



Generix Collaborative
Infrastructure



TradeXpress

RTE

Reference Guide

Software Version: 5.2

Date: September 2013

Generix Group Solution

Terms of use

Copyright

Generix Group may hold patents, applications for patents, or may own rights to trademark, copyright or other intellectual property rights covering all or part of the components covered by this document. This guide does not constitute in any case a license for these above-mentioned patents, trademarks copyrights or intellectual property rights, unless otherwise stated in a written license agreement issued by Generix Group.

© 2013 Generix Group. All rights reserved.

Disclaimer

Generix Group reserves the right to make changes to this document without prior notice. Except where otherwise stated, names of companies, organizations, products, persons or events mentioned in this document are given as example only. Any resemblance to actual names or events is coincidental. Users must comply with the Copyright law applicable in the country of use. No part of this document shall be reproduced, stored or entered into a retrieval system, or disseminated for any reason or by any means whatsoever, without the express written consent of Generix Group.

Trademarks

The names of companies and products mentioned in this document are trademarks or registered trademarks of their respective owners.

Version History

Document version	Revision date	Product version	Contents
1.0	-	-	Initial version
1.1	24/09/2013		Layout modification. Script correction. Removal of the "group" argument for the <code>valid</code> function.
1.2	15/11/2013		Added: "Error detection within Pass-through mode" section, "XML to XML translator" and "RTE LDAP Library" appendices, information on decimal separator in EDI to EDI translators.
1.3	12/12/2013		Added: SFTP test programm Windows compilation command line. Modified: SFTP test programm Unix compilation command (libraries' order).

Table of Contents

Terms of use	2
Copyright	2
Disclaimer	2
Trademarks	2
Version History	3
Table of Contents	4
Introduction 11	
Prerequisites	11
Overview of RTE	11
Operating Principle	12
The Window Model	12
RTE Compilation	13
Chapter 1 - Tutorial	15
1.1 Overview	16
1.2 The First Program	16
1.3 A Line Counter	17
1.4 Building an EDI Message	19
1.4.1 Specifying the Message	19
1.4.2 Data Lines	20
1.4.3 End Statement List	21
1.4.4 User Functions	21
1.4.5 RTE Program Source Code	21
1.4.6 The Inhouse Data File	24
1.4.7 The Resulting Messages	25
1.4.8 The Resulting Log File	25
1.5 Receiving an EDI Message	26
1.5.1 Specifying the Message	26
1.5.2 Buffered Output	26
1.5.3 RTE Program Source Code	26
1.5.4 The Original Messages	28
1.5.5 Resulting Data File	28
Chapter 2 - Program structure	29
2.1 Overview	30

2.2	Statement Lists	30
2.3	Line Matching	31
2.3.1	Text at a Given Position.....	31
2.3.2	Text Anywhere	31
2.3.3	Line Number	32
2.3.4	Logical Operators.....	32
2.3.5	Regular Expression Matching	32
2.3.6	Multi Matching	33
2.4	Definitions	33
2.4.1	Message Definition	33
2.4.2	Database Definition	34
2.5	Functions.....	34
2.6	Comments.....	35
2.6.1	Source Code Documentation	35
2.6.2	Runtime Information.....	35
Chapter 3 -	Program statements	36
3.1	Overview	37
3.2	Assignment Statements.....	37
3.2.1	Text Assignments	37
3.2.2	Numeric Assignments	37
3.2.3	Boolean Assignments.....	38
3.2.4	File Assignments	38
3.2.5	Database Assignments	38
3.2.6	Array Assignments	38
3.3	Increment and Decrement Statements	38
3.4	Control Statements.....	39
3.4.1	Conditional Statement.....	39
3.4.2	Boolean Expressions.....	39
3.5	Loop Statements.....	40
3.5.1	General Loop Statement	40
3.5.2	Array Scanning	40
3.5.3	List Scanning	41
3.5.4	Directory Scanning	41
3.5.5	Database Scanning	42
3.5.6	Switch Statement.....	42
3.5.7	Switch Expression	43
3.5.8	Switch Statement with Coprocesses	43
3.5.9	Break Statement.....	43
3.5.10	Continue Statement.....	43
3.5.11	Next Statement	44

3.5.12	Statement	44
3.5.13	RTE exceptions.....	44
3.5.14	Inline C Block Statement	48
3.5.15	Inline C Statement	48
Chapter 4 -	EDI translation	49
4.1	Overview	50
4.2	Operating Principle	50
4.3	Specifying the Message	50
4.4	Naming the Elements	51
4.5	Building Messages.....	53
4.5.1	Segment Building Statement	53
4.5.2	Element Assignments	54
4.5.3	Packing Data in Repeating Elements.....	55
4.5.4	Decimal Separator Conversion	57
4.5.5	Specifying the Segment Location.....	57
4.5.6	Specifying the Segment Repetition	59
4.5.7	Checking the Syntax	60
4.5.8	Segment Counter	60
4.6	Independent Segments	61
4.6.1	Independent Segments	61
4.7	Receiving EDI Messages	61
4.7.1	Decimal Separators	63
4.8	Receiving Direction Error Handling.....	63
4.8.1	Ignoring Erroneous Messages	63
4.8.2	Using a Separate Error Routine	63
4.8.3	Processing Erroneous Messages in Statement Lists	64
4.8.4	Preparing for the EDIFACT CONTRL Message	64
4.8.5	Checking the Validity of a Segment	65
4.8.6	Checking the Validity of a Group	66
4.8.7	Examples.....	66
4.8.8	Error Codes	68
Chapter 5 -	Input and Output	69
5.1	Standard Streams.....	70
5.2	Input	70
5.2.1	Window Model.....	70
5.2.2	Picking.....	70
5.2.3	Read	71
5.3	Output	72

5.3.1	Printing	72
5.3.2	Newline Character	72
5.4	Redirection	72
5.4.1	Using Coprocesses	74
5.4.2	Redirecting the Standard Streams.....	74
5.5	Buffered Output	75
5.5.1	Putting Text in the Buffer	75
5.5.2	Flushing the Output Buffer	75
5.5.3	Examples.....	76
5.6	Print List	76
5.7	Print Items	77
5.8	Print Format	77
5.8.1	Text Print Format	77
5.8.2	Numeric Print Format.....	77
5.8.3	Time Printing.....	78
5.9	Logging	79
5.10	File Variables.....	80
5.10.1	File Attributes.....	80
5.10.2	Assignment	80
5.10.3	Usage	81
Chapter 6 -	Database Interface.....	82
6.1	Background Information.....	83
6.2	Definition	83
6.3	Database Fields	84
6.4	Database Files.....	85
6.5	Finding an Entry	85
6.6	Creating a New Entry	86
6.7	Removing an Entry	87
Chapter 7 -	Running RTE Programs	88
7.1	Overview	89
7.2	Options	89
7.3	Parameters	90
7.4	Arguments	91
7.5	Startup File	92
Chapter 8 -	Built-in Functions Reference	94
8.1	Overview	95
8.2	Function Categories.....	95

8.2.1	Process Control	95
8.2.2	Text Handling	95
8.2.3	File Handling	96
8.2.4	Output	96
8.2.5	Database Access	97
8.2.6	Input	97
8.2.7	Multi-Purpose	97
8.2.8	Time	97
8.3	Functions Reference	98
8.3.1	background	98
8.3.2	build	98
8.3.3	close	99
8.3.4	compare	100
8.3.5	copy	101
8.3.6	debug	102
8.3.7	edierrordump	103
8.3.8	edierrorlist	103
8.3.9	exec	104
8.3.10	exit	105
8.3.11	find	106
8.3.12	flush	107
8.3.13	index	107
8.3.14	length	108
8.3.15	link	109
8.3.16	load	110
8.3.17	log	114
8.3.18	new	114
8.3.19	number	115
8.3.20	peel	115
8.3.21	pick	116
8.3.22	print	117
8.3.23	put	118
8.3.24	read	119
8.3.25	redirect	119
8.3.26	remove	120
8.3.27	rename	121
8.3.28	replace	122
8.3.29	spawn	123
8.3.30	split	124
8.3.31	strip	124
8.3.32	substr	125
8.3.33	system	126

8.3.34	time.....	127
8.3.35	tolower.....	131
8.3.36	toupper.....	131
8.3.37	valid.....	132
Chapter 9 -	Appendices	133
9.1	Appendix A: Identifier Types	134
9.1.1	Overview	134
9.1.2	Basic Types	134
9.1.3	Special Types.....	135
9.2	Appendix B: Regular Expressions	136
9.2.1	Regular Expression Usage	136
9.2.2	Regular Expression Definition	136
9.3	Appendix C: Message Storage Format.....	139
9.3.1	Message World	139
9.3.2	Message Description Files	139
9.4	Appendix D: Character Sets.....	144
9.4.1	Character Set Definition	144
9.4.2	The Description Language	144
9.5	Appendix E: Code Conversion in RTE.....	148
9.6	Appendix F: Advanced Utilities	150
9.6.1	Introduction	150
9.6.2	Pass-through Mode in RTE.....	150
9.6.3	Segment Look-ahead in the Receiving Translator	156
9.6.4	C Programming in RTE	159
9.6.5	RTE Coprocess Communications	161
9.6.6	RTE FTP Client Routine Library	168
9.7	Appendix G: RTE SQL Access Library	180
9.7.1	Introduction	180
9.7.2	RTE SQL Access Library Reference List	181
9.7.3	ODBC interface	184
9.7.4	JDBC Interface	184
9.8	Appendix H: XML to XML translator	189
9.8.1	Principles	189
9.8.2	Syntax	191
9.8.3	Building an XML to XML translator.....	193
9.9	Appendix I: RTE LDAP Library	195
9.9.1	Prerequisites	195
9.9.2	Compiling your RTE program	195
9.9.3	RTE Library function	195

9.9.4	Example of RTE file using all the functions available.....	199
-------	--	-----

Introduction

Prerequisites

This manual assumes that you are somewhat familiar with the EDI syntax you are using. Terms such as “message,” “segment,” and “data element” are used without explanation. A basic understanding of computer operation is also assumed, including the use of terms such as “executable program,” “input file,” “output file,” and “program source code.”

Parts of this manual assume that you have at least a basic knowledge of computer programming with a high-level language such as C, PASCAL, or FORTRAN. You should be familiar with terms such as “variables,” “functions,” “expressions,” and “statements” in order to understand the entire structure of the language.

These assumptions aside, RTE is easy to program and programming is rarely needed in EDI translations. The translators usually contain only simple assignments or references to data elements, and often these are automatically produced by the TradeXpress tools.

When programming is needed, RTE is much easier to handle than conventional high-level languages. Most of the tedious work relating to memory management, file handling, process startup, and other advanced features are taken care of transparently.

Overview of RTE

Although RTE was originally designed for EDI translation, it is suitable for a wide range of other applications. In many ways, it resembles a Unix text processing program called *awk*. Like *awk*, it is data driven, it supports associative memory, variables are used with no prior definition, and so on. The most obvious differences are the EDI translation module, links to EDIBASE databases, and a large suite of built-in functions in RTE. Also, RTE programs are compiled into executable modules, whereas *awk* is an interpreter.

RTE is a high-level programming language with its own memory management and associative memory. These features allow you to concentrate on building your application rather than worrying about reserving room for all variables, checking array boundaries, freeing unused memory segments, and similar tasks common to conventional high-level languages.

The EDI translator module has no EDI syntax (like EDIFACT or ANSI X12) rules built into it, but the syntax used is derived from the message description files. RTE can then be used to create a translator for a specific grammar, but since the rules are not a part of RTE, future versions and completely new grammars can also be supported with the same tool.

Outgoing EDI messages are built one segment at a time. The order in which the segments are built is completely free: the EDI translator module inserts the segments in their respective positions in the message. This architecture obviates the need for any of the tailored preprocessors commonly used today to “translate the inhouse data into a format ready to be translated to EDI format.” Data segments are referenced by their names and positions in the message tree. Data elements in segments are referenced by their names.

Associative memory makes it possible to use free form text as an array index side by side with numeric indexes. The length of the array index or the value is almost unlimited ("almost" is a necessary qualifier because each computer has only a finite amount of memory).

Input from files can be taken one line at a time and run through the matching line statement list (data-driven operation), or the program can be designed in a linear way so that the input is explicitly read in. Text can be retrieved from a specified position in a file and in output it can be inserted in a specified position.

RTE contains text processing functions to perform various tasks for text data. These include taking a substring, stripping off extraneous characters, combining several strings to one, and converting strings to upper or lower case.

RTE supports numeric data types that can be used in arithmetic calculations. Numeric values can be integers or floating point numbers.

RTE supports user written functions. Functions can be written to handle actions that are performed at several instances to avoid redundant coding. These functions can receive arguments and return values. Recursive functions are not supported.

The file system interface provides mechanisms to scan directories, print and remove files, and access file attributes. Multiple files can be read line by line, and likewise the output of the program can be written to several files.

The code conversion facility aids in mapping between different coding systems used by communicating partners. A single conversion table can be used for multiple partners and in both directions.

RTE contains built-in functions to start new processes. These processes can be waited for, they can be put to background mode, or they can be used to replace the running program. Coprocesses can be used just like regular text files in input and output operations.

With the C-language interface, any software library with a C-interface can be linked to an RTE program. This makes it possible, for example, to use an SQL library to fetch data elements directly from a relational database with no intermediate file.

The C-preprocessor can be used to define symbols and macros, include code from other files, and perform conditional compilation

Operating Principle

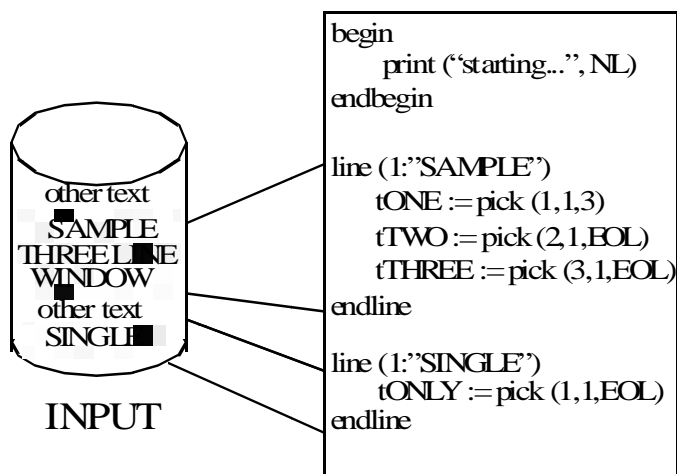
By default, an RTE program is data driven and reads input one line at a time.

The Window Model

There is an imaginary window that slides over the data. The window is anchored at a position by matching a given condition in the file. The matched line becomes the first line of the window and the window size is determined by the furthest reference relative to the matched line. These references are made with the built-in function pick. The size of a window can range from one line to a maximum of 1024 lines.

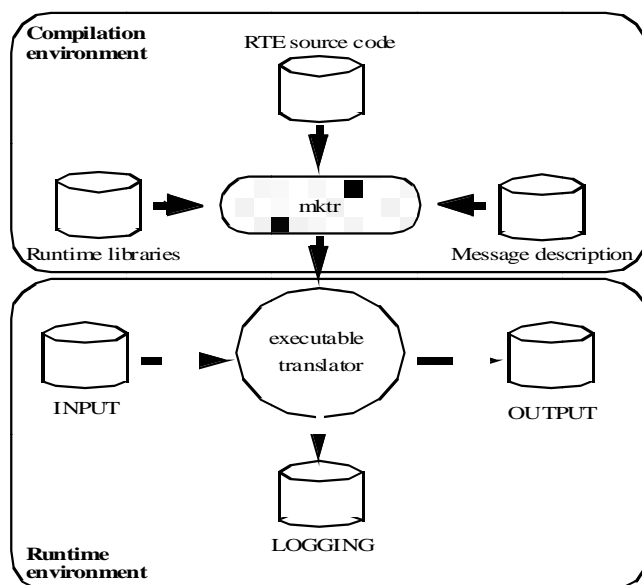
In the following figure, the input file is read in line by line. The line "SAMPLE" matches the first line matching condition, which describes the operation principle of the translator when dealing with line matching conditions, so the entire line statement list is executed. The line statement list itself reads two additional lines from the input by doing a pick from lines two and three relative to the matching

line. The next line read from the file is "other text," which does not match any line statement list. The line "SINGLE" again matches a condition and the second line statement list is executed



RTE Compilation

RTE programs are compiled into executable programs. The source code contains the reference to the message description that may be used, and this is loaded by mktr at compile time. Depending on the functionality used (that is, which built-in functions are called), runtime libraries are loaded automatically by mktr. The figure below gives a quick overview of the compilation and runtime environments. There are no additional references to the actual message description file at runtime, since it is compiled into the executable module.



The RTE compiler is named mktr (for make translator). The compiler is started from the command line, as follows:

```
mktr [options] filename [filename, ...]
```

For example:

```
$ mktr sample.RTE
```

Several options can be used to affect the behavior of the compiler.

- a** Causes all matching statement lists to be executed instead of the default action, where only the first match is executed.
- Afile** File to be used instead of \$EDIHOME/lib/ccargs.
- Copt** The option is given to the C-compiler.
- Dopt** The option is defined when preprocessing RTE.
- e** Does not generate code, just checks the grammar. This option is used by the TradeXpress editor to perform RTE grammar checks.
- g** Produces symbol table information for symbolic debugging. This option is not available in Windows NT.
- h** Gives a summary of available options.
- ldir** Adds pathname to the list of directories to be searched when an #include file is not found in the directory containing the current source file, \$HOME/include, \$EDIHOME/include, or whenever angle brackets (< >) enclose the file name. If the file cannot be found in directories in this list, directories in a standard list are searched. -I can be used several times, but only applies to source files appearing after it on the command line.
- m** The resulting translator keeps the message after the print and log commands and it must be explicitly removed. This enables the writing of the same message to several files.
- o file** The resulting file is named after this argument. The default name for the resulting translator is the source file name without the .RTE extension.
For example, to produce an executable file named test:

```
$ mktr sample.RTE -o test
```
- v** Does not generate code, just produces a list of used variables. This list can be used, for example, to detect errors caused by mistyping a variable name and therefore not getting the expected results. The listing shows the number of assignments and references for each variable.
- V** Prints out various command lines before they are executed.
- w** The resulting translator keeps the write buffer after the flush command and it must be explicitly removed. This allows for writing the buffer at several stages of filling, for debugging purposes, for example.



Chapter 1 - Tutorial

Overview

The First Program

A Line Counter

Building an EDI Message

Receiving an EDI Message

1.1 Overview

This chapter provides a tutorial introduction to the RTE functionality. The intention is to get you started writing your own programs, since that is the only way you can really learn a programming language. For this reason, most of the advanced features are not covered here.

The first part of this chapter covers the basic principles and simple text processing tasks. The second part explains how an EDI message is built, and the third part presents an example in which an EDI message is received.

1.2 The First Program

Here is the first program you can try:

```
!- - - - -
! A humble beginning...
!- - - - -
begin
    print ("The first program", NL)
endbegin
```

The exclamation mark starts a comment that continues to the end of the line. The program above contains only one statement list (begin), which in turn contains only one statement. The begin statement list is terminated with the keyword endbegin, which in this case is the end point of the entire program.

Save the program to a file. You can name the file anything as long as you use an .rte extension so that the translator maker (mktr) will recognize it. Let's assume the program was saved under the name "first.rte":

```
$ mktr first.rte
$
```

The mktr command generates an executable program named "first" that can now be run from the command line.

```
$ first
The first program
$
```

You have now completed your first program.

1.3 A Line Counter

In this section, we write a simple line counter that counts the number of empty and nonempty lines in a file. We use three statement lists to accomplish this task: two line statement lists that are triggered by the lines in the file, and an end statement list to print out the information at the end.

```
!=====
! A simple line counter
!=====
!-----
! Lines that contain no characters trigger this statement list.
!-----
line (")
    nEmpty++
endline
!-----
! Lines that contain any characters trigger this statement list.
!-----
line (not ")
    nNonEmpty++
endline
!-----
! Having read all the lines in the input this statement list is run.
!-----
end
    print ("There were ", nEmpty, " empty lines", NL)
    print ("and ", nNonEmpty, " lines with text.", NL)
endend
```

Variables do not need to be defined before they are used. Numeric variables, such as `nEmpty` and `nNonEmpty` above, always start with a lower case “n” and initially contain the value zero.

The `print` function takes as its arguments a variable number of text and numeric items. There is no automatic line feed insertion, so we use the built-in newline character, `NL`. Note that inserting an actual line feed in the middle of the `print` function will cause an error. If you try something like:

```
print ("and ", nNonEmpty, " lines with text.")
```

the RTE compiler will produce an error message about a new line in a string.

After compiling the above program under the name “counter,” you can test it. If you do not redirect the input to be from a file, the program waits for you to type the text from the keyboard. To terminate the input, press `ctrl-d` at the beginning of a line in the Unix environment, or `ctrl-z` in the NT environment.

```
$ counter
TEXT
TEXT

TEXT
TEXT
```

```
There were 2 empty lines  
and 4 lines with text.  
$
```

Now, assume that you have a file named `model.txt` that contains the following lines:

```
some text  
some text  
  
again some text
```

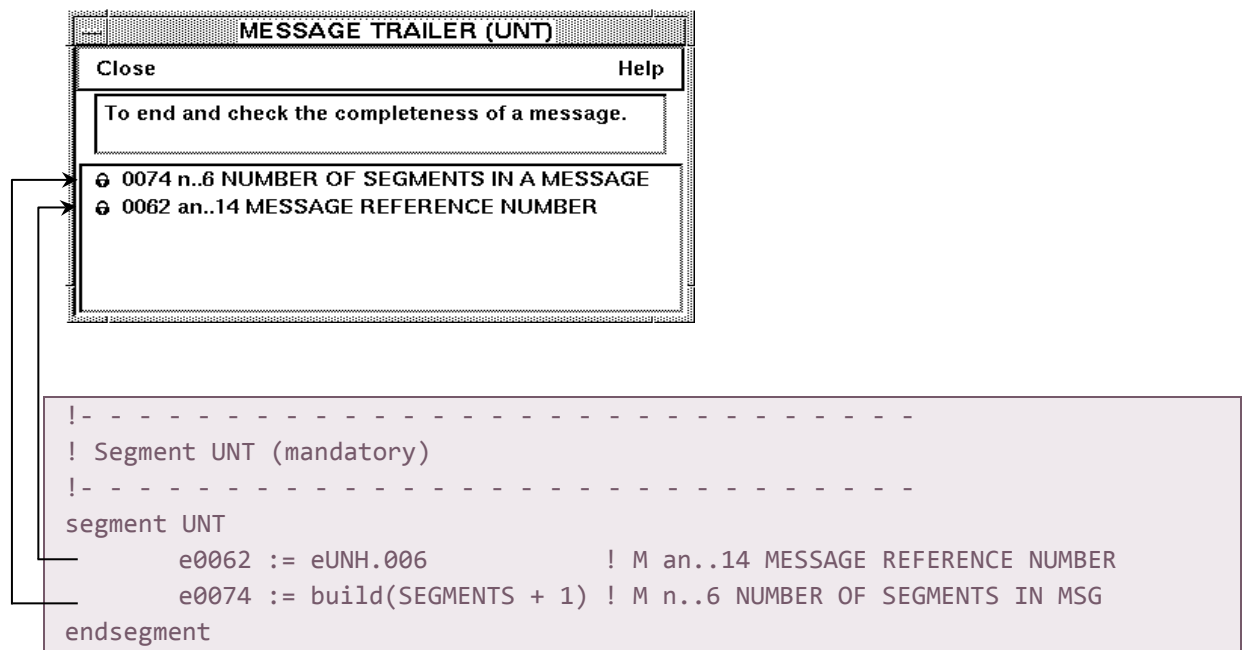
You could run your new counter for the file, as shown below:

```
$ counter < model.txt  
There were 1 empty line and 3 lines with text.  
$
```

1.4 Building an EDI Message

For the sake of simplicity, we will use a very small subset of the UN/EDIFACT INVOIC.2.912- message. This subset is not usable in any actual application, but for the sake of this example, it allows us to cover the most basic principles of building an EDI message.

