Dedication

To Mr. Levy, for preparing me for the endless wonders and excitement of mathematics.

WLF
# r-tree® Report Generator

## Reference Guide

## Table of Contents

1. Introduction
   1.1 Installation .................................................. 12
   1.2 r-tree Sample Programs .................................. 12
   1.3 r-tree Utility Programs ................................. 14

2. Basic Concepts
   2.1 Building a program using r-tree .......................... 17
   2.2 c-tree Sample Application ................................. 17
   2.3 Data Object Definition Array ............................. 18
   2.4 ISAM Parameter file ......................................... 19
   2.5 Sample ISAM Parameter file ............................... 20
   2.4 Incremental ISAM Structure ............................... 20
   2.6 Report Script ................................................ 21
   2.7 Call r-tree .................................................. 25
   Include Files .................................................. 25
   Link with the library ......................................... 25
   File Interpreter ................................................. 26
   Compiled File .................................................. 26
   Memory Interpreter ............................................. 26

3. Files and r-tree
   3.1 Master Files .................................................. 27
   3.2 Detail Files .................................................. 27
   3.3 Link Files .................................................... 28
   3.4 Index File Linkage .......................................... 28
   3.5 Variable Length Files ...................................... 29
   3.6 Customer Order Example ................................... 29
   3.7 File Interrelationship Summary ........................... 32

4. Data Object Definition Array (DODA)
   4.1 Header File .................................................. 33
   4.2 How To Setup a DODA ....................................... 34

5. c-tree Parameter Information
   5.1 Incremental ISAM Structure ................................ 37
       IFIL Structure ................................................. 37
# Table of Contents

IIDX Structure .................................................. 38
5.2 ISAM Parameter File ........................................ 39
Data File Description Record ................................. 39
Index File Description Record ............................... 39
Index File Member Record ................................... 40
Sample Parameter File ....................................... 40

## 6. Report Script

6.1 Report Script Sections ..................................... 43
   Some Report Script Ground Rules ......................... 43
Special Terms .................................................. 44
6.2 VIRTUAL .................................................... 45
6.3 SEARCH .................................................... 47
   Secondary Files ............................................ 48
   Multi-segment Keys ....................................... 50
6.4 SELECT .................................................... 51
6.5 SORT ...................................................... 53
6.6 CONTROL .................................................. 55
6.7 ACCUMULATOR ............................................ 56
6.8 DISPLAY ................................................... 58
6.9 IMAGE ..................................................... 60
   Control Footers ........................................... 64
   Arithmetic Conversions .................................. 65
   String Comparisons ..................................... 67
   String Concatenation .................................... 67

## 7. r-tree Operation

7.1 r-tree Library .............................................. 69
7.2 Syntax Checker ............................................. 70
7.3 The Report Function ....................................... 71
   Error Returns ............................................ 71
   Report Script Pointer (rsp) ............................. 72
   Data Object Definition Array Pointer (doda) ........... 72
   Optional Message Pointer (message) ..................... 72
   Optional User Function Pointer (userfn) ............... 72
7.4 Program Structure ......................................... 74
7.5 Report Script Processing Modes ......................... 75
   Text File .................................................. 76
   Memory Image ............................................. 77
   Compiled Script .......................................... 77
   Run-Time Processor ..................................... 78

## 8. Advanced Concepts

8.1 File Hierarchy ............................................. 79
   JOIN TO Relationship .................................... 79
   IS_DETAIL FOR Relationship .............................. 79
   Sample File Hierarchy .................................... 79
   Detail Record Set ......................................... 80
9. Advanced Report Layout

9.1 Image Line ......................................................... 110
9.2 Format Specifications ............................................. 110
   Alphanumeric Formats ........................................... 113
   Dates and Alphanumeric Outputs ................................. 114
   Numeric Formats ................................................ 115
   Raw Dump Format ............................................... 118
   Very Short Format Specifications ............................... 119
9.3 Format Symbol List ................................................. 119
   Field Values ................................................... 120
   Display Logic Variables ........................................ 121
   Physical Device Control Strings ............................... 124
   Output Device Initialization ................................... 125
   Device String Example ......................................... 125
9.4 DISPLAY Variables ................................................ 126
   Display Attribute Variables .................................. 126
   Modifying Display Attributes from Your Application ....... 128

8.4 Record Selection / File Hierarchy Analysis ................. 92
   SELECT ......................................................... 93
   SORT ................................................................ 93
   Example: Selection / Hierarchy Analysis ................. 94

8.5 File Alias ............................................................ 95
   SEARCH Syntax .................................................. 95
   Aliased Data Fields ............................................. 97

8.6 File Interrelationship Alternatives ......................... 98
   Missing Records ................................................ 98
   Detail Record Ordering ......................................... 99

8.7 Built-In Functions ................................................. 100
   Description of Built-In Functions ............................ 100
   SUBSTRING and WILDCARD String Comparisons .......... 101
   Adding Your Own Built-In Functions ......................... 103
   Function Arguments ........................................... 105
   Input Values .................................................... 105
   Multiple Functions ............................................ 105

8.9 Using Set Functions With Virtual Fields ................. 106

8.10 Temporary Files Used By r-tree ......................... 106

8.1 Set Functions ....................................................... 82

8.2 Set Functions ....................................................... 82
   Computational Set Functions .................................. 83
   Boolean Set Functions ......................................... 83
   Set Function Example ......................................... 84
   Using Set Functions With The Primary File ............... 86
   Virtual Field / File Hierarchy Analysis ................. 87
   When Are Virtual Fields Computed? ..................... 88

8.3 Virtual Fields ....................................................... 86
   When Are Virtual Fields Computed? ..................... 88
   Aliased Data Fields ............................................ 87
   SEARCH Syntax .................................................. 95
   Example: Selection / Hierarchy Analysis ................. 94

8.4 Record Selection / File Hierarchy Analysis ................. 92
   SELECT ......................................................... 93
   SORT ................................................................ 93
   Example: Selection / Hierarchy Analysis ................. 94

8.5 File Alias ............................................................ 95
   SEARCH Syntax .................................................. 95
   Aliased Data Fields ............................................. 97

8.6 File Interrelationship Alternatives ......................... 98
   Missing Records ................................................ 98
   Detail Record Ordering ......................................... 99

8.7 Built-In Functions ................................................. 100
   Description of Built-In Functions ............................ 100
   SUBSTRING and WILDCARD String Comparisons .......... 101
   Adding Your Own Built-In Functions ......................... 103
   Function Arguments ........................................... 105
   Input Values .................................................... 105
   Multiple Functions ............................................ 105

8.9 Using Set Functions With Virtual Fields ................. 106

8.10 Temporary Files Used By r-tree ......................... 106

9. Advanced Report Layout

9.1 Image Line ......................................................... 110
9.2 Format Specifications ............................................. 110
   Alphanumeric Formats ........................................... 113
   Dates and Alphanumeric Outputs ................................. 114
   Numeric Formats ................................................ 115
   Raw Dump Format ............................................... 118
   Very Short Format Specifications ............................... 119
9.3 Format Symbol List ................................................. 119
   Field Values ................................................... 120
   Display Logic Variables ........................................ 121
   Physical Device Control Strings ............................... 124
   Output Device Initialization ................................... 125
   Device String Example ......................................... 125
9.4 DISPLAY Variables ................................................ 126
   Display Attribute Variables .................................. 126
   Modifying Display Attributes from Your Application ....... 128

Table of Contents
### Table of Contents

- **Device Strings** .................................................. 128
- **Modifying Device Strings from Your Application** .................. 129
- **Adding New Device String Variables** ................................ 130
- **9.5 Image Line/Format Symbol List Coordination** ................. 132
- **9.6 Debit/Credit Output** ........................................ 133
  - **Single Column Output** ........................................ 133
  - **Debit Credit Columnar Output** ................................ 134
- **9.7 Aged Account Balances** ...................................... 134
- **9.8 Report Chaining** .............................................. 135
- **9.9 PRV/LST: Previous and Last Accumulators** .................... 139
- **9.10 Listing Two Detail Files** .................................... 141
- **9.11 Position Dependent Layout Control** .......................... 142
- **10. r-tree Alphabetical Reference** ................................ 169
- **11. Debugging Hints** .............................................. 169
  - **11.1 Missing Data (Records not selected)** ....................... 169
  - **SEARCH/SELECT** ............................................... 169
  - **JOINS_TO** ..................................................... 170
  - **Record Retrieval** ............................................... 170
  - **Target problems** ............................................... 170
  - **11.2 Header Changes** .......................................... 170
- **A. Reserved Words and Operators** ................................ 171
  - **Report Script Section Headings** ................................ 171
  - **Operators** ..................................................... 171
    - **Computational** ............................................... 172
    - **Relational** ................................................... 172
    - **Boolean** ..................................................... 172
  - **Field Types** ..................................................... 172
    - **Virtual Fields** ............................................... 172
    - **Accumulators** ............................................... 173
    - **Sort Key Segments** .......................................... 173
  - **Status Variables and Symbolic Constants** ........................ 173
    - **Status Variables** ............................................ 173
    - **Symbolic Constants Value** ................................... 174
    - **Display Attribute Variables** ................................ 175
    - **Display Logic Variables** ................................... 175
    - **Device Strings** .............................................. 176
    - **Built-In Function Names** .................................... 177
    - **Set Functions** ............................................... 178
    - **Syntactic Elements** .......................................... 178
- **B. r-tree Report Script Syntax** .................................. 179
- **C. r-tree Errors** .................................................. 183

---

r-tree® Report Generator
r-tree Error Codes ................................................................. 191
Extended c-tree Error Codes ............................................... 194
This page intentionally left blank.
1. Introduction

One of the more difficult tasks in creating an application program managing a set of data files is building reports to fit customers’ needs. The task can be tedious and time consuming. In addition, the reports may satisfy one customer, but leave another looking for additional ways to extract information from the files. Building applications using the FairCom c-tree® Plus File Handler and the r-tree® Report Generator can solve these problems.

r-tree is a report generation system based on c-tree Plus data files and indices, a simple set of control parameters describing the files, and a report script. Once the control parameters (called a Data Object Definition Array, or DODA) have been created you simply call the r-tree report function, and provide it a script created with any simple text editor.

With the DODA, you simply create report scripts instead of building complex programs to create reports from the files. In addition to saving programmer time, it is possible to allow your customers to create their own scripts, making the application more flexible and increasing its value.

The report script can specify:

- how multiple c-tree Plus data files will be searched;
- what groups of records will be selected for the report;
- how, if desired, the selected records will be sorted;
- the computation of virtual fields;
- the number of control break levels;
- special report control options; and
- the actual report layout.

r-tree offers two ways of providing Report scripts to your customers, giving you the flexibility of choosing between protecting proprietary information and providing customers with guidelines for developing their own scripts. Scripts can be provided as:

- original text, which may be edited by your customers to create their own custom reports; or
- as compiled report scripts which cannot be changed by the customer, also permitting faster start up of the report generator.
1.1 Installation

*r-tree®* installation steps are located in the *r-tree* section of the FairCom Installation Guide. After completing the installation steps, please proceed to the next section (Section 1.2) for a discussion on the *r-tree* sample programs.

1.2 r-tree Sample Programs

The *r-tree* distribution diskettes contain many sample programs to assist with learning *r-tree*. All of the programs assume the desired report script is in the same directory as the executable. In addition, the sample programs that utilize parameter files (rtsamplp and optionally rtest) must have the parameter file in the current directory with the sample program.

- **RTSAMPLI** - simple ISAM incremental structure database example
  
  `rtsampli <create flag>`

  RTSAMPLI utilizes the incremental file structures defined in rtsampli.c for its file definitions. This program is an extension of the isam1.c sample from c-tree® Plus and uses the report script ISAM.REP.

  The `<create flag>` parameter is used to create the data and index file. If the files do not exist, the create flag will be required during the first execution of the program and can be any command line argument, its presence, not its value, signifies that the files are to be created.

  When rtsampli is executed, select the ‘V’ option to generate a report which is routed to the screen.

- **RTSAMPLP** - simple ISAM parameter file database example
  
  `rtsamplp <create flag>`

  RTSAMPLP uses the parameter file isamp.p supplied with *r-tree*. This program is an extension of the isam3.c sample from c-tree Plus. It uses the report script ISAM.REP.

  The `<create flag>` parameter is used to create the data and index file. If the files do not exist, the create flag will be required during the first execution of the program and can be any command line argument, its presence, not its value, signifies that the files are to be created.

  When rtsamplp is executed, select the ‘V’ option to generate a report which is routed to the screen. If isamp.p is not in the working directory with the RTSAMPLP executable, an error 102 will result.
• **RTTEST** - sample report program for the customer order example

  \[ \text{rttest } <\text{report script file name}> \]

  RTTEST is based on the customer order example introduced in Chapter 3 of this guide. There are seven sample report scripts named COS.0 through COS.6. All of the scripts route the output to the screen by default. RTTEST uses the interpretive file interface (rtintr.c): it processes a script stored in a text file.

  For example,

  \[ \text{rttest cos.0} \]

  parses the report script COS.0 on the fly, and produces the actual report output.

  Before running rttest, be sure that the files required by rttest (CUSTMAST.DAT, CUSTMAST.IDX, ...), are in the directory with the rttest executable. The sample report scripts (COS.0, COS.1, ..., COS.6) must also be in the same directory as the executable sample program. By default, rttest uses Incremental ISAM file definitions. By activating the \#define PARMFILE in rttest.c, this sample program can be converted to using parameter file definitions. If PARMFILE is defined, the parameter file rttest.p must be present, or an error 102 (could not open parameter file) will occur.

  Note, the rttest data files (CUSTMAST.DAT ... ) are variable length files, requiring VARLDATA to be defined in the c-tree® library. In addition the data files in the root r-tree® directory are in LOW/HIGH format. If rttest is to be executed on a HIGH/LOW machine, be sure to use the data and index files from the hghlow directory.

• **RTDEMO** - sample r-tree application based on CTDEMO

  \[ \text{win rtdemo} \]

  RTDEMO is available in the Windows subdirectories only. This example is based on the c-tree Plus Windows application CTDEMO.

• **RTDRVR** - c-tree Plus sample program that automatically opens any specified data file in the report script passed on the command line.

  The r-tree driver program will simplify the use of r-tree by removing the requirements for C programming. rtdvr accepts multiple r-tree script files on the command line (each script separated by a space; i.e., cos.1 cos.2 cos.3). The rtdvr program will execute each script independently in the order specified on the command line. Each script is initially opened to determine the file(s) that need to be opened. rtdvr then opens the proper files and executes the script. Upon completion, any opened files are closed, and any additional specified scripts are processed.
For rtdrvr to be able to work properly the following requirements must be met:

1. The ISAM file definitions (required by r-tree®) must be Incremental File Structures (IFIL, ISEG and IIDX structures). Parameter files are not supported.

2. All data files specified in an r-tree script must have been created with RESOURCE’s defined (the default) in the c-tree® Plus ctoptn.h. When a file is created with the RESOURCE define enabled, the c-tree Plus IFIL information is written into the data file automatically unless the special file mode, DISABLERES, is specified at file creation time. See the c-tree Plus Programmer’s Reference Guide for more information.

3. All data files specified in an r-tree script must have a DODA present (see PUT-DODA() in the c-tree Plus Function Reference Guide).

The sample programs rtsampli and rttest previously mentioned can be used with rtdrvr. For example, to execute rtdrvr on the rtsampli data file (invent.dat) simply pass the r-tree script (isam.rep) on the command line as follows:

```
rtdrvr isam.rep
```

To experience the power of passing multiple scripts, pass multiple cos.? scripts from rttest as follows: rtdrvr cos.1 cos.2 cos.3

Experiment with the power of rtdrvr on any existing r-tree scripts that meet the above requirements.

### 1.3 r-tree Utility Programs

The r-tree distribution diskettes contain the following utility programs:

- **RTCTST** - sample report script compiler
  ```
  rtctst <report script file name> [optional compiled script name]
  ```
  RTCTST is based on the customer order example, rttest. RTCTST is linked with the compiler interface (rtcmp1.c). It processes a report script stored in a text file and produces a compiled script. If no compiled script name is provided on the command line, it places the compiled script in the file named “rp.cmp.”

For example,

```
rtctst cos.0 coscmp.0
```

generates a compiled report script in the file coscmp.0.
• **RTRTST** - sample compiled script processor

  `rtrtst <name of compiled script file>`

  RTRTST processes compiled scripts. RTRTST is linked with the run-time script interface (rtrtim.c). Its input is the output of RTCTST. Continuing with the previous example

  ```
  rtrtst coscmp.0
  ```

  produces the same output as

  ```
  rttest cos.0.
  ```

  since coscmp.0 is just the compiled form of cos.0.

• **RTSYNT** - report script syntax checker

  `rtsynt <report script file name>`

  RTSYNT checks the syntax of any report script. It does not use the DODA (Data Object Definition Array), and hence it cannot check for undefined symbols or bad file names.

  For example,

  ```
  rtsynt cos.0
  ```

  tests the syntax of the report script contained in cos.0.
This page intentionally left blank.
2. Basic Concepts

To introduce the basic r-tree® concepts we are going to use a simple example. We will use the sample database program found in the c-tree® Plus Programmer’s Reference Guide to illustrate the use of r-tree. This is a very simple database, with just one data file and one index. More advanced examples will be used later in the manual.

This sample program is included on the distribution diskette in two versions, rtsampli.c (IFIL based) and rtsamplp.c (Parameter file based) along with the supporting report script isam.rep. rtsamplp is based on the c-tree Plus parameter file example program isam3.c and requires the parameter file isam.p. The rtsampli sample program is based on the c-tree Plus incremental structure sample program isam1.c which uses the same record structure as isam3.c. (isam1.c and isam3.c are discussed in the c-tree Plus Programmer’s Reference Guide). The only difference between these two sample programs is the method specifying the file definitions.

2.1 Building a program using r-tree

To build a program that creates a report using r-tree, the following steps are required:

- We must have an application that uses the c-tree Plus file handler. In addition, we must be able to open the files and indices as Index Sequential Access Method (ISAM) files, either with Incremental ISAM structures (like rtsampli.c) or with a parameter file (like rtsamplp.c).
- Build a Data Object Definition Array (DODA) that describes the fields in the data records for each file.
- Modify the Incremental ISAM structure or the ISAM Parameter file to include information to associate the data files with the appropriate DODA.
- Create a Report Script, which describes how to prepare and produce a report based on the data files.
- Call the r-tree report () function from within the program, and link the appropriate r-tree routines.

2.2 c-tree Sample Application

As mentioned earlier, we will base our example on the isam1.c and isam3.c sample programs, discussed in the c-tree Plus manual. These programs are very simple to build and maintain. Both programs use the same inventory data file. There is one index, the
The structure for the file is illustrated in Figure 2-1.

```
struct INVENTORY {
    COUNT delete_flag;
    TEXT item[32];
    TEXT descrip[51];
    double quantity;
    double cost;
    TEXT padding[213];
} invent,old_invent;
```

Figure 2-1.

This structure is used as the data buffer for the inventory file.

### 2.3 Data Object Definition Array

The Data Object Definition Array (DODA) is used to assign a symbolic name to each of the fields in the data record, as well as to define the field location, type, and length. You will use the symbolic name from the DODA to refer to the record fields in the Report Script.

You must add the following include statement at the top of the program module that will access r-tree®:

```c
#include “rtdoda.h”
```

This contains a typedef for the structure DATOBJ, which is used to build the DODA, as well as some symbolic constants that you will use to describe the field types.

Figure 2-2 contains the DODA for our example. Make sure that it is declared AFTER the data buffer structure.

```
DATOBJ invodada1 = C
   C”Deeflag”. (pTEXT)&invent.delete_flag, RTINT2),
   C”Item”. invent.item, RTSTRING, 25),
   C”Description” invent.descrip, RTSTRING, 50),
   C”Location”. invent.location, RTSTRING, 25),
   C”Quantity”. (pTEXT)&invent.quantity, RTDFLOW72),
   C”Cost”. (pTEXT)&invent.cost, RTDFLOW72),
   C”Buffer”. invent.padding, RTSTRING, 213)
   C””,”.0,0,-13
```

Figure 2-2.
Each data field has an element in the DODA structure. Each element has 4 fields:

- **symbolic name** - this is the name used to refer to this field in the report script. Each name must be unique, and must be enclosed in quotes.
- **address** - the address of the data field in the data buffer. In the case of an array (such as a character array or string) use just the field name. For other fields, such as integers, you must place an ampersand (&) in front of the name to avoid indirection warnings during compile.

Note, since each field must have a unique address in the DODA, only character arrays are supported. If a multi-dimensional character array or an array of numerics must be used, each element of the array must have its own unique entry in the DODA.

- **field type** - use one of the following symbolic constants to define the type of field:

<table>
<thead>
<tr>
<th>DODA Constant</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCHAR</td>
<td>one-byte signed integer</td>
</tr>
<tr>
<td>RTCHARU</td>
<td>one-byte unsigned integer</td>
</tr>
<tr>
<td>RTINT2</td>
<td>two-byte signed integer</td>
</tr>
<tr>
<td>RTINT2U</td>
<td>two-byte unsigned integer</td>
</tr>
<tr>
<td>RTINT4</td>
<td>four-byte signed integer</td>
</tr>
<tr>
<td>RTINT4U</td>
<td>four-byte unsigned integer</td>
</tr>
<tr>
<td>RTDATE</td>
<td>four-byte signed integer containing date in four-byte computational form</td>
</tr>
<tr>
<td>RTMONEY</td>
<td>four-byte signed integer with implied decimal point</td>
</tr>
<tr>
<td>RTSFLOAT</td>
<td>single-precision floating point</td>
</tr>
<tr>
<td>RTDFLOAT</td>
<td>double-precision floating point</td>
</tr>
<tr>
<td>RTSTRING</td>
<td>character array</td>
</tr>
</tbody>
</table>

- **field length in bytes** - only required for strings. You can omit it for other field types.

### 2.5 ISAM Parameter file

Information must be added to the ISAM parameter file for r-tree®.

**Data File Description Record**

Two parameters are added to the Data File Description Record. Field 7 will be the symbolic name (from the DODA) of the first field in the data record, and field 8 will be the symbolic name of the last field in the data record.
One parameter will be added to each Index File Description Record. Field 12 will be the symbolic name for the index file.

Similar to the Index File Description Record, we will add the symbolic name for the index file as field 8.

Sample ISAM Parameter file

For our example, we have added the necessary fields to the parameter file isamp as shown in Figure 2-3.

```
10 1 4 1
0  invent.dat 128 4096 1 1 Delflag Buffer
1 invent.idx 25 0 0 0 4096 1 0 32 1 Itemidx
2 25 2
```

Figure 2-3.

Note that we have added Delflag and Buffer to the second line, as the first and last fields in the DODA for this data file. We have also added Itemidx to the third line, as a symbolic name representing this index.

2.4 Incremental ISAM Structure

The Incremental ISAM IFIL and IIDX structures require the following information:

**IFIL Structure**

The 10th element (TEXT *rfstfld) of the IFIL structure is the symbolic name (from the DODA) of the first field in the data record, and the 11th element (TEXT *rlstfld) of the IFIL structure will be the symbolic name of the last field in the data record.

**IIDX Structure**

The 8th element (TEXT *ridxnam) of the IIDX structure will be the symbolic name for the index file.
The structures in Figure 2-4 are from rtsampl.c:

```c
/* INCREMENTAL ISAM STRUCTURES - The following structures are used in place
   * of the ISAM parameter file. ISAM is the key segment structure. IIDX is
   * the index structure. IFIL is the main ISAM structure.
   * The values used here match those found in the isam.p parameter file. */
ISEG inv_seq = {
    2, 25, 2
};
IIDX inv_idx = {
    25,  /* key length */
    0,  /* key type */
    0,  /* dup off */
    0,  /* null off */
    32, /* empty char */
    1,  /* number of key segments */
    &inv_seq,  /* pointer to segment array */
    "itemIdx",  /* index name ("Idx" is always assumed) */
};
IFIL inv_dat = {
    "invent",  /* data file name ("dat" is always assumed) */
    0,  /* data file number */
    128, /* data record length */
    4096, /* data extension size */
    1,  /* data file mode */
    1,  /* number of indices */
    4096, /* index extension size */
    1,  /* index file mode */
    &inv_idx,  /* pointer to index array */
    "Nefiling", /* r-tree info (not used here) */
    "Buffer", /* r-tree info (not used here) */
};
```

**Figure 2-4.**

### 2.6 Report Script

The Report Script illustrated is a very simple one. We use it to introduce the basic concepts involved in creating a Report Script. There are many variations and features which are explained in detail in later sections, along with more complex examples.
Perhaps we want to create a report as illustrated in Figure 2-5, listing all items with a quantity greater than 0, and listed in order of the on-hand value (which is the product of the unit cost and the quantity on hand). We also want a total of the on-hand value.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Quantity</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>IBM AT Computer System</td>
<td>1.00</td>
<td>10.00</td>
</tr>
<tr>
<td>Ventura</td>
<td>Xerox Ventura Publisher</td>
<td>5.00</td>
<td>100.00</td>
</tr>
<tr>
<td>HDD-180MB</td>
<td>Hard Disk Drive/180 MB</td>
<td>2.00</td>
<td>10000.00</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>Total Value</strong></td>
<td></td>
<td>10110.00</td>
</tr>
</tbody>
</table>

*Figure 2-5.*

This very simple report (Figure 2-5) can be created by building a Report Script (Figure 2-6) that will work with the DODA already defined. The Report Script is divided into several sections which are used to define the fields and index to use, and to lay out the report.

*Figure 2-6* contains the Report Script creating the report we have shown above. We then explain each of the features used.

```
START
VIRTUAL
Value DFLT @ Cost = Quantity
SEARCH FILE "invent.dat" USING KEY ItemIdx
SELECT Quantity > 0
SORT USING KEY
NO HDR Value
@ACUMULATOR
Total SUM Value
DISPLAY DEVICE 1
IMAGE
PAGE_HDR =#00000#
SYN_DATE
PAGE_NO #Item
+Description Quantity Value

BODY
```
Every Report Script must start with the keyword START and end with the keyword EXIT.

VIRTUAL

This section is used to define a symbolic name that represents a value that can be a variable from your program, a computed value using fields from your file, and so forth. In our example we are defining a double float variable named Value, which will be the product of the Cost and Quantity fields for each line of the report. The syntax of a VIRTUAL field definition is provided in detail in later sections of this manual.

SEARCH

The SEARCH section defines the data files and indices to use. Here we use the FILE keyword to indicate that we are going to use the file invent.dat, and that this file will be accessed by the index Itemidx. Recall that Itemidx is the symbolic name defined in the DODA. If we do not include the USING_KEY keyword, the file will be processed in physical order.

SELECT

The SELECT section is used to select a subset from the data file. In this case we select only those records with a Quantity field with a value greater than 0.

SORT

NO_MOD

Since we want this report to list the records in an order other than the index order, we use a SORT section. In this case we are going to sort on the Value field, which is the virtual field that is calculated for each record. For each sort key we must define the sorting mode. Later on we will define a variety of sort modes, but for most cases you will use NO_MOD for ascending order, DSC_NO_MOD for descending order, UPPER for ascending order with upper-case translation, and DSC_UPPER for descending order with upper-case translation.

ACCUMULATOR

This section allows the report designer to direct r-tree® to automatically maintain accumulators. In our example we are defining a virtual field Total as the sum of the Value fields for each record selected. The accumulator type can be:

<table>
<thead>
<tr>
<th>atype</th>
<th>Tally</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRQ</td>
<td>Frequency count</td>
<td>long integer</td>
</tr>
<tr>
<td>SUM</td>
<td>Summation</td>
<td>double float</td>
</tr>
</tbody>
</table>

Figure 2-6.
This section allows the overall nature of the report to be controlled from the script itself. We can control things like the page length and width, the number of test pages to print (for forms alignment), what character to use as the currency mark, and more. In our example we are sending the report to printer 1, which in a DOS environment would be LPT1. We could also send the report to the screen, another printer, or a disk file.

The IMAGE section is the key to the report. It defines the actual output of the report itself. There are a number of subsections to the IMAGE section, as defined in a later chapter. In our example we are using the PAGE_HDR, BODY, and REPORT_FTR subsections.

The PAGE_HDR subsection defines the information to be printed at the top of each page. Our script defines three lines to be printed in the page header. Every line that is to print on the report must start with a `+' symbol. Within each line you can define fields and literals. Fields start with a `@' character, and are followed by a format string. In the first line of the page header we have two fields and two literals.

Following the first format symbol list we have two additional output lines that contain only output literals for the column headings. Note that both lines start with the `+' symbol.

The BODY subsection is printed repeatedly for each selected record. There are two lines, the output line that shows the position and the length of the fields to be printed, and the format symbol list that defines the values to be printed. Note that the output line starts with the `+' symbol. The values listed are the symbolic names that we defined for the data fields in the DODA above, or as VIRTUAL fields.

This section defines the information to be printed at the end of the report. The format parallels the other sections, with output lines followed by a format symbol list. In our example we have printed the value Total, which we defined in the ACCUMULATOR section.
2.7 Call r-tree

We have defined a DODA, modified the ISAM parameter file, and created a Report Script. How do we get the actual report to print? This is actually the easiest step of all. At the place that you want to start the report, simply add a function call in the following format:

```c
retval = report(rsp, doda, message, userfn)
```

- **retval**: The report function will return a COUNT value of 0 if it is successful, or an error number if not. You should declare a variable to capture this error value.
- **rsp**: This is a pointer to the report script, which can either be a file name or a buffer in memory that contains the report script.
- **doda**: This is a pointer to the DODA in memory.
- **message**: This is a pointer to a message that will be displayed when control is passed to r-tree®. If you do not want to use this, pass a null text pointer (NULL).
- **userfn**: The last parameter is a pointer to a function that you define to which control will be passed after each major r-tree iteration. It allows you to perform special operations under certain conditions. If you aren't using this, pass a null text pointer.

**Include Files**

The following r-tree specific #includes must be added to the C program module (.c) containing the `report()` function call:

- `#include "rtpars.h"
- `#include "rtstrc.h"

If `ctNOGLOBAL` is defined in `ctoptn.h`, `rtgvar.h` must be included in the C program module with `report()`. In addition, the r-tree function `rtivar()` must be called to initialize r-tree’s global structure. `ctNOGLOBAL` is a c-tree Plus option for moving all global variables into an allocated structure. See the c-tree Plus Programmer’s Reference Guide for additional information.

**Link with the library**

In most operating environments the r-tree library will be `rtalib.lib`, or something similar. You will link this library with your application. In addition to this you will need to link in a module that defines the method that r-tree will use to obtain the script.
**File Interpreter**

If you want to use scripts that are in a file, and you want r-tree® to interpret the script at run time, you will use the **Interpreter** mode. This is the most common way of using r-tree. Link the module *rtintr.obj* with your application. The *rsp* parameter in the *report* function call will be the name of the report script file to be used.

**Compiled File**

It is possible to “compile” a script file so that it cannot be changed by the end user. Compiling is described in the "r-tree Operation" chapter. If you want to use a compiled script file, link your application with *rtcmpl.obj*. As before, the *rsp* parameter is the name of the compiled report script file.

**Memory Interpreter**

You can create your script in a buffer in memory instead of using a text file. Link your application with *rtmmyr.obj*. The *rsp* parameter will be a pointer to the null terminated memory buffer that contains the script.
3. Files and r-tree

c-tree® Plus and r-tree® cannot be considered as true database products; they are high performance kernels for building database applications. However, to the extent that c-tree Plus and r-tree embrace any conceptual database model, we would identify with the relational model. A c-tree Plus data file roughly corresponds to a relation. Each record contains the equivalent of a row in a table used to graphically depict a relation.

As in the relational model, c-tree Plus/r-tree do not support repeating groups. Consider a data record which contains information pertaining to a customer order. Using a repeating group, we would store each item on the order in the customer order record. In essence, the field "item" would be repeated within the customer order record for each item ordered by the customer. With c-tree Plus/r-tree we would use two separate files to represent the customer order and the items on the order. Each order spawns a customer order record, and each item on an order spawns a record in the item file.

We can identify three types of files for use with r-tree:
- master files;
- detail files; and
- link files.

Data files are interrelated through the use of c-tree Plus index files. Unlike the relational model, we assume that the programmer knows which indices relate to which data files. In other words, we do not assume data independence.

3.1 Master Files

Each record in a master file represents a self-contained set of independent information. There are no explicit references to other files. For example, a customer master record could contain fields for: customer number, name, company, address, telephone number, city, state and zip code. This file is meaningful without any reference to other files. In most instances, a master file will have at least one unique index; that is, each record will have a key value which is not found in any other record of the master file.

3.2 Detail Files

Each record in a detail file represents one instance of a set of information which may be repeated one or more times for a particular record in an associated file (sometimes called the parent file). The records in a detail file have no independent existence. If a record in the associated file is deleted, then the detail records which relate to it are effectively deleted. An associated file may be a master file, another detail file, or a link file.


**3.3 Link Files**

**r-tree** assumes an index file will be used to relate an associated file and its detail file. The index must allow a partial key search (based on information in the associated file or derived from the associated file) to retrieve all associated records in the detail file. By partial key search, we mean **FRSSET** and **NXTSET** from c-tree**®** Plus will find the associated records. (We discuss the use of indices and the pertinent c-tree Plus functions below.)

The following topic does not pertain to c-tree Plus. The index used to relate a detail file to its associated file cannot be a right-to-left scan variety (right-to-left scans supported only with c-tree 4.3 and prior); i.e., the **c-tree key type parameter cannot have a value of one (1).**

In a customer order system, we are likely to have a customer file, a customer order file, and an order-item file. The customer file is a master file. The customer order file is a detail file whose associated file is the customer master. Every time a customer places an order, a record is added to the customer order file. In addition, each order is made up of one or more items which have been ordered by the customer. For each item, a record is added to the order-item file which is a detail file for the customer order file.

**3.4 Index File Linkage**

**r-tree** assumes that data files are linked through an index file in one of two ways: through an exact key match or a partial key match.

If you are trying to join the contents of a master file to another file, the linkage is accomplished through a unique index defined over the master file. In this case **r-tree** will automatically invoke an EQLREC call in order to retrieve the associated master file record.
If you are trying to find all records in a detail file which relate to a particular record in its associated file, the linkage is accomplished through an index defined over the detail file. This index’s key values are comprised of a leading (i.e., most significant) segment which corresponds to a field (or derived field) of the associated file. In this case r-tree® will automatically invoke FRSET and NXTSET to retrieve all associated records in the detail file.

3.5 Variable Length Files

r-tree® will work with a mix of fixed and variable length data files. The only restrictions placed on variable length files are:

- While the length of individual fields and the overall record may vary, there must be a fixed number of fields in the record. This is consistent with the relational model.
- The varying length fields must come after the fixed length portion of the record (although the fixed length portion may be defined to be zero bytes).
- The varying length fields must have a defined field delimiter byte. r-tree® is shipped assuming that the varying length fields are terminated by a null byte, but this may be changed at compile time (#define FIELD_DELIM in ctopt2.h or ctoptn.h for c-tree® V4.3 or earlier). If you wish to use another method of determining the beginning and end of varying length fields, then some reprogramming of the r-tree® source code (rtread.c) must be accomplished.

3.6 Customer Order Example

To further illustrate these file concepts, and to provide an example environment for the rest of the manual, let us look closely at a customer order system. We will assume that the customer order system is made up of the following data files:

- customer master file;
- item master file;
- customer order file; and
- order-item file.

These files are comprised of the following fields:
### File and r-tree

#### 3.6 Customer Order Example

<table>
<thead>
<tr>
<th>Data File</th>
<th>Symbolic Field Name</th>
<th>Interpretation</th>
<th>Unique Values*</th>
<th>Fixed Length Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>custmast.dat</td>
<td>cm_custnumb</td>
<td>customer number</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>cm_custratg</td>
<td>customer credit rating</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>cm_custname</td>
<td>customer name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cm_custadr</td>
<td>customer address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>itemmast.dat</td>
<td>im_itemnumb</td>
<td>item number</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>im_itemwght</td>
<td>item shipping weight</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>im_itempric</td>
<td>item price</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>im_itemdesc</td>
<td>item description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>custordr.dat</td>
<td>co_ordrnumb</td>
<td>order number</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>co_custnumb</td>
<td>customer number</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_ordrdate</td>
<td>order date</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>co_promdate</td>
<td>promised delivery date</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>ordritem.dat</td>
<td>oi_ordrnumb</td>
<td>order number</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oi_itemnumb</td>
<td>item number</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oi_sequnumb</td>
<td>sequence number</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oi_quantity</td>
<td>quantity ordered</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

*A field has a unique value if no other record in the same file has the same value for that field.*

These data files will be indexed as follows:

<table>
<thead>
<tr>
<th>Data File</th>
<th>Symbolic Index Name</th>
<th>Key Segment(s)</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>custmast.dat</td>
<td>cm_custnumb_idx</td>
<td>cm_custnumb</td>
<td>yes</td>
</tr>
<tr>
<td>itemmast.dat</td>
<td>im_itemnumb_idx</td>
<td>im_itemnumb</td>
<td>yes</td>
</tr>
<tr>
<td>custordr.dat</td>
<td>co_ordrnumb_idx</td>
<td>co_ordrnumb + co_custnumb</td>
<td>yes</td>
</tr>
<tr>
<td>ordritem.dat</td>
<td>oi_ordrnumb_idx</td>
<td>oi_ordrnumb + oi_sequnumb</td>
<td>yes</td>
</tr>
<tr>
<td>ordritem.dat</td>
<td>oi_itemnumb_idx</td>
<td>oi_itemnumb</td>
<td>no</td>
</tr>
</tbody>
</table>

**r-tree® Report Generator**

Chapter - 3
With the above file and index definitions in mind, let us see how we can link the item master file to the order-item file. At a given order-item record, we have a field which contains the item number. This item number uniquely identifies the item. Therefore, the r-tree will issue an EQLREC call to the item master file using the item number key defined over the item master data file. This relationship is depicted below:

```
Order-Item Record  |  order number | item number | sequence number | quantity ordered
                  |              |             |                 |
                  |              |             |                 |
im_itemnumb_idx:   |  links each order-item record to a particular item master record
                  |              |             |                 |

Item Master Record |  item number | item shipping weight | item price | description
                    |              |                     |
```

Continuing with this example, let us see how we can link the order-item detail file to the customer order file. At a given customer order record, we have a unique order number. This order number corresponds to the leading key segment in the index defined over the order-item detail file. Hence, once r-tree is informed of this file interrelationship, it will (automatically) issue a FRSET and subsequent NXTSET's using the order number as the partial key. This file interrelationship is depicted as follows:

```
Customer Order Record |  order number | customer number | order date | delivery date
                      |              |               |           |
                      |              |               |           |
oi_ordnumb_idx:       |  links each customer order record to all order item records with matching order numbers
                      |              |               |           |
Order-Item Records    |  order number | item number | sequence # | quantity ordered
                      |              |             |            |
                      |              |             |            |
                      |              |             |            |
```

```
Figure 3-1. One to One Link
```

```
Figure 3-2. One to Many Link
```
3.7 File Interrelationship Summary

There are three main items which the programmer must deal with in setting up a c-tree® Plus / r-tree® application:

- The programmer is responsible for setting up indices which will link multiple files. If prepared properly, r-tree will be able to automatically link and traverse multiple files.
- To link a record from file A to a single record in file B, there must be a field in file A (or a virtual field derived from file A) which corresponds to a unique index for file B.
- To link a record from file A to one or more records in file B, there must be a field in file A (or a virtual field derived from file A) which corresponds to the leading key segment in an index for file B. The index for file B need not be a unique index.
4. Data Object Definition Array (DODA)

The first step to implementing \texttt{r-tree} in an application using \texttt{c-tree Plus} is to create a Data Object Definition Array, or DODA. This will be a description of the data fields (data objects) to use with \texttt{r-tree}, along with any other variable whose value you wish to make accessible to \texttt{r-tree}. The DODA contains the following information for each data object:

- symbolic name;
- (absolute) address;
- type of value; and
- length (in bytes).

