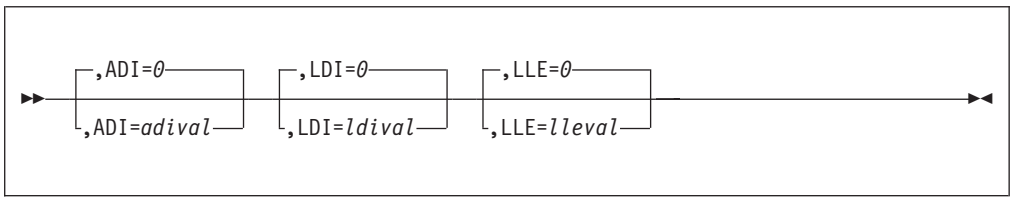
- An equate that represents the 2-byte data identifier (for example, DID=#GR00D1000)
- An explicit term that represents the 2-byte data identifier (for example, X'1000').

Notes:

1. The *dataid* value must resolve to a value from X'0000'–X'FFFF'.
2. The DIT=*dataid* value in the index file must be the same as the DID=*dataid* value in this detail or intermediate-index file. See the DIT parameter in “Forward Index Path Parameters” on page 98.

IFR=*slot*
specifies the index slot, where *slot* is the number of an index slot that contains a reference to the index file to which this detail or intermediate-index file is chained. This number must be the same as the index slot coded with the INDEX parameter in the index file. See the INDEX parameter in “Forward Index Path Parameters” on page 98.

Data Extraction Parameters



ADI=*adival*
specifies a displacement value, where *adival* is a decimal value.

Note: The ADI displacement value is used by the AREA parameter in the DBRED macro to identify where to put an extracted data string.

LDI=*ldival*
The displacement (in an index LREC) of the start of the data to extract, where *ldival* is a decimal value.

LLE=*adival*
The number of bytes to extract, where *lleval* is a decimal value.

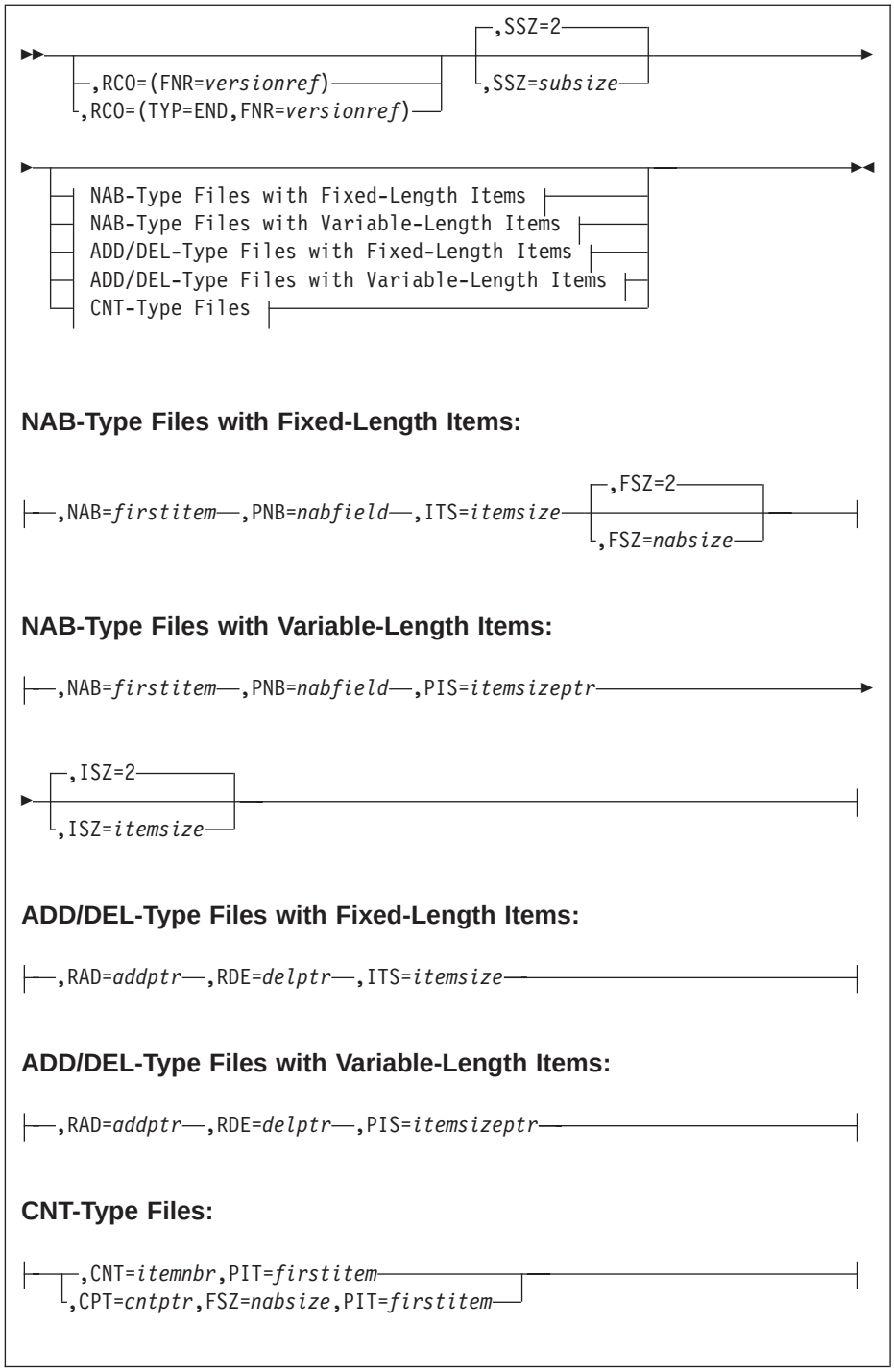
Parameters for TPFDF Recoup and TPFDF CRUISE Processing for Customer-Format Files

Besides support for TPFDF files used in recoup and CRUISE, the DBDEF parameters support the following type of customer-format files:

- Next available byte (NAB) type files with:
  - Fixed-length items
  - Variable-length items.
- Add and delete (ADD/DEL) type files with:
  - Fixed-length items
  - Variable-length items.

  These files are also known as *add index* and *delete index* files.
- Count (CNT) type files
- Files that contain only fixed-position references
- Files that use forward chains.

Note: For more information and examples of customer-format files, see “Using Customer-Format Files” on page 181.



The following parameters are used with the CBV parameter to describe the basic characteristics of a customer-format file when CBV=1, CBV=2, CBV=3, or CBV=4 is specified. For more information about the CBV parameter, see “Forward Index Path Parameters” on page 98.

RCO

specifies the recoup concatenation value.

(FNR=*versionref*)

specifies the next file version to chain chase, where *versionref* is a number from 0–255.

(TYP=END,FNR=*versionref*)
specifies the last file version for this file and the file version where the chain chasing starts again, where *versionref* is a number from 0–255.

Complicated customer-format data structures such as a mixture of CNT (CBV=2) and fixed-position references (CBV=3) require different file versions for each structure. Each file version has a different recoup table associated with it. The recoup concatenation order (RCO) parameter avoids separate chain chasing for each of the tables, as shown in the following example:

```
DBDEF FILE=GR21SR,FVN=0,     FILE VERSION 0                 X
      CBV=3,.....,           THE CBV=3 LAYOUT               X
      ......,                                               X
      RCO=(FNR=1)            CHAIN CHASE FILE VERSION 1 NEXT
DBDEF FILE=GR21SR,FVN=1,     FILE VERSION 1                 X
      CBV=2,.....,           THE CBV=2 LAYOUT, NAB TYPE     X
      ......,                                               X
      RCO=(FNR=2)            CHAIN CHASE FILE VERSION 2 NEXT
DBDEF FILE=GR21SR,FVN=2,     FILE VERSION 2                 X
      CBV=2,.....,           THE CBV=2 LAYOUT, CNT TYPE     X
      ......,                                               X
      RCO=(TYP=END,FNR=0)    END OF LIST, GO BACK TO FVN 0
```

RAD=*addptr*
specifies the location of the ADD field in an ADD/DEL type record, where *addptr* equals the address of the ADD field minus the address of the beginning of the file. For example:

```
RAD=GP01FLD-GP01BID
```

RDE=*delptr*
specifies the location of the DEL field in an ADD/DEL type record, where *delptr* equals the address of the DEL field minus the address of the beginning of the file. For example:

```
RAD=GP01FLD-GP01BID
```

Note: The value can also be specified as an absolute displacement.

ITS=*itemsize*
specifies the item size for fixed length items in CBV=1 or CBV=2 type files. where *itemsize* equals the length of an item. For example:

```
ITS=L'GP01ITM
```

Notes:

1. All items must be the same size.
2. The value can also be specified as an absolute displacement.

NAB=*firstitem*
specifies the position of the first item in the block in a CBV=1 or CBV=4 type file, where *firstitem* equals the address of the first item minus the address of the beginning of the file. For example:

```
NAB=GP01ITM-GP01BID
```

PNB=*nabfield*
specifies the position of the NAB field in a CBV=1 or CBV=2 type file, where *nabfield* equals the address of the NAB field minus the address of the beginning of the file. For example:

```
PNB=GP01NAB-GP01BID
```

Note: The value can also be specified as an absolute displacement.

FSZ=*nabsize*
for CBV=1, CBV=2, or CBV=3 type files, FSZ specifies the length in bytes of the CNT or NAB field, where *nabsize* equals 1, 2, 3, or 4. For example:

```
FSZ=1
```

PIS=*itemsizeptr*
specifies the location of the field in each item that contains the item size for CBV=4 type files, where *itemsizeptr* equals the address of the SIZ field minus the address of the beginning of the item. For example:

```
PIS=GP01SIZ-GP01ITM
```

Note: The value can also be specified as an absolute displacement.

ISZ=*itemsize*
for CBV=4 type files, ISZ specifies the length (in bytes) of the item size field, where *itemsize* equals 1, 2, 3, or 4. For example:

```
ISZ=3
```

CNT=*itemnbr*
specifies the number of items in the record for a CBV=2 type file, where *itemnbr* is a decimal number. For example:

```
CNT=3
```

PIT=*firstitem*
for CBV=2 type files, PIT specifies the position of the first item, where *firstitem* equals the address of the first item minus the address of the beginning of the file. For example:

```
PIT=GP01ITM-GP01BID
```

Note: The value can also be specified as an absolute displacement.

CPT=*cntptr*
specifies the location of the CNT field for CBV=2 type files, where *cntptr* equals the address of the CNT field minus the address of the beginning of the file. For example:

```
CPT=GP01CNT-GP01BID
```

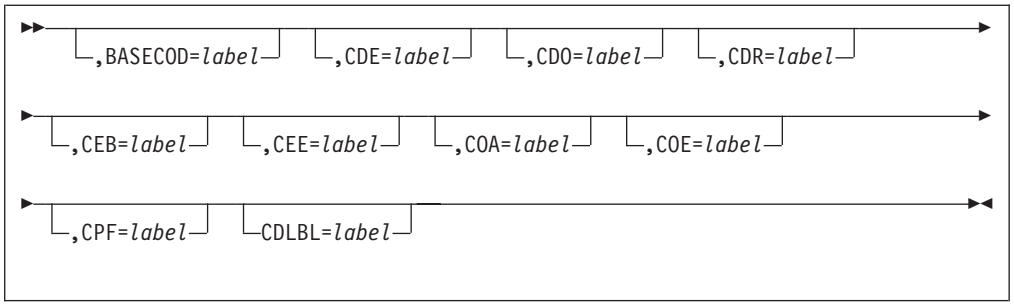
SSZ=*subsize*
for CBV=1, CBV=2, CBV=3, or CBV=4 type files, SSZ specifies the length, in bytes, of the subitem count, where *subsize* equals 1, 2, 3, or 4. For example:

```
SSZ=1
```

TPFDF Recoup User Exits

The TPFDF product has recoup and CRUISE user exits that allow you to run user-defined code:

- If a forward index pointer causes an error (CDE)
- Before a forward index pointer is retrieved (CDO)
- To override the CNT, PNB, NAB, or PIT value at run time (CDR)
- Before a block is referenced (CEB)
- After a block is referenced (CEE)
- After a block is retrieved (COA)
- If there is an error when a block is retrieved (COE)
- To determine the location of the forward chain field (CPF).



label
specifies a user-supplied name that identifies user-defined code that starts at the CDLBL parameter. A user exit *label* must match the *label* specified with the CDLBL parameter.

BASECOD
specifies the label of installation-wide exit code to determine the location of the file. The installation-wide exit code:

- Is coded in a corresponding DBDEF CDLBL=*label* statement.
- Stores the location of the referenced file in register 15 (R15) before returning.
- Uses the `BR R7` statement to return to TPFDF recoup.

Notes:

1. BASE and BASECOD are not allowed together.
2. BASECOD requires STP=1.
3. Ensure that register R15 contains the file address or core address of the user code.
4. If the reference itself is a main storage address rather than a file address, specify CORE=YES.

CDE
specifies code to be used when the retrieval of the forward index pointer caused an error.

Notes about the user code:

1. The CDE parameter is not valid for ALCS.
2. The address of the block that contains the forward index pointer is contained in level D0 (field name EBCFA0) of the ECB.
3. The address of the block that caused the retrieval error (forward index pointer) is in level D1 (field name EBCFA1) of the ECB.
4. EBCFA1 might be changed for a second attempt of retrieval.
5. The user code must include one of the following statements to return to TPFDF recoup:
   - To inhibit TPFDF recoup from continuing, use:

     ```
     B 0(,R6)
     ```

     (The current ECB exits, forcing a time-out in the main monitor.)
   - To allow the TPFDF product to retry the retrieval of this reference for a second time, use:

     ```
     B 4(,R6)
     ```

CDO
specifies code to be used before the forward index pointer is retrieved.

Notes about the user code:

1. The address of the block that contains the forward index pointer is in level D0 (field name EBCFA0) of the ECB.
2. Register 5 (R5) is pointing to the LREC where the forward index pointer is located.
3. The user code must include one of the following statements to return to TPFDF recoup:
   - To inhibit TPFDF recoup from chain chasing all references, use:

     ```
     B 0(,R6)
     ```
   - To inhibit TPFDF recoup from chain chasing this reference, use:

     ```
     B 4(,R6)
     ```
   - To allow TPFDF recoup to continue with this reference, use:

     ```
     B 8(,R6)
     ```

CDR

User code to override the following values at run time:

- CNT
- PNB
- NAB
- PIT.

You can use CDR to specify an exit to user code that overrides various values at run time.

Notes about the user code: Table 35 shows how the installation-wide exit code should write values into fields in the ECB (EBW050 and EBW054), or into register 5 (R5) to override the values used by TPFDF recoup.

Table 35. Using CDR to Override the CNT, PNB, NAB, PIT Values at Run Time

CBV=1 or CBV=4	• The maximum NAB value is used for ADD/DEL-type files and NAB-type files. • The location of the NAB is determined using *dsnd*NAB-*dsnd*BID • EBW050 is used to override the maximum NAB value. • The value in R5 replaces the *initial NAB value* (INB) Use register 5 (R5) to change the INB. INB=X'1A' is for TPFDF files (The TPFDF header is 26 bytes).
CBV=2	• The user code stores the position of the first item in register R5. • TPFDF recoup takes the value in R5 and adds it to the value specified by PIT= (for example, see Figure 45 on page 118). • The user code can also put the number of items in the work area EBW054. • Because you can modify EBW054, the value is not added to the value in CNT field.
CBV=3	The user code places the number of references in R5.

Notes:

1. The CDR parameter is not valid for ALCS.
2. The address of the block that contains the reference is in level D0 (field name EBCCR0) of the ECB.
3. EBCCR0 is the main storage reference word of level D0.
4. CE1CC0 is the 2-byte area which contains the block size of D0.
5. EBCFA0 is the file address of D0.

6. The user code must use the following statement to return to TPFDF recoup:
   ```
   B 0(,R6)
   ```

Figure 45 shows an example of how the CDR parameter is used.

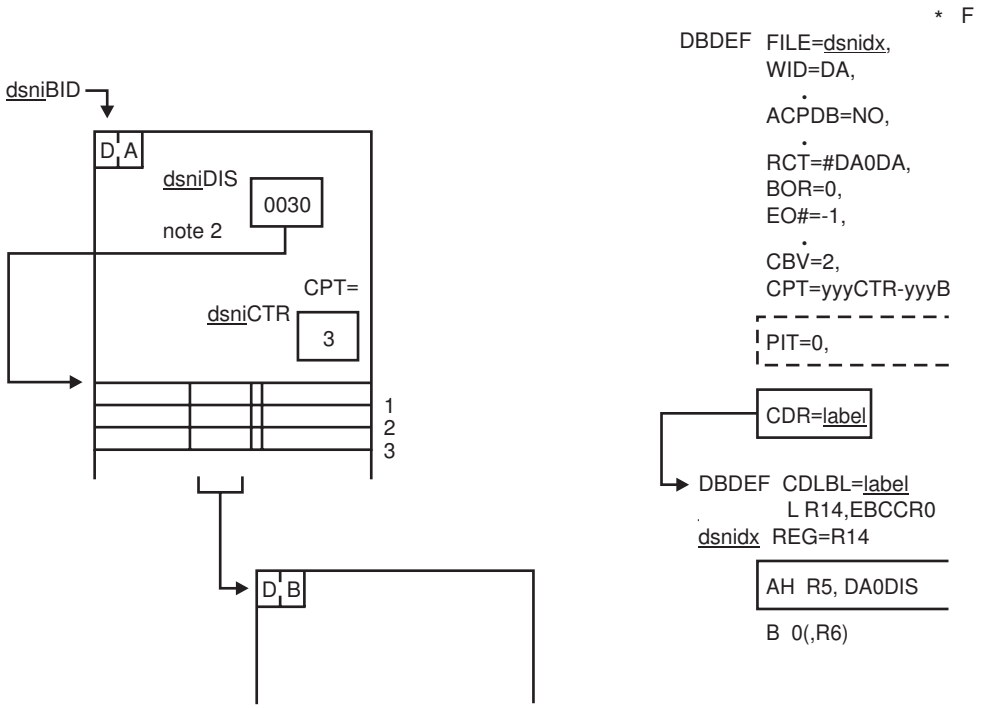


Figure 45. Using the CDR Parameter to Override PIT Parameter Value

Notes:

1. The COA parameter is not valid for ALCS.
2. Figure 45 shows a CNT-type file in which the position of the first item is not fixed and cannot be determined when the DBDEF is defined.
3. Each block contains a displacement field (*dsni*DIS), which contains the position of the first item in the block.
4. User code is required to override the PIT value.
5. Register 5 (R5) points to the first item (based on the PIT parameter), so the PIT parameter is zero (0).

CEB

User code to be run before references in the block are processed.

Notes about the user code:

1. The CDE parameter is not valid for ALCS.
2. The address of the block about to be processed is in level D0 (field name EBCFA0) of the ECB.
3. The user code must use one of the following statements to return to TPFDF recoup:
   - To inhibit TPFDF recoup from processing this block, use:
     ```
     B 0(,R6)
     ```
   - To allow TPFDF recoup to process this block, use:
     ```
     B 4(,R6)
     ```

CEE

User code to be run after references in the block are processed.

Notes about the user code:

1. The address of the block to be processed is in level D0 (field name EBCFA0) of the ECB.
2. The user code must use the following statement to return to TPFDF recoup:

   ```
   B 0(,R6)
   ```

COA

User code to be run just after a block is retrieved.

Notes about the user code:

1. The COA parameter is not valid for ALCS.
2. The address of the block that has just been retrieved is in level D0 (field name EBCCR0) of the ECB.
3. The user code must include one of the following statements to return to TPFDF recoup:
   - To inhibit TPFDF recoup from chain chasing a block, use:

     ```
     B 0(,R6)
     ```
   - To allow TPFDF recoup to process this block, use:

     ```
     B 4(,R6)
     ```

COE

User code to be run if there is an error when retrieving a block.