The governing principle in the translator is to read the input file one line at a time and collect and insert the data to the segments. When all the necessary segments have been inserted, the message is written to the output. A segment must be written as one unit, but the segments can be written in any order.



The elements can be assigned values in any order in the segment building statement. Only text values can be assigned to the elements. Values can be assigned only to the elements in the current segment, but values from previously inserted segments can be referenced (see the reference to an element in UNH above).

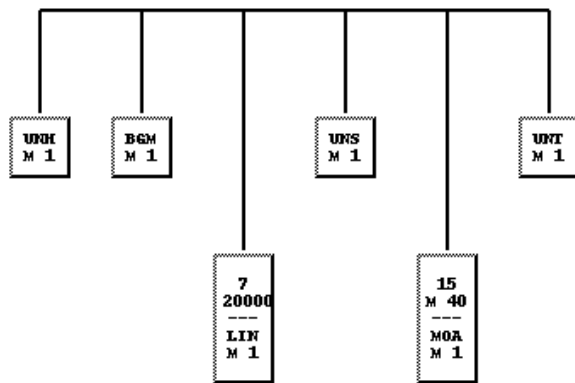
An element is always referenced by its own name preceded by a lower case "e".

1.4.1 Specifying the Message

The program states at the beginning which message it applies to.

```
message "UN-EDIFACT/91.2/invoic.msg" building
```

The path for the file containing the message description is given first, and then the working direction. In this case we are building EDIFACT from inhouse data, so we define the direction as "building."



1.4.2 Data Lines

The sample inhouse file contains only two types of records: lines starting with the text "INVOICE#" contain header information for the entire message, and lines starting with "LINE#" contain detailed information for line items.

```
INVOICE#123001 INVOICE FOR DISTRIBUTOR
LINE#1 3 007234110128.50
LINE#2 30072341502120.00
```

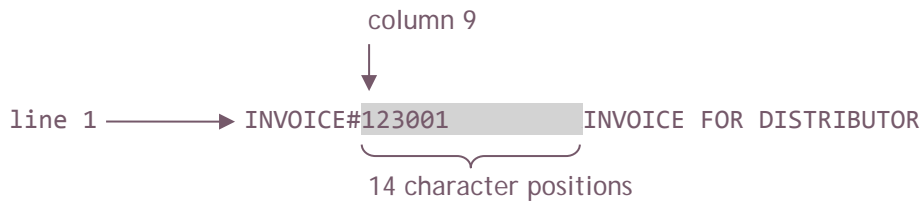
The translator reads the input file and executes a matching line statement list for each line. The program contains two line statement lists:

```
line (1:"INVOICE#")
...
endline

line (1:"LINE#")
...
endline
```

For each line starting with "INVOICE#," the first line statement list is executed. In the line statement list, a user-written function, `nfCompletePrevious()`, is called first to complete the previous message (see below). Then, segments UNH, BGM, and UNS are inserted in the message. Data for UNH and BGM is partly constant and partly retrieved from the inhouse file. The figure below illustrates the principle of the pick function used in the line statement lists to retrieve data from the inhouse file.

```
line (1:"INVOICE#")
...
segment UNH
...
e0062 := pick (1,9, 14)
...
endsegment
endline
```



Segment UNS is mandatory in the invoice message (although it would not be necessary in this application), so we must insert it in the message. The UNS segment must come after the line items, but it can be inserted here. For lines starting with "LINE#," the second line statement list is executed. From group seven, we use only the LIN segment. Note that in this subset there is no actual group seven, because there is only one segment (LIN), but the segments are always named and referenced according to the full message description.

```
segment LIN g7
...
endsegment
```

1.4.3 End Statement List

In the end statement list, we call the function `nfCompletePrevious` to ensure that the last message is properly terminated and written to the output. We also write the total number of messages to the logging file.

1.4.4 User Functions

Functions can be used whenever exactly the same action must be taken in two or more instances. For this EDI message, we call the function `nfCompletePrevious` each time we start a new message and after the entire input has been processed. This function inserts segments MOA (in group 15) and UNT in the message. If the message is valid (that is, if it complies with all the requirements for the INVOIC message), it is written to the output. If not, it is written to the logging file together with all the error messages.

1.4.5 RTE Program Source Code

The complete RTE source code is shown below. This program must be compiled with the `mktr` command to generate the actual translator. See the program comments for details.

```
!=====
!
!TradeXpress 4.0
! Translator from a sample inhouse file to INVOIC.2.912
!
!=====

message "/usr/edi/messages/UN-EDIFACT/91.2/invoic.msg" building

!=====
! For each line starting with " INVOICE#" there starts a new message.
! Previous message is completed first and then segments UNH, BGM and
! UNS are inserted to the new message.
!=====
```

```

line(1:"INVOICE#")
!-----
! Fill up and write the previous message if there was one.
!-----
nfCompletePrevious ()
!-----
! Prepare the UNH segment: the reference number is picked from the
! inhouse, all other fields are constants.
!-----

segment UNH
    e0062 := pick(1,9,14) ! M an..14 MESSAGE REFERENCE NUMBER
    eS009.0065 := "INVOIC" ! M an..6 Message type identifier
    eS009.0052 := "2" ! M an..3 Message type version number
    eS009.0054 := "912" ! M an..3 Message type release number
    eS009.0051 := "UN" ! M an..2 Controlling agency
endsegment

bIncomplete := TRUE

!-----
! Segment BGM (mandatory)
! The current system time is inserted in format YYMMDD to element
! C507.1.2380. The qualifier for time format is set accordingly
! to "101".
!-----
segment BGM
    eC002.1000 := pick(1,23,35)!C an..35 Document/message name
    eC507.1.2005 := "3"!M an..3 Date/time/period qualifier
    eC507.1.2380 := time("%y%m%d")
    !M an..35 Date/time/period
    eC507.1.2379 := "101"!M an..3 Date/time format qualifier
endsegment

!-----
! Segment UNS (mandatory)
!-----
segment UNS
    e0081 := "S"! M a1 SECTION IDENTIFICATION
endsegment
endline

!=====
! Lines starting with "LINE#" are line items to the
! current message. The only segment used in the group here is LIN.
! Most data is picked from the in-house file.
!=====
line(1:"LINE#")
!-----
! Segment LIN g7 (mandatory)

```

```

!-----
segment LIN g7
    e1233 := "1"! M an..3 RELATIONAL QUALIFIER
    e1082 := pick(1,6,8)! C n..6 LINE ITEM NUMBER
    eC511.1.7139 := pick(1,14,3)! M an..3 Item qualifier
    eC511.1.7140 := pick(1,17,35)! C an..35 Item number
    eC186.6063 := "1"! M an..3 Quantity qualifier
    eC186.6060 := pick(1,52,17)! M n..15 Quantity
    eC509.1.5125 := "INV"! M an..3 Price qualifier
    eC509.1.5118 := pick(1,69,17)
    !-----
    ! Add this line item to the total sum. Note that we have
    ! to convert the element values to numeric types before
    ! performing any arithmetic operations.
    !-----
    nTotal := nTotal + number(eC186.6060) * number(eC509.1.5118)
endsegment
endline
!=====
! This statement list is executed after the entire input has been
! consumed.
!=====

end
!-----
!Fill up and write the previous message if there was one.
!-----
nfCompletePrevious ()
log ("There were ", nMessages, " completed messages.", NL)
if nErroneous > 0 then
    log ("(Discarded ", nErroneous, " erroneous messages.)", NL)
endif
endend

!=====
! This function is called each time a new message is started to fill
! up and output the previous message.
!=====
function nfCompletePrevious ()
    if bIncomplete = FALSE then
        return 0
    endif

    !-----
    ! Segment MOA g15 (mandatory)
    !-----
    segment MOA g15
        e5007 := "3"! M an..3 MONETARY FUNCTION QUALIFIER
        eC516.1.5025 := "9"! M an..3 Monetary amount type qualifier

```

```

        eC516.1.5004 := build(nTotal:1.2)
        ! C n..18 Monetary amount
        eC516.1.6345 := "USD"! C an..3 Currency, coded
endsegment

!-----
! Segment UNT (mandatory)
!-----
segment UNT
    e0062 := eUNH.0062 !M an..14 MESSAGE REFERENCE NUMBER
    e0074 := build(SEGMENTS + 1)
    !M n..6 NUMBER OF SEGMENTS IN A MSG
endsegment

!-----
! Check the message.
! If the message is OK it is written to the output, otherwise to the log file.
!-----
if valid(MESSAGE) then
    log ("Message ", eUNH.0062, " completed.", NL)
    print (MESSAGE)
    nMessages++
else
    log ("Message ", eUNH.0062, " contained errors:", NL)
    log (MESSAGE)
    nErroneous++
endif

!-----
! Reset the variables.
!-----
bIncomplete := FALSE
nTotal := 0
endfunction

```

1.4.6 The Inhouse Data File

This sample inhouse file contains data for three messages. To show how an error situation is handled, the last message includes an intentional error in a numeric field.

INVOICE#123001	INVOICE FOR DISTRIBUTOR		
LINE#1	3	007234110 1	28.50
LINE#2	3	007234150 2	120.00
LINE#3	3	007234199 1	28.50
INVOICE#123002	INVOICE FOR SUPPLIER		
LINE#1	3	007234110 1	28.50
LINE#2	3	007234150 2	120.00
LINE#3	3	007234199 1	28.50
INVOICE#123003	INVOICE FOR SUPPLIER		

LINE#1	3	007234110 N	28.50
LINE#2	3	007234150 2	120.00
LINE#3	3	007234199 1	28.50

1.4.7 The Resulting Messages

The translator sends two complete messages to the output. For easier debugging, the messages are printed one segment per line. The modules that create the interchanges are responsible for removing extraneous line feeds.

```

UNH+123001+INVOIC:2:912:UN'
BGM+:::INVOICE FOR DISTRIBUTOR++3:930125:101'
LIN+1+1++3:007234110++1:1+INV:28.50'
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
MOA+3+9:297.00:USD'
UNT+8+123001'
UNH+123002+INVOIC:2:912:UN'
BGM+:::INVOICE FOR SUPPLIER++3:930125:101'
LIN+1+1++3:007234110++1:1+INV:28.50'
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
MOA+3+9:297.00:USD'
UNT+8+123002'

```

1.4.8 The Resulting Log File

All user-written logging statements appear in the logging file, together with the systemwritten warnings. Note the system's initial warning about the character "N" in the numeric field.

```

Message 123001    completed.
Message 123002    completed.
invoic: Line 85 (data line 10)
WARNING: String "N" contains non numeric
characters.Conversion yields 0.
Message 123003 contained errors:
UNH+123003+INVOIC:2:912:UN'
BGM+:::INVOICE FOR SUPPLIER++3:930125:101'
LIN+1+1++3:007234110++1:N+INV:28.50'
  ^ C186.6060: Invalid type NUMERIC field
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
MOA+3+9:268.50:USD'
UNT+8+123003'
There were 2 completed messages.
(Discarded 1 erroneous messages.)

```

1.5 Receiving an EDI Message

In this example, the translator receives the INVOIC.2.912 messages built in the previous example and converts the data to the same inhouse format that was used at the starting point. One exception is made: the message total is written to the header line to demonstrate the possibilities achieved with buffered output.

The received EDI message is traversed through segment by segment, starting from the beginning. Each user-supplied segment statement list is executed as the named segment is reached in the message. The data elements in the segment are usable as text constants with their own names (the name again preceded by a lower case "e"). You can reference elements in previous segments by stating the segment name before the element name. In this case a lower case "e" precedes the entire name.

```
segment UNT
    print (e0062, NL)      ! 0062 in UNT
    print (eUNH.0062, NL) ! 0062 in UNH
endsegment
```

1.5.1 Specifying the Message

The program states at the beginning which message it applies to:

```
message "UN-EDIFACT/91.2/invoic.msg" receiving
```

At this point we are receiving EDIFACT, so we define the direction as "receiving."

1.5.2 Buffered Output

When building the EDI messages, we were able to insert the segments in the message in any order, thus freeing up the order in which the data appeared in the inhouse file. The counterpart in the receiving direction is the concept of buffered output. The basic idea is that the output is written to a dynamic-sized matrix in any order and then flushed to the file. This makes it possible to insert data from later segments into the output file before data from earlier segments.

1.5.3 RTE Program Source Code

Here is the listing of the complete source code to generate the translator. Again, this source must be compiled with mktr before the translator can be used.

```
!=====
!
!TradeXpress 4.0
!Translator from UN/EDIFACT INVOIC.2.912 to sample inhouse.
!=====
message "/usr/edi/messages/UN-EDIFACT/91.2/invoic.msg" receiving
!-----
! For each new message write the text "INVOICE#"
! followed by the message reference number to the inhouse file.
!-----
segment UNH
    put(1,1,"INVOICE#")
```

```

    put(1,9,e0062)! M an..14 MESSAGE REFERENCE NUMBER
    !-----
    ! Start line items from line two relative to this first line.
    !-----
    nLINE := 2
endsegment
!-----
! Also the message name is written to the header line.
!-----

segment BGM
    put(1,23,eC002.1000)! C an..35 Document/message name
endsegment

!-----
! Line items are inserted to the buffer.
!-----

segment LIN g7
    put(nLINE, 1, "LINE#")
    put(nLINE, 6, e1082) ! C n..6 LINE ITEM NUMBER
    put(nLINE, 14, eC511.1.7139) ! M an..3 Item qualifier
    put(nLINE, 17, eC511.1.7140) ! C an..35 Item number
    put(nLINE, 52, eC186.6060:o17)! M n..15 Quantity
    put(nLINE, 69, eC509.1.5118:o17)
    !-----
    ! Add up the line total to the grand total of this message.
    ! We will do a cross check in the summary section.
    !-----
    nTotal := nTotal + number (eC186.6060) * number
    (eC509.1.5118)
    nLINE++
endsegment

!-----
! Check the total sum against the calculated total.
! Write the sum to the header line.
!-----

segment MOA g15
    if nTotal <> number (eC516.1.5004) then
        log ("The total sum (" , eC516.1.5004, ") for " , eUNH.0062, NL)
        log ("does not match line item sums (" , nTotal, ")", NL)
        put (1,58,"(Total unknown)")
    else
        put (1,58,build("(Total: " , number (eC516.1.5004):1.2, " USD)"))
    endif
endsegment

```

```
!-----
! Flush the buffer out to the output file and reset the line item total to zero.
! The arguments to flush specify the minimum and the maximum width
! for the resulting lines. Giving a value zero to both values the lines
! are written to the file exactly as they were in the buffer.
!-----
segment UNT
    nTotal := 0
    flush(0, 0, NL)
endsegment
```

1.5.4 The Original Messages

The EDIFACT messages that we started with are displayed below.

```
UNH+123001+INVOIC:2:912:UN'
BGM+:::INVOICE FOR DISTRIBUTOR++3:930125:101'
LIN+1+1++3:007234110++1:1+INV:28.50'
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
MOA+3+9:297.00:USD'
UNT+8+123001'
UNH+123002+INVOIC:2:912:UN'
BGM+:::INVOICE FOR SUPPLIER++3:930125:101'
LIN+1+1++3:007234110++1:1+INV:28.50'
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
MOA+3+9:297.00:USD'
UNT+8+123002'The
```

1.5.5 Resulting Data File

The translator produces a data file that is identical to the original data file used in building the messages in the previous example. The only exception is the insertion of the total sum at the end of the INVOICE line.

```
INVOICE#123001 INVOICE FOR DISTRIBUTOR (Total: 297.00 USD)
LINE#1      3    007234110    1    28.50
LINE#2      3    007234150    2    120.00
LINE#3      3    007234199    1    28.50
INVOICE#123002 INVOICE FOR SUPPLIER (Total: 297.00 USD)
LINE#1      3    007234110    1    28.50
LINE#2      3    007234150    2    120.00
LINE#3      3    007234199    2    28.50
```



Chapter 2 - Program structure

Overview

Statement Lists

Line Matching

Definitions

Functions

Comments

2.1 Overview

An RTE program consists of one or more statement lists, optional message and database definitions, and possible functions. Statement lists and functions are built from statements. The source program may also contain comments that document and clarify the implemented solutions. This chapter provides a full description of these main components in RTE. Statements and expressions are covered in subsequent chapters.

2.2 Statement Lists

Each statement list is supplied with a rule that is matched against the input line (or segment). The first matching rule causes the statement list to be executed. Line and segment statement lists cannot both appear in the same RTE program.

Line	The matching rule can be either a given text at a given position, the appearance of a given text anywhere on the line, or an absolute line number, or any of the above combined with the logical operators and, or and not.
segment	The matching rule is the name of the segment and its position in the message tree. The physical order of the statement lists in the program file determines the order in which the rules are compared against the input lines. Since the programs are data driven, the order of the statement lists does not determine the sequence in which they are executed, which depends entirely on the data.

Four special statement lists are provided by the system.

Begin	Executed before any lines (or segments) are read from the input. Must appear before all other statement lists in the program file.
Default	Executed if none of the line (or segment) statements is triggered by the input line (or segment). Must appear after all line (or segment) statement lists, but before any end statement lists.
End	Executed after the entire input has been processed. Must appear as the last statement lists in the program.
Function	Executed when this function is explicitly called.

```
database "description file" DBSYMBOL
message "description file" mode
begin
    !-----
    ! Statements here are executed before anything is read from the input.
    !-----
endbegin
```

```

line
    segment
        !-----
        ! Executed based on the rule
        !-----
    endsegment
endline

default
    !-----
    ! Statements here are executed if no line (or segment) statement list
    ! matched the input line.
    !-----
enddefault

end
    !-----
    ! Statements here are executed after the input has been processed.
    !-----
endend

function
    !-----
    ! Statements here are executed when this function is explicitly called.
    !-----
endfunction

```

2.3 Line Matching

The basic line matching rules -- text at a given position, text anywhere on the line, or a line number -- can be used alone, or they can be combined with the logical operators and, or, and not.

2.3.1 Text at a Given Position

This kind of matching allows the user to trigger a statement list if the input line has the specified text at an exact position. The text can be given as a constant string if it is known at compile time. It can also be a variable or any other text expression, which is evaluated at run time.

```

line (3:"TEXT")
line (2:tTRIGGER)

```

A special case is the matching at the end of the line. The position indicator is EOL.

```

line (EOL:"LAST")

```

2.3.2 Text Anywhere

If the text position does not matter, the matching statement is as follows.

```

line ("FIND ME")

```

2.3.3 Line Number

An absolute line number is specified with a single integer. This line number is matched against the actual line number in the input. The first line in the input is one (1).

```
line (3)
line (100)
```

2.3.4 Logical Operators

The basic rules can be combined with logical operators to form more complex matching rules.

```
line (3 and "FIND ME")
line ("FIND ME" or 3:"ME TOO")
line (not "")
line ("ONE" and "TWO")
```

The precedence of the logical operators from highest to lowest is:

- not
- and
- or

You can alter this precedence by using parentheses. The line matching rule:

```
line ("ONE" or "TWO" and "THREE")
```

reads, "Line contains either the text 'ONE' in any position or alternatively both 'TWO' and 'THREE' in any position." If instead you want to specify, "Line contains either 'ONE' or 'TWO' in any position and must contain 'THREE,'" you must write:

```
line (("ONE" or "TWO") and "THREE")
```

Note that if the back slash character (\) is to be matched, it must be doubled, because a single back slash character constitutes the escape character.

```
line ("^\\R1032\\")
```

The above example would match a line beginning with \R1032\

2.3.5 Regular Expression Matching

The basic matching rules provide for a wide range of possibilities, but you may sometimes need more advanced capabilities. Advanced matching rules can be implemented with regular expressions. These are widely used in Unix systems and novice users often find them difficult, but once you get familiar with them you will find them very powerful. Regular expressions are also used elsewhere in RTE, and their syntax is presented in "Appendix B: Regular Expressions", page 136.



Regular expressions are not available in the NT environment.

A regular expression is inserted between single quotation marks following the line keyword. No parentheses are allowed.

```
line 'regular expression'
```


For example, the matching rule

```
line '^([A-Z])+([A-Za-z])*'
```

specifies, "Line where the first character on the line is an upper case letter that may optionally be followed by any number of upper or lower case letters."

Here is an example of a case where the expression power of the basic rules is insufficient: we need to match a line that contains the word "TEXT" and nothing else. The rule

```
line (1:"TEXT" and EOL:"TEXT")
```

is a good attempt, but the line "TEXT-TEXT" (to give one example) would match this rule also. A regular expression can be used to solve the problem:

```
line '^TEXT$'
```

2.3.6 Multi Matching

An RTE program can be directed to execute all matching statement lists instead of executing only the first match. You can achieve this by compiling the source program with the a-option.

2.4 Definitions

Because RTE is a very high-level programming language, you do not have to define variables, worry about memory allocation and deallocation, or work with awkward file I/O functions.

There are, however, two types of definitions that you must make: message and database definitions. Both define an interface to external data.

2.4.1 Message Definition

Message definition is required when the RTE translator is used to build or receive EDI messages. A program can contain only one message definition.

The message definition states the name of the message description file and the intended usage. The usage is either building or receiving. The interpretation of the message description file depends on the keyword used in front of the file name. The keyword **message** implies that all governing message-level rules are in force and that the segments are handled as part of the message. The keyword **segments** imply that segments in the message description file are used independently, with no governing message-level rules applied to them.

```
message "UN-EDIFACT/91.2/INVOIC.msg" building
message "ANSIX12/810.msg" receiving
segments "UN-EDIFACT/91.2/UN-segments.msg" building
```



For more information on EDI translation, please refer to "Chapter 4 - EDI translation", page 49.