The entries in the DODA describe the fields of the data files as well as any other variable used in your program that you wish to access in a report. Variables defined for use exclusively in the report itself are NOT created in the DODA, but will be defined as "virtual fields" in the report script, as defined in Chapter 6. The report script can define "virtual fields" based on data record fields or other entries in the DODA; but these virtual fields are NOT defined by you in the DODA. They are defined in the script itself as discussed in Chapter 6.

4.1 Header File

In every application that you will be creating a DODA you must include the header file \texttt{rtdoda.h}. This file contains a typedef for the structure \texttt{DATOBJ}, which contains the four structure members listed above (plus some others used internally by \texttt{r-tree}). In addition, \texttt{rtdoda.h} contains symbolic constants to refer to the field types supported by \texttt{r-tree}. These symbolic constants are:

<table>
<thead>
<tr>
<th>DODA Field Type-Constants</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCHAR</td>
<td>one-byte signed integer</td>
</tr>
<tr>
<td>RTCHARU</td>
<td>one-byte unsigned integer</td>
</tr>
<tr>
<td>RTINT2</td>
<td>two-byte signed integer</td>
</tr>
<tr>
<td>RTINT2U</td>
<td>two-byte unsigned integer</td>
</tr>
<tr>
<td>RTINT4</td>
<td>four-byte signed integer</td>
</tr>
<tr>
<td>RTINT4U</td>
<td>four-byte unsigned integer</td>
</tr>
<tr>
<td>RTDATE</td>
<td>date in four-byte computational form</td>
</tr>
</tbody>
</table>

Continued on Next Page
DODA Field Type-Constants | Interpretation
--- | ---
RTMONEY | four-byte signed integer with implied decimal point
RTSFLOAT | single-precision floating point
RTDFLOAT | double-precision floating point
RTSTRING | character array

4.2 How To Setup a DODA

All or most of the entries in the DODA will be for data record fields. r-tree® assumes that you have assigned global structures (or buffers) for each data file. To show how to create a DODA, we will use the data structures in Figure 4-1 representing the customer order example started in the previous chapter.

```c
/* Maximum Field Length Definitions */
define CNUM 4  /* customer number */
define ZIPC 9  /* zip code */
define STWT 3  /* state abbrev. */
define INUM 5  /* item number */
define ONUM 6  /* order number */
struct { /* CUSTOMER MASTER FILE BUFFER */
  TEXT cm_cnum(CNUM);  /* customer number */
  TEXT cm_zip(ZIPC);  /* customer zip code */
  TEXT cm_stat(STWT);  /* customer state */
  TEXT cm_creatc;  /* customer credit rating */
  TEXT cm_cname(ULEN + 1);  /* customer name */
  TEXT cm_cadres(ULEN + 1);  /* customer address */
  TEXT cm_ccity(ULEN + 1);  /* customer city */
} cm;
struct { /* ITEM MASTER FILE BUFFER */
  LONG im_weight;  /* item weight */
  LONG im_pric;  /* item price */
  TEXT im_idem(ULEN + 1);  /* item description */
  TEXT im_stat(ULEN + 1);  /* item status */
  TEXT im_deldate(ULEN + 1);  /* item delivery date */
} mm;
struct { /* ORDER-ITEM FILE BUFFER */
  COUNT oi_deldate(ULEN);  /* order date */
  TEXT oi_deldate(ULEN);  /* order number */
  TEXT oi_cust(ULEN);  /* customer number */
} oo;
struct { /* ORDER-ITEM FILE BUFFER */
  COUNT oi_deldate(ULEN);  /* order date */
  TEXT oi_deldate(ULEN);  /* order number */
  TEXT oi_cust(ULEN);  /* customer number */
} oo;
TEXT path(163);  /* directory path string variable */
```

Figure 4-1. Customer Order Structures.
Given the data structures above, we would create the following DODA (remember to include rtdoda.h in your application).

```c
DODA app1 = {
  "cm.cm_cnumb", cm.cm_cnumb, RTSTRING, CMPU,
  "cm.cm_csize", cm.cm_csize, RTSTRING, ZIPC,
  "cm.cm_customer", cm.cm_customer, RTSTRING, STATE,
  "cm.cm_customer", cm.cm_customer, RTSTRING, CITY,
  "cm.cm_cname", cm.cm_cname, RTSTRING, ULEN,
  "cm.cm_caddr", cm.cm_caddr, RTSTRING, ULEN,
  "cm.cm_ccity", cm.cm_ccity, RTSTRING, ULEN,
  "cm.cm_cnumb", cm.cm_cnumb, RTSTRING, ULEN,
  "cm.cm_csize", cm.cm_csize, RTSTRING, ZIPC,
  "cm.cm_customer", cm.cm_customer, RTSTRING, STATE,
  "cm.cm_customer", cm.cm_customer, RTSTRING, CITY,
  "cm.cm_cname", cm.cm_cname, RTSTRING, ULEN,
  "cm.cm_caddr", cm.cm_caddr, RTSTRING, ULEN,
  "in_itemnum", (pTEXT *)&in_itemnum, RTINT4,
  "in_itemprice", (pTEXT *)&in_itemprice, RTMONEY,
  "cm.cm_iaddr", cm.cm_iaddr, RTSTRING, INPU,
  "cm.cm_icity", cm.cm_icity, RTSTRING, ULEN,
  "co_delflag", (pTEXT *)&co_delflag, RTINT4,
  "co_orderdate", (pTEXT *)&co_orderdate, RTDATE,
  "co_ordernum", (pTEXT *)&co_ordernum, RTNUM,
  "oi.oi_delflag", oi.oi_delflag, RTINT4,
  "oi.oi_inumb", oi.oi_inumb, RTNUM,
  "oi.oi_iaddr", oi.oi_iaddr, RTSTRING, INPU,
  "oi.oi_icity", oi.oi_icity, RTSTRING, ULEN,
  "path", path, RTSTRING, ID, /* Not part of any record */
};
```

Figure 4-2. Customer Order DODA.

Please note the following key features about the DODA in Figure 4-2.

- **app** is an arbitrary name assigned to the DODA and passed to the report () functions at run-time;
- the quoted symbolic names are the symbolic names which will be used to reference the data objects in the report script;
- this one DODA structure is used to define all of the data structures. `cm.cm_cnumb` and `cm.cm_ccity` are the addresses of the first and last fields of the customer master file data record buffer;
- `&oi.oi_delflag` and `oi.oi_inumb` are the addresses of the first and last fields of the order-item file data record buffer;
- there is no address operator (&) and (pTEXT) cast in front of `cm.cm_cnumb` (amongst others) since it is an array; and
- the `{"","",0,0,-1}` entry in the DODA serves as the end of DODA signal.
• Except for strings, the field length is determined by the field type, so you can eliminate the length specifier for all but the string fields.

When a data object is an array (e.g., a character array), you should not use the address operator (&) when specifying the data object address since the array name is an address constant.

In addition to the above rules for each data object represented in the DODA, there are several overall rules which must be followed:

• Your application program must define all the variables and record buffers before it specifies the DODA; otherwise, the address references in the DODA will not be resolved.
• The fields from a data record must be listed consecutively in the DODA in the order they appear in the record buffer.
• Each data object with a non-null symbolic name must be assigned a unique symbolic name.
• The first and last fields of each data record must not have null symbolic names.
• You must define the first and last fields of the data record buffer in the DODA, even if they are fields that are not to be used. For example, we won’t use $oi.oi_delflag$ in any reports, but it must be defined in the DODA because it is the first field of the data record buffer.
• The rtdrvr application requires that all fields in the record structure be present in the DODA. In addition, all field lengths must be consistent between the record structure and the DODA. This requirement is due to rtdrvr reading the DODA and ISAM file definitions from the data file.
• Multi-dimensional character arrays and arrays of numerics are not supported. If a record structure has either of these types of arrays, each individual element of the array must be listed separately in the DODA.
• The DODA is useful for passing information between the driving C program and the r-tree® script (.rts). See the path variable in the DODA example on the previous page.

You may use a dynamically allocated (built at run-time) DODA if you have the following for each DODA entry:

• your own means to store the symbolic name, field type, field length, and relative offset from the beginning of the appropriate data buffer; and
• a method to convert the relative offset to an absolute data address.
5. c-tree Parameter Information

After you have defined your DODA you must make a connection between \texttt{r-tree} and \texttt{c-tree} Plus. This is done by modifying the \texttt{c-tree Plus} Incremental ISAM or ISAM parameter file structure.

Please note that if you have the \texttt{c-tree} source code you must have the two following define statements in the header file \texttt{ctoptn.h} (note, if \texttt{ctoptn.h} is changed all of \texttt{c-tree} and \texttt{r-tree} must be recompiled):

\begin{verbatim}
#define CTS.ISAM
#define RTREE
\end{verbatim}

5.1 Incremental ISAM Structure

The incremental ISAM structures permit your application program to create, open, close and rebuild individual ISAM files independent of an ISAM parameter file. The incremental ISAM structures contain essentially the same information as the parameter file, but it permits the information to be stored in your application instead of an external file, and it permits direct control of the opening and closing of individual files instead of a complete set of files.

There are three structures which work together to enable the incremental ISAM capability: IFIL which defines the data file, IIDX which defines the indices, and ISEG which defines the key segments. Details on how to create these structures can be found in the \texttt{c-tree Plus} Programmer’s Reference Guide. We will briefly review the information that relates to \texttt{r-tree}.

\textbf{IFIL Structure}

All incremental ISAM functions require a pointer to an IFIL structure as their input parameter. The IFIL structure defines the characteristics of the data file and includes a pointer to a structure which defines the associated indices. The formal type definition for IFIL is illustrated in \textit{Figure 5-1}. 

Programmer’s Reference Guide Chapter - 5 37
There are two fields that relate to r-tree®, \texttt{rfstfld} and \texttt{rlstfld}.

\texttt{rfstfld}

This is a pointer to the symbolic name of the first data field in the record as specified in the r-tree DODA.

\texttt{rlstfld}

This is a pointer to the symbolic name of the last data field in the record.

**IIDX Structure**

Each index is described by an IIDX structure. If a data file has associated indices, then the ix member of the IFIL structure must point to an array of IIDX structures. In particular, there must be \texttt{dnumidx} elements (i.e., IIDX structures) in the array. The formal type definition for IIDX can be seen in Figure 5-2:

```c
typedef struct ifil {
    TEXT *pfilnam, /* file name (w/o ext) */
    COUNT dfiles, /* data file number */
    COUNT dreclem, /* data record length */
    COUNT dfilesz, /* data file ext size */
    COUNT dfilemod, /* data file mode */
    COUNT dnumidx, /* number of indices */
    COUNT dxdiclz, /* index file ext size */
    COUNT dxdimod, /* index file mode */
    IIDX *ix, /* index information */
    TEXT *rfstfld, /* r-tree list fld name */
    TEXT *rlstfld, /* r-tree list fld name */
    COUNT tfiles, /* temporary file number */
} IFIL;
```

*Figure 5-1.*

There are two fields that relate to r-tree®, \texttt{rfstfld} and \texttt{rlstfld}.

```c
typedef struct idx {
    COUNT lkeylen, /* key length */
    COUNT lkeytype, /* key type */
    COUNT lkeydup, /* duplicate flag */
    TEXT lnullkey, /* null key flag */
    TEXT lexemptyc, /* empty character */
    TEXT lexemptyx, /* number of segments */
    TEXT lexseg, /* segment information */
    TEXT *rxidnam, /* r-tree symbolic name */
    TEXT *ridxnam, /* optional index file name */
    COUNT raltseq, /* optional alternate sequence */
    TEXT *pdxbyte, /* optional pointer to pad byte */
} IDX;
```

*Figure 5-2.*
There is one field that relates to r-tree®, ridxnam.

This is a pointer to the symbolic index name for use with r-tree.

5.2 ISAM Parameter File

As described in the c-tree® Plus Programmer’s Reference Guide, you must add fields to several of the records of the ISAM parameter file.

Data File Description Record

Each data file description record now has two more parameters following the standard six parameters:

Original Parameters
1) data file number 4) file size extension
2) data file name 5) file mode
3) record length 6) number of indices

New Parameters
7) symbolic name of first field in record
8) symbolic name of last field in record

Index File Description Record

Each index file description record now has an additional parameter following the standard eleven parameters:

Original Parameters
1) index file number 7) file size extension
2) index file name 8) file mode
3) key length 9) null key flag
4) key type 10) empty character
5) duplicate flag 11) number of key segments
6) additional members

New Parameter
12) symbolic index name
Index File Member Record

Each index member record has an additional parameter following the standard seven parameters:

**Original Parameters**

1) index number  5) null key flag  
2) key length  6) empty character  
3) key type  7) number of key segments  
4) duplicate flag

**New Parameter**

8) symbolic index name

Sample Parameter File

The sample ISAM parameter file in Figure 5-3 matches the customer order example that we introduced in the previous chapter:

```
12 6 4 5
  0   dureqlck.dt  120  0  3  0  co_delflag  co_delflag
  1   custcust.dt 17  0  5  1  co_customer  cs_customer
      5 custcust.ix 400 0 0 10 0 1  co_customer_idx  
             8 4 2
  2   itenitem.dt 13  0  5  1  ln_item_weight  ln_item_desc
      6 itenitem.ix 500 0 0 10 0 1  ln_item_idex  
             8 5 2
  3   custordr.dt 22  0  1  2  co_delflag  co_customer
      7 custordr.ix 640 1 0 10 0 1  co_ordnumb_idx  
             12 6 2
     8  8 4 1  0  0  1  co_customer_idx  
     7  18 4 2
  4   orditem. dt 17  0  1  2  co_delflag  oi_itemnum
      9 orditem.ix 840 1 0 10 0 2  oi_ordnumb_idx  
     6 6 2
     7  2 1
     10  9 4 1  0  0  1  oi_item_idex

Figure 5-3. Customer Order Parameter File.
• Even though we do not have a data record buffer for the dummy lock file (only needed in some operating environments, such as DOS networks), we still need to add symbolic names for r-tree®. As in our example, these symbolic names can be taken from any other buffer.

• Be sure to use the actual first field of the data record buffer for the beginning symbolic name. This will be used as the starting point for r-tree to read the record into.

• Each index is given a unique symbolic index name.
This page intentionally left blank.
6. Report Script

An r-tree® report script describes how to prepare and produce a report based on one or more c-tree® Plus data files. The real power of r-tree lies in the fact that, with proper formulation, you can make r-tree perform arbitrarily complex virtual field computations, automatic traversal of multiple files, and arbitrarily complex record selection decisions.

6.1 Report Script Sections

Each report script is made up of one or more of the following optional sections. However, without SEARCH and IMAGE sections no output will be generated.

- **VIRTUAL**: Virtual field definitions.
- **SEARCH**: File traversal hierarchy.
- **SELECT**: Multiple file record selection.
- **SORT**: Multiple segment sort key definition.
- **CONTROL**: Control break definitions.
- **ACCUMULATOR**: Accumulator definitions.
- **DISPLAY**: Output control specifications.
- **IMAGE**: Report layout.

There may be more than one VIRTUAL section in a report script. This is useful when some virtual fields are needed in the SEARCH section (and must therefore be defined before the SEARCH section) while other virtual fields depend on information in the SEARCH section (and must therefore be defined after the SEARCH section which is likely to happen when file aliases, as discussed in chapter 8, are utilized). Also, there may be more than one DISPLAY section.

While a report script has no spacing or alignment restrictions, and comments may be placed wherever you wish, the script must conform to the r-tree syntax. This syntax is presented formally in Appendix B. In the sections which follow we will demonstrate the correct form of a report script somewhat more informally.

**Some Report Script Ground Rules**

- **START**
- **EXIT**

Each report script must begin with the keyword START and end with the keyword EXIT.
Keywords

*r-tree®* keywords begin with an alphabetic character, and are comprised of upper-case alphabetic characters, digits and underscores.

Comments

Comments are delimited by a /* ... */ sequence.

Symbolic Field Names

Symbolic field names are formed under the same rules as keywords except that they are not restricted to the use of upper-case letters. Tabs, new lines and/or comments may appear anywhere that a space is allowed in the syntax.

Special Terms

In the sections that follow we use the following special terms:

cvalue

A constant value of one of the following five types;

- **string**: A sequence of any number of characters enclosed in double quotes ("abc"). *r-tree* does NOT recognize a single quoted constant (e.g., 'S'); the decimal equivalent of its ASCII representation must be used.
- **integer**: A sequence of decimal digits (123).
- **float**: A sequence of decimal digits with a decimal point (123.45).
- **date**: A string of exactly eight characters of the form "mm/dd/yy" which represents a valid date. The constant is converted by *r-tree* to a four-byte computational form, ("01/01/87").
- **money**: A dollar sign followed by an integer or float constant. The constant is stored as a four-byte integer with an implied decimal point. ($123 would be used to represent $1.23)

int2u

A small positive integer constant;

symbol

Symbolic name of an application defined data object, virtual field, accumulator, *r-tree* status variable (such as PAGE_NO or SYS_DATE), or *r-tree* symbolic constant (such as YES or PRINTER1);

function

Symbolic name of a built-in function defined in the rtuser.c module. User defined functions may be added to this module;

value

A cvalue, a symbol, or a function(symbol) where the symbol or function are evaluated by *r-tree* to determine the value;

cmpexp

A computational expression involving value’s; the operators: +, -, *, / and # (concatenation); and any level of parenthesized expressions; and

bolexp

A boolean expression involving cmpexp’s; the relational operators: <, <=, =, >=, >, ><; the logical connectives: NOT, AND, and OR; and any level of parenthesized expres-
sions. Unlike language C, a bolexp cannot be simply a numeric value; it must involve a relational operator.

**field_type**

A definition of the type of field to be used, chosen from one of the following:

<table>
<thead>
<tr>
<th>field_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>Single character.</td>
</tr>
<tr>
<td>CHARU</td>
<td>Unsigned single character.</td>
</tr>
<tr>
<td>INT2</td>
<td>Two-byte integer.</td>
</tr>
<tr>
<td>INT2U</td>
<td>Unsigned two-byte integer.</td>
</tr>
<tr>
<td>INT4</td>
<td>Four-byte integer.</td>
</tr>
<tr>
<td>INT4U</td>
<td>Unsigned four-byte integer.</td>
</tr>
<tr>
<td>SFLOAT</td>
<td>Single precision float.</td>
</tr>
<tr>
<td>DFLOAT</td>
<td>Double precision float.</td>
</tr>
<tr>
<td>STRING</td>
<td>Character string.</td>
</tr>
<tr>
<td>DATE</td>
<td>Four-byte integer.</td>
</tr>
<tr>
<td>MONEY</td>
<td>A four-byte integer with an implied decimal.</td>
</tr>
</tbody>
</table>

Expressions delimited by the curly braces "{" and "}" are optional. Expressions separated by a vertical bar "|" represent alternatives. An ellipsis "..." in the curly braces means that the optional items may be repeated zero or more times.

Appendix A presents a brief overview of the r-tree® reserved words and operators, and a complete, alphabetically organized reference to the reserved words.

### 6.2 VIRTUAL

The VIRTUAL section is used to define symbolic names for fields internal to r-tree, as opposed to fields external to r-tree defined in the DODA. r-tree recognizes four types of virtual field definitions: **simple**, **boolean**, **overlay**, and **coded**. All virtual field definitions begin:

```plaintext
symbol field_type int2u
```

where the `symbol` is the virtual field symbolic name; the `field_type` is one of the choices as defined above, and `int2u` is the field length.

Field lengths for the various field_type's can vary from one operating environment to another. For the sake of transportability you should define a symbolic constant for the length of each field_type. The following chart gives an example for a typical DOS environment:
The field length for a STRING field_type would be the maximum length of the string.

### Simple

A simple virtual field definition is given by:

```plaintext
symbol field_type int2u cmpexp
```

These will be simple calculated fields, typically used when creating a new value from others in the DODA. For example:

```plaintext
extended_price MONEY 4 oi_quantity * im_itemprice
```

### Boolean

A boolean virtual field definition is given by:

```plaintext
symbol field_type int2u bolexp ? cmpexp : cmpexp
```

where the first computational expression is used if the boolean expression is true; otherwise the second computational expression is evaluated. For example:

```plaintext
days_late INT2 2 co_promdate < SYS_DATE ? SYS_DATE - co_promdate : 0
```

### Overlay

An overlay virtual field definition is given by:

```plaintext
symbol field_type int2u & symbol { + cmpexp | - cmpexp }
```

where the optional expression computes a byte offset from the beginning of the field following the address operator &. For example, the following definition would take the third and fourth characters from a string field:

```plaintext
item_class STRING 2 &im_itemnumb + 2
```

### Coded

A coded virtual field definition is given by;

<table>
<thead>
<tr>
<th>field_type</th>
<th>define</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>#define CHARLEN 1</td>
</tr>
<tr>
<td>CHARU</td>
<td>#define CHARULEN 1</td>
</tr>
<tr>
<td>INT2</td>
<td>#define INT2LEN 2</td>
</tr>
<tr>
<td>INT2U</td>
<td>#define INT2ULEN 2</td>
</tr>
<tr>
<td>INT4</td>
<td>#define INT4LEN4</td>
</tr>
<tr>
<td>INT4U</td>
<td>#define INT4ULEN 4</td>
</tr>
<tr>
<td>SFLOAT</td>
<td>#define SFLOATLEN 4</td>
</tr>
<tr>
<td>DFLOAT</td>
<td>#define DFLOATLEN 8</td>
</tr>
<tr>
<td>DATE</td>
<td>#define DATELEN 4</td>
</tr>
<tr>
<td>MONEY</td>
<td>#define MONEYLEN 4</td>
</tr>
</tbody>
</table>
symbol field_type int2u  : cmpexp int2u cvalue [int2u cvalue ... ]

The computational expression is evaluated and the result compared with the small positive constants. If a match is found, then the virtual field is assigned the corresponding constant value. If no match is found, then the field is assigned the last constant value in the list. Figure 6-1 illustrates this process.

```
colorful_rating  STRING 12  : cn_cust strs
  1  "Very Good"
  2  "Satisfactory"
  3  "Fair"
  4  "Bad Debt"
  5  "???????????"
```

Figure 6-1.

6.3 SEARCH

The SEARCH section defines the file(s) to be searched. Each definition begins with the key word FILE. The first file defined in the SEARCH section is the primary file. Primary files may be searched in three modes: **physical sequential order**, **key sequential order**, and **key sequential order subject to range restrictions** on the keys.

**Physical Order**

Physical order is specified by a statement of the form:

```
FILE value ALL
```

where value contains the file name. For example,

```
FILE "custmast.dat" ALL
FILE customer_data ALL
```

are both legal (as long as customer_data evaluates to a string value). The following statement is NOT legal since it involves a computational expression concatenating two symbols:

```
FILE path # customer_data ALL
```

However, concatenating a path name to a file name can be accomplished if the expression is moved to a virtual field definition.

**Key Sequential Order**

Key sequential order is specified by a statement of the form:

```
FILE value USING_KEY symbol
```
where the symbol must be a symbolic index name for this data file, as defined in the ISAM parameter file, or Incremental ISAM structure. You cannot use the actual index file name since c-tree® Plus allows for more than one index in a physical index file.

Key sequential order with range restrictions is specified by a statement of the form:

```
FILE value USING_KEY symbol range { range ...}
```

where a range is defined to be one of the following:

- \([ value \) lower limit inclusive
- \) value lower limit exclusive
- value \] upper limit inclusive
- value \] upper limit exclusive

For example,

```
FILE "custmast.dat" USING_KEY cm_custmast_idx ["1000" "2000"]
```

specifies that the data file custmast.dat is to be searched in key sequential order for keys greater than or equal to "1000" and strictly less than "2000".

Since the range is based on a value, it does not have to be a constant. You may use fields defined in your data object definition array in range expressions. You may also use virtual fields if they have been defined before the SEARCH section, and they do not depend on information from a data file.

### Secondary Files

Files listed after the primary file are considered to be secondary files. Each secondary file must be linked to exactly one of the files specified before it in the SEARCH section. The form of a secondary file statement is

```
FILE value file_relation value BY_FIELD symbol USING_KEY symbol
```

The first value must evaluate to the file name. The second value must evaluate to a file already named in the SEARCH section. The file_relation alternatives are JOINS_TO and IS_DETAIL_FOR.

**JOINS_TO**

This is used when you want r-tree® to use an EQLREC call to associate two files. In other words, there will be a specific field in the first file that can be used as an exact key value for the index for the second file. When this is the case, then the two files are conceptually joined together as though they were one expanded file. JOINS_TO corresponds to one-to-one or many-to-one file interrelationships.
Note: A JOINS_TO relationship cannot be used with a duplicate allowed index because the c-tree® EQLREC function requires a one-to-one interrelationship.

This is used when you want r-tree® to use the FRSET, NXTSET calls to link the files. In this case, the file is considered to be one step below its associated file in a hierarchy of files. r-tree allows up to eight different files to be linked to a given file by the IS_DETAIL_FOR keyword. Further, there may be up to six levels in the file hierarchy specified in the SEARCH section. IS_DETAIL_FOR corresponds to a one-to-many file interrelationship.

The first symbol represents a data object defined by the application (usually a field from the parent or associated file) or a virtual field (usually derived from one or more fields of the associated file). The second symbol must evaluate to the index (over the detail or master file) which will be searched. In other words, the first symbol represents the target used to search the index represented by the second symbol.

Although the concept of file hierarchy will be discussed in chapter 8, let us look at a simple multi-file example for SEARCH based on the customer order example.

Figure 6-2’s SEARCH section represents a three-level file hierarchy with the customer master on the top, the customer order file in the middle, and the order-item file (joined to the item master file) at the bottom. Specifying the files in this manner implies r-tree will begin reading the customer master file and work its way down the hierarchy as required to satisfy the SELECTION and SORT sections of the report script.

```plaintext
VIRTUAL

/* assume path is defined in the application DDM */
customer_master  STRING  32  path = "customer.dat"
item_master      STRING  32  path = "item.dat"
customer_order   STRING  32  path = "customer.dat"
order_item       STRING  32  path = "order_item.dat"

SEARCH

FILE customer_master USING KEY cm_cnumb_idx
FILE customer_order  IS_DETAIL_FOR customer_master BY FIELD cm_cnumb USING KEY co_cnumb_idx
/* the above statement implies that the customer number field of the customer master will serve as the target key for searching the customer number index of the customer order file */
FILE order_item  IS_DETAIL FOR customer_order BY_FIELD co_cnumb USING KEY o1_oitem_nb_idx
FILE item_master USING TO order_item BY FIELD o1_item_nb USING KEY in_item_nb_idx
/* the above statement implies that the item number field of the order-item detail file will serve as the DURMC target when joining the item master to the order-item detail file */
```

Figure 6-2. Multi-file Search Example.

If the focus of the report script had been on the customer orders (instead of the customers), the SEARCH section for the same data files could be formulated as shown in Figure 6-3. In this SEARCH section, there are only two levels in the hierarchy.
Customer order file joined to the customer master file is at the top, and the order-item file joined to the item master file is at the bottom.

```
SEARCH
  FILE customer_order ALL
  FILE customer_master JOINED_TO customer_order
  BY_FIELD
    co_custnumb USING KEY cm_custnumb_idx
  FILE order_item IS DETAIL FOR customer_order
  BY_FIELD
    co_custnumb USING KEY ol_custnumb_idx
    FILE item_master JOINED_TO order_item
  BY_FIELD
    ol_custnumb USING KEY im_custnumb_idx
```

Figure 6-3. Two-level File Hierarchy.

Multi-segment Keys

The SEARCH section within r-tree® supports a single token to identify the "target" portion of the BY_FIELD relationship. Many keys are made up of more than one key segment and require this "target" field to be made up of more than one field. To do this you must use a VIRTUAL field as the BY_FIELD target. This VIRTUAL field can be a result of a concatenation of more than one field. The trick is that r-tree® only supports concatenating STRING type fields. If you have a segment made up of non-string type data, you must create a work VIRTUAL field to help in the concatenation process. See the following example. We have created two VIRTUAL target fields. Note the VIRTUAL work fields we have used for the numeric type segments:

- **company** is a two byte integer
- **department** is a 4 byte integer
- **employee** is a 6 byte string
- **em_company_department_ids** is an index over the employee master file made up of two numeric segments 'company' and 'department'.
- **ed_comp_dept_emp_ids** is an index over the employee detail file made up of three segments 'company' and 'department' and 'employee'.

```
6.4 SELECT

The SELECT section is used to select a subset of the records to be included in the file. There are two statement types in the SELECT section. First,

SELECT ALL

which selects all record combinations; and

SELECT boolexp

which selects all combinations of records from the hierarchy represented in the SEARCH section for which the boolean expression is true.

Figure 6-5 demonstrates use of the SELECT ALL statement. The SEARCH segment defines a two-level file hierarchy. The SELECT ALL statement means each customer order record and order-item record linked by the index \( \text{oi}._{\text{ordnumb}}._{\text{idx}} \) will be
selected. In addition, the files joined via the JOINS_TO keywords will be automatically read and included in the selection by r-tree®.

Assuming the same VIRTUAL and SEARCH sections as above, consider the SELECT section in Figure 6-6.

This SELECT expression will cause customer order records to be selected if and only if the promised date is more than thirty days after the order date and the promised date was before today (SYS_DATE represents the current system date) or the customer has a rating of 1.

NOTE: the above SELECT expression will not cause any of the order-item records to be read or selected since no reference is made to the order-item data fields (either directly or through virtual fields which depend on the order-item fields).

By modifying the SELECT expression as shown in Figure 6-7 we retain the same customer order records to be potentially selected; but, in addition, r-tree® goes down to

---

**Figure 6-5. Select All Example.**

**Figure 6-6. Select Boolean Expression.**

This SELECT expression will cause customer order records to be selected if and only if the promised date is more than thirty days after the order date and the promised date was before today (SYS_DATE represents the current system date) or the customer has a rating of 1.

NOTE: the above SELECT expression will not cause any of the order-item records to be read or selected since no reference is made to the order-item data fields (either directly or through virtual fields which depend on the order-item fields).
the next level in the hierarchy to select those order-item records which are for an order quantity not equal to zero, and linked to the potentially selected customer order record.

If a customer order record satisfies the expression;

\[(\text{co\_promdate} - \text{co\_ordrdate}) > 30 \text{ AND } \text{co\_promdate} < \text{SYS\_DATE OR cm\_custratg} = 1\] AND \oi\_quantity <> 0

but no corresponding order-item record satisfies the expression;

\oi\_quantity <> 0

the customer order record will not be selected. In short, the combined group of records must satisfy the entire boolean expression.

The important point about SELECT expressions with fields (or virtual fields) from multiple levels of a file hierarchy is that r-tree® automatically works its way down the file hierarchy to the lowest level referenced in the SELECT expression. If the boolean expression is satisfied, then the group of records (linked down the hierarchy) which satisfy the SELECT expression is treated as a single selected entity to be output in the report. This means a record near the top of the hierarchy may be repeatedly "selected" as it is combined with other associated records lower in the hierarchy.

6.5 SORT

The SORT section is used when the existing index structures (explicitly referenced in the SEARCH section) do not correspond to the desired report organization. The syntax of the SORT section is

\text{SORT USING\_KEY} \\begin{align*}
\text{segment\_translation\_mode} & \quad \text{symbol} \\
\{\text{segment\_translation\_mode} & \quad \text{symbol} \ldots\}
\end{align*}
where the symbol must be the name of an application-defined data object or a virtual field. The segment_translation_mode alternatives are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_MOD</td>
<td>no translation</td>
</tr>
<tr>
<td>DSC_NO_MOD</td>
<td>descending order</td>
</tr>
<tr>
<td>UPPER</td>
<td>upper-case translation</td>
</tr>
<tr>
<td>DSC_UPPER</td>
<td>upper-case translation</td>
</tr>
<tr>
<td>ALT_SEG</td>
<td>alternative collating sequence</td>
</tr>
<tr>
<td>ALT_DSC</td>
<td>alternative sequence and descending order</td>
</tr>
<tr>
<td>INTEGER</td>
<td>unsigned integer segment and/or date</td>
</tr>
<tr>
<td>DSC_INTEGER</td>
<td>unsigned integer segment and/or date, descending order</td>
</tr>
</tbody>
</table>

The characteristics of the sort segments are taken directly from the DODA and/or virtual field definition. Therefore, you can use NO_MOD for any sort segment (signed or unsigned integers, single or double precision floats, money or dates, and strings) which does not require special modifications such as upper case or alternative collating sequence. Descending order is handled with DSC_NO_MOD.

For example, if a sort segment has been defined as a INT4, then the segment automatically will be set for signed integers; if it is defined as INT4U, then it will be set as an unsigned integer segment.

ALT_SEG and ALT_DSC permit you to incorporate an alternative collating sequence with a sort segment. See your c-tree Plus Programmer’s Reference Guide for details on alternative collating sequences.

With c-tree® V4.1 only the first segment of the sort key can be a float or double float field. In this case, use either the NO_MOD or DSC_NO_MOD alternatives. With c-tree V4.1 the INTEGER and DSC_INTEGER modes can also be used with MONEY fields if they are always positive. If a MONEY field may be negative, then define a virtual field with a DFLOAT type and assign it the value of the money field, and then use the DFLOAT field for the sort segment. ALT_SEQ and ALT_DSC cannot be used with c-tree V4.1.
Figure 6-8 illustrates the use of multiple sort segments.

The SORT section shown above will cause the selected records to be sorted in descending order by lead_time (the longest lead time will be first) and for equal lead times, the records will be in customer number order.

NOTE: a SORT section may include segments which span multiple levels of the file hierarchy.

It is common to use the SORT section in conjunction with the CONTROL section.

### 6.6 CONTROL

Many reports include headings and other descriptions which should only be printed at the beginning of a section of a report, and totals or other summary information which are printed only at the end of a section. The CONTROL section of a report script allows you to define control break fields which are monitored by r-tree®. When a control break field changes value, summary information is output for the concluding section(s), and introductory information is output for the new section(s).

The CONTROL section syntax is

```
CONTROL int2u symbol [int2u symbol ...]
```

where the symbol is a symbolic field name, and int2u represents the control break level.

The highest control break field is at level 1. When a control break field at level i changes, summaries are output for levels n, n - 1, ..., i; and then introductions are output for levels i, i + 1, ..., n. n is the lowest control break level but has the highest associated integer value.
The data must be ordered in a manner consistent with control breaks, otherwise the output will be meaningless. When the key value ordering implied by the SEARCH section is not consistent with control break levels, the SORT section should be used to reorganize the data. Frequently, there will be one sort segment for each control break level in the CONTROL section.

The number of control breaks determines the number of levels maintained for each accumulator. The accumulators are discussed in the next section.

In addition to the explicit control break fields defined by the programmer in the CONTROL section, r-tree® automatically maintains two additional levels of control breaks and hence two additional accumulator levels:

- overall report; and
- page breaks.

With respect to our control break numbering scheme from 1 to n, the overall report level is assigned control break level zero (0), and the page control break is assigned control break level n+1.

6.7 ACCUMULATOR

The ACCUMULATOR section allows the report designer to direct r-tree to automatically maintain accumulators which can be output as part of a control break summary, at the bottom of each page, and/or at the end of a report. The syntax of the ACCUMULATOR section is:

```
ACCUMULATOR  symbol atype symbol {symbol atype symbol ...}
```

where the first symbol is the symbolic name attached to this accumulator, and the second symbol is the name of the target field which will be accumulated. atype may be one of these alternatives:

<table>
<thead>
<tr>
<th>atype</th>
<th>Tally</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRQ</td>
<td>frequency count</td>
<td>long integer</td>
</tr>
<tr>
<td>SUM</td>
<td>summation</td>
<td>double float</td>
</tr>
<tr>
<td>AVG</td>
<td>average</td>
<td>double float</td>
</tr>
<tr>
<td>MIN</td>
<td>minimum</td>
<td>double float or string</td>
</tr>
<tr>
<td>MAX</td>
<td>maximum</td>
<td>double float or string</td>
</tr>
<tr>
<td>FRS</td>
<td>first occurrence</td>
<td>double float or string</td>
</tr>
<tr>
<td>LST</td>
<td>last occurrence</td>
<td>double float or string</td>
</tr>
<tr>
<td>PRV</td>
<td>previous occurrence</td>
<td>double float or string</td>
</tr>
</tbody>
</table>

r-tree® Report Generator Chapter - 6 56
If \( \text{SUM} \) or \( \text{AVG} \) is applied to a string field, \textit{r-tree}® will automatically convert the ASCII string to a double float before performing the summation. If \( \text{MIN} \), \( \text{MAX} \), \( \text{FRS} \), \( \text{LST} \) or \( \text{PRV} \) is applied to a string field, the operation is performed on the string values themselves (i.e., no conversion takes place).

The accumulator is updated each time the target field is updated as a result of a successful record selection. A target field which is not printed in the report will still be accumulated.

If an accumulator (e.g., \( \text{LST} \)) is used to track a \( \text{DATE} \) valued field, the accumulator will maintain a double precision float internal representation. This will not yield the standard date output capabilities. However, you may define a virtual, defined as a \( \text{DATE} \) type, which is set equal to the accumulator.

Notice in Figure 6-9 how the control field breaks correspond to the sort segments in the next table. Also, by defining the accumulator \textit{order_total}, \textit{r-tree} will automatically sum the extended price. If directed in the IMAGE section, \textit{r-tree} will also output the total order price at up to five different levels: the three levels explicitly defined in the CONTROL section as well as the page and overall report level. The three explicit levels of \textit{order_total} are: for each customer order (3), the total order price across all orders for the same customer on the same date (2), and the total order price across all orders for the same customer (1).

<table>
<thead>
<tr>
<th>Virtual</th>
<th>customer_order</th>
<th>STRING</th>
<th>32</th>
<th>&quot;custorder.dt&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>item_order</td>
<td>STRING</td>
<td>32</td>
<td>&quot;itemorder.dt&quot;</td>
</tr>
<tr>
<td></td>
<td>order_item</td>
<td>STRING</td>
<td>32</td>
<td>&quot;orditems.dt&quot;</td>
</tr>
<tr>
<td></td>
<td>extended_price</td>
<td>MONEY 4</td>
<td>4</td>
<td>\text{oi_quantity}</td>
</tr>
<tr>
<td></td>
<td>\text{ln_itemprice}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{SEARCH}

\text{FILE} \text{customer_order USING KEY co_custnumb_idx}
\text{FILE} \text{customer_order JOIN TO customer_order BY FIELD co_custnumb USING EKEY co_custnumb_idx}
\text{FILE} \text{order_item IS DETAIL FOR customer_order BY FIELD co_ordernumb USING KEY ol_itemnumb_idx}
\text{FILE} \text{item_order JOIN TO order_item BY FIELD ol_itemnumb USING KEY ln_itemnumb_idx}

\text{SORT USING KEY}
\text{UPPER co_custnumb}
\text{MBMDD co_orderdate}
\text{UPPER ol_ordernumb}
1. co_custnumb
2. co_orderdate
3. ol_ordernumb

\text{#ACCUMULATOR}

\text{order_total} \ SUM extended_price

\text{Figure 6-9. Sort-Control-accumulator Example.}
The DISPLAY section allows the overall nature of the report to be controlled from the report script itself. In the Advanced Report Layout Chapter we discuss how to modify and use this control from your own application. The syntax of the DISPLAY section is

```
DISPLAY {display_attribute value ...} {device_string value {value ...} ...}
```

where `value` must evaluate to a positive integer, and there must be at least one `display_attribute` or `device_string` referenced. (Note that you may eliminate the entire DISPLAY section if desired.)