Notes about the user code:

1. The address of the block retrieved is in level D0 (field name EBCFA0) of the ECB.
2. The user code must use one of the following statements to return to TPFDF recoup:
   - To inhibit TPFDF recoup from trying to retrieve the block again, use:

     ```
     B 0(,R6)
     ```
   - To retrieve this block again, use:

     ```
     B 4(,R6)
     ```

CPF

User code to determine the location of the forward chain field.

Notes about the user code:

1. CPF is not allowed with PFC.
2. PFC or CPF is necessary if FCH is present.
3. The user code should write the location of the forward chain field into EBW030.
4. The address of the block that contains the reference is in level D0 (field name EBCCR0) of the ECB.
5. The user code must use the following statement to return to TPFDF recoup:

   ```
   B 0(,R6)
   ```

CDLBL

specifies an assembler label that defines the start of user code.

Notes:

1. CDLBL must be the last DBDEF macro statement coded in a segment or the DBDEF table will be corrupted.

2. If a work area is required, the following sequence is recommended:

```
GETCC Dx,Lx
DBDEF CDLBL=RAP
```

3. If there is an SVC in the user code, it must be followed by a `DBDEF CDLBL=RAP` statement to reinstall the program base. For example:

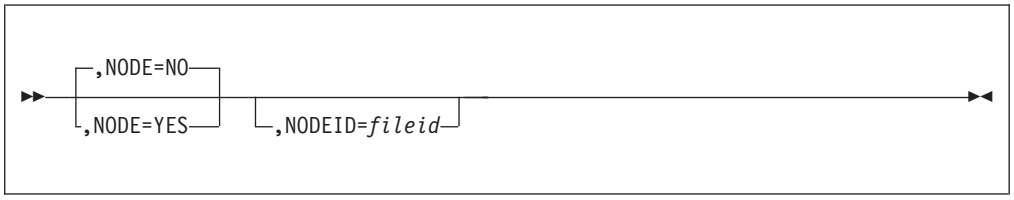
```
DBDEF CDLBL=label
       .          * User code
       .          * More user code
      GETCC DE,L1 *   (example of a macro that issues an SVC)

DBDEF CDLBL=RAP
       .          * More user code (continued)
      B 0(,R6)    *   (return to TPFDF recoup)
```

4. If the following registers that are used to return to TPFDF recoup have been modified, restore them:
   - R6, for all user code parameters except the BASECOD parameter
   - R7, for the BASECOD parameter.
5. All TPFDF recoup user code is entered in 31-bit mode.

B+Tree File Parameters

The following describes DBDEF parameters that are used exclusively for B+Tree files. B+Tree indexing requires other DBDEF parameters and has other considerations than are listed here. For more information, see “B+Tree Indexing” on page 149.



NODE
specifies one of the following:

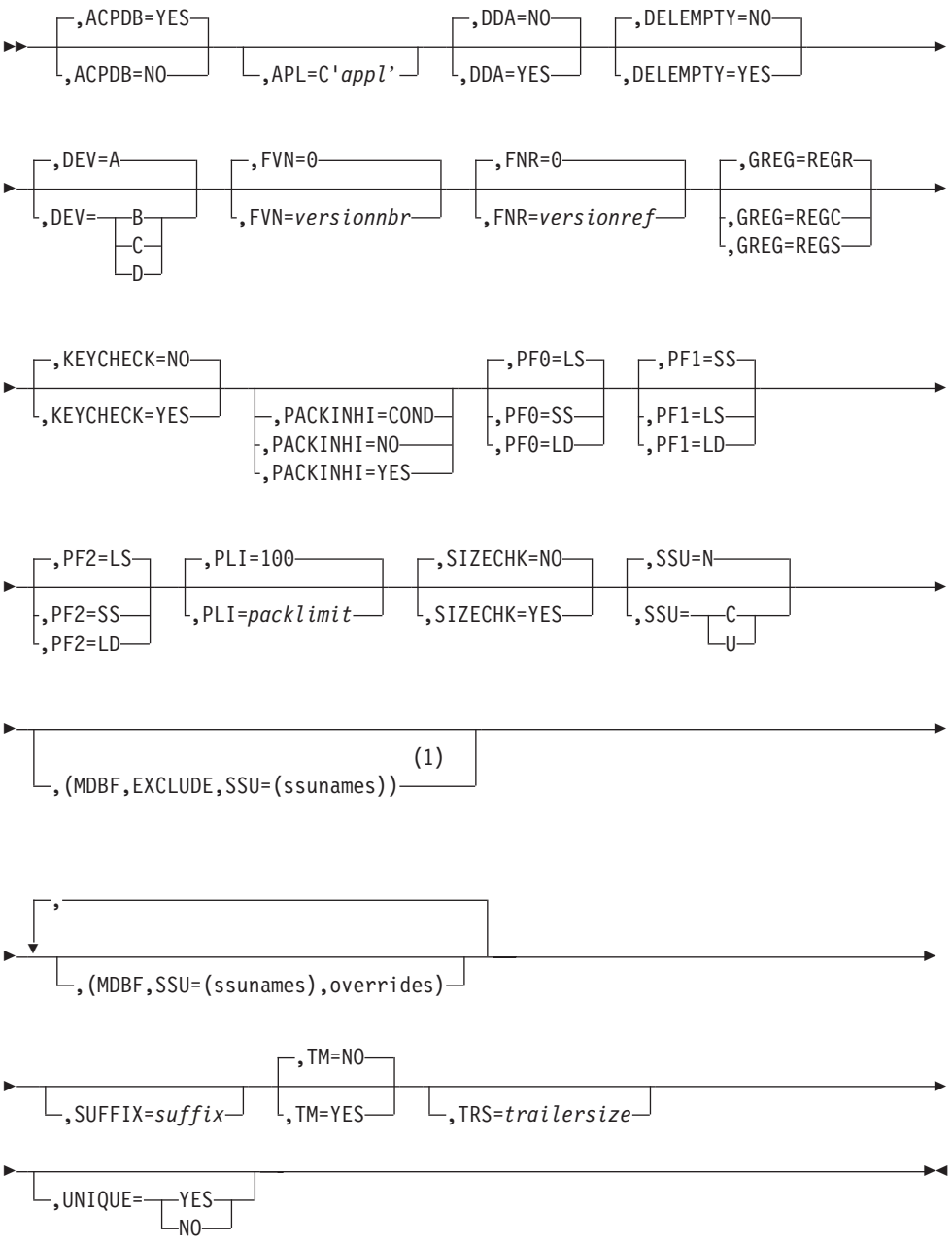
NO
specifies that this file is not a B+Tree index file.

YES
specifies that this file is a B+Tree index file.

NODEID=*fileid*
specifies the *fileid*, where *fileid* is the file ID of a B+Tree index file associated with this data file. You can find this value in the DSECT of the selected B+Tree index file.

Note: NODE=YES and NODEID cannot be used in the same DBDEF statement.

Miscellaneous Parameters



Notes:

1 MDBF parameter statements are only used if SSU=U.

ACPDB
specifies whether this is a TPFDF file.

NO specifies that this a non-TPFDF file. The TPFDF product uses the

DBDEF macro definitions for chain chasing during the processing of recoup and CRUISE, but cannot access or manipulate the file using other macros and functions.

YES specifies that this is a TPFDF file. The TPFDF product uses the DBDEF macro definitions for chain chasing during the processing of recoup and CRUISE, and can access and manipulate the file using other macros and functions. The file type must be R, W, T, or P. See “File Names” on page 69 for more information about file types.

APL=C'*appl***'**
overrides any application type that was specified in the first character of the 6-character DSECT name, where *appl* is a one-character type of application. For example:

```
APL=C'G'
```

See “File Names” on page 69 for more information about application types.

DDA
specifies the distributed data access support.

Note: DDA is not allowed for P-type files.

DELEMPTY
specifies one of the following:

YES
specifies that the TPFDF product will delete empty blocks without packing the subfile. Setting DELEMPTY=YES requires the following:

- The file must be R-type.
- Backward chaining must be indicated by bit 0 of global set symbol &SW00OP1.
- The DBDEF NODE parameter must be NO.
- The DBDEF FVN parameter must be 0.

NO
specifies that the TPFDF product will delete empty blocks only when the subfile is packed.

DEV
specifies the device type used by CRUISE with the TPF GETFC macro. The DEV parameter has no effect on ALCS systems.

Note: A, B, C, and D refer to different DASD types defined at system initialization. See *TPF System Generation* for more information.

FVN=*versionnbr*
identifies the version of a file, where *versionnbr* is a number from 0–255. For example,

```
FVN=3
```

identifies the file as version 3. Other versions of the file can be defined with a different block layout.

Note: The DBDEF macro statements for the different versions of a file should be defined contiguously using increasing file version numbers. Only the first file version (FVN=0) is indexed in the DBDEF index table.

FNR=*versionref*

identifies the version of a file that this file refers to, where *versionref* is a number from 0–255. The FNR value equals the FVN value of the referenced file. For example,

```
FNR=3
```

refers to version 3 (FVN=3) of the referenced file.

GREG

specifies the register call for global base.

REGR

specifies that R14 is the base register for global area 1.

REGC

specifies that R14 is the base register for global area 3.

REGS

specifies that R14 is the base register for global area 3.

KEYCHECK

specify YES to validate the default keys before replacing or modifying an LREC, which will prevent corruption of the file organization.

Notes:

1. If you use global modification when KEYCHECK=YES, and any of the fields being modified overlap any default key fields for that primary key in the file, the TPFDF product issues a system error (DB0139) and processing ends. All records that were changed before processing ended remain changed.
2. Before you can use global modification with KEYCHECK=YES in an ALCS environment, enable C language support. See *TPFDF Installation and Customization* for more information.
3. If you enter the DBREP macro with KEYCHECK=YES, any keys in effect from previous TPFDF macros become unpredictable.
4. If you enter the DBMOD macro without the ALL parameter after you have entered the DBRED INLINE macro with KEYCHECK=YES, the TPFDF product may not check the keys.

PACKINHI

specifies if packing should be inhibited for this file during recoup phase 1:

COND packing is inhibited only if this file includes a forward index path (see “Forward Index Path Parameters” on page 98) and bit 1 of &SW00OP2 is 0, indicating that pool blocks are reused during a pack operation. Packing is inhibited for these files because it is possible for recoup to miss index LRECs if they are moved to a block that has already been chain chased.

NO packing is not inhibited for this file during recoup phase 1.

YES packing is inhibited for this file during recoup phase 1.

Notes:

1. The default for the PACKINHI parameter is specified by set symbol &INHIDEF in the DBLCL macro. See *TPFDF Installation and Customization* for more information about the DBLCL macro.
2. This parameter has no affect in an ALCS environment.

PF0

specifies the type of pool blocks that are associated with pool file 0 (PF0). The

pool block type associated with PF0 is used by a TPFDF macro or function when the POOLTYP=0 parameter is specified with that macro or function. For example, when PF0=SS:

```
DBOPN REF=IR00DFX,POOLTYP=0
```

opens a short-term pool block.

Note: For W-type files, the default is always short-term (SS) and must not be changed.

LS Long-term nonduplicated pool.

SS Short-term pool.

LD Long-term duplicate pool.

PF1

specifies the type of pool blocks that are associated with pool file 1 (PF1). The pool block type associated with PF1 is used by a TPFDF macro or function when the POOLTYP=1 parameter is specified with that macro or function. For example, when PF1=SS:

```
DBOPN REF=IR00DFX,POOLTYP=1
```

opens a short-term pool block.

PF2

specifies the type of pool blocks that are associated with pool file 2 (PF2). The pool block type associated with PF2 is used by a TPFDF macro or function when the POOLTYP=2 parameter is specified with that macro or function. For example, when PF2=SS:

```
DBOPN REF=IR00DFX,POOLTYP=2
```

opens a short-term pool block.

PLI=*packlimit*

specifies the packing limit in percentage, where *packlimit* is a decimal number from 50–100.

You can use PLI to specify the amount of space to which each block in the subfile should be packed. The value of PLI can be specified between 50% and 100%.

A value of 50% results in half of the space in each block being taken up. A value of 100% results in all of the space being taken up. The values for PLI are:
50–100, for L1 and L2 size blocks
75–100, for L4 size blocks.

Always specify a PLI value greater than the PIN value; otherwise, the file is always below the packing threshold and is packed continuously.

Note: Using the PLI parameter does not ensure that B+Tree files are packed unless there are no nodes in the B+Tree index. To ensure that B+Tree files are packed, use one of the following:

- The ZUDFM OAP command
- A ZFCRU command with the pack option
- The DBCLS macro with the pack option
- The `dfcls` function with the pack option.

SIZECHK
specifies one of the following:

NO specifies that TPFDF macros and functions that use search keys, search each LREC in the current core block.

YES specifies that TPFDF macros and functions that use search keys, obtain a core block and copy the current LREC into that core block before searching the LREC in the copied core block. Specifying SIZECHK=YES prevents the TPFDF product from comparing a search key past the end of a data block, which could cause a TPF system error. Comparing a search key past the end of a data block could occur if any of the following are true:

- The last field in an LREC of the data block has a variable length.
- Short variable length LRECs are used in the data block.
- A long variable length search key is specified with a TPFDF macro or function.
- A search key that is longer than than the length of the searched LREC is specified with a TPFDF macro or function.

Note: While specifying SIZECHK=YES prevents the unlikely possibility of a system error, it significantly impacts system performance and additional system resources are required.

SSU
specifies the subsystem user.

N ignores MDBF parameter statements coded in the DBDEF.

C The file is common; MDBF parameter statements cannot be coded.

U The file is unique. The U parameter only applies for the TPF system.

To aid in migrating database definitions from an MDBF system to a non-MDBF system, the SSU parameter can be set to `N` to override any MDBF parameter statements that are coded in the DBDEF.

```
DBDEF FILE=zzzzzz,   standard DBDEF
      .........,     additional DBDEF parameters
      SSU=N,         ignore any MDBF parameters
      (MDBF,SSU=(DF,LH,AE),WRS=L4),
      (MDBF,SSU=(RF),WRS=L1,EO#=99),
      (MDBF,SSU=(RT),WRS=L1,EO#=9)
```

(MDBF,EXCLUDE,SSU=(*ssunames***))**
prevents the specified subsystem users from issuing TPFDF macros and functions, where *ssunames* is a list of valid SSUs such as AF,LH,BA,AA. For example,

```
DBDEF FILE=zzzzzz,   standard DBDEF
      .........,     additional DBDEF parameters
      SSU=U,         file is not common
      (MDBF,SSU=(AA,BA,UA),RBV=#TPFDB03),
      (MDBF,EXCLUDE,SSU=(RF))
```

Notes:

1. The MDBF parameter only applies for the TPF system.
2. If an application program issues a TPFDF macro or functions from an ***excluded*** subsystem user, the application will end with a DB0137 system error.

Therefore, in the previous example, if an application program issues a TPFDF macro when it is in SSU RF, the application will end with a DB0137 system error.

3. It is important to code the EXCLUDE information so TPFDF utilities run only on the subsystem users that you select. The ZUDFM, ZFCRU, and ZRECP commands will not run on files in a subsystem user that has been excluded.
4. The specified subsystem user must be defined in the SSUDEF copy member or the DBDEF macro will send an error when it is assembled.

(MDBF,SSU=(*ssunames***),***overrides***)**
provides unique parameters for the specified subsystem users.

ssunames
specifies a list of valid SSUs such as AF,LH,BA,AA.

overrides
specifies a list of override values for the specified subsystem users (SSUs). The ARS, BOR, EO#, EOR, ILV, INB, NLR, NOC, OP1, OP2, OP3, PF0, PF1, PF2, PIN, PLI, PTN, RBV, RCT, and WRS DBDEF parameters are allowed.

Note: Because of the different characteristics of algorithms, algorithms can only be overridden (using the RBV override parameter) by specific algorithms that share similar characteristics (are in the same group). For example, #TPFDB09 cannot override #TPFDB01 because the expected algorithm arguments are different. #TPFDB01 expects an alphabetic character while #TPFDB09 expects 8 bytes of data.

Table 36 shows the groups of algorithms that can override each other. The size of the algorithm string (defined by the DBDEF ALG parameter or in the DSECT macro for the file) must be large enough to allow the algorithm to be overridden. For example, if ALG=1 was specified in the detail file, you cannot override the #TPFDB01 algorithm with the #TPFDB02 algorithm in the index file (even though they are in the same group) because the file was defined to allow only a one-byte algorithm. To avoid this problem, define the algorithm to be as large as the maximum length of the algorithm group.

Table 36. Algorithm Groups for Overriding

Group				**Maximum Length**
1	#TPFDB01 #TPFDB06	#TPFDB02 #TPFDB07	#TPFDB03 #TPFDB08	3
2	#TPFDB0A	#TPFDB0B		2
3	#TPFDB04			N/A
4	#TPFDB05			4
5	#TPFDB0C			2
6	#TPFDB0D			N/A
7	#TPFDB09	#TPFDB10		8
8	#TPFDB0F			10

More than one MDBF definition can be defined for a particular file. For example:

```
DBDEF FILE=dsname,   standard DBDEF
      .........,     additional DBDEF parameters
      (MDBF,SSU=(AA,BA,UA),RBV=#TPFDB03),
      (MDBF,SSU=(DF,LH,AE),WRS=L4),
      (MDBF,SSU=(RF),WRS=L1,EO#=99),
      (MDBF,SSU=(RT),WRS=L1,EO#=9)
```

Each SSU takes one of the following values for the file (in order of precedence):

1. The SSU override values (if they are specified)
2. The standard DBDEF override values (if they are specified), RBV, WRS and so on.
3. The &SW00... values specified in the file DSECT.

If you want to code a DBDEF subsystem user override, the name of the subsystem user must be listed in SSUDEF. The SSUDEF copy member is installation specific and lists the subsystem users that can be included in an override.

For each subsystem user, a variable is added to SSUDEF as follows:

```
&SSUV(1)  SETC 'RED'     SSU mnemonic name        (MU0NAM in MS0UT)
&SSUV(2)  SETC 'BLUE'    SSU mnemonic name        (MU0NAM in MS0UT)
&SSUV(3)  SETC 'OR'      SSU mnemonic name        (MU0NAM in MS0UT)
 ⋮
&SSUV(n)  SETC 'GOLD'    SSU mnemonic name        (MU0NAM in MS0UT)
```

If the variable to define a particular subsystem user is not set up, the DBDEF macro issues an MNOTE at assembly time.

Notes:

1. The MDBF parameter only applies for the TPF system.
2. A maximum of 128 subsystem users can be listed in SSUDEF.
3. The DBDEF EXCLUDE parameter, which specifies subsystem users that cannot issue TPFDF macros on a particular file, is considered an override. Therefore, these subsystem users must also be listed in SSUDEF.
4. SSUDEF can be updated to include subsystem users from systems with different MDBF configurations. This allows the DBDEFs to be assembled once against the common SSUDEF and loaded to the two different systems.

SUFFIX=*suffix*

specifies the suffix that can make a file unique, where *suffix* is an alphanumeric character or characters. Some P-type file definitions can generate the same symbols for different files or file versions. Use the SUFFIX parameter to add a suffix to the labels to make the labels exclusive to a file version.

If you code the SUFFIX parameter with a DBDEF macro statement, ensure that the DSECT for the P-type file also supports a SUFFIX parameter because the DSECT for the P-type file is invoked internally by the DBDEF macro with the supplied suffix. For example,

```
DBDEF FILE=FRED1,FVN=0,
  generates
  FREDID   DS H
  FREDFCH  DS F
      ......
      ......
DBDEF FILE=FRED1,FVN=1,SUFFIX=A,
```

```
also generates FREDID and FREDFCH, but adds an 'A' to them
FREDIDA  DS H
FREDFCHA DS F
    ......
```

TM

specifies one of the following:

NO specifies that commit scopes are not used during checkpoint and close processing.

YES specifies that commit scopes are used during checkpoint and close processing. This option is valuable when many files are to be filed out during checkpoint and close processing (for example, detac mode, extensive B+Tree indexing updates, and requests to pack indexes). See *TPFDF Programming Concepts and Reference* for more information about commit scopes.

Note: The application can override the DBDEF TM value by specifying a TM value with the DBCKP or DBCLS macros, or the `dfckp` or `dfcls` functions. See *TPFDF Programming Concepts and Reference* for more information about these macros and functions.