2.4.2 Database Definition

If an RTE program refers to EDIBASE databases, there must be a separate database definition for each database used. These definitions must appear before any statement lists. The syntax to be used is the following:

```
base "configuration_file" entry_identifiers, option
```

2.4.2.1 Example

```
base "syslog.cfg" LOG
base "syslog.cfg" LOGENTRY THISLOG DUPLICATE, autoflush off
```

If the configuration file name does not include the entire path from the root directory, it is assumed to be relative to a directory named \$HOME/database or, secondarily, \$EDIHOME/ database.

2.4.2.2 The possible database options are:

- **readonly:** if added this option states that the RTE instructions should not write into the database. Setting this option allows improving database performance.
- **autoflush on/off:** this option states whether the RTE is used in a autoflush mode. If autoflush is on, data are written to the database as they are assigned, if autoflush is off, the database writing operations are written only when flushed. By default this option is on.

The actual database instance used at run time does not have to exist at compile time; the database definition file is sufficient.

2.5 Functions

You can write your own functions to perform various tasks. The functions can receive the basic data types and arrays as parameters and they can return the basic data types as their values.

All user-defined functions must be located after all statement lists. The order of the functions is not relevant. A function located later in the code can be called from an earlier function.

User-defined function names start with one of the letters "t," "n," or "b," and always have a lower case "f" as the second letter. The first letter identifies the return type and the second letter identifies the symbol as a function name (in contrast to a text variable, for example). The third character must be an upper case letter. The remaining characters can be letters, digits, and underscores.

```
function tfGetTime ()
    return time ("%d.%m.%y")
endfunction

function nfSum (nFirst, nSecond)
    return nFirst + nSecond
endfunction

function bfEqual (tOne, tTwo)
    return (tOne = tTwo)
endfunction
```

Any passed parameters are local variables to the function. All other variables used in the function are global and can thus be referenced outside the function as well.

All text variables passed as function parameters should be treated as read-only variables and they should be used as a left side of an assignment operation.

The return value must be the same type as the function. This is checked for at compile time.

A function must include a return statement if the code that invokes it makes use of the value returned by the function. Examples include numeric functions that make use of the maths coprocessor. However, we recommend that you use a return statement in any case, to make the code more readable.



Do not use a global variable as return value, instead use local variable. If you do so, calling a function many times inside another function will set the same value for each parameter.

2.6 Comments

There are the following ways to write comments in the source code.

2.6.1 Source Code Documentation

An exclamation mark (!) starts a comment that extends to the end of the line. Comments can appear anywhere in the program. Commenting the source code is highly encouraged; a clearly commented program is easy to read and maintain.

2.6.2 Runtime Information

RTE provides a facility to retrieve information about the executable translator module. This information can be used to document the purpose, methods of use, required arguments, and other items for the translator.

Information comments are started with the percent sign (%) and extend to the end of the line like regular comments. These comments can be freely distributed in the source code, but when printed they are concatenated together.

After an RTE program has been compiled with mktr, this information can be retrieved by running the translator with the i-option.

```
%
% A sample information comment
%
begin
    !- - - - -
    ! Imagine there is some actual code here.
    !- - - - -
endbegin

$ mktr sample.rte
$ sample -i
A sample information comment
$
```



Chapter 3 - Program statements

Overview

Assignment Statements

Increment and Decrement Statements

Control Statements

Loop Statements

3.1 Overview

An RTE program consists of statement lists that contain statements. The statements can be assignments, increment and decrement statements, control statements, and function calls. Statements are separated by the newline character.

3.2 Assignment Statements

In assignment statements, variables of all types can be given new values. The variables can be system or user defined, database entries, or items in arrays. The allowed variables for all types are covered in detail below.

3.2.1 Text Assignments

On the left side, a text assignment can contain a user-defined text variable, a textual system variable (APPL_DEC_SEP, MSG_DEC_SEP, FILL_CHAR, RETURN_BUFFER, NL), an element in a text array, or a text or time field in a database or parameter. Note that inside a function, none of the function arguments can be used on the left side of an assignment. The right side of the assignment must be a text expression.

```
tText := "sample text"
taTexts[1] := tText
APPL_DEC_SEP := "."
MSG_DEC_SEP := tfGetSeparator ()
FILL_CHAR := pick (1,1,1)
RETURN_BUFFER := taTexts[2]
NL := "\n"
ENTRY.NAME := "myname"
ENTRY.EXPIRES := "31.12.1999"
pSAMPLE := ENTRY.NAME
```

Text constants can be concatenated as follows:

```
"str" "ing"
```

which becomes

```
"string"
```

The above-mentioned concatenation is useful with cpp macros, for example.

3.2.2 Numeric Assignments

On the left side, a numeric assignment can contain a user-defined numeric variable, a numeric system variable (LOGLEVEL), an element in a numeric array, or a numeric field in a database. The right side of the assignment must be a numeric expression.

```
nNumber := 3
naNumbers[1] := nNumber
naNumbers["text"] := nfGetValue ()
ENTRY.AGE := 45
```

```
LOGLEVEL := 1
```

3.2.3 Boolean Assignments

On the left side, a boolean assignment can contain a user-defined boolean variable, a Boolean system variable (PROCESS_ER-RONEOUS), or an element in a boolean array. The right side of the assignment must be a boolean expression.

```
bBoolean := TRUE
baBooleans[1] := FALSE
baBooleans[2] := nNumber > 2
PROCESS_ERRONEOUS := TRUE
```

3.2.4 File Assignments

A file assignment applies only to user-defined file variables. The right side of the assignment must be a text expression.

```
fFile := "/etc/hosts"
```

3.2.5 Database Assignments

A database assignment is used to link a database identifier to a physical record in the database. The left side must be a predefined database entry, and the right side must contain either "find" or a new function.

```
ENTRY := find ("mybase", NAME="myname")
ENTRY := new ("mybase")
```

3.2.6 Array Assignments

An array assignment is used to copy an array to another array.

```
taArray1 := taArray2
```

3.3 Increment and Decrement Statements

Increment and decrement statements are used with numeric variables only. An increment statement adds one to the value of the variable, and a decrement statement subtracts one from the value.

```
nNumber++
naNumbers[2]--
```

The effect shown above could also be achieved with the following assignment statements. In this case, the advantage of increment and decrement statements over an assignment statement is more compact code.

```
nNumber := nNumber + 1
naNumbers[2] := naNumbers[2] - 1
```

3.4 Control Statements

You can use the following control statements in the source code.

3.4.1 Conditional Statement

With a conditional statement, one or more statements can be executed conditionally.

```
if boolean expression then
    !- - - - -
    ! One or more statements
    !- - - - -
endif
```

The conditional statement may contain an optional "else" part.

```
if boolean expression then
    !- - - - -
    ! One or more statements
    !- - - - -
else
    !- - - - -
    ! One or more statements
    !- - - - -
endif

if bDoIt = TRUE then
    print ("I am doing it...", NL)
else
    print ("Did not do it", NL)
endif
```

3.4.2 Boolean Expressions

Boolean expressions can contain comparisons of boolean, text, and numeric expressions, alone or combined with the logical operators **and**, **or**, and **not**.

```
3 >= nNumber
LOGLEVEL > 1
naNumbers[2] = 12
nNumber < 18.24
13.67 <= naNumbers[3]
tText <> "comparison"
taArray[1] = tText
bBoolean = TRUE
baBooleans[4] <> FALSE
3 >= nNumber and tText <> "comparison"
nNumber < 18.24 or bBoolean = TRUE
not taArray[1] = tText
```

3.5 Loop Statements

RTE supports five different loop statements. A general loop is controlled with a Boolean expression. In addition, there are special loop statements to traverse arrays, lists, databases, and file system directories.

3.5.1 General Loop Statement

In the general form of a “while” statement, the loop is controlled with a boolean expression.

```
while booleanexpr do
    !- - - - -
    ! One or more statements
    !- - - - -
endwhile
```

The following program prints the members from one to ten.

```
nIndex := 1
while nIndex <= 10 do
    print (nIndex, NL)
    nIndex++
endwhile
```

3.5.2 Array Scanning

The array scanning loop statement makes it possible to scan arrays with no prior knowledge about the number of items or the naming of the related indexes. The loop scans the array indexes in alphabetical order. For each repetition, the index variable (tIndex) is assigned to the index of the array element.

```
while tIndex in taArray do
    !- - - - -
    ! One or more statements
    !- - - - -
endwhile

while tTmp in taNames do
    print (tTmp, " has value ", taNames[tTmp], NL)
endwhile
```


3.5.3 List Scanning

The list scanning loop statement simplifies the scanning of the lists created by a load command. The scanned list must be an element in an array that has been created with the load function.

```
while tValue in taArray[tIndex] do
    !- - - - -
    ! One or more statements.
    !
    ! Each item in the list is referenced as tValue.
    !- - - - -
endwhile

while tValue in taArray[tIndex] do
    print (tValue, NL)
endwhile
```



For details, see the "8.3.16 - load" section, on page 110.

The list scanning loop can also be implemented with a split function with the use of the general form of the loop statement to scan the generated array. The array scanning loop can also be used to scan the array. These are the only ways to scan a list when the list is not an element in an array created by the load function.

```
nCount := split (taArray[tIndex], taTmp, ":")
nIndex := 1

while nIndex <= nCount do
    print (taTmp[nIndex])
    nIndex++
endwhile

nCount := split (taArray[tIndex], taTmp, ":")

while tTmp in taTmp do
    print (taTmp[tTmp])
endwhile
```

3.5.4 Directory Scanning

A directory in the Unix file system can be scanned with the directory scanning loop statement.

The directory and the template for file names are stated in a text expression (tFilename). The template can contain wild characters in the actual file name, but not in the directory path. The scanning loop cannot process any subdirectories.

The running variable (fFile) contains the current file for each repetition. The order in which the files arrive depends on the physical order in which they are stored in the directory, and is thus undefined.

```
while fFile in tFilename do
    !- - - - -
    ! One or more statements
    !- - - - -
endwhile
```

This example counts the total size for all files that start with "MyFile" in the directory "/" tmp."

```
nTotal := 0

while fFile in "/tmp/MyFile*" do
    nTotal := nTotal + fFile.SIZE
endwhile

print ("My files occupy total ", nTotal, " bytes.", NL)
```

3.5.5 Database Scanning

```
while database entry in filtered database do
    !- - - - -
    ! One or more statements
    !- - - - -
endwhile
```

The following example shows how all entries that have been created over three days are removed from the database.

```
while entry in ("mybase", CREATED < "now - 3d") do
    remove (entry)
endwhile
```

3.5.6 Switch Statement

The switch statement can be used to make a selection between several text values.

```
switch tSelector
    case tOpt1:
        !- - - - -
        ! One or more statements
        !- - - - -
    case tOpt2:
        !- - - - -
        ! One or more statements
        !- - - - -
    default:
        !- - - - -
        ! The default branch is optional. One or more statements
        !- - - - -
endswitch
```

The statements after the matching case expression are executed. If none of the options matches the selection, then the statements in the optional default section are executed. If the default is used, it must be the last option in the switch statement.

3.5.7 Switch Expression

Switch expressions are related to the switch statement. The switch expression is type text and its syntax is as follows:

```
switch (tSelect, tOpt1:tRes1, tOpt2:tRes2, default:tResDef)\
```

As its value, the expression gets the result that was paired with the option which is equal to the first argument (tSelect). If none of the options matches the selector, then the value of the expression is EMPTY.

The following example shows a basic switch expression.

```
tCurrency := eC516.1.6345 ! C an..3 Currency, coded
print ( "The currency used for this transaction was ", switch(tCurrency, "USD":"the
Greenback", "FIM":"Finnish Marks", default:"other than FIM or USD"))
```

3.5.8 Switch Statement with Coprocesses

The switch statement can be used together with the TradeXpress coprocess library to provide an easy way of matching input data from coprocesses.

```
switch expect <PID> <time-out>
  case <string>:
    <RTE statements>
    ...
endswitch
```

For a more comprehensive explanation of the coprocess library and the switch statement, see “9.6 - Appendix F: Advanced Utilities”, on page 150



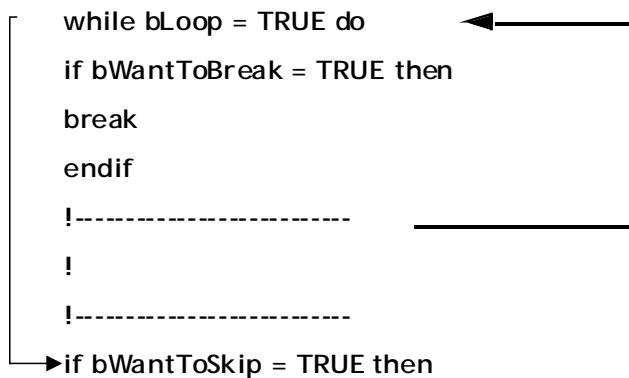
The switch statement with coprocesses is not available in the Windows environment

3.5.9 Break Statement

The break statement causes the termination of the innermost enclosing loop. This action is illustrated in the example below.

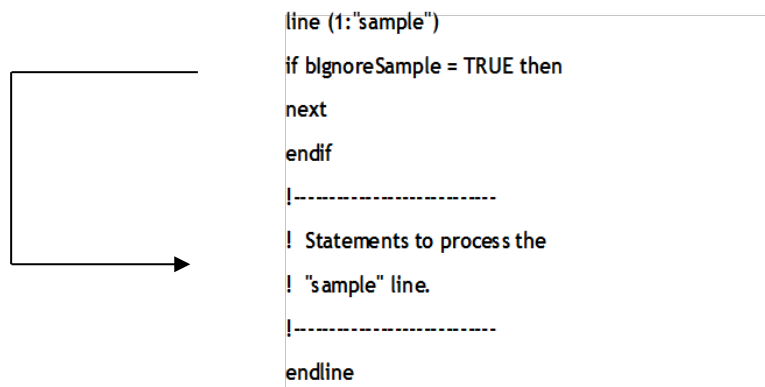
3.5.10 Continue Statement

The continue statement causes the execution to skip the remaining statements of the innermost enclosing loop. The execution continues from the first statement if the looping does not terminate in the comparison.



3.5.11 Next Statement

The next statement can only be used within the line statement or segment statement lists. It causes the execution to skip the rest of the current statement list and causes new input to be read.



3.5.12 Statement

The statement can only be used in programs that receive EDI messages. It causes the program to ignore the remaining segments in the current message and a new message to be read in.

```

segment UNH
  if valid (MESSAGE) = FALSE then

  endif
endsegment
  
```

3.5.13 RTE exceptions

The RTE language, as any high level language, offers a significant of abstractions in comparison with low-level operations such as writing /reading files. One of the consequences of this abstraction is that it is difficult, or even impossible to handle low-level errors.

In standard mode, these errors trigger, according to their severity, either a warning, either a fatal error. Warnings may be ignored or cause the program's termination, according to the log level. Fatal errors cause consistently the program's termination. This basic mechanism does not to tolerate unexpected errors, and requires to anticipate at best all error cases, which is almost impossible and

involves an excess of tests into the program. Handling exceptions is a simple and standard means to deal with this kind of errors.

3.5.13.1 Principle

An exception handling is declared by using the try ...catch...finally ...endtry instructions into the program's lines.


try	begins a protected section
catch	states the end of a protected section, and the start of exceptions processing
finally	states the end of a protected section (if not done previously with the catch instruction) and the start of an unconditional processing
endtry	states the end of an exception handling

The program located after the try instruction is executed as if no exception handler existed. If when the program is running an error occurs, then, the standard sequence is interrupted and the program goes to the first line preceding the catch instruction. Then the program is executed until the end. If no error occurs during the execution of the bloc located between the try and the catch instruction, then the catch bloc is ignored and the program goes to the finally bloc. The protection ends with the catch or finally instruction. If an exception occurs and no processing has been previsted for this exception, then the exception is spread after the endtry instruction (which leads to the program termination, unless this occurs in a higher level protected section)

The exception handling may be conditional, i.e. is executed only for some exception classes.

An additional instruction **throw()** allows to trigger its own exceptions which may be then processed with an exception handler.

3.5.13.2 Syntaxe

try	begins a protected section
catch [exp1,[exp2,..[exp n]...]]]	ends a protected section, begins an exception processing, and cancel the exception if the exception class is as argument in the whole expressions. exp1...expn are character strings (or any instruction/ function returning a character string). The catch instruction is optional, but it can be multiple in aprotected section.
finally	ends the preceding section and begins the unconditional processing; does not cancel the current exception. The finally instruction is optional. Note: For now, the finally section is not always be executed. Most notable cases are : <ul style="list-style-type: none"> • when you return, break, exit out in any section • when you decide to throw an exception in a catch statement. This behaviour may change in future versions
 You need to have at least one catch instruction or one finally instruction in an exception handle	

endtry	ends the exception handler bloc
EXCEPTION_CLASS, EXCEPTION_REASON, EXCEPTION_CODE, EXCEPTION_MESSAGE	allow respectively to retrieve the exception class, its cause, its code and its associated message. Code is in numeric format Other fields are text values
throw([class[, reason[, reasoncode, [message]]]])	triggers an exception. When parameters are missing, allows to spread the exception
bfPrintExceptionStack()	prints the exceptions stack in the standard output



You can interleave as many exceptions handler as necessary

3.5.13.3 Use

Example 1: protecting a calling to a number function which allows to ensure the running of a program whatever the log level is, and to convert it into a string non-convertible to -1 (as standard, number would trigger a warning and return 0)

```
try
    nNum := number(tNumber)
catch
    nNum := -1
endtry
```

Example 2: intercepting of all warnings excluding the fatal errors (or any other exception classes)

```
try
    nNum1 := number(tNumber)
    nNum2 := nNum3 / nNum4

    catch "general"

        switch EXCEPTION_CODE
            case 1 : nNum1 := -1
                    log(EXCEPTION_MESSAGE, " Valeur non numérique remplacée par -1",
NL)
            default : throw()
        endswitch
    endtry
```

Example 3: interleaving handlers

```
try
  try
    nNum1 := number(tNumber)
    nNum2 := nNum3 / nNum4

    catch "general"
      switch EXCEPTION_CODE
        case 1 : nNum1 := -1
        default : throw()
      endswitch
    endtry
  catch
    print(« Ce programme a provoqué une erreur non récupérable », NL)
  finally
    print(« Merci d'avoir utilisé ce programme », NL)
  endtry
```

3.5.13.4 RTE internal exceptions

All internal exceptions are "general" class exceptions

- **Warning:** class exception "general", reason « warning », code=1 ; triggered if internal non-blocking error (eg: conversion into numeric of a non-convertible string), if log level higher than 2.
- **Fatal:** class exception "general", reason « fatal », code=2 ; triggered if severe internal (eg: segmentation fault)

If the exception is protected, the message may be retrieved on the EXCEPTION_MESSAGE field.

3.5.13.5 Limitations

Handling exceptions should not be used into programs implementing multi-threading (which is not the case into RTE standard programs).

Handling exceptions does not allow to correct a failing program, especially in case of serious error. we advise you against intercepting an exception in order to continue the program as if no error had occurred. On the other hand, handling exceptions may be useful to cleanly end a program by attempting to close started transactions, freeing connections...

A section **try ... endtry** should be included into the same instructions bloc and cannot be used as main bloc at the same level as **begin ... endbegin** or **line ... endline**.

3.5.14 Inline C Block Statement

The inline C block statement provides access to the description area of an RTE program. Users can use inline C block statements to define their own C functions and data structures.

Syntax:

```
inline<nl>
    <C source code>
endinline<nl>
```

For a more comprehensive explanation of the inline statement, see “9.6 - Appendix F: Advanced Utilities”, on page 150.

3.5.15 Inline C Statement

The inline C statement provides access to the RTE program user statement area, allowing users to define their own C statements, such as function calls and data structures. Inline C statements are also useful when a user attaches API functions such as database access directly to an RTE program.

Syntax:

```
inline "<C source code %N >",<macro>, %1, ..., %n<nl>
```

where %N represents a replaceable, ordinal macro variable. These variables can be located anywhere within the C source code. The variables obtain values from the argument list corresponding to the order in the list. There is no limit to the number of variables.

For a more comprehensive explanation of the inline statement, see “9.6 - Appendix F: Advanced Utilities”, on page 150.



Chapter 4 - EDI translation

Overview

Operating Principle

Specifying the Message

Naming the Elements

Building Messages

Independent Segments

Receiving EDI Messages

Receiving Direction Error Handling

4.1 Overview

The RTE EDI translator module can handle various EDI syntaxes. The most widely used syntax is UN/EDIFACT (United Nations Rules for Electronic Data Interchange for Accounting, Commerce and Transport). Other supported syntaxes include ANSI ASC X12, TRADACOMS, VDA, and SPEC2000.

4.2 Operating Principle

The translator is based on data segments and their elements, which can be either simple data elements or composite data elements that consist of two or more component data elements. The highest level in the hierarchy is the message. A message definition consists of a list of segments and their status and repetition information. The segments in a message can be gathered into groups, each with its own status and repetition information. For more details on describing messages, see “9.3 - Appendix C: Message Storage Format”, on page 139

The status of a segment in a message can be mandatory, conditional, or floating, which also implies conditional status.

One RTE program works for one message in one direction. However, there are some peculiarities in the case of syntax to syntax translators. For more information, refer to the “Appendix H: XML to XML translator”, on page 189.

4.3 Specifying the Message

The message definition file and the working direction are specified at the beginning of the program, before any statement lists.

```
message "EDIFACT/UN/91.2/invoic.msg" receiving
message "EDIFACT/UN/91.1/orders.msg" building
segments "EDIFACT/UN/91.1/ifcsum.msg" building
```

The message definition file is an ordinary ASCII file that contains all the included segments and grouping information. For more information, see “9.3 - Appendix C: Message Storage Format” on page 139

The keyword (message or segments) before the definition file name specifies how the message definition is to be interpreted. The keyword (building or receiving) after the file name specifies the direction in which the program works.

Message	The definition file is interpreted to contain one message with all specified repetition and grouping information.
Building	The resulting program collects the segments into a message that is validated and printed as one unit.
Receiving	The resulting program reads one message at a time from input and hands the segments one at a time to user-written segment statement lists.

Segments	The definition file is interpreted to contain independent segments with no governing message structure. None of the repetition or grouping information is applicable. The description file must follow the same rules as when used for a message (that is, segment status and repetition information must be present for correct presentation syntax, even though they are not used).
Building	The resulting program writes individual segments to the output.
Receiving	The resulting program reads input one segment at a time. These segments are processed in the user-written segment statement lists.