The `display_attributes` and their default values are listed below.

<table>
<thead>
<tr>
<th>Display Attribute</th>
<th>Default</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_WIDTH</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>PAGE_LENGTH</td>
<td>66</td>
<td>total physical lines per page</td>
</tr>
<tr>
<td>SCREEN_WIDTH</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>SCREEN_LINES</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>DATE_ORDER</td>
<td>0</td>
<td>modify month</td>
</tr>
<tr>
<td>CURRENCY_MARK</td>
<td>36</td>
<td>'$'</td>
</tr>
<tr>
<td>THOUSAND_SEPARATOR</td>
<td>44</td>
<td>','</td>
</tr>
<tr>
<td>DECIMAL_POINT</td>
<td>46</td>
<td>'.'</td>
</tr>
<tr>
<td>DATE_SEPARATOR</td>
<td>47</td>
<td>'/'</td>
</tr>
<tr>
<td>LINE_MARK</td>
<td>43</td>
<td>'+'</td>
</tr>
<tr>
<td>FORMAT_MARK</td>
<td>64</td>
<td>'@'</td>
</tr>
<tr>
<td>PCNTRL_MARK</td>
<td>64</td>
<td>'^'</td>
</tr>
<tr>
<td>FIRST_PAGE_HDR</td>
<td>1</td>
<td>if zero, then 1st page has no page header</td>
</tr>
<tr>
<td>LAST_PAGE_FTR</td>
<td>1</td>
<td>if zero, then last page has no page footer</td>
</tr>
<tr>
<td>ZERO_FILL</td>
<td>42</td>
<td>'*'</td>
</tr>
</tbody>
</table>
While the display_attributes control various characteristics of the display environment, the device_strings are used to send control strings to the output device (i.e., to change fonts or other display characteristics).

The device_string alternatives are listed below.

<table>
<thead>
<tr>
<th>Device_Strings</th>
<th>Typical Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_INITIAL</td>
<td>initialize output device</td>
</tr>
<tr>
<td>SETFONTO</td>
<td>set current font to font 0</td>
</tr>
<tr>
<td>SETFONT1</td>
<td>set current font to font 1</td>
</tr>
<tr>
<td>SETFONT2</td>
<td>set current font to font 2</td>
</tr>
<tr>
<td>SET_LPP</td>
<td>set physical lines per page</td>
</tr>
<tr>
<td>SET_VAR0</td>
<td></td>
</tr>
<tr>
<td>SET_VAR1</td>
<td></td>
</tr>
<tr>
<td>SET_VAR2,</td>
<td></td>
</tr>
<tr>
<td>SET_VAR3</td>
<td></td>
</tr>
</tbody>
</table>

The reason we have labeled these "Typical Usage" is that r-tree® only sends the control strings associated with these names to the output device when your IMAGE section directs r-tree to do so. The SET_INITIAL control string is not automatically sent prior to printing the report, so you can make whatever use you desire of these display strings.
The IMAGE section is ultimately what r-tree® is all about. Without an IMAGE section, no output will be produced. The IMAGE section specifies what each line of the report should look like. The IMAGE section is broken down into the following optional subsections:

<table>
<thead>
<tr>
<th>IMAGE Subsections</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>initialize output device</td>
</tr>
<tr>
<td>PAGE_HDR</td>
<td>printed at the top of each page</td>
</tr>
<tr>
<td>REPORT_HDR</td>
<td>printed at the beginning of the report</td>
</tr>
<tr>
<td>CONTROL_HDR(i)</td>
<td>level i control header</td>
</tr>
<tr>
<td>BODY</td>
<td>printed for each group of selected records</td>
</tr>
<tr>
<td>CONTROL_FTR(i)</td>
<td>level i control footer</td>
</tr>
<tr>
<td>REPORT_FTR</td>
<td>printed at the end of the report</td>
</tr>
<tr>
<td>PAGE_FTR</td>
<td>printed at the bottom of each page</td>
</tr>
</tbody>
</table>

The report layout rules are essentially the same for each IMAGE subsection:

- Each line to be output (referred to as an output line) must appear with a LINE_MARK (default `+`) in the first position. This line mark is not actually printed. It simply serves to tell the report script parser that the line of text should be treated as a literal output string. Comments cannot be placed in a literal output string.

- Within each output line, you may place any number of format specifications. Each format specification begins with a FORMAT_MARK (default `@`). The format specification ends when the first non-format character (including a blank) is encountered or the PCNTRL_MARK (default `^`) is encountered. For now, simply use X, x, or 9 as your format characters. "X" and "x" signify string output, and "9" signifies numeric output. A complete description of the format specifications is presented in the Format Specifications section of Chapter 9.

- After each output line which contains format specifications, there should be a list of values which are to be substituted into the formats (called the format symbol list). Ordinarily, these values are presented as symbolic field names, status variable names (such as PAGE_NO and SYS_DATE), or display_logic variables. The display_logic variables allow for real-time control of the report output.

The display_logic variables are presented below. The column Consume Following Value indicates whether or not the display_logic variable must be followed by an indicator value which is not sent to a format specification. The column Requires Following Format indicates whether or not the display_logic variable can appear without a format specification in the associate output line. A no means that even if
there are no values to output in the output line, you can still use the display_logic variable. A yes means there must be a format still to be satisfied on the output line to ensure that the display_logic variable will be activated.

<table>
<thead>
<tr>
<th>Display_Logic Variables</th>
<th>Consume Following Value</th>
<th>Requires Following Format</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN</td>
<td>yes</td>
<td>yes</td>
<td>Set tab (to following value) for next format specification.</td>
</tr>
<tr>
<td>COND_HDR</td>
<td>yes</td>
<td>no</td>
<td>Print CONTROL_HDR(i) where i is the value following COND_HDR. The header is not printed if r-tree is in the middle of an actual control break.</td>
</tr>
<tr>
<td>COND_PAGE</td>
<td>yes</td>
<td>no</td>
<td>If number of lines specified by the following value will not fit on current page, issue a page break.</td>
</tr>
<tr>
<td>FORM_FEED</td>
<td>no</td>
<td>no</td>
<td>Issue a form feed after printing the line.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>yes</td>
<td>yes</td>
<td>FORMAT permits the printing of a value in the i-th of N format specifications where i is the first value following FORMAT, N is the second value and the actual value to be printed is the third. The other N - 1 format specifications are printed as blanks.</td>
</tr>
<tr>
<td>HORIZONTAL</td>
<td>no</td>
<td>no</td>
<td>Mark first line of BODY as horizontal repeat template.</td>
</tr>
<tr>
<td>LEFT_MARGIN</td>
<td>yes</td>
<td>yes</td>
<td>Set left margin to following value.</td>
</tr>
<tr>
<td>NO_LINEFEED</td>
<td>no</td>
<td>no</td>
<td>Do not output a linefeed at the beginning of this line.</td>
</tr>
<tr>
<td>OUTLINE</td>
<td>yes</td>
<td>yes</td>
<td>Only print field if control field (level given by following value) has changed.</td>
</tr>
<tr>
<td>PAGE_BREAK</td>
<td>no</td>
<td>no</td>
<td>Issue a page break after printing the line.</td>
</tr>
<tr>
<td>PAUSE</td>
<td>no</td>
<td>no</td>
<td>Issue a pause after printing the line.</td>
</tr>
<tr>
<td>RESET_PAGE</td>
<td>no</td>
<td>no</td>
<td>Print report footer, reset page number to 1, and resume printing as though a new report was just beginning.</td>
</tr>
<tr>
<td>RETURN</td>
<td>no</td>
<td>no</td>
<td>Print this line on top of previous line: usually used to create bold images.</td>
</tr>
<tr>
<td>ROWS</td>
<td>yes</td>
<td>yes</td>
<td>Skip down N lines where N is the value following ROWS.</td>
</tr>
<tr>
<td>SKIP_LINE</td>
<td>yes</td>
<td>no</td>
<td>Only print line if value following SKIP_LINE is non-zero. SKIP_LINE must be the first symbol in the symbol list following the output line. The opposite of TEST_LINE.</td>
</tr>
</tbody>
</table>
The best way to create an IMAGE section is to use a text editor to "paint" your report layout without concern for the r-tree® subsection keywords or the format symbol lists. After you are satisfied, you can decide to which subsection each group of report lines belongs, and add the necessary report script information.

*Figure 6-10* contains a report layout (before adding the report script information) for the customer order system example. This layout applies to the report script fragment shown in the discussion on the ACCUMULATOR Section.
Once we are satisfied with the layout, we can put the contents into report script form. In Figure 6-11 we have put this sample layout into a complete report script.

<table>
<thead>
<tr>
<th>START</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRTUAL</td>
<td></td>
</tr>
<tr>
<td>customer_rexer</td>
<td>STRING 32 &quot;custmast.dl&quot;</td>
</tr>
<tr>
<td>item_rexer</td>
<td>STRING 32 &quot;itemmast.dl&quot;</td>
</tr>
<tr>
<td>customer_order</td>
<td>STRING 32 &quot;custord.dl&quot;</td>
</tr>
<tr>
<td>order_line</td>
<td>STRING 32 &quot;ordline.dl&quot;</td>
</tr>
<tr>
<td>extended_price</td>
<td>MONEY 4</td>
</tr>
<tr>
<td>ol_quantity</td>
<td>int_itemprice</td>
</tr>
</tbody>
</table>

| SEARCH |  |
| FILE customer_order USING KEY co_custnum #idx |
| FILE customer_rexer JOIN ON customer_order BY FIELD co_custnum# USING KEY co_custnum #idx |
| FILE order_line IS Detail FOR customer_order BY FIELD co_ordernum# USING KEY co_ordernum #idx |
| FILE item_rexer JOIN ON order_line BY FIELD ol_itemnum# USING KEY ln_itemprice #idx |

| SELECT |  |
| co_orderdate = "09/01/86" AND co_orderdate <= "09/30/86" AND ol_quantity <= 0 |
| UPPER on_custnum |
| NO MOD co_orderdate |
| UPPER ol_ordernum |

| CONTROL |  |
| 1 on_custnum |
| 2 co_orderdate |
| 3 ol_ordernum |

| ACCUMULATOR |  |
| order_total | SUM extended_price |

| DISPLAY |  |
| PAGE_LENGTH 42 |
| TEST_PAGES 2 |

| IMAGE |  |
| PAGE_HDR |
| SYSDATE | PAGE.NO |
| * |  |
| COMD_HDR 3 CONTROL_HDR(3) |

| COMD_PAGE B |  |
| Customer # | "" |
| Order # | "" |
| * |  |
| Customer Name | "" |
| co_ordernum | "" |
| on_custnum# | "" |
| co_custnum # | "" |
| co_orderdate | "" |
| * |  |
| Item | "" |
| Code | "" |
| Description | "" |
| Qty | "" |
| Unit | "" |
| Extended | "" |

| BODY |  |
| "" |  |
| co_custnum# | "" |
| on_custnum# | "" |
| co_ordernum | "" |
| co_custnum # | "" |
| ol_ordernum | "" |
| ol_itemnum# | "" |
| ln_itemprice | "" |
| ol_quantity | "" |
| extended_price | "" |
Control Footers

CONTROL_FTR(i) subsections operate with a subtle limitation: the only data fields that you can reliably substitute into format specifications are accumulators or control fields. The reason is that by the time the control footer is to be output, r-tree® will have already read the next set of data. Therefore, ordinary data fields and virtual fields may contain information pertinent to the next set of data to be output.

Accumulators are referenced by their symbolic names as shown for order_total in the example of the previous section. Control fields do not have symbolic names. The contents of a control field can be referenced by the expression

C\[int2u\]

where int2u is the control field level.

In Figure 6-12, if we wanted the date to be in the data subtotal footer, we would change the CONTROL_FTR(2) subsection to that shown in Figure 6-11. In Figure 6-11, C[2] contains the appropriate date. If, instead of C[2], we had used co_ordrdate in the format symbol list, the date printed would be for the customer order record just read, but not yet processed in the BODY subsection.

Figure 6-11. Control Field Contents.

Figure 6-12. Control Field Contents.
Because of this limitation, you may find it useful to use the LST accumulator to output descriptive information in a control footer. For instance, you may want control breaks every time a customer number changes. However, if you wish to print out the customer name in the control footer, you should define an accumulator based on the last value in the customer name field. The ACCUMULATOR and CONTROL_FTR(1) portions of the script would be changed as shown in Figure 6-13.

```
<table>
<thead>
<tr>
<th>ACCUMULATOR</th>
<th>SUM</th>
<th>extended_price</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer_name</td>
<td>LST</td>
<td>on_customer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+-----------------+
<table>
<thead>
<tr>
<th>CONTROL_FTR(1)</th>
</tr>
</thead>
</table>

```

**Figure 6-13. LST Accumulator In Control Footer.**

Arithmetic Conversions

**r-tree** provides a wide array of numeric data types including a monetary data type which combines the speed and compactness of (long) integer arithmetic with two decimal places of precision. It is important to understand how these types are converted during arithmetic operations, logical comparisons, and virtual field assignments.

All numeric types less than four bytes (CHAR, CHARU, INT2 and INT2U) are converted to a signed long (INT4) before being used for computations or comparisons. Four-byte reals are converted to eight-byte reals before use. In general, operations are performed using long integer arithmetic when neither operand is a real. If one or both operands are real, then the operations are performed using double precision float operands.

The only exceptions to these rules apply to monetary data where the implied decimal location would not be maintained. In particular, if two MONEY types are multiplied or divided, they are first converted to eight-byte reals with the proper decimal places.

In virtual field assignments, longer integers are truncated on the left (the most significant bytes are lost) when they are assigned to shorter integers. Integers and money types are assigned without any scale factor. That is, an integer with a value of 100 becomes a money type with a value of one dollar (or 100 cents). In the following:

```
VIRTUAL
    test    MONEY    4    100
```
test will be assigned a value of one dollar; whereas in this virtual field:

```
VIRTUAL
  test  MONEY  4  $100
```

test will be assigned a value of one-hundred dollars.

SUM and AVG accumulators are maintained as double precision reals regardless of
the underlying numeric type. MIN, MAX, FRS, LST and PRV are maintained as double
precision reals if the underlying type is numeric; otherwise they are maintained as
strings. FRQ accumulators are always maintained as long integers. See Figure 6-14 for
eamples of accumulator storage classes.

| VIRTUAL   | number_field | INTZ 2 S + B + C |
| SUM       | string_field | STRING 10 D = E = F |
| ACCUMULATOR | sum_number  | SUM number_field /* double */ |
|           | sum_string   | SUM string_field  /* double */ |
|           | freq_number  | FREQ number_field /* long integer */ |
|           | freq_string  | FREQ string_field /* long integer */ |
|           | max_number   | MAX number_field  /* double */ |
|           | max_string   | MAX string_field  /* string */ |

Figure 6-14. Accumulator Storage Class Examples.

The easiest place for mistakes to occur, because of a misunderstanding of inter-type
conversions, is with integer and money fields and constants. If an integer and money
field are added or compared, r-tree® assumes that the integer field is already in money
form (that is, has accounted for the pennies), and does not multiply by one-hundred to
put it in dollar terms. Therefore,

```
im_itempric > $100
```

is completely different from

```
im_itempric > 100
```

since in the latter, 100 is equivalent to one dollar.

If integer and money fields are multiplied, as in

```
oi_quantity * im_itempric,
```

then we assume that the integer has no decimal places. For example, if in the above
product, oi_quantity is 100 and im_itempric is $10, the product is a monetary result
with a value of $1000.
**String Comparisons**

When `r-tree` compares two string valued quantities, it automatically pads the shorter of the two strings with ASCII blanks (i.e., decimal 32 bytes). Therefore, the following two strings are considered equal by `r-tree` when used in boolean expressions:

```
"abc"     "abc   
```

If you have defined an entry in your data object definition array (DODA) to be a string of a certain length, but whose actual length may vary, then be sure that there is a null byte terminating any value which does not achieve the full length of the string.

See the Advanced Concepts Chapter for details of SUBSTRING and WILDCARD functions in string comparisons.

**String Concatenation**

The concatenation operator (#) allows two string-valued fields to be concatenated.

To permit very general application of this operator, null bytes ARE NOT used to determine the end position of the first operand. Therefore, even if the left-hand operand is null terminated, the right-hand operand will be joined to the very end of the left-hand operand as defined by its DODA or VIRTUAL field length.

Use the STRIP built-in function on the left-hand operand to ensure that the operand is stripped of trailing blanks and/or does not extend past a null byte. Figure 6-15 demonstrates the affect of the STRIP function during concatenations.

```sql
VIRTUAL
str1 STRING 10 "abc"
str2 STRING 3 "def"
str3b STRING 5 STRIP(str1) # str2
str3b STRING 5 STRIP(str1) # str2
/*
  will lead to the following results:
  */
  * str3a = "abc   
  * str3b = "abcdef"
```

**Figure 6-15. String Concatenation.**

The key to this example is the definition of `str1` as 10 characters. `str1#str2` carried out to 13 characters yields "abc def", which is then truncated to 5 characters when it is assigned to `str3a`. `STRIP(str1) # str2` yields "abcdef", which is then truncated to 5 characters when it is assigned to `str3b`. 
This page intentionally left blank.
7. r-tree Operation

Once you have created the DODA, modified the ISAM file definitions (incremental ISAM structure or parameter file), and created a report script, you are ready to integrate r-tree with your application. This chapter discusses how to:

- check your report script syntax;
- invoke r-tree in one of several modes, and;
- link r-tree to your application.

7.1 r-tree Library

To facilitate maintenance, r-tree is comprised of many code modules. Many C compilers are supplied with an object module library manager. Once you have compiled your r-tree modules, you will find it convenient to combine the object modules into a r-tree object library. The make files supplied with r-tree automatically build such r-tree libraries and perform the r-tree application linking.

The r-tree library should contain all the r-tree modules except those used to establish the type of front-end interface (and of course the library does not contain the sample programs). Therefore, the library should be made up of the following modules in the order shown:

1. RTPARS 10. RTLINE
2. RTLEXS 11. RTLIN2
3. RTRED1 12. RTRUNT
4. RTRED2 13. RTSPR2
5. RTRUN2 14. RTUSER
6. RTCUT 15. RTDATE
7. RTLEVEL 16. RTDEV
8. RTREAD 17. RTSPRT
9. RTREA2

The following modules will NOT be in the r-tree library:

```
RTCMPL       RTTIM       RTSAMPLI
RTCTST       RTSYNT      RTSAMPLP
RTINTR       RTTEST      RTDRVR
RTMMRY       RILJET      RTCTDV
```

For the rest of this chapter we assume that you have compiled all the r-tree modules and created an r-tree library named rtailib.lib. Also, we assume that you already have created a c-tree library called ctailib.lib. We assume that the compiler’s run-time AND
floating point libraries are automatically included in the link. Object modules are shown
with .OBJ file extensions. rttest.obj contains the object code for the customer order
equation that we have been using throughout this manual. rectt.obj is almost identical
to rttest.obj except that it calls on the script compiler function rcompiler instead of the
report function, as described later.

7.2 Syntax Checker

rsynt.c is a syntax checker that can be used to ensure that an r-tree® report script
conforms to the syntax of the r-tree® statements. When developing a new report script
this can be used to quickly check its structure. The syntax checker does not utilize the
data object definition array (DODA). Therefore, it does NOT check to see if symbolic
references actually exist, nor does it check for compatibility of operands in expressions
since it does not have any field type information.

The reason we do not use the DODA in the syntax checker is that you can create a
self-standing syntax checker which is application independent. Simply compile the
module rtsynt.c supplied on your distribution disk, and link it to the r-tree® library, and
you have an executable utility program that can be used on any report script for any
application. The modules to link are:

    rtsynt.obj  rtalib.lib

If the syntax checker detects a syntax error, it will print out the line of the script
containing the error with an up-arrow pointing to the offending item, as well as the two
lines preceding the error. Sometimes the offending item is indeed incorrect, but, just
as often, it is a mistake made before the offending item which actually causes the
problem. The reason the "mistake" is not caught by the syntax checker is that the
"mistake" does not violate the r-tree® syntax.

To run the syntax checker, simply invoke the checker with the name of the report script
file as the command line argument. For example:

    rtsynt tscript

would cause the report script in the file tscript to be syntax checked. Figure 7-1 presents
sample output from the syntax checker.

<table>
<thead>
<tr>
<th>★★★  R-tree® Syntax Check OF Tscript ★★★</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE 28:  ★★★ ILLEGAL SYNTAX ★★★</td>
</tr>
<tr>
<td>CONTROL</td>
</tr>
<tr>
<td>order_summary      SUM     extended_pric</td>
</tr>
<tr>
<td>SYNTAX CHECK TERMINATED WITH ERROR CODE 3B1</td>
</tr>
</tbody>
</table>

Figure 7-1. Sample Syntax Checker Output.
As discussed above, the real error may not be with the token `order_accum`, but with some earlier mistake. In this case, line 28 of the script was mistakenly preceded by the keyword `CONTROL` instead of the correct keyword `ACCUMULATOR`. In other words, line 28 is actually correct, but line 27 must be corrected.

### 7.3 The Report Function

**r-tree** provides complete report preparation with a minimum of C programming. There is only one **r-tree** function to call in order to generate reports. This function, called `report`, requires four parameters:

- pointer to the report script (file name);
- pointer to the data object definition array;
- pointer to an optional message to be displayed as soon as control is transferred to the report function; and
- pointer to an optional programmer supplied function which, if present, will be called after each major **r-tree** iteration.

The **r-tree** `report` function is defined as:

```
COUNT report(rsp, doda, message, userfn)
```

`rsp` points to the report script, `doda` points to the data object definition array, `message` points to the optional banner, and `userfn` is a pointer to the optional programmer supplied function. `report` returns a zero for successful completion, a one if the report is aborted in the optional function pointed to by `userfn`, one of the error codes from the header file `rterrc.h`, or a `c-tree` error code from the file `citerrc.h`.

#### Error Returns

With only two exceptions, when an error occurs, **r-tree** immediately terminates processing and `report` returns the appropriate error code. The exceptions are insufficient space in a numeric format and division by zero. When a numeric format is too small, the format is filled with question marks in the output, and the report function returns error code `PNFM_ERR` (343) upon completion of the report. If division by zero is attempted, the numerator remains unchanged, and the report function returns error code `PDVZ_ERR` (346) upon completion of the report. If both problems occur, then the report function returns `PDVZ_ERR`. In any event, the global short integers `nfmerr` and `ndverr` will be set to non-zero values if there are numeric format errors and/or divide by zero errors, respectively.
To provide maximum design flexibility, and a wide range of control for the application programmer, \texttt{r-tree} permits report scripts to be passed to \texttt{r-tree} in three different forms:

- as a text file;
- as a text memory image; and
- as a compiled script.

\textbf{Text File} \hspace{1cm} When a text file is used, \texttt{rsp} points to the name of the file containing the report script. The advantage of the text file approach is that any text editor can be used to modify the report script or create new scripts. If you do not want the end users to modify the scripts or create their own, do not use the text file approach.

\textbf{Memory Image} \hspace{1cm} When a text memory image is used, \texttt{rsp} points directly to a null terminated character string in memory which contains the report script. This approach is useful when you want to create an end-user friendly front-end for \texttt{r-tree}; once the front-end gathers the user request, you create a report script in memory, and pass a pointer to it to the report function.

\textbf{Compiled Script} \hspace{1cm} When a compiled script is used, \texttt{rsp} points to the name of the file containing the compiled report script. Compiling report scripts is described later. A compiled report script cannot be changed without modifying the original text file script and recompiling. Therefore, compiled scripts provide a means to ensure that end users cannot modify or create new scripts.

\textbf{Data Object Definition Array Pointer (doda)}

The data object definition array is discussed in Chapter 4. Once you have created this array, \texttt{doda} is simply the name of the array.

\textbf{Optional Message Pointer (message)}

If you wish \texttt{r-tree} to display a message as soon as control is passed to the report function, then \texttt{message} should point to the desired text. If you do not want a message displayed, then pass a null text pointer (NULL) for the \texttt{message} parameter.

\textbf{Optional User Function Pointer (userfn)}

The last parameter of the report function permits you to specify a function to which control will be transferred after each major \texttt{r-tree} iteration. By major \texttt{r-tree} iteration we mean the evaluation of the SELECT expression and the output of a report line. In addition, control will be transferred to this function whenever \texttt{r-tree} evaluates the INTERFACE built-in function.
The purposes of this user-supplied function are to provide the abilities:

- to abort or otherwise interrupt the report function;
- to capture the report output for processing external to r-tree®; and/or
- to perform special processing whenever the INTERFACE function is evaluated.

See rltjet.c on your distribution disk for an example of INTERFACE processing.

If you want such capabilities, pass the name of the function for the userfn parameter. If you do not wish to utilize such a function, pass a null function pointer (NULLfp) for the userfn parameter (NULLfp is defined in rtdoda.h). Figure 7-2 outlines how a user supplied function should be organized:

```c
int your_func_name(mode, tp, tlen)
int mode; /* tells where call came from */
char *tp; tlen;
{
    switch (mode) {
    case LOOP_SEL: /* transfer from record selection */
    case LOOP_SRT: /* transfer from sorted record read */
        /* tp and tlen are undefined. */
        /* Test to see if you want to abort report. */
        /* Before outputting a prompt, be sure to call */
        /* the put_list() routine. */
        /* To abort, return a non-zero value. To continue, */
        /* report return a zero. */
        return(0);
    case LOOP_IMG: /* transfer from line output */
        /* tp points to output image with length tlen. */
        /* PROCESS IMAGE HERE */
        /* Return a 1 for r-tree NOT to output image. */
        /* Return a 0 for r-tree to output image. */
        /* */
        /* NOTE: */
        /* The output line DOES NOT contain any */
        /* device control strings or any padding */
        /* for the left margin. */
        return(0);
    case LOOP_IMP: /* transfer from INTERFACE builtin function */
        /* tp points to local copy of INTERFACE input */
        /* argument with length tlen. */
        put_list();
        /* INTERFACE PROCESSING HERE */
        /* Return value is not used by INTERFACE. */
        return(0);
    }
}
```

Figure 7-2. User Supplied Function.
It is important to note that the user supplied function may be called from several places. Test the value of `mode` to determine at what point the function has been invoked. The values are:

**LOOP_SEL**
Each time a record has passed the selection criteria the user defined function will be called.

**LOOP_SRT**
The records are sorted after the selection process. Each time a record is read to be sorted, the user defined function is called.

Typically you are going to use **LOOP_SEL** and **LOOP_SRT** just to see if the user wants to abort the report.

**LOOP_IMG**
Just before a line is to be output the user defined function will be called. You can do any additional processing to the output line (as pointed to by `tp`) if you wish, or you can cancel the output line.

**LOOP_INP**
This mode is the result of a call to the INTERFACE built-in function and provides a way for adding your own functionality on a case-by-case basis. See "Built-in-function" in the Index for a method of adding functionality to r-tree® in general. The file rthints.doc supplied on r-tree®’s distribution disk contains an example accessing the scripts ACCUMULATOR and VIRTUAL values from the user supplied function.

NOTE: the r-tree® example program rtest.c, also on r-tree®’s distribution disk contains a sample user supplied function. However, since it must interface to the keyboard, it is somewhat hardware-dependent. So be aware!

### 7.4 Program Structure

In most situations, you will use a very small C program to invoke r-tree®. (One exception is the syntax checker discussed below which does not require any of your own interface code.) The typical structure of such an interface program follows.

```c
/* File: rtest.c */

#include <stdio.h>
#include "rtdsk.h"

struct {? . . .} A;
struct {? . . .} B;

/* Data Object Definition Array */

DMTRDS ds[I] = {
    "name", m_first, type_1, length_1,
    . . .
    "name", m_first, type_n, length_n,
    . . .
};

/* Optional User-supplied FUNCTION */
int loop_check(mode, tp, tlen)
```
As discussed earlier, there are three modes for processing report scripts: as a text file, as a memory image, and as a compiled script. The structure of your program is the
same for all, with a minor variance in the report function and how you link your application.

**Text File**

Report scripts in text files may be interpreted, producing complete reports, providing no errors are discovered while analyzing the script. Before attempting to produce a report from a report script in text form, r-tree® subjects it to a rigorous analysis which will find any syntax errors, and most potential run-time errors. If a syntax error is discovered, then output as described in the previous section will be sent to the screen. If a potential run-time error is discovered (such as an undefined symbolic reference or mixing incompatible operands in an expression), r-tree® will report the nature of the error and the line in which it occurred.

For example, if we correct the error on line 27 described in the previous section, and attempt to interpret the report script, we might get the following output:

```
LINE  28: *** SYMBOL UNDEFINED ***
extended_price
```

In this case, we have mistyped the virtual field symbolic name which should have been `extended_price`. Hence the accumulator definition, while syntactically correct, was in error due to an undefined symbol.

Note that corrections to the report script are accomplished without changes to the C code. No compilations or relinks are required.

To create an interpreter for report scripts stored in text files, create an interface program as described above. Pass the name of the report script as the `rsp` parameter. Then link the object version of this program with the object version of `rtintr.c` (supplied on your distribution disk), the r-tree® library, and your c-tree® Plus library. In order for the resulting executable code to be more useful, be sure your small interface program permits you to change the report script file name without any recoding. We supply you with a test program, `rttest.c`, that can be used to test r-tree®. Link the following modules:

```
rttest.obj, rtintr.obj, rtalib.lib, ctalib.lib
```

Your executable report script interpreter is tied to a particular data object definition array (DODA). Virtually any report based on the particular DODA can be generated from the interpreter without additional C coding or compiling. However, to change the DODA, you must recompile the interface program, and create a new report script interpreter.
Memory Image

To interpret report scripts stored in memory, modify your small interface program so that report scripts are somehow constructed or placed into memory. Then link as shown above for a report script, except that \texttt{rtntr.c} is replaced by \texttt{rtmmyr.c} (also supplied on your distribution disk), and the \texttt{rsp} parameter will point to the buffer in memory that contains the script. If you place a script in memory (instead of using a script file) be sure that image lines are preceded and followed by new line characters (\texttt{\textbackslash n}). Otherwise, the interpreter will not identify the image lines which will result in syntax errors.

Compiled Script

Once you have prepared a report script and used an interpreter (as outlined above) to be sure that the report script produces the output you require, then you may compile the script to produce an “object” form of the script. The object form of the script is usually about 800 bytes larger than the text form. However, the compiled script requires less code for processing and starts more quickly since the rigorous analysis is already completed.

The first step is to compile the script. This is done with your application program. The only difference between preparing an \texttt{r-tree} compiler tied to your application and an \texttt{r-tree} interpreter is that instead of calling the report function, you call:

\begin{verbatim}
rtcompiler(rsp,doda,crsp)
\end{verbatim}

\texttt{rsp} points to the name of the file containing the report script text; \texttt{doda} points the data object definition array; and \texttt{crsp} points the file name for the compiled script.

If the file name pointed to by \texttt{crsp} already exists, then \texttt{rtcompiler} will overwrite the file.

\texttt{rtcompiler} returns a zero upon successful compilation, or an error code found in \texttt{rterrc.h}. Note that if an error is detected, the same diagnostic messages produced by the syntax checker and interpreter are output by the compiler.
A successful compilation of the report script "tscript" with "tscript.cmp" designated as the compiler output file will produce output similar to that shown in Figure 7-4.

```
*** r-tree(TM) Report Script Compilation Of tscript ***
Copyright 1995 FairCom
Report Script Entry Completed...
Parse Phase 1 Completed...
Parse Phase 2 Completed...
File Traverse & Image Analysis Completed...
Begin Compiler Output...
Output OF tscript.cmp Completed
*** SUCCESSFUL r-tree(TM) COMPILATION ***
```

Figure 7-4. Sample r-tree Compiler Output.

We supply you with the program `rtctst.c`, which is the same program as `rttest.c` discussed earlier, with the exception that it calls `rcreompiler`. Link the following modules:

```
rtctst.obj, rtcmpl.obj, rtalib.lib, ctalib.lib
```

**Run-Time Processor**

Once a report script is successfully compiled, it is executed with the same interface program that you used to interpret the report script. The difference is that instead of linking the interface program with the interpreter module `rtintr.c`, you link it to the object version of `rtrtim.c`. Link the following modules:

```
rttest.obj, rtrtim.obj, rtalib.lib, ctalib.lib
```

Because of the nature of the compiled report script, you cannot change your data object definition array once the script is compiled. If the r-tree® run-time detects a change, it will issue an error message, and not process the compiled script. Further, you cannot change the size of a pointer to data objects between the r-tree® compiler and the r-tree® run-time program. For example, if on an 8086 compatible machine (viz., 8088, 8086, 80186, 80286, 80386 . . .) you used a large memory model (4-byte code addresses and 4-byte data addresses) to create your r-tree® compiler, you could not use the medium model (4-byte code addresses and 2-byte data addresses) for the run-time processor. Again, an error message will be issued if the size of a data address changes between compilation and execution.
8. Advanced Concepts

r-tree® provides a number of very powerful features which permit record selection criteria and virtual field computation to go far beyond ordinary boolean and computational expressions. These advanced features as well as "programming" techniques for virtual variables are discussed below. The basis for many of these features is the r-tree file hierarchy. Do not let the term "file hierarchy" mislead you; through the use of file aliases discussed later in this chapter, an r-tree file hierarchy can represent a wide array of file interrelationships.

8.1 File Hierarchy

It is important to understand the concept of an r-tree file hierarchy. The r-tree SEARCH section defines the file hierarchy. The first file listed in the SEARCH section is the primary file; it is at the top of the file hierarchy. Each file listed after the primary file is a subsidiary file. Each subsidiary file must be related through the "JOINS_TO" or "IS_DETAIL_FOR" clause to a file preceding (but not necessarily immediately preceding) it in the SEARCH section.

A file hierarchy is represented as an inverted tree structure with the root on top. The r-tree primary file occupies the root node of the file hierarchy tree. This is considered to be level zero of the hierarchy.

JOINS_TO Relationship

If file B "JOINS_TO" file A, then files A and B occupy the same position (i.e., node) in the file hierarchy. It is as though the files have become one. Any number of files may be joined together in this way.

Note that any file which "JOINS_TO" the primary file is also considered to be at the top of the file hierarchy.

IS_DETAIL_FOR Relationship

If file B "IS_DETAIL_FOR" file A, then file B is considered to be one level below file A in the file hierarchy. If file C also "IS_DETAIL_FOR" file A, then file C is also considered to be one level below file A; but files B and C do not occupy the same position in the file hierarchy. They occupy parallel positions or sibling nodes. Note: see Record Selection at the end of this section for possible limitations to "non-simple" paths. r-tree permits up to eight sibling nodes below any node in the hierarchy. Also, r-tree supports up to six different levels in the file hierarchy.
Sample File Hierarchy

Consider the following file relationships:

- File p is the primary file
- File q JOINs TO p
- File r ISDETAIL FOR q
- File t JOINs TO s
- File u IS_DETAL_FOR p
- File v IS_DETAL_FOR u
- File w IS_DETAL_FOR t
- File x JOINs TO v
- File y IS_DETAL FOR u

Based on the rules for JOINs_TO and IS_DETAL_FOR, these file relationships produce the following file hierarchy:

```
[p q r]  
/   \  
|     |   
[\ s t \]  [\ u \]
       /   
[\ w ]  [\ v x \ y ]
```

In general, the only relationship between different nodes at the same level of the hierarchy (e.g., between \{s t\} and \{u\}) is expressed through the parent node. For example, a record from \{p q r\} is related to a set of records in \{s t\} and a set of records in \{u\}. The only thing these two sets have in common is the parent record from \{p q r\}. If there exists a one-to-one correspondence between the records from \{s t\} and the records from \{u\}, they should have been joined together instead of forming two separate nodes in the file hierarchy.

Detail Record Set

The r-tree® SEARCH statement as illustrated in **Figure 8-1**

```
FILE subsidiary IS_DETAL_FOR parent 
BY_FIELD parent_field USING KEY subsidiary_key
```

*Figure 8-1.*
specifies how a record in a parent file relates to a set of records in the subsidiary file. Specifically, **parent_field** is the symbolic name of a field from the parent file (or for a virtual field derived from such a field). The **subsidiary_key** is the symbolic name of an index defined for the subsidiary file. Whenever **r-tree** must read a set of subsidiary records corresponding to a given parent record, **r-tree** performs as follows:

- the value in the **parent_field** is treated as the leading segment of the target key used to search the **subsidiary_key**;
- **r-tree** issues a FRSSET (see c-tree documentation) with this leading segment to read the first record in the set of associated records (which matches the leading segment); and
- **r-tree** issues subsequent NXTSET’s (see c-tree documentation) to read the complete set of associated records.

In our customer order example, the order number field of a customer order record serves as the leading key segment to search the order number index of the order-item detail file.

### Sample Detail Records

**Master Customer Order Record**

<table>
<thead>
<tr>
<th><strong>parent_field</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>order number</td>
</tr>
</tbody>
</table>

**subsidiary_key**: order-number index of the order-item data file links each customer order to all order items with matching order numbers. Note that the index is based on order number and sequence number, but only the leading segment must match the parent field.

**Detail Order Item Records**

<table>
<thead>
<tr>
<th>order number</th>
<th>item number</th>
<th>sequence #</th>
<th>quantity ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>order number</th>
<th>item number</th>
<th>sequence #</th>
<th>quantity ordered</th>
</tr>
</thead>
</table>

Programmer’s Reference Guide  Chapter - 8 81
Record Selection

Given a file hierarchy, r-tree® restricts your record selection criteria to a simple path down the hierarchy, always rooted at the primary file (level 0). In the example, this means that you could select records from:

- level 0 {p q r}
- level 0 and 1 {p q r {s t}} or {p q r {u}}
- level 0, 1 and 2 {p q r {s t {w}}}; {p q r {u {v x}}}; or {p q r {u {y}}}

where the nested braces represent the parent/subsidiary relationship. Notice that we do not select records from two different nodes at the same level of the hierarchy. Selecting records from {p q r {u}} means that for each record in {p q r}, we examine all associated records in {u}; any {p q r} record and associated {u} records which meet the SELECT criteria are selected.

While the selected records must fall on a simple path down the hierarchy, the set functions discussed below permit records from all nodes of the file hierarchy to be evaluated while selecting particular records on the simple path.

8.2 Set Functions

If your application involves a file hierarchy with more than one level, you may find the r-tree set functions very useful. A set function causes r-tree to read a set of related detail records, and compute a result based on the set of records. Set functions will NOT read a set of records from the primary file (or any file joined to the primary file with the JOINS_TO clause).

The primary use of the set functions is to permit record selection criteria which go beyond values found in individual records. For example, in our customer order system, we may want to select customer orders which:

- have ten or more items on the order; or
- have a total order price in excess of $100; or
- only have items on the order with a unit price below $20.

The set functions allow customer order selection based on information which spans the associated order-item detail records.

r-tree® permits set functions to be nested in a manner which is consistent with the underlying file hierarchy. Set functions cause records of the file hierarchy to be read.
Nested functions cause records from more than one level of the file hierarchy to be read. Keep in mind that \texttt{r-tree} allows the file hierarchy to be traversed once for the purpose of computing set functions, and then again to see which records may be selected. \texttt{r-tree} will not support set usage if two passes over the file hierarchy are required to compute the set functions. \texttt{r-tree} automatically detects this situation.