TRS=*trailersize*

specifies the trailer size at the bottom of each block, where *trailersize* is a nonnegative number.

Notes:

1. Standard-format files that do not use algorithm #TPFDB0D have a default TRS value of TRS=36.
2. Standard-format files that use algorithm #TPFDB0D have a default TRS value of TRS=0.
3. For customer-format files, the TRS value has no meaning.

UNIQUE

specify YES to ensure that all LRECs added to a file are unique even if the UNIQUE parameter is not specified with a DBADD macro or `dfadd` function statement. See *TPFDF Programming Concepts and Reference* for more information about the UNIQUE parameter of the DBADD macro or `dfadd` function. The default for non-B+Tree data files is UNIQUE=NO. The default for B+Tree data files is UNIQUE=YES.

Note: UNIQUE=NO is not allowed for B+Tree data files.

Part 3. Examples and Concepts

Database Optimization Examples

In Optimizing the Database Design data tables are optimized through data duplication. The following discusses a number of additional points that you may find helpful.

When optimizing, your aim should always be to reduce the amount of physical I/O processing. This ensures the greatest possible performance gains.

Reducing I/O Processing

You can reduce I/O processing in a number of ways. The following are the most common methods:

- Duplicate selected data
- Use block indexing
- Use B+Tree indexing
- Reduce the number of file accesses
- Combine 2 or more files
- Use an algorithm instead of an index.

Optimizing the Database Design discusses data duplication at length. “Indexing” on page 135 discusses block and B+Tree indexing. The following discusses the remaining methods of reducing I/O processing.

Reducing File Accesses

You may be able to reduce the number of file accesses by moving data from detail subfiles into the index file. Figure 46 shows a TPFDF index file with a number of detail subfiles.

Note: The *Name* fields refer to the passenger name.

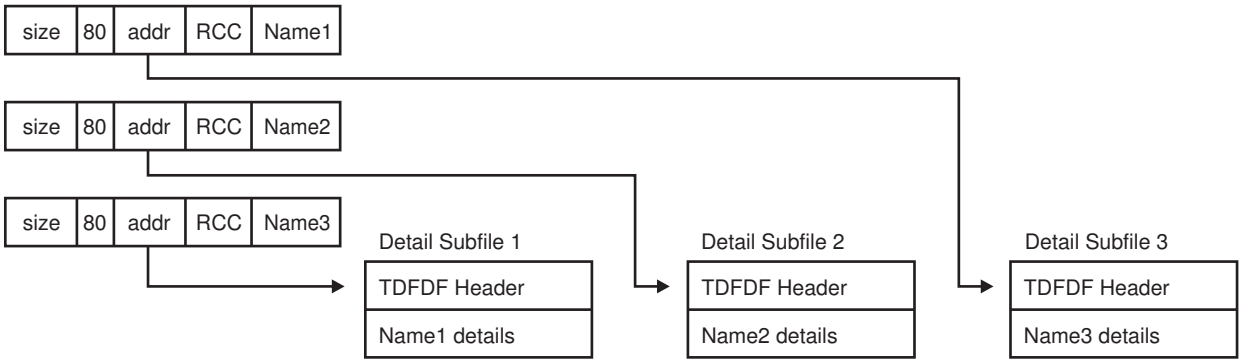


Figure 46. Index File Pointing to Detail Subfiles

The number of file accesses could be reduced considerably if the data in the subfiles were moved to the index file. Figure 47 shows the resulting single file.

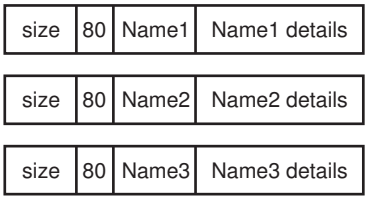


Figure 47. Subfile Data Moved to Index File

The original index file contained pointers to the detail subfiles. Now that these have been removed, the pointers are no longer needed. As you can see here, they have been removed from the index file.

Note: If the detail subfiles contain a lot of data, it may not be feasible to move this data to the index file. Working from the requirements of your application, you will need to estimate how much data can be stored in the index file without lowering performance levels.

Combining Files

When combining files, you may need to increase the number of ordinals in the top-level index file to prevent a large overflow in the upper levels of the files.

Figure 48 shows a TPFDF index file with pointers to different detail subfiles.

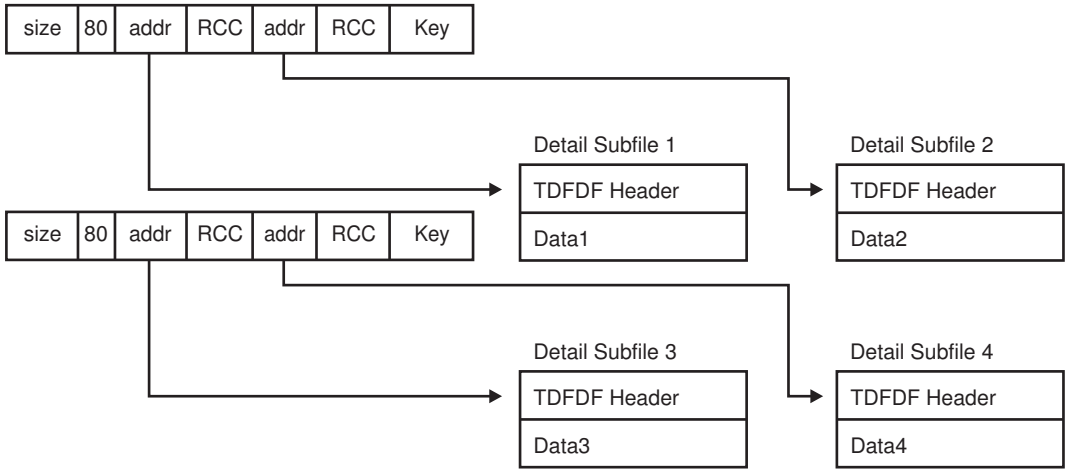


Figure 48. Index File Pointing to Different Subfiles

In Figure 49, the two detail subfiles have been combined to reduce I/O processing:

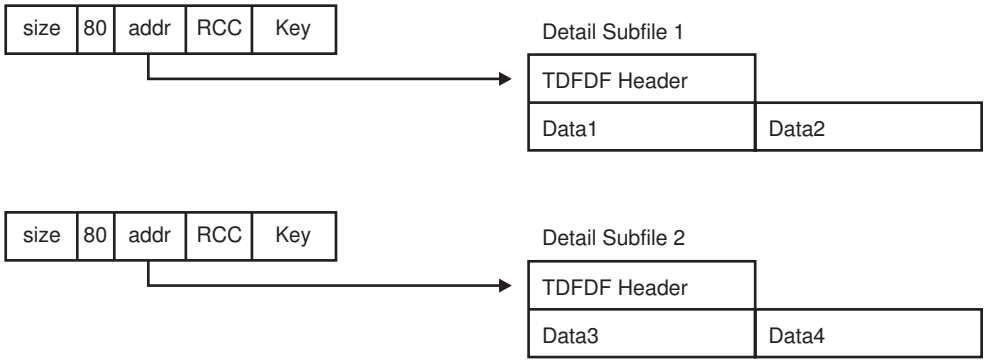


Figure 49. Index File Pointing to Combined Subfiles

Using Algorithms instead of Indexing

This method of reducing I/O is especially useful where some parts of a file are more frequently accessed than others. For example, it may not be efficient to index every date in a flight file when only the earlier dates are being accessed frequently.

You can code a mathematical function to translate the index key (for example, *date*) and then use TPFDF algorithm #TPFDB05 to locate the ordinal directly. Alternatively, you could use TPFDF user exit UWBD to define an algorithm of your own. See page 79 for more information about creating user-defined algorithms.

Figure 50 shows an index file with each date pointing to a separate detail subfile:

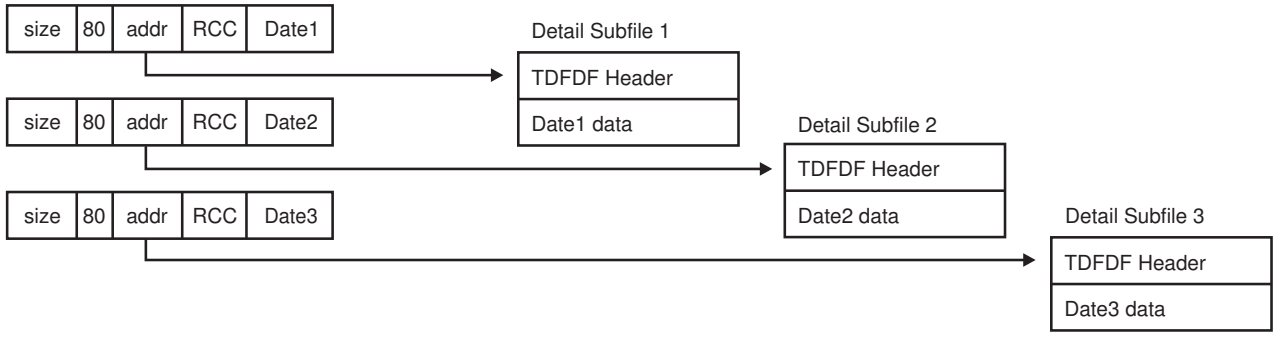


Figure 50. Subfiles Accessed from an Index File

In Figure 51, the index file is replaced by a calculated value, which is passed to the #TPFDB05 algorithm to locate the ordinal directly.

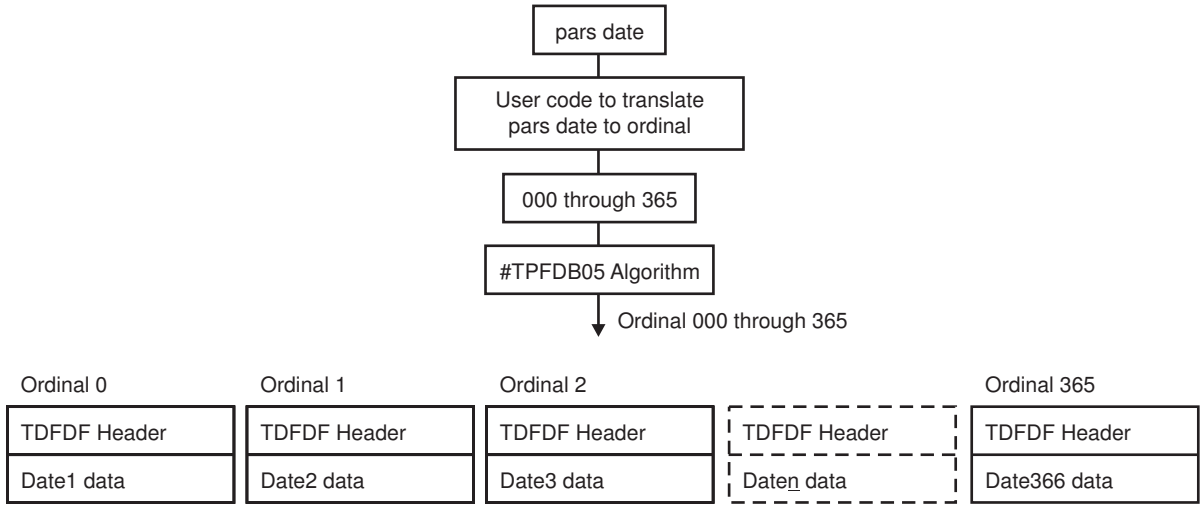


Figure 51. Subfiles Accessed Using Algorithm #TPFDB05

Note: Where an uneven distribution of LRECs is likely, user-defined algorithms can generate a better distribution. For example, where passenger names are being distributed alphabetically, there are a greater number of names beginning with S than with X.

The distribution will probably be very uneven if you use algorithms #TPFDB01, #TPFDB02, or #TPFDB03. However, if you define your own algorithm, you can accommodate this difficulty by creating extra ordinals for the more common characters. See 79 for more information about user-defined algorithms.

Indexing

TPFDF index support allows you to split large amounts of data into logical groups. This reduces the amount of physical I/O processing required, allowing faster access to data.

However, with small files, sequential accessing may provide a more effective means of accessing data.

TPFDF index support consists of the following:

- Basic indexing
- Block indexing
- B^+Tree indexing.

Basic Indexing

There is no real limit to the number of different basic indexing methods available in the TPFDF product. However, the methods that follow are adequate for most applications:

- Simple indexing
- Multiple indexing to a single detail subfile
- Multiple-level indexing
- Single indexing to multiple detail files.

Note: In the sample DBDEFs in this section, the *forward* path defines the indexing structure for recoup purposes. The *backward* path defines the relationship between files in the index structure.

Simple Indexing

Figure 52 shows an index file containing index LRECs. These LRECs reference the subfiles in a detail file.

In this example, there is only one ordinal in the index file, so algorithm #TPFDB04 is used. The detail file contains two sets of LRECs. It is organized in ascending order (UP organized) by the LREC ID (PKY=).

The LRECs in LREC ID X'80' are further organized in ascending order on field GR25DF2. The LRECs in LREC ID X'90' are organized in descending order (DOWN organized).

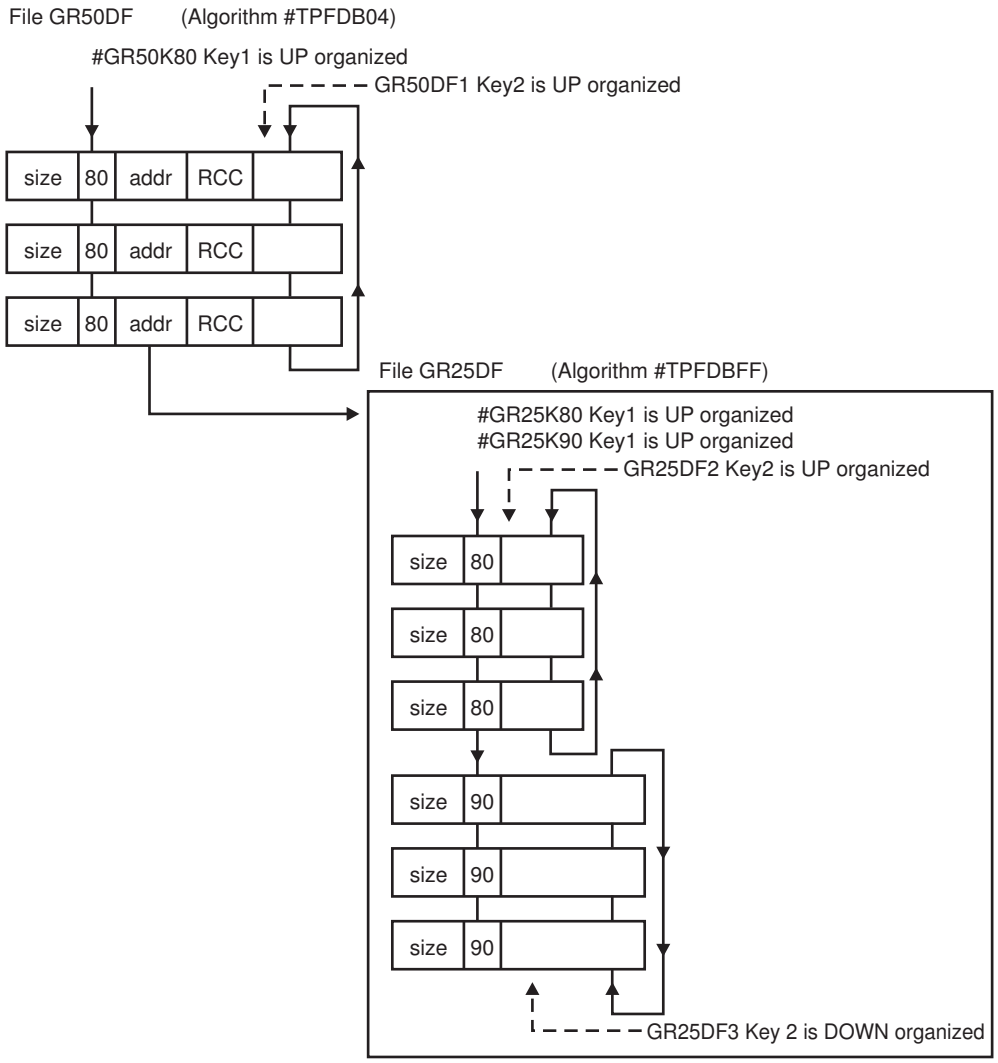


Figure 52. File Description for Simple Indexing

DBDEFs

Top-level index file

```
DBDEF FILE=GR50DF,              file GR50DF                        -
      (PKY=#GR50K80,                 primary key X'80'             -
      KEY1=(PKY=#GR50K80,UP),          - default keys              -
      KEY2=(R=GR50DF1,UP)),                                        -
```

```
        (ITK=#GR50K80,                forward path, index LREC      -
        ID2=,                         type of reference             -
        INDEX=(GR25DF,0))             indexed file, index slot 0
```

Detail file

```
   DBDEF FILE=GR25DF,              file GR25DF                       -
        (PKY=#GR25K80,                 primary key X'80'             -
        KEY1=(PKY=#GR25K80,UP),          - default keys              -
        KEY2=(R=GR25DF2,UP)),                                        -
        (PKY=#GR25K90,                 primary key X'90'             -
        KEY1=(PKY=#GR25K90,UP),          - default keys              -
        KEY2=(R=GR25DF3,DOWN)),                                      -
        (IID=GR50DF,                   backward path, default path 0 -
        IKY=#GR50K80,                  index LREC                    -
        IPA=0,                         offset of addressing argument -
        ILA=0,                         length of addressing argument -
        IPK=0,                         offset of index key           -
        ILK=4,                         length of index key           -
        KEY1=(PKY=#GR50K80,UP),        search keys for index         -
        KEY2=(R=GR50DF1,S=0,UP))       default length assumed
```

Multiple Indexing to a Single Detail Subfile

Figure 53 shows an example of a file indexed by 2 files. One index file uses a 6-byte field as an index key. The other file uses a different 8-byte field as an index key.

The detail file (GR25DF) is indexed by two files, so any chain chasing in the structure GRY1DF to GR25DF is repeated for the structure GRY2DF to GR25DF. The duplicate processing is avoided when RCIDID=1234 (and ID2=(RCI)) is specified in the forward path.