4.4 Naming the Elements

All data elements are referenced by a unique name, which is always preceded by a lower case "e." Elements are always handled within an enclosing segment building statement (when building a message), or within a segment statement list (when receiving a message). Thus each segment has a unique name.

Simple data elements that appear only once in a segment are written as shown below.

e1445

A simple data element that appears more than once in a segment will be followed by a period and then an ordinal number.

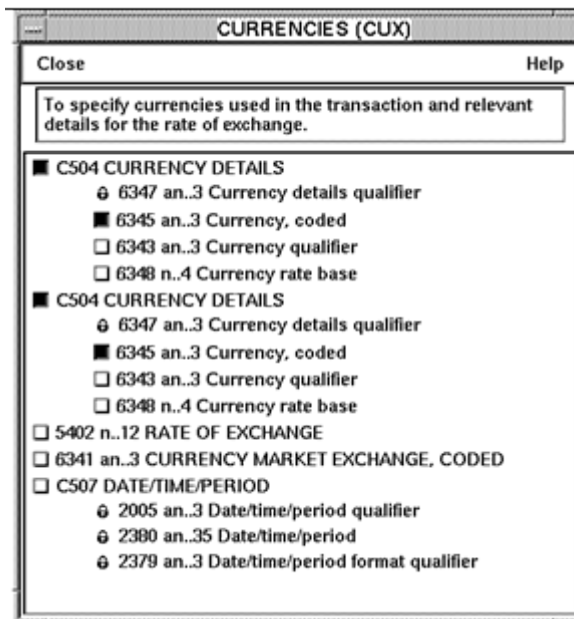
e1338.1

e1338.2

Component data elements are named so that the governing composite name comes first, followed by a period and the name of the data element.

eC507.2005

For a component data element that appears more than once in a segment, the ordinal number is specified immediately after the name of the composite. Again, all parts of the name are separated by periods.



CURRENCIES (CUX)

Close Help

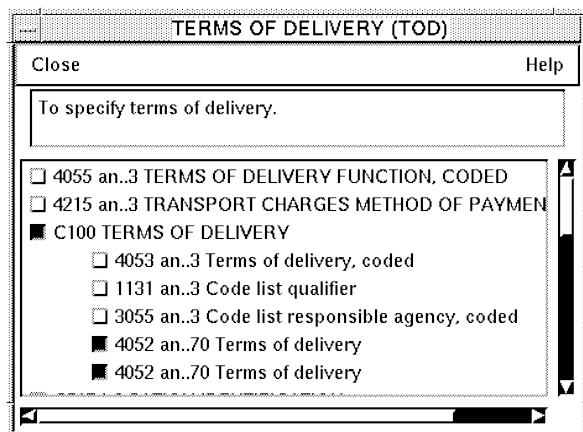
To specify currencies used in the transaction and relevant details for the rate of exchange.

- ☒ **C504 CURRENCY DETAILS**
 - ☐ 6347 an..3 Currency details qualifier
 - ☒ 6345 an..3 Currency, coded
 - ☐ 6343 an..3 Currency qualifier
 - ☐ 6348 n..4 Currency rate base
- ☒ **C504 CURRENCY DETAILS**
 - ☐ 6347 an..3 Currency details qualifier
 - ☒ 6345 an..3 Currency, coded
 - ☐ 6343 an..3 Currency qualifier
 - ☐ 6348 n..4 Currency rate base
- ☐ 5402 n..12 RATE OF EXCHANGE
- ☐ 6341 an..3 CURRENCY MARKET EXCHANGE, CODED
- ☐ **C507 DATE/TIME/PERIOD**
 - ☐ 2005 an..3 Date/time/period qualifier
 - ☐ 2380 an..35 Date/time/period
 - ☐ 2379 an..3 Date/time/period format qualifier

Here is a list of references to the segment CUX elements shown in the previous figure:

eC504.1.6347
eC504.1.6345
eC504.2.6347
eC504.2.6345

If a component data element appears more than once in a composite, it will be followed by a period and an ordinal number.



TERMS OF DELIVERY (TOD)

Close Help

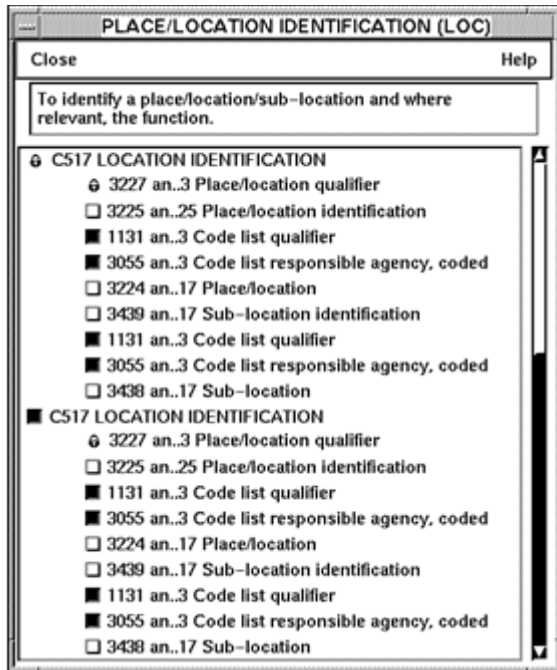
To specify terms of delivery.

- ☐ 4055 an..3 TERMS OF DELIVERY FUNCTION, CODED
- ☐ 4215 an..3 TRANSPORT CHARGES METHOD OF PAYMENT
- ☒ **C100 TERMS OF DELIVERY**
 - ☐ 4053 an..3 Terms of delivery, coded
 - ☐ 1131 an..3 Code list qualifier
 - ☐ 3055 an..3 Code list responsible agency, coded
 - ☒ 4052 an..70 Terms of delivery
 - ☒ 4052 an..70 Terms of delivery

Here is a list of references to the TOD segment elements shown in the previous figure:

eC100.4052.1
eC100.4052.2

Finally, here is the rule for all component data elements that appear more than once in a composite which in turn appears in the segment more than once: the ordinal number for the composite comes between the names, and the ordinal number for the component inside the composite comes last.



Here is a list of references to the previous segment LOC elements:

eC517.1.1131.1
eC517.1.3055.1
eC517.1.1131.2
eC517.1.3055.2
eC517.2.1131.1
eC517.2.3055.1
eC517.2.1131.2
eC517.2.3055.2

4.5 Building Messages

Messages are built from the application data. This data may come to the program as input, it may be extracted directly from a database, it may be totally or partially constant, or any combination of the above. A message building RTE program is a description of how a set of application data is converted to a given message structure. The complexity of this program depends entirely on the complexity of the inhouse data: it may contain only straightforward assignments, or it may be fully loaded with conditional statements, calls to external programs, and databases.

Data is inserted in the message tree one segment at a time with the segment statements. After all segments have been inserted, the message syntax can be checked. Depending on the validity of the message, the user can choose to ignore the message or send it to output.

The segments can be inserted in the message in any order.

4.5.1 Segment Building Statement

The segment statement can be viewed as a work area where the elements are assigned their values. The segment statement starts with the keyword `segment`, followed by the segment name and possible

location specification. You can also specify repetition information for each enclosing group (see "4.5.5 - Specifying the Segment Location" on page 57).

The segment building statements can contain a selection of any other available statements, excluding another segment building statement. At its simplest, the segment building statement is a list of assignment statements:

```
segment UNH
    e0062 := build (nMessages)
    eS009.0065 := "INVOIC"
    eS009.0052 := "1"
    eS009.0054 := "911"
    eS009.0051 := "UN"
endsegment
```



Message constants such as name and version can be obtained through the use of message variables such as mMESSAGE, mVERSION, mRELEASE, and mAGENCY

At the end of the statement, the segment is inserted in the message. It cannot be changed later in the program. The data in the elements is available for reference up to the point in the program where the next segment statement for the same segment is entered.

Here is a sample segment building statement that contains a conditional statement:

```
segment BGM
    eC002.1001 := pick (1, 1, 3)
    if eC002.1001 = "AAA" then
        eC002.1131 := "BBB"
    else
        eC002.1131 := "CCC"
    endif
    eC002.3055 := pick (1, 4, EOL)
endsegment
```

A segment building statement can also be part of a conditional statement:

```
if bIncludeFreeText = TRUE then
    segment FTX
        e4451 := "100"
        eC108.4440.1 := "Free form text"
    endsegment
endif
```

4.5.2 Element Assignments

Element values are given in assignment statements. All element assignment statements must be enclosed in the governing segment statement, and assignments can be made only to elements that belong to the current segment.

Elements are always type text in RTE. The EDI related types (such as alphanumeric, alpha, and numeric) specify the data representation for the element (that is, which characters are allowed). These two different concepts are explained below.

EDIFACT element e1082 is numeric as far as content is concerned, but since all elements are type text in RTE, the line below would result in a compiler error.

```
e1082 := 3C n..6 Line item number
```

The RTE compiler is satisfied with the following assignment, since the element is assigned type text. However, the EDI translator module will report an "invalid type" error at run time, because "N" is not a number.

```
e1082 := "N"C n..6 Line item number
```

The following assignment satisfies both the compiler and the EDI translator module. The right side of the assignment is type text that represents numeric data.

```
e1082 := "3"! C n..6 Line item number
```

If you have a value in a numeric variable that needs to be assigned to an element, you can use the build function to make the conversion from numeric type to type text.

```
e1082 := build (nNum)! C n..6 Line item number
```

Element values can be assigned in free order within a segment statement. The following segment statement contains the elements for the UNH segment, but in reverse order compared to the example above. The result is exactly the same.

```
segment UNH
    eS009.0051 := "UN"
    eS009.0054 := "911"
    eS009.0052 := "1"
    eS009.0065 := "INVOIC"
    e0062 := build (nMessages)
endsegment
```

4.5.3 Packing Data in Repeating Elements

By default, the TradeXpress translator packs repeating elements and components of the composite element. This default cannot be changed through the user interface. The changes must be made in \$HOME/.tclrc.

When a segment contains repeating elements, the translator module packs the data to the beginning of the repeating set. The following example shows how a CST segment is built from five separate lines of input. Each input line provides information for one composite data element, C246.

CUSTOMS STATUS OF GOODS (CST)

Close Help

To specify goods in terms of customs identities, status and intended use.

- 1496 n..5 GOODS ITEM NUMBER
- C246 CUSTOMS IDENTITY CODES
 - ⊖ 7361 an..18 Customs code identification
 - 1131 an..3 Code list qualifier
 - 3055 an..3 Code list responsible agency, coded
- C246 CUSTOMS IDENTITY CODES
 - ⊖ 7361 an..18 Customs code identification
 - 1131 an..3 Code list qualifier
 - 3055 an..3 Code list responsible agency, coded
- C246 CUSTOMS IDENTITY CODES
 - ⊖ 7361 an..18 Customs code identification
 - 1131 an..3 Code list qualifier
 - 3055 an..3 Code list responsible agency, coded
- C246 CUSTOMS IDENTITY CODES
 - ⊖ 7361 an..18 Customs code identification
 - 1131 an..3 Code list qualifier
 - 3055 an..3 Code list responsible agency, coded
- C246 CUSTOMS IDENTITY CODES
 - ⊖ 7361 an..18 Customs code identification
 - 1131 an..3 Code list qualifier
 - 3055 an..3 Code list responsible agency, coded

```
segment CST
  e1496 := "1"
  eC246.1.7361 := pick (1,1,18)
  eC246.1.3055 := pick (1,19,3)
  if eC246.1.3055 != EMPTY then eC246.1.1131 := "101"
  endif

  eC246.2.7361 := pick (2,1,18)
  eC246.2.3055 := pick (2,19,3)
  if eC246.2.3055 != EMPTY then eC246.2.1131 := "25"
  endif

  eC246.3.7361 := pick (3,1,18)
  eC246.3.3055 := pick (3,19,3)
  if eC246.3.3055 != EMPTY then eC246.3.1131 := "38"
  endif

  eC246.4.7361 := pick (4,1,18)
  eC246.4.3055 := pick (4,19,3)
  if eC246.4.3055 != EMPTY then eC246.4.1131 := "104"
  endif

  eC246.5.7361 := pick (5,1,18)
  eC246.5.3055 := pick (5,19,3)
  if eC246.5.3055 != EMPTY then eC246.5.1131 := "110"
  endif
endsegment
```


By way of example, here is some sample inhouse data:

```
CODE1 AA
    (empty line)
    (empty line)
CODE4 AB
CODE5 AE
```

The segment will contain the following data. The second and third composites do not receive any values and are left empty. The translator module packs the composites so that the empty ones come last, which means that no extra separators are needed.

```
CST+1+CODE1:101:AA+CODE4:104:AB+CODE5:110:AE '
```

4.5.4 Decimal Separator Conversion

General case

The character set used in an EDI message may have been configured to contain a different decimal separator from the one used in the application data file. RTE provides for automatic decimal separator conversion for numeric data elements. The application decimal separator is stored in the variable `APPL_DEC_SEP`, and the one used in the EDI message is stored in `MSG_DEC_SEP`.

In EDI to EDI translation

To allow different decimal separator for receiving and building in EDI to EDI translator, the `-n` option is set by the router if needed. This option sets the incoming separator for the RTE translator.

If it is set, only the given decimal separator is allowed, otherwise validation is made according the charset decimal separators.

If you don't need this fonctionnality and don't want to recompile all the RTE you may have had already, you can set the environment variable `RECV_USE_MSG_DEC_SEP` to "1".

4.5.5 Specifying the Segment Location

Each segment in the message tree has a unique identification by which it can be located in the message hierarchy. The identification is the segment name alone for segments that are not enclosed in any group.

- segment UNH
- segment BGM
- segment UNT

If the segment is enclosed in a group, its name is the segment name plus the names of all the enclosing groups. The list below contains all the CUX segments that are included in the UN/ EDIFACT message CUSDEC.2.912.

```
segment CUX g6
segment CUX g7, g8
segment CUX g7, g14
segment CUX g7, g15, g16
segment CUX g7, g15, g22
```

```
segment CUX g23, g26  
segment CUX g23, g34, g35
```

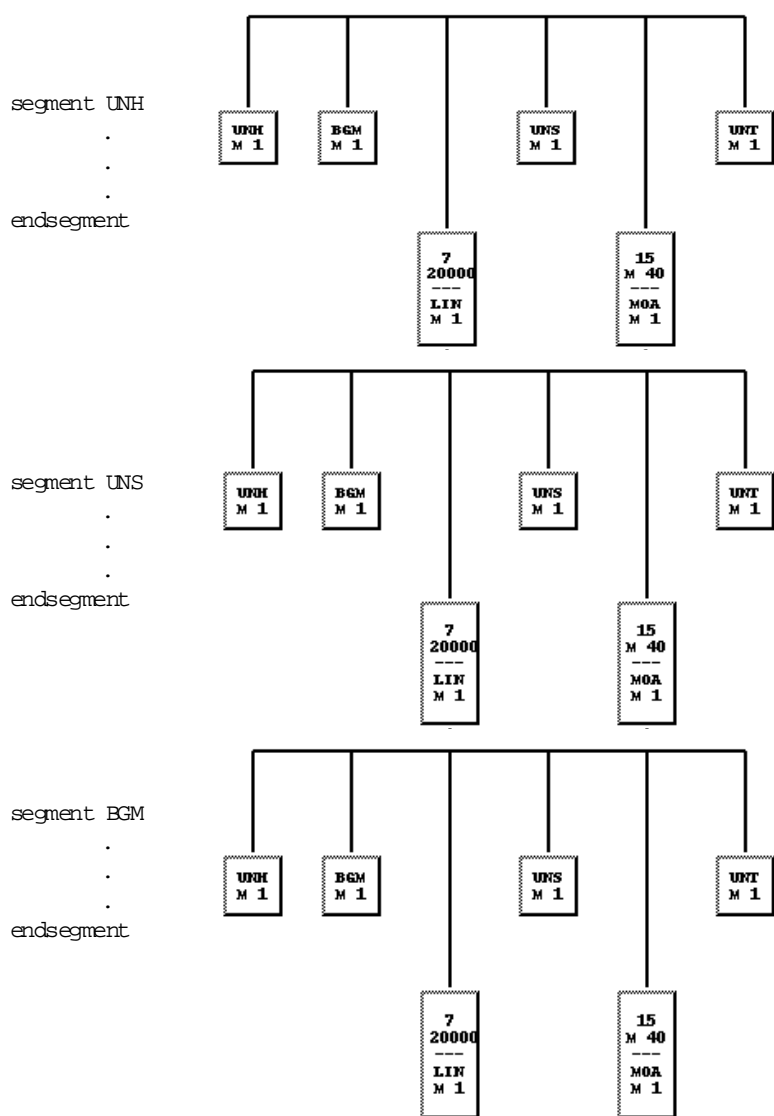
The group identifiers are separated by commas, and all group identifiers must be included. If a segment appears more than once in the same group path or outside any groups, an ordinal number in front of the segment name is used to distinguish the segments. This example is also taken from the UN/EDIFACT CUSDEC.2.912 message.

```
segment 1.UNS  
    e0081 := "D"  
endsegment  
  
segment 2.UNS  
    e0081 := "S"  
endsegment
```

The ordinal number can be left out for the first segment. Thus this:

```
segment UNS  
    e0081 := "D"  
endsegment
```

is the same as the first UNS specification above.



4.5.6 Specifying the Segment Repetition

Depending on the structure of the inhouse data, it may be necessary to insert segments in mixed order. The following examples show how repetition can be specified for the DTM segment in group 23 in the UN/EDIFACT CUSDEC.2.912 message.

segment DTM g23

The DTM segment is inserted in the next available slot under group 23.

segment DTM=2 g23

The DTM segment is inserted in the second slot (out of five) under group 23.

segment DTM g23=2

The DTM segment is inserted in the next available slot under group 23, when group 23 is in its second repetition.

```
segment DTM=3 g23=3
```

The DTM segment is inserted in the third slot (out of five) under group 23, when group 23 is in its third repetition.

4.5.7 Checking the Syntax

The segments created with the segment statement lists form the message structure in the system's memory. The validity of the message can be checked with the general purpose function **valid** before the message is written to the output. If the message contains no EDI errors, the function returns the value TRUE; otherwise it returns the value FALSE.

You can print the message with the print function by specifying the keyword MESSAGE as the only argument to print.

The message will be removed from memory immediately after it has been printed. To keep the message in memory after printing, you must compile the RTE source code with the -m option

```
if valid (MESSAGE) = TRUE then
    print (MESSAGE)
else
    !- - - - -
    ! User specified alarms, counters
    ! for erroneous messages go here.
    !- - - - -
    log (MESSAGE)
endif
```

If the message contains errors, the error messages are included at appropriate places in the printed message.

```
UNH+123001+INVOIC:2:912:UN'
BGM+:::INVOICE FOR DISTRIBUTOR++3:930301:101'
LIN++1++3:007234110++1:1+INV:28.50'
    ^ 1233: Missing mandatory element (11)
LIN+1+2++3:007234150++1:2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
UNT+8+123001'
Too few groups 15 (0), 1 minimum
```

4.5.8 Segment Counter

A numeric read-only variable, SEGMENTS, contains the number of segments that have been inserted in a message so far. Most EDI grammars require the segment count in the trailer segment.

Since the RTE EDI translator module is completely independent of any grammar, it does not know which data element should contain the segment count. This means it is the user's responsibility to insert the count in the appropriate segment. The example below shows how this is done in an EDIFACT message in which the trailer segment is UNT. The segment count in UNT includes the UNT itself, and since SEGMENTS gives the number of segments inserted so far, we must add one to it to get the grand total.

```
segment UNT
    e0074 := build (SEGMENTS + 1)
    e0062 := eUNH.0062
endsegment
```



For a complete but compact example of building and sending EDI messages, see “Chapter 1 - Tutorial”, on page 15.

4.6 Independent Segments

4.6.1 Independent Segments

If the message definition keyword is `segments` instead of `message`, then the segments in the message description are treated independently from the message structure. Each segment statement list causes the segment to be written immediately to the output. If the segment contains errors, there will be no output. This condition can be checked for after the segment statement list.

```
segment UNH
    e0062 := build (nMessages)
    eS009.0065 := "INVOIC"
    eS009.0052 := "1"
    eS009.0054 := "911"
    eS009.0051 := "UN"
endsegment

if valid (SEGMENT) = FALSE then
    log ("Segment UNH contained errors.", NL)
    edierrordump (SEGMENT)
endif
```



Note that the segment statement list shown above does not output anything if an error occurs. The content of the segment, with appropriate error messages, can be printed with the `edierrordump` function, which prints by default to the logging stream instead of to the output.

4.7 Receiving EDI Messages

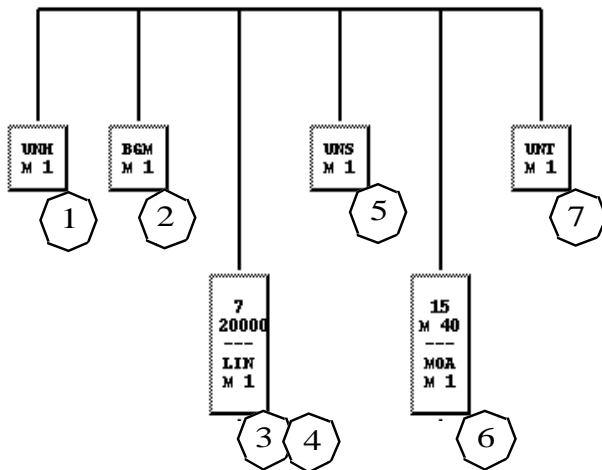
The translator reads messages one at a time from the input and checks them for correct syntax. If the message is valid according to the message description being used, it is then delivered one segment at a time to be processed by the user-written statements.

An RTE program for processing incoming messages consists of segment statement lists that are triggered by incoming segments. The elements can be referenced as if they were constants of type text. They cannot be assigned any values.

The segments come in the traversal order of the message tree. The physical order in which the statement lists appear in the control file does not affect the processing order, but for better maintainability, we recommend that the statement lists be written to reflect the order of the message.

The figure below shows the traversal order for a small subset of UN/EDIFACT INVOIC.2.912 where there are two line items.