### Computational Set Functions

\texttt{r-tree} supports seven computational set functions. The argument of each computational function must be a computational expression, and may also include nested computational set functions (wherever a value might occur in a computational expression). Except for \texttt{FREQUENCY}, the argument of the function is evaluated for each record in the set of records in the detail file, and the function is applied to the set of values. The functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQUENCY</td>
<td>counts number of detail records in a set</td>
</tr>
<tr>
<td>SUMMATION</td>
<td>sums expression over the records</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>finds the average of the expression</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>finds the minimum of the expression</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>finds the maximum of the expression</td>
</tr>
<tr>
<td>FIRST</td>
<td>saves first occurrence of the expression</td>
</tr>
<tr>
<td>LAST</td>
<td>saves last occurrence of the expression</td>
</tr>
</tbody>
</table>

The syntax of a computational function is:

\[
\text{function [ cmpexp ]}
\]

where \texttt{cmpexp} is a computational expression made up of symbolic values, constant values, operators (\texttt{*,/,+,-,#}), parentheses, ordinary functions and other computational set functions.

**NOTE:** Unlike the accumulator functions \texttt{SUM} and \texttt{AVG}, \texttt{SUMMATION} and \texttt{AVERAGE} cannot be called for string valued expressions.

### Boolean Set Functions

\texttt{r-tree} supports two boolean set functions. The argument of each function must be a boolean expression which can include nested references to other boolean set functions.
Anywhere a boolean expression can be used, a boolean set function can be used. The functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXISTS</td>
<td>evaluates to true if at least one record in the set of detailed records makes the argument (boolean expression) true; and</td>
</tr>
<tr>
<td>FORALL</td>
<td>evaluates to true if all the records in the detailed record set make the argument (boolean expression) true.</td>
</tr>
</tbody>
</table>

The syntax of a boolean function is

```
function [ bolexp ]
```

where `bolexp` is a boolean expression made up of computational expressions (including computational set functions), relational operators (=, <> , <, <=, >, >=), boolean operators (NOT, AND, OR), parentheses and other boolean set functions.

NOTE: the NOT operator can be used with EXISTS and FORALL to expand their expressive power.

**Set Function Example**

The customer order example defined earlier will serve as the basis for our set examples. Imagine that, along with the data object definition array for the example, we have the report script fragment shown in Figure 8-2.

```
VIRTUAL
  cm STRING 32 path #"custmast.dat" // customer master
  im STRING 32 path #"itemmast.dat" // item master
  co STRING 32 path #"custordr.dat" // customer order
  oi STRING 32 path #"orditem.dat" // order-item
  extended_price MONEY 4 im_itemprice = oi_quantity

SEARCH
FILE cm ALL
FILE co IS_DETAIL FOR cm BY_FIELD cm_custnum
USING KEY cm_custnum_idx
FILE ci IS_DETAIL FOR co BY_FIELD co_ordnum
USING KEY co_ordnum_idx
FILE im JOIN TO ci BY_FIELD ci_itemnum
USING KEY im_itemnum_idx
```

Figure 8-2. Search Section For Set Examples.
The file hierarchy corresponding to the SEARCH section looks like:

```
{customer master} Level 0
  | {customer order} Level 1
  | {order-item / item master} Level 2
```

Below we show how set functions can be used to satisfy various selection criteria. (The underlined entity is what we are trying to select.)

- **customers with at least ten orders**: count the number of customer order records for each customer, and select those customers with at least ten orders
  
  \[
  \text{FREQUENCY}[\text{co}_{-}\text{ordrnumb}] \geq 10
  \]

  Note that we could have used any data field from the customer order data file since FREQUENCY simply counts the number of records in the detailed record set.

- **customer orders worth over 1000 dollars**: multiply the unit price times the quantity ordered and sum over each order-item detail record, and select those orders with a total in excess of 1000
  
  \[
  \text{SUMMATION}[\text{oi}_{-}\text{quantity} \times \text{im}_{-}\text{itempric}] > 1000
  \]

  Notice how we have taken advantage of the joining of order-item records and item master records to compute an extended price. It is as though both fields were part of the same logical record.

- **customers with at least ten orders, and no order worth more than 1000 dollars**
  
  \[
  \text{FREQUENCY}[\text{co}_{-}\text{ordrnumb}] > 10 \text{ AND } \text{NOT EXISTS } [ \text{SUMMATION}[\text{oi}_{-}\text{quantity} \times \text{im}_{-}\text{itempric}] > 1000 ]
  \]

- **customer orders worth more than the average customer order**
  
  \[
  \text{SUMMATION}[\text{extended}_{-}\text{price}] > \text{AVERAGE}[\text{SUMMATION}[\text{extended}_{-}\text{price}]]
  \]

  This boolean expression **WILL NOT WORK** because it requires the hierarchy to be traversed twice to compute the set functions (once to find the average over all orders, and once to find the individual order prices). With file aliases (see File Alias section in this chapter) you can use a file in more than one location of the hierarchy, which may be useful in attempting the above selection. An alternative is to use two report scripts as discussed later.
• customers with at least one order which contains items all priced over $100

   EXISTS [ FORALL [ im_itempric > $100 ] ]

Please note that the FORALL [] expression finds orders that satisfy the item price requirement and the surrounding EXISTS finds customers that have at least one such order. When r-tree selects a customer which satisfies the above expression, it does not know which order(s) satisfied the FORALL clause. If you wish to examine customer orders with all items priced over $100, then simply use the expression FORALL(im_itempric > 100).

Using Set Functions With The Primary File

As noted earlier, set functions cannot be used with the files at the top (i.e., root) of the file hierarchy. The first file in the SEARCH section (i.e., the primary file) and any files joined to the first file are at the top of the hierarchy. However, if you use a dummy file with only one record for the primary file, the actual primary file can be specified as a detail file for the dummy file using an empty field as demonstrated in the Search Syntax section of this chapter (By “empty” field, we mean a virtual STRING field of length zero). When a detail file is linked to its parent file through an empty field, all the records of the detail file are associated with each record of the parent file.

Instead of using the set functions on the primary file, it may be better to call the report function once to save in a disk file the summary information for which the set functions were originally desired; and then call the report function a second time to produce the actual report output.

8.3 Virtual Fields

The two most powerful features of r-tree are its virtual fields and its set functions. Virtual fields allow information which is derived from application data fields (or data objects) to be treated as though it actually is resident in the data. Not only can virtual fields be derived from data fields, but through the use of set functions, virtual fields also can convey information spread over a set of detail records.

There are four types of virtual field definitions. All begin with the same three items:

   virtual_field_symbolic_name field_type field_length

where the field_type alternatives are: CHAR, CHARU, INT2, INT2U, INT4, INT4U, SFLOAT, DFLOAT, STRING, DATE, or MONEY; and field_length is a positive constant.

A simple virtual field definition ends with a computational expression:

   cmpexp
A boolean virtual field ends with an expression of the form

\[ \text{bolexp} \ ? \ \text{cmpexp} : \ \text{cmpexp} \]

where the first computational expression is used if the boolean expression is true; otherwise the second computational expression is evaluated.

An overlay virtual field definition ends with an expression of the form

\& symbol { + cmpexp | - cmpexp }

where the optional expression computes a byte offset from the beginning of the field following the address operator &.

A coded virtual field definition ends with the following

\[ : \ \text{cmpexp} \ \text{int2u cvalue} \ \{ \text{int2u cvalue} \ ... \} \]

The computational expression is evaluated and the result compared with the small positive constants. If a match is found, then the virtual field is assigned the corresponding constant value. If no match is found, then the field is assigned the last constant value in the list.

**Virtual Field / File Hierarchy Analysis**

Virtual field definitions can include computational and/or boolean set functions wherever ordinary expressions of the same type can be used.

When a virtual field is computed without the use of set functions, then you can associate the virtual field with those nodes of the file hierarchy from which the components of the virtual field definition are drawn. For example, using the file hierarchy shown earlier, we note that the virtual field

\[ \text{extended_price} \ \text{MONEY} \ 4 \ \text{oi_quantity} \ * \ \text{im_itemprice} \]

is associated with level 2 of the file hierarchy since both fields used to compute \text{extended_price} come from level 2.

When a virtual field definition includes data objects from more than one level of the file hierarchy, the virtual field is associated with the lowest level.

When a virtual field includes set functions, the identification of the associated file hierarchy position is not obvious. Using the same example situation, note that the virtual field

\[ \text{order_price} \ \text{MONEY} \ 4 \ \text{SUMMATION[oi_quantity} \ * \ \text{im_itemprice]} \]

Programmer’s Reference Guide  Chapter - 8 87
is actually associated with level 1 of the file hierarchy since the summation is performed over the level 2 file.

When a file hierarchy has more than one branch emanating down from a node, \texttt{r-tree}\textsuperscript{®} places certain restrictions on how field values may be combined. Recall the file hierarchy from the File Hierarchy section at the beginning of this chapter.

\[ \begin{align*}
\{p, q, r\} & \quad \text{Level 0} \\
\{s, t\} & \quad \text{Level 1} \\
\{w\} & \quad \text{Level 2} \\
\{u\} & \\
\{v, x\} & \\
\{y\} &
\end{align*} \]

\texttt{r-tree} will not allow a virtual field to combine values from fields which do not fall on a simple path down the hierarchy.

For instance,

- you could multiply a data field from \{s, t\} times a data field from \{p, q, r\};
- you could add a data field from \{p, q, r\} to a data field from \{v, x\};
- you could NOT add a data field from \{v, x\} to a field from \{y\} because they do not fall on a simple path down the hierarchy; and
- you could NOT multiply a data field from \{y\} times a field from \{s, t\}.

Set functions somewhat relax this conflict because they raise the associated file hierarchy node up one level for each (nested) set function. For example, the MINIMUM over a set of records from \{v, x\} is actually a value associated with \{u\}. Therefore,

- you could add the MINIMUM[\{vx\}] to the MAXIMUM[\{y\}] since the latter is also associated with \{u\}; and
- you could add the result of SUMMATION[AVERAGE[\{y\}]] (which is associated with \{p, q, r\}) to a data field from \{w\}.

\textbf{When Are Virtual Fields Computed?}

In order to provide efficient processing, \texttt{r-tree} only recomputes a virtual field value when data are read from the file at the lowest level node in the file hierarchy associated with the virtual field. This approach does not cause problems for virtual fields that depend on more than one level of the hierarchy since \texttt{r-tree} must read records at the higher levels before reading records at the lower levels.
For a set of virtual fields which are eligible to be recomputed, the actual field computations are performed in the order they are listed in the report script.

It is possible to define virtual variables dependent on accumulators. Such virtual fields are recomputed after the accumulators are updated. Figure 8-3 shows how a virtual is used to add a beginning balance to an accumulator resulting in an ending balance.

<table>
<thead>
<tr>
<th>VIRTUAL</th>
<th>BEG_BALANCE</th>
<th>DLAYOUT</th>
<th>0.1234.56</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRTUAL</td>
<td>EXTENDED_PRICE</td>
<td>MONEY</td>
<td>4</td>
</tr>
<tr>
<td>ACCUMULATOR</td>
<td>TOTAL_PRICE</td>
<td>SUM</td>
<td>EXTENDED_PRICE</td>
</tr>
<tr>
<td>VIRTUAL</td>
<td>END_BALANCE</td>
<td>DLAYOUT</td>
<td>BEG_BALANCE + TOTAL_PRICE</td>
</tr>
</tbody>
</table>

Figure 8-3. Dependency of Virtual On Accumulator.

end_balance will be equal to the beginning balance plus the accumulated total_price.

NOTE: see the "Field Values" section of Advanced Report Layout chapter for a discussion of the A[i j] notation for specifying the control level of an accumulator outside of a control footer.

However, end_balance will not have the correct value until after accumulators have been updated. Therefore, you cannot use a virtual field in a SELECT clause that depends on an accumulator since it does not have the correct value until after the accumulators are updated which does not occur until after the SELECT expression has been evaluated. If you wish an accumulator value to be used in the SELECT expression, try using one of the set functions which does get computed before the SELECT expression is evaluated.

It is also possible to obtain a pointer to specific accumulator and virtual fields. The file rthints.doc included on the r-tree distribution diskette contains a complete example.

Virtuals which depend on an accumulator can be used with TEST_LINE or SKIP_LINE so that the form of the output can depend on accumulator values.

NOTE: Virtuals fields which depend on the status variables LINE_NO and PAGE_NO are automatically recomputed just as each output line is processed. This permits these virtuals to control the page layout. If you wish to make another status variable, such as COUNTER, force virtual computations after each line of output, you must modify the r-tree header file RTIVAL.H as follows:
In the module RTIVAL.H, find the statement

```c
"COUNTER", (pTEXT) &rtcntr[0], RTINT4, 4, 0, NO_LEVEL,
```

Then change the NO_LEVEL to MX_LEVEL so the statement becomes:

```c
"COUNTER", (pTEXT) &rtcntr[0], RTINT4, 4, 0, MX_LEVEL,
```

Then any virtual depending on COUNTER will also be recomputed after each line of output.

Note: If you modify RTVAL.H, be sure to recompile the following modules: RTINTR.C, RTRTIM.C, RTCMPL.C, RTMMRY.C AND RTSYNT.C.

**r-tree Programming**

By "r-tree® programming" we mean the manipulation of virtual fields to achieve complex computations. The four basic virtual field definitions specified above accommodate most virtual field computations. Sometimes it is necessary to chain together several virtual fields to achieve the desired result.

The boolean virtual field permits two different computational expressions to be used to define a virtual field value. But what if you need more than two possible outcomes, and the coded virtual field (which only permits constant values) is not adequate? In language C, such a problem is handled as illustrated in *Figure 8-4*.

```c
if (condition1)
    vf = expression1;
else if (condition2)
    vf = expression2;
    ...
else if (conditionn)
    vf = expressionn;
else
    vf = expressionn+1;
```

*Figure 8-4.*
Figure 8-5 illustrates how you would use r-tree to handle this situation by defining the related chain of n+1 virtual fields.

```plaintext
vf1 ftype flength condition1 ?
expression1 : null_expr
vf2 ftype flength vf1 = null_expr AND condition2 ?
expression2 : vf1
...
vf_n ftype flength vf_n-1 = null_expr AND condition_n ?
expression_n : vf_n-1
vf_n+1 ftype flength vf_n = null_expr ?
expression_n+1 : vf_n
```

Figure 8-5.

where ftype is the same field type and flength the same field length for all n+1 virtual fields, and null_expr is a value which is guaranteed not to be equal to any of the expressions from expression1 to expression_n.

As soon as a condition is true, the corresponding virtual field will have a non-null value; hence, all subsequent virtual fields will be assigned its value. Therefore, vf_n+1 always has the desired value just as if the language C code had been executed.

Another "programming" situation arises when the outcomes are constants, and hence the coded virtual field applies, but the code value (that must match the target codes corresponding to the constant outcome values) is not computable as a simple computational expression. Instead, it relies on a series of boolean conditions. In this case the virtual field definitions in Figure 8-6 can be useful:

```plaintext
vf1 INTZ Z condition1 ? 1 : 0
vf2 INTZ Z condition2 ? 2 : 0
...
vf_n INTZ Z condition_n ? 2n-1 : 0
vf_f INTZ Z vf1 + vfZ + ... + vf_n
vf ftype flength : vf_f
  outcome0
  outcome1
  ...
  2n-1 outcome2n-1
```

Figure 8-6.
The occasion sometimes arises for two fields which are not string fields to be concatenated. The concatenation operator, #, only applies to string valued items. Overlay virtual fields can be used to overcome the concatenation limitation. If field "a" is \textit{len}_a bytes long and field "b" is \textit{len}_b bytes long, Figure 8-7 illustrates concatenation.

<table>
<thead>
<tr>
<th>Virtual</th>
<th>STRING</th>
<th>\textit{len}_a</th>
<th>\textit{a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{b}</td>
<td>STRING</td>
<td>\textit{len}_b</td>
<td>\textit{b}</td>
</tr>
<tr>
<td>\textit{c}</td>
<td>STRING</td>
<td>\textit{len}_c</td>
<td>\textit{a} # \textit{b}</td>
</tr>
</tbody>
</table>

\texttt{* where \textit{len}_c = \textit{len}_a + \textit{len}_b}

Figure 8-7. Concatenation of Non-String Fields.

There are some situations in which it is necessary to generate a sequence of numbers. You may, for example, wish to place sequence numbers on each detail line of a report. The ZERO, INCREMENT and DECREMENT functions can be used for this purpose.

Each time the ZERO function is called, it resets the \textit{r-tree}® status variable COUNTER to zero. Each time the INCREMENT(DECREMENT) function is called, it causes COUNTER to be incremented(decremented) by one.

NOTE: ZERO, INCREMENT and DECREMENT do not return the value of COUNTER. Instead, they return, unaltered, the value of their argument.

The reason for this approach is that you can control when the COUNTER status variable will be updated. The easiest way to control when COUNTER is updated, is to use these functions in the format symbol lists of the IMAGE section. (See the "Format Symbol List" section of the Advanced Report Layout chapter for a discussion of the legitimate entries for a format symbol list which include a built-in function call.) You might, for example, zero the COUNTER each time a control header is output, and increment COUNTER each time a particular line of the BODY subsection is output.

8.4 Record Selection / File Hierarchy Analysis

The \textit{SEARCH} section of an \textit{r-tree} report script defines a file hierarchy as discussed in the "Search" section of the Report Script chapter and "File Hierarchy"section of this chapter. The \textit{SELECT} section of the report script specifies what conditions must be satisfied by data objects to be selected for the report. The \textit{SORT} section dictates how the selected data objects will be organized in the report. \textit{r-tree} ensures that these three report script sections are internally and mutually consistent.
Further, it is very important to understand how your use of SELECT expressions and SORT segments will determine what information is passed to the IMAGE section for reporting.

**SELECT**

Each application-defined data object used in the boolean SELECT expression can be identified with a particular node in the file hierarchy. Each virtual field can likewise be identified. However, if the virtual field definition and/or the SELECT expression utilize set functions, then the mapping to a file hierarchy node must reflect the scope of the set function.

*r-tree*® will not accept a SELECT expression unless all the fields (after adjusting for set functions) used in the expression fall on one simple path down the file hierarchy. A simple path is one that starts at the root and goes down the hierarchy without any bifurcation. (It is not necessary for the path to go to the bottom level of the hierarchy.)

**SORT**

For a given SELECT expression, the fields which make up the SORT segments must fall on the same simple path as defined by the SELECT expression. If the simple path defined by the SELECT expression does not go to the bottom of the file hierarchy, it is permissible for SORT segments to fall on an extension of the path which maintains it as simple.

**What Arrives for IMAGE Processing**

If the SELECT ALL clause is used, then *r-tree* will send the IMAGE section all related combinations of records down the left-most simple path of the file hierarchy. (If there are no branches in the hierarchy, then there is only one path down the hierarchy). If two or more files both share a parent file through the IS_DETAIL_FOR relationship, the one specified first in the SEARCH section will be the left-most file.

If the SELECT ALL clause is not used, then the IMAGE section will receive all related combinations of records which fall on the simple path defined by the SELECT expression and the SORT segments, and which satisfy the SELECT expression.

NOTE: when we say “all related combinations of records which fall on the simple path,” we mean that *r-tree* logically joins the records at each level of the hierarchy (in addition to the explicit joins specified by the JOINS_TO clause). Hence, the records near the top of the hierarchy will be repeatedly joined with different detail records near the bottom of the hierarchy. Therefore, all of the information at each level of the hierarchy is available to each and every line of the IMAGE section.
Example: Selection / Hierarchy Analysis

The file hierarchy for the SEARCH section below is given by:

```
{customer master}
  {customer orders}
  {order-item / item master}
```

Assuming the SEARCH section in Figure 8-8, the following SELECT and SORT sections will pass the data records noted below to the Image section:

- **SELECT cm_custnumb = "C100" AND co_ordrdate < "10/01/86"**
  
  IMAGE will be passed all pairs of customer master and customer order records which satisfy the SELECT clause and which are related according to the SEARCH section. No order-item / item master records will be passed to IMAGE because the SELECT expression does not explicitly include a field from the bottom level of the hierarchy.

- **SELECT cm_custnumb = "C100" AND co_ordrdate < "10/01/86"**
  
  SORT USING_KEY INTEGER co_ordrdate

  The same records as 1 above will be passed to IMAGE, but their order will be based on the customer order date.

- **SELECT cm_custnumb = "C100" AND co_ordrdate < "10/01/86"**
  
  SORT USING_KEY INTEGER oi_itemnumb

  Since the SORT segment goes down the hierarchy one level further than the SELECT expression, the IMAGE section will receive all related triplets of customer master, customer order and order-item/item master records, in item number order.

- **SELECT SUMMATION[extended_price] < $1000 AND oi_quantity > 2**

  The IMAGE section will receive all related triplets of customer master, customer order and order-item records such that the total order price is below $1000 and order quantity for an item is greater than 2. Customer master records are included because the simple path down the hierarchy is always rooted at the
8.5 File Alias

A hierarchical structure is typically considered to be rather inflexible; not capable of representing a full range of file inter-relationships. However, r-tree's ability to place a file at more than one node in the file hierarchy effectively removes many of the drawbacks to a pure hierarchical structure.

A file used in more than one place in the hierarchy is said to be "aliased." Whenever a file is used more than once in the SEARCH section, it must be assigned a file alias for each subsequent use.

The real power of the alias is reflected in the type of queries (SELECT expressions) that can be accommodated. For instance, using aliases, we can find all customer orders that contain at least one item that is also contained on some other specified order.

**SEARCH Syntax**

The standard syntax for a secondary file in the r-tree SEARCH section is

```
FILE value file_relation value BY_FIELD symbol USING_KEY symbol
```

Whenever a file is to be used a second time or more, it must be assigned a unique alias for each subsequent use. The syntax is:

```
FILE cm IS_DETAIL FOR cm BY_FIELD cm_custnumb
USING_KEY cm_custnumb_idx
```

```
FILE co IS_DETAIL FOR co BY_FIELD co_custnumb
USING_KEY co_custnumb_idx
```

```
FILE oi IS_DETAIL FOR oi BY_FIELD oi_ordnumb
USING_KEY oi_ordnumb_idx
```

```
FILE li IS_DETAIL FOR li BY_FIELD li_itemnumb
USING_KEY li_itemnumb_idx
```

---

**Figure 8-8. Example Three-Level File Hierarchy.**
A schematic of the file hierarchy represented by the SEARCH section shown in Figure 8-9 follows:

```

{customer order / customer master} Level 0

{order-item / item master} Level 1

{customer order} Level 2

{order-item} Level 3

```

There are two subtle points brought out by this example. First, the file name following a JOINS_TO or IS_DETAIL_FOR keyword always references the closest preceding occurrence of the file. Therefore, the "IS_DETAIL_FOR co" which appears in the last FILE statement above refers to the customer order file in the immediately preceding FILE statement and not the customer order file in the first FILE statement. Second, the choice of fields to link aliased files to their associated files is not always straightforward. In this example, we have used a field called empty to link the aliased customer order file to the order-item file. empty should be defined as a string of zero length. Then, whenever the aliased customer order file is traversed as a child of the order-item file, it will have all of its records read, not simply those tied directly to the particular order-item record from level 1. We have chosen this approach because of the nature of the query which is used in the example of the next Section.
NOTE: a SYNTAX ERROR will occur if you attempt to alias the primary file to itself in the first two statements in the SEARCH clause and the primary file has a USING KEY clause. The example in Figure 8-10 will produce a syntax error.

The syntax error occurs because \texttt{r-tree} thinks the \texttt{co2} is part of a range expression for the primary file, and it expects a [ ] or ] instead of the keyword "IS_ALIAS_FOR".

### Aliased Data Fields

Except for the SEARCH section itself, any time that \texttt{r-tree} accepts a symbolic value, it may be replaced by the dot notation

\texttt{alias . symbol}

where \texttt{alias} must be an alias assigned to the file of which \texttt{symbol} is a data field.

The symbolic references \texttt{alias . symbol} and \texttt{symbol} are not equivalent. The former refers to a data field at a different position in the file hierarchy from the latter, even though they both reference the "same" logical field. Therefore, the use of one form or the other can drastically affect the combinations of records that will be sent to the IMAGE section.

\texttt{r-tree automatically maintains data definitions for the aliased data fields. You should NOT create entries in your DODA for the aliased data fields.}

Continuing with the SEARCH section introduced above, consider the following query:

- Find all customer orders which have at least one item in common with any of the orders for customer number “C100”.

This query can be handled (in part) by the boolean expression

\texttt{oi\_itemnumb = oi2.oi\_itemnumb AND co2.co\_custnumb = "C100"}

Examining the associated file hierarchy reveals that this boolean expression involves data fields from levels 1, 2 and 3. Since all selections fall on a path rooted at level 0,
the above simple boolean expression would cause all 4-tuples of related records from levels 0 through 3 (which satisfy the boolean expression) to be passed to the IMAGE section. Since we only want the customer orders for which the boolean is true, we use the EXISTS function to force the selection criteria to level 0:

```
EXISTS [ EXISTS [ EXISTS [ oi_itemnumb = oi2.oi_itemnumb AND co2.co_custnumb = "C100" ] ] ]
```

The above SELECT expression with the nested EXISTS is interpreted as:

- Find all customer order records from level 0 such that there exists at least one related order-item record from level 1, such that there exists at least one customer order record from level 2 (not necessarily related because the empty field is used to link the files), such that there exists at least one related order item record from level 3, whereby the item numbers match and the level 2 customer number is "C100".

### 8.6 File Interrelationship Alternatives

r-tree® provides alternatives with respect to the treatment of missing records and the order in which to search detail records.

#### Missing Records

r-tree recognizes two ways to relate files:

- JOINS_TO (one-to-one or many-to-one)
- IS_DETAIL_FOR (one-to-many)

However, r-tree must be prepared if matching records are not found between the associated files. r-tree’s three options, which can be handled on a file-by-file basis, are:

- skip the record which does not have a match in the subsidiary file;
- report an r-tree error if a matching record is not found and terminate report processing; and
- treat the missing record as a “empty” record, comprised entirely of zero bytes.
The file relationship keywords determine which option is in effect. The keywords (including synonyms) for files “joined” together via a one-to-one or many-to-one relationship are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Non-Matching Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOINS_TO</td>
<td>skip record</td>
</tr>
<tr>
<td>JOINS_TO_SKIP</td>
<td>skip record</td>
</tr>
<tr>
<td>JTS</td>
<td>skip record</td>
</tr>
<tr>
<td>JOINS_TO_ERROR</td>
<td>return error code</td>
</tr>
<tr>
<td>JTE</td>
<td>return error code</td>
</tr>
<tr>
<td>JOINS_TO_NULL</td>
<td>treat missing record as null</td>
</tr>
<tr>
<td>JTN</td>
<td>treat missing record as null</td>
</tr>
</tbody>
</table>

The keywords (including synonyms) for parent-detail file relationships are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Non-Matching Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS_DETAIL_FOR</td>
<td>skip record</td>
</tr>
<tr>
<td>IS_DETAIL_FOR_SKIP</td>
<td>skip record</td>
</tr>
<tr>
<td>DTS</td>
<td>skip record</td>
</tr>
<tr>
<td>IS_DETAIL_FOR_ERROR</td>
<td>return error code</td>
</tr>
<tr>
<td>DTE</td>
<td>return error code</td>
</tr>
<tr>
<td>IS_DETAIL_FOR_NULL</td>
<td>treat missing record as null</td>
</tr>
<tr>
<td>DTN</td>
<td>treat missing record as null</td>
</tr>
</tbody>
</table>

Some care must be taken when using the "treat missing record as null" alternative. Ordinarily, this alternative will not make sense for a subsidiary file which itself has other subsidiary files below it in the hierarchy since the missing record problem will cascade down the hierarchy.

**Detail Record Ordering**

Ordinarily, detail records are retrieved via the FRSSET and NXTSET functions which means that they are retrieved in ascending key value order. By substituting USING_REVKEY in place of USING_KEY, the detail records will be read via the LSTSET (last set), PRVSET (previous set) functions. That is, they will be read in descending key value order.
8.7 Built-In Functions

**r-tree** supports a simple form of built-in functions. Each function takes only one argument which must be either an entry from your data object definition array, or one of the virtual fields defined in the VIRTUAL section of your report script. You cannot use constants or accumulators as arguments to functions. Adding built-in functions, as discussed below, provides a method for making your own functionality available to all **r-tree** programs. See "User defined functions" in the Index for adding functionality to individual **r-tree** programs.

**NOTE:** the built-in functions do not modify the actual argument. Instead, they work on a local copy of the argument, and return a pointer to the local copy.

### Description of Built-In Functions

**r-tree** is shipped with a wide variety of built-in functions which are described in the following table:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Type of Input</th>
<th>Type of Output</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDTO</td>
<td>any</td>
<td>same</td>
<td>Same as input, and computational result for virtual field is added to current contents of virtual instead of replacing contents</td>
</tr>
<tr>
<td>DAT_DAY</td>
<td>DATE</td>
<td>STRING</td>
<td>Name of day of week: e.g., &quot;Monday&quot;</td>
</tr>
<tr>
<td>DAT_DD</td>
<td>DATE</td>
<td>STRING</td>
<td>Day of month: &quot;01&quot; to &quot;31&quot;</td>
</tr>
<tr>
<td>DAT_MM</td>
<td>DATE</td>
<td>STRING</td>
<td>Month: &quot;01&quot; to &quot;12&quot;</td>
</tr>
<tr>
<td>DAT_MTH</td>
<td>DATE</td>
<td>STRING</td>
<td>Name of month: e.g., &quot;January&quot;</td>
</tr>
<tr>
<td>DAT_YYYY</td>
<td>DATE</td>
<td>STRING</td>
<td>4-digit year: e.g., &quot;1989&quot;</td>
</tr>
<tr>
<td>DBL_STR</td>
<td>DFLOAT</td>
<td>STRING</td>
<td>ASCII string representation of double</td>
</tr>
<tr>
<td>DECREMENT</td>
<td>any</td>
<td>same</td>
<td>Same as input, and COUNTER is decremented</td>
</tr>
<tr>
<td>D1,D2,D3,D4</td>
<td>any</td>
<td>same</td>
<td>Same as input, and Cx (x=1, 2, 3, 4) is decremented</td>
</tr>
<tr>
<td>INCREMENT</td>
<td>any</td>
<td>same</td>
<td>Same as input, and COUNTER is incremented</td>
</tr>
<tr>
<td>I1,I2,I3,I4</td>
<td>any</td>
<td>same</td>
<td>Same as input, and Cx (x=1, 2, 3, 4) is incremented</td>
</tr>
</tbody>
</table>
**SUBSTRING and WILDCARD String Comparisons**

*r-tree®* scripts frequently include comparisons of strings in either the SELECT section or as part of the boolean test in virtual field computations. The SUBSTRING and.
WILDCARD built-in functions permit powerful string comparison expressions beyond the traditional tests for equality or inequality.

These functions indicate that their arguments are patterns to be compared with other strings. The patterns are made up of regular characters and the special characters: the question mark ("?"), the asterisk ("*"), and the back-slash ("\"). The question mark matches any single character, the asterisk matches zero or more occurrences of any characters, and the back-slash is used to escape special characters (i.e., when you do not want a special character to be treated special, you precede it by the back-slash).

The following table demonstrates how patterns match target strings. Following the table Figure 8-11 contains a script fragment which demonstrates the proper use of the wildcard functions.

The SUBSTRING function is simply a special case of the WILDCARD function in which r-tree automatically adds an asterisk before and after the pattern. Then a match will be successful if the original pattern occurs anywhere in the target string.

### Pattern Matching Examples

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Target</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*B</td>
<td>AxyzwB</td>
<td>Yes</td>
</tr>
<tr>
<td>A*B</td>
<td>AB</td>
<td>Yes</td>
</tr>
<tr>
<td>A*B</td>
<td>AxyzwBx</td>
<td>NO</td>
</tr>
<tr>
<td>A<em>B</em></td>
<td>AxyzwBx</td>
<td>YES</td>
</tr>
<tr>
<td>A*?B</td>
<td>AB</td>
<td>NO</td>
</tr>
<tr>
<td>A*?B</td>
<td>AxB</td>
<td>YES</td>
</tr>
<tr>
<td>AB*</td>
<td>ABxyz</td>
<td>YES</td>
</tr>
<tr>
<td>AB*</td>
<td>wAB</td>
<td>NO</td>
</tr>
<tr>
<td><em>AB</em></td>
<td>AB</td>
<td>YES</td>
</tr>
<tr>
<td><em>AB</em></td>
<td>jhgABkjh</td>
<td>YES</td>
</tr>
<tr>
<td><em>AB</em></td>
<td>kjbAbkjh</td>
<td>NO</td>
</tr>
<tr>
<td>??<em>AB</em></td>
<td>xABxcv</td>
<td>NO</td>
</tr>
<tr>
<td>??<em>AB</em></td>
<td>xcbAB</td>
<td>YES</td>
</tr>
<tr>
<td>AB?C?D</td>
<td>ABxxCxD</td>
<td>NO</td>
</tr>
<tr>
<td>AB?C?D</td>
<td>ABxCxDx</td>
<td>NO</td>
</tr>
<tr>
<td>AB?C?D</td>
<td>ABxCxD</td>
<td>YES</td>
</tr>
<tr>
<td>AB?C?D*</td>
<td>ABxCxDx</td>
<td>YES</td>
</tr>
<tr>
<td>AB?C?D</td>
<td>xABxCxD</td>
<td>NO</td>
</tr>
</tbody>
</table>
NOTE: the SUBSTRING and WILDCARD functions must be used in string comparisons using the equality (=) and inequality (<> ) operators.

Adding Your Own Built-In Functions

In order to add your own built-in function to \( \text{r-tree}^{\circ} \), you must follow the steps below. To help with their understanding, we will use an example based on a function which returns the first character of a string.

1) Assign your function a unique name. The name must begin with a letter and may contain other letters, digits and underscores; e.g.,

```
FIRST_CHAR
```

2) Modify the initialization of the \( \text{fn}[] \) structure array at the beginning of \( \text{rival.h} \) by adding your function name and a unique function number. For example, if last entry is now:

```
"DBL_STR", 39   /* double to string */
```

then change the above to:

```
"DBL_STR", 39, /* double to string */
"FIRST_CHAR",40 /* first character */
```

3) This step and the following steps apply to the \( \text{ruser.c} \) module. In this and the following steps, the case statements are based on the unique function number assigned to the function in Step 2 above.

If the value returned by the function is not of the same length as the input argument, add a case statement under the comment "adjust length of result if necessary" which
specifies the longest possible length of the return value. The local variable `rlen` should be assigned this length as illustrated in Figure 8-12.

```
case 4B:
    rlen = 1;
    break;
```

**Figure 8-12.**

4) Add a case statement as illustrated in Figure 8-13 under the comment "actual function computation" (in `rtuser.c`) in which the return value is computed at the address specified by the local char pointer `bp`. At this point in the code, `bp` points to a temporary, local area `rlen` bytes long in which the return value should be stored. `tp` points to the first byte of the input argument, and `r->elen` specifies the length of the input argument.

```
IF THE LENGTH OF THE RETURN VALUE IS NOT EXACTLY THE SAME AS THE LENGTH OF THE INPUT ARGUMENT, THIS CODE MUST ALSO SET THE VALUE OF `r->elen` TO THE ACTUAL RETURN LENGTH.

UNDER NO CIRCUMSTANCES SHOULD THE INPUT ARGUMENT BE CHANGED BY THE CODE ADDED HERE! FURTHER, THE RETURN VALUE MUST NOT EXCEED `rlen` BYTES!

```
case 4B:
    if (r->elen >> 0) {
        *bp = *tp; /* set return to 1st byte */
        r->elen = 1; /* reset length to one */
    } else {
        *bp = '\0';
        break;
    }
```

**Figure 8-13.**

```
IF THE TYPE OF THE RETURN VALUE IS DIFFERENT FROM THE INPUT ARGUMENT TYPE, CHANGE THE VALUE OF `r->etype` TO THE RETURN VALUE TYPE. SEE "case 4: STR_FLT" FOR AN EXAMPLE.
```
5) In the function *usertyp* under the comment "check input argument type," find the correct subsection which specifies what restrictions exist, if any, on the input argument. For example, since we desire only string input arguments, we would add a "case 40" statement under the subsection comment "string input required" as illustrated in Figure 8-13.

6) The last step is to add a case statement under the appropriate subsection of the area entitled "compute return type." Since, in our example, the return type is the same as the input type (both are STRINGs), we simply add our "case 40;" statement under the comment "return type same as input type."

Be sure all *r-tree*® files and your entire application are recompiled after making the specified changes. It is imperative to recompile *r-tree*’s script module (i.e., rtintr, rtcmpl, rtmnry).

### Function Arguments

The *r-tree* built-in functions only permit one input argument. If you wish to pass more than one argument to a function, it is necessary to concatenate the arguments (see Concatenation of Non-string Fields in the Virtual Fields section of this chapter), and to separate the arguments as part of the function computation.

#### Input Values

Ordinarily, if you concatenate arguments, the function should be set up to accept STRING input values. When your function is called during *r-tree* processing, the local text pointer *tp* points to the concatenated arguments (see Step 4 of "Adding Your Own Built-In Functions" section in this chapter). Use the *cpybuf()* function to copy each component of the input argument into a local variable of the proper type. For example, if the input argument consists of two doubles, then separate the input arguments as in Figure 8-14.