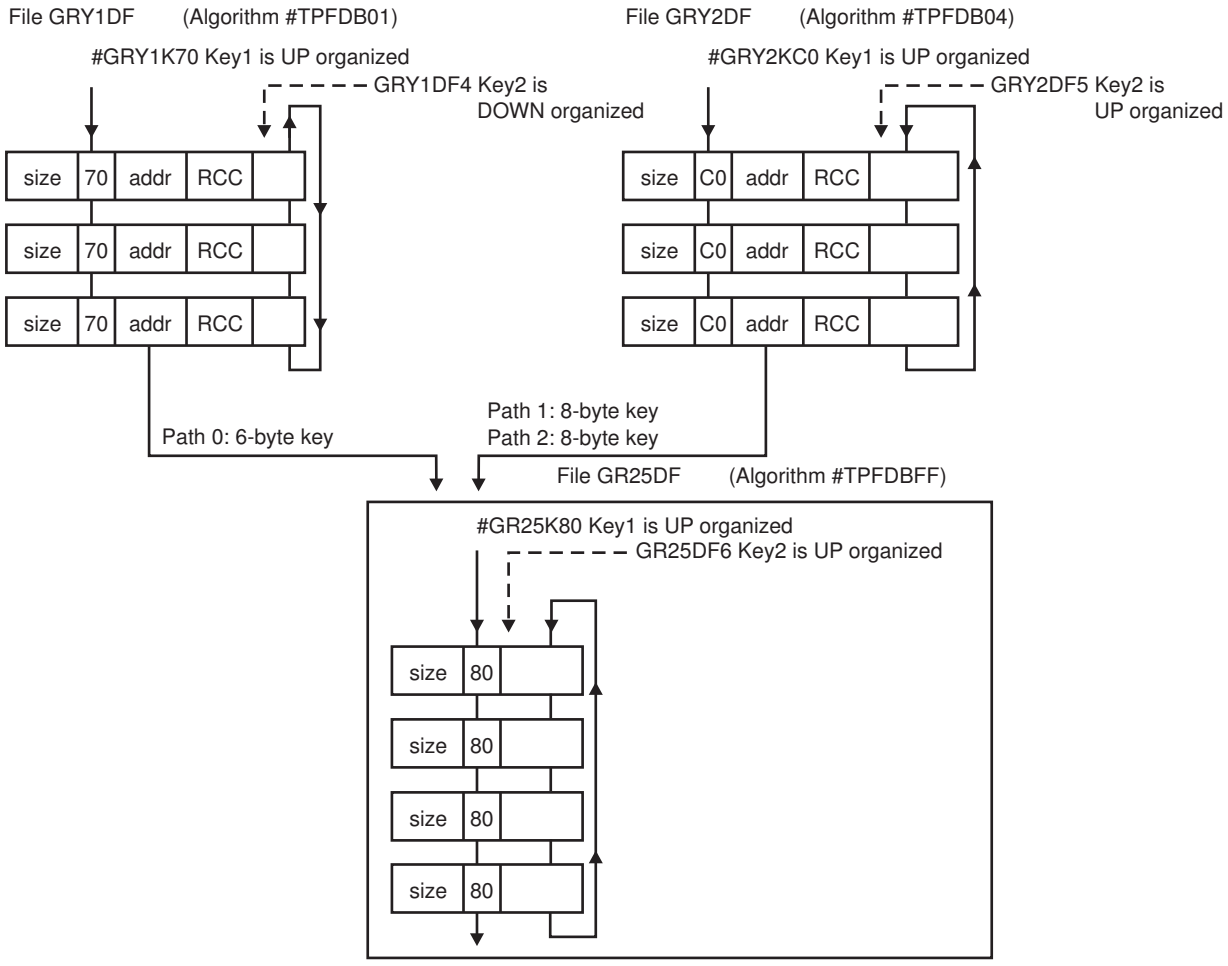


Figure 53. File Description for Multiple Indexing to a Single Detail Subfile

Paths 0 and 1

The index files (GRY1DF and GRY2DF) must be updated each time the detail file (GR25DF) is changed. The DBDEF statements for path 0 and path 1 specify *partial* update paths. The two paths are combined when PATH=ALL is specified in a macro or function. This ensures that both index paths are updated at the same time.

Figure 54 shows how the IPK and ILK parameters are defined in the detail file DBDEF. IPK and ILK define the offset into and the length to use of an algorithm string passed with the ALG parameter in a macro or function. The algorithm string

contains both index keys. For more information, see “Creating a DBDEF Macro Definition” on page 89.

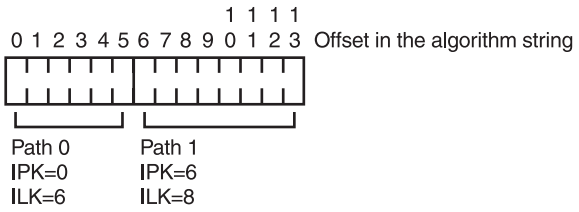


Figure 54. Algorithm String for the Update Path

Read Paths

Figure 55 shows how path 0 is used to update or read the detail file (GR25DF). Path 0 uses the first part of the algorithm string, passed with the ALG parameter in a macro or function, as an index key to read the GRY1DF file, which locates the detail file.

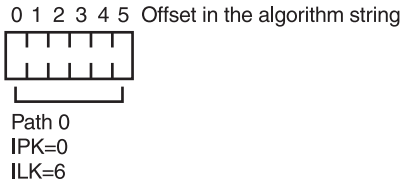


Figure 55. Reading the Detail File through Index File GRY1DF

Figure 56 shows how path 2 is used to read the detail file (GR25DF). (In the definition for path 2, IMI=#BIT0 specifies that path 2 is a *read-only* path.) Path 2 also uses the first part of the algorithm string, passed with the ALG parameter in a macro or function, as an index key to read the GRY2DF file, which locates the detail file.

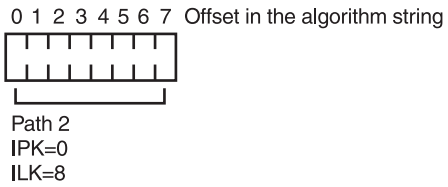


Figure 56. Reading the Detail File through Index File GRY2DF

DBDEFs

Top-level index file 1

```
   DBDEF FILE=GRY1DF,        file GRY1DF                          -
         RCIDID=1234,              RCI processing in this structure    -
         (PKY=#GRY1K70,            primary key X'70'                   -
         KEY1=(PKY=#GRY1K70,UP),     - default keys                    -
         KEY2=(R=GRY1DF4,DOWN)),                                       -
         (ITK=#GRY1K70,            forward path, index LREC            -
         ID2=(RCI),                type of reference                   -
         INDEX=(GR25DF,0))         indexed file
```

Top-level index file 2

```
   DBDEF FILE=GRY2DF,        file GRY2DF                          -
         RCIDID=1234,              RCI processing in this structure    -
         (PKY=#GRY2KC0,            primary key X'C0'                   -
```

```
      KEY1=(PKY=#GRY2KC0,UP),     - default keys                  -
      KEY2=(R=GRY2DF5,UP),                                        -
      (ITK=#GRY2KC0,            forward path, index LREC          -
      ID2=(RCI),                type of reference                 -
      INDEX=(GR25DF,0))         indexed file
```

Detail file

```
   DBDEF FILE=GR25DF,       file GR25DF                            -
      (PKY=#GR25K80,          primary key X'80'                     -
      KEY1=(PKY=#GR25K80,UP),    - default keys                     -
      KEY2=(R=GR25DF6,UP),                                          -
      (IID=GRY1DF,            backward path, default path 0         -
      IKY=#GRY1K70,           index LREC                            -
      IPA=0,                  offset of addressing argument         -
      ILA=1,                  length of addressing argument         -
      IPK=0,                  offset of index key                   -
      ILK=6,                  length of index key                   -
      KEY1=(PKY=#GRY1K70,UP), search keys for index                 -
      KEY2=(R=GRY1DF4,S=0,DOWN)),                                   -
      (IID=GRY2DF,PTH=1,      backward path, path 1                 -
      IKY=#GRY2KC0,           index LREC                            -
      IPA=0,                  offset of addressing argument         -
      ILA=0,                  length of addressing argument         -
      IPK=6,                  offset of index key                   -
      ILK=8,                  length of index key                   -
      KEY1=(PKY=#GRY2KC0,UP), search keys for index                 -
      KEY2=(R=GRY2DF5,S=6,UP)),                                     -
      (IID=GRY2DF,PTH=2,      backward path, path 2                 -
      IMI=#BIT0,              read-only path                        -
      IKY=#GRY2KC0,           index LREC                            -
      IPA=0,                  offset of addressing argument         -
      ILA=0,                  length of addressing argument         -
      IPK=0,                  offset of index key                   -
      ILK=8,                  length of index key
      KEY1=(PKY=#GRY2KC0,UP), search keys for index                 -
      KEY2=(R=GRY2DF5,S=0,UP))
```

Multiple-Level Indexing

Figure 57 shows a two-level index structure where

TPFDF algorithm #TPFDB02 is used to locate the required prime block in the top-level index file. The #TPFDB02 algorithm is used only as an example here, but the #TPFDBFF algorithm must be used for the lower-level files because they are indexed by another file.

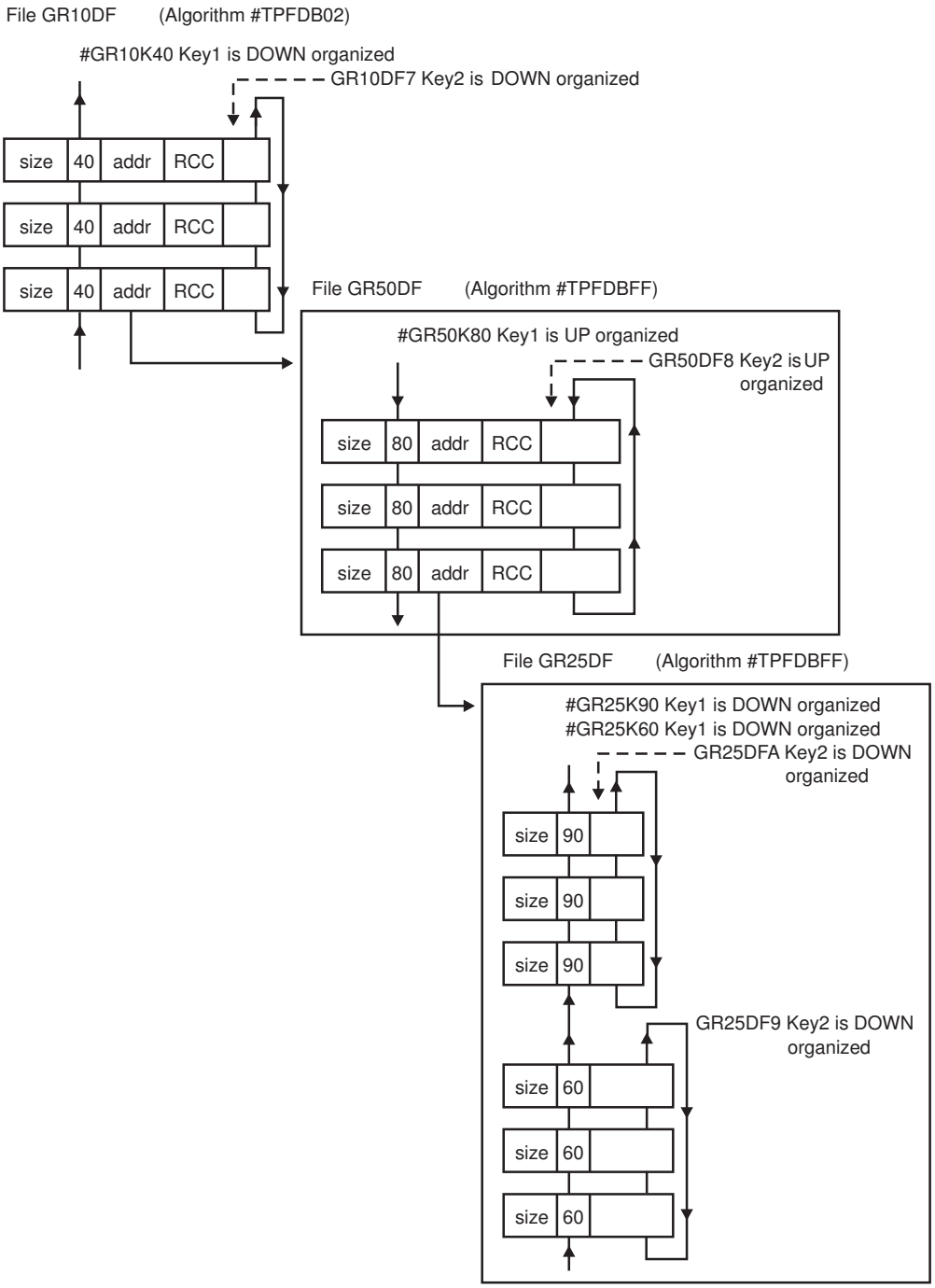


Figure 57. File Description for Multiple-Level Indexing

Addressing Argument and Index Keys for File GR10DF

The top-level index file is addressed using the #TPFDB02 algorithm. #TPFDB02 requires a 2-byte alphabetic input argument (addressing argument). The addressing argument part of the string is defined by IPA= and ILA=. Figure 58 shows how the IPA and ILA parameters are used in the DBDEF for the GR50DF file to specify the addressing argument part for the file GR10DF.

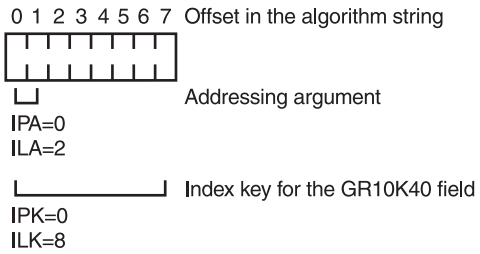


Figure 58. Addressing Argument and Index Key for the Top-Level Index File

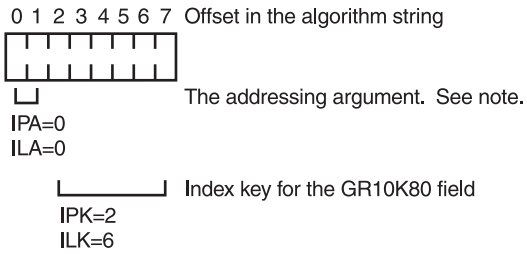


Figure 59. Addressing Argument and Index Key for the Intermediate-Level Index File

Note: The addressing argument is not used by files that are not indexed by a top-level index, but the IPA and ILA parameters must be specified.

DBDEFs

Top-level index file

```
   DBDEF FILE=GR10DF,        file GR10DF                          -
         (PKY=#GR10K40,            primary key X'40'              -
         KEY1=(PKY=#GR10K40,DOWN),   - default keys               -
         KEY2=(R=GR10DF7,DOWN)),                                  -
         (ITK=#GR10K40,            forward path, index LREC       -
         ID2=,                     type of reference              -
         INDEX=(GR50DF,0))         indexed file                   -
```

Intermediate-level index file

```
   DBDEF FILE=GR50DF,        file GR50DF                          -
         (PKY=#GR50K80,            primary key X'80'              -
         KEY1=(PKY=#GR50K80,UP),     - default keys               -
         KEY2=(R=GR50DF8,UP)),                                    -
         (ITK=#GR50K80,            forward path, index LREC       -
         ID2=,                     type of reference              -
         INDEX=(GR25DF,0)),        indexed file                   -
         (IID=GR10DF,              backward path, default path 0  -
         IKY=#GR10K40,             index LREC                     -
         IPA=0,                    offset of addressing argument  -
         ILA=2,                    length of addressing argument  -
         IPK=0,                    offset of index key            -
         ILK=8,                    length of index key            -
         KEY1=(PKY=#GR10K40,UP),   search keys for index          -
         KEY2=(R=GR10DF7,S=0,DOWN))
```

Detail file

```
   DBDEF FILE=GR25DF,        file GR25DF                              -
         (PKY=#GR25K60,           primary key X'60'                   -
         KEY1=(PKY=#GR25K60,DOWN),   - default keys                   -
         KEY2=(R=GR25DF9,DOWN)),                                      -
         (PKY=#GR25K90,           primary key X'90'                   -
         KEY1=(PKY=#GR25K90,DOWN),   - default keys                   -
         KEY2=(R=GR25DFA,DOWN)),                                      -
         (IID=GR50DF,             backward path, default path 0       -
         IKY=#GR50K80,            index LREC                          -
         IPA=0,                   offset of addressing argument       -
         ILA=0,                   length of addressing argument       -
         IPK=2,                   offset of index key                 -
         ILK=6,                   length of index key                 -
         KEY1=(PKY=#GR50K80,UP),  search keys for index               -
         KEY2=(R=GR50DF8,S=2,UP))
```

Single Indexing to Multiple Detail Files

Figure 60 shows a file structure where a passenger name is used to refer to a passenger detail file and another detail file. The duplicate passenger name can be avoided if you have two pointers in the same file related to the same passenger name.

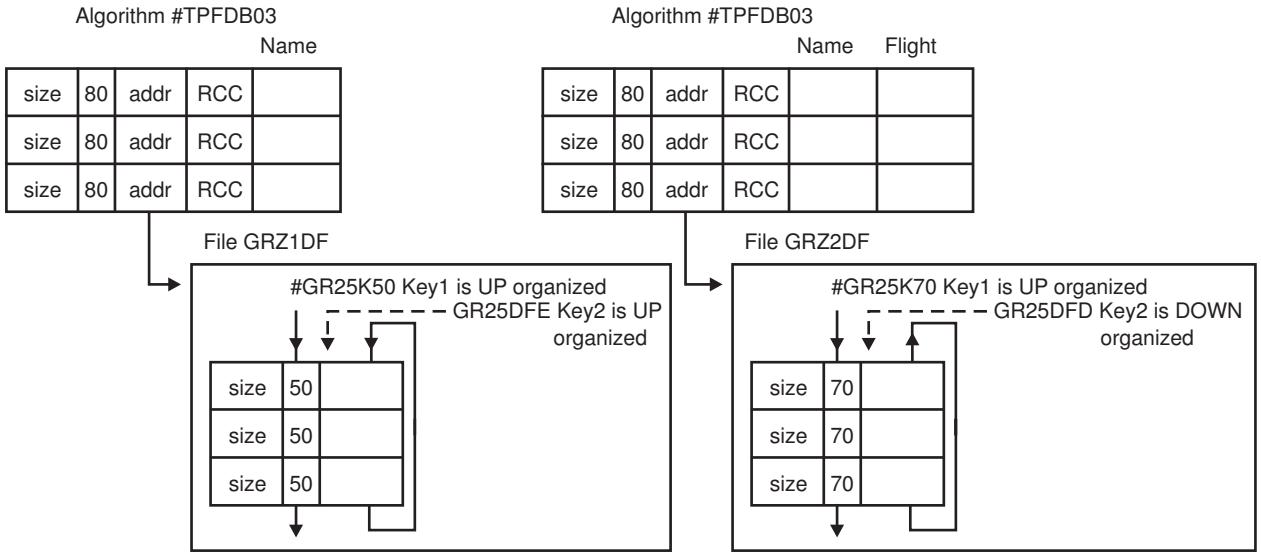


Figure 60. Separate Entries for One Passenger Name

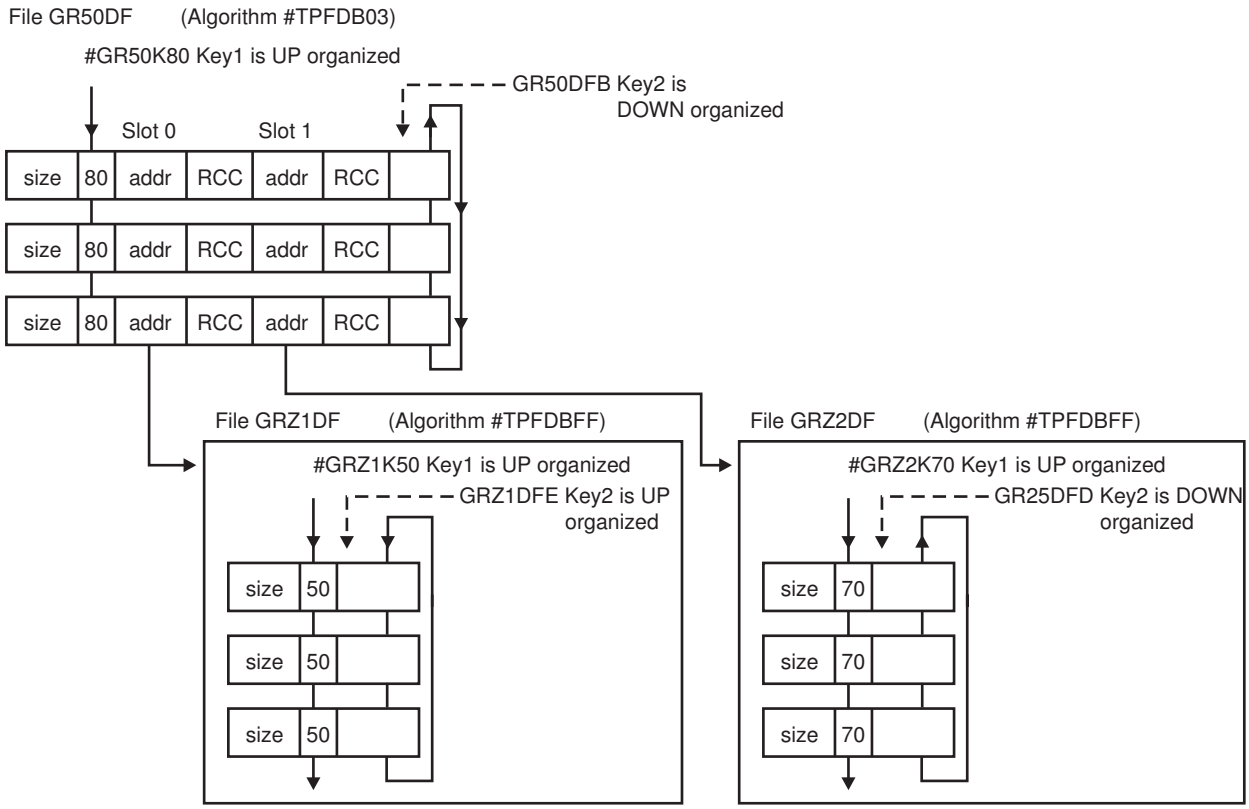


Figure 61. One Index File Pointing to Two Detail Files

DBDEFs

Top-level index file

```
   DBDEF=GR50DF,             file GR50DF                              -
         (PKY=#GR50K80,            primary key X'80'                  -
         KEY1=(PKY=#GR50K80,UP),     - default keys                   -
         KEY2=(R=GR50DFB,DOWN)),                                      -
         (ITK=#GR50K80,            forward path, index LREC           -
         ID2=,                     type of reference                  -
         INDEX=(GRZ1DF,0)),        indexed file, slot 0               -
         (ID2=,                    type of reference                  -
         INDEX=(GRZ2DF,1))         indexed file, slot 1
```

Detail files

```
   DBDEF FILE=GRZ2DF,        file GRZ2DF                              -
         (PKY=#GRZ2K70,            primary key X'70'                  -
         KEY1=(PKY=#GRZ2K70,UP),     - default keys                   -
         KEY2=(R=GRZ2DFD,DOWN)),                                      -
         (IID=GR50DF,              backward path, default path 0      -
         IKY=#GR50K80,             index LREC                         -
         IPA=0,                    offset of addressing argument      -
         ILA=3,                    length of addressing argument      -
         IPK=0,                    offset of index key                -
         ILK=25,                   length of index key                -
         KEY1=(PKY=#GR50K80,UP),   search keys for index              -
         KEY2=(R=GR50DFB,DOWN))
   DBDEF FILE=GRZ1DF,        file GRZ1DF                              -
         (PKY=#GRZ1K50,            primary key X'50'                  -
         KEY1=(PKY=#GRZ1K50,UP),     - default keys                   -
         KEY2=(R=GRZ1DFE,UP)),                                        -
         (IID=GR50DF,              backward path,                     -
         IKY=#GR50K80,             index LREC                         -
         IFR=1,                    index slot 1                       -
         IPA=0,                    offset of addressing argument      -
         ILA=3,                    length of addressing argument      -
         IPK=0,                    offset of index key                -
         ILK=25,                   length of index key                -
         KEY1=(PKY=#GR50K80,UP),   search keys for index              -
         KEY2=(R=GR50DFB,DOWN))
```

Block Indexing

Block indexing considerably reduces I/O processing when files are being read but can make updating slower.