The segment statement list starts with the keyword segment, which is followed by the possible location information. The segment statement list looks very much like the segment building statement.



```

line (1:"HEADER")
  segment UNH
    !- - - - -
    ! This is a segment building statement and
    ! must be contained in a statement list.
    !
    ! Elements are assigned values.
    !- - - - -
    e0062 := "MSGREF"
  endsegment
endline

segment UNH
  !- - - - -
  ! This is a segment statement list used in the receiving direction.
  ! It can contain any other statements except segment building statements.
  !
  ! Elements values are printed to the application data file.
  !- - - - -
  print (e0062, NL)
endsegment

```

The segment statement list is executed separately for each occurrence of the identified segment. All elements belonging to the current segment can be referenced by their names (see "4.4 - Naming the Elements" on page 51), but elements in earlier segments must also contain the segment name.

```

segment UNT

  print (eBGM.C002.1000, NL)
  ! C an..35 Document/Message name

```

```
print (eUNH.0062, NL)
! M an..14 MESSAGE REFERENCE NUMBER

print (e0074, NL)
! M n..6 NUM OF SEGMENTS

endsegment
```

Only the latest occurrence of any received segment is available at any given time. For example, you cannot reference elements in previous LIN segments while processing the current LIN. If the data in given elements must be carried along, you can use text variables or text tables.



For a complete but compact example on receiving EDI messages, see “Chapter 1 -Tutorial” on page 15.

4.7.1 Decimal Separators

All decimal separators in numeric data elements are converted to the inhouse decimal separator before the segments are delivered to the user statements. For more information, see “4.5.4 - Decimal Separator Conversion” on page 57.

4.8 Receiving Direction Error Handling

When erroneous messages are received, there are three ways to handle them:

- ignore erroneous messages completely
- handle erroneous messages in a separate routine
- handle erroneous messages in segment statement lists.

RTE provides two built-in functions, `edierrorlist` and `edierrordump`, that can be used to diagnose erroneous messages. The statement skips over the remaining segments in the current message.

4.8.1 Ignoring Erroneous Messages

The default action is to ignore incoming erroneous messages completely. If the translator module detects an erroneous message, it skips it and reads in the next message. The segment statement lists are activated only by segments in valid messages.

4.8.2 Using a Separate Error Routine

You may want to get information about erroneous messages; their number, for example. If you include a function called `bfIncomingError` in a program, all erroneous messages will be directed to that routine. Below is an example of a `bfIncomingError` routine:

```
function bfIncomingError ()
    edierrordump (MESSAGE)
    exit (1
endfunction
```



The `bfIncomingError` function must be the first user function that is defined.

`bfIncomingError` can provide for tasks such as counting, logging, and reacting to erroneous messages, but it does not give you access to the individual data segments. Therefore, this method cannot be used when specific information about the message (for example, the message identifier) must be retrieved from the segments.

4.8.3 Processing Erroneous Messages in Statement Lists

To gain access to the individual segments, you must process the erroneous messages in the segment statement lists. The translator module will deliver the segments to the segment statement lists when the boolean variable `PROCESS_ERRONEOUS` is `TRUE`. The default value for this variable is `FALSE`. If `PROCESS_ERRONEOUS` is `TRUE`, it will override the `bfIncomingError` function, which will no longer be called.

With this method, it is up to the user to detect and handle all erroneous messages. Either the validity of the entire message can be checked in the first incoming segment (for example, `UNH` for `EDIFACT`), or you can check the validity for each individual segment. These are just suggestions; you can adopt any strategy that fits your needs and that can be implemented with the tools you have available.

4.8.4 Preparing for the EDIFACT CONTRL Message

The following example shows how data can be collected for the `EDIFACT CONTRL` message. The `edierrorlist` function provides all the error data for the segments, but in order to get to the message header information, we need to process the `UNH` segment.

```
!- - - - -
! allow user's error handling
!- - - - -

begin
    PROCESS_ERRONEOUS:=TRUE
endbegin

!- - - - -
! Verify the validity of the message in the very first segment.
!- - - - -

segment UNH
    if valid (MESSAGE) = FALSE then
        bfCONTRLSupport ()
    endif
    !- - - - -
    ! Normal processing for UNH
    !- - - - -
endsegment

!- - - - -
! All other segment lists...
!- - - - -
```



```
!- - - - -
! Write error information to the log.
!- - - - -

function bfCONTRLsupport ()
    !- - - - -
    ! Write the start of a new message to the log file.
    !- - - - -
    log ("*** CONTROL NEW ", eUNH.0062, ":", \
        eUNH.S009.0065, ":", \
        eUNH.S009.0052, ":", \
        eUNH.S009.0054, ":", \
        eUNH.S009.0051, NL)
    !- - - - -
    ! Write all error conditions and skip the rest of the message.
    !- - - - -
    edierrorlist (MESSAGE)
endfunction
```

4.8.5 Checking the Validity of a Segment

The validity of a single segment can be checked with the valid function in the same way that we checked the message. The edierrordump and edierrorlist functions can also be applied to segments. Only the current segment and its errors are written to the log file.

```
segment UNH
    if valid (SEGMENT) = FALSE then
        log ("Segment UNH contained errors", NL)
        edierrordump (SEGMENT)
    endif
    !- - - - -
    ! Normal processing for UNH
    !- - - - -
endsegment

segment BGM
    if valid (SEGMENT) = FALSE then
        log ("Segment BGM contained errors", NL)
        edierrordump (SEGMENT)
    endif
    !- - - - -
    ! Normal processing for BGM
    !- - - - -
endsegment
```



Segment statement will be executed only if PROCESS_ERRONEOUS=TRUE (see section "4.8.3 - Processing Erroneous Messages in Statement Lists" on page 64).

4.8.6 Checking the Validity of a Group

The validity of a single group can be checked with the valid function in the same way that we checked the message.

```
if valid (GROUP g25) = FALSE then
    print ("Groupe non valide", NL)
else
    print ("Groupe valide", NL)
endif
```

4.8.7 Examples

The following INVOIC message contains three errors:

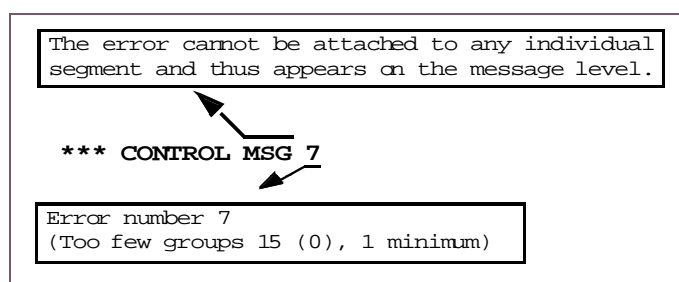
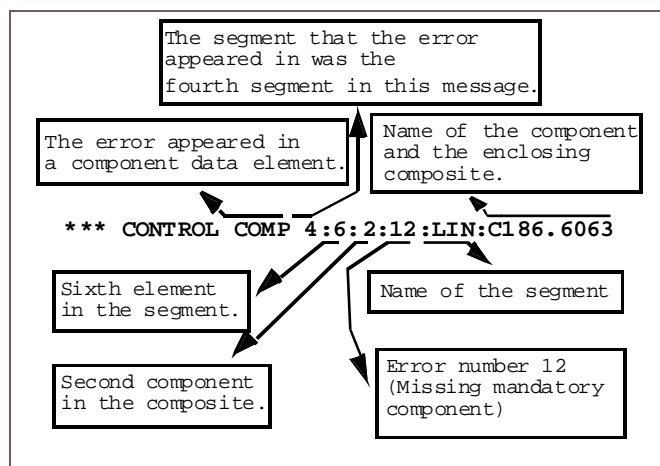
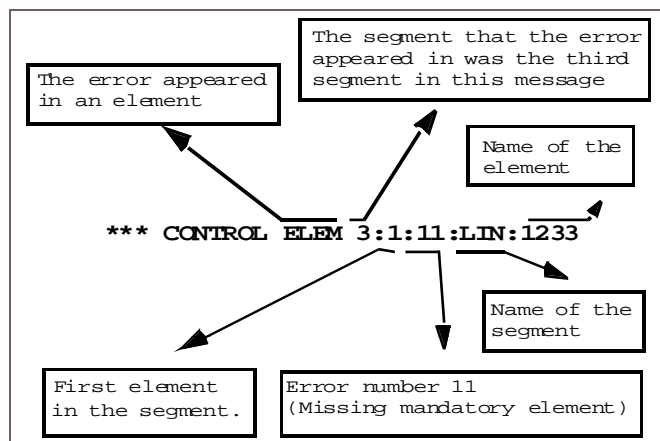
- The element 1233 in the first LIN segment is missing.
- The component 6063 in composite C186 in the second LIN segment is missing.
- Segment MOA, which starts group 15, is missing after the UNS segment.

```
UNH+123001+INVOIC:2:912:UN'
BGM+:::INVOICE FOR DISTRIBUTOR++3:930301:101'
LIN++1++3:007234110++1:1+INV:28.50'
LIN+1+2++3:007234150++2+INV:120.00'
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
UNT+8+123001'
```

The bfCONTRLsupport function in the example earlier in this chapter would produce the following data in the log file. The first line results from the log function call that we made in bfCONTRLsupport. The last three lines are the output from the edierrorlist function.

```
*** CONTROL NEW 123001:INVOIC:2:912:UN
*** CONTROL ELEM 3:1:11:LIN:1233
*** CONTROL COMP 4:6:2:12:LIN:C186.6063
*** CONTROL MSG 7
```

The lines produced by the edierrorlist function are explained below. The common string "*** CONTROL " at the beginning of each error line makes it simple to extract this data from the log file.



The `edierrordump` function outputs the message in EDI format, with all error messages attached in the appropriate positions.

```
UNH+123001+INVOIC:2:912:UN'
BGM+:::INVOICE FOR DISTRIBUTOR++3:930301:101'
LIN++1++3:007234110++1:1+INV:28.50'
    ^ 1233: Missing mandatory element (11)
LIN+1+2++3:007234150++2+INV:120.00'
    ^ C186.6063: Missing mandatory component (12)
LIN+1+3++3:007234199++1:1+INV:28.50'
UNS+S'
UNT+8+123001'
Too few groups 15 (0), 1 minimum
```

4.8.8 Error Codes

The meaningful error codes from the translator module are listed below. These error codes refer to the contents of the messages either directly or indirectly. There are other error messages in addition to these, but they relate more to the internal functions of the translating module.

- | | |
|----|---|
| 1 | Too much data in the element |
| 3 | Too short: %d minimum |
| 4 | Too many %s groups (%d), %d maximum |
| 5 | Too many %s segments (%d), %d maximum |
| 6 | Too long: %d maximum |
| 7 | Too few groups %s (%d), %d minimum |
| 8 | Too few segments %s (%d), %d minimum |
| 9 | Cannot find character set %s. |
| 11 | Missing mandatory element |
| 12 | Missing mandatory component |
| 16 | Segment '%s' does not belong to the message |
| 17 | Excess data in the segment |
| 20 | Corrupted character set %s |
| 21 | Message contains errors22 Segment contains errors |
| 23 | Invalid type %s field%s |



Chapter 5 - Input and Output

Standard Streams

Input

Output

Redirection

Buffered Output

Print List

Print Items

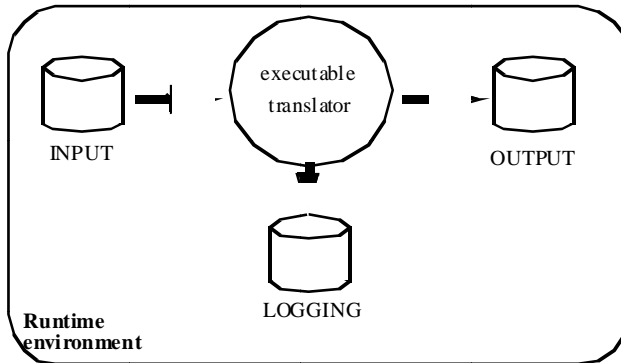
Print Format

Logging

File Variables

5.1 Standard Streams

By default, an RTE program reads its standard input, outputs to standard output, and writes the logging information to the standard error. In Unix systems, these file descriptors are inherited from the parent process.



The RTE language provides an easy access to the file system.

5.2 Input

The RTE programs are data driven by default. This means that the incoming data determines which statement lists are executed, instead of the programs being linearly executed (although this is possible, too).

5.2.1 Window Model

The data is read in line by line. The maximum line length is 4096 characters. Each line is compared with the line matching rules for each line statement until a match is found. The rules are checked in the order in which they appear in the source file. If there is no match, then the optional default statement list is executed.

Any number of lines from the input file can be processed. The process matches the statements in the statement list to the lines in the input file ("window"). These lines form an imaginary window view of a file that is visible to the current statement list process. The lines in the window will not be available to the next statement list. The system moves to the next unread line and does the matching based on the rules in the statement list.

5.2.2 Picking

Pick is the built-in function used to read from a file. It always works relative to the line that matched the matching rule for the statement list. The matched line is the first one in the window.

The following function call will read the entire first line from the window:

```
pick (1, 1, EOL)
```

The size of the window is always determined by the furthest pick. The size of the window is not affected by whether or not all intermediate lines are explicitly read.

The following line statement list always processes two lines from the input.

```
line (1:"SAMPLE")
    tTmp1 := pick (1, 1, EOL)
    tTmp2 := pick (2, 1, EOL)
endline
```

5.2.2.1 Returning a Line

The internal variable RETURN_BUFFER can be used as an input replacement buffer. Assigning this variable a value causes the program to read its content as the next input line. The content is automatically erased after it has been used. Only one line can be returned.

5.2.2.2 Line Numbers

The numeric constant LINE contains the current input line number. The value is meaningful only when text data is being read.

5.2.2.3 End of File

The pick function returns the constant string EOF when end of file is encountered. An error message is also written to the logging stream, and if the LOGLEVEL is high enough, the execution of the program is terminated.

5.2.3 Read

The read function reads a line from a file or a process, and returns the line just read (or EOF if there are no more lines). The file from which the lines are read must be closed before it can be reread. It is also good practice to close all processes that are read from. The read function takes a file name as its argument.

```
read (tFile)
```

The following example reads the **/etc/passwd** file and prints it.

```
tFile := "/etc/passwd"
print("List of registered users:", NL)
tLine := read(tFile)
while tLine <> EOF do
    print(tLine,NL)
    tLine := read(tFile)
endwhile
close(tFile)
```

The following example reads the machine name from the command **uname**, using it as a coprocess. Note the **"|"** character at the beginning of the command line.

```
tFile := "|uname -a"
tLine := read(tFile)
close(tFile)
print("Machine ", tLine, NL)
```

Redirection can also be used with the read function to redirect the input.

5.3 Output

RTE supports two types of output: direct and buffered. Direct output writes the given text to the output stream in the order in which the print functions are called in the program. Buffered output allows you to build the file image in memory, and that image is then output to an actual file at the desired time.

5.3.1 Printing

The built-in function `print` prints its arguments to standard output. The arguments are written as a print list (see “5.6 - Print List” on page 76). The output of a single print statement can be redirected to a named file or process, as shown below.

```
! Print contents of tTMP plus a new line
print (tTMP, NL)
```

5.3.2 Newline Character

The text variable `NL` contains the line separator used by the internal functions that produce multiple lines. By default, this is the newline character (ASCII 012 in octal presentation). Note that the `print` function does not automatically produce a new line; you have to include it in the print list. The value of the `NL` variable can be set in the source code.

5.4 Redirection

You can redirect the output from print commands to any file or coprocess by using the redirection symbols after the call to the print function.

```
print (print list) > tFilename
print (print list) >> tFilename
```

The symbol “>” opens the specified file in overwrite mode. A new file is created, or the possibly existing file is truncated to length zero. The append symbol “>>” always writes after the current end of file.

The symbols have a different meaning only at the first invocation of any redirected output. The subsequent print statements redirected to the same file (or process) work in append mode. This is illustrated in the following example.

```
begin
  - - - - -
  ! This call creates file "/tmp/file" or truncates it if it existed.
  - - - - -
  print ("Input lines:", NL) > "/tmp/file"
endbegin
```



```
default
    !- - - - -
    ! This call continues to write to the same file ("/tmp/file")
    !- - - - -
    print (pick (1, 1, EOL), NL) > "/tmp/file"
enddefault

end
    !- - - - -
    ! Close the file "/tmp/file".
    !- - - - -
    close ("/tmp/file")
endend
```

This program appends another line to the file created in the previous example and then reads that file line by line.

```
begin
    !- - - - -
    ! This call opens file "/tmp/file" in append mode.
    !- - - - -
    print ("ANOTHER PROGRAM:", NL) >> "/tmp/file"
    close ("/tmp/file")
    !- - - - -
    ! Start reading from the same file.
    ! Note that we closed the file so that all data will be available.
    !- - - - -
    tLine := read ("/tmp/file")

    while tLine <> EOF do
        log (tLine, NL)
        tLine := read ("/tmp/file")
    endwhile

    close ("/tmp/file")
endbegin
```

The file name must be the same for each statement; even one extra space in front of a file name will cause the system to regard it as a different file. This approach avoids file descriptors and low-level file handling routines, but in return requires the user to be very precise and careful with file names. A safe way to handle multiple redirections is to store the used file name in a text variable and use that variable globally.

The file can be closed with the close function. All operating system buffers are flushed to the disk and the internal file descriptor is returned for reuse. Files do not have to be closed, but note the following: if you first write to a file and then plan to read from the same file, you must close it to be sure that all the lines you wrote are indeed in the file. Another reason for closing files is that they are a limited resource: only 40 different files can be in use at any given time.

5.4.1 Using Coprocesses

In addition to regular files, the output can be redirected to a coprocess. This means that there will be another process running and the output is redirected to the standard input of that process.

A coprocess is indicated by a "|" character as the first character in the file name. No spaces are allowed before the "pipe" indicator. The file name may be a complete shell command line with its own arguments.

The file name must be exactly the same after each statement; otherwise a new coprocess is created. It pays to use a text variable instead of always writing out the entire command line.

```
tMailer := "|mailx friend@machine"
print ("TESTMESSAGE", NL) > tMailer
print ("See you tonite!", NL) > tMailer
close (tMailer)
```

The read function can also read output from another program. In the example below, we read lines from the program tr (a Unix character translator utility) that in turn reads the file /tmp/ file.

```
tToLower := "|tr \"[A-Z]\" \"[a-z]\" < /tmp/file"
tLine := read (tToLower)
while tLine <> EOF do
    ...
    tLine := read (tToLower)
endwhile
```

5.4.2 Redirecting the Standard Streams

The standard streams can be directed to a regular file or a coprocess. The redirect function reopens the given stream and the file assigned to the stream is read/written from the beginning.

```
redirect (INPUT, tFilename)
redirect (OUTPUT, tFilename)
redirect (LOGGING, tFilename)
```

The following example reads the input from a Unix utility, strings, and prints the lines to the sort program. Logging is closed so nothing will be written to the standard error stream.

```
begin
    redirect (INPUT, "| strings /tmp/exefile")
    close (LOGGING)
    redirect (OUTPUT, "| sort >/tmp/strings.sorted")
endbegin

default
    print (pick (1, 1, EOL), NL)
enddefault
```

5.5 Buffered Output

Buffered output lets you print data to the output in non-linear order. The buffer is filled with the **put** function and printed with the **flush** function.

5.5.1 Putting Text in the Buffer

In the put function, the line and column of the output buffer are given as arguments.

```
put (nLine, nPos, print item)
```

Arguments **nLine** and **nPos** are numeric values and the print item can be either a number or text value. The print item can also have a format definition. See the description of the print item later in this chapter.

```
put (1, 1, "ordinary")
put (nTmp, nColumn + 2, 38)
put (nTmp + 1, nColumn, nCount:10.2)
```

If there are untouched areas in the output, they are filled with the FILL_CHAR character. For example, if the line text does not begin in the first column or is shorter than the specified minimum width, the filling mechanism is used. By default, FILL_CHAR is the space character, but it can be set to any character in the RTE program.

```
FILL_CHAR := "#"
```



If print formats are used, the fill character for those formats is always a space.

```
put (1, 1, "Begin":5)
```

The previous example prints the text "Begin" at the beginning of the first line in the output buffer, followed by five spaces. These characters are not considered an untouched area and are not filled with the FILL_CHAR character.

The buffer used by the **put** function can be set to a user-defined buffer. The buffer type is text array.

```
put (taArray, nLine, nPos, print item)
put (taTmp, 1, nColumn, tText)
```

5.5.2 Flushing the Output Buffer

The content of the output buffer is written with the **flush** function. The **flush** function takes the line minimum length, line maximum length, and line separator character as arguments.

```
flush ( nMin, nMax, tSeparator)
flush ( taArray, nMin, nMax, tSeparator)
```

nMin and nMax are numeric values. If their value is zero, they have no meaning. Otherwise they behave as follows.

- **min** - All printed lines contain at least nMin characters. If the line contains fewer characters than nMin, the line is filled with the FILL_CHAR character.

- **max** - Only nMax characters are printed per line. If the line has more than nMax characters, the entire line and the descriptive error message are printed in the logging stream. The printed line contains only nMax characters.

```
flush (80, 0, NL)
flush (0, 80, NL)
flush (72, 72, NL)
```

Flush works only with those buffers that are filled with the put function. Other buffers (arrays) can be printed with the print function.

The buffer can be flushed to file with the redirection mechanisms.

```
flush (80, 0, NL) > "lines"
flush (0, 80, NL) >> "lines"
flush (72, 72, NL) > "| teletex 564221"
```

The buffer content is destroyed after flush is completed. If the user wants to keep the buffer contents after the flush operation, the RTE program must be compiled with the -w option. When the buffer is no longer needed, it can be removed with the remove function.

If no buffer name is given in the put function, the default buffer is used (BUFFER).

5.5.3 Examples

```
FILL_CHAR := "#"
put (1, 4, "BEGIN")
put (1, 12, "END")
flush (20, 20, NL)
###BEGIN###END#####
FILL_CHAR := "#"
put (1, 4, "BEGIN":6)
put (1, 12, "END")
flush (20, 20, NL)
###BEGIN ##END#####
nNumber := 345.67
put (1, 1, nNumber:08.3)
put (2, 1, nNumber)
put (2, 6, "OVERRUN")
flush (0, 0, NL)
0345.670
345.6OVERRUN
```

5.6 Print List

The following built-in functions use a print list:

- print
- log
- debug

- build.

The print list consists of printable items separated with commas. The number of items is unlimited. There is no automatic line feed in any of the printing functions.

```
print ("The total sum is ", nTOTAL, " FMK", NL)
```

5.7 Print Items

Print items are text and numeric values. A print format specification can be added in all allowed contexts. Print items are used in print lists and in the put function.

5.8 Print Format

Print format can be used with print items. The format always begins with a colon immediately following the print item. The min and max values can be either constant numbers, or "*" characters indicating that the value is to be evaluated at run time from the following argument.

5.8.1 Text Print Format

The formal presentation for the text print format is:

```
: [LR]min[.max]
```

where the colon and the value that specifies the minimum width are mandatory. Optionally, a letter can immediately follow the colon to indicate justification. The possible values are:

```
L left justification
R right justification
```

For text data, a missing justification specification will result in left justification. The optional maximum length is separated with a period (".") from the minimum width value. The max value dictates the maximum number of characters that will be extracted from the text argument.