```c
double d(2);
cpybuf(d, tp, sizeof(double));
cpybuf(d + 1, tp + sizeof(double), sizeof(double));
```

*Figure 8-14.*

#### Multiple Functions

If you have a need for a large number of functions, each for a different type of parameter, or for a different computation, you may use the method of this section in the following...
manner. Concatenate the function type with the argument. Then separate the type and the value as described above. You may then use a switch statement or some other device to branch to the appropriate computation.

8.9 Using Set Functions With Virtual Fields

When using set functions with virtual fields, it is sometimes necessary to modify the SELECT expression. In particular, if the SELECT expression does not contain any set functions and does not depend on virtual fields which themselves depend on set functions, then r-tree® will not compute the virtual field which uses the set function until after the select operation, or it may generate an error message indicating that more than one pass is required.

For example, the script fragment shown in Figure 8-15 will not work (i.e., `order_total` and `order_freq` will not be computed prior to record selection or record output) because the SELECT expression does not use or reference any set functions. However, the following SELECT expression causes the SUMMATION and FREQUENCY set functions to be invoked since the SELECT expression includes a reference (viz., `order_total`) to the set functions. Since the reference is trivially satisfied, it does not change the records selected.

8.10 Temporary Files Used By r-tree

r-tree® may use up to three temporary files processing a script. Therefore, applications that call report should have at least three additional file handles available prior to calling the report() function.

If the output device is a file instead of a display device, then r-tree creates a file by the name “RTREE.OUT.” The name of this temporary file can be changed by copying your desired name into the TEXT array `rtfname[]`. Such a string copy should be performed prior to invoking the report function.

When a SORT section is included in a script, then two additional temporary files are created: “RTSORTI” and “RTSORTD.” When more than one r-tree script is in operation at the same time, you may want to change these names to avoid multi-user
conflicts. These file names are stored in the TEXT arrays \textit{rtsname[]} and \textit{rtdname[]} respectively.

All three of the above mentioned TEXT arrays are initialized in \textit{rtival.h}, and are configured to hold file names (including paths) of up to sixty-three (63) characters. To change one of these arrays, say \textit{rtsname[]}, use a statement of the form:

\begin{verbatim}
strcpy(rtsname,myname);
\end{verbatim}

where \textit{myname} points to a NULL terminated text string, or \textit{myname} is replaced by a quoted string constant.
The IMAGE section of the r-tree report script allows you to design a report layout which specifies five classes of information to be sent to the output device:

- literal character strings;
- formatted field values;
- binary (raw, unformatted) field values;
- physical device control strings; and
- logical control information.

The IMAGE section is comprised of one or more of the following subsections:

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>device initialization data</td>
</tr>
<tr>
<td>PAGE_HDR</td>
<td>top of page layout; may be suppressed for the first page</td>
</tr>
<tr>
<td>REPORT_HDR</td>
<td>report header layout</td>
</tr>
<tr>
<td>CONTROL_HDR(i)</td>
<td>control level i introductory layout</td>
</tr>
<tr>
<td>BODY</td>
<td>layout for information in each group of selected records</td>
</tr>
<tr>
<td>CONTROL_FTR(i)</td>
<td>control level i summary layout</td>
</tr>
<tr>
<td>REPORT_FTR</td>
<td>report summary layout</td>
</tr>
<tr>
<td>PAGE_FTR</td>
<td>bottom of page layout; may be suppressed for the last page</td>
</tr>
</tbody>
</table>

Each subsection may be of any length, although the page header and footer combined must fit on one page. None of the subsections is required, although no output will be produced without at least one IMAGE subsection.

In general, each entry in the IMAGE section (or more properly each entry in one of the subsections of the IMAGE section) is comprised of an image line and a format symbol list. The image line and the format symbol list work together to permit all five classes of information to be used in your reports.

The DISPLAY section of the report script allows you to modify overall characteristics of the report output such as page length or type of currency mark to use.

In this section, we will examine these two components of the report script in some detail.
9.1 Image Line

Each image line begins with a LINE_MARK in the first column. The default value for LINE_MARK is an ASCII plus sign (i.e., ‘+’ or decimal 43). The LINE_MARK is followed by a string of literal characters and format specifications. When the image line is sent to the output device, the LINE_MARK is not sent. However, the LINE_MARK symbol may appear anywhere else in the image line and is not treated as a special symbol in any location other than the first column of the line.

The literal characters included in the image line are sent to the output device without any translation, provided they are within the page width defined for the output device. If they fall beyond the page width, then they are not sent to the device.

Interspersed with the literal characters are format specifications. The format specifications determine how a field value will appear in the report output. A format specification begins with a FORMAT_MARK. The default value for the FORMAT_MARK is an ASCII at sign (i.e., ‘@’ or decimal 64). Following the FORMAT_MARK are format characters. For example, ‘x’ and/or ‘X’ are used to represent the left-justified output of the characters of string valued fields and dates.

```
IMAGE
REPORT_HDR
  +--------------------------------------------------+
  + MONTHLY TRIAL BALANCE +
  +--------------------------------------------------+
  + Company: xxxxxxxxxxxx Date: xxxxxxxx +
```

*Figure 9-1. Sample Report Header Without Symbol List.*

Notice that the REPORT_HDR shown in Figure 9-1 is not complete because we have not supplied the format symbol list which will show the field values to be substituted into the format specifications in the last line of the subsection.

9.2 Format Specifications

Each format specification begins with the FORMAT_MARK (‘@’) character. Except for two special cases, the character following the FORMAT_MARK determines the format type. PLEASE NOTE that the format types in the following table are NOT case sensitive, i.e. @X is the same as @x. The r-tree format types are:
In addition to the symbols used to denote the different format types, a format specification may contain the following additional characters:

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>£</td>
<td>dollar sign</td>
</tr>
<tr>
<td>(</td>
<td>left parentheses</td>
</tr>
<tr>
<td>)</td>
<td>right parentheses</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>-</td>
<td>dash</td>
</tr>
<tr>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>:</td>
<td>semicolon</td>
</tr>
<tr>
<td>b</td>
<td>lower-case b</td>
</tr>
<tr>
<td>z</td>
<td>lower-case z</td>
</tr>
</tbody>
</table>

The interpretation of these additional characters depends on the format type, and is discussed in the following sections.

NOTE: The lower-case "b" always represents an embedded blank to be inserted in the output.
In order to determine where a format specification terminates and the literal characters resume, r-tree® follows these rules:

A format specification terminates

- and another format specification begins when a FORMAT_MARK is encountered;
- with the occurrence of a PCNTRL_MARK (default '^'). The PCNTRL_MARK is treated as the last character of the format specification; and
- with the occurrence of a character not in the set of characters: "$()*,-./09:CDERVWXbcdevwxz"

Figure 9-2 shows some sample image lines with the first and last character of each format specification marked. Notice a blank space will always terminate a format specification.

It is necessary to terminate a format specification with the PCNTRL_MARK only when the character following the PCNTRL_MARK is meant to be interpreted as a literal character, but is a legal character for a format specification. That is the difference between Figures 9-1 and 9-2 where the trailing dashes ('-') are legal format characters.

In order to embed a blank space in a format, you must use the 'b' format character since a blank space (' ') will terminate the format specification.
Alphanumeric Formats

There are five types of alphanumeric formats:

- left-justified
- centered
- left-justified variable
- right-justified
- word-wrapped

A left-justified format causes the field value to begin at the FORMAT_MARK and is truncated at the last format character or padded with blanks on the right, whichever is necessary. A right-justified format causes the last non-blank character of the field value to be output at the last format character position and truncated on the left or padded with blanks on the left, whichever is necessary. A centered format causes left-justified output if the field value (after being stripped of trailing blanks) is longer than the format; otherwise the field value is padded on the left and right to center the value in the format.

A word-wrapped format causes left-justified output with the automatic continuation (instead of truncation) of the field value on additional lines. The field value is treated as a series of words; new lines are started at the most convenient blank positions within the field value. A new line can be forced by embedding a carriage return ("\r") within the field value at the location of the desired new line break.

A left-justified variable format causes the field value to be stripped of trailing blanks, and then output is begun at the FORMAT_MARK. If the stripped value is shorter than the format, then no padding is added. This causes all subsequent output on the line to be shifted to the left. If the value is longer than the format, then it is truncated to the length of the format.

In addition to the output of the characters in the field value, the alphanumeric formats allow for embedded characters which are inserted in place. The list of embedded characters for alphanumeric formats are:

"$()*,-./09:DEbedz"
Figure 9-3 illustrates how to use alphanumeric formats. It assumes the output value is the 24-character string: 

"Happy New Year 1989"

<table>
<thead>
<tr>
<th>Output value: &quot;Happy New Year 1989&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>#XXXXXXXXXXXXX  YYYYYYYYYYYYYYYYYY</td>
</tr>
<tr>
<td>Happy New Year  New Year 1989 New Year 1989</td>
</tr>
<tr>
<td>#XXXXXXXXXXXXX  YYYYYYYYYYYYYYYYYY</td>
</tr>
<tr>
<td>Happy New Year  New Year 1989 New Year 1989</td>
</tr>
<tr>
<td>#XXXXXXXXXXXXX  YYYYYYYYYYYYYYYYYY</td>
</tr>
<tr>
<td>Happy New Year  New Year 1989 New Year 1989</td>
</tr>
</tbody>
</table>

Figure 9-3. Sample Alphanumeric Formats.

**Dates and Alphanumeric Outputs**

r-tree® represents dates internally as a four-byte integer value which can be used in computations. However, when r-tree outputs a date value, it automatically converts it into a character string format. There are three ways to output a date:

- use the automatic r-tree date-to-string conversion which is in the form
  "mm/dd/yy"
  where mm is a month from 01 to 12, dd is a day from 01 to 31, and yy is the last two digits of the year.

- using the DATE_SEPARATOR and DATE_ORDER display attribute variables, you can change the '/' to any other symbol, and change the order in which the mm, dd and yy are output.

- convert the four-byte computational form to strings with the following built-in functions:
  
  DAT_DAY name of day of week
  DAT_DD month day as a two-byte string
  DAT_MM month number as two-byte string
  DAT_MTH name of month
  DAT_YYYY year as four-byte string
Figure 9-4 demonstrates the use of these data functions.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>STRING</td>
<td>DMY(YYYY)</td>
<td>DMY(YYYY)</td>
</tr>
<tr>
<td>MONTH</td>
<td>STRING</td>
<td>DMY(MM)</td>
<td>DMY(MM)</td>
</tr>
<tr>
<td>DAY</td>
<td>STRING</td>
<td>DMY(DD)</td>
<td>DMY(DD)</td>
</tr>
<tr>
<td>MONTH_DAY</td>
<td>STRING</td>
<td>DMY(MM/DD)</td>
<td>DMY(MM/DD)</td>
</tr>
<tr>
<td>LONG_DATE</td>
<td>STRING</td>
<td>DMY(MM/DD/YY)</td>
<td>DMY(MM/DD/YY)</td>
</tr>
</tbody>
</table>

**NOTE:** numeric format specifications use ‘$’ to denote a currency mark, ‘,’ for a thousand separator, ‘.’ for a decimal point, and ‘*’ for zero fill. However, using the display attribute variables CURRENCY_MARK, THOUSAND_SEPARATOR, DECIMAL_POINT and ZERO_FILL, you can change what actually gets placed in the output stream to any alternative set of characters.

**Numeric Formats**

Numeric formats provide a wide range of ways to output numbers and financial data. The optional special features include:

- floating currency mark
- automatic thousands separators
- zero fill with blanks, zeros or special zero fill character
- negative values represented by: -, ( ... ), DR or CR
- negatives may be shown preceding or following the value
- exponential form

NOTE: numeric format specifications use ‘$’ to denote a currency mark, ‘,’ for a thousand separator, ‘.’ for a decimal point, and ‘*’ for zero fill. However, using the display attribute variables CURRENCY_MARK, THOUSAND_SEPARATOR, DECIMAL_POINT and ZERO_FILL, you can change what actually gets placed in the output stream to any alternative set of characters.
The basic rules for forming a numeric format specification are:

- Numeric formats always begin with: @9, @0, @* or @E. The first three of
  these permit the full range of numeric format options. The @E implies the
  exponential format option, and no special features such as currency marks or
  zero fill are supported. Further, the exponential formats only apply to values
  stored in single or double precision floating point.

- To signify that a number should be prefixed by a floating currency mark,
  include a ´$´ in the format specification (before the decimal point, if any).

- To include automatic thousand separators, include a ´,´ in the format
  specification (before the decimal point, if any).

- The default treatment for negatives is a floating ´-´ in front of the first digit or
  currency mark, if any.

- To cause a trailing ´-´ or ´CR´ or ´DR´ to be used for negative values, place the
  respective character(s) at the very end of the format specification.

- To use parentheses for negative numbers, place a ´(´ in the format specification
  (before the decimal point, if any), and a ´)´ in the last position of the format.

- Use a ´0´ to denote a place to be padded with zeros if no significant digit
  applies.

- Use a ´9´ to denote a place to be padded with blanks if no significant digit
  applies.

- Use a ´z´ to denote the decimal point (in place of a ´.´) for output which should
  be blank (i.e., not even the decimal point will print) on a zero value.

- ´b´, ´/´ and ´:´ will be treated as embedded characters (where ´b´ is replace by a
  blank).

Ordinarily, only numeric valued fields are used with numeric formats. However, it is
possible to send a string valued field to a numeric format specification if the number
of decimal places in the string match exactly with the number of decimal places in the
format specification.

When you want numeric output left-justified or otherwise treated differently than the
numeric formats, you may use the alphanumeric formats with numeric values, espe-
cially if they are integer values (as opposed to floating point).

NOTE: if a number will not properly fit in the space reserved by the format specifica-
tion, r-tree® will output question marks (´?´) in place of the number. However, r-tree
will continue to process the report. If no other more serious run-time error occurs, then
the report function will return PNFM_ERR to signify the format problem.

If format problems have occurred, the global variable nfmerr will be non-zero. To check
nfmerr, include an external declaration of the form in your application code:
The number of decimal positions in an exponential format is determined by the length of the format, not by an explicit decimal point in the format specification. In fact all special and/or embedded format characters are ignored in the exponential format. The number of decimal positions is eight (8) less than the length of the format. An exponential format must be at least eight characters long. Figure 9-5 shows sample numeric values as produced by sample script cos6 found on the r-tree® diskette.

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#99999999999</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>#99999999999.99</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000000</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000001</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000002</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000003</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000004</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
<tr>
<td>00000000000</td>
<td>1234.56</td>
<td>-1234.50</td>
</tr>
</tbody>
</table>

Figure 9-5. Sample Numeric Formats.

NOTE: data fields of type RTCHAR or RTCHARU, or virtual fields of type CHAR or CHARU are treated as numeric quantities in all operations except for output. On output, such fields use the equivalent of a language C "%c" format specification. To print such fields as numeric quantities, use a virtual field of type INT2 which is defined to be equal to the character field. If you want a character field to be treated like a STRING, then change its definition to RTSTRING or STRING with a length of one.
Raw Dump Format

If you wish to use r-tree® to create a c-tree® Plus fixed-length data file containing selected fields and virtual values, you may find the raw dump format helpful. Instead of translating an internal value to an ASCII string, the raw dump format simply causes the internal binary representation of each field to be output. This is usually meaningful only when trying to construct a c-tree Plus data file as the output of the report generator. To construct a c-tree Plus fixed-length data file as the output of a report, set the DEVICE attribute (in the DISPLAY section) to INPFILE. Unless you have modified the contents of the TEXT array rfname[], the name of the file created will be "RTREE.OUT".

The format is specified as simply @D or @d. Unlike all the other format types, the length of the format specification is not of any significance. The underlying binary representation of the field value determines how many bytes will be output.

Unlike all the other format specifications for which each format applies to one field in the format symbol list, a single raw dump format specification applies to all as-yet-unprocessed fields in the format symbol list. No padding is inserted between the binary output for each field.

Remember that a fixed-length c-tree Plus data file reserves the first byte for a delete flag. When the first byte of a record is set to FF hex, the record is assumed deleted. If possible, simply select an alphanumeric string field as the first field to output with the @D format. If there are no string fields to output, then add a dummy field guaranteed to be zero to the front of your format symbol list.

The record length of the c-tree Plus fixed-length data file is determined by the PAGE_WIDTH display attribute. It defaults to 132, and can be set to any reasonable value (greater than or equal to the total bytes output by the raw dump format) in the DISPLAY section of your report script.

In Figure 9-6 we assume string_field is a ten-byte string and money_field a four-byte integer. The raw dump will cause fourteen bytes of raw data to be output each time this line is processed by r-tree®. No padding is inserted by r-tree® between the fields. If we want to minimize the disk space consumed by the c-tree Plus fixed-length data file, we should set the PAGE_WIDTH to 14 to reflect the amount of data in each fixed-length record.

```
@D
string_field money_field
```

Figure 9-6. Raw Dump Format.
Very Short Format Specifications

There are two special cases in which the second character of the format specification need not be one of

"XxRzCcWwVv09*EeDd"

If the format is only one character long, then it is simply specified as ‘@’, and there is no second character. If the second character is the PCNTRL_MARK (‘^’), then r-tree assumes that the PCNTRL_MARK is in place of an ‘x’ for a string valued field, or a ‘9’ for a numeric field.

9.3 Format Symbol List

Each entry in the IMAGE section is made up of an image line containing literal characters and format specifications, and an optional format symbol list. When present, the format symbol list must start on a new physical line. It cannot be on the same physical line of the report script since r-tree could not distinguish between the image line and the format symbol list.

The format symbol list is required if the image line contains one or more format specifications. If there are no format specifications, then the image line may be followed by a format symbol list containing display logic variables.

In the simplest case, a format symbol list contains one symbolic name (or constant value) for each format specification in the preceding image line.

The order in which the symbols are listed determines the order in which they are substituted into the format specifications as illustrated in Figure 9-7’s image line.

```
* Company = @company_name Date = @SYSDATE Run = #99
  company_name  SYS_DATE  run
```

Figure 9-7 Simple Image W/ Matching Format Symbol List

However, in addition to the symbolic field values to be substituted into the format specifications, the format symbol list may contain constants, display logic variables (and their arguments) and device strings. The display logic variables and device strings are not substituted into the format specifications.
Field Values

A format specification may apply to one of the following types of values:

- data field from the data object definition array;
- virtual field from the VIRTUAL section;
- a built-in function applied to either a data field or a virtual field;
- accumulator field from the ACCUMULATOR section;
- a status variable;
- control field from the CONTROL section; and
- constant value.

The values for the first five categories above are referenced by the appropriate symbolic name(s). The last category, constants, are simply listed in their literal form in the format symbol list. The contents of a control field are referenced via an expression of the form

\[ C[i_{nt2u}] \]

where the ‘C’ must be upper-case and \( i_{nt2u} \) represents a small positive integer corresponding to the desired control break level. Control fields are referenced in this manner since we do not assign them symbolic names.

Recall that accumulators are not single-valued entities. Each accumulator has a value for each control break level you have defined, plus an overall report value and a page value. When accumulators are used in the IMAGE section, r-tree® makes the following assumptions:

- if the accumulator is used in footer subsection (i.e., CONTROL_FTR(i), PAGE_FTR or REPORT_FTR), then r-tree® uses the accumulator value associated with that footer; otherwise
- r-tree® uses the accumulator value associated with the lowest control break level (i.e., the control level with the highest number in the CONTROL section).

You can override this automatic accumulator usage by referring to the accumulator with the following notation instead of its symbolic name:

\[ A[i_{nt2u} i_{nt2u}] \]

where the ‘A’ must be in upper-case, the first \( i_{nt2u} \) specifies the particular accumulator, and the second \( i_{nt2u} \) specifies which control break level. The first accumulator listed in the ACCUMULATOR section corresponds to a 0 for the first subscript, the second accumulator corresponds to a 1, and so on. If there are \( n \) control break levels defined
in the CONTROL section, then the second subscript may run from 0 to n+1. Zero corresponds to the overall report values, 1 through n correspond to the n control break levels, and n+1 corresponds to the page footer values. Figure 9-8 shows an example of printing three different accumulator values.

<table>
<thead>
<tr>
<th>Control</th>
<th>1</th>
<th>co_ordnumb</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>all_page</td>
</tr>
<tr>
<td>ACCUMULATOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number_items</td>
<td>FRW</td>
<td>extended_price</td>
</tr>
<tr>
<td>order_total</td>
<td>SUM</td>
<td>extended_price</td>
</tr>
<tr>
<td>IMAGE</td>
<td>BODY</td>
<td></td>
</tr>
<tr>
<td>+ 999.990.00</td>
<td>999.990.00</td>
<td>999990</td>
</tr>
<tr>
<td>A1,1.1</td>
<td>A1(0,1)</td>
<td>A10,31</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*= A1.1</td>
<td>= order_total accumulator for control level 1</td>
<td></td>
</tr>
<tr>
<td>*= A1.01</td>
<td>= order_total accumulator for overall report</td>
<td></td>
</tr>
<tr>
<td>*= A10.31</td>
<td>= number_items accumulator for the current page</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*** NOTE THAT #1.33 IS EQUIVALENT TO A (1,3) ***</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9-8. Accumulators In Body Section.

Display Logic Variables

The importance of the display logic variables for producing a polished report cannot be overstated. The display logic variables allow you to control which image lines will be selected for printing, set absolute and conditional page breaks, change the left margin on the fly, concatenate output lines, insert form feeds into the output stream, display field values in outline form, vertical tab, conditionally select an output column, and control report headings and pagination.

With the exception of TEST_LINE and SKIP_LINE which, if used, must be the first item in the format symbol list, the order of the display logic variables is usually not important. However, some display logic variables, such as VERT_TAB, will not be active unless they are consumed while the r-tree™ searches for a value (in the format symbol list) to substitute into a format specification (in the associated output line). These will be noted as yes under Format Scan Required in the following table. A no in this column of the table means that even if there is no format specification in the output line, the display logic variable will still be active. Also, a no implies that the action will take place either before or after the line is output. Whether it is before or after depends on the particular display variable.
Some display logic variables, such as TEST_LINE, must be followed in the format symbol list by an argument which is evaluated to determine the appropriate action. When an argument is required, the argument must evaluate to a numeric value. Such display logic variables will be noted yes under the **Argument Required** column in the table below:

<table>
<thead>
<tr>
<th>Display Logic Variables</th>
<th>Argument Required</th>
<th>Format Scan Required</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN</td>
<td>yes</td>
<td>yes</td>
<td>Tab over to the column specified by the argument before printing the next format specification.</td>
</tr>
<tr>
<td>COND_HDR</td>
<td>yes</td>
<td>no</td>
<td>After the current line, conditionally print CONTROL_HDR(i) where i is the argument following COND_HDR. The header is not printed if r-tree® is in the middle of outputting the associated control footers and headers because of an actual control break.</td>
</tr>
<tr>
<td>COND_PAGE</td>
<td>yes</td>
<td>no</td>
<td>If the number of lines specified by the argument will not fit on current page, issue a page break.</td>
</tr>
<tr>
<td>FORM_FEED</td>
<td>no</td>
<td>no</td>
<td>Issue a form feed after printing the line.</td>
</tr>
<tr>
<td>FORMAT</td>
<td>yes</td>
<td>yes</td>
<td>FORMAT permits the printing of a value in the i'th of N format specifications where i is the first argument, N is the second argument, and the actual value to be printed is the third. The other N - 1 format specifications are printed as blanks.</td>
</tr>
<tr>
<td>HORIZONTAL</td>
<td>no</td>
<td>no</td>
<td>Used only with first line of BODY to define a horizontal repeat template.</td>
</tr>
<tr>
<td>LEFT_MARGIN</td>
<td>yes</td>
<td>yes</td>
<td>Set left margin to argument.</td>
</tr>
<tr>
<td>NO_LINEFEED</td>
<td>no</td>
<td>no</td>
<td>Do not output a linefeed at the beginning of this line.</td>
</tr>
<tr>
<td>OUTLINE</td>
<td>yes</td>
<td>yes</td>
<td>If the control field (with level given by the first argument) has not changed, substitute blanks for the next format specification; otherwise use the second value following OUTLINE in the format.</td>
</tr>
<tr>
<td>PAGE_BREAK</td>
<td>no</td>
<td>no</td>
<td>Issue a page break after printing the line.</td>
</tr>
<tr>
<td>PAUSE</td>
<td>no</td>
<td>no</td>
<td>Issue a pause after printing the line.</td>
</tr>
<tr>
<td>RESET_PAGE</td>
<td>no</td>
<td>no</td>
<td>After printing the current line, print report footer, reset page number to 1, and resume printing as though a new report was just beginning.</td>
</tr>
<tr>
<td>RETURN</td>
<td>no</td>
<td>no</td>
<td>Print this line on top of previous line: usually used to create bold images.</td>
</tr>
</tbody>
</table>
### 9.3 Format Symbol List

<table>
<thead>
<tr>
<th>Display Logic Variables</th>
<th>Argument Required</th>
<th>Format Scan Required</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWS</td>
<td>yes</td>
<td>yes</td>
<td>Skip down N lines where N is given by the argument.</td>
</tr>
<tr>
<td>SKIP_LINE</td>
<td>yes</td>
<td>no</td>
<td>Only print line if value following SKIP_LINE is non-zero. SKIP_LINE must be the first symbol in the symbol list following the output line. The opposite of TEST_LINE.</td>
</tr>
<tr>
<td>TEST_LINE</td>
<td>yes</td>
<td>no</td>
<td>Only print line if value following TEST_LINE is non-zero. TEST_LINE must be the first symbol in the symbol list following the output line. The opposite of SKIP_LINE.</td>
</tr>
<tr>
<td>SPACES</td>
<td>yes</td>
<td>yes</td>
<td>Print N spaces before next format specification where N is the argument.</td>
</tr>
<tr>
<td>VERT_TAB</td>
<td>yes</td>
<td>yes</td>
<td>Skip down to the Nth line of the page where N is the argument.</td>
</tr>
</tbody>
</table>

*NOTE: COND_HDR and COND_PAGE cannot be used on the same output line.*

To understand the significance of the Argument Required and Format Scan Required columns of the above table, consider Figure 9-9 below:

```plaintext
IMAGE
BODY
/*
 * PAGE_BREAK (Arg/No Scan/No): a NO-NO variable can appear any
 * where on the format symbol list, does not consume the value
 * (if any) which follows it on the format symbol list, and the
 * image line may or may not have any format specifications.
 */
PAGE_BREAK
* Final Total $39,998.00
  fin_total PAGE_BREAK
* Final Total $99,998.00
PAGE_BREAK  fin_total
/
/*
 * LUMP_MULTI (Arg/Yes Scan/Yes): a YES-YES variable can appear any
 * where on the format symbol list, it does consume the value
 * following it on the format symbol list (i.e., this value does
 * not get output, but is used by the display logic variable)
 * and the output line may or may not have any format spaces.
 */
continue...
BCODE =
COND_BREAK 5
* Final Total $99,998.00
  fin_total COND_BREAK 5
* Final Total $99,998.00
COND_BREAK 5  fin_total
```

Programmer’s Reference Guide  Chapter - 9  123
Physical Device Control Strings

It is frequently necessary to send control strings to an output device to setup the device for your report, or to change some attribute during the middle of a report. The control strings can be sent via the r-tree® device strings. The values for each control string are assigned to the device string variables in the DISPLAY section of the report script (to be discussed in the "Device Strings" section below). By including the device strings in the format symbol list, you cause them to be sent to the output device.

Device strings must be followed (not necessarily immediately) by an entry in the format symbol list which is destined to be substituted into a format specification. Otherwise, they will not be sent to the output device by r-tree. A device string is sent to the output device just before the closest following field value is substituted into its corresponding format specification.

The device strings have no intrinsic interpretation nor are they assigned default values. Each device string variable can be assigned up to a thirty-two-byte control string. The maximum length of the control string can be set at compile time by adjusting MAX_DSTRING in rtdoda.h.

The names of the device strings are

- SET_INITIAL
- SETFONT0
- SETFONT1
- SETFONT2
- SET_LPP
- SET_VAR0
- SET_VAR1
- SET_VAR2
- SET_VAR3

Figure 9-9. Display Logic Examples.
Output Device Initialization

To initialize the output device, create an image line in the SETUP subsection with at least one format specification. Then place all the desired device strings in the format symbol list followed by a field value or a constant to substitute into the format specification.

```
IMAGE
  SETUP
  "NO_LINEFEED SET_INITIAL SET_LPP SET_VAR2"
```

*Figure 9-10. Sample Device Initialization.*

The SETUP subsection in Figure 9-10 will cause the control strings assigned to SET_INITIAL, SET_LPP and SET_VAR2 to be sent to the output device (followed by the empty string " "). If the format specification (" @") is omitted from the image line, no control strings will be sent. If the empty string (" ") is omitted from the format symbol list, r-tree ® will run out of entries in the format symbol list and issue a run-time error. If the NO_LINEFEED is omitted from the format symbol list, then the initialization sequence will cause a linefeed to be sent to the output device (before it is initialized).

Device String Example

Let us suppose that the display control strings SET_VAR0 and SET_VAR1 have been set up in the DISPLAY section (see "Device Strings" section below) to turn underlining on and off, respectively. Figure 9-11 shows how to underline a item in the report.

```
* @
  SET_VAR0 your_variable SET_VAR1
```

*Figure 9-11. Underline Example.*

As discussed in the "Device Strings" section, device strings are sent to the output device just before the closest following field value is substituted into its corresponding format specification. In Figure 9-11, SET_VAR0 (which turns underlining on) is sent to the output device just before your_variable is substituted into the @xxxxxxxxxxxx format. To turn off underlining, a second format specification, @, immediately follows the main format. Just before the empty string (" ") is substituted into this format, SET_VAR1 is sent to the output device. SET_VAR1 turns off the underlining.
For more complex control over physical devices, consider use of the INTERFACE function to send complex, parameterized control strings to output devices. Source code module rtljet.c contains an example of a function which can be called from the INTERFACE section of the user supplied function. See Optional User Function Pointer in “The Report Function” section of the r-tree® Operation chapter for discussion of drawing boxes and lines on a HP Laserjet Plus printer.

9.4 DISPLAY Variables

The display section of an r-tree report script allows you to modify the display attribute variables and assign device control strings to the device string variables.

Display Attribute Variables

The display attribute variables control the overall aspects of the report output such as:

- the device to which the report will be sent;
- length and width of the output page;
- which characters to use for the currency mark, thousand separator, and decimal point; and
- the order in which the components of a date will be output.

Display attribute variables are assigned small, non-negative integer values. Integer values can be specified as a constant or as a variable which evaluates to a small positive integer. For display attributes requiring a printable character, specify the decimal equivalent. The decimal equivalent can be specified by one of the Symbolic Constants: DOLLAR, DASH, SLASH, PERIOD or COMMA.

<table>
<thead>
<tr>
<th>Display Attribute</th>
<th>Default</th>
<th>Special Interpretation</th>
</tr>
</thead>
</table>
| DEVICE            | 3       | 0, 1, 2 - printers  
                  |          | 3 - screen  
                  |          | 4 - disk file  
                  |          | 5 - c-tree® data file  
                  |          | 6 - no device at all  
                  |          | 7 - append to disk file  
                  |          | 8 - append to c-tree data file  
| PAGE_WIDTH        | 132     | applies to all output devices except the screen; for printers and stream disk file, it gives the maximum line width not counting the left margin (see display logic variables); for the c-tree® Plus data file, it specifies the record length. |
| PAGE_LENGTH       | 66      | total physical number of lines per page. It applies to all output devices except the screen. A value of zero implies no automatic pagination. |
| SCREEN_WIDTH      | 80      | applies only to the screen |
## 9.4 DISPLAY Variables

<table>
<thead>
<tr>
<th>Display Attribute</th>
<th>Default</th>
<th>Special Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCREEN_LINES</td>
<td>24</td>
<td>applies only to the screen. A value of zero implies no automatic pagination.</td>
</tr>
<tr>
<td>DATE_ORDER</td>
<td>0</td>
<td>modify month, day and year sequence in literal date strings: 0: &quot;mm/dd/yy&quot; 1: &quot;dd/mm/yy&quot; 2: &quot;yy/mm/dd&quot;</td>
</tr>
<tr>
<td>CURRENCY_MARK</td>
<td>36</td>
<td><code>$</code></td>
</tr>
<tr>
<td>THOUSAND_SEPARATOR</td>
<td>44</td>
<td><code>,</code></td>
</tr>
<tr>
<td>DECIMAL_POINT</td>
<td>46</td>
<td><code>.</code></td>
</tr>
<tr>
<td>DATE_SEPARATOR</td>
<td>47</td>
<td><code>/</code></td>
</tr>
<tr>
<td>LINE_MARK</td>
<td>43</td>
<td><code>+</code></td>
</tr>
<tr>
<td>FORMAT_MARK</td>
<td>64</td>
<td><code>@</code></td>
</tr>
<tr>
<td>PCNTRL_MARK</td>
<td>94</td>
<td><code>^</code></td>
</tr>
<tr>
<td>FIRST_PAGE_HDR</td>
<td>1</td>
<td>if zero, then 1st page has no page header</td>
</tr>
<tr>
<td>LAST_PAGE_FTR</td>
<td>1</td>
<td>if zero, then last page has no page footer</td>
</tr>
<tr>
<td>ZERO_FILL</td>
<td>42</td>
<td><code>*</code></td>
</tr>
<tr>
<td>BODY_COUNT</td>
<td>0</td>
<td>if non-zero, maximum number of record groups selected</td>
</tr>
<tr>
<td>TEST_PAGES</td>
<td>0</td>
<td>if non-zero, the number of times the IMAGE section is output to test the output alignment</td>
</tr>
<tr>
<td>STARTING_PAGE</td>
<td>0</td>
<td>if non-zero, the first page sent to the output device</td>
</tr>
<tr>
<td>ENDING_PAGE</td>
<td>0</td>
<td>if non-zero, the last page sent to the output device</td>
</tr>
</tbody>
</table>

When specifying a PAGE_LENGTH for a physical device, specify the total number of physical lines lines that fit on a page. r-tree® automatically adjusts for the length of your PAGE_HDR and PAGE_FTR sections to determine the effective page length.

See the Raw Dump Format section of this chapter for a detailed discussion of assigning a fixed-length c-tree® Plus data file as the DEVICE.

To modify these display attributes, say to use a comma for the decimal point and a period for the thousands separator, see Figure 9-12:

```
DISPLAY  DECIMAL_POINT  COMMA  /\ COMMA = 46  /\  
         THOUSAND_SEPARATOR PERIOD  /\ PERIOD = 46  /\  
```

*Figure 9-12. Display Section Example.*
To change display attributes LINE_MARK, FORMAT_MARK, PCNTRL_MARK, DATE_ORDER and DATE_SEPARATOR, their new values must be specified by literal constants (e.g., 44), not symbolic constants (e.g., COMMA). Further, the DISPLAY section must come before the IMAGE section; and if the VIRTUAL section includes virtual fields with literal date constants, then the DISPLAY section must also precede the VIRTUAL section.

If LINE_MARK, FORMAT_MARK, PCNTRL_MARK, DATE_ORDER and/or DATE_SEPARATOR have been modified, then the IMAGE section must reflect these changes. For example, if the LINE_MARK has been changed to a period, then each image line must begin with a period in the first column. However, if one of the attributes appearing in a numeric format specification has been changed, you must still use the original symbol as described in Numeric Formats of the "Format Specifications" section above.

Modifying Display Attributes from Your Application

If you wish to modify a display attribute from your application program, you may do so with a C assignment statement of the form:

```
attribute_name = attribute_value;
```

where your application has included the following at the beginning of your application:

```
#include "rtdoda.h"
extern COUNT dtype[];
```

`rtdoda.h` contains `#define`'s for each of the display attribute variables. For example, to make the zero fill character a vertical bar, directly from your application, the assignment statement would look like:

```
ZERO_FILL = '|' ;
```

The DISPLAY section will override any display attribute assignment made from your program. The assignments from your program only override the defaults shown above.

Device Strings

The device strings are used to send control strings to the physical output device. They are the only r-tree® entities which do not have a defined usage. You may set each of the nine device strings to any arbitrary byte string up to thirty-two bytes in length, and you completely control when these control strings are sent to the output device. The assignment of the byte strings is accomplished in the DISPLAY section. The placement of the device strings in the format symbol list determines when they are sent.
The device strings are initialized to empty (i.e., zero length) strings. To assign a control string, simply list the DECIMAL values of each byte of the control string after the device string symbolic name.

```c
DISPLAY
SET_LFP  27 COMMA 65
/* equivalent to: ESC , A */
```

Figure 9-13. Device String Example.

Note in Figure 9-13 that the values must either be in decimal form, or a symbolic constant.

Ordinarily, FORM_FEED is a display logic variable. In the rare instance that one must modify the byte(s) sent to issue a form feed, such a modification can be performed in the DISPLAY section: simply treat FORM_FEED as a device string for the purpose of assigning a byte string.

Modifying Device Strings from Your Application

If you wish to modify a device string from your application program, you may do so with C assignment statements of the form illustrated in Figure 9-14.

```c
#include "rtcode.h"
extern TEXT dstring[MAX_DSTRING + 1]:
    ...
dstring[variable_name][0] = n; /* string length */
dstring[variable_name][1] = 1st_byte:
    ...
dstring[variable_name][n] = nth_byte:
```

Figure 9-14.
`rtdoda.h` contains `#define`'s for each of the device variable names. For example, to make the same assignment shown in the previous section in your application program, the assignment statements would appear as follows in Figure 9-15:

```c
#define SET_LPPH10 3
#define SET_LPPH11 27
#define SET_LPPH12 44
#define SET_LPPH13 65
```

*Figure 9-15.*

The DISPLAY section will override any device string assignment made from your program. The assignments from your program only override the defaults shown in Figure 9-15.

**Adding New Device String Variables**

In order to increase the number of device control strings, you must follow these steps which apply to the `rtival.h` module:

1) Find the last entry in the initialization of the TEXT array `dstring[][MAX_DSTRNG + 1]` which should be as shown in Figure 9-16:

   ```c
   (0) /* not used */
   ```

   *Figure 9-16.*

2) Insert before this last entry the initial values for the new control string where the first value is a count of the number of bytes in the control string. For example, to initialize the new control string to the three-byte sequence: ESCAPE 'A' 'B' , your entry will be like Figure 9-17 (along with the last entry):

   ```c
   C3, 27, 65, 66
   ```

   *Figure 9-17.*
where the "3" indicates a three-byte string, and the numbers following the "3" are the decimal equivalents of the ASCII characters. To initialize the new control string to an empty string, simply add an entry of the form illustrated in Figure 9-18:

```
 0 , /* new control */
 0 , /* not used */
 3 ;
```

Figure 9-18.

3) Find the last DSTRING entry in the initialization of the KEYWORD structure array kw[] which will look something like Figure 9-19:

```
"SET_VAR3", DSTRING,
```

Figure 9-19.

4) Append after this entry, the unique name of the new control string along with the symbolic constant DSTRING followed by a comma (','). The name must begin with a letter and may be comprised of other letters, numbers and underscores as shown in Figure 9-20.

```
"SET_VAR3", DSTRING, 
"NEW_CONTROL", DSTRING,
```

Figure 9-20.

5) Go back to the entry made as part of Step 2 above, and verify that the new initialization occurred just after the initialization entry which corresponds to the last entry found in Step 3 above. In our example, we would check that our entry in Step 2 occurred after the SET_VAR3 initialization as illustrated in Figure 9-21.