The prime block of a block index subfile contains technical LRECs (TLRECs), which are maintained internally by the TPFDF product.

As you can see in Figure 62, each index TLREC contains the following:
- Primary key (always 02)
- File address of the overflow block
- Record code check (RCC) of the prime block
- Data identifying the keys of the first LREC in an overflow block.

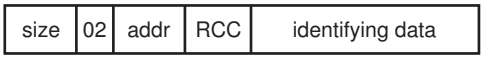


Figure 62. Index TLREC

Note: Not all TLRECs hold the same amount of user data. Use the DSECT to define how much data a TLREC can hold.

To avoid having to look at every LREC in every block, the TPFDF product stores indexing data in the TLRECs. The TLRECs hold data extracted from the first LREC in each overflow block.

The TPFDF product compares each TLREC with the LREC keys that the application program is searching for. If the LREC keys that the application program is searching for are less than or equal to the keys of the first LREC in the current block, the TPFDF product looks back to the previous block to find the matching LREC.

Figure 63 on page 147 shows an example of this process. In this example, the application program is searching for the key, SMITH. The search process is as follows:

1 The TPFDF product looks at the TLREC for overflow block 1 (DAVIDSON). The key held there is less than SMITH.

2 The TPFDF product looks at the TLREC for overflow block 2 (FREEMAN). The key held there is less than SMITH.

3 The TPFDF product looks at the TLREC for overflow block 3. The key held there is equal to SMITH .

4 The TPFDF product searches the previous overflow block (2) for SMITH and finds the first instance of it.

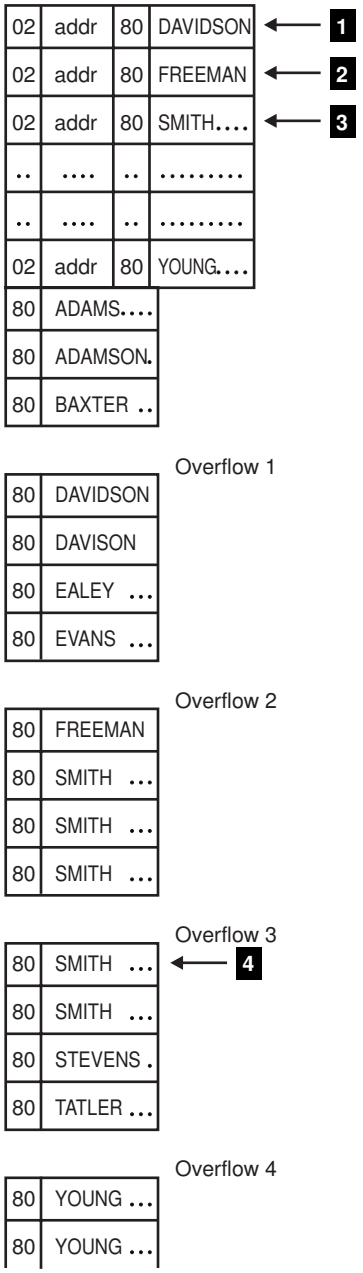


Figure 63. Block Indexing

Implementing Block Index Support

To implement block index support, set global set symbol &SW00SKE in the file DSECT macro definition to the size, in bytes, of the key fields of the LRECs. If there are different types of LRECs in the file, set &SW00SKE to the longest key field.

If you update an existing file to use block indexing, the change is not immediate. The database administrator must first reassemble the DBDEF macro. The TPFDF product adds block index support LRECs to the file when LRECs are deleted from the file or when the file is packed and closed.

Note: The TPFDF product uses X'02' as the LREC ID in LRECs that it uses for block index support. Ensure the value for global symbol &SW00TQK is greater than X'02'.

Block Index File Characteristics

Block index files have the following characteristics:

- LRECs must be variable-length.
- LRECs must be organized UP or DOWN. Do not use NOORG (no organization specified). If you do, the TPFDF product might not locate the correct LREC.
- Every key field in the LREC must have some organization (for example, if KEY1 is UP organized, KEY2 must be UP or DOWN organized). Do not use NOORG (no organization specified). If you do, the TPFDF product might not locate the correct LREC.
- TLRECs are not immediately generated if a file expands and overflow blocks are added. During a pack of the subfile, the TPFDF product determines how many chains there are and stores this number in the first TLREC. During the next pack, the TPFDF product generates this many TLRECs, initializes them, and determines the number of chains (which may have changed since the previous pack). Therefore, the number of TLRECs does not always accurately reflect the number of overflow blocks in a file.
- You must pack a block index file to validate the file references after CRUISE capture and restore processing because CRUISE capture and restore processing nullifies the validation of block index technical LRECs (TLRs).
- TLRECs are only held in the prime block of the subfile. If the prime block becomes full, only those overflow blocks that have corresponding TLRECs are accessed with two physical block accesses. For example, if the prime block containing LRECs pointing to overflow blocks A, B, and C was full, and there were overflow blocks for A–Z, it would take two accesses to access the B overflow block but 24 accesses to access the Z overflow block.
- LREC keys should be unique (for performance reasons).

B+Tree Indexing

B+Tree indexing is a method of accessing and maintaining data. It should be used for large files that have unusual, unknown, or changing distributions because it reduces I/O processing when files are read. Also consider B+Tree indexing for files with long overflow chains.

Note: Unlike block indexing, where the number of I/Os can increase substantially as the file gets larger, the number of I/Os using B+Tree indexing remains minimal regardless of the size of the file.

A B+Tree file consists of a data file, which contains logical records (LRECs), and an index file, which contains technical logical records (TLRECs). The B+Tree index file, which consists of blocks (also called nodes), is maintained internally by the TPFDF product. The index file has its own file ID, DSECT, and DBDEF statements. The prime block of the B+Tree index file (also called the root node) is pointed to by the header in the prime block of the B+Tree data file.

B+Tree indexing is similar to block indexing but has the following advantages:

- Dynamically updates TLRECs
- Has an unlimited number of TLRECs
- Can use mixed key organization for file operations
- Is used for all LREC searches when the specified keys entirely or partially match the default keys in the DBDEF statement.

Notes:

1. If the distribution of overflows can be predicted, you may want to use an algorithm. A well-distributed algorithm that has one or two overflows for each subfile will probably outperform a B+Tree index.
2. You can define a file to use an algorithm and B+Tree indexing. The algorithm accesses the subfile and B+Tree indexing accesses the record in the subfile.

B+Tree Index File Node Blocks

B+Tree index file node blocks contain TLRECs that contain file addresses of the blocks at a lower level of a file. Those lower-level blocks may be other node blocks or data blocks. TLRECs in node blocks have the following layout:

- Size of the TLREC (2 bytes).
- Primary key (1 byte).
  - Key 03 for a node pointing to another node
  - Key 04 for a node pointing to a data block.
- File address of the lower-level node or data block (4 bytes).
- Record code check (RCC) equal to X'00' for leaf nodes, or the last byte of the data file ID for nonleaf nodes (1 byte).
- The concatenation of the primary key and its associated default keys found in the DBDEF of the data file. These keys point to the first TLREC of a child node or the first LREC of a block in the B+Tree data file.

B+Tree Data File Data Blocks

B+Tree data blocks are the same for B+Tree files as they are for other files except the STDSBA field in the prime block header points to the root node of the B+Tree index.

B+Tree Data File Characteristics

To use B+Tree indexing, set the DBDEF macro parameters, equivalent DSECT parameters, or equivalent default values for a B+Tree data file as follows:

- OP1 bits 2, 3, and 6 must be off.
- OP3 bit 5 must be on.
- The RBV algorithm value cannot be #TPFDB0D.
- The TQK value must be greater than 4.
- If present, the PIN value must be less than or equal to 50.
- The TYP value must be R.
- The NOC variable cannot be present.
- The SKE variable cannot be present.
- The NLR variable cannot be present.

Also, the B+Tree data file DBDEF must have the following:

- A PKY statement to define default keys
- NODEID=*fileid* (where *fileid* is the &SW00WID value shown in the associated B+Tree index file)
- KEYCHECK=YES
- UNIQUE=YES
- DELEMPTY=YES if the DBDEF includes statements for recoup to perform multiple ECB chain chasing (see “Multiple ECB Chain Chasing” on page 154 for information about multiple ECB chain chasing).

Note: Before you can implement B+Tree indexing in an ALCS environment, enable C language support. See *TPFDF Installation and Customization* for more information.

Additional Considerations When Using B+Tree Indexing

If you decide to use B+Tree indexing support, keep the following points in mind:

- The sum of all keys, including the primary key, must be unique. Individual keys do not have to be unique. For example, in Figure 64 on page 152, there is more than one LREC with a salary of $90000 but only one with a salary of $90000 and the name ADAMS.
- Each subfile has its own B+Tree index. For example, a file using algorithm #TPFDB01 (26 subfiles) would have 26 separate B+Tree structures.
- More than one file can use the same B+Tree index file DSECT. Each file would have its own B+Tree index structure. For example, data files IR26DF and IR27DF can both have a DBDEF statement for NODEID=B070.
- Packing a B+Tree data file builds or rebuilds the B+Tree structure unless the file contains only a prime block with no overflow blocks.
- You must pack a B+Tree file to validate the file references after CRUISE capture and restore processing because CRUISE capture and restore processing releases B+Tree files.
- If a file is changed from block indexing to B+Tree indexing, existing TLRECs of the block index file are deleted when the file is packed.
- B+Tree indexing does not support extended LRECs.
- Each B+Tree node block must be large enough to contain at least four TLRECs.
- The number of TLRECs that can fit into a node block depends on the size of the blocks and the size of the keys that are used. Because each node header file is

26 bytes, and each TLREC uses 8 bytes in addition to the key size, use the following formula to calculate the number of TLRECs that will be in your node blocks.

```
Number of TLRECs   =   (block size - 26) / (8 + key size)
```

- If you convert an existing file to allow it to use B+Tree indexing, reassemble any assembler applications that use it, to flag any incompatible options. Applications that assemble cleanly do not have to be reloaded.
- A B+Tree index file should ***never*** be packed.

Structure of a Data File That Uses B+Tree Indexing

A data file that uses B+Tree indexing has a B+Tree index file associated with it. The data file consists of data blocks that contain LRECs. The B+Tree index file consists of node blocks that contain TLRECs.

Figure 64 shows data file, GR91SR, which uses B+Tree index file IR70SR. The figure only shows a portion of the index and data files and is not intended to show a complete B+Tree structure. Data file GR91SR shows 4 data blocks. B+Tree index file IR70SR shows a root node and 4 leaf nodes.

Figure 65 on page 153 shows the DSECT and the DBDEF statements for GR91SR. Figure 66 on page 154 shows the DSECT and the DBDEF statements for IR70SR.

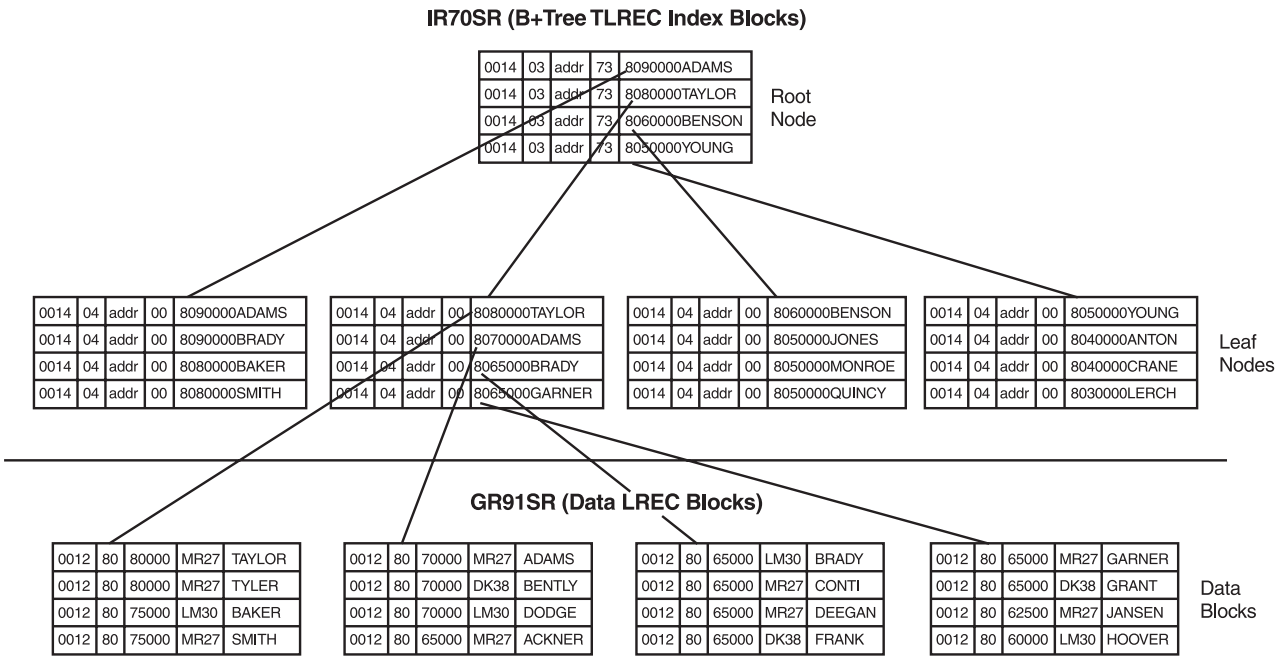


Figure 64. Sample B+Tree File

Defining the DSECT and DBDEF for a Data File That Uses B+Tree Indexing

Figure 65 on page 153 shows part of the DSECT and DBDEF for data file GR91SR, which uses B+Tree indexing. No matter what data is in an LREC, it is organized according to this definition.

The DBDEF includes statements that are necessary for recoup to perform single-ECB chain chasing. Chain chasing a B+Tree file involves chasing a normal chain of data blocks and a companion chain of node blocks.

```
&SW00WID SETC  'B073'      ** FILE ID
&SW00TQK SETC  '15'        ** HIGHEST TLREC

GR91SIZ   DS  H            ** SIZE OF LOGICAL RECORD
GR91KEY   DS  X            ** LOGICAL RECORD IDENTIFIER
#GR91K80  EQU   X'80'       ** LOGICAL RECORD KEY X'80'
GR91ORG   EQU *            ** START OF LOGICAL RECORD DESCRIPTION
GR91SAL   DS  CL5          ** SALARY
GR91DPT   DS  CL4          ** DEPARTMENT
GR91NAM   DS  CL6          ** LAST NAME
GR91E80   EQU *



DBDEF FILE=GR91SR,
      NODEID=B070,                ** B+TREE INDEX FILE
      KEYCHECK=YES,               ** REQUIRED FOR A B+TREE FILE
      UNIQUE=YES,                 ** REQUIRED FOR A B+TREE FILE
      (ID3=(CHK0),RID=B070,ADR=STDSBA-STDREC),
      (PKY=#GR91K80,              ** KEY x'80'
      KEY1=(PKY=#GR91K80,UP),     ** UP ORG ON PKY
      KEY2=(R=GR91SAL,DOWN),      ** DOWN ORG ON SALARY
      KEY3=(R=GR91NAM,UP)), ...   ** UP ORG ON LAST NAME
```

Figure 65. B+Tree Data File DSECT and DBDEF

Defining the DSECT and DBDEF for a B+Tree Index File

Use the sample B+Tree index file DSECT, SAMTSR, to build your own DSECT. You can add statements to define a B+Tree index file with its own characteristics (for example, file ID, WRS size, and so on), but do not change the existing statements. The only DBDEF override values that you can use are:

WRS Sets the block size of the nodes. WRS can be set to any value.

PF0 Sets the type of pool record used to create node blocks; LS (long-term nonduplicated pool), SS (short-term pool), or LD (long-term duplicate pool). PF0 defaults to LS.

PF1 Sets the type of pool record used to create temporary node blocks. Temporary node blocks are used by B+Tree indexing if the number of changed nodes exceeds the number of nodes defined in #TPFNODE in the ACPDBE segment. PF1 defaults to SS.