"TEXT":8	"TEXT"
"TEXT":L8	"TEXT"
"TEXT":8.2	"TE "
"TEXT":L8.2	"TE "
"TEXT":*, 6	"TEXT "
"TEXT":L*, 6	"TEXT "
"TEXT":L*.*, 6, 3	"TEX "
"TEXT":R8	"TEXT"
"TEXT":R8.2	"TE"
"TEXT":R*, 6	"TEXT"
"TEXT":R*.*, 6, 3	"TEX"

5.8.2 Numeric Print Format

Numeric print format is the following:

```
: [LR][0]min[.decimals]
```

The starting colon and the minimum width are mandatory. A possible decimal separator is included in the total width. Before the width specification, the letter R or L can indicate justification, and the character "0" can indicate filling with zeroes. For numeric data, default justification is to the right. The number of decimals can be specified in the last field, separated by a period (".") from the minimum width value.

nVALUE :=	345.67
nVALUE	"345.67 "
nVALUE:8	" 345.67"
nVALUE:08	"00345.67"
nVALUE:R8	" 345.67"
nVALUE:R08	"00345.67"
nVALUE:8.3	" 345.670"
nVALUE:08.3	"0345.670"
nVALUE:L8	"345.67 "
nVALUE:L08	"345.67 "
nVALUE:L8.3	"345.670 "

5.8.3 Time Printing

Time values are printed with the time function, which returns the time in the requested text format. For full details on how to use this function, see "8.3.34 - time" on page 127.

```
print (time ("%d.%m.%y"), NL)
print (ENTRY.EXPIRES, "%y%m%d%H%S", NL)
```

The format string can contain regular characters, in addition to the following special fields.

```
print (time ("Just now the time is %H:%S"), NL)
```

Fields with special meaning in time printing are:

```
%T Time in AM/PM notation (HH:MM AM|PM)
%t Time in AM/PM notation (HH:MM:SS AM|PM)
%M Minutes (00-59)
%H Hours (00-24)
%S Seconds (00-59)
%d Day of the month (01-31)
%m Month of the year (01-12)
%y Year in two digits (..98,99,00,01..)
%Y Absolute year
%N Month of the year in long format
(e. g. January)
%n Month of the year in short format
(e. g. Jan)
%W Day of the week in long format
(e. g. Monday)
%w Day of the week in abbreviated format
(e. g. Mon)
%k Week of the year (1 - 53)
%j Day of the year (1 - 366)
%a Absolute value
```

```
(in seconds since Jan 1, 1970 GMT)
%M Elapsed minutes to current system time
%H Elapsed hours
%d Elapsed days
%ew Elapsed weeks
```

A negative value in elapsed time indicates that the given time is in the future

5.9 Logging

Logging can be used to track the execution of a program and to collect applicable information for later review. The logging stream is by default the standard error.

- log
- debug

The LOGLEVEL numeric variable indicates the logging level being used. The values are interpreted as follows:

- 0 - No system warnings are written and the debug commands are not active.
- 1 - All system warnings are written to the logging stream.
- 2 - The debug commands are active.
- 3 - All system warnings are considered fatal and the execution of the entire program is terminated.

The higher values imply the lower values. For example, at level two, in addition to the debug commands, the system warnings are also active.

The initial value can also be set with the command line option -l.

The logging stream is by default the standard error. It can be directed to another file or process with the redirect command.

```
redirect (LOGGING, "/tmp/log")
redirect (LOGGING, "| grep ERROR > /tmp/errors")
```

If you do not want to see any logging information, the logging stream can also be closed.

```
close (LOGGING)
```

After closing the logging stream, you can reopen it to the desired file.

```
redirect (LOGGING, "/tmp/newlog")
```

5.10 File Variables

Files and their attributes can be accessed conveniently with file variables. A file variable is an identifier that can be used alone as an argument to the built-in functions print, log, debug and remove.

5.10.1 File Attributes

The following table lists all the possible file attributes. The data type and a short explanation are provided for each attribute.

Name	Type	Description
EXIST	Boolean	Is true if before doing any other operations on the file.the file exists. This is a useful attribute to check
NAME	Text	Contains the name component of the file
PATH	Text	Contains the path component of the file
FULLNAME	Text	Contains the full name of the file.
OWNER	Text	Contains the user name of the file owner
GROUP	Text	Contains the owner group for the file.
SIZE	Number	Contains the number of bytes in the specified file.
LINES	Number.	Contains the number of lines in the file. For non-ASCII files, this value is meaningless
ATIME	Time	Time of last access to the file. This information is updated every time the data in the file is accessed.
MTIME	Time	Time of last data modification. This information is updated every time the data in the file is modified
CTIME	Time	Creation time.
READ	Boolean	Is true if the user has a read access to the file.
WRITE	Boolean	Is true if the user has a write access to the file
EXEC	Boolean	Is true if the user has an execute access to the file.
TYPE	Text	Contains either "REGULAR" for regular files, "DIRECTORY" for directories, or "SPECIAL" for any other types.
CONTENT	Text	This value is retrieved from an external program that is used to determine file content.

5.10.2 Assignment

File variable assignment is as follows:

```
fFile := tFileName
```

Where fFile is the file variable and tFileName is the name of the file.

The file variable can also be assigned in the while statement:

```
while fFile in "/usr/tmp/A*" do
    !      ! File manipulation actions.
    ! Here the fFile variable gets a different content for each loop.
endwhile
```

5.10.3 Usage

A file variable is most useful in the following situations:

- file existence and permission checks

```
fFile := tOutputFile
if fFile.EXIST = TRUE then
    print ("File exists", NL)
endif
```

- file name manipulation; for example, base name and directory name extractions

```
fFile := tTestFile
print ("Filename: ", fFile.NAME)
print ("Dirname : ", fFile.PATH)
print ("Fullname: ", fFile.FULLNAME)
```

- directory scanning for selected files

```
while fFile in "/usr/tmp/A*" do
    print ("File: ", fFile.NAME)
endwhile
```

- getting the file size in bytes

```
fFile := tTestFile
print ("Size: ", fFile.SIZE, " bytes")
```



Chapter 6 - Database Interface

Background Information

Definition

Database Fields

Database Files

Finding an Entry

Creating a New Entry

Removing an Entry

6.1 Background Information

An EDIBASE database is a small general purpose database system designed to provide easy access to textual, numeric, and time data. The data consists of key fields and an undefined amount of additional information stored in related files. The EDIBASE database system contains an interface to RTE and a command line package. Only the RTE interface is covered here.

The key fields must be defined in the configuration file. The number of related files does not have to be known at configuration time. Below is a sample configuration file:

```
#=====
# User defined fields.
#=====
FIELD=NAME,16,%-16.16s, Name
FIELD=ADDRESS,32,%-32.32s, Address
FIELD=AGE,N,%3.0lf, Age
FIELD=EXPIRES,T,%d.%m.%Y %H:%M, Order expires
#=====
# Our names for the standard system fields.
#=====
INDEX=INDEX,%4.0lf,
MTIME=MODIFIED,%d.%m.%Y %H:%M, Modified
CTIME=CREATED,%d.%m.%Y %H:%M, Created
```

The configuration file specifies that the database contains four user-defined key fields: NAME, ADDRESS, AGE, and EXPIRES. NAME is text and contains a maximum of 16 characters. ADDRESS is also text but can contain up to 32 characters. AGE is numeric and EXPIRES is a time field. The two last comma-separated fields for each definition contain the default output format and the header for that column. Only the default print format for time fields is applicable to RTE; otherwise these two fields affect the command line package.

Each EDIBASE database contains the fields INDEX, MTIME, and CTIME. These fields are controlled by the system: they can be referenced by the user but no values can be assigned to them. In the configuration file, the user can give new names to these fields

6.2 Definition

Unlike all other variables, the database entries must be defined at the beginning of the program.

```
base "/usr/tmp/mybase.cfg" ENTRY
```

Several entries can be given in the same base statement.

```
base "/usr/tmp/mybase.cfg" ENTRY1 ENTRY2
```

Several base statements can exist in one RTE program file.

```
base "/usr/tmp/mybase.cfg" MY_ENTRY
base "/usr/tmp/extrabase.cfg" EXTRA_ENTRY
```

The following options are possible:

- readonly
- autoflush (on/off)
- default (on/off)

The database definition shown above causes the configuration file to be read into the RTE compiler. If the definition does not contain the complete path, it is assumed to be relative to the \$HOME/database or (on a secondary basis) the \$EDIRHOME/database directory.

6.3 Database Fields

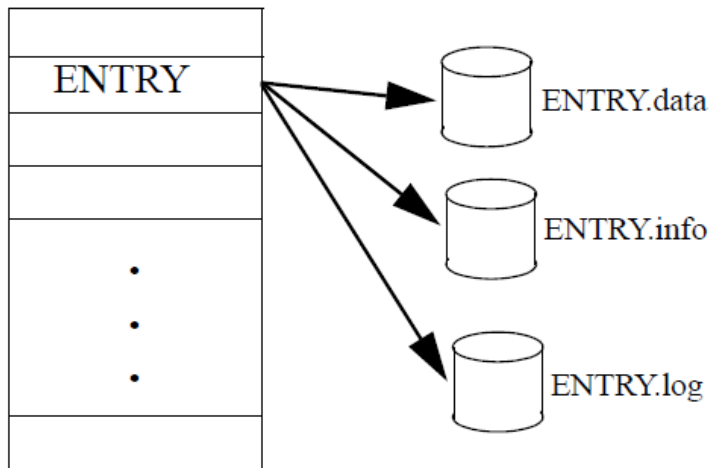
Database fields are referenced by specifying the entry and the field name, separated by a period. These fields can now be used anywhere a variable of the same type can be used.

If text is assigned to the text field and the text is longer than the field, the value is truncated to the maximum length. This is not an error situation in RTE and no error or warning messages are printed.

Field	Description
ENTRY.NAME	Corresponds to a text variable. ENTRY.NAME := pick (1, 1, EOL) print (ENTRY.NAME, NL)
ENTRY.ADDRESS	Corresponds to a text variable. ENTRY.ADDRESS := tfGetAddress(ENTRY.NAME) log ("Address is ", ENTRY.ADDRESS, NL)
ENTRY.AGE	Corresponds to a numeric variable. ENTRY.AGE := 38 nYearsToWork := 65 - ENTRY.AGE
ENTRY.EXPIRES	Type time. The values for this field are assigned with a text expression. The value can be referenced with the time function. ENTRY.EXPIRES := "1.10.1994" print (time (ENTRY.EXPIRES), NL)

6.4 Database Files

Files related to a database entry are referenced by specifying the entry identifier and the file extension, separated by a period. The actual physical file name and path are hidden from the user.



6.5 Finding an Entry

The function find is used to assign a database identifier to an actual database entry. The database given as an argument must be an actual EDIBASE database and its schema must match the definition.

```
ENTRY := find (database name [, filter]) [sort field]
```

The database name must specify the complete path for the physical database. If the path does not start from the root directory, it is assumed to be relative to the \$HOME environment variable. All the examples in this chapter assume that there is a database called "mybase" in the user's home directory.

The filter can consist of one or more specifications that are in the format **field op value**

Field must be included in the defined database. Operator can be:

- "=" and "<>" for type TEXT
- "<", "<=", ">=", or ">" for NUMERIC and TIME fields

With text fields, comparisons can be done with wild card characters.

```
NAME="RISC*"
```

Several comparison operations can be given. If more than one operation is given, the operations are gathered with the logical operator AND. The same field can be used as many times as needed.

```
find ("mybase", NAME="myname", AGE<10)
```

If more complex filters are needed, users can write their own code to handle the comparison operations. For example, you can implement the logical operator OR by retrieving all entries and doing the comparison in the RTE code.

```
while ENTRY in ("mybase") do
    if (ENTRY.NAME="myname" or ENTRY.SIZE<10) then
        !
        ! Matching actions.
        !
    endif
endwhile
```

The sort field specifies the sort order. The order can be either ascending or descending. A lower case "a" in front of the field name defines it as ascending; a lower case "d" defines it as descending.

```
aNAME
dEXPIRES
```

The ENTRY identifier is assigned the first entry that matches the given filter. If no sort field is specified, the index is used as the sort key. Find does not support secondary sort keys.

If the filter matches more than one entry in the database, you can retrieve the next entries by calling the find function with no arguments. The same filter and sort order are applied as in the previous call to find, with the same ENTRY identifier. Assigned values to different identifiers from the same database operate independently from each other.

```
ENTRY := find ()
```

The existence of an entry in a database can be checked with the **valid** function.

```
ENTRY := find ("mybase", NAME="myname")
if valid (ENTRY) = FALSE then
    log ("Could not find an entry.", NL)
endif
```

Using an ENTRY without checking its validity causes the system to write error messages to the logging stream.

6.6 Creating a New Entry

A new entry to a database can be created with the **new** function. It takes as its only parameter the name of the actual database.

```
ENTRY := new (database name)
```

The initial values are assigned after the call to the new function.

```
ENTRY := new ("mybase")
ENTRY.NAME := "myname"
ENTRY.ADDRESS := "myaddress"
ENTRY.AGE := 45
ENTRY.EXPIRES := "now+10d"

print ("Created :", time (ENTRY.CREATED), NL) > ENTRY.log
```

6.7 Removing an Entry

To remove an entry from the database, call the general purpose function **remove** with the database identifier as the only argument.

```
remove (ENTRY)
```

This call removes the key information and all related files.

The following example shows how all entries that have been created over three days are removed from the database.

```
while ENTRY in ("mybase", CREATED < "now - 3d") do  
    remove (ENTRY)  
endwhile
```



Chapter 7 - Running RTE Programs

Overview

Options

Parameters

Arguments

Startup File

7.1 Overview

An RTE program command line can contain options, parameters, and arguments. All options must be specified before the parameters, which in turn must appear before any free form arguments.

```
$ program [options] [parameters] [arguments]
```

7.2 Options

Options can be either user options or system-specified control options. An option is a oneletter flag preceded by a minus sign.

User options are always upper case letters from A to Z. You have access to 26 option variables in an RTE program. The option variables are named oA to oZ. By default, these options contain the value FALSE. When you give an option on the command line, the value automatically becomes TRUE.

```
begin
  if oA = TRUE then
    print ("Option A selected.", NL)
  else
    print ("Option A not selected", NL)
  endif
endbegin

$ myprog -A
Option A selected
$ myprog
Option A not selected
```

System options are written in lower case letters and they are mainly used to initialize system variables.

The following list includes all the system options.

- l** Sets the logging level; that is, gives an initial value to the variable LOGLEVEL.
- c** Specifies the character set file that is to be used in the translation.
- d** Specifies the path component for the character set file. This option can repeat up to sixteen times and the directories are searched in the specified order. The current working directory is always checked last.
- f** Takes the input from the following file.
- p** Reads the parameters from the following file name. The parameters must be specified as name-value pairs and the separator is an equal sign. The parameters, possibly changed, are written back to this file at the termination of the program.

- s** Sets the separators to contain the following values. The values are specified as one character string containing CS, ES, ST, RC, and optionally also TS and RS.
- **CS** - Component separator. Used in EDIFACT to separate component data elements.
 - **ES** - Element separator. Used in all grammars to separate data elements.
 - **ST** - Segment terminator. Used in all grammars to indicate the end of a data segment.
 - **RC** - Release character. Used in all EDI grammars (except ANSI X12) to cancel the special function of a reserved character in data.
 - **TS** - Tag separator. Used in GTDI to separate the segment tag from the actual data. If this is not specified, the value from the element separator is used here.
 - **RS** - Repetition Separator. Used in ANSI X12 to separate repeating standalone or composite data.
- a** Sets the application decimal separator to the following value.
- m** Sets the message decimal separator to the following value.
- i** Prints information about the program. This information is collected from the source file comments that start with a percent sign.
- q** Cancels the updating of the possible parameter file at the termination of the program.
- b** Sets the output and logging to be unbuffered. The default is to use system-defined I/O buffers for all input and output.

7.3 Parameters

Parameters are special text variables in RTE. Parameters are name-value pairs that can get their value from a named file and/or from the command line. They are referenced with the parameter name preceded by a lower case "p."

```
begin
  if pFILE = EMPTY then
    log ("No file given.", NL)
    exit (1)
  else
    print ("File was ", pFILE, ".", NL)
  endif
endbegin

$ myprog FILE=tmp001
File was tmp001.
$ myprog
No file given.
```

Assume the content of the file params is as follows:

```
FILE=tmp002
ACTION=remove
```

The parameters can be read from a specified file. They must be specified as name-value pairs, one per line, and the separator is the equal sign.

```
$ myprog -p params
File was tmp002.
```

If the parameters were read from a file, they are automatically written back to the same file when the program terminates. The parameters set on the command line and those created in the program are also written to the file. If the command line contains the -q option, the parameter file is not updated. After the call:

```
$ myprog -p params NEW=newparam
```

the file **params** has the following content:

```
ACTION=remove
FILE=tmp002
NEW=newparam
```

Finally, if the source for **myprog** is changed to contain the creation of a new parameter:

```
begin
  if pFILE = EMPTY then
    log ("No file given.", NL)
    exit (1)
  else
    print ("File was ", pFILE, ".", NL)
  endif
  pMADE_IN_PROGRAM := "Created in the program"
endbegin
```

then after running the following command line (assuming that the file **params** contains what was put there in the previous example):

```
$ myprog -p params NEWEST=another
```

the file **params** now has the following content:

```
ACTION=remove
FILE=tmp002
MADE_IN_PROGRAM=Created in the program
NEW=newparam
NEWEST=another
```

7.4 Arguments

Free form arguments come after all the options and parameters on the command line. RTE contains two built-in variables to control the arguments:

- **ARGC:** a numeric variable that contains the number of arguments given to the program. The ARGV[0] variable is not counted in this number.
- **ARGV:** a text array that contains all the arguments. ARGV[0] is the name of the program and must always be present. The other arguments are from ARGV[1] to ARGV[ARGC].

The options and parameters are not counted as arguments and do not appear in the ARGV variable.

```
begin
    print (ARGC, NL)
    print (ARGV)
endbegin

$ myprog -A -13 NEWPARAM=test first second
2
0001=first
0002=second
```

7.5 Startup File

When an RTE program is started, the user's home directory is checked for a special startup file. The file is named **.RTErc** and it contains initial values for three different purposes. The startup file is optional and no error message is produced if it does not exist. The three different categories are described below.

EDI exceptions

These values specify which types of EDI translation exceptions are ignored and which ones cause an error condition. The default for each exception is to cause an error condition.

Initial values

Initial values can be given to decimal separators and the logging level. These values are overwritten if other values are given on the command line and in the code.

System tuning

These values affect the program's behavior. The value `ARRAY_INDEX_WIDTH` specifies the way in which numeric array indexes are converted to text indexes. The value `CODE_CONVERSION_BUFFER` specifies how many lines of the code conversion files are kept in memory.

EOF_STRING

Contains the text string that is used to signal the end of file condition. This needs to be changed only if the processed data may contain the default indicator `"__EOF__"`.

A sample startup file.

```
#- - - - - #
@ (#)TradeXpress RTE start-up file
# Exceptions causing an error in segment
# translation.
#- - - - -
INVALID_CONTENT=1
MISSING_ELEMENTS=1
MISSING_COMPONENTS=1
EXCESS_COMPONENTS=1
EXCESS_ELEMENTS=1
TOO_LONG_ELEMENTS=1
TOO_SHORT_ELEMENTS=1
```

```
MISSING_SEGMENTS=1
MISSING_GROUPS=1
TOO_MANY_SEGMENTS=1
TOO_MANY_GROUPS=1
GARBAGE_OUTSIDE=1
GARBAGE_INSIDE=1
EMPTY_SEGMENTS=1
PACK_COMPONENTS=1
PACK_ELEMENTS=1
#- - - - -
# Defaults for selected values.
#- - - - -
INHOUSE_DECIMAL_SEP=.
MESSAGE_DECIMAL_SEP=.
LOGGING_LEVEL=1
#- - - - -
# System tuning.
#- - - - -
ARRAY_INDEX_WIDTH=4
CODE_CONVERSION_BUFFER=1024
EOF_STRING=__EOF__
```



Chapter 8 - Built-in Functions Reference

[Overview](#)

[Function Categories](#)

[Functions Reference](#)

8.1 Overview

This chapter provides a reference for every RTE built-in function and gives the following information for each function.

- **Usage:** a function prototype with explanatory arguments.
- **Arguments:** a detailed description of each argument.
- **Returns:** a description of the return value.
- **Description:** a comprehensive description of the function.
- **See also:** references to the functions related to this topic.
- **Example:** one or two examples of the most common function usage.

The functions are arranged in alphabetical order

8.2 Function Categories

The functions are grouped in categories depending on their nature and behavior. These categories and their functions are described in the following sections.

8.2.1 Process Control

The process control functions are related to Unix processes. They allow the user to run programs from the RTE program in a similar way to running programs in Unix.

Function	Description
background	Creates a new process and executes the given command as a daemon process.
exec	Executes the command by replacing the existing process.
exit	Terminates the current process.
spawn	Executes the command and waits until the process ends.
system	Executes the given command with the shell.

8.2.2 Text Handling

The text functions are used with text strings. They allow the user to create, manipulate, and transform text strings in various ways.

Function	Description
build	Constructs a new string from given strings.
compare	Compares text strings.
index	Finds the location of a given character in the given string.
length	Counts the characters in the given string.
number	Transforms a string to a number.

Function	Description
peel	Peels unwanted characters from the given string.
replace	Replaces given characters in the given string.
split	Splits the given string.
strip	Strips the given characters from the given string.
substr	Extracts a substring from the given string.
toupper	Transforms lower case letters to uppercase letters.
tolower	Transforms upper case letters to lower case letters.

8.2.3 File Handling

The file handling functions allow the user to manipulate files in the Unix file system. See also “Chapter 5 -Input and Output” on page 69, for a more comprehensive description of all the RTE file handling capabilities.

Function	Description
close	Closes the given file.
copy	Copies the given file to another file.
link	Creates a new link to the given file.
redirect	Redirects the given i/o stream.
remove	Removes the given file.
rename	Renames the given file.

8.2.4 Output

The functions in this category allow the user to write to various targets. The functions can handle different data types automatically, such as strings, arrays, and so on. See “Chapter 5 -Input and Output” on page 69, for a more comprehensive description of RTE I/O mechanisms.

Function	Description
debug	Writes a message to the logging stream depending on the logging level.
ederrorlist	Writes an error list of the EDI errors found to the logging stream..
edierrordump	Writes the erroneous EDI message to the logging stream..
flush	Flushes the given buffers built with the put function.
log	Writes the given text to the logging stream.
print	Writes the given text to standard output.
put	Writes the given text to a buffer.