```
 0 ; /* set_var3 */
 0 , /* new control */
 0 , /* not used */
 3 ;
```

Figure 9-21.
When the format symbol list contains no display logic variables nor device strings, then the entries in the symbol list are simply substituted into the corresponding image line’s format specifications from left to right. Extra entries in the symbol list will simply not be used. If there are not enough entries in the format symbol list, r-tree® will issue an error. Any format specifications which fall completely beyond the page width are ignored.

When display logic variables and/or device strings are included in the format symbol list, the substitution of entries from the format symbol list is not quite as obvious. r-tree follows these rules:

1) send literal characters to the output device until:
   a) the end of the image line is encountered; or
   b) the “page” width is exceeded; or
   c) a format specification is encountered;
   whichever comes first;

2) if step 1 terminated in:
   a) step 1a or 1b, the image line processing is complete;
   b) step 1c, then go to step 4 if the format is a raw dump (@D); otherwise go to step 3.

3) get the next entry in the format symbol list:
   a) if the entry is a display logic variable, process the variable and its argument(s) if required, and repeat step 3;
   b) if the entry is a device string, send its string of control bytes to the output device (unless the output device is a c-tree® Plus fixed-length file in which case the control string is ignored), and repeat step 3;
   c) if there are no more entries, issue a run-time error;
   d) otherwise, substitute the value of the entry into the format specification, send the result to the output device, and go back to step 1.

4) get the next entry in the format symbol list:
   a) if the entry is a display logic variable, process the variable and its argument(s) if required, and repeat step 4;
b) if the entry is a device string, skip it and repeat step 4;

c) if there are no more entries, the image line processing is complete.

d) otherwise, send the field’s raw bytes to the output device, and repeat step 4.

NOTE: See the Display Logic Variables section of this chapter. Those marked no in the Format Scan Required column will be effective even if they are not encountered in steps 3a or 4a above.

### 9.6 Debit/Credit Output

*r-tree* provides two basic methods to output financial data in terms of debits and credits: single column and double column.

#### Single Column Output

If your application requires the output in a single column, then you should use the appropriate format specification to show the negative values as either debits or credits.

For example, if you are listing the dollar values of a series of assets (which have positive debit balances, and negatives are shown as credits), then your IMAGE section might look like the example found in Figure 9-22:

```
IMAGE
CONTROL_HDR(
)*
   Asset_Name Asset_Value
BODY
)*
   #9,9999990E00CR
   asset_name asset_value
```

*Figure 9-22. Single Column Debit Output.*

The numeric format specification (@9,9999990E00CR) in the example above will be blank when asset_value is zero, will have a value followed by CR when asset_value is negative, and will have the CR suppressed when asset_value is positive.

Single column output for accounts which carry positive credit balances is handled by replacing the CR by DR at the end of the numeric format specification. Then negative values will be followed by the debit mark DR.
Debit Credit Columnar Output

If your application requirements debits and credits to be shown in separate columns, you may take advantage of the FORMAT display logic variable which permits one of several format specifications to be selected for a given output value.

In Figure 9-23, negative asset values will be printed in the credit column, and non-negative values will be printed in the debit column.

### 9.7 Aged Account Balances

It is often necessary to compute balances which are classed into date ranges. For example, a balance due account may be divided into the following classes: current, thirty to sixty days, and over sixty days. In the following example, we assume that all payments and charges are entered in the customer transaction (detail) file in the `ct_amount` field. Payments are entered as negative quantities and charges as positive quantities. We also assume that payments will be applied to the oldest charges first. The system date (`SYS_DATE`) is assumed to be the report date.

The basic technique used in the example is to classify each value in `ct_amount` into the `paid`, `current`, `over30` and `over60` virtuals. Accumulators are used to total these values. Finally, virtuals which depend on the accumulators are used to spread the total payments over the total charges for each time class to determine the balances by time class. Notice that to ensure that the virtuals use the accumulator value associated with each customer number control break (control level 1), we use the `Ai[j]` notation to reference the
accumulator values. (see "Field Values" in Section 9.3 above for a discussion of the A[ij] notation.)

The important values computed in Figure 9-24 are:

- \( \text{bal}_\text{tot} \) the total balance due (or credit balance)
- \( \text{bal}_\text{cur} \) current balance (30 days or less)
- \( \text{bal}_\text{30} \) balance due over 30 days and less than 61 days
- \( \text{bal}_\text{60} \) balance due over 60 days old

9.8 Report Chaining

It is sometimes necessary to call the report function more than once in order to achieve the desired results. Usually, the output of the first call to report is in the form of a temporary c-tree® Plus data file which serves as the input to the second report.
The Double Sort

The most common reason for chaining two report scripts is when the final result is based on the sorted output of the first script, and there is no way to sort the data appropriately in the first script itself.

In our customer order example, consider how we might produce a report which shows items sorted in descending order by total sales. Further, assume that we do not have an index based on item number over the order-item detail records. Then to determine sales by item, we must sort the order-item detail records by item number and accumulate the total sales for each item. These results can be output to a temporary c-tree® data file. Then a second report script can read this file for input, perform a sort over the total sales figures, and produce the desired output.

In Figure 9-25 we present the report scripts, "temporary" parameter file, and DODA modifications required to produce a report of items sorted by total sales. (If we had taken advantage of the item number index which actually does exist, we could have generated this output with only one script.)

```
START
VIRTUAL
  item_master   STRING 32   "itewnast.dt"
  order_item    STRING 32   "ordritem.dt"
  extended_price MONEY 4     ol_quantity * in_itempric
SEARCH
  FILE order_item ALL
  FILE item_master JOINS TO order_item BY FIELD ol_itemnumb
  USING KEY in_itemnumb_idx
SELECT
  ALL
  SORT USING KEY
  NO_MDO ol_itemnumb /* sort detail records by item number */
CONTROL
  1 ol_itemnumb
ACCUMULATOR
  item_total SUM extended_price
DISPLAY
  DEVICE INFILE */ create fixed length data file */
  PAGE_WIDTH 13 */ record length of fixed file */
IMAGE
  CONTROL_FTR(1)
  */?
C11 item_total
EXIT
```

Figure 9-25. First Script - Determine Sales By Item.
The output generated by this first script consists of pairs of item numbers and total sales amounts. This information is in the c-tree® Plus fixed length data file "RTREE.OUT" in binary form since the raw dump format was used. It is important to emphasize that there is no padding between the fields when a raw dump is performed. This effects the way the record structure should be setup when reading in the RTREE.OUT data for the second script. The record structure for RTREE.OUT is illustrated in Figure 9-26:

```
struct {
    TEXT itm(S);
    TEXT nai(B);
} tmp;
```

Figure 9-26.

There are two noteworthy points to be made here. First, in order to set up this structure, we had to know the length of each field as output by the raw dump format. By referring to "How To Setup a DODA" in the DODA chapter, we know that the item number is five bytes. From the "Accumulator" section of the Report Script chapter, we know that a SUM accumulator is maintained as a double precision float which requires eight bytes.

```
struct {
    TEXT itm(S);
    double nai;
} bad;
```

Figure 9-27.

Second, we DID NOT use a structure of the form shown in Figure 9-27 because the raw dump format does not pad the fields to ensure word or double word alignment of numeric fields. This does not cause a problem since r-tree® only performs numeric operations on copies of the fields, and the copies are properly aligned. It does mean that structure bad above may not correspond to the record contents since the sal field may not start immediately after the itm field in the structure (many compilers ensure that floats start on word or double word boundaries and insert padding to accomplish this) whereas there is no padding in the actual data record.
To reference the contents of the temporary c-tree® data file, we must augment our original DODA as illustrated in Figure 9-28 (see “How To Setup A DODA” in the DODA chapter) with two new field definitions.

```
("tmp_itemnumb", tmp_item_num),
("tmp_sales", tmp_sales, RTD_FLOAT),
(NULL, NULL, 0, 0, -1)
```

*Figure 9-28.*

The parameter file for the second report script includes the temporary data file and the item master file. Its contents (which may be compared with the Customer Order Parameter File found in the “ISAM Parameter File” section of the c-tree Parameter Information chapter) are as shown in Figure 9-29:

```
10 1 4 2
 0 "RTREE.OUT" 13 0 0 0 tmp_itemnumb tmp_sales
 1 itemreact.it 13 0 5 1 im_itemmght im_itemmlenc
 2 itemwant.ix 5 0 0 0 1 0 0 1im_itemnumb_idx
 8 5 2
```

*Figure 9-29.*

The second report script shown in Figure 9-30 sorts the item records by total sales in descending order, and outputs the results to the screen:

```
START
SEARCH "RTREE.OUT" ALL
  FILE "itemwant.it" JOINING TO "RTREE.OUT"
  BY FIELD tmp_itemnumb USING KEY im_itemnumb_idx
SELECT ALL
SORT USING KEY
  DSC NO MOD tmp_sales
&ACUMULATOR
  total_sales SUM tmp_sales
IMAGE
  PAGE_HDR
  + * Item#   Item Description  Total Sales
```

r-tree® Report Generator  Chapter - 9  138
The result of running these two report scripts one after the other is shown below:

<table>
<thead>
<tr>
<th>Item#</th>
<th>Item Description</th>
<th>Total Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>13000</td>
<td>10,000 GPH Circulating Pump</td>
<td>44,640.00</td>
</tr>
<tr>
<td>12800</td>
<td>Stabilizing Base</td>
<td>6,650.00</td>
</tr>
<tr>
<td>12181</td>
<td>1/4&quot; Rotating Shield</td>
<td>6,440.00</td>
</tr>
<tr>
<td>13180</td>
<td>Pump Valve Assembly</td>
<td>4,350.00</td>
</tr>
<tr>
<td>13200</td>
<td>Pump Coupling</td>
<td>2,751.25</td>
</tr>
<tr>
<td>12800</td>
<td>Inertial Platform Mount</td>
<td>875.00</td>
</tr>
</tbody>
</table>

Figure 9-31. Chained Report Output.

9.9 PRV/LST: Previous and Last Accumulators

It is not always easy to see the difference between a PRV accumulator and a LST accumulator. Basically, a LST accumulator has the most recent value for the given variable. The LST accumulator is useful when you want to print a value, other than the routine totals or frequencies maintained by other accumulators, in a control footer. The reason the LST accumulator is required is that by the time r-tree® determines that a footer is required, it has already read the next data record(s), possibly clobbering the value you want to output. The LST accumulator keeps a copy of this value safe. The Control Footers portion of the "Image" section of the Report Script chapter contains a more complete discussion of this topic.

The PRV accumulator has the value prior to the most recent value. However, it is not often used in control footers. Instead, it can be used to determine if there is a break in
a sequence or to calculate the difference between successive values. Most often, you will use the overall report component of the PRV accumulator. It is not reset at each control break.

NOTE: a PRV accumulator does not have a valid value until the second iteration of the specified control break level. STRING valued PRV accumulators are blank until the second iteration (at which time they contain the value from the first iteration). Numeric valued PRV accumulators are set to the floating point equivalent of negative infinity until the second iteration.

The example in Figure 9-32 contains an example of the PRV accumulator used to check if a sequence of check numbers is interrupted. We assume that the field `check_no` contains an ASCII string image of the check number.

```
START
SEARCH
FILE "check.lst" ALL
ACUMULATOR
  /* check_no is a field in the "check.lst" file which contains a six byte string image of the check number*/
  prev_check PRV check_no
VIRTUAL
  /* convert check_no to integer */
  check_val INT4 4 STR_LEN(check_no)
  /* prev_check_no is required since an accumulator cannot be a function argument */
  prev_check_no STRING 6 #B,03
  prev_check_val INT4 4 STR_LEN(prev_check_no)
  /* COUNTER ensures that first check is not flagged */
  test 1 INC 2 LMINVAL = 0 MNU
  (check_val - prev_check_val) > 17 YES : NO
IMAGE
  REPORT_HDR
  * Check Listing
  * Check Number (Missing Check Flagged With *)
  = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
  BODY
  * Rrrrrrrr
  * INCDEMENT(check_no)
  ** TEST_LINE test MD_LINEFEED
  /* A "-" is printed next to check numbers with missing predecessors */
EXIT
```

Figure 9-32. PRV Example.
Ordinarily, given a master file A with two detail files B and C, r-tree\(^6\) cannot list the contents of both B and C. r-tree scripts can, using the SET functions, perform computations and boolean evaluations over one detail file while listing the contents of the other. However, r-tree scripts can be made to list both bodies, but the processing is inefficient.

Assume the following relationships: field \(F_{ab}\) in file A links file A to file B; field \(F_{ac}\) in file A links file A to file C; field \(F_{bu}\) is a unique, 8 byte string field in file B (i.e., no two records from B have the same value for \(F_{bu}\)); file B is indexed by \(key_B\); field \(F_{c}\) is a field of interest from file C; and file C is indexed by \(key_C\). Ordinarily, the SEARCH section for A, B and C would appear as in Figure 9-33:

![Figure 9-33. Typical Two-detail-file Search Section.](image)

To force a listing of both detail files, we use the proforma script of the example shown below in which file C is made a detail for B (instead of A), file B is printed as part of a header, and file C is printed in the body.

NOTE: file C can be a detail for file B even though B and C might not share any fields in common because the field from file A (\(F_{ac}\)) is still used to access file C.

By placing two trivially satisfied equalities in the SELECT statement as in Figure 9-34, we are sure of selecting all records and forcing a prepass for the computation of \(lastbu\) which depends on the LAST set function. The "trick" in this approach is that records from C only print after the last record from B.

![Figure 9-34. Proforma Script for Listing Both Detail Files.](image)
9.11 Position Dependent Layout Control

Through the use of display logic variables (e.g., COND_PAGE, VERT_TAB, etc.), you may control the layout of your report contents. However, sometimes it is necessary to introduce very complex layout requirements. By combining the TEST_LINE and SKIP_LINE display logic variables with the COND_PAGE, VERT_TAB and NO_LINEFEED display logic variables, you can introduce very complex control. If the arguments of the TEST_LINE and SKIP_LINE variables depend on the LINE_NO status variable, this control will be position dependent. Figure 9-35 shows how to combine TEST_LINE, LINE_NO, NO_LINEFEED and VERT_TAB to print some output either at the top or bottom of the page, but not the middle.

Figure 9-35. Conditional Vertical Tab.

The first image line of the BODY section of Figure 9-35 contains a format specification (\@) because (see the Display Logic Examples illustration found in the “Format Symbol List” section of this chapter) VERT_TAB requires a format scan in order to be active. The empty quotes at the end of the first format symbol list combined with the NO_LINEFEED ensure that this line does not result on in any output except for the conditional vertical tab.
10. r-tree Alphabetical Reference

This alphabetical reference provides a quick look-up capability for r-tree® features, keywords and some of the important concepts appearing in the text.

ACCUMULATOR

Report script section heading. In the ACCUMULATOR section you define automatic accumulators which can tally the minimum, maximum, sum, average, frequency count, first occurrence and last occurrence of specified fields during report processing. These tallies are maintained for each control break level defined in the CONTROL section of the report script as well as for each page and the overall report.

ADDTO

ADDTO is a built-in function which permits a virtual field to act as an accumulator. In most situations, it is easier and more appropriate to use a standard r-tree ACCUMULATOR; however, there may be situations in which the ADDTO function increases your flexibility. ADDTO simply returns its input argument without any modification; but it signals r-tree that the value computed for the virtual field should be added to the current contents of the virtual instead of replacing the current contents. For example, the script fragment in Figure 10-1 shows how a virtual can be initialized and added to:

```
SEARCH
  FILE "custmstr.dat" USING_KEY cr_custnumb_idx (beg_no end_no)
  FILE "transact.dat" IS_DETAIL FOR "custmstr.dat" BY FIELD cm_custnumb USING_KEY tr_custnumb_idx

VIRTUAL
total DFLOAT 0
  READ Level = YES ? tr_transact : ADDTO(tr_transact)
  /* Whenever a new customer master record is read (which sets */
  /* READ Level equal to YES), the virtual field total is set to */
  /* the first transaction amount for the customer; otherwise the */
  /* transaction amount will be added to total. */
```

Figure 10-1. Addto Example.

ALIAS

ALIAS is a synonym for IS_ALIAS_FOR.

ALL

ALL is used in the SEARCH section to designate that you want to search the entire primary file in physical, sequential order.

ALL is also used to replace a boolean expression in the SELECT section of the report script when you intend to retrieve all possible records from the left-most path of the file hierarchy defined in the SEARCH section of the report script. The same affect may be achieved by omitting the SELECT section altogether.
ALT_DSC
ALT_DSC is used in the SORT section to indicate that a sort key segment should be
collated in descending order using an alternative collating sequence. An alternative
collating sequence changes the order in which values are sorted. The alternative
collating sequence is specified by setting global variable altseq as specified in c-tree®

ALT_SEG
ALT_SEG is used in the SORT section to indicate that a sort key segment should be
collated using an alternative collating sequence. The alternative collating sequence is
specified by setting global variable altseq as specified in the c-tree Plus Programmer’s
Reference Guide.

AND
AND is a logical connective. expr1 AND expr2 is true if and only if the boolean
expressions expr1 and expr2 are both true.

associated file

r-tree® permits two or more files to be logically joined together in two distinct ways:
through the JOINS_TO and IS_DETAIL_FOR relationships.

In a JOINS_TO relation, the associated file is the file which contains some sort of code
or other field which is the basis of the joining. The other file is usually a master file
which contains the information corresponding to the code in the associated file.

In an IS_DETAIL_FOR relation, the associated file is also referred to as a parent file.
It contains a code which is repeated in one or more detail records of the detail file. All
of the detail records containing the matching code belong to the record in the associated
file.

AVERAGE
AVERAGE [expr] is a computational set function which returns the average value of
the computational expression expr evaluated over the set of detail records related to
expr. The computational expression cannot be string valued.

AVG
AVG is an automatic accumulator type. It is used in the ACCUMULATOR section of
the report script to designate tallies based on a computed average. If the field to be
averaged is string valued, then the strings are converted to numeric values before
computing the average.

BODY
BODY is a subsection of the IMAGE section of the report script. If present, the BODY
section is output for each group of records which satisfies the SELECT expression.

BODY_COUNT
BODY_COUNT is a display attribute variable whose value is set in the DISPLAY
section of the report script. BODY_COUNT defaults to zero. If BODY_COUNT is
larger than zero, then it serves as an upper limit on the number of groups of records
which satisfy the SELECT expression and are passed to the IMAGE section for output.
Therefore, BODY_COUNT permits the report output to be truncated to only the first
N groups of output where N is the value assigned to BODY_COUNT in the DISPLAY
section.
bolexp

bolexp refers to a boolean expression. In r-tree® a boolean expression is comprised of terms which may be:

- computational expressions which are compared via the relational operators: <, <=, =, <>, >, >=;
- nested boolean set functions whose arguments are themselves boolean expressions;
- parenthesized boolean expressions;
- connected and/or modified by the boolean operators: AND, NOT and OR.

BY

BY is a synonym for the BY_FIELD keyword defined below.

BY_FIELD

The symbolic reference which follows the BY_FIELD keyword specifies the field (either a data field or a virtual field) which contains the target key value for the index which links a file with a subsidiary file.

When two files are "joined" in the SEARCH section, then the field defines the target key value for an exact match search of the subsidiary file. When the subsidiary file acts as a detail file, then the field defines the target key value for a partial match search of the subsidiary file.

CHAR / CHARU

CHAR and CHARU are one-byte virtual field types. They are used in the VIRTUAL section of the report script to define the type of a virtual field. CHAR is a signed byte and CHARU is an unsigned byte.

If you wish to have a one-byte character treated like a string valued entity, use the STRING virtual field type with a length of one.

cmpexp

cmpexp refers to a computational function. In r-tree a computational expression is comprised of terms which may be

- constants;
- data and virtual fields;
- built-in functions applied to data or virtual fields;
- status variables maintained by r-tree;
- nested computational set functions whose arguments are computational expressions themselves;
- parenthesized computational expressions; and
- connected via the computational operators: +, -, *, /, and # (string concatenation).

COLUMN

COLUMN is a display logic variable which may appear one or more times in one or more format symbol lists in the IMAGE section of the report script. COLUMN acts as
a tab for the next format specification. The value following COLUMN specifies the beginning column position for the next format specification. COLUMN will be ignored if the current output position is greater than the tab position.

**COMMA**

COMMA is a symbolic constant which evaluates to the decimal equivalent of an ASCII comma (','). COMMA may be used anywhere a value is required except if the value must be in the form of a literal constant.

**COND_HDR**

COND_HDR is a display logic variable which may appear in one or more format symbol lists of the IMAGE section. It is most commonly found in the format symbol list of the last line of a PAGE_HDR subsection. COND_HDR must be followed by a value which specifies a control break level. If r-tree® is not in the middle of processing footers and/or headers associated with the specified control break level (or one of the higher control break levels), then the control header section associated with the specified control break level will be output after the current line is output. This is useful to print column headings on the top of a continuation page. COND_HDR cannot be used on the same line with COND_PAGE.

**COND_PAGE**

COND_PAGE is a display logic variable used in one or more format symbol lists in the IMAGE section of the report script. The value following COND_PAGE in the format symbol list specifies the number of lines which must fit on the current page; otherwise a page break is issued before continuing with the output. COND_PAGE cannot be used on the same line with COND_HDR.

**CONTROL**

CONTROL is a report script section heading. In the CONTROL section, the control break fields are specified along with their relative control break levels. The lower the control break level, the wider the scope of the associated control field. The lowest control break level you can define in the CONTROL section is one. There is no limit on how many levels you may have above the first level.

**control break / control field**

A control break occurs whenever a control field changes value. A control field is defined in the CONTROL section of the report script. When a control break occurs, r-tree® automatically updates accumulators as necessary, and outputs the appropriate control footers and control headers. With the exception of the page break, the r-tree® control fields and control breaks are nested. That means that when a high level control field changes, it automatically causes all lower levels to be treated as though they changed.

**CONTROL_FTR** / **CONTROL_HDR**

CONTROL_FTR(i) and CONTROL_HDR(i) are subsection headings for the IMAGE section of the report script. Unlike the other IMAGE subsection headings, CONTROL_FTR and CONTROL_HDR require "subscripts" designated by the parenthesized 1’s. In actual use the parenthesized 1’s must be small integers which correspond to the control levels assigned in the CONTROL section of the report script.

It is not necessary to have matching CONTROL_HDR’s and CONTROL_FTR’s, nor is it necessary to have them for each control break level.

**COUNTER**

COUNTER is a status variable maintained by r-tree®. Each time the ZERO function is executed, COUNTER is reset to zero. Each time the INCREMENT function is
executed, COUNTER is increased by one. Each time the DECREMENT function is executed, COUNTER is decreased by one. COUNTER can be used to generate sequential numbers for various reporting purposes.

CURRENCY_MARK

CURRENCY_MARK is a display attribute used in the DISPLAY section of the report script. CURRENCY_MARK defaults to the decimal equivalent of an ASCII dollar sign (‘$’). If you wish monetary format specifications to print a symbol other than the dollar sign, you must specify another decimal equivalent in the DISPLAY section.

cvalue

cvalue refers to a constant value. r-tree supports the following constants: string, integer, floating point, date and money.

C1, C2, C3, C4

C1 through C4 are four additional status variables which are used in the same manner as the COUNTER status variable except that the functions Zx, Ix, and Dx (x=1,2,3,4) are used to zero, increment and decrement the status variables, respectively.

DASH

DASH is a symbolic constant which evaluates to the decimal equivalent of an ASCII dash (‘-’). DASH may be used anywhere a value is required except if the value must be in the form of a literal constant.

data object definition array

The data object definition array (DODA) is the run-time data dictionary used by r-tree to reference your data fields and any other report parameter that you wish to use in your report script. The DODA must NOT contain entries for virtual fields, accumulators or other entities defined in the report script.

DATE

DATE is a virtual field type. It is used in the VIRTUAL section to specify a virtual field which contains a four-byte integer which represents a date in a computational form. r-tree can easily move between DATE valued fields and their more traditional string valued equivalents (e.g., “10/21/86”).

DATE_ORDER

DATE_ORDER is a display attribute used in the DISPLAY section to specify the order in which the year, month and day will (should) appear in output (input) when converting between a computational date and an ASCII string. DATE_ORDER defaults to zero. DATE_ORDER values affect the order as follows:

0 - "mm/dd/yy"
1 - "dd/mm/yy"
2 - "yy/mm/dd"

If DATE_ORDER is assigned an alternative value (1 or 2), it must be done in a display section defined prior to any use of date-valued fields, and the value must be assigned with a literal constant, not a symbolic constant or field value.

DATE_SEPARATOR

DATE_SEPARATOR is a display attribute whose value is set in the DISPLAY section. DATE_SEPARATOR defaults to the decimal equivalent of a (forward) slash (‘/’). If you wish the separator between the year, month and day numbers in date input/output to be other than a slash, you must change DATE_SEPARATOR to the decimal
equivalent of the ASCII character you wish to use. The value used to reassign
DATE_SEPARATOR must be a literal constant, not a symbolic constant or field value.

**DAT_DAY**
Built-in function to convert a date in computational form to an ASCII string with the
name of the day of the week.

**DAT_DD**
Built-in function to convert a date in computational form to an ASCII string with the
day of the month represented from "01" to "31".

**DAT_MM**
Built-in function to convert a date in computational form to an ASCII string with the
month of the year represented from "01" to "12".

**DAT_MTH**
Built-in function to convert a date in computational form to an ASCII string with the
name of the month.

**DAT_YYYY**
Built-in function to convert a date in computational form to an ASCII string with the
year represented as a four digit number such as "1987".

**DBL_STR**
DBL_STR is a built-in function which returns an ASCII string representation of the
double precision floating point argument.

**DECIMAL_POINT**
DECIMAL_POINT is a display attribute whose value is set in the DISPLAY section.
DECIMAL_POINT defaults to the decimal equivalent of an ASCII period ("."). If you
change DECIMAL_POINT to the decimal equivalent of another ASCII character, it
will be used in place of the period in the output of numeric fields.

**DECREMENT**
DECREMENT is a built-in function whose side-effect is more important than its return
value. DECREMENT returns its argument unmodified, but, more importantly, causes
the status variable COUNTER to be decreased by one.

**DELIMITER**
DELIMITER is a symbolic string constant set to the ASCII string "^". It is used by the
built-in functions LFT_STR and RGT_STR which return the left part of the argument
string up to but not including the DELIMITER, and the right part after the DELIM-
ITER, respectively. You can use DELIMITER in string concatenation operations to
put operands together that you will want to separate later.

You can modify the value of delimiter from your r-tree® interface program by
including the illustration in Figure 10-2:

```c
extern TEXT rtdlstr();

rtdlstr("1") = decimal Equivalent;
```

Figure 10-2.
where decimal Equivalent is the numeric value of the character you want to use for the DELIMITER. You may even make the DELIMITER a non-printing character.

detail file

A detail file contains data records, sets of which can be mapped back to a specific record in its parent file. Detail files are used instead of repeating groups. For example, a customer order record might be designed so that it could hold information on up to twelve items, where each item is a member of the item repeating group. Both c-tree® and r-tree® are designed to use a detail file to represent a repeating group: in our example, the ordered items would be placed in the order-item detail file instead of the repeating group.

DEVICE

DEVICE is a display attribute whose value is set in the DISPLAY section. DEVICE defaults to the value 3. DEVICE controls where the report output is routed. The alternative values for DEVICE are shown in Figure 10-3:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PRINTER0</td>
</tr>
<tr>
<td>1</td>
<td>PRINTER1</td>
</tr>
<tr>
<td>2</td>
<td>PRINTER2</td>
</tr>
<tr>
<td>3</td>
<td>SCREEN</td>
</tr>
<tr>
<td>4</td>
<td>OUTPUT (a stream i/o disk file)</td>
</tr>
<tr>
<td>5</td>
<td>TRUNCFILE (a fixed length c-tree data file)</td>
</tr>
<tr>
<td>6</td>
<td>NODEVICE (no output device)</td>
</tr>
<tr>
<td>7</td>
<td>OUTPUTN (append to stream file)</td>
</tr>
<tr>
<td>8</td>
<td>TRUNCN (append to c-tree data file)</td>
</tr>
</tbody>
</table>

Figure 10-3.

DFLOAT

DFLOAT is a virtual field type. It is used in the VIRTUAL section to specify a virtual field which contains a double precision floating point value.

DISPLAY

DISPLAY is a report script section HEADING. The DISPLAY section is used to modify the display attributes and/or to assign control strings to the device control variables. There may be any number of DISPLAY sections in a single report script.

DODA

DODA is an abbreviation for data object definition array.

DOLLAR

DOLLAR is a symbolic constant which evaluates to the decimal equivalent of an ASCII dollar sign (‘$’). The symbolic constant can be used anywhere a value is required except when a literal constant is required.

DSC_INTEGER

DSC_INTEGER is a sort segment type used in the SORT section of the report script. DSC_INTEGER indicates that the segment should be treated as an unsigned integer to be sorted in descending order. With c-tree V4.3 or later, this sort mode is not necessary.
DSC_NO_MOD is a sort segment type used in the SORT section of the report script. DSC_NO_MOD indicates that the segment should be used without any modification, and is to be sorted in descending order.

DSC_UPPER is a sort segment type used in the SORT section of the report script. DSC_UPPER indicates that the segment should be translated to upper case, and is to be sorted in descending order.

DTE is a synonym for IS_DETAIL_FOR_ERROR.

DTN is a synonym for IS_DETAIL_FOR_NULL.

DTS is a synonym for IS_DETAIL_FOR_SKIP.

D1 through D4 are built-in functions which decrement the status variables C1, C2, C3 and C4, respectively. These functions do not alter their input arguments. See the DECREMENT built-in function.

ENDING_PAGE is a display attribute whose value is set in the DISPLAY section. ENDING_PAGE defaults to zero. If non-zero, ENDING_PAGE is the last page number which will be output for the report.

ENDING_PAGE and STARTING_PAGE are particularly useful when a report must be rerun because of device or paper problems.

EQLREC is a c-tree® Plus ISAM level function which reads data records based on an exact match of a key value target. EQLREC cannot be used with duplicate keys. Since EQLREC is used by r-tree® to perform the JOINSTO operation, this means that the index involved in the join must not allow duplicate entries.

EXISTS [expr] is a boolean set function which evaluates to true if at least one record in the set of detail records related to expr makes the boolean expression expr true.

EXIT must be the last item in a report script.

FILE is a key word used in the SEARCH section which is followed by a file name. The file name can be either a string constant or a symbolic field which evaluates to a string.
FILENAME
FILENAME is a symbolic string constant which contains the file name to be used when DEVICE is set to 4 (stream file output), 5 (c-tree® data file output), 7 (append to stream), or 8 (append to c-tree Plus data). FILENAME defaults to "RTREE.OUT". You may use the illustration in Figure 10-4 to modify FILENAME in your interface code.

```
extern TEXT rtfname();
strcpy(rtfname,"WHAT-YOU-WANT");
```

Figure 10-4.

file hierarchy
A file hierarchy is a conceptual structure which represents the relationship among a set of files. In r-tree®, when a file is immediately below another file in the hierarchy, then the file on top is the parent file, and the file below it is a detail file. r-tree® allows files to be placed at more than one location in the "hierarchy" by use of the IS_ALIAS_FOR clause. Two or more files joined via the JOINS_TO clause are conceptually treated as one logical file in the hierarchy.

FIRST
FIRST [expr] is a computational set function which returns the first occurrence of expr over the set of detail records related to expr. If expr is numeric valued, then FIRST returns a floating point value, otherwise FIRST returns a string-valued result.

FIRST_PAGE_HDR
FIRST_PAGE_HDR is a display attribute whose value is set in the DISPLAY section. FIRST_PAGE_HDR defaults to one (YES). If non-zero, then the first page of the report will carry a page header if a page header is defined in the IMAGE section. If zero, then no page header will appear on the first page.

FORALL
FORALL [expr] is a boolean set function which evaluates to true only if all the detail records related to expr make the boolean expression expr true.

format specification
Each image line in the IMAGE section of the report script may contain literal characters which are sent to the output device as is, and format specifications which show how to output the value of an entry in the format symbol list. Each format specification begins with the FORMAT_MARK.

format symbol list
The format symbol list contains field values, constants, accumulators, control fields, and functions applied to field values which are to be substituted into the format specifications of the associated image line. The format symbol list begins immediately after the associated image line, and may span any number of physical lines in the report script.

In addition to the values for the format specifications, the format symbol list also contains display logic variables (which control optional features such as conditional
FORMAT_MARK

FORMAT_MARK is a display attribute whose value is set in the DISPLAY section. FORMAT_MARK defaults to the decimal equivalent of an ASCII at sign (‘@’). The FORMAT_MARK delimits the beginning of a format specification in an image line. To be effective, the DISPLAY section containing FORMAT_MARK must precede the IMAGE section. The value used to modify FORMAT_MARK must be a literal constant, not a symbolic constant or field value.

FORM_FEED

FORM_FEED is a display logic variable which may appear in one or more format symbol lists in the IMAGE section of the report script. After the image line associated with the format symbol list is output, r-tree® sends a FORM_FEED sequence to the output device.

The FORM_FEED sequence defaults to a single byte with decimal value 12. This is the standard form feed control character. You can modify this default to a sequence of from one to thirty-two bytes by redefining FORM_FEED in the DISPLAY section of your script.

FORMAT

FORMAT is a display logic variable illustrated in Figure 10-5 which may appear in one or more format symbol lists of the IMAGE section. FORMAT is used to print a value in one of several columns where each column has its own format specification. This is particularly useful in columnar output such as debit/credit columns. FORMAT is followed by three values (i.e., three symbols and/or constants). The second value following the FORMAT key word, call it N, specifies the number of format specifications (in the associated image line) under control of the FORMAT logic variable. The first value specifies which of these N format specifications shall be used. The first value must be in the range 1 to N. The third value is the actual output value. The other N-1 format specifications are filled with blanks.

```
+ @@@@@@@  000000  000000  "abc"  FORMAT 2 3 123  "cba"
/* produces this output: */
abc   00123      cba
```

Figure 10-5.

FREQUENCY

FREQUENCY [symbol] is a computational set function which returns the number of detail records in the set of records related to the symbol.

FRQ

FRQ is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate tallies based on a count. The count tally is always maintained as a four-byte integer.
FRS
FRS is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate tallies based on the first occurrence of the field. The accumulator returns a string-valued result for string-valued fields, and a double precision float for numeric fields.

FRSSET
FRSSET is a r-tree® ISAM level function which reads the first record matching the specified partial key. By partial key we mean a specification of the most significant bytes of a target key. FRSSET is used to begin traversing a detail file. The ISDETAILFOR clause specifies what field value is to be used as the partial key target in the call to FRSSET.

function (symbol)
function refers to one of the r-tree® built-in functions. symbol is the argument of the function. While symbol ordinarily refers to any symbolic value supported by r-tree, built-in functions can only take on data field or virtual fields as arguments. The built-in functions never modify the arguments themselves: it is as though the arguments are passed by value, not reference.

HORIZONTAL
HORIZONTAL is used with the first line of a BODY subsection to indicate that the body will be repeated horizontally (e.g., to permit multiple mailing labels to be printed across the page). The first line acts as a template, and it is not actually output. HORIZONTAL is ignored except when it occurs in the first line of the BODY.

The example in Figure 10-6 demonstrates how to use the HORIZONTAL display logic variable. The first line of the BODY must carry the HORIZONTAL display logic variable in its format symbol list. Each FORMAT_MARK ("@") in the first line of the BODY defines the successive left margins where successive BODY output will be left shifted. The number of FORMAT_MARKS in the first line determines the maximum number of bodies which can be repeated horizontally. When HORIZONTAL is used (in the first line of BODY), the entire BODY is repeated horizontally. There are several important restrictions regarding HORIZONTAL which you must be aware of.

NOTE: do not use word-wrapped format specifications or device control strings in a horizontally repeated BODY; and if TEST_LINE is required, it should typically be used on all lines of the BODY including the first line of the BODY which defines the horizontal template.