Figure 66 on page 154 shows part of the DSECT and DBDEF for B+Tree index file IR70SR.

```
&SW00WID SETC  'B070'      ** FILE ID
&SW00RBV SETC  '#TPFDBFF'  ** FILE ALGORITHM
&SW00OP1 SETC  '00000000'  ** OPT BYTE1
&SW00OP2 SETC  '00000110'  ** OPT BYTE2
&SW00OP3 SETC  '00000000'  ** OPT BYTE3
&SW00TQK SETC  '02'        ** HIGHEST TLREC
&SW00NOC SETA  0           ** NUMBER OF CHAINS -FOR ADD CURRENT ONLY-
&SW00PIN SETC  '00'        ** ENSURE NODES ARE NEVER PACKED

IR70SIZ   DS  H            ** SIZE OF VARIABLE LEN LREC
IR70KEY   DS  X            ** PRIMARY KEY
IR70ORG   EQU *            ** START OF LOGICAL RECORD DESCRIPTION
IR70FA1   DS  XL4          ** LOWER LEVEL FADDR
IR70RC1   DS  XL1          ** RECORD CODE CHECK
IR70A03   DS  0CL1         ** KEY FIELDS
IR70E03   EQU *            ** END OF LOGICAL RECORD WITH KEY = X'03'
IR70FA2   DS  XL4          ** LOWER LEVEL FADDR
IR70RC2   DS  XL1          ** RECORD CODE CHECK
IR70A04   DS  0CL1         ** KEY FIELDS
IR70E04   EQU *            ** END OF LOGICAL RECORD WITH KEY = X'04'


DBDEF FILE=IR70SR,TRS=0,NODE=YES
```

Figure 66. B+Tree Index File DSECT and DBDEF

Multiple ECB Chain Chasing

The DBDEF shown in Figure 65 on page 153 includes statements necessary for recoup to perform single ECB chain chasing. This may not be adequate for large data files. As an alternative, you can define the DBDEF for the B+Tree data and index files to allow multiple ECB chain chasing. Figure 67 on page 155 shows one example of how multiple ECB chain chasing can be defined. Depending on the size of the chains and their location in the overall data structure, different methods of chain chasing might be necessary in each customer environment.

```
          STDHD REG=R14

          DBDEF FILE=GR91SR,PFC=-1,DELEMPTY=YES,
                NODEID=B070,
                KEYCHECK=YES,
                UNIQUE=YES,
                (PKY=#GR91K80,
                KEY1=(PKY=#GR91K80,UP),
                KEY2=(R=GR91SAL,DOWN),
                KEY3=(R=GR91NAM,UP)),
                (ID3=(CHK0),RID=B073,FNR=2,ADR=STDFCH-STDREC,CDO=CHK),
                (ID3=(CHK0),RID=B070,ADR=STDSBA-STDREC)
          DBDEF FILE=GR91SR,FVN=1,PFC=-1,
                RCT=0,BOR=0,EO#=0,EOR=0
          DBDEF FILE=GR91SR,FVN=2,
                RCT=0,BOR=0,EO#=0,EOR=0


          DBDEF FILE=IR70SR,TRS=0,NODE=YES,
                (ITK=X'04',ID2=,FNR=1,RID=STDSBA-STDREC,ADR=IR70FA1-IR70REC)


          DBDEF CDLBL=CHK
                L R14,EBCCR0      Load base of block
          STDHD REG=R14
                OC STDSBA,STDSBA  Is root node there ?
                BZ DO_FCH         No - chain chase forward chains
                B 4(,R6)          Yes - don't chain chase fch
DO_FCH          B 8(,R6)          Chain chase forward chains
```

Figure 67. DBDEF for Multiple ECB Chain Chasing

The chain chasing of the structure in Figure 67 is as follows:

- The prime block of GR91SR is found.
- Consider GR91SR as a no forward chain file (PFC=−1).
- Evaluate the CHK code. If there are no nodes, chain chase the remaining data blocks via STDFCH using file version 2.
- Start chain chasing the nodes via STDSBA by going to X'B070' (IR70SR). If it is zero (no nodes), this will stop immediately.
- IR70SR will invoke file version 1 of the data file with a new ECB whenever a X'04' TLREC is found (thus in a leaf node).
- File version 1 of the data file will only cause the current block to be chain chased because it has no forward chains (PFC=−1).

Note: To chain chase a B$^+$Tree index and data file using multiple ECBs, you must code the DELEMPTY parameter as YES on the DBDEF of the B$^+$Tree data file.

Partitioning and Interleaving

You can subdivide the ordinals in a fixed file into groups called *partitions* or *interleaves*. Each collection of blocks in a partition or interleave is still referred to as a file. An application program normally processes a single file, that is a single partition or interleave, but it need not specify the partition or interleave until run time (it sets a PARTITN or INTERLV parameter in the program to define which partition or interleave it is to access). The total number of prime blocks (ordinals) required is:

```
Number of partitions or interleaves  x  Number of prime blocks in each file
```

In each partition or interleave, each file contains the same number of prime blocks (ordinals) as the other files in the partition or interleave. Partitioning and interleaving is not available with miscellaneous files.

With partitioned or interleaved files, you can specify the total number of blocks required by setting the EO# parameter in the DBDEF macro.

Partitions

Figure 68 shows an example of three partitioned files. Seventy-eight blocks of a fixed file are subdivided into three partitions. The relative ordinal numbers in each partition are numbered starting at zero.

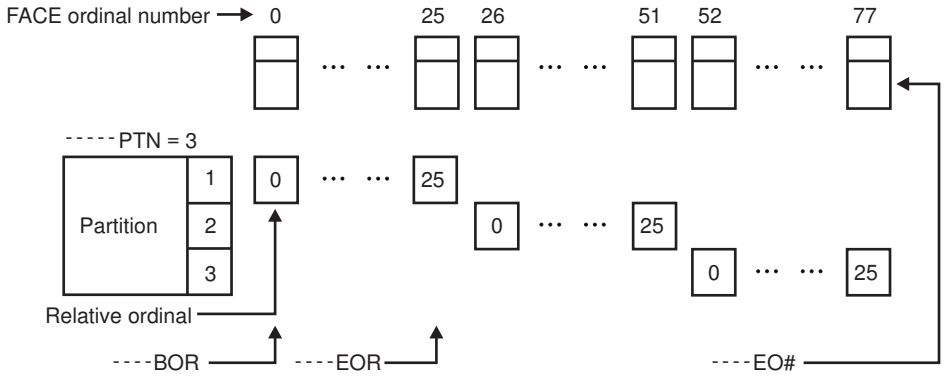


Figure 68. Partitioning: &SW00xxx: PTN, BOR, EOR, and EO#

Note: For partitioned files, EOR must be set to the number of ordinals in the partition. EO# can be set to −1 or to a specific ordinal.

Advantages of Partitioned Files

You can easily add a new partition (for example, to accommodate a new set of customers) by doing the following:

1. Increase the number of ordinals in the fixed file
2. Reset the parameter EO# to the new number of ordinals (if not −1)
3. Redefine the number of partitions in the DSECT macro (using &SW00PTN).

It is more difficult to increase the number of blocks in each partition. This requires a reorganization of the files.

Partitioning is appropriate when the number of subfiles in each table is likely to stay the same but the number of files might change.

Example of Partitioning

Suppose you maintain a set of customer name files. The customers live in three cities. Sometimes you need to access the customer LRECs directly and sometimes you need to process a set by city.

Assume that the number of customers is such that algorithm #TPFDB01 is appropriate. This means that you need 26 prime blocks for each file (partition).

Table 37 shows the customers allocated to ordinals in each file on the basis of the first character in their surname (algorithm #TPFDB01) and to a partition on the basis of which city they live in (Rome=1, Paris=2, London=3).

The total number of ordinals (blocks) required for all 3 partitions is 78 (26 x 3).

Table 37. Allocation in Partitioned File

Customer name	City	Partition number	Ordinal number in the file (partition)	Ordinal number in the fixed file
Adams	Rome	1	0	0
Andrews	Rome	1	0	0
Bernoulli	Rome	1	1	1
Rodriquez	Rome	1	17	17
Zerlini	Rome	1	25	25
Andrews	Paris	2	0	26
Zeisel	Paris	2	25	51
Alans	London	3	0	52
Zimmerman	London	3	0	77

Coding the DSECT for Partitioned Files

The following statements are needed in the DSECT macro of a file, to allow application programs to access data in any of three partitions.

```
&SW00PTN SETC '3'         Number of partitions
&SW00EOR SETC '25'        Ending ordinal of partition 0
&SW00RBV SETC '#TPFDB01'  Addressing algorithm
&SW00EO# SETC '77'
```

Note: &SW00PTN defines a set of three partitioned files. Code the remaining &SW00*xxx* fields, including &SW00RBV, as if the file were not partitioned.

The application program can specify which partition it is to access by using the PARTITN parameter in a DBOPN macro. For example:

```
DBOPN REF=zzzzzz,...,PARTITN=2
```

Alternatively, the program can omit the PARTITN parameter with the DBOPN macro and specify the partition by using the PARTITN parameter with other macros (for example, DBRED or DBADD). This is less controlled and is not recommended.

For C applications, a program accesses a partition using the `dfopt` function with the DFOPT_PARTITION option.

Adding a New Partition

To add a new partition to the fixed file (in the example, to add another city), allocate another 26 blocks and reset the EO# parameter in the DBDEF to 104:

```
EO#=104    Set total number of blocks in the fixed file
```

Also, update the &SW00PTN setting in the DSECT macro to 4:

```
&SW00PTN SETC '4'         Number of partitions (override)
```

Alternatively, you could override any DSECT setting of &SW00PTN by setting a parameter PTN in the DBDEF:

```
PTN=4     Set number of partitions
```

Interleaves

Figure 69 shows the same 78 blocks of a fixed file divided into three interleaves. The ordinal numbers in each interleave start at zero.

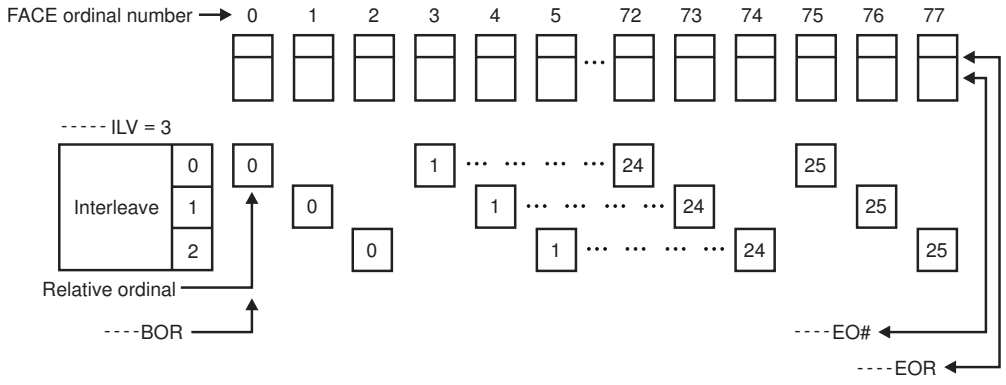


Figure 69. Interleaving: &SW00xxx: ILV, BOR, EOR, and EO#

Note: For interleaved files, EO# and EOR can be set to −1 or to a specific number of ordinals.

Advantages of Interleaved Files

With interleaved files, it is easy to increase the size of each interleave (that is, the number of blocks in each file).

In the previous example (relating to customers in three cities) interleaving would be appropriate if the number of cities is unlikely to change, but the number of customers in each city is likely to change substantially. This would require a change in the number of prime blocks (ordinals) in each interleave.

It is difficult to add a new file to an interleaved fixed file. This requires reorganization of the files.

Interleaving is appropriate when the number of files is likely to stay the same but the number of subfiles in each file might change.

Coding the DSECT Macro for Interleaved Files

The following statements are needed in the DSECT of any program that accesses data in any of the three interleaves:

```
&SW00ILV SETC '3'         Number of interleaves
&SW00EOR SETC '77'        Ending ordinal
&SW00RBV SETC '#TPFDB01'  Addressing algorithm
&SW00EO# SETC '77'
```

&SW00ILV defines a set of three interleaved files. Code the remaining &SW00*xxx* fields, including &SW00RBV as if the file were non-interleaved.

The application program can specify the interleave by coding the INTERLV parameter in a DBOPN macro. For example:

```
DBOPN REF=zzzzzz,...,INTERLV=label
```

In this example, store the interleave number in location *label* before coding the DBOPN macro statement.

Alternatively, the program can specify the interleave by using the INTERLV parameter when it calls other macros (for example, DBRED or DBADD). This is less controlled, and is not recommended.

For C applications, a program accesses an interleave using the `dfopt` function with the DFOPT_INTERLEAVE option.

Adding Blocks to an Interleave

With interleaved files, it is easy to increase the number of blocks in each interleave. For example, to add one subfile to each interleave in the previous diagram, add 3 blocks to the fixed file. Set the DBDEF EO# parameter to the new number of blocks:

```
EO#=80     Set end ordinal number in the fixed file
```

Change the DSECT statements for the interleaved file, as follows:

```
&SW00EOR SETC '80'
&SW00EO# SETC '80'
```

Database Design Hints and Tips

The following contains a number of hints and tips that may be of some help to you when designing your TPFDF database.

The material is presented in the form of a number of problems, each with its appropriate TPFDF solution. Where applicable, DSECT settings, DBDEF statements, and application code samples are provided with the solution.

File Integrity

The following scenario addresses file integrity.

Problem

An application has destroyed an UP organized file because the program did not specify key parameters correctly when LRECs were added to the file.

Solution

With UP or DOWN organized files, always define the update keys as default keys in the DBDEF macro. The DBDEF statement overrules any key specifications in the DBADD macro or `dfadd` function. This maintains file integrity and eliminates the risk of application programs using incorrect key parameters when adding LRECs to a subfile.

The default keys are also used by the various ZUDFM commands. This prevents accidental corruption of the file organization. See *TPFDF Utilities* and *TPFDF Commands* for more information about the ZUDFM commands.

The TPFDF product checks that the primary key of the LREC to be added to the file is defined in the DBDEF. A mismatch results in a system error.

Note: You can use any keys that you want, including partial keys, to read LRECs from a subfile. However, if you are adding LRECs, use the keys defined in the DBDEF.

DSECT Set Symbols

Not applicable.

DBDEF Statements

```
DBDEF FILE=zzzzzz,                        -
      (PKY=#zzzzK80,                      -
      KEY1=(PKY=#zzzzK80,UP),             -
      KEY2=(R=zzzzFLD,UP))

or

DBDEF FILE=zzzzzz,                        -
      (PKY=#zzzzK80,                      -
      ORG=UP,                             -
      KEY1=(PKY=#zzzzK80),                -
      KEY2=(R=zzzzFLD))
```

Application Coding

```
DBADD REF=zzzzzz,NEWLREC=location
  TPFDF uses the keys defined in the DBDEF.
DBRED REF=zzzzzz,KEY1=(PKY=#zzzzk80)
  TPFDF uses the keys specified in the application program
```

Selecting an Optimum Block Size

The following scenario addresses selecting an optimum block size.

Problem

A TPFDF file contains subfiles that require about 800 bytes each. Some subfiles are much larger, requiring about 3800 bytes each. Select an optimum block size to balance performance considerations against storage considerations.

For good performance, the number of overflow blocks in a subfile should be as small as possible. It would be a waste of DASD space to allocate 4095-byte prime blocks to each subfile, because most of the blocks would contain only about 800 bytes of data.

However, if you are expecting a large overflow, it is probably better to use one large overflow block than to use many smaller blocks, even if this means that you may waste some DASD space. This is because the same amount of I/O processing is required to access each block, regardless of its size. The fewer blocks there are, the less I/O processing there is to be done.

You should be particularly aware of this where a high performance level is required because heavy I/O demands can reduce performance levels significantly.

If you need to check how fully the blocks are being utilized, use the ZRECP STA command.

Solution

The TPFDF product lets you define different sizes for prime blocks and overflow blocks.

Set the prime block size using the set symbol &SW00WRS. Set the overflow block size using the set symbol &SW00ARS. These set symbols are both specified in the file DSECT.

Set the prime blocks to L2 and the overflow blocks to L4. Both types of block are then of optimum size. This also ensures that the large subfiles have only 1 overflow block. If you had set the same size for overflow blocks and prime blocks, the large subfiles would have had 3 overflow blocks.

Note: You can use the DBDEF macro WRS and ARS parameters to override the defaults in the DSECT.

If you change block sizes after the file has been used, the TPFDF product gradually adjusts to the new block sizes as the file is changed and packed. If you want to change the block sizes immediately, use the ZUDFM OAP command.

DSECT Set Symbols

```
&SW00WRS    SETC   'L2'
&SW00ARS    SETC   'L4'
```

If you do not set &SW00ARS, the TPFDF product uses the prime block size (&SW00WRS) for the overflow blocks as well as for the prime blocks.

DBDEF Statements

```
DBDEF   FILE=zzzzzz,(WRS=L1,ARS=L4)
```

Application Coding

Not applicable.

Reducing the Number of Overflow Blocks

The following scenario addresses reducing the number of overflow blocks.

Problem

A fixed file has been defined as containing 1055-byte prime blocks. This has resulted in a large number of chained overflow blocks. Reduce this by changing the prime block size to 4095.

Solution

Reorganize the file by copying the 1055-byte prime blocks into 4095-byte prime blocks. (The application must not modify the file during the reorganization process.) You can then change the prime block size by resetting the set symbol &SW00WRS in the file DSECT. If necessary, you can also change the overflow block size by resetting &SW00ARS.

When you have done this, update the DSECT with the new &SW00WRS value, and check that the DBDEF table is ready to be loaded. Load the new DBDEF before any application accesses the data.

DSECT Set Symbols

```
&SW00WRS    SETC   'L4'
&SW00ARS    SETC   'L2'
```

Note: If you omit the &SW00ARS setting, the TPFDF product uses the prime block size (&SW00WRS) value for the overflow block size.

DBDEF Statements

```
DBDEF   FILE=zzzzzz,(WRS=L4,ARS=L2)
```

Note: The DBDEF statement overrides any DSECT values.

Application Coding

Not applicable.

Setting Different Sizes for Overflow Blocks

The following scenario addresses setting different sizes for overflow blocks.

Problem

Nearly all (90%) of the subfiles in a file contain about 1200 bytes. The remaining subfiles are much larger, each containing approximately 5000 bytes.

You do not want to allocate 4095-byte overflow blocks because this means that most (90%) of the overflow blocks will have 2895 bytes of wasted DASD space.

Solution

In this example, set the prime block size (&SW00WRS) to L2, the overflow block size (&SW00ARS) to L4, and set &SW00OP1,#BIT5.

The &SW00OP1 indicator tells the TPFDF product to process overflow blocks in an economical way. If the overflow data fits into blocks the same size as the prime blocks (L2), the TPFDF product will use overflow blocks of this size. This is an economical solution for the 90% of the subfiles that contain about 1200 bytes.

If the TPFDF product cannot fit the overflow data into these small blocks, it uses overflow blocks of a size specified by the &SW00ARS set symbol. In this example, the larger subfiles (10%) use 4095-byte blocks (size L4).

DSECT Set Symbols

```
&SW00WRS  SETC   'L2'
&SW00ARS  SETC   'L4'
&SW00OP1  SETC   '.....1..'      (bit 5 set)
```

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

Not applicable.

Packing Files Regularly

The following scenario addresses packing files regularly.

Problem

A TPFDF file is updated frequently in high-performance transactions. It contains 300-byte LRECs in 1055-byte blocks in the subfile. Each LREC that is deleted causes an internal pack operation that creates substantial processing overhead.

Solution

Use the NOPACK parameter when closing the subfile to disable the packing operation. Write a small application to reorganize and pack the file or use the ZUDFM OAP command.

DSECT Set Symbols

Not applicable.

DBDEF Statements

Not applicable.

Application Coding

```
DBCLS REF=zzzzzz,RELEASE,NOPACK
```

Reducing Overflow by Frequent Packing

The following scenario addresses reducing overflow by frequent packing.

Problem

A TPFDF subfile is often read but rarely updated. It contains small LRECs (20 bytes). Because the LRECs are small in comparison with the block size, the subfiles are not usually repacked when an LREC is deleted. As a result, LRECs are eventually scattered over a large number of blocks. This makes reading the subfile unnecessarily slow.

Solution

You could use the PACK parameter every time a subfile is closed. However, this would involve a substantial processing overhead. When closing a subfile, the TPFDF product packs it if any block is less than the threshold specified in global set symbol &SW00PIN.

If the LRECs in the block occupy less than the percentage specified in &SW00PIN, the TPFDF product packs the block up to the level defined in the DBDEF PLI parameter. If the PLI parameter is not specified, the blocks are packed until they are full.

You can override the &SW00PIN set symbol with the DBDEF PIN parameter. If you increase the PIN value to a higher figure, the TPFDF product packs the file more frequently. This reduces the number of overflow blocks in the subfile.

DSECT Set Symbols

No changes are necessary if the changes are made in the DBDEF macro.

DBDEF Statements

```
DBDEF   FILE=zzzzzz,PIN=80
```

Application Coding

Not applicable.

Packing Subfiles after Replacing an LREC

The following scenario addresses packing subfiles after replacing an LREC.

Problem

DBREP macros and `dfrep` functions can result in LRECs occupying less space in a block than that specified in the packing threshold. (The packing threshold is either the TPFDF default for the block type, a value specified in a &SW00PIN DSECT statement, or a value specified with the DBDEF PIN parameter.)