8.2.5 Database Access

RTE contains functions used to access the TradeXpress database system. Databases and programming with databases is explained in “Chapter 6 -Database Interface” on page 82

Function	Description
find	Finds an entry in the given database with the given filter.
new	Creates a new entry in the given database.

8.2.6 Input

The functions in this category allow the user to read data from various sources. The functions can handle different data types automatically, such as strings, arrays, name-value pairs, and so on. See “Chapter 5 -Input and Output” on page 69, for a more comprehensive description of RTE I/O mechanisms.

Function	Description
load	Loads name-value pairs from the given file.
pick	Picks a text string from the given position in the current input window.
read	Reads a line from the given file or process.

8.2.7 Multi-Purpose

Some of the RTE functions can handle different types of arguments. The function’s behavior and actions depend on the given argument type.

Function	Description
remove	Removes the given object.
valid	Checks the validity of the given object.

8.2.8 Time

RTE has access to the system time, which can be fetched with the time function. The time print format conversion is also supplied.

Function	Description
time	Converts system time, file access times, database times, or user-given times to text.

8.3 Functions Reference

The rest of this chapter contains documentation for all the RTE functions.

8.3.1 background

Usage	background (tCommand, taArgs, tStdin, tStdout, tStderr)
Arguments	<ul style="list-style-type: none"> • tCommand: command to execute • taArgs: argument list for the command • tStdin: text file to be fed in to process standard input • tStdout: writes data from the standard output to a text file • tStderr: writes data from the standard error list to a text file
Returns	Nothing
Description	<p>Creates a new process that runs the program specified by tCommand as a daemon. Items in the table taArgs are given as arguments to the program. The standard file descriptors stdin, stdout, and stderr are redirected to files specified by the respective arguments.</p> <p>The process will become a child of the unit process and all relationships with this process are terminated.</p>
See also	<ul style="list-style-type: none"> • exec • spawn • system

Example:

```
begin
    !Run the mailchecker process as a daemon process
    !with environment variable LOGIN and number 60 as arguments.

    taArgs[1] := sLOGIN
    taArgs[2] := "60"
    background ("mailchecker", taArgs, "/dev/null", "/dev/null", "/dev/null")
endbegin
```

8.3.2 build

Usage	build (print list)
Arguments	print list : see "5.6 - Print List" on page 76, for a print list description.
Returns	A new text string

Description	<p>Combines all print items in the print list to form a single text, and returns it. This is similar to the print function, but the result is returned as a value instead of being printed to a file</p> <p>This function is used with a single numeric argument to convert from numeric representation to text format.</p>
Note	<p>The resulting text string cannot be more than eight kilobytes in size. Longer strings result in internal buffer overflow and immediate termination of the program.</p> <p>Any print format enhancement can be used.</p>
See also	Print list

Example:

```
begin

    ! Build database name for later use.
    ! Variable $HOME refers to environment variable HOME
    tBASE := build($HOME, "/example")

    ! Build logfile name and redirect logging
    redirect(LOGGING, build(time("%y%m%d"), ".log"))

endbegin
```

8.3.3 close

Usage	close (tFile)
Arguments	tFilefile: (name to be closed)
Returns	Exit code of the waited process, if tFile is a process.
Description	<p>Closes the file tFile. All data buffered by the operating system is written to the named file, and internally the file descriptor is released for reuse. If tFile is a process, then it is terminated and waited for. The exit code of the process is returned.</p> <p>A subsequent write or read to the same file identifier causes the file to be reopened or, in the case of a process, the process to be restarted.</p> <p>The argument tFile can also specify any of the built-in streams (INPUT, OUTPUT, LOGGING).</p>
Note	If the user has written to a file and wants to read from that file, the file must be closed first.

See also	<ul style="list-style-type: none"> • print • read • redirect
----------	---

Example:

```
end
    ! Print the ending time and close the file
    print("Ending time ", time(), NL) > tFile
    close(tFile)
endend
```

8.3.4 compare

Usage	compare (tObject, tWildcard)
Arguments	<ul style="list-style-type: none"> • tObject: text to be compared • tWildcard: comparison text
Returns	Boolean value TRUE if matched and FALSE otherwise
Description	<p>Compares the text tObject with the text tWildcard and returns TRUE if they matched and FALSE otherwise. tWildcard can contain the following wildcards:</p> <ul style="list-style-type: none"> • *: Matches any number of characters. • ?: Matches a single character. • []: Specifies a set of characters that are a valid match at a given point. The set can contain ranges specified with a minus sign. In the range, the earlier character must be alphabetically less than the later character. <p>If any of the special wildcards must be used to represent itself, it must be escaped by double backslashes. For example, the text "*" used as tWildcard will find all texts that start with an asterisk and then contain any number of characters.</p>

Examples:

```
! Example 1
! Comparison results with various wildcard strings.

compare ("Sample Text", "Sam*")-> TRUE
compare ("Sample Text", "*T*")-> TRUE
compare ("Sample Text", "*a")-> FALSE
compare ("Sample Text", "[Ss]* [Tt]*")-> TRUE
compare ("Sample Text", "Sa??le *")-> TRUE
```

```
! Example 2
! Function nfGetType returns special purpose type of given argument.

function nfGetType(tParameter)

    if compare(tParameter, "*TEXT*") then
        ! text contains "TEXT"
        return(1)
    endif

    if compare(tParameter, "[Aa]*") then
        ! text begins with 'a' or 'A'
        return(2)
    endif

    if compare(tParameter, "\\**") then
        ! text begins with '*'
        return(3)
    else
        return(0)
    endif

endfunction
```

8.3.5 copy

Usage	copy (tSource, tDestination)
Arguments	<ul style="list-style-type: none"> • tSource: file name to copy from • tDestination: destination file name
Returns	Boolean value TRUE if the copy succeeds; otherwise FALSE.
Description	The file tSource is copied to the file tDestination . If the copying is successful, TRUE is returned; otherwise the return value is FALSE and the reason for the failure is written to the logging stream. The file names cannot contain any wildcards.
See also	<ul style="list-style-type: none"> • link • remove • rename

Examples:

```
! Example 1
copy (".profile", tFilename)
```

```
! Example 2
copy (build (sEDIHOME, "/sample"), "/tmp/sample")
```

```
! Example 3
% copies first argument (filename) to /tmp directory
begin
    if ARGV[1] = EMPTY then
        print("Usage: ", ARGV[0], " filename", NL)
        print(" copies 'filename' to /tmp", NL)
    else
        ! copy argument file to /tmp directory
        ! under the same name
        fFile := ARGV[1]
        tFile := fFile.NAME
        copy(ARGV[1], build("/tmp/", tFile))
    endif
endbegin
```

8.3.6 debug

Usage	debug (print list)
Arguments	print list : see "5.6 - Print List" on page 76, for a print list description.
Returns	Nothing
Description	The print list is written to the logging stream if the built-in variable LOGLEVEL is set to value 2 or higher. Otherwise, no output is produced.
See also	log

Example:

```
% run with parameter -l2 to test debug
begin
    debug("Run time is ", time(), NL)
    tBASE := build(sHOME, "/database")
    debug("Database is ", tBASE, NL)
endbegin
```

8.3.7 edierrordump

Usage	edierrordump (MESSAGE) edierrordump (SEGMENT)
Arguments	<ul style="list-style-type: none"> • MESSAGE: EDI message to be dumped • SEGMENT: a single segment to be dumped
Returns	Nothing
Description	Dumps the erroneous message or a segment into the logging stream. The dumped message or segment contains the error references placed in the message.
See also	<ul style="list-style-type: none"> • edierrorlist • valid

Example:

```
segment UNH
    if not valid(MESSAGE) then
        edierrordump (MESSAGE)
    endif
endsegment
```

8.3.8 edierrorlist

Usage	edierrorlist (MESSAGE) edierrorlist (SEGMENT)
Arguments	<ul style="list-style-type: none"> • MESSAGE: EDI message from which the list is generated • SEGMENT: a single segment from which the list is generated
Returns	Nothing
Description	Creates an error list for an EDI message and prints the error list into the logging stream. Similarly, if SEGMENT is given as the argument, the error list is printed for the current segment only.
See also	<ul style="list-style-type: none"> • edierrordump • valid

Example:

```
! This segment statement list processes incoming BGM segment.

segment BGM
    if valid(SEGMENT) then
        edierrorlist(SEGMENT)
```

```
endif
...
endsegment
```

8.3.9 exec

Usage	<pre>exec (tCommand, taArgs) exec (tCommand, tArg1, tArg2, ... , tArgN)</pre>
Arguments	<ul style="list-style-type: none"> • tCommand: command to be executed • taArgs: arguments for the command in array form • tArg1, tArg2, ... , tArgN: arguments for the command in list form
Returns	Nothing
Description	<p>Executes the program tCommand without creating a new process. Arguments to the new program can be given in a table or as an argument list. The list can be used only when the number of arguments is known at compile time.</p> <p>A successful call to <code>exec()</code> never returns.</p>
See also	<ul style="list-style-type: none"> • background • spawn • system

Examples:

```
! Example 1
  taArgs[1] := build (nIndex)
  taArgs[2] := tFilename
  exec ("transport", taArgs)
```

```
! Example 2
  exec ("transport", build (nIndex), tFilename)
%   executes command 'ls -la [param [param...]]'

begin
  taArgs[0] := "-la"
!
  nParam := 1
  while ARGV[nParam] <> EMPTY do
    taArgs[nParam] := ARGV[nParam]
    nParam++
  endwhile
  exec("ls", taArgs)
  print("This should never print", NL)
endbegin
```



```
end
    print("This point is never reached", NL)
    exec("not.exist", "dummy", "parameter")
endend
```

8.3.10 exit

Usage	exit (nExitcode)
Arguments	nExitcode: value to pass to the waiting process
Returns	Never returns
Description	Terminates the current program and passes the value nExitcode to the waiting process as the exit status. Normally a successful program returns a zero and other values are used to indicate error conditions.
See also	<ul style="list-style-type: none"> • exec • spawn

Examples:

```
! Example 1
if bAllok = TRUE then
    exit (0)
else
    exit (1)
endif
```

```
! Example 2
%   Return 0 if parameter file exists. Else
%   return 1.

begin
    fFile := ARGV[1]
    if fFile.EXIST then
        exit(0)
    else
        exit(1)
    endif
endbegin
```

8.3.11 find

Usage	find (tDatabase, filter) [sort order]
Arguments	<ul style="list-style-type: none"> • tDatabase: database name • filter: filter definition • sort order: sort order definition
Returns	Database entry
Description	<p>Used only with assignment to a database entry.</p> <p>This command, used with the database name and filter, finds the first entry that matches the filter. When no arguments are given, this command finds the next entry with the previous filter. Sort order specifies a field that is used as the sort key. The field must be preceded with "a" or "d" denoting ascending or descending sort order.</p>

Examples:

```
! Example 1
base "sample.cfg" ENTRY
begin
    ENTRY := find (tSample, NAME="*i*") dAGE
endbegin
```

```
! Example 2
base "example.cfg" ENTRY

begin
    tBASE := build(sHOME, "/example")
    ! Ascending list of all names containing "i"
    ENTRY := find(tBASE, NAME="*i*") aNAME
    while valid(ENTRY) do
        print(ENTRY.NAME, NL)
        ENTRY := find()
    endwhile
endbegin
```

8.3.12 flush

Usage	flush ([taMatrix], nMin, nMax, tLinesep) flush (logentry)
Arguments	<ul style="list-style-type: none"> • taMatrix: buffer to be flushed • nMin: minimum length of the printed line • nMax: maximum length of the printed line • tLinesep: separator used at the end of the line • logentry: logentry to be flushed
Returns	Nothing
Description	<p>Flush prints the contents of the buffer. If the buffer is not defined, the default buffer named BUFFER is used. If the nMin or nMax are zeros, they are ignored; otherwise they are used. If the line to be printed contains fewer characters than the nMin, the space is filled with the defined fill character named FILL_CHAR.</p> <p>If the line to be printed is longer than the nMax, only nMax characters are printed and the error message is written to the logging stream.</p> <p>If the autoflush off option has been stated, the variables in question are never written implicitly to the system, and a call to flush (logentry) is needed. By default, the variable is written in each assignment, so flushing is not necessary.</p>
See also	put

Example:

```
begin
  put (1, 2, "|1st line 2nd column")
  put (2, 5, "|2nd line 5th column")
  flush (0, 80, NL) > "/tmp/example"
endbegin
```

8.3.13 index

Usage	index (tOriginal, tSearch) index(tOriginal,tSearch,nOffset)
Arguments	<ul style="list-style-type: none"> • tOriginal: string to be searched • tSearch: searched string • nOffset: start searching at the offset nOffset
Returns	Numeric value containing the character position
Description	Returns the position of tSearch in tOriginal . If tSearch does not appear, the return value is zero.

Example:

```
!- - - - -
! nPosition gets 10 as its value
!- - - - -
nPosition := index ("Sample string", "ring")

!- - - - -
! ...and here nPosition gets 0.
!- - - - -
nPosition := index ("Sample string", "not there")
% Underlines existence of arg2 in arg1

begin
  print(ARGV[1], NL)
  nCount := index(ARGV[1], ARGV[2])
  if nCount > 1 then
    while nCount > 1 do
      print(" ")
      nCount--
    endwhile
    nCount := length(ARGV[2])
    while nCount > 0 do
      print("-")
      nCount--
    endwhile
  endif
  print(NL)
endbegin
```

8.3.14 length

Usage	length (tText)
Arguments	tText: text string
Returns	Numeric value containing the text string length.
Description	Returns the length of the argument.

Example:

```
begin
  print("Printing length of input lines", NL)
endbegin
```

```
default
    tLine := pick(1,1,EOL)
    print(length(tLine), " chars: ", tLine, NL)
enddefault

end
    print("All done", NL)
endend
```

8.3.15 link

Usage	link (tOriginal, tLink)
Arguments	<ul style="list-style-type: none"> • tOriginal: target file name • tLink: link name
Returns	Boolean value TRUE if successful; otherwise FALSE.
Description	The file referred to with the name tOriginal gets a new link named tLink . Both links refer to the same physical file. If the linking was successful, then TRUE is returned; otherwise the return value is FALSE and the reason for the failure is written to the logging stream. The file names cannot contain any wildcard characters.
Note	This function works only under UNIX. Link can't be made if the two files are located on different files system.
See also	<ul style="list-style-type: none"> • copy • remove • rename

Example:

```
link (".profile", tFilename)
link (build (sEDIHOME, "/sample"), "/tmp/sample")
% Simple command procedure to build a link to a file

begin
    tOriginal := build(sEDIHOME, "/database/example.cfg")
    tLink := build(sHOME, "/example.cfg")
    print("Enter commands", NL)
endbegin

line '[Hh][Ee][Ll][Pp]'
    print("Help not installed", NL)
endline
```

```
line (1:"S" or 1:"s")
    link(tOriginal, tLink)
endline

line (1:"R" or 1:"r")
    remove(tLink)
endline

line (1:"Q" or 1:"q")
    exit(0)
endline

default
    print("Unknown command.")
    print("Valid commands are S, R and Q", NL)
enddefault
```

8.3.16 load

Usage	load ([nMode,] tSrcFile, taArray [,tMultisep [,tSeparator]]) load ([nMode,] SrcLog, tSrcExt, taArray [,tMultisep [,tSeparator]])
Arguments	<ul style="list-style-type: none"> • nMode: load mode • tSrcFile: source file (in NAME=VALUE format) • taArray: text array to load into • SrcLog: database entry of the source • tSrcExt: "name" of the database table to load • tMultisep: separator character for multiple entry value • tSeparator: separator character for single entry value
Returns	A numeric value that contains the number of entries generated in the taArray .

Description	<p>Loads name-value pairs from the source file (tSrcFile) or database table (SrcLog. tSrcExt) to the elements in the text array (taArray).</p> <p>The content of the file is loaded line by line. The first token on the line is assumed to be the name, followed by the separator, which in turn is immediately followed by the value. The default separator is '='.</p> <p>Load Mode</p> <ul style="list-style-type: none"> • 0: Creates a new table. May create multivalues. Used as a default. • 1: Overwrites existing values. The values already in the table are left intact if the specified file does not contain new values for them; otherwise they are replaced with the new values. Only the last value for each new name is used, since no multivalue fields are generated. • 2: Appends to existing values. This may turn some single values to multivalue fields. • 3: Adds new entries. Only new values are read in, so the existing values will not change, but there may be additions. • 4: The same as load mode 1, but this mode deletes all empty parameters merged from the file.
Note	<p>Parameters at the Beginning of a Data File:</p> <p>If the data file contains parameters followed by other data, parameters can be loaded from the beginning of the file with the load function. Parameters and data must be separated with a line that contains only a single separator character, such as the equal sign (=). After a successful load function call, the file is left open and the file pointer points to the beginning of the data.</p>

Example:

Example file "acronyms.txt":

- AA=Amazing Acronyms
- BB=Baked Beans
- BB=Bank Bill

If the file (or table) contains more than one line with the same name, the result depends on the loadmode being used. With load modes other than 1, all these values are loaded to the same table entry, separated by the tMultisep. The default value for tMultisep is ':'. Thus the command

```
load (0, "acronyms.txt", taFORM)
```

or

```
load ("acronyms.txt", taFORM)
```

would result in:

```
taFORM["AA"] contains "Amazing Acronyms"
taForm["BB"] contains "Baked Beans:Bank Bill"
```

On the other hand,

```
load (1, "acronyms.txt", taFORM)
```

would result in:

```
taFORM["AA"] contains "Amazing Acronyms"
taForm["BB"] contains "Bank Bill"
```

Effect of different load modes, "load.rte"

```
! A simple RTE-program to demonstrate different load modes

begin

    ! Set the load mode to the one given in command ! line,
    ! or use 0 (zero) as a default if none given

    if ARGV <> 1 then
        nMode := 0
    else
        nMode := number(ARGV[1])
    endif

    ! Fill in some data
    taFORM["ONE"] := "First line"
    taFORM["TWO"] := "Second line"

    ! Print out the original contents of the table
    print ("Array taFORM contains: (before loading)",NL)
    print(taFORM)
    print(NL)

    ! Load the file "table.txt" into table taFORM
    load(nMode, "table.txt", taFORM)

    ! Print out the new contents of the table
    print("Array taFORM contains: (after loading,\ mode=",nMode,")",NL)
    print(taFORM)

endbegin
```

And a sample file, "table.txt":

- ONE=Number one
- TWO=Number two
- And sample usage and output: \$ load 0

Array taFORM contains (before loading):

- ONE=First line
- TWO=Second line

Array taFORM contains (after loading, mode=0):

- ONE=Number one
- TWO=Number two
- \$ load 1

Array taFORM contains (after loading, mode=1):

- ONE=Number one
- TWO=Number two
- \$ load 2

Array taFORM contains (before loading):

- ONE=First line
- TWO=Second line

Array taFORM contains (after loading, mode=2):

- ONE=First line
- ONE=Number one
- TWO=Second line
- TWO=Number two
- \$ load 3

Array taFORM contains (before loading):

- ONE=First line
- TWO=Second line

Array taFORM contains (after loading, mode=3):

- ONE=First line
- TWO=Second line
- \$ load 4

Array taFORM contains: (before loading)

- ONE=First line
- TWO=Second line

Array taFORM contains: (after loading, mode=4)

- ONE=Number one
- TWO=Number two

8.3.17 log

Usage	log (print list)
Arguments	print list : see "5.6 - Print List" on page 76, for a print list description.
Returns	Nothing
Description	Writes the print list to the logging stream
See also	print

Example:

```
! Print a log entry.
begin
    redirect(LOGGING, build(time("%y%m%d"), ".log"))
    log("Running ", ARGV[0], NL)
    log("Run time ", time(), NL)
endbegin
```

8.3.18 new

Usage	new (tDatabase)
Arguments	tDatabase: name of the database
Returns	Database entry
Description	Creates an empty entry to the named database.
See also	find

Example:

```
! new creates an empty entry to the database.
! No values for fields are set except CTIME (creation time)
! if addprog is defined for database it is called.

base "example.cfg"      ENTRY

begin
    ! Database is located in home directory
    tBASE := build($HOME, "/example")

    ! Create new entry and set default values
    ENTRY := new(tBASE)
```

```

    if valid(ENTRY) then
        ENTRY.NAME := "Nick Noname"
        ENTRY.ADDRESS := "Quezon City"
        ENTRY.AGE := 999
    endif
endbegin

```

8.3.19 number

Usage	number (tNumText)
Arguments	tNumText: text string
Returns	A numeric value that contains the converted value.
Description	This function converts the argument tNumText to its numeric representative. If the text does not represent a valid number, the return value is zero and an error message is written to the logging stream.
See also	build

Example:

```

! Calculate the sum of numbers given on input lines
line '[0-9]*'
    nGot := number(pick(1,1,EOL))
    nSum := nSum + nGot
    print("Number=", nGot, " Sum=", nSum, NL)
endline

```

8.3.20 peel

Usage	peel (tOriginal, tExtras)
Arguments	<ul style="list-style-type: none"> tOriginal: original text string tExtras: text string containing the extraneous characters
Returns	A text string that contains the peeled string.
Description	Peels off all characters included in tExtras from the beginning and end of the tOriginal and returns the resulting text. tOriginal always remains unaltered. The order of characters in tExtras is insignificant and the characters are not combined to form strings. In other words, peeling is done on a character-by-character basis and no strings can be specified. The peeling stops in one direction as soon as a character appears that is not included in tExtras .
See also	replace

Examples:

```
! Example 1
tResult := peel (" Sample text ", " ")
! tResult contains as its value "Sample text".
```

```
! Example 2
! peeling spaces and tabs

default
    print(peel(pick(1,1,EOL), " "), NL)
enddefault
```

8.3.21 pick

Usage	pick (nLine, nPosition, nLength)
Arguments	<ul style="list-style-type: none"> • nLine: line position • nPosition: column position • nLength: number of characters to be picked
Returns	A text string that contains the pick value.
Description	nLine specifies the line relative to the top of the current window in the input stream. nPosition gives the position on that line and nLength the number of characters to be picked. The last argument can also be EOL, which directs the picking to retrieve all characters up to the end of the line.
See also	read

Examples:

```
! Example 1

line (1:"SAMPLE")
    tText1 := pick (1, 8, EOL)
    tText2 := pick (2, 1, 10)
endline

! Example 1 with data lines:
! SAMPLE PICKING
! A SECOND LINE
! tText1 contains "PICKING"
! tText2 contains "A SECOND L"
```

```
! Example 2
! example of pick function run executable with source as input file
! SAMPLE PICKING
! SECOND SAMPLE LINE

line (1:"SAMPLE")
    tText1 := pick(1, 9, EOL)
    tText2 := pick(2, 8, 7)
    print("HIT - ", tText1, tText2, NL)
endline
```

8.3.22 print

Usage	print (print list)
Arguments	print list : see "5.6 - Print List" on page 76, for a print list description.
Returns	Nothing
Description	Prints the print list to standard output.