```
IMAGE
BODY
/* The following line is the horizontal repeat template */
* @ @
* @ HORIZONTAL
* @ cm_custname
* @ cm_zipcode cm_telephone
```

Figure 10-6. Horizontal Display Logic Variable.
The BODY section in the example above would cause each two-line listing (of name, Zip Code and telephone number) to be listed three across where the @’s in the template image line determine the left margins for each listing.

**IMAGE**

IMAGE is a section heading of the report script. The IMAGE section contains the layout of the report including report headers and footers, page headers and footers, control break headers and footers, as well as the main body of the report.

**image line**

Image lines contain the literal characters and format specifications which define how the report will look. An image line may be followed by a format symbol list. If the image line contains format specifications, it must be followed by a format symbol list.

**INCREMENT**

INCREMENT is a built-in function whose side effect is more important than its return value. INCREMENT returns its argument unmodified, but, more importantly, causes the status variable COUNTER to be increased by one.

**INPAPND**

INPAPND is a symbolic constant which evaluates to 8. It is one of the arguments to the DEVICE display attribute, and is used when appending output to a c-tree fixed-length data file. See INPFILE.

**INPFILE**

INPFILE is a symbolic constant which evaluates to 5. It is designed to be used when specifying a c-tree fixed-length data file as the report DEVICE in the DISPLAY section. See INPAPND.

**INT2 / INT2U / INT4 / INT4U**

These INT’s are virtual field types used in the VIRTUAL section of the report script. They represent virtual fields which contain 2-byte signed, 2-byte unsigned, 4-byte signed and 4-byte unsigned integers, respectively.

**Int2u**

int2u refers to a small non-negative constant. It is used in the report syntax to indicate a field which must be specified as a small, non-negative integer constant.

**INTEGER**

INTEGER is a sort key segment type used in the SORT section of the report script. INTEGER designates a key segment based on an unsigned integer value to be sorted in ascending order.

**INTERFACE**

INTERFACE is a built-in function which causes control to be passed to the optional user-supplied function whenever it is evaluated. Ordinarily, the return value is the same as the input argument; however, you may as part of the processing in the user-supplied function modify the input argument. The significance of the INTERFACE function is that it permits control to be passed back to your own code.

**IS_ALIAS_FOR**

IS_ALIAS_FOR is a keyword preceded by a symbolic name which becomes the alias for a file which appears at more than one location in the file hierarchy defined in the SEARCH section of the report script. The symbolic name appearing before IS_ALIAS_FOR becomes the prefix for the field names of the aliased file.
IS_DETAIL_FOR is a synonym for IS_DETAIL_FOR_SKIP.

IS_DETAIL_FOR_ERROR is a keyword which designates that a subsidiary file will be searched for one or more related detail records. If at least one related record is not found for each parent file record accessed, r-tree® will terminate with an error.

IS_DETAIL_FOR_NULL is a keyword which designates that a subsidiary file will be searched for one or more related detail records. If at least one related record is not found for a parent file record, r-tree will treat the detail file as having one related record comprised of all zero bytes.

IS_DETAIL_FOR_SKIP is a keyword which designates that a subsidiary file will be searched for one or more related detail records. If at least one related record is not found for a parent file record, r-tree will skip the parent file record.

I1, I2, I3, I4 I1 through I4 are built-in functions which increment the status variables C1, C2, C3 and C4, respectively. These functions do not alter their input arguments. See the INCREMENT built-in function.

JOINS_TO is a synonym for JOINS_TO_SKIP.

JOINS_TO_ERROR is a keyword which designates that a subsidiary file will be searched for exactly one matching record. If a matching record is not found, r-tree will terminate with an error.

JOINS_TO_NULL is a keyword which designates that a subsidiary file will be searched for exactly one matching record. If a matching record is not found, r-tree will act as if there is a “matching” record comprised entirely of zero bytes.

JOINS_TO_SKIP is a keyword which designates that a subsidiary file will be searched for exactly one matching record. If a matching record is not found, the parent file record is skipped.

JTE is a synonym for JOINS_TO_ERROR.

JTN is a synonym for JOINS_TO_NULL.

JTS is a synonym for JOINS_TO_SKIP.

LAST [expr] is a computational set function which returns the last occurrence of the computational expression expr over the set of detail records related to expr.

LAST_PAGE_FTR is a display attribute whose value is set in the DISPLAY section. LAST_PAGE_FTR defaults to a one. If non-zero, then the last page of the report will have a page footer (if a page footer is defined in the IMAGE section). If zero, no page footer will be printed on the last page of the report.
LEFT_MARGIN

LEFT_MARGIN is a display logic variable which may appear in one or more format symbol lists in the IMAGE section of the report script. The value following LEFT_MARGIN in the format symbol list becomes the current left margin for the report. Increasing the left margin only shifts the report output to the right. It does not decrease the number of columns which will be printed (although they may be truncated by the output device itself).

link file

Link file refers to a file whose purpose is to relate two or more other files. Frequently, the other associated files are master files. r-tree® does not distinguish between link files and detail files.

LFT_STR

LFT_STR is a built-in function which returns the left part of a string argument up to, but not including, the byte matching the DELIMITER symbolic string constant.

LINE_MARK

LINE_MARK is a display attribute whose value is set in the DISPLAY section. LINE_MARK defaults to the decimal equivalent of the ASCII plus sign ('+'). LINE_MARK delimits the beginning of an image line in the IMAGE section of the report script.

To change the LINE_MARK, set it to the decimal equivalent of the desired character in a DISPLAY section which precedes the IMAGE section. The value used to modify LINE_MARK must be a literal constant, not a symbolic constant or field value.

LINE_NO

LINE_NO is a status variable which contains the current line number of the report output. The line number is page relative with the first line of the page header having a line number of one. LINE_NO should never have a value greater than the page length.

LNG_STR

LNG_STR is a built-in function which returns an ASCII string representation of the 4-byte integer argument.

LST

LST is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate tallies based on the last occurrence of the field. The accumulator returns a string-valued result for string-valued fields, and a double precision float for numeric fields.

master file

A master file is a data file which contains information about an entity that can be accessed by a unique key field.

MAX

MAX is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate tallies based on the maximum value of the field. The accumulator returns a string-valued result for string-valued fields, and a double precision float for numeric fields.

MAXIMUM

MAXIMUM [expr] is a computational set function which returns the maximum value of the computational expression expr evaluated over the set of detail records related to the expr.
MIN
MIN is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate tallies based on the minimum value of the field. The accumulator returns a string-valued result for string-valued fields, and a double precision float for numeric fields.

MINIMUM
MINIMUM [expr] is a computational set function which returns the minimum value of the computational expression expr evaluated over the set of detail records related to the expr.

MONEY
MONEY is a virtual field type. It is used in the VIRTUAL section to designate virtual fields which contain four-byte integer values representing dollars and cents with an implied 2 decimal places of precision.

A value of 150 in such a field is equivalent to $1.50. The range of monetary values which can be represented is approximately -20,000,000.00 dollars to 20,000,000.00 dollars.

NXTSET
NXTSET is a c-tree® ISAM level function which reads the next data record which matches the partial key (used with the last call to FRSET). r-tree® uses NXTSET to traverse all the records in the detail file which are related to a specific record in the parent file.

NO
NO is a symbolic constant with value zero. It can be used anywhere a value is required unless the value must be a literal constant.

NO_DEVICE
NO_DEVICE is a symbolic constant with value 6. It is used in the DISPLAY section to specify no output device.

NO_LINEFEED
NO_LINEFEED is a display logic variable which may appear in one or more format symbol lists in the IMAGE section of a report script. The image line associated with the format symbol list containing the NO_LINEFEED will be concatenated to the end of the existing line of report output, instead of being placed at the beginning of a new line.

NO_MOD
NO_MOD is used in the SORT section of a script to indicate that a sort key segment should be used without any modifications, and sorted in ascending order.

NONE
NONE is used in the SORT section to explicitly indicate that no sorting is to take place. The result is the same as omitting the SORT section altogether. In either case, no sorting of the selected information will take place.

NOT
NOT is the logical negation operator. NOT expr is true only if the boolean expression expr is not true.

OR
OR is a logical connective. expr1 OR expr2 is true if either expr1 or expr2 is true, or if both boolean expressions are true.
OUTAPND is a symbolic constant which evaluates to 7. It is used in the DISPLAY section to designate a DEVICE value corresponding to appending output to a stream oriented disk file output. See OUTFILE.

OUTFILE is a symbolic constant which evaluates to 4. It is used in the DISPLAY section to designate a DEVICE value corresponding to a stream oriented disk file output. See OUTAPND.

OUTLINE is a display logic variable which may appear one or more times in one or more format symbol lists in the IMAGE section of the report script. Each instance of OUTLINE must be followed by two items in the format symbol list. The value immediately following OUTLINE is treated as a control break level number corresponding to the levels defined in the CONTROL section of the script. The next item is the field to be substituted into a format specification.

If the control break level designated by the value following OUTLINE has not changed, then the field following the level number is output as if it were blank. If the control break level has changed, then the actual field value is used for output.

PAGE_BREAK is a display logic variable which may appear in one or more format symbol lists in the IMAGE section of the report script. After the current line is output, PAGE_BREAK causes a page break to be issued. A page break causes the page footer to be output before regular output resumes.

PAGE_BREAK’s should not be included in PAGE_HDR or PAGE_FTR subsections of the IMAGE section since they will result in infinite recursions.

PAGE_FTR is a subsection heading for the IMAGE section of the report script. The image lines in the PAGE_FTR subsection will be automatically placed at the end of each page, except possibly the last page. (If LAST_PAGE_FTR is set to zero, then the last page will not contain a page footer.) The page footer can contain accumulator results since r-tree® automatically maintains tallies for all accumulators by page.

The total number of lines in the PAGE_FTR and PAGE_HDR combined must be less than the page length. PAGE_FTR should not contain any PAGE_BREAK’s or COND_PAGE’s. See PAGE_LENGTH also.

PAGE_HDR is a subsection heading for the IMAGE section of the report script. The image lines in the PAGE_HDR subsection will be automatically placed at the top of each page, except possibly the first page. (If FIRST_PAGE_HDR is set to zero, then the first page will not contain a page header.)

The total number of lines in the PAGE_HDR and PAGE_FTR combined must be less than the page length. PAGE_HDR should not contain any PAGE_BREAK’s or COND_PAGE’s. See PAGE_LENGTH also.
**PAGE_LENGTH**

PAGE_LENGTH is a display attribute that applies to all output devices but the screen. If non-zero, PAGE_LENGTH specifies the total number of physical lines on a page of the output device. r-tree® automatically adjusts PAGE_LENGTH for the PAGE_HDR / PAGE_FTR sections to arrive at an effective page length.

If you do not utilize FORM_FEED’s, then be sure that the PAGE_HDR and PAGE_FTR sections contain enough beginning and trailing blank lines, respectively, to accommodate the desired top and bottom margins.

**PAGE_NO**

PAGE_NO is a status variable which contains the current output page number. Note that PAGE_NO starts with a value of zero, and is incremented just before the first page is output.

**PAGE_WIDTH**

PAGE_WIDTH is a display attribute that applies to all output devices except the screen. PAGE_WIDTH specifies the number of printable columns on the output device. PAGE_WIDTH columns will be output after the left margin has been output.

**parent file**

A parent file is the file associated with a detail file. Each record of the parent file maps into one or more records of the detail file. r-tree uses an index to relate a parent file with each of its detail files. A parent file may have up to eight (8) detail files.

**partial key match**

A partial key match occurs when the FRSSET or NXTSET functions of c-tree® find a record whose key value matches the significant bytes of the partial key target (used in FRSSET). From the r-tree perspective, the significance of a partial key is that the field used to relate a parent file to one of its detail files should be a valid partial key for the index used to traverse the detail file.

In our customer order example, the order-item detail records can be traversed by an index made up of a order number segment and a sequence number segment. The field in the customer order parent file need only relate to the beginning of the index key. In this case, we would use the order number field of the customer order file to specify a partial key for the detail file.

**PAUSE**

PAUSE is a display logic variable which may appear in one or more format symbol lists in the IMAGE section of the report script. PAUSE causes a prompt to appear on the screen after the current report line is sent to the output device (which may be the screen).

Pressing RETURN will cause the report output to resume. PAUSE could be used to ensure that output on a screen does not scroll off the screen until it has been read. PAUSE may also be used to temporarily halt output so that forms may be loaded into an output device.

**PCNTRL_MARK**

PCNTRL_MARK is a display attribute whose value is set in the DISPLAY section. PCNTRL_MARK defaults to the decimal equivalent of the ASCII circumflex (‘^’). PCNTRL_MARK is an optional end-of-format specification delimiter. It is used to separate a format specification from a literal character that otherwise would be
considered a legitimate format character. The value used to modify PCNTRL_MARK must be a literal constant, not a symbolic constant or field value.

PERIOD
PERIOD is a symbolic constant which evaluates to the decimal equivalent of an ASCII period (‘.’). It can be used in the DISPLAY section to modify the THOUSAND_SEPARATOR for the European convention.

PRINTER0 / PRINTER1 / PRINTER2
These PRINTER symbolic constants evaluate to 0, 1 and 2, respectively. They are to be used in the DISPLAY section to specify the DEVICE for report output. The source code module rdevc.c contains the symbolic names to use when opening a printer for output.

PRV
PRV is an automatic accumulator type. It is used in the ACCUMULATOR section of the report script to designate an accumulator based on the previous occurrence of the field. Such an accumulator may be useful to check for proper sequence numbers. The accumulator returns a string-valued result for string-valued fields, and a double precision float for numeric fields.

READ_LEV0 / READ_LEV1 / READ_LEV2 / READ_LEV3 / READ_LEV4 / READ_LEV5
These six status variables indicate when records from a specified level of the file hierarchy have just been read. A value of one (1) for READ_LEVx indicates that records from level x of the file hierarchy have just been read. A value of zero (0) for READ_LEVx indicates that r-tree® is reading records at levels below level x in the file hierarchy, and that new values have not been input for the level x fields. Remember that level 0 corresponds to those files at the top of the hierarchy. Level 5 is the lowest level supported by r-tree.

relation
A set of data can be said to form a relation if it can be represented as a table whose rows may be considered un-ordered, whose columns each represent an attribute (e.g., name, height, weight), and for which each entry in the table is a scalar. The actual physical representation of the data is not important; only whether conceptually it can be represented as a relation.

In practice, most c-tree® Plus data files satisfy the properties of a relation.

relational model
The relational model is an abstract way of looking at the data in a database. It is concerned with three aspects of the data: data structure, data integrity, and data manipulation. A relational database system is one that is consistent with the important aspects of the relational model. Few, if any, actual database systems satisfy all the requirements of the relational model. c-tree Plus and r-tree do not formally ascribe to the relational model. For a good discussion of the relational model, see A Guide to DB2 by C.J. Date (Addison Wesley, 1984).

repeating group
A repeating group is a set of values, all of which apply to a given data field. Neither c-tree Plus nor r-tree support repeating groups. Instead, detail files are used to represent the multiple occurrences of the otherwise repeating data field.
REPORT_HDR and REPORT_FTR are subsections in the IMAGE section of the report script. They specify the output to appear at the beginning and end of each report, respectively. FIRST_PAGE_HDR and LAST_PAGE_FTR control whether or not page headers and footers will appear on the report header and footer, respectively.

The RESET_PAGE display logic variable permits a single run to generate multiple "reports": each time RESET_PAGE is encountered it causes a report footer to be output, the page number to be reset to one, and a report header to be output.

RETURN

RETURN is a display logic variable that may appear in one or more format symbol lists of the IMAGE section of the report script. RETURN causes its associated image line to be printed over the previous line. This can be useful when you wish to cause bold lettering or other special visual effects.

RESET_PAGE

RESET_PAGE is a display logic variable that may appear in one or more format symbol lists of the IMAGE section of the report script. RESET_PAGE causes the report footer and header to be invoked as described in the REPORT_FTR / REPORT_HDR discussion.

REVBYT

REVBYT is a built-in function which returns its argument with the bytes reversed. Note that the actual argument is not affected, but that a copy is returned with the bytes reversed. This function is intended primarily for STRING variables. However, numeric values can be used as depicted by the following example:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Virtual</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIRTUAL</td>
<td>U1</td>
<td>INTZ</td>
</tr>
<tr>
<td>U2</td>
<td>STRING</td>
<td>8U1</td>
</tr>
<tr>
<td>U3</td>
<td>STRING</td>
<td>REVBYT(U2)</td>
</tr>
<tr>
<td>U4</td>
<td>INTZ</td>
<td>8U3</td>
</tr>
</tbody>
</table>

Figure 10-7.

RGT_STR

RGT_STR is a built-in function which returns the right part of the string argument starting with the byte following the DELIMITER symbolic string constant.

ROWS

ROWS is a display logic variable used in the format symbol list to specify that the output should skip down N lines, where N is the value following ROWS.

RTCHAR / RTCHARU

RTCHAR and RTCHARU are constants defined in rtdoda.h for use in your data object definition array. They specify a field type of character and unsigned character, respectively.

RTDATE

RTDATE is a constant defined in rtdoda.h for use in your data object definition array. It specifies a field type for a 4-byte computational date. r-tree® can convert back and forth between an ASCII string of the form "mm/dd/yy" (or "dd/mm/yy" or "yy/mm/dd" depending on DATE_ORDER) and a 4-byte computational form.
RTDFLOAT

RTDFLOAT is a constant defined in *rtdoda.h* for use in your data object definition array. It specifies a field type for double precision floating point reals.

RTINT2 / RTINT2U / RTINT4 / RTINT4U / RTMONEY

These are constants defined in *rtdoda.h* for use in your data object definition array. They specify field types for signed 2-byte, unsigned 2-byte, signed 4-byte, and unsigned 4-byte integers, respectively.

RTMONEY is a constant defined in *rtdoda.h* for use in your data object definition array. It specifies a field type for 4-byte signed integers which will be treated as having an implied 2 decimal places of precision. For example, an integer value of 1025 stored in a money field will be interpreted as 10.25.

RTSFLOAT

RTSFLOAT is a constant defined in *rtdoda.h* for use in your data object definition array. It specifies a field type for single precision floating point reals.

RTSTRING

RTSTRING is a constant defined in *rtdoda.h* for use in your data object definition array. It specifies a string-valued field. Each string-valued field must have a maximum length. *r-tree*® can strip trailing blanks (see STRIP) and/or determine the string length not including trailing blanks (see STR_LEN) of any string field.

SCREEN

SCREEN is a symbolic constant which evaluates to three. It is for use in the DISPLAY section to indicate that the output DEVICE for the report will be the standard output (typically the terminal screen).

The output DEVICE defaults to the screen if none is specified in the DISPLAY section.

SCREEN_LINES

SCREEN_LINES specifies the number of lines on the output page if and only if the output DEVICE is the screen.

SCREEN_WIDTH

SCREEN_WIDTH specifies the columns for output if and only if the output DEVICE is the screen. Output beyond the SCREEN_WIDTH will be truncated by *r-tree*. The screen width does not count the columns used by the current left margin.

SEARCH

SEARCH is a section heading for the report script. The SEARCH section is used to specify the file hierarchy which will be automatically traversed by *r-tree*. Files may appear in more than one node of the hierarchy through the file alias mechanism.

The hierarchy may have up to six levels, and each node in the first five levels may have up to eight branches emanating from it.

SELECT

SELECT is a section heading for the report script. The SELECT section is used to specify the criteria for selecting records from the file hierarchy. *r-tree* restricts you to a selection down a simple path on the hierarchy, although the computational set functions (AVERAGE, FIRST, FREQUENCY, LAST, MAXIMUM, MINIMUM and SUMMATION) and the boolean set functions (EXISTS and FORALL) allow any nodes of the hierarchy to be used for selection criteria.
SETUP is a subsection of the IMAGE section of the report script. The SETUP subsection should contain the device control strings necessary to initialize the output device. Also, if you wish to use the TEST_BODY display logic variable, it must be in the format symbol list of the SETUP subsection.

Usually the format specification in the image line of the SETUP subsection only contains a single FORMAT_MARK. The format symbol list usually contains a NO_LINEFEED display logic variable, the device control variables required to initialize the device, and an empty string constant ("") at the end. The FORMAT_MARK and empty string force the format symbol list to be scanned which causes the device control strings to be sent. The NO_LINEFEED keeps a line feed from being sent to the device during initialization.

These device control variables are assigned up to thirty-two-byte control strings in the DISPLAY section of the script. Their purpose is to permit arbitrary byte strings to be interspersed with the report output for the purpose of initializing and/or modifying the output device characteristics such as physical page length or font.

There is no special significance to the names of these device control variables: SETFONT0 can be used for any type of control string, not just for controlling the type font.

SFLOAT

SFLOAT is a virtual field type. It is used in the VIRTUAL section of the report script to designate a virtual field which contains a single precision floating point result.

SKIP_LINE

SKIP_LINE is a display logic variable which may appear in one or more format symbol lists of the IMAGE section of the report script. If present, SKIP_LINE must be the first entry in the format symbol list. The symbolic field following SKIP_LINE is evaluated just before the associated image line is to be processed. If the field evaluates to zero, the line is output; otherwise it is not output. See TEST_LINE which is the opposite of SKIP_LINE.

SLASH

SLASH is a symbolic constant which evaluates to the decimal equivalent of an ASCII forward slash ('/'). It may be used anywhere a value is required unless the value must be a literal constant.

SORT

SORT is a section heading for a report script. The SORT section defines how the records which satisfy the SELECT section will be sorted before output. The sort criteria may be based on any number of fields (although the underlying c-tree® Plus file handler must be setup for a sufficiently long maximum key length) which occupy one or more levels of the file hierarchy.

If the SORT section is omitted or the NONE keyword is used in the SORT section, no sorting will take place. Frequently, the SORT section and the CONTROL section are
setup in a similar fashion since multiple control breaks will produce meaningless results if the data do not adhere to the implied nesting of the control break levels.

**SPACES**

SPACES is a display logic variable which may appear one or more times in one or more format symbol lists in the IMAGE section of the report script. The value following SPACES determines how many spaces will precede the next format specification.

**START**

The START keyword must be the first non-blank entry in a report script.

**STARTING_PAGE**

STARTING_PAGE is a display attribute whose value is set in the DISPLAY section. STARTING_PAGE defaults to zero. If non-zero, r-tree® will not send output to the output device until the STARTING_PAGE number is reached.

STARTING_PAGE and ENDING_PAGE are particularly useful when a report must be rerun because of device or paper problems.

**STRING**

STRING is a virtual field type. It is used in the VIRTUAL section to designate a virtual field which contains a string value. r-tree allows arbitrary bytes in strings including null bytes, but null bytes may cause unexpected results during output.

**STRIP**

STRIP is a built-in function which returns a string with the trailing blanks removed from the string-valued argument. In fact, STRIP actually strips any byte values whose decimal equivalent is less than or equal to the decimal equivalent of an ASCII blank.

STRIP is frequently used during string concatenation operations to eliminate undesired blanks between two string fields.

**STR_DAT**

STR_DAT is a built-in function which returns a four-byte computational date based on a string argument of the form "mm/dd/yy" (or "dd/mm/yy" or "yy/mm/dd" depending on DATE_ORDER). Note that the string must be exactly eight bytes in length and that the slashes must be replaced if DATE_SEPARATOR is not at its default. STR_DAT returns zero if the argument is not in correct form or is not a valid date.

**STR_FLT**

STR_FLT is a built-in function which returns a double precision floating point value for a string-valued argument.

**STR_LEN**

STR_LEN is a built-in function which returns the length of its string-valued argument. Trailing bytes with a value less than or equal to the decimal equivalent of an ASCII blank are not included in the length.

**STR_LNG**

STR_LNG is a built-in function which returns the integer value represented by the string-valued argument.

**SUBSTRING**

SUBSTRING is a built-in function which marks a string argument as a pattern which may be matched anywhere in a target string. The target string appears on the other side from the "=" or "<>" comparison operators. SUBSTRING is a special case of WILDCARD.
**SUM**

SUM is an automatic accumulator type. SUM is used in the ACCUMULATOR section to designate a tally based on the sum of a specified field. If the field is string-valued, the sum is based on the value of the field after it is converted to a number.

The SUM tally is maintained as a double precision floating point real.

**SUMMATION**

SUMMATION [expr] is a computational set function which computes the sum of the computational expression expr over the set of detail records related to expr. Unlike SUM above, SUMMATION will not operate on expr if it is string-valued.

**SYS_DATE**

SYS_DATE is a status variable which contains the current system date in 4-byte computational form. SYS_DATE requires that the rtdate.c module have a valid routine to access the system date. The rtdate.c shipped on the distribution disk assumes the ANSI routines to access local time and date.

**TEST_BODY**

NO LONGER IMPLEMENTED

**TEST_LINE**

TEST_LINE is a display logic variable which may appear in one or more format symbol lists of the IMAGE section of the report script. If present, TEST_LINE must be the first entry in the format symbol list. The symbolic field following TEST_LINE is evaluated just before the associated image line is to be processed. If the field evaluates to zero, the line is not output; otherwise it is output. See SKIP_LINE which is the opposite of TEST_LINE.

```
VIRTUAL test

INTZ 2 apples > oranges ? YES : NO

IMAGE

BODY

+ apples = 299990
TEST_LINE test apples
+ oranges = 999990
SKIP_LINE test oranges
```

Figure 10-8. Test_Line Example.

In Figure 10-8, the virtual field test will be non-zero if apples is greater than oranges or zero otherwise. Therefore, the apples line will print if apples is greater, otherwise the oranges line will print, but not both of them.

**TEST_PAGES**

TEST_PAGES is a display attribute whose value is set in the DISPLAY section. TEST_PAGES defaults to zero. If non-zero, then all the image lines in the IMAGE section are sent to the output device TEST_PAGES times. The format specifications are simply output without field value substitutions. This permits you to test if the output device is properly aligned with the forms before actually producing the report output. After the test pages have been output, you have an opportunity to abort the report or continue.
THOUSAND_SEPARATOR is a display attribute whose value is set in the DISPLAY section of the report script. THOUSAND_SEPARATOR defaults to the decimal equivalent of an ASCII comma (','). If you modify THOUSAND_SEPARATOR, then numeric output which is specified to use automatic thousand separators will use the modified character. Along with DECIMAL_POINT, this permits you to easily adapt to the European expression of numbers.

UCASE UCASE is a built-in function which returns a string-valued result made up of the string argument shifted to all upper-case letters.

UPPER UPPER is used in the SORT section of the report script to indicate that a sort key segment must be translated into upper-case, and sorted in ascending order.

user supplied function You may pass an optional function pointer to your own function in the call to the r-tree® report function. This "user supplied function" provides a window into r-tree for your own code.

USING_KEY USING_KEY is a keyword used in the SEARCH section and SORT section of the report script. In the SEARCH section, USING_KEY is followed by a symbolic key (index) name specified in the c-tree® parameter file. In the SORT section, USING_KEY is followed by the sort segment specifications.

When used with a detail file in the SEARCH section, USING_KEY implies that the detail file will be searched in ascending key order. See USING_REVKEY below.

USING_REVKEY USING_REVKEY is a keyword used with detail files in the SEARCH section of the report script. USING_REVKEY causes the detail file to be searched in descending key value order. See USING_KEY above.

VERT_TAB VERT_TAB is a display logic variable used in the format symbol list to specify that the output should skip down to the Nth line, where N is the value following VERT_TAB.

VIRTUAL VIRTUAL is a section heading for a report script. There can be any number of VIRTUAL sections. A VIRTUAL section contains virtual field definitions.

WILDCARD WILDCARD is a built-in function which marks a string argument as a pattern. The pattern may include the special characters "?" (which matches any single character), "*" (which matches zero or more occurrences of any characters), and "\" which acts as an escape character (i.e., a character following a "\" is treated as a literal, not as a special character). The target string appears on the other side of the "=" or "<>" comparison operators.

YES YES is a symbolic constant which evaluates to one. YES may be used anywhere a value is needed unless a literal constant is required.
**ZERO**

ZERO is a built-in function whose side effect is more important than its return value. ZERO returns its argument unmodified, but, more importantly, sets the status variable COUNTER to zero.

ZERO, DECREMENT, INCREMENT and COUNTER work together to permit r-tree® to generate a sequence of numbers.

**ZERO_FILL**

ZERO_FILL is a display attribute whose value is set in the DISPLAY section. ZERO_FILL defaults to the decimal equivalent of the ASCII asterisk (*). If ZERO_FILL is modified, then numeric output which has specified automatic zero fill will use the modified character.

**Z1, Z2, Z3, Z4**

Z1 through Z4 are built-in functions which zero the status variables C1, C2, C3 and C4, respectively. These functions do not alter their input arguments. See the ZERO built-in function.
This page intentionally left blank.
11. Debugging Hints

11.1 Missing Data (Records not selected)

One common tech support question concerning r-tree involves fields that don’t appear to be printing. Several causes might apply, including missing records, improper SELECT and SORT statements, or index target problems.

**SEARCH/SELECT**

A common dilemma occurs when r-tree does not seem to be considering data in an “IS_DETAIL_FOR” file. r-tree may need to be specifically instructed to address this detail level of the file search hierarchy. This can be done by simply addressing any field from the detail record in the SELECT statement. A simple statement comparing a field to itself may solve this problem:

```sql
SELECT field1 = field1
```

Another common reason for missing data may be that no records can be found in a subordinate file. As r-tree proceeds down the file hierarchy with “JOINS_TO” or “IS_DETAIL_FOR” relationships, a “no hit” situation in a subordinate file will cause processing to be skipped for the parent record. A typical example might be:

**SEARCH**

```plaintext
FILE customer ALL
FILE order IS_DETAIL_FOR customer ....
FILE parts JOINS_TO order BY_FIELD order_part_no......
```

As each customer record is read, all related order records are also read, one at a time. If any particular order had a part number that could not be resolved from the part file, that particular order would be skipped. The solution is to use JOINS_TO_NULL (or IS_DETAIL_FOR_NULL) as follows:

**SEARCH**

```plaintext
FILE customer ALL
FILE order IS_DETAIL_FOR customer ....
FILE parts JOINS_TO_NULL order ....
```

In this example, all order records will be processed, regardless if the part number can be resolved.
The JOINS_TO file relationship is only possible with unique keys. Indices that support duplicates cannot be used in a JOINS_TO relationship.

Record Retrieval

If the previous suggestions have not helped to resolve any type of missing data problem, the quickest method for resolving the problem is to use a debugger to interrogate the r-tree® record retrieval function (filsys()). The filsys() function is found in rthread.c and is responsible for retrieving records from the c-tree® data file. The function is merely a switch-case statement that calls c-tree’s API (EQLREC(), FRSET(), NXTSET() . . .). By following the flow of r-tree on a working script and then comparing to the script not working it will be possible to determine where the problem is.

Target problems

If an index problem is suspected, using the CTDEBUG define found in the c-tree source module ctcomp.c will help. CTDEBUG prints a byte-by-byte comparison of the target compared to every key value found in the proper index node. For more information on CTDEBUG see the c-tree Plus documentation.

11.2 Header Changes

- c-tree® - If any c-tree header file is changed, all of c-tree and r-tree must be recompiled. Be especially careful to ensure all report script processing modules get recompiled (i.e., rtintr, rtcmpl, rtmmry) in addition to all of the application object modules.

- r-tree - If any r-tree header file is changed, all of r-tree must be recompiled. Be especially careful to ensure all report script processing modules get recompiled (i.e., rtintr, rtcmpl, rtmmry) in addition to all of the application object modules.
A. Reserved Words and Operators

The r-tree reserved words are setup in the source-code header file RTIVAL.H. Some of the reserved words act as keywords to identify the type of r-tree statement. The rest of the reserved words represent special r-tree variables and parameters. The following lists contain most, but not all, of the reserved words. Some synonyms and special alternatives have been omitted. Chapter 10 presents all of the reserved words in alphabetical order.

Report Script Section Headings

<table>
<thead>
<tr>
<th>Reserved Word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUMULATOR</td>
<td>accumulator field definitions</td>
</tr>
<tr>
<td>CONTROL</td>
<td>control break field definition</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>output control definitions</td>
</tr>
<tr>
<td>EXIT</td>
<td>terminate script</td>
</tr>
<tr>
<td>IMAGE</td>
<td>layout subsections to follow</td>
</tr>
<tr>
<td>SETUP</td>
<td>initialization sequence for display device</td>
</tr>
<tr>
<td>PAGE_HDR</td>
<td>beginning of page layout</td>
</tr>
<tr>
<td>REPORT_HDR</td>
<td>beginning of report layout</td>
</tr>
<tr>
<td>CONTROL_HDR</td>
<td>control break header layouts</td>
</tr>
<tr>
<td>BODY</td>
<td>layout for selected information</td>
</tr>
<tr>
<td>CONTROL_FTR</td>
<td>control break footer layouts</td>
</tr>
<tr>
<td>REPORT_FTR</td>
<td>end of report footer layout</td>
</tr>
<tr>
<td>PAGE_FTR</td>
<td>page footer layout</td>
</tr>
<tr>
<td>SEARCH</td>
<td>file interrelationships</td>
</tr>
<tr>
<td>SELECT</td>
<td>record selection criteria</td>
</tr>
<tr>
<td>SORT</td>
<td>sorting requirements</td>
</tr>
<tr>
<td>START</td>
<td>begin report script</td>
</tr>
<tr>
<td>VIRTUAL</td>
<td>virtual field definitions</td>
</tr>
</tbody>
</table>

Operators

Precedence levels are shown in parentheses. A lower number means a higher precedence. Operands are grouped left to right for equal precedence operators.
### Computational

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>address of</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>#</td>
<td>string concatenation</td>
</tr>
</tbody>
</table>

### Relational

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
</tbody>
</table>

### Boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>logical negation</td>
</tr>
<tr>
<td>AND</td>
<td>true if and only if both expressions are true</td>
</tr>
<tr>
<td>OR</td>
<td>true if one or both expressions are true</td>
</tr>
</tbody>
</table>

### Field Types

#### Virtual Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>signed one-byte integer</td>
</tr>
<tr>
<td>CHARU</td>
<td>unsigned one-byte integer</td>
</tr>
<tr>
<td>INT2</td>
<td>signed two-byte integer</td>
</tr>
<tr>
<td>INT2U</td>
<td>unsigned two-byte integer</td>
</tr>
<tr>
<td>INT4</td>
<td>signed four-byte integer</td>
</tr>
<tr>
<td>INT4U</td>
<td>unsigned four-byte integer</td>
</tr>
<tr>
<td>Reserved Words and Operators</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Status Variables and Symbolic Constants</strong></td>
<td></td>
</tr>
<tr>
<td>SFLOAT</td>
<td>single-precision floating point</td>
</tr>
<tr>
<td>DFLOAT</td>
<td>double-precision floating point</td>
</tr>
<tr>
<td>DATE</td>
<td>four-byte computational form</td>
</tr>
<tr>
<td>MONEY</td>
<td>signed four-byte integer with implied decimal point before the last two digits</td>
</tr>
<tr>
<td>STRING</td>
<td>string (character array)</td>
</tr>
</tbody>
</table>

**Accumulators**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FRQ</td>
<td>frequency count</td>
</tr>
<tr>
<td>MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>SUM</td>
<td>summation</td>
</tr>
<tr>
<td>AVG</td>
<td>average</td>
</tr>
<tr>
<td>FRS</td>
<td>first occurrence</td>
</tr>
<tr>
<td>LST</td>
<td>last occurrence</td>
</tr>
<tr>
<td>PRV</td>
<td>previous occurrence</td>
</tr>
</tbody>
</table>

**Sort Key Segments**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_MOD</td>
<td>no translation</td>
</tr>
<tr>
<td>DSC_NO_MOD</td>
<td>no translation, descending order</td>
</tr>
<tr>
<td>UPPER</td>
<td>upper-case translation</td>
</tr>
<tr>
<td>DSC_UPPER</td>
<td>upper-case translation, descending order</td>
</tr>
<tr>
<td>ALT_SEG</td>
<td>alternative collating sequence</td>
</tr>
<tr>
<td>ALT_DSC</td>
<td>alternative sequence, descending order</td>
</tr>
<tr>
<td>INTEGER</td>
<td>unsigned integer segment and/or date</td>
</tr>
<tr>
<td>DSC_INTEGER</td>
<td>unsigned integer segment and/or date, descending order</td>
</tr>
</tbody>
</table>

**Status Variables and Symbolic Constants**

The following status variables and constants can be used anywhere a value is required, unless the value must be in the form of a literal constant.

**Status Variables**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_NO</td>
<td>page number</td>
</tr>
</tbody>
</table>
Reserved Words and Operators

Status Variables and Symbolic Constants

<table>
<thead>
<tr>
<th>Symbolic Constants Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
</tr>
<tr>
<td>PRINTER0</td>
</tr>
<tr>
<td>PRINTER1</td>
</tr>
<tr>
<td>PRINTER2</td>
</tr>
<tr>
<td>SCREEN</td>
</tr>
<tr>
<td>OUTFILE</td>
</tr>
<tr>
<td>INPFILE</td>
</tr>
<tr>
<td>NO_DEVICE</td>
</tr>
<tr>
<td>OUTAPND</td>
</tr>
<tr>
<td>INFAPND</td>
</tr>
<tr>
<td>DOLLAR</td>
</tr>
<tr>
<td>PERIOD</td>
</tr>
<tr>
<td>COMMA</td>
</tr>
<tr>
<td>SLASH</td>
</tr>
<tr>
<td>DASH</td>
</tr>
<tr>
<td>DELIMITER</td>
</tr>
</tbody>
</table>
Display Attribute Variables

The display attributes help define the overall characteristics of the report output. Their values can be changed in the DISPLAY section of the report script.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE_WIDTH</td>
<td>132</td>
</tr>
<tr>
<td>PAGE_LENGTH</td>
<td>66</td>
</tr>
<tr>
<td>SCREEN_WIDTH</td>
<td>80</td>
</tr>
<tr>
<td>SCREEN_LINES</td>
<td>24</td>
</tr>
<tr>
<td>DATE_ORDER</td>
<td>0 (mm/dd/yy)</td>
</tr>
<tr>
<td>CURRENCY_MARK</td>
<td>DOLLAR°</td>
</tr>
<tr>
<td>THOUSAND_SEPARATOR</td>
<td>COMMA°</td>
</tr>
<tr>
<td>DECIMAL_POINT</td>
<td>PERIOD°</td>
</tr>
<tr>
<td>DATE_SEPARATOR</td>
<td>SLASH°</td>
</tr>
<tr>
<td>LINE_MARK</td>
<td>43 ('+')</td>
</tr>
<tr>
<td>FORMAT_MARK</td>
<td>64 ('@')</td>
</tr>
<tr>
<td>PCNTRL_MARK</td>
<td>94 ('^')</td>
</tr>
<tr>
<td>FIRST_PAGE_HDR</td>
<td>YES°</td>
</tr>
<tr>
<td>LAST_PAGE_FTR</td>
<td>YES°</td>
</tr>
<tr>
<td>ZERO_FILL</td>
<td>42 ('*')</td>
</tr>
<tr>
<td>DEVICE</td>
<td>SCREEN°</td>
</tr>
<tr>
<td>BODY_COUNT</td>
<td>0</td>
</tr>
<tr>
<td>TEST_PAGES</td>
<td>0</td>
</tr>
<tr>
<td>STARTING_PAGE</td>
<td>0</td>
</tr>
<tr>
<td>ENDING_PAGE</td>
<td>0</td>
</tr>
</tbody>
</table>

*These are references to the symbolic constants defined above.

Display Logic Variables

These display logic variables are placed after a layout line along with the field values to be substituted into the format specifications. "NextValue" refers to the value of the symbol following the display logic variable.
COLUMN set tab for next format specification to NextValue
COND_HDR conditional print of control header specified by NextValue
COND_PAGE if NextValue lines do not fit on current page, issue a page break
FORM_FEED output a form feed after line is output
FORMAT select ith format specification for output and make other N - 1 format specifications blank where NextValue is i, and the value following NextValue is N
HORIZONTAL define horizontal repeat template for BODY
LEFT_MARGIN set left margin to NextValue
NO_LINEFEED do not start output on new line
OUTLINE print blank field if control field (level determined by NextValue) value has not changed
PAGE_BREAK issue a page break after line is output
PAUSE issue a pause after line is output
RESET_PAGE output optional REPORT_FTR, reset page number to 1, and begin with optional REPORT_HDR
RETURN output on top of previous line
ROWS skip down NextValue lines
SKIP_LINE skip line if followed by non-zero NextValue
SPACES add NextValue spaces before next format specification
TEST_LINE skip line if followed by zero NextValue
VERT_TAB skip down to the Nth line where N equals NextValue

Device Strings

The following device strings allow you to send arbitrary byte strings to an output device. You may place up to thirty-two (32) bytes in each device variable. These variables are interspersed with other entries in the format symbol list. In order for a device string to be sent to the output device, there must be additional values in the format symbol list to cause r-tree®’s algorithm to scan the device strings. (There is no real significance to symbolic names of the device strings.)

r-tree® Report Generator Appendix A 176
Built-In Function Names

These functions are defined in the module rtuser.c. With some care you can add your own functions (see Section 3.7.3).

**ADDTO**
add result of virtual field computation to current contents of virtual field (instead of replacing contents). Return argument unaltered.

**DAT_DAY**
convert date to day of week name

**DAT_DD**
convert date to month-day number (01-31) in ASCII string

**DAT_MM**
convert date to month number (01-12) in ASCII string

**DAT_MTH**
convert date to month name

**DAT_YYYY**
convert date to year (4 digit) ASCII string

**DBL_STR**
convert a double to number in ASCII string

**DECREMENT**
decrement COUNTER. Return argument unaltered.

**D1,D2,D3,D4**
same as DECREMENT, but for Cx (x=1,2,3,4)

**INCREMENT**
increment COUNTER. Return argument unaltered.

**I1,I2,I3,I4**
same as INCREMENT, but for Cx (x=1,2,3,4)

**INTERFACE**
pass control to the optional user supplied function with mode specified as LOOP_INP. Any reasonable operation may be performed at this time. See Section 5.1.4

**LFT_STR**
left part of string up to DELIMITER

**LNG_STR**
convert long to number in ASCII string

**REVBYT**
reverse bytes in field

**RG1_STR**
right part of string after DELIMITER

**STR_DAT**
convert a string of the form "mm/dd/yy" to four-byte computational form

**STR_FLT**
convert string to double float

**STR_LEN**
return string length (trailing blanks are not included in length)

**STR_LNG**
convert string to long integer

**STRIP**
strip trailing blanks from field

**SUBSTRING**
mark argument as substring pattern for string compare.

**UCASE**
convert string to upper case
Reserved Words and Operators

Set Functions

**WILDCARD** mark argument as a general pattern for use in a string compare. The pattern may include a '?' which matches any single character, and a '*' which matches zero or more instances of any character.

**ZERO** reset COUNTER to zero. Return argument unaltered.

**Z1, Z2, Z3, Z4** same as ZERO, but for Cx (x=1,2,3,4)

### Set Functions

These functions cause detail records to be automatically traversed in order to perform the indicated function. The computational set functions accept computational expressions as arguments. The boolean set functions accept boolean expressions as arguments. Ordinarily, these powerful functions are only used as part of the record selection criteria. The ACCUMULATOR section is used to compute these types of values for information being output as part of the report.

<table>
<thead>
<tr>
<th>Computational Set Functions</th>
<th>Boolean Set Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQUENCY</td>
<td>AVERAGE</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>FIRST</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>LAST</td>
</tr>
<tr>
<td>SUMMATION</td>
<td>EXISTS</td>
</tr>
<tr>
<td></td>
<td>FORALL</td>
</tr>
</tbody>
</table>

### Syntactic Elements

- **BY_FIELD**: data object or virtual field follows
- **IS_ALIAS_FOR**: allows file to appear more than once in SEARCH section
- **IS_DETAIL_FOR**: associated data file (one-to-many) name follows
- **FILE**: data file name follows
- **JOINs_TO**: associated data file (one-to-one or many-to-one) name follows
- **USING_KEY**: index name or definition follows
- **ALL**: in place of boolean SELECT expression
- **NONE**: no sort key
- **?**: part of boolean virtual field
- **:** part of boolean or coded virtual field
- **&**: part of overlay virtual field
**B. r-tree Report Script Syntax**