However, the TPFDF product normally packs subfiles only after deleting LRECs, not after replacing LRECs. As a result, the number of blocks in a subfile can become unnecessarily high.

Solution

Set indicator &SW00OP1, #BIT6 in the DSECT to 1. When the TPFDF product closes a subfile, it packs the subfile if the LRECs in any block are below the packing threshold.

Note: Subfile packing results in significant I/O processing. This can affect system performance.

DSECT Set Symbols

```
&SW00OP1  SETC   '......1.'
```

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

Not applicable.

Using New Pool Blocks for Overflow Blocks

The following scenario addresses using new pool blocks for overflow blocks.

Problem

The pack operation can require heavy processing of LRECs between 1 pool block and another. This can be a problem for a file with high integrity requirements if the system fails during the pack operation.

Solution

When packing a subfile, the TPFDF product normally uses the same block chain as the subfile. It releases any pool blocks that it no longer needs.

Set the option in the DSECT (&SW00OP2,#BIT1) to tell the TPFDF product to use new pool blocks for the new compressed chain of overflow blocks.

As the last activity in the pack operation, the TPFDF product updates the prime block's forward chain pointer to the new chain reference. If the pack operation fails, the pointer will still point to the old chain.

Note: This has potential implications for virtual file assist (VFA) performance. The new addresses need new slots in VFA, which will reduce its effectiveness. Because new pool storage is used every time a file is packed, be sure to allocate enough pool storage. You should also run PDU and RECOUP frequently.

DSECT Set Symbols

```
&SW00OP2  SETC   '.1......'
```

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

Not applicable.

Specifying a Lower Packing Limit

The following scenario addresses specifying a lower packing limit.

Problem

A TPFDF file is heavily read and updated. Because there are frequent DBDEL macros, each subfile is packed to 100%.

This is efficient for reading LRECs, but unsatisfactory when an application is adding new LRECs to the middle of a subfile. Because the subfile is so fully packed, when the TPFDF product adds or replaces an LREC, it may have to move some of the LRECs to an overflow block. This creates an unnecessary processing overhead.

Solution

When the TPFDF product packs this subfile, it normally packs each block to 100%. You can use the PLI parameter in the DBDEF to override this by specifying that the TPFDF product should pack blocks only up to a particular limit, not to 100%. Each block then has some space available for subsequent DBADD macros or `dfadd` functions.

In the DBDEF, specify a PLI value between the default packing limit for the subfile and 100%. The default packing density is 50% for L2 blocks, and 75% for L4 blocks.

DSECT Set Symbols

Not applicable.

DBDEF Statements

```
DBDEF   FILE=zzzzzz,PLI=80
```

Application Coding

Not applicable.

Logging Data at Optimum Intervals

The following scenario addresses logging data at optimum intervals.

Problem

An application requires data to be logged to a tape. Writing each modified LREC to tape would create an unacceptably high overhead.

Solution

Specify the TAPE parameter with the DBOPN macro or `dfopn` function for the file. When an LREC is added to the prime block, the TPFDF product does not write it to tape but waits until the block is full. When the prime block is full, the TPFDF product writes the block to the tape specified in the TAPE parameter and reinitializes the prime block.

Notes:

1. The application can read data in the prime block between tape logging operations.
2. B^+Tree files cannot be opened using the TAPE parameter.

DSECT Set Symbols

Not applicable.

DBDEF Statements

Not applicable.

Application Coding

```
DBOPN REF=zzzzzz,HOLD,....,TAPE=xxx
DBADD REF=zzzzzz,NEWLREC=location
```

Maintaining a Log File

The following scenario addresses maintaining a log file.

Problem

You want to log transaction data to a file for online access. The data becomes obsolete after a short time, so you want to keep only recent data. You do not want to write special maintenance and cleanup programs to maintain the log file.

Solution

The TPFDF product lets you control the number of overflow blocks by defining add current files. Each subfile consists of a prime block and a maximum number of overflow (pool) blocks. You specify the maximum number of overflow blocks in the DSECT (the range is 0 to 255).

The TPFDF product adds LRECs normally until it has allocated and filled all the allowed blocks. It then overwrites the oldest LREC with the new LREC that it is adding.

Implement this option by setting the number of blocks in the DSECT set symbol (&SW00NOC) and by setting &SW00OP1,#BIT2.

Caution: Do not set this indicator for normal files. If you do, data will be lost when the wraparound process starts.

DSECT Set Symbols

```
&SW00NOC  SETC   '3'
&SW00OP1  SETC   '..1.....'
```

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

```
DBADD REF=zzzzzz,NEWLREC=location
```

Balancing Updating Speed against Accessing Speed

The following scenario addresses balancing updating speed against accessing speed.

Problem

The application requires a file that can be updated frequently. It also needs frequent access to the latest data in the file. You need to balance the accessing speed against the updating speed.

Fast accessing

You can organize the file so that the latest LRECs are inserted at the start of a subfile, ideally in the prime block. To do this, use a transaction number or date as the key, and specify DOWN organization.

This provides fast access for DBRED macros and `dfred` functions. However, each time you insert a new LREC in the prime block, the TPFDF product has to shuffle LRECs to a new block. This results in poor update performance.

Fast updating

You can organize the file so that new LRECs are added to the end of a subfile. To do this, use a transaction number or date as the key, and specify UP organization.

This provides fast updating, but DBRED macros and `dfred` functions might be slow. In the prime block, the TPFDF product maintains a backward chain to the last overflow block to ensure fast updates. However, as the TPFDF product has to read through the whole chain to get the latest LREC, DBRED macros or `dfred` functions could create a considerable overhead.

Solution

You could solve the problem by following the *fast updating* process and using B^+Tree or block indexing as well. Alternatively, you could specify “pushdown chaining” for the file by setting &SW00OP1,#BIT3.

The TPFDF product then maintains the latest LRECs in the prime block. When the prime block is full the TPFDF product pushes the data into an overflow block and then clears the prime block and makes it available for new LRECs.

Pushdown chaining requires minimal data shuffling on updates. In addition, DBRED macros and `dfred` functions always find the latest information in the prime block of the subfile. The TPFDF product does not need to read through long chains. However, older LRECs are still available in overflow blocks, in case they should be needed.

DSECT Set Symbols

```
&SW00OP1  SETC   '...1....'
```

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

Not applicable.

Getting the Right Amount of Working Storage

The following scenario addresses getting the right amount of working storage.

Problem

An application using the TPFDF product requires working storage. This can be for TPFDF macros or functions, such as building LRECs, or for functions unrelated to the TPFDF product, such as saving application data.

Solution

Use TPFDF T-type files for working storage. The application can then use assembler or C language instructions to manipulate whatever working storage it needs without considering the amount of space available.

Each T-type DSECT describes one particular LREC. TPFDF macros and functions handle T-type DSECTs as if they were TPFDF LRECs.

You can add multiple T-type DSECTs, describing different LRECs, to a common file. This common file must be a W-type file using short-term pool records. The T-type LREC DSECT contains a reference to the underlying W-type file by using the &SW00REF symbol.

There are several advantages of using T-type and underlying W-type files for applications:

- An application can use larger block sizes without the need for reprogramming.
- If program changes lead to a block overflow, the TPFDF product supports the overflow chaining automatically.
- The application can be ported to a system using smaller block sizes without the need for reprogramming. However, the T-type LREC size must not exceed the block size defined for the W-type file.

If a DBOPN macro is coded for the underlying W-type file in the TPFDF UF0B program, the application can directly add, read, and delete T-type LRECs without opening the file. If the DBOPN macro is not coded in UF0B, the application must open the W-type file. For better performance, use the DETAC parameter when opening the W-type file.

Note: If there are storage constraints, do not use the DETAC parameter when opening the W-type file.

When the application performs a DBADD macro or `dfadd` function, the TPFDF product adds 11 bytes (containing the reference name of the T-type file plus the LREC size and primary key) to the beginning of the LREC. This LREC is then inserted into the W-type file.

Note: The TPFDF product is shipped with one predefined W-type file (GW01SR) that is automatically opened in UF0B.

DSECT Set Symbols

```
&SW00REF    SETC   'GW01SR'
.
.
.
.
```

```
zzzzzFL1    DS     CL104          label1 (example)
zzzzzFL2    DS     CL104          label2 (example)
.
.
zzzzzEC0    EQU    *              (example)
```

GW01SR is a default W-type file included in the TPFDF product. Customers can use it or define multiple W-type files to be used with T-type files based on applications or packages.

DBDEF Statements

Not applicable.

Application Coding

```
zzzzzz REG=reg
DBADD REF=zzzzzz,NULLREC==AL2(#zzzzLC0),REG=reg
               The #zzzzLC0 identifies the defined
               length of a particular LREC defined in
               the T-type DSECT.

MVC  zzzzFL1(104),EBW000
               save the EBW000-104 work area in the T-type LREC
MVC  zzzzFL2(104),EBX000
               save the EBX000-104 work area in the T-type LREC
```

Specifying a Display Order for LRECs

The following scenario addresses specifying a display order for LRECs.

Problem

You want to display LRECs from a TPFDF subfile. However, you want to display the LRECs in a different order from their order in the subfile. The DBDSP macro and `dfdsp` function can only display LRECs in the order in which they have been organized in the subfile.

Solution

Create a work file with a DSECT describing the order that you want. To do this, read the TPFDF file and add the data fields into LRECs in a W-type work file. The LRECs of the work file then represent the required display layout.

The final DBDSP macro or `dfdsp` function will display the file and release the work blocks.

If you need to display the data several times, use an R-type file as the intermediate file for storing the data between displays. An R-type file can use short-term or long-term pool blocks.

DSECT Set Symbols

Not applicable.

DBDEF Statements

Not applicable.

Application Coding

```
      DBOPN REF=zzzzzz,REG=reg
      DBOPN REF=zzzzzz,REG=reg,HOLD
      #DO INF
         DBRED REF=zzzzzz

set up new LREC by rearranging the fields from zzzzzz

         DBADD REF=zzzzzz,NEWLREC=location
      #EDO
      DBCLS REF=zzzzzz,RELEASE
      DBDSP REF=zzzzzz,STRIP=...
```

Linking Logically Related Data

The following scenario addresses linking logically related data.

Problem

A file contains different LREC types. The contents of an LREC depend on the LREC ID, as follows:

LREC ID	Contents of LREC
X'80'	Surname and first name
X'90'	Address
X'A0'	Salary details
X'B0'	Telephone number, and so on.

Generally, information is requested by type rather than specifically (for example, by name, address, or telephone information). However, there must be a unique link connecting the address, salary, and telephone information to the name. To make this link, you could use a unique value based on the person's name. However, this would waste space because you would have to repeat the value for all LREC types (90, A0, B0) related to a particular name.

Solution

The TPFDF product supports a unique key generator that returns a unique value for a file. You can store this value in LRECs that are related to each other. The unique key can define relationships in a file, between files, between subsystems, between systems, and between companies.

The unique key is a 4-byte hexadecimal value.

DSECT Set Symbols

```
&SW00OP3 SETC '.......1'
```

To hold the unique key, you must define a 4-byte field in each LREC.

Note: UKY needs an expanded header. Use a sample DSECT (SAMESR, SAMFSR, SAMHSR, or SAMGSR) for UKY.

DBDEF Statements

No changes are necessary if the changes are made in the DSECT macro.

Application Coding

In this example, a subfile is created. Name, address, and salary records are then added:

```
DBOPN REF=zzzzzz,HOLD
DBUKY REF=zzzzzz

                TPFDF puts the unique value in field SW00UKY
```

To set up the LREC (including size and primary key fields):

```
MVC  zzzzUKY,SW00UKY
                Insert the unique key value into the LREC

DBADD REF=zzzzzz,KEY1=(PKY=#zzzzK80,UP)
```

Managing a First-In-First-Out (FIFO) File

The following scenario addresses managing a first-in-first-out (fifo) file.

Problem

A file is treated as a first-in-first-out (FIFO) queue, where LRECs are always deleted from the beginning and added to the end of each subfile. As records are deleted, when the prime block falls below the packing threshold (&SW00PIN), the subfile is packed even though the overflow blocks are still full. This packing operation is not necessary and causes a large amount of unnecessary inputs and outputs (I/Os).

Setting the packing threshold to zero is not adequate because the pack still occurs when the prime block becomes empty. Using the NOPACK parameter on the DBCLS macro prevents the pack, but as records are deleted, the number of empty blocks grows. This increases search time unnecessarily when the subfile is read.

Solution

Set the DELEMPTY parameter on the DBDEF macro to YES, which causes empty blocks to be deleted without a pack operation. Using DELEMPTY=YES with a packing threshold (&SW00PIN) of zero prevents packing while removing empty blocks that increase search time. You can still pack these files by using the ZUDFM OAP command or a ZFCRU command with the pack function specified.

Note: Coding DELEMPTY=YES requires that you define backward chains by setting bit 0 of set symbol &SW00OP1.

DSECT Set Symbols

```
&SW00PIN  SETC   '00'            Packing threshold
&SW00OP1  SETC   '1.......'      Backward chains required
```

DBDEF Statements

```
DBDEF FILE=zzzzzz,DELEMPTY=YES
```

Application Coding

Not applicable.

Using Customer-Format Files

Customer-format files are files that do not follow the same layout as standard-format files (R, W, and T). Customer-format files can be P-type files or non-TPFDF files.

Non-TPFDF files are files that can use the TPFDF recoup and TPFDF capture/restore utility, information and statistics environment (CRUISE), but cannot use other TPFDF macros, commands, and functions to open, close, read, or do other operations on a file.

Notes:

1. Specify ACPDB=NO on a DBDEF statement for a non-TPFDF file.
2. Specify ACPDB=YES on a DBDEF statement for a P-type file.

The following contains examples of customer-format files that can be chain chased using the monitors that are provided with the TPFDF product. A monitor is specified using the CBV parameter on a DBDEF macro statement. If you have customer-format files that cannot be chain chased using the monitors that are provided with the TPFDF product, you can code your own monitor. See “Forward Index Path Parameters” on page 98 for more information about the CBV parameter.

NAB-Type Files with Fixed-Length Items

Figure 70 shows an example of the DBDEF statements to describe a NAB-type file with fixed-length items.

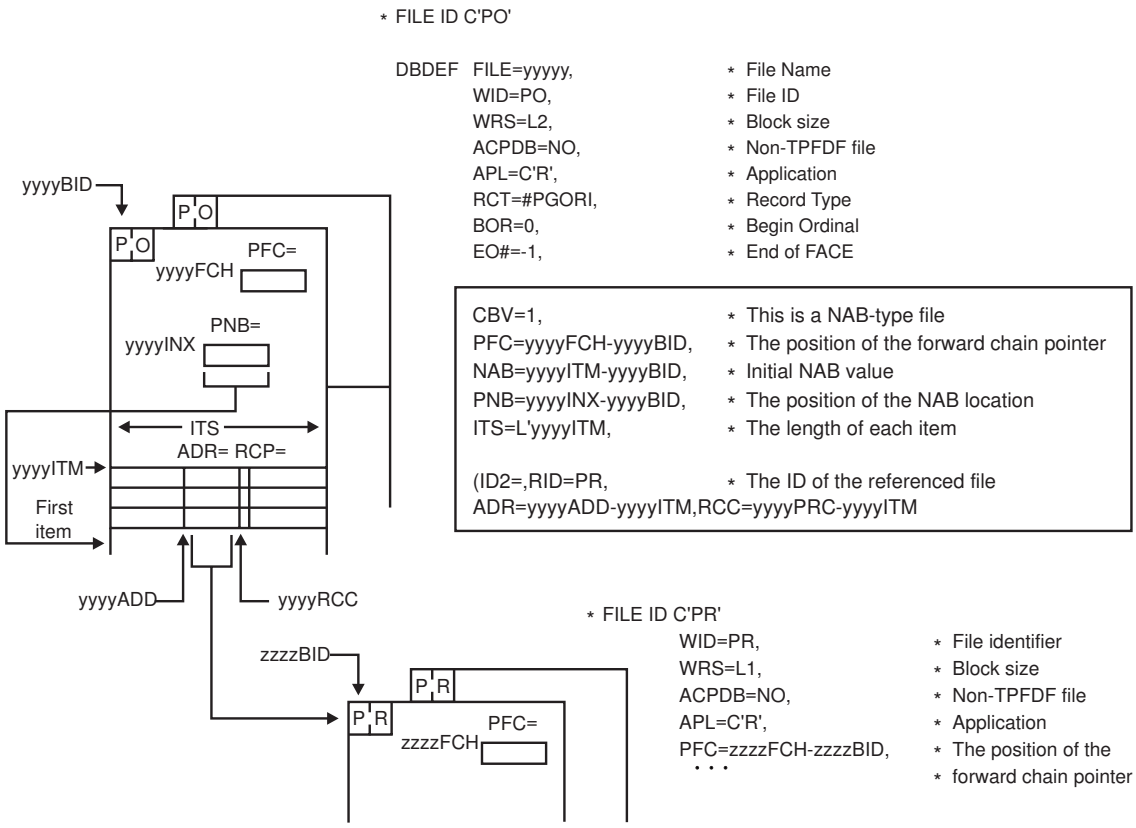


Figure 70. NAB-Type File with Fixed-Length Items (CBV=1)

Notes:

1. This figure shows a file ("PO") which contains a field called yyyyINX.
2. yyyyINX contains a next available byte (NAB) pointer, which defines where the last item in the block is located.
3. The items have a fixed-length, and each contain a reference to a "PR" block.
4. The reference is in an item, so the ID2 parameter is specified.

NAB-Type Files with Variable-Length Items

Figure 71 shows an example of the DBDEF statements to describe a NAB-type file with variable-length items.

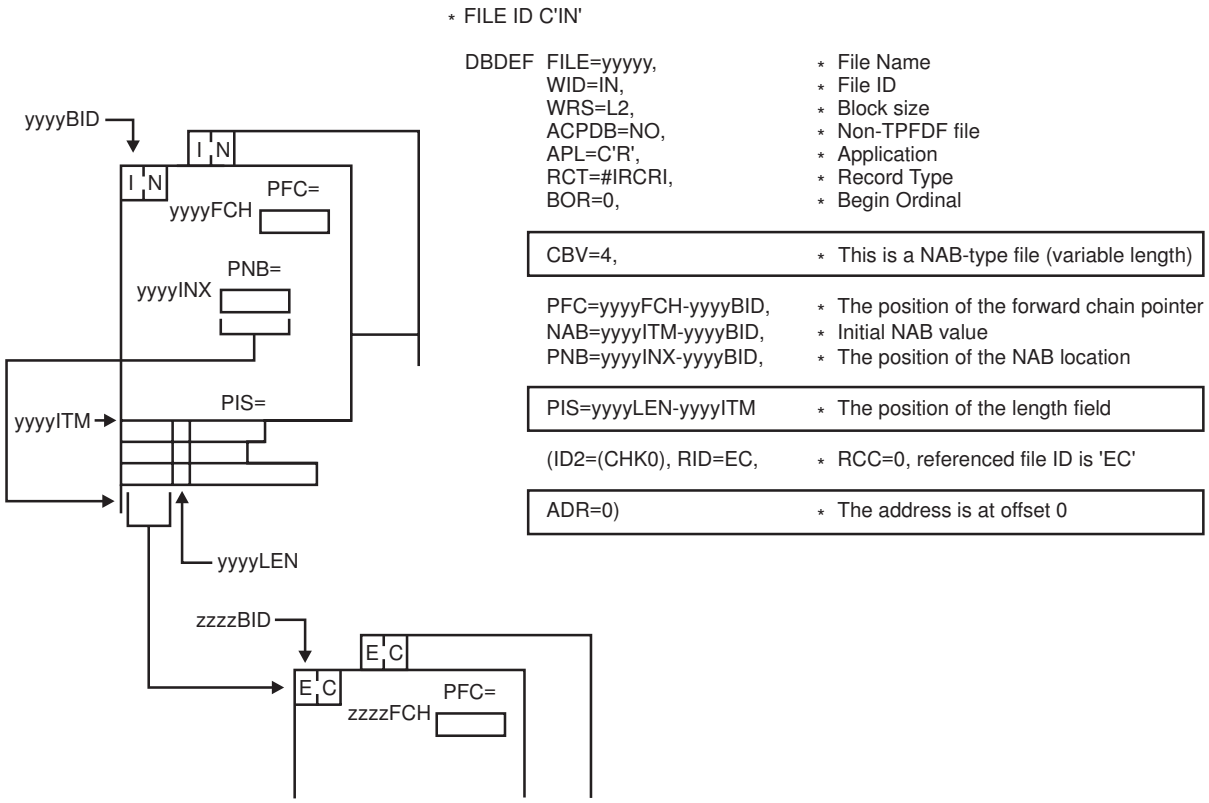


Figure 71. NAB-Type File with Variable-Length Items (CBV=4)

Notes:

1. This example shows a NAB-type file ("IN") which contains variable-length items.
2. Each item contains a reference at location 0.
3. The size of each item is in a halfword field (yyyyLEN) in each item.
4. The reference is in an item, so the ID2 parameter is specified.
5. In this example ID2=(CHK0), so a record code check (RCC) of X'00' is used by TPFDF recoup and the TPFDF capture/restore utility, information and statistics environment (CRUISE). Therefore, the RCP parameter is not needed.