Example:

```
! print can take any kind of arguments including
! numeric and ascii variables and tables...
! Print example

begin
    nX := 1
    nY := 2.37
    tC := "ASCII"
    print("This appears on the stdout", NL)
    print("Today is ", time(), NL)
    print("Value of variable nX is ", nX, NL)
    print("Value of variable nY is ", nY, NL)
    print("Value of variable nY is ", nY:6, NL)
    print("Variable tC (tC) '", tC, "'", NL)
    print("Variable tC (tC:3) '", tC:3, "'", NL)
    print("Variable tC (tC:7) '", tC:7, "'", NL)
    print("This goes to file '/tmp/tmp'", NL) > "/tmp/tmp"
    print("This is echoed by /bin/more", NL) > "|/bin/more"
    redirect(OUTPUT, "/tmp/tmp")
    print("Now every printed line ")
    print("goes to file /tmp/tmp", NL)
endbegin
```

8.3.23 put

Usage	put ([taMatrix,] nLine, nPos, print item)
Arguments	<ul style="list-style-type: none"> • taMatrix: output buffer • nLine: line in the output buffer • nPos: column of the output buffer • print item: See "5.7 - Print Items" on page 77, for a print item description.
Returns	Nothing
Description	<p>The put function is used when buffered output is needed. The function prints its print item to the given buffer or to the default output buffer named BUFFER. The print item is put in the print buffer position assigned with nLine and nPos. nLine is the line number in the output buffer and nPos is the column in the output buffer.</p> <p>If there are undefined places in the output buffer, they are filled with the fill character named FILL_CHAR, which must be set before put is used for the first time for the buffer.</p> <p>If a formatted print item is used, the fill character for the print formats is always a space, regardless of the FILL_CHAR value.</p>
See also	flush

Example:

```
! If a fill character is used it must be set before the first call to put function.
! Fill character is used in any output location that is not explicitly set.
! Build output in unforeseen order

begin
  ! If used, fill char must be set before printing
  FILL_CHAR := "+"
  put(3, 1, "first")
  put(2, 21, "second")

  ! use 8 characters wide field for "third"
  put(1, 21, "third":8)
  put(2, 11, "fourth")
  put(2, 1, "fifth")
  put(1, 1, "sixth")
  put(3, 21, "seventh")
  put(1, 11, "eighth")
  put(3, 11, "ninth")

  ! Print every line exactly 30 characters long
  flush(30, 30, NL)
endbegin
```

8.3.24 read

Usage	read (tFile)
Arguments	tFile: file name of the file to be read from
Returns	A text string that contains the read value.
Description	The read function reads a line from a file or from a process. The function returns the line just read (or EOF if there are no more lines). The file from which lines are read must be closed before it can be reread. It is also good practice to close all processes that are read from.
See also	<ul style="list-style-type: none"> load pick

Example:

```
! List of registered users in this machine

begin
    tFile := "/etc/passwd"
    print("List of registered users:", NL)
    tLine := read(tFile)
    while tLine <> EOF do
        print(substr(tLine, 1, index(tLine, ":") - 1), NL)
        tLine := read(tFile)
    endwhile
    close(tFile)
endbegin

end
    tFile := "|uname -a"
    tLine := read(tFile)
    close(tFile)
    print("in machine ", tLine, NL)
endend
```

8.3.25 redirect

Usage	redirect (stream, tFile)
Arguments	<ul style="list-style-type: none"> stream: name of the stream tFile: file name of the file to be opened
Returns	Nothing

Description	There are three standard I/O devices named INPUT, OUTPUT, and LOGGING. By default, they refer to stdin , stdout , and stderr , respectively. Any of them can be redirected to any file or process, but after redirection, there is no way to return them to their default values. Note that if you start a new process after redirection, it will also inherit the redirected I/O devices.
-------------	---

Example:

```
! Redirection example
! Prints users with initial letter 'r'
begin
    print("Output will be in file /tmp/tmp", NL)
    redirect(INPUT, "/etc/passwd")
    redirect(OUTPUT, "/tmp/tmp")
    redirect(LOGGING, build(time("%y%m%d"), ".log"))
    log("Running ", ARGV[0], NL)
    log("Run time ", time(), NL)
    print("Usernames with initial letter 'r':", NL)
endbegin

line (1:"r")
    print(pick(1, 1, EOL), NL)
endline
```

8.3.26 remove

Usage	remove (tFile) remove (log entry) remove (MESSAGE) remove (taArray) remove (naArray) remove (baArray)
Arguments	<ul style="list-style-type: none"> • tFile: file name of the file to be removed • log entry: log entry handle of the log entry to be removed • MESSAGE: MESSAGE to be removed • taArray: text array to be removed • naArray: numeric array to be removed • baArray: Boolean array to be removed
Returns	The Boolean value TRUE if the remove is successful; otherwise FALSE.
Description	The remove function removes the given object, which can be a file, log entry, MESSAGE, text array, Boolean array, or numeric array. If the removal succeeds, the Boolean value TRUE is returned; otherwise, FALSE is returned and an error message is written to the logging stream.

Example:

```
! Print /etc/passwd
begin
    ! Load userlines in username indexed table print table in file
    ! print file line by line
    tWorkFile := "/tmp/tmp"
    load("/etc/passwd", taUsers, "@", ":")
    print(taUsers) > tWorkFile
    close(tWorkFile)
    remove(taUsers)
    redirect(INPUT, tWorkFile)
endbegin

default
    print(pick(1, 1, EOL), NL)
enddefault

end
    remove(tWorkFile)
endend
```

8.3.27 rename

Usage	rename (tOldname, tNewname)
Arguments	<ul style="list-style-type: none"> • tOldname: file name of the file to be renamed • tNewname: new file name for the file
Returns	The Boolean value TRUE if the rename is successful; otherwise FALSE.
Description	This function renames the file tOldname to tNewname . If the rename is successful, the Boolean value TRUE is returned; otherwise, the function returns FALSE and writes an error message to the logging stream. Rename works by first creating a new link to the file's inode, and then removing the old name from the inode. This allows the system to give more detailed reports if an error occurs.

Example:

```
! rename example and verification
begin
    tFile1 := "/tmp/tmp"
    tFile2 := "/tmp/tmp.tmp"
    system(build("touch ", tFile1))
    system("ls -al /tmp/tmp*")
    rename(tFile1, tFile2)
    system("ls -al /tmp/tmp*")
    remove(tFile2)
endbegin
```

8.3.28 replace

Usage	replace (tOriginal, tRemoved, tReplacements)
Arguments	<ul style="list-style-type: none"> • tOriginal: original text string • tRemoved: text string containing characters to be removed • tReplacements: text string containing substituting characters
Returns	A text string that contains the replaced string.
Description	<p>This function replaces in the text tOriginal all the characters in tRemoved that have corresponding characters in tReplacements, and returns the resulting text. tOriginal is unaltered.</p> <p>tOriginal is scanned for all characters in tRemoved individually, so no strings can be specified to be replaced. As a character is found, it is replaced with a character from the same position in tReplacements. If there is no character at that position, then the character specified in tRemoved is removed from tOriginal.</p>
See also	<ul style="list-style-type: none"> • peel • strip

Examples:

```
! Example 1
tResult := replace (" Sample text ", " e", ".E")
! tResult contains "..SampleE.tExt..".
! All spaces were replaced by a period and all lower case e's with a capital E.
```

```
! Example 2
tResult := replace (" Sample text ", " et", ".E")
! tResult contains "..SampleE.Ex..".
! All spaces were replaced by a period
! and all lower case e's with a capital E
! and all lower case t's were removed
! since there was no character at the third position in the tReplacements (".E").
```

```
! Example 3
! Prints all lines in /etc/passwd starting with 'r'
! Some changes are made...

begin
    redirect(INPUT, "/etc/passwd")
endbegin
```

```
line (1:"r")
  tLine := pick(1, 1, 30)
  print(tLine, NL)
  print(replace(tLine, "r:/", "R \\"), NL, NL)
endline
```

8.3.29 spawn

Usage	spawn (tCommand, taArgs) spawn (tCommand [,tArg1 [,tArg2 [, ...[, tArgN]]]])
Arguments	<ul style="list-style-type: none"> • tCommand: command to be spawned • taArgs: arguments for the command in table form • tArg1, tArg2, ..., tArgN: arguments for the command in list form
Returns	A numeric value that contains the exit code from the spawned process.
Description	The function spawn creates a new child process. The process that calls spawn waits until the spawned process returns. Spawn returns the exit code from the spawned process. Note that the spawned process inherits the environment of the calling process.
See also	<ul style="list-style-type: none"> • background • exec • system

Example:

```
! spawns command 'ls -la [param [param...]]'

begin
  taArgs[0] := "-la"
  nParam := 1
  while ARGV[nParam] <> EMPTY do
    taArgs[nParam] := ARGV[nParam]
    nParam++
  endwhile
  spawn("ls", taArgs)
  print("This will be printed", NL)
endbegin

end
  spawn("not.exist", "dummy", "parameter")
  print("This will be printed, too", NL)
endend
```

8.3.30 split

Usage	split (tSource, taArray, tSeparator) split (tSource, taArray, regular expression)
Arguments	<ul style="list-style-type: none"> • tSource: text string to be split • taArray: resulting text array • tSeparator: text string containing separator character • regular expression: regular expression reguused in splitting
Returns	A numeric value that contains the number of items created in taArray.
Description	<p>This function splits tSource into substrings that are cut wherever the separator character tSeparator exists. The resulting substrings are put into the taArray text array, and the number of substrings is returned.</p> <p>If a regular expression is used instead of the separator character, tSource is split according to the named sub-expressions in the regular expression if the regular expression matches tSource. The name of the sub-expression must be a number from 0 to 9. taArray is filled such that sub- expression names are used as indexes to the taArray.</p>
Note	Regular expressions are not available in Windows NT.
See also	substr

Example:

```
! Example of split
begin
    tText1 := "one,two,three,four,five,six,seven"
    tText2 := "user@machine"
    nNumbers := split(tText1, taNumbers, ",")
    nFields := split(tText2, taAddr, '^([^\@]+)$0@(.+)$1')
    print("There are ", nNumbers, " numbers:", NL)
    print(taNumbers)
    print("Addr should have 2 fields (", nFields, ")", NL)
    print(taAddr)
endbegin
```

8.3.31 strip

Usage	strip (tOriginal, tExtras)
Arguments	<ul style="list-style-type: none"> • toriginal: original text string • tExtras: text string containing characters to be removed
Returns	The text string that results from the stripping operation.

Description	This function strips away from tOriginal all the characters contained in tExtras and returns the resulting text. tOriginal is unchanged. The order of characters in tExtras does not matter, and the characters are not combined to form strings. In other words, stripping is done on character-by- character basis and no strings can be specified in tExtras .
See also	<ul style="list-style-type: none"> • peel • replace

Example:

```
! Example 1
tResult := strip (" Sample text ", " ")
! tResult contains "Sampletext" as its value
! since all spaces have been stripped away.
```

```
! Example 2
begin
    redirect(INPUT, "/etc/passwd")
endbegin

line (1:"r")
    ! strip all r's, : 's and / 's print both stripped and unstrapped lines
    tline := pick(1, 1, 30)
    print(tline, NL)
    print(strip(tline, "r:/"), NL, NL)
endline
```

8.3.32 substr

Usage	substr (tOriginal, nPosition, nLength)
Arguments	<ul style="list-style-type: none"> • tOriginal: original text string • nPosition: start position of the substring • nLength: number of characters in the substring
Returns	A text value that contains the extracted substring.
Description	From the text tOriginal , this function extracts a substring that starts from nPosition and is nLength long.
See also	index

Example:

```
! Substr sample
begin
    tText := "Sample string for examples"
    ! This will print "Sampling"
    print(substr(tText, 1, 5))
    print(substr(tText, 11, 3), NL)
endbegin
```

8.3.33 system

Usage	system (tCommandLine)
Arguments	tCommandLine: command line to be executed
Returns	A numeric value, which is the exit code of the executed shell.
Description	The system causes the tCommandLine to be given to the shell as input, as if the tCommandLine had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.
See also	<ul style="list-style-type: none"> • background • exec • spawn

Example:

```
! runs system with command 'ls -la [param [param...]]
begin
    tCommand := "ls -la"
    nParam := 1
    while ARGV[nParam] <> EMPTY do
        tCommand := build(tCommand, " ", ARGV[nParam])
        nParam++
    endwhile
    print("This will be printed before", NL)
    system(tCommand)
    print("This will be printed after", NL)
endbegin
```

8.3.34 time

Usage	time ([TIME] [,tFormat])
Arguments	<ul style="list-style-type: none"> • TIME: time specifier • tFormat: time print format
Returns	A text string that contains the specified time.
Description	<p>Time provides access to the system time and the time values in the database and file system.*</p> <p>When called with no arguments, time returns the current system time in the default format (dd.mm.yyyy HH:MM).</p> <p>When tFormat is provided, the current system time is returned in the specified format.</p> <ul style="list-style-type: none"> • time () • time (tFormat) • time (TIME) • time (TIME, format) <p>The format string can contain time components and regular characters. The time components are marked with a percent sign.</p> <p>The time format fields are:</p> <ul style="list-style-type: none"> • %T: time in AM/PM notation (HH:MM AM PM) • %t: time in AM/PM notation (HH:MM:SS AM PM) • %d: day of the month • %m: month • %y: year in two digits (...98, 99, 00, 01) • %Y: absolute year • %H: hour • %M: minute • %S: seconds • %W: day of the week in long format • %w: day of the week in abbreviated format • %k: week of the year (1 - 53) • %j: day of the year (1 - 366) • %a: absolute value, used in comparisons • %eM: elapsed minutes to current system time • %eH: elapsed hours • %eW: elapsed weeks <p>A negative value in elapsed time indicates that the given time is in the future.</p> <p>The time format specifiers are today, yesterday, tomorrow, now, and none. Any of these specifiers except "none" can be followed by the + or - operator and a formatted value.</p>

Example 1:

```
! time() example
begin
    print ("a"), NL
    print("Time is now ( ) ", time(), NL)
    print(" in AM/PM notation (T) ", time("%T"), NL)
    print(" in AM/PM notation (%t) ", time("%t"), NL)
    print("Minutes (%M) ", time("%M"), NL)
    print("Hours (%H) ", time("%H"), NL)
    print("Seconds (%S) ", time("%S"), NL)
    print("Day of the month (%d) ", time("%d"), NL)
    print("Month of the year (%m) ", time("%m"), NL)
    print("Short year (%y) ", time("%y"), NL)
    print("=> year 2001 (short) (%y) ", \
        time ("31.1.2001", "%y"), NL)
    print("Day of week (long) (%W) ", time("%W"), NL)
    print("Day of week (short) (%w) ", time("%w"), NL)
    print("Week of the year (%k) ", time("%k"), NL)
    print("Day of the year (%j) ", time("%j"), NL)
    print("=> days in 1999 (%j) ", \
        time("31.12.1999", "%j"), NL)
    print("=> days in 2000 (%j) ", \
        time("12/31/00", "%j"), NL)
    print("Absolute value (%a) ", time("%a"), NL)
    print("Elapsed minutes (%eM) ", time("%eM"), NL)
    print("Elapsed hours (eH) ", time("%eH"), NL)
    print("Elapsed days (%ed) ", time("%ed"), NL)
    print("Elapsed weeks (%ew) ", time("%ew"), NL)
    print("b"), NL)
    print("After a minute it will be ", \
        time("now + 1M", "%d.%m.%y %t"), NL)
    print("After an hour it will be ", \
        time("now + 1H", "%d.%m.%y %t"), NL)
    print("After a day it will be ", \
        time("now + 1d", "%d.%m.%y %t"), NL)
    print("After a week it will be ", \
        time("now + 1w", "%d.%m.%y %t"), NL)
    print("c"), NL)
    print("Date 20000229 printed ", \
        time("20000229", "%d.%m.%Y"), NL)
    print("Date 02/29/00 printed ", \
        time("02/29/00", "%d.%m.%Y"), NL)
    print("Date 29.2.2000 printed ", \
        time("29.2.2000", "%d.%m.%Y"), NL)
    print ("d"), NL
```



```
! Different outputs from same input format
print("Date 20000229 printed ", \
      time("20000229", "%d.%m.%Y"), NL)
print("Date 20000229 printed ", \
      time("20000229", "%m/%d/%Y"), NL)
print("Date 20000229 printed ", \
      time("20000229", "%d%m%Y"), NL)
print (NL)

! NOTE: Year 2001 is not a leap year, so there is no February 29th.
print("Date 29.2.2001 printed ", \
      time("29.2.2001", "%d.%m.%y"), NL)

endbegin
```

produces the following output:

a)

Time is now	()	10.01.2000 10:03
in AM/PM notation	(%T)	10:03 AM
in AM/PM notation	(%t)	10:03:20 AM
Minutes	(%M)	03
Hours	(%H)	10
Seconds	(%S)	20
Day of the month	(%d)	10
Month of the year	(%m)	01
Short year	(%y)	00
=> year 2001 (short)	(%y)	01
Day of week (long)	(%W)	Monday
Day of week (short)	(%w)	Mon
Week of the year	(%k)	2
Day of the year	(%j)	10
=> days in 1999	(%j)	365
=> days in 2000	(%j)	366
Absolute value	(%a)	947491400
Elapsed minutes	(%eM)	0
Elapsed hours	(%eH)	0
Elapsed days	(%ed)	0
Elapsed weeks	(%ew)	0

b)

After a minute it will be	10.01.00 10:04:20 AM
After an hour it will be	10.01.00 11:03:20 AM
After a day it will be	11.01.00 10:03:20 AM

After a week it will be	17.01.00 10:03:20 AM
c)	
Date 20000229 printed	29.02.2000
Date 02/29/00 printed	29.02.2000
Date 29.2.2000 printed	29.02.2000
d)	
Date 20000229 printed	29.02.2000
Date 20000229 printed	02/29/2000
Date 20000229 printed	29022000
Date 29.2.2001 printed	01.03.01

Example 2:

```
!
! Comparing time-fields from system logs with given range.
!
! Entries older than ARGV[1] are printed as old,
! entries newer than ARGV[2] are printed as new.
!
base "syslog.cfg" LOG
...
  nOLD := number(time(ARGV[1], "%Y%m%d"))
  nNEW := number(time(ARGV[2], "%Y%m%d"))
  while LOG in (sSYSLOG) then
    if (number(time(LOG.CREATED, "%Y%m%d")) < nOLD) then
      print("old ")
    else
      if (number(time(LOG.CREATED, "%Y%m%d")) > nNEW) then
        print("new ")
      else
        print(" ")
      endif
    endif
    print(LOG.INDEX, " ", time(LOG.CREATED), " ", \LOG.INFO, NL)
  endwhile
```

8.3.35 tolower

Usage	tolower (tOriginal)
Arguments	tOriginal : text string to be converted
Returns	A text string that contains the mapped string.
Description	This function converts all upper case letters in tOriginal to lower case and returns the result. tOriginal remains unchanged. The conversion takes place according to the currently loaded character set.
See also	toupper

Example:

```
! Converts all input to lowercase characters
default
    print(tolower(pick(1, 1, EOL)), NL)
enddefault
```

8.3.36 toupper

Usage	toupper (tOriginal)
Arguments	tOriginal : text string to be converted
Returns	A text string that contains the mapped string.
Description	This function converts all lower case letters in tOriginal to upper case letters and returns the result. tOriginal remains unchanged. The conversion takes place according to the currently loaded character set.
See also	tolower

Example:

```
! Converts all input to uppercase characters
default
    print(toupper(pick(1, 1, EOL)), NL)
enddefault
```

8.3.37 valid

Usage	valid (MESSAGE) valid (SEGMENT) valid (GROUP <group specifier>) valid (<log entry>) valid (<objecttype>, <text>)
Arguments	<ul style="list-style-type: none"> • MESSAGE: check validity of message • SEGMENT: check validity of current segment • GROUP <group>: check validity of a given group • <log entry>: check validity of a given log entry • <objecttype>: object type to check against • <text>: text to be checked
Returns	The Boolean value TRUE if valid; otherwise FALSE.
Description	<p>Valid checks the validity of the item against the rules set for each of its arguments.</p> <p>For MESSAGE and SEGMENT, and group specifier, various syntax and message definition related checks are made.</p> <p>For log entries, validity means the existence of the log entry.</p> <p>If object is given, the tText is validated against the rules set for that particular object class. The object can be one of the following: ALPHA, ALPHANUM, DATE, DECIMAL, ID, INTEGER, NUMERIC, STRING, TIME.</p>

Example:

```

segment UNH
  if valid (MESSAGE) = FALSE then
    edierrordump(MESSAGE)
  endif
  ...
Endsegment
Group
  If valid (Group g25) = FALSE then
    endif
  
```



Chapter 9 - Appendices

Appendix A: Identifier Types

Appendix B: Regular Expressions

Appendix C: Message Storage Format

Appendix D: Character Sets

Appendix E: Code Conversion in RTE

Appendix F: Advanced Utilities

Appendix G: RTE SQL Access Library

Appendix I: RTE LDAP Library

9.1 Appendix A: Identifier Types

9.1.1 Overview

RTE does not contain declarations for identifiers. The names of the identifiers are used to distinguish the different types.

9.1.2 Basic Types

All user-defined data types and other objects are identified with the first one or two letters of the identifier name. This convention obviates the need for variable definitions. The letter following the type tag must be an upper case letter. Here is a list of all the identifier types.

- e Data element. These are always type text. For complete information about element naming, see "Chapter 4 - EDI translation" on page 49.
- g Group symbol. These can be used only to specify the position of a segment in a message.
- m Message variable. These are type text. Variables that are always available are mMESSAGE, mVERSION, mRELEASE, and mAGENCY. Other variables can be defined in the message description file. These behave like constants and you cannot assign values to them.
- c Static counter. These are numeric values that can be referenced and set to a value. The identifier name after the initial "c" is a file name in the directory \$HOME/.counters. The numeric content of that file is the next value received when the counter is referenced. The value is incremented each time the counter is referenced and a locking mechanism is used to guarantee that no two references receive the same value.

To assign a value to a counter, (and reset it if needed) : cMYCOUNTER := <anynumber>



Counters are not automatically reset when they reach their limit. You need to reset the counter in the RTE.

- s Environment variable. All those Unix shell variables (also called environment variables) that were set when the program started can be referenced. The identifier