```plaintext
report ::= START section_list EXIT
section_list ::= section_list section | section
section ::= SELECT select | SORT sort | SEARCH search | CONTROL control | ACCUMULATOR accumulator | VIRTUAL virtual | IMAGE image | DISPLAY display
select ::= boolean_expr | ALL
sort ::= USING_KEY sort_list | NONE
sort_list ::= sort_list keyseg | keyseg
keyseg ::= otype data_obj
otype ::= NO_MOD | INTEGER | UPPER | DSC_NO_MOD | DSC_INTEGER | DSC_UPPER
search ::= FILE value key_spec search2
key_spec ::= ALL | USING_KEY data_obj | USING_KEY data_obj range_beg
range_beg ::= null | upper_val
range_mid ::= null | range_mid lower_val upper_val
range_end ::= null | lower_val
lower_val ::= [ value | ] value
upper_val ::= value [ ]
search2 ::= null | search2 file2
file2 ::= alias FILE value file_relation value
        BY data_obj USING_KEY data_obj
alias ::= null |
```

Programmer's Reference Guide Appendix B 179
SYMBOL IS_ALIAS_FOR

file_relation ::= JOINS_TO | IS_DETAIL_FOR

control ::= control control_def |
control_def ::= INT2U data_obj

accumulator ::= accumulator accumulator_def |
accumulator_def ::= SYMBOL atype data_obj

atype ::= AVG | FRS | FRQ | LST |
MAX | MIN | SUM

virtual ::= virtual virtual_def |
virtual_def ::= SYMBOL ftype INT2U vir_exp

ftype ::= CHAR | CHARU | INT2 | INT2U |
INT4 | INT4U | SFLOAT | DFLOAT |
MONEY | DATE | STRING

vir_exp ::= overlay_expr |
computation_expr | boolean_expr ? computation_expr :
computation_expr |
: computation_expr code_list

overlay_expr ::= & data_obj + computation_expr |
& data_obj - computation_expr |
& data_obj

code_list ::= code_list code_pair |
code_pair

code_pair ::= INT2U cvalue

boolean_expr ::= boolean_expr bool_op bool_trm |
boolean_trm

bool_trm ::= NOT logic_value |
logic_value

bool_op ::= OR | AND

logic_value ::= ( boolean_expr ) |
comparison |
boolean_set [ expr ]

boolean_set ::= EXISTS | FORALL

comparison ::= computation_expr rel_op |
computation_expr

rel_op ::= < | <= | = | >= | > | <>
computation_expr ::= computation_expr comp_op cmp_trm |
                 cmp_trm

cmp_trm ::= - cmp_value |
          cmp_value

comp_op ::= + | - | # | * | /

cmp_value ::= ( computation_expr ) |
             value |
             computational_set [ computation_expr ]

computational_set ::= AVERAGE | FIRST | LAST | MAXIMUM | MINIMUM | SUMMATION

image ::= image image_def |
        image_def

image_def ::= itype lines |
             itype_sub ( INT2U ) lines

itype ::= SETUP | REPORT_HDR | PAGE_HDR |
        BODY | PAGE_FTR | REPORT_FTR

itype_sub ::= CONTROL_HDR | CONTROL_FTR

lines ::= lines line |
        line

line ::= IMAGE_LINE symbol_list

symbol_list ::= null |
              symbol_list value |
              symbol_list DEVICE_CONTROL |
              symbol_list DISPLAY_LOGIC

display ::= display display_comp |
           display_comp

display_comp ::= DISPLAY_ATTRIBUTE value |
               DEVICE_CONTROL intlst

intlst ::= intlst value |
         value

value ::= data_obj |
        SYMBOL ( data_obj ) |

data_obj ::= SYMBOL |
            STATUS_VARIABLE |
            SYMBOLIC_CONSTANT |
            C [ INT2U ] |
            A [ INT2U INT2U ] |
            SYMBOL . SYMBOL

cvalue ::= INT2U |
          NUMERIC |
          STRING |
          special
r-tree Report Script Syntax

special ::= DATE_CONSTANT | MONEY_CONSTANT

null ::=
C. r-tree Errors

Error Messages

This section lists the error messages produced by r-tree® in alphabetical order. Most of these messages are produced before r-tree attempts the report, and hence the report script can be modified to correct the problem.

In the following messages, the "###" characters represent numbers that will be output as part of the error message.

*** A KEY (index) IS REQUIRED HERE ***
symbolic_name

The symbolic name of a key (index) is required. The symbolic names are found in the c-tree® parameter file.

*** ACCUMULATOR CANNOT BE FUNCTION ARGUMENT ***

An accumulator field cannot be the argument of a built-in function.

*** ALIAS alias_name MUST BE FOR A FILE ALREADY USED IN THE HIERARCHY ***

The alias must apply to a file which has already appeared in the SEARCH section.

*** BAD FORMAT SPECIFICATION (at offset ###) ***

At the byte offset indicated from the beginning of the image line, an illegal character was detected or the format specification became too long.

*** BAD PARAMETER FILE FIELD SPECIFICATION FOR filename ***

field_name field_name

The field names used in the parameter file for the specified file are not consistent with the data object definition array.

*** Compiler Memory Model Inconsistency ***

The size of data addresses has been modified from the compiler to the run-time program. Recompile the report script with a consistent memory model.

*** Control Field Length Discrepancy With Data Object Definition ***

The length of a field used for a control break has changed in the data object definition array. Recompile the report script with the current DODA.
*** COULD NOT ALLOCATE ### SEARCH KEY BUFFERS ***
Not enough dynamic memory to allocate key buffers to use for the range limits in the
SEARCH section.

*** Could not allocate report
buffer of size ### ***
Not enough dynamic memory to hold report script.

*** COULD NOT COMPUTE FILE NAME (###) ***
A symbolic file name reference in the SEARCH section could not be evaluated. Be
sure that a valid file name has been referenced.

*** COULD NOT CREATE OUTPUT FILE. File error code ###. <###> ***
The output DEVICE is specified as a c-tree® data file (DEVICE = five or INPFILE);
but the data file could not be created. The first number is a c-tree error code. The second
number is the record length.

*** COULD NOT CREATE REPORT SCRIPT FILE <file_name>.
File Error ### ***
r-tree could not create the compiler output file. The error code is a c-tree error number.

*** COULD NOT CREATE SORT DATA. File error code ###. <###> ***
During sort, r-tree could not open a temporary data file. The error code is a c-tree error
number. The number in brackets is the record length of the temporary file.

*** COULD NOT CREATE TEMPORARY SORT INDEX.
File Error ###. <###> ***
The temporary index used as part of the sort process could not be created. The error
code is a c-tree error number. The second number is the key length required by the
temporary index. If the error code is 45, then the MAXLEN parameter in CTOPTN.H
must be increased, and the c-tree library rebuilt.

*** Could Not Open Compiled Report Script <script_name>. File Error ### ***
The r-tree run-time executor could not open a compiled report script. The error number
is a c-tree error code.

*** Could not open report file <script_name> ***
A file named script_name is not found in the specified directory (or the current directory
if script_name does not include a directory path).
*** COULD NOT OPEN <device_name> FOR REPORT OUTPUT ***

Could not open the output device. Make sure device_name is legal on your system. See RTDEV.C in order to change the printer device names.

*** COULD NOT WRITE TO COMPILER OUTPUT FILE ***

r-tree could not write the compiler output file. Is there room on the disk for the compiler output?

*** Could not write to output file. File Error Code ###. ***

Could not write to a c-tree® data file acting as the output DEVICE. The error code reported is a c-tree error number.

*** Data Object Inconsistency ***

The data object definition array has been changed since report script compilation. Recompile the report script with the current DODA.

*** FILE filename DOES NOT LINK TO ANY OF THE ABOVE FILES ***

Each subsidiary file in the SEARCH section must relate to a file already specified in the file hierarchy.

*** FILE filename IS NOT A DATA FILE ***

The file name does not reference a data file. It must be an index file.

*** FREQUENCY REQUIRES EXACTLY ONE ARGUMENT ***

The FREQUENCY computational set function must have a field name for its argument. Unlike the other set functions, FREQUENCY cannot have an expression for its argument.

*** Function func_name Called With Improper Argument Type: Error ### ***

func_name was called with the wrong type of argument. The error code number indicates the particular problem.

*** Header/Footer Control Level Out Of Range ***

The subscript of a control header or footer subsection heading is less than one or greater than the highest control break level.

*** Illegal Alias Field Reference <alias.field> ***
The aliased field reference is not allowed since either alias is not an actual alias name assigned in the SEARCH, or the field is not a data field for the aliased file.

*** ILLEGAL CHARACTER ***
Lexical scanner detects a character in the report script which is not allowed (at least in the context in which it was found). For example, a '@' sign could cause this error unless it occurs in a valid image line.

*** ILLEGAL CLASS ***
r-tree accepts control field and accumulators in a short-hand subscript form: C[i] for control fields, and A[i] or A[i j] for accumulators. Improper use of the "subscript" notation can lead to this error.

*** ILLEGAL FIELD TYPE ### FOR DODA ENTRY <symbol> ***
Your Data Object Definition Array contains an illegal field type number. Ordinarily, these field types are specified by the constants RTCHAR, ..., RTSTRING defined in RTDODA.H.

*** ILLEGAL NUMERIC CONSTANT ***
A numeric constant of unknown type is detected during the first parsing pass. This is an unexpected error.

*** ILLEGAL OUTPUT DEVICE: ### ***
The DEVICE display attribute must be followed by a value in the range from zero to six. The value may be specified by a literal constant, a symbolic constant (e.g., PRINTER1), or a symbolic field name. The number shown in the error message is the value from the report script.

*** ILLEGAL SYNTAX ***
The report script is not formulated according to the proper syntax. Either the item pointed to by the '^' is incorrect, or a preceding entry, while syntactically correct, is not what you intended. There is only one case which is formally correct according to the syntax (see Appendix A), but which causes a syntax error. This is discussed in Section 3.5.1.

*** ILLEGAL VALUE REFERENCE ***
Symbolic index name is the same as a data field name in the DODA. Change the symbolic index name in the parameter file.

*** IMPROPER VALUE FOLLOWS TEST_LINE ***
*** IMPROPER VALUE FOLLOWS LEFT_MARGIN ***
Each of the referenced display logic variables must be followed in the format symbol list by a constant or a field which evaluates to a numeric value.

*** ITEM UNDEFINED ***

Unexpected error.

*** KEY key_name (###) DOES NOT INDEX DATA FILE file_name (###) ***

The key with symbolic key_name is not an index for the data file specified. The numbers in parentheses correspond to the parameter file numbers.

*** KEY SEGMENT MUST BE A VIRTUAL VARIABLE OR A FIELD IN YOUR DODA ***

The key segments defined in the SORT section must be based on a data field, an aliased data field or a virtual field.

*** Key Word Inconsistency ***

The number of keywords in rival.h has been modified. Recompile the report script.

*** MAX CONTROL BREAK LEVELS EXCEEDED ***

accumulator name

Unexpected error.

*** MAX_DSTRNG (in rtdoda.h) Changed From ### To ### ***

Recompile report script with new MAX_DSTRNG setting in affect.

*** NAME: name IS NOT A FILE NAME ***

The name used in the SEARCH section is not a file name.

*** NOT ENOUGH VALUES FOR FORMATS ***

There are not enough entries in the format symbol list to substitute into the format specifications.

*** ONLY ACCUMULATOR MAY HAVE A DOUBLE SUBSCRIPT ***

accumulator name
The accumulator notation A[i j] cannot be used with any letter (e.g., C[i j] is not allowed).

*** OPERANDS ARE INCOMPATIBLE: Operator = op ***

The left and right operands for the specified operator are incompatible. r-tree will not automatically convert between string and numeric fields (except in the case of SUM and AVG accumulator fields).

*** OUT OF DYNAMIC MEMORY ***

Not enough dynamic memory available.

*** POSITIVE INTEGER VALUE REQUIRED ***

The values assigned in the DISPLAY section must be greater than or equal to zero.

*** Right-Hand-Side of symbol does not match symbol’s Type ***

The computational definition of the specified virtual symbol does not yield a value consistent with the type specification of the virtual. For example, a string virtual has a numeric or date valued right-hand-side; or a numeric virtual has a string valued right-hand-side.

*** RUNTIME STACK OVERFLOW <###,###> <###,###,###> ***

Not enough run-time space. An unexpected error most likely caused by memory being overwritten. If you report this error to FairCom, please have the five numbers output with the message.

*** Script Compiled Under Different r-tree/c-tree Version ***

Recompile report script with current version.

*** SCRIPT OVERFLOW ***

Not enough space for the translated report script. This is an internal error.

*** SEARCH FILE <###> Hierarchy Error ### ***

The file hierarchy has more than six levels and/or more than eight branches emanating from a node. The number in brackets is the position of the file in the SEARCH section which violated the limit. The second number is an r-tree error code from rterrc.h.

*** SEARCH RANGE VALUE ### IS OUT OF SEQUENCE ***
The range limits in the SEARCH section must be in non-decreasing order. Reorder your range limits.

*** SECTION <section_heading> REPEATED ***
Designated section appears more than once.

*** SEE ERROR CODE <###> ***
The specified error code occurred during the second parse phase of the report script. Check the error code listing for details.

*** SELECTION Level/Hierarchy Error ### ***
The SELECT expression uses incompatible fields with respect to the file hierarchy. The number is the r-tree error code from rterrc.h.

*** SET FUNCTION (LINE ###) File Hierarchy Error ### ***
A set function (on the script line indicated) attempts to combine fields from incompatible parts of the file hierarchy, or would require more than one pass to compute.

*** SET FUNCTION (LINE ###) Illegal Level (###) ***
A set function (on the script line indicated) has been applied to fields from the top of the file hierarchy.

*** SORT Segment ### is incompatible with preceding Sort Segments. ***
The sort segment indicated is not compatible (with respect to the file hierarchy) with the sort segments preceding it.

*** SORT Segment ### is incompatible with your SELECTION criteria. ***
The sort segment indicated is not compatible with the path down the hierarchy established by the SELECT expression. Either modify the SELECT expression or the sort segments.

*** Status Variable Inconsistency <###,###> ***
The number of status variables and symbolic constants has been modified since the report script was compiled. Recompile the report script.

*** STRING OPERAND NOT ALLOWED WITH SUMMATION OR AVERAGE ***
The SUMMATION and AVERAGE computational set functions cannot have string valued expressions for arguments.

*** STRING VALUED OPERAND NOT ALLOWED: Operator = op ***
A string valued operand is used in an illegal context with the specified operator.

*** STRING VALUED OPERAND REQUIRED: Operator = op ***
An operand must be string valued for use with the specified operator.

*** SYMBOL LIST DOES NOT MATCH FORMAT(s) ***
The format symbol list does not have enough entries. Note that some display logic variables used in the format symbol list

*** SYMBOL UNDEFINED ***
symbol
The symbol listed is not defined in the data object definition array, nor is it defined in any of the preceding lines of the report script.

*** SYMBOL <symbol> ALREADY DEFINED ***
The symbol has already been defined in the data object definition array, or a previous line of the report script.

*** SYMBOL <symbol> IS INCONSISTENT WITH SELECT / SORT EXPRESSIONS ***
Symbol, contained in a format symbol list, does not fall on the simple path down the file hierarchy defined by the SELECT section and the SORT section. Symbol may simply be further down the file hierarchy than the SELECT and SORT expressions go. It is always possible to add a trivially satisfied condition to the SELECT expression which will ensure that the path down the file hierarchy extends down to symbol. For example, in our customer order system, if we wish to include the order-item detail lines in our report, but neither the SELECT nor the SORT sections reference them, then add this term to the SELECT expression:

AND oi_quantity = oi_quantity

This will ensure that the path down the hierarchy extends to the order-item level.

*** TOO MANY BYTES FOR DISPLAY CONTROL STRING. LIMIT IS ### BYTES ***
The maximum number of bytes for a device control variable has been exceeded. The
#define constant MAX_DSTRING (in RTDOMDA.H) can be increased, provided the
r-tree library is remade, and any compiled report scripts are recompiled.

*** UNDEFINED USER FUNCTION ***
function_name

The named function is not part of the r-tree built-in set of functions. Review RTIVAL.H
and RTUSER.C.

*** Unexpected run-time stack value type (###) ***

Unexpected run-time error.

*** VIRTUAL <symbol> Level/Hierarchy Error ### ***

The virtual field with symbolic name symbol uses incompatible fields with respect to
the file hierarchy in its computational definition. r-tree cannot properly compute the
virtual field. In some cases, the desired result can be achieved by running two report
scripts in succession. See Section 5.6.

*** VIRTUAL <symbol> requires more than one pass to compute ***

The virtual field designated by symbol requires more than one pass to compute (due in
part to set functions) and the SELECT expression. In some cases, the desired result can
be achieved by running two report scripts in succession. See Section 5.6.

*** WORD WRAPPED Format requires string valued field (w/o functions) ***

The field to be substituted into a word wrapped format specification must be string
valued, and cannot be the result of a built-in function.

r-tree Error Codes

The r-tree® report function may return one of these error codes if the r-tree report
generator experiences a problem condition. Further, if a c-tree® error occurs during
r-tree processing, then the report function may return a c-tree error code (in the range
2 to 199) instead of one of the following codes. Remember, if a c-tree header file is
changed, everything must be recompiled! If an r-tree header is changed, everything
except c-tree must be recompiled!

<table>
<thead>
<tr>
<th>Symbolic Error Code</th>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSR_ERR</td>
<td>1</td>
<td>userfn aborts report</td>
</tr>
<tr>
<td>PSYN_ERR</td>
<td>301</td>
<td>syntax error</td>
</tr>
<tr>
<td>Symbolic Error Code</td>
<td>Error Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PSTK_ERR</td>
<td>302</td>
<td>parse stack overflow: increase MAX_PSTACK in rtpars.h and recompile r-tree</td>
</tr>
<tr>
<td>PUST_ERR</td>
<td>303</td>
<td>user stack overflow: increase MAX_USTACK in rtpars.h and recompile r-tree</td>
</tr>
<tr>
<td>PEOI_ERR</td>
<td>304</td>
<td>unexpected end of report script</td>
</tr>
<tr>
<td>PRED_ERR</td>
<td>305</td>
<td>bad case in parse reduction</td>
</tr>
<tr>
<td>PILL_ERR</td>
<td>306</td>
<td>illegal term encountered in lexical scan</td>
</tr>
<tr>
<td>PLEN_ERR</td>
<td>307</td>
<td>max. symbol length exceeded: increase MAX_TOK_LEN in rtpars.h and recompile r-tree</td>
</tr>
<tr>
<td>PDUP_ERR</td>
<td>308</td>
<td>report script section called more than once</td>
</tr>
<tr>
<td>PROV_ERR</td>
<td>309</td>
<td>program script overflow</td>
</tr>
<tr>
<td>PSMB_ERR</td>
<td>310</td>
<td>undefined SYMBOL</td>
</tr>
<tr>
<td>PCLS_ERR</td>
<td>311</td>
<td>undefined class character</td>
</tr>
<tr>
<td>PKEY_ERR</td>
<td>312</td>
<td>index key required</td>
</tr>
<tr>
<td>PMEM_ERR</td>
<td>313</td>
<td>not enough dynamic memory</td>
</tr>
<tr>
<td>PVAL_ERR</td>
<td>314</td>
<td>symbolic index name matches data field name</td>
</tr>
<tr>
<td>PCMP_ERR</td>
<td>315</td>
<td>incompatible operands</td>
</tr>
<tr>
<td>PSTR_ERR</td>
<td>316</td>
<td>string operand not allowed</td>
</tr>
<tr>
<td>PCLU_ERR</td>
<td>317</td>
<td>class member undefined</td>
</tr>
<tr>
<td>PACC_ERR</td>
<td>318</td>
<td>improper accumulator usage</td>
</tr>
<tr>
<td>PCVL_ERR</td>
<td>319</td>
<td>unexpected cvalue type</td>
</tr>
<tr>
<td>PNUN_ERR</td>
<td>320</td>
<td>non-string value in concatenate</td>
</tr>
<tr>
<td>PRST_ERR</td>
<td>321</td>
<td>run-time stack overflow</td>
</tr>
<tr>
<td>PCON_ERR</td>
<td>322</td>
<td>not enough memory for concatenation</td>
</tr>
<tr>
<td>PCST_ERR</td>
<td>323</td>
<td>unexpected number of concatenations</td>
</tr>
<tr>
<td>PBAC_ERR</td>
<td>324</td>
<td>bad action in run_prog</td>
</tr>
<tr>
<td>PFUN_ERR</td>
<td>325</td>
<td>undefined function name</td>
</tr>
<tr>
<td>PRFH_ERR</td>
<td>326</td>
<td>undefined function number in rtuser.c</td>
</tr>
<tr>
<td>PGRP_ERR</td>
<td>327</td>
<td>internal signal used with RESET_PAGE</td>
</tr>
<tr>
<td>PFILE_ERR</td>
<td>328</td>
<td>file naming error</td>
</tr>
<tr>
<td>PFR1_ERR</td>
<td>329</td>
<td>file relationship error</td>
</tr>
<tr>
<td>PRVL_ERR</td>
<td>330</td>
<td>range value sequence error</td>
</tr>
<tr>
<td>PSMEM_ERR</td>
<td>331</td>
<td>set memory error</td>
</tr>
<tr>
<td>PVER_ERR</td>
<td>332</td>
<td>compiled script incompatible with current r-tree/k-tree versions</td>
</tr>
<tr>
<td>Symbolic Error Code</td>
<td>Error Number</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PSNG_ERR</td>
<td>333</td>
<td>MAX_DSTRING in rtdoda.h changed since compilation</td>
</tr>
<tr>
<td>PMEX_ERR</td>
<td>334</td>
<td>set function conflict with file hierarchy</td>
</tr>
<tr>
<td>PREX_ERR</td>
<td>335</td>
<td>more than one pass for set function</td>
</tr>
<tr>
<td>PRFO_ERR</td>
<td>336</td>
<td>could not open report file</td>
</tr>
<tr>
<td>PRFB_ERR</td>
<td>337</td>
<td>could not allocate report script buffer</td>
</tr>
<tr>
<td>PDYM_ERR</td>
<td>338</td>
<td>could not allocate dynamic memory</td>
</tr>
<tr>
<td>PMHR_ERR</td>
<td>339</td>
<td>mixed file hierarchies</td>
</tr>
<tr>
<td>PFLV_ERR</td>
<td>340</td>
<td>too many file levels</td>
</tr>
<tr>
<td>PFHR_ERR</td>
<td>341</td>
<td>too many file branches in hierarchy</td>
</tr>
<tr>
<td>PEXL_ERR</td>
<td>342</td>
<td>set function applied to top of hierarchy</td>
</tr>
<tr>
<td>PNFM_ERR</td>
<td>343</td>
<td>run-time numeric format error</td>
</tr>
<tr>
<td>PLSH_ERR</td>
<td>344</td>
<td>SELECT / SORT incompatible</td>
</tr>
<tr>
<td>PSSH_ERR</td>
<td>345</td>
<td>SORT segments incompatible</td>
</tr>
<tr>
<td>PDVZ_ERR</td>
<td>346</td>
<td>run-time divide by zero error</td>
</tr>
<tr>
<td>PSRO_ERR</td>
<td>347</td>
<td>could not create temporary sort file</td>
</tr>
<tr>
<td>PXST_ERR</td>
<td>348</td>
<td>could not allocate memory for set function</td>
</tr>
<tr>
<td>PFSF_ERR</td>
<td>349</td>
<td>string required as function argument</td>
</tr>
<tr>
<td>PFNM_ERR</td>
<td>350</td>
<td>numeric required as function argument</td>
</tr>
<tr>
<td>PDSY_ERR</td>
<td>351</td>
<td>data field name not in DODA</td>
</tr>
<tr>
<td>PRDY_ERR</td>
<td>352</td>
<td>symbol already defined</td>
</tr>
<tr>
<td>PFRQ_ERR</td>
<td>353</td>
<td>FREQUENCY has more than 1 argument</td>
</tr>
<tr>
<td>PFRM_ERR</td>
<td>354</td>
<td>integer &gt;= 0 required</td>
</tr>
<tr>
<td>PDMX_ERR</td>
<td>355</td>
<td>too many bytes for device control variable</td>
</tr>
<tr>
<td>PDEI_ERR</td>
<td>356</td>
<td>date required as function argument</td>
</tr>
<tr>
<td>PLNG_ERR</td>
<td>357</td>
<td>long required as function argument</td>
</tr>
<tr>
<td>PLEV_ERR</td>
<td>358</td>
<td>IMAGE reference out of range</td>
</tr>
<tr>
<td>PFNA_ERR</td>
<td>359</td>
<td>accumulator cannot be function argument</td>
</tr>
<tr>
<td>PFRM_ERR</td>
<td>360</td>
<td>too few entries in format symbol list</td>
</tr>
<tr>
<td>PEND_ERR</td>
<td>361</td>
<td>end of report SIGNAL</td>
</tr>
<tr>
<td>PSVL_ERR</td>
<td>362</td>
<td>ran out of values in format symbol list</td>
</tr>
<tr>
<td>PDEV_ERR</td>
<td>363</td>
<td>output device error</td>
</tr>
<tr>
<td>PRTN_ERR</td>
<td>364</td>
<td>illegal field type in DODA</td>
</tr>
</tbody>
</table>
### Symbolic Error Code | Error Number | Description
---|---|---
PSEG_ERR | 365 | illegal sort segment
PSTI_ERR | 366 | status variable inconsistency
PCSI_ERR | 367 | could not open compiled file
PMPI_ERR | 368 | memory model inconsistency
PDOI_ERR | 369 | data object inconsistency
PKWI_ERR | 370 | keyword inconsistency
PCLN_ERR | 371 | compiler record length error
PCRD_ERR | 372 | could not read compiled script
PCDM_ERR | 373 | dynamic memory failure
PAIS_ERR | 374 | alias error
PSPC_ERR | 375 | format specification error
PWNA_ERR | 376 | active word wrap SIGNAL
PWNO_ERR | 377 | too many active word wraps; increase MAX_WW in rline.c and recompile r-tree
IDRU_ERR | 180 | could not read symbolic key name
IKMS_ERR | 181 | could not get memory for symbolic key name
IKSR_ERR | 182 | no room for extra r-tree files: increase MAXFIL in ctopt2.h or cmaxfil in ctoptn.h and recompile ctree and rtree
IDRU_ERR | 183 | could not read data field names
ISDP_ERR | 184 | attempt to reallocate set space
ISAL_ERR | 185 | not enough memory for multiple sets
ISNM_ERR | 186 | set number out of range

### Extended c-tree Error Codes
These error codes apply to c-tree® processing in conjunction with r-tree® and its augmented ISAM file definitions.

### Symbolic Error Code | Error Number | Description
---|---|---
IKRU_ERR | 180 | could not read symbolic key name
IKMS_ERR | 181 | could not get memory for symbolic key name
IKSR_ERR | 182 | no room for extra r-tree files: increase MAXFIL in ctopt2.h or cmaxfil in ctoptn.h and recompile ctree and rtree
IDRU_ERR | 183 | could not read data field names
ISDP_ERR | 184 | attempt to reallocate set space
ISAL_ERR | 185 | not enough memory for multiple sets
ISNM_ERR | 186 | set number out of range
<table>
<thead>
<tr>
<th>Symbolic Error Code</th>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRBF_ERR</td>
<td>187</td>
<td>null buffer in rtread.c</td>
</tr>
<tr>
<td>ITBF_ERR</td>
<td>188</td>
<td>null target buffer in rtread.c</td>
</tr>
<tr>
<td>IJSK_ERR</td>
<td>189</td>
<td>join_to skip</td>
</tr>
<tr>
<td>IJER_ERR</td>
<td>190</td>
<td>join_to error</td>
</tr>
<tr>
<td>IJNL_ERR</td>
<td>191</td>
<td>join_to null fill</td>
</tr>
<tr>
<td>IDSK_ERR</td>
<td>192</td>
<td>detail_for skip</td>
</tr>
<tr>
<td>IDER_ERR</td>
<td>193</td>
<td>detail_for error</td>
</tr>
<tr>
<td>IDNL_ERR</td>
<td>194</td>
<td>detail_for null fill</td>
</tr>
<tr>
<td>IDMU_ERR</td>
<td>195</td>
<td>could not get memory for symbolic data field</td>
</tr>
<tr>
<td>ITML_ERR</td>
<td>196</td>
<td>RETRY_LIMIT (due to c-tree® error ITIM_ERR) exceeded in rtread.c</td>
</tr>
</tbody>
</table>
This page is intentionally left blank.
Index

values ............................................ 44
CONTROL ........................................ 55, 120, 146
Control break .................................. 55, 146
See also Control field
summary .......................................... 56
Control field ..................................... 64, 146
breaks ............................................. 57
also .................................................. 120
Control footer ................................... 64
CONTROL_FTR .................................. 60, 64, 109, 146
Control header
CONTROL_HDR .................................. 60, 109, 146
Control strings ................................. 59
cos6 - Numeric format example ............... 117
COUNTER ........................................ 92, 146, 148, 167
CURRENCY_MARK ................................ 58, 115, 127, 147
Currency mark .................................... 115
cerror.h ........................................... 71
cerror.h ........................................... 29, 37
Currency mark .................................... 115
floating ........................................... 115
CURRENCY_MARK ................................ 58, 115, 127, 147
evalue ............................................. 44, 147

D
D1 .................................................. 100, 150
D2,D3,D4 .......................................... 136
See D1
DASH ............................................... 147
DAT_D ............................................. 148
DAT_DA ........................................... 148
DAT_DAY ......................................... 100, 114
DAT_DD ........................................... 100, 114
DAT_M ............................................. 148
DAT_MM ........................................... 100, 114
DAT_MT ........................................... 148
DAT_MTH ......................................... 100, 114
DAT_YYYY ....................................... 148
DAT_YYYYY ....................................... 100, 114
Data
Data File Description Record .................. 39
Data record fields ............................... 34
Data File Description Record .................. 19
definitions ....................................... 97
Data Object Definition Array
See DODA
database .......................................... 27
Date ............................................... 44
DATE ............................................. 45, 147
Date-to-string conversion ....................... 114
DATE_ORDER ..................................... 58, 114,
DATE_ORDER ..................................... 127 - 128, 147, 161
DATE_ORDER ..................................... 164
DATE_SEPARATOR ................................ 58, 114, 127 - 128,
DATE_SEPARATOR ................................ 147, 164
Dates and Alphanumeric Outputs ............... 114
used in control footers ......................... 57
DATOBJ .......................................... 18, 33
DBL_ST ........................................... 148
DBL_STR .......................................... 100
Debit notation ................................... 133
Debugging ........................................ 169 - 170
Decimal point .................................... 115
DECIMAL_POINT ................................ 58, 115, 127, 148, 166
DECREMENT ....................................... 92, 100,
[Continued on next page]
Embedded characters 113
ENDING_PAGE 59, 127, 150
EQLREC 28, 31, 48, 150
Error c-tree error codes 194 - 196
tree error codes 191
tree error messages 183 - 190
EXISTS 86, 150
EXIT 23, 43, 150
Exponential form 111, 117

Field definitions: 45
delimiter 29
length 36
value 118, 120
FILE 23, 150
Alias 95
file interrelationship alternatives 98
file interrelationship summary 32
file relationship keywords 99
files and r-tree 27
hierarchy 49, 55, 79, 87, 151
hierarchy analysis 92
FILENAME 151
FIRST 83, 151
First field 39
First occurrence 56
FIRST_PAGE_HDR 58, 127, 151, 161
Fixed length c-tree data file 118
Float field 54
Floating currency mark
See Currency mark
FORALL 84, 86, 151
FORM_FEED 61, 122, 129, 152, 159

Format
FORMAT 61, 122, 134, 152
format characters 60
format problem 116
format specification 151
format symbol list 24, 60, 109, 119, 124, 151
FORMAT_MARK 58, 60, 110, 112 - 113, 127 - 128,
151 - 153, 163
specification 116
specifications 110, 119
FREQUENCY 83, 85, 106, 152
Frequency count 56
FRQ 23, 56, 66, 152
FRS 56, 66, 153
FRSSET 28 - 29, 31, 49, 81, 99, 153, 159
function arguments 44
specifications 116
specifications 110, 119

Global variables 25

Include files
INCREMENT 92, 100, 146, 154 - 155, 167
Incremental ISAM structure 37
Index
file description record 20, 39
## Index

<table>
<thead>
<tr>
<th>File linkage</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>File member record</td>
<td>20, 40</td>
</tr>
<tr>
<td>Symbolic name</td>
<td>39</td>
</tr>
<tr>
<td>INPAPND</td>
<td>154</td>
</tr>
<tr>
<td>INPFILe</td>
<td>118, 154</td>
</tr>
<tr>
<td>INT 4</td>
<td>65, 154</td>
</tr>
<tr>
<td>INT2</td>
<td>65, 120, 154</td>
</tr>
<tr>
<td>INT4</td>
<td>154</td>
</tr>
<tr>
<td>INT4U</td>
<td>45</td>
</tr>
<tr>
<td>INTEGER</td>
<td>54, 154</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>72, 101, 126, 154</td>
</tr>
<tr>
<td>Interface program</td>
<td>76</td>
</tr>
<tr>
<td>IS_ALIAS_FOR</td>
<td>143, 151, 154</td>
</tr>
<tr>
<td>IS_AM</td>
<td>49, 79, 93, 96, 98-99, 144, 153, 155</td>
</tr>
<tr>
<td>IS_Detail_FOR</td>
<td>49, 150, 155</td>
</tr>
<tr>
<td>IS_Detail_FOR_Relationship</td>
<td>79</td>
</tr>
<tr>
<td>IS_Detail_FOR_ERROR</td>
<td>99, 150, 155</td>
</tr>
<tr>
<td>IS_Detail_FOR_NULL</td>
<td>99, 150, 155</td>
</tr>
<tr>
<td>IS_Detail_FOR_SKIP</td>
<td>99, 155</td>
</tr>
<tr>
<td>ISAM</td>
<td>17</td>
</tr>
<tr>
<td>JOINS_TO</td>
<td>48, 52, 79, 82, 96, 98, 144, 150-151, 155</td>
</tr>
<tr>
<td>JOINS_TO_ERROR</td>
<td>99, 155</td>
</tr>
<tr>
<td>JOINS_TO_NULL</td>
<td>99, 155</td>
</tr>
<tr>
<td>JOINS_TO_SKIP</td>
<td>99, 155</td>
</tr>
<tr>
<td>Relationship</td>
<td>79</td>
</tr>
<tr>
<td>JTE</td>
<td>99, 155</td>
</tr>
<tr>
<td>JTN</td>
<td>99, 155</td>
</tr>
<tr>
<td>LTS</td>
<td>23, 56, 65-66, 139, 156</td>
</tr>
<tr>
<td>LSTSET</td>
<td>99</td>
</tr>
<tr>
<td>Keys</td>
<td></td>
</tr>
<tr>
<td>Alternate collating sequence</td>
<td>144</td>
</tr>
<tr>
<td>Exact key match</td>
<td>28, 150</td>
</tr>
<tr>
<td>Key Segment</td>
<td>29</td>
</tr>
<tr>
<td>Key Sequential Order</td>
<td>47</td>
</tr>
<tr>
<td>Key Sequential Order</td>
<td>47</td>
</tr>
<tr>
<td>Partial key match</td>
<td>159</td>
</tr>
<tr>
<td>Partial key match</td>
<td>28</td>
</tr>
<tr>
<td>Partial key search</td>
<td>28</td>
</tr>
<tr>
<td>Range restrictions</td>
<td>47</td>
</tr>
<tr>
<td>Keys</td>
<td></td>
</tr>
<tr>
<td>LAST</td>
<td>83, 155</td>
</tr>
<tr>
<td>Last accumulator</td>
<td>65</td>
</tr>
<tr>
<td>Last field</td>
<td>39</td>
</tr>
<tr>
<td>Last occurrence</td>
<td>56</td>
</tr>
<tr>
<td>LAST_PAGE_FTR</td>
<td>58, 127, 155, 161</td>
</tr>
<tr>
<td>Left justified alphanumeric string</td>
<td>111</td>
</tr>
<tr>
<td>LEFT_MARGIN</td>
<td>61, 122, 156</td>
</tr>
<tr>
<td>LEFT_STR</td>
<td>101, 148, 156</td>
</tr>
<tr>
<td>Library</td>
<td>69</td>
</tr>
<tr>
<td>LINE_MARK</td>
<td>58, 60, 110, 127-128, 156</td>
</tr>
<tr>
<td>LINE_NO</td>
<td>89, 156</td>
</tr>
<tr>
<td>Lines</td>
<td>126</td>
</tr>
<tr>
<td>Link files</td>
<td>28, 156</td>
</tr>
<tr>
<td>Linking r-tree</td>
<td>69</td>
</tr>
<tr>
<td>LNG_STR</td>
<td>101, 156</td>
</tr>
<tr>
<td>Logical comparisons</td>
<td>65</td>
</tr>
<tr>
<td>Logical record</td>
<td>85</td>
</tr>
<tr>
<td>Logically joins</td>
<td>93</td>
</tr>
<tr>
<td>LPT1</td>
<td>24</td>
</tr>
<tr>
<td>LST</td>
<td>23, 56, 65-66, 139, 156</td>
</tr>
<tr>
<td>LSTSET</td>
<td>99</td>
</tr>
<tr>
<td>Money</td>
<td>44-45, 65, 157</td>
</tr>
<tr>
<td>Matching records</td>
<td>98</td>
</tr>
<tr>
<td>Master Files</td>
<td>27, 156</td>
</tr>
<tr>
<td>Many-to-one</td>
<td>48</td>
</tr>
<tr>
<td>MAX</td>
<td>23, 56, 66, 156</td>
</tr>
<tr>
<td>MAX_DSTRING</td>
<td>124</td>
</tr>
<tr>
<td>Maximum</td>
<td>56</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>83, 88, 156</td>
</tr>
<tr>
<td>Memory script</td>
<td>77</td>
</tr>
<tr>
<td>Message Pointer</td>
<td>72</td>
</tr>
<tr>
<td>MIN</td>
<td>23, 56, 66, 157</td>
</tr>
<tr>
<td>Minimum</td>
<td>56, 83, 88, 157</td>
</tr>
<tr>
<td>Missing Data</td>
<td>169</td>
</tr>
<tr>
<td>Missing Records</td>
<td>98</td>
</tr>
<tr>
<td>Modifying Device Strings</td>
<td>129</td>
</tr>
<tr>
<td>Money</td>
<td>44-45, 65, 157</td>
</tr>
</tbody>
</table>
SUBSTRING .......................... 101 - 102, 164
String Comparisons .................. 101
SUM .................................. 23, 56, 66, 165
Summary information .................. 55
Summation .......................... 56, 83, 85, 106, 165
symbol .................................. 44
Symbolic Field Names .................. 44
Symbolic index name .................. 39
Symbolic name .................. 33, 35 - 36, 39, 44
Syntax Checker .......................... 70
SYNTAX ERROR .................. 24, 52, 60, 165

T
Target .................................. 49
Target field .................................. 57
Temporary files .................. 106
TEST_LINE .......................... 89, 121, 123, 153, 165
TEST_PAGES .................. 59, 127, 165
Thousand separator .................. 115
automatic .................. 115
THOUSAND_SEPARATOR .................. 58, 115, 127, 166
Typedef .......................... 33

U
UCASE .................................. 101, 166
Unformatted data .................. 111
Unique index .................. 28
UPPER .................................. 23, 54, 166
User defined function .............. 25, 72, 74, 166
userfn
See User defined function
USING_KEY .................. 23, 48, 97, 99, 166
USING_REVKEY .................. 99, 166

V
value .................................. 44
Variable Length Files .................. 29
Varying length fields .................. 29
VERT_TAB .......................... 62, 121, 123, 166
Very Short Format Specifications ......... 119
VIRTUAL .................................. 23, 43, 45 - 46, 166
Virtual field .................. 86 - 91
assignments .................. 88

W
WILDCARD .......................... 101 - 102, 166
Word-wrapped format .................. 113
output .................. 111

Y
YES .................................. 166

Z
Z1 .................................. 101
Z1, .................................. 167
ZERO .................................. 92, 101, 167
ZERO_FILL .......................... 58, 115, 127, 167