ADD/DEL-Type Files with Fixed-Length Items

ADD/DEL-type files with fixed-length items have 2 NAB pointers. One NAB value identifies the start of the active items, the other identifies the end. Figure 72 shows an example of the DBDEF statements to describe this type of file.

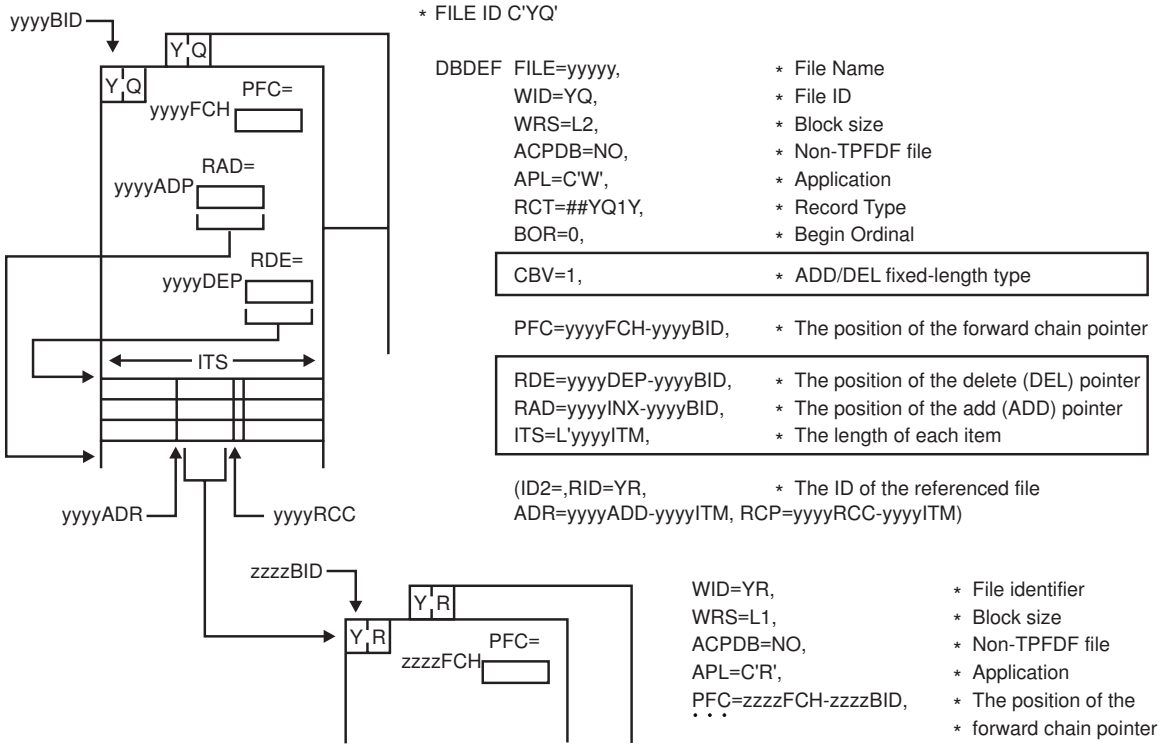


Figure 72. ADD/DEL-Type File with Fixed-Length Items (CBV=1)

Notes:

1. This figure shows a file ("YQ"), which contains 2 fields (yyyyADP, and yyyyDEP).
2. yyyyADP contains an ADD pointer, which defines where the next item should be added.
3. yyyyDEP contains a DEL pointer, which defines the next item to be deleted.
4. The items are fixed-length, and contain a reference to another file ("YR").

ADD/DEL-Type Files with Variable-Length Items

ADD/DEL-type files with variable-length items have 2 NAB pointers. One NAB value identifies the start of the active items, the other identifies the end.

Figure 73 shows an example of the DBDEF statements to describe this type of file.

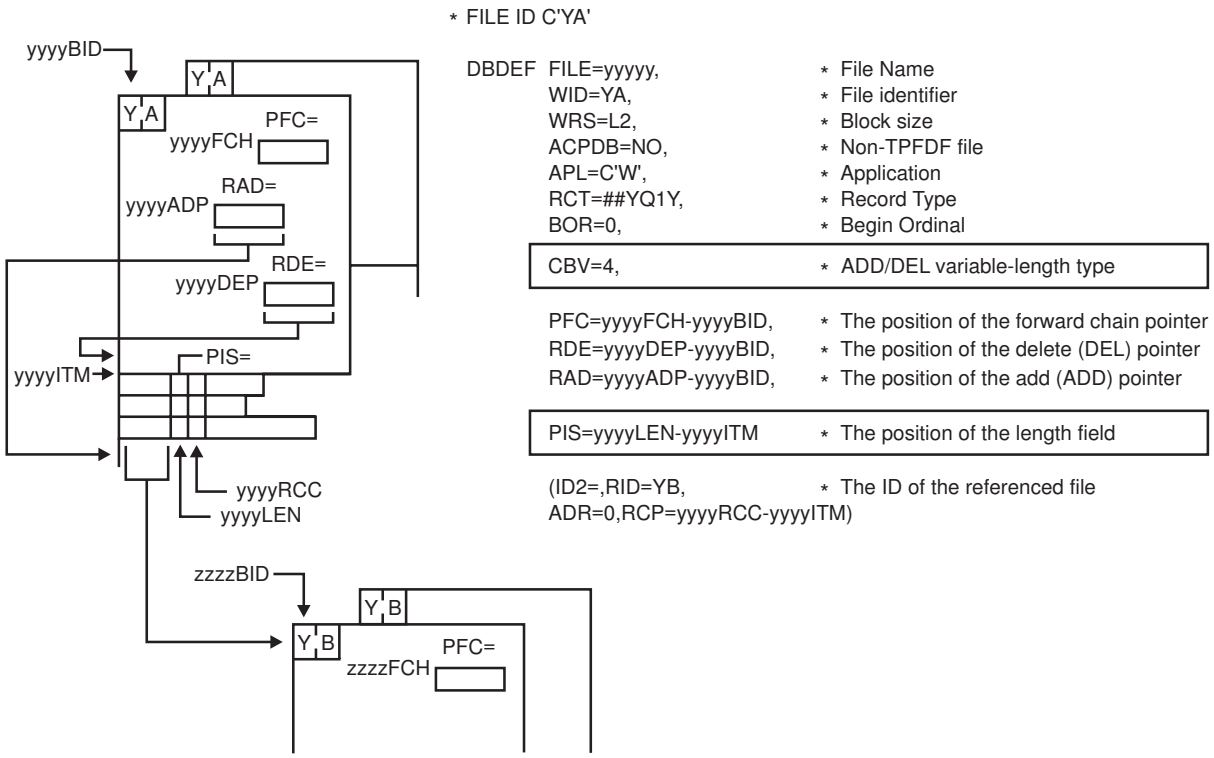


Figure 73. ADD/DEL-Type File with Variable-Length Items (CBV=4)

Notes:

1. This figure shows a file ("YA"), which contains 2 fields (yyyyADP, and yyyyDEP).
2. yyyyADP contains an ADD pointer, which defines where the next item should be added.
3. yyyyDEP contains a DEL pointer, which defines where next item to be deleted.
4. The items are variable-length, and contain a reference to another file ("YR").

CNT Files Using the CNT Parameter

Figure 74 shows an example of the DBDEF statements to describe a CNT file using the CNT parameter.

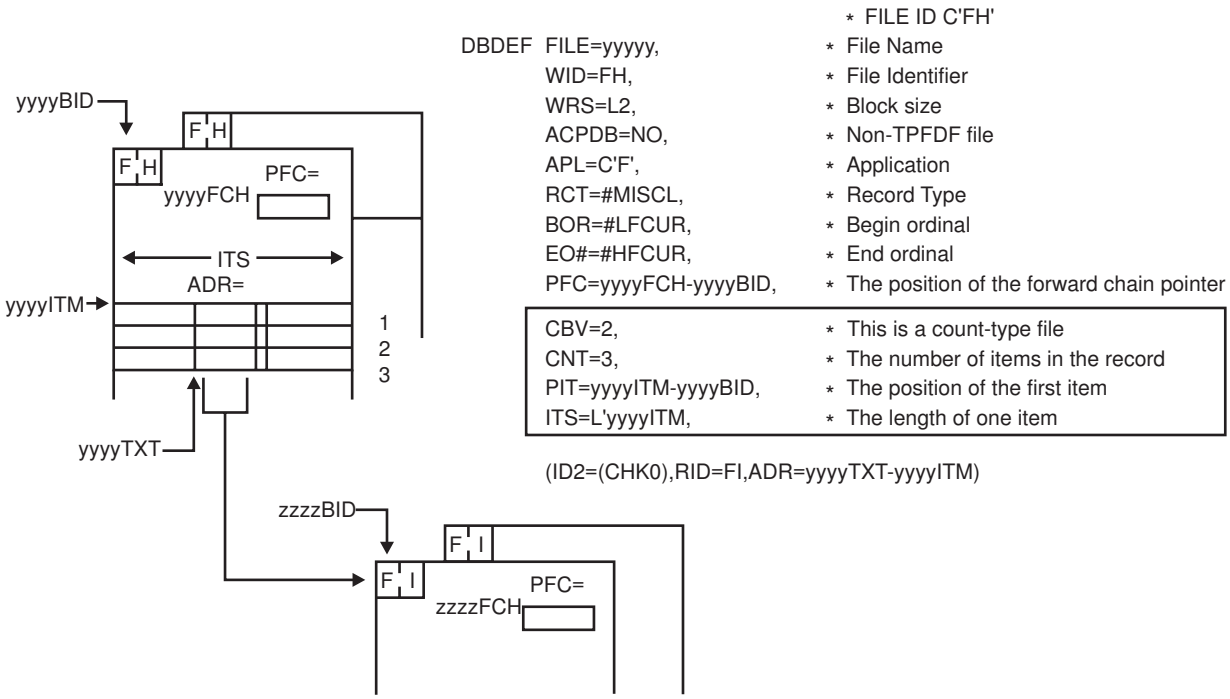


Figure 74. CNT-Type File (CBV=2): Using the CNT Parameter

Notes:

1. This example shows a file "FH".
2. CNT specifies the number of items in the block
3. The items have a fixed-length, and each contain a reference to a "FI" record.
4. In this example ID2=(CHK0), so a record code check (RCC) of X'00' is used by TPFDF recoup and the TPFDF capture/restore utility, information and statistics environment (CRUISE). Therefore, the RCP parameter is not needed.

CNT Files Using the CPT Parameter

Figure 75 shows an example of the DBDEF statements to describe a CNT file using the CPT parameter.

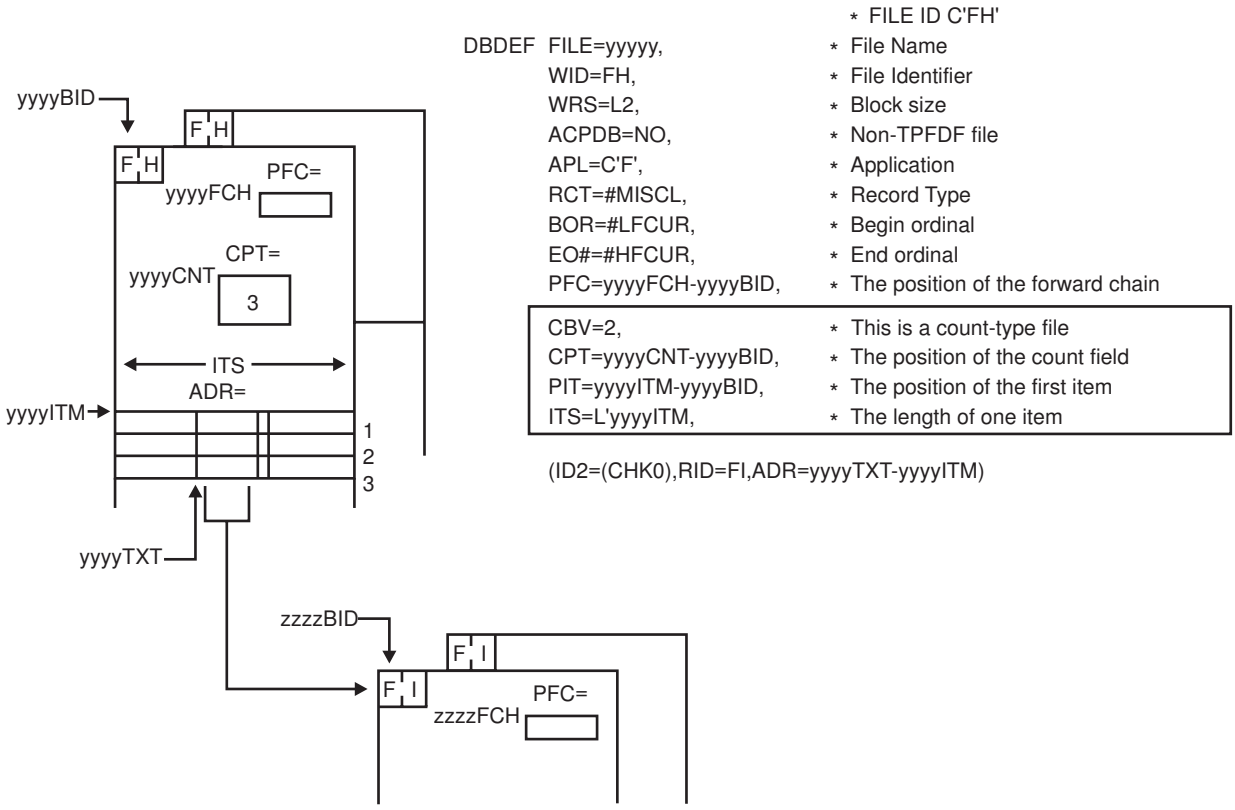


Figure 75. CNT-Type File (CBV=2): Using the CPT Parameter

Notes:

1. This example shows file "FH", which contains a field named yyyyCNT.
2. yyyyCNT contains a value equal to the number of items in the block
3. The items have a fixed-length, and each contain a reference to a "FI" record.
4. In this example ID2=(CHK0), so a record code check (RCC) of X'00' is used by TPFDF recoup and the TPFDF capture/restore utility, information and statistics environment (CRUISE). Therefore, the RCP parameter is not needed.

Files Containing Fixed-Position References

Figure 76 shows the DBDEF statements required to define this type of file structure. This example includes the FVN parameter because the overflow blocks are in a different format from the prime blocks.

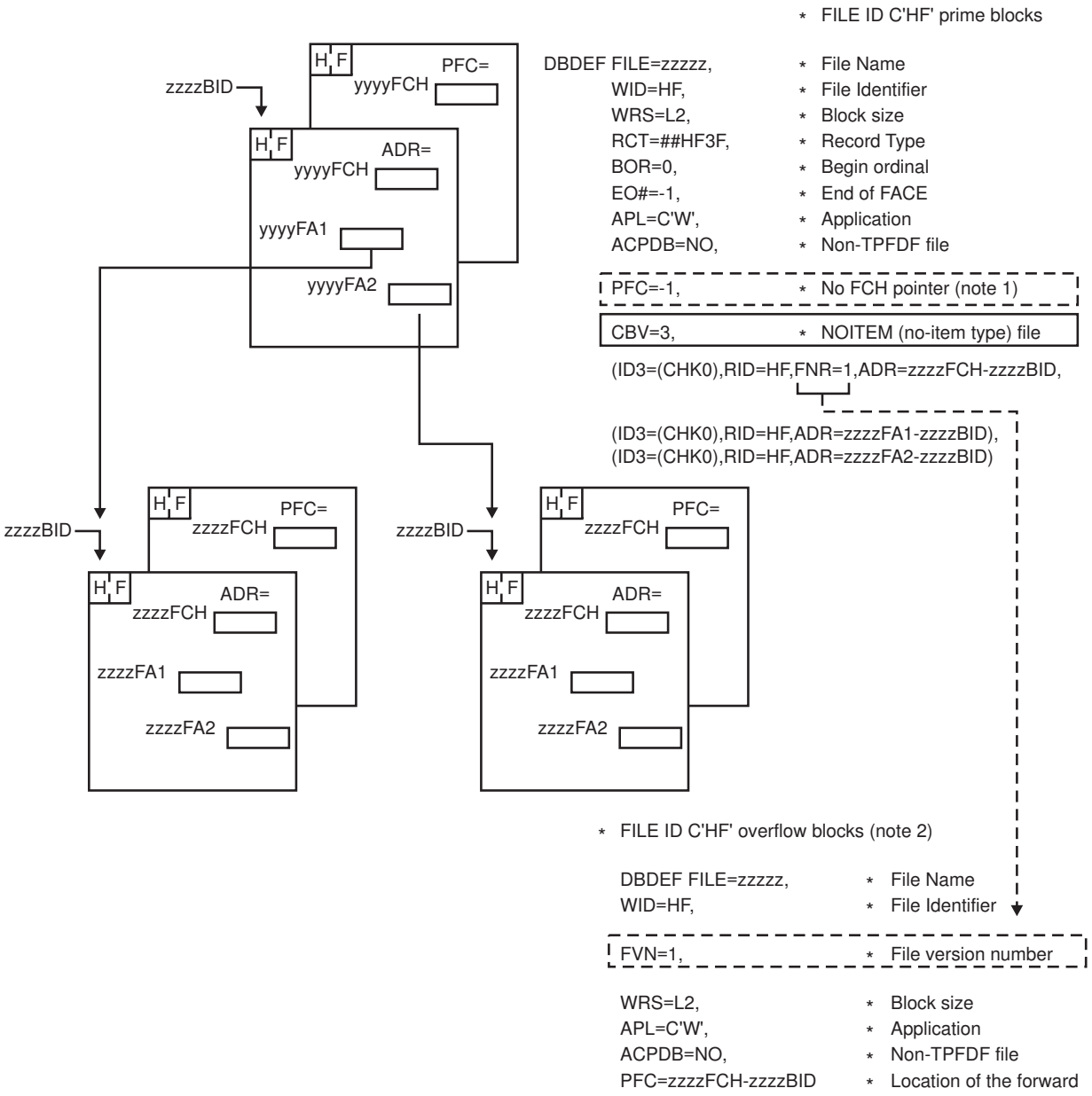


Figure 76. File Containing Only Fixed-Position References (CBV=3)

Notes:

1. The overflow blocks have a different format (no embedded addresses). If the PFC parameter is used, the overflow block would be identical to the prime block. The PFC parameter is set to -1 (no forward chain pointer), but `ADR=zzzzFCH-zzzzBID` defines the position of the pointer.

2. A separate definition is used for the overflow blocks. The FVN parameter is used to identify the definition. The FNR parameter is used in the ID3 statement to refer to that definition.
3. This example shows a file ("HF") which contains 2 references, at fixed locations in the file (zzzzFA1 and zzzzFA2).
4. Both references point to more "HF" files.
5. The "HF" overflow records use standard forward chaining, but contain no embedded references.

Index

Special characters

A

B

C

D

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

File Number: S370/30XX-34
Program Number: 5706-196

Printed in U.S.A.

SH31-0175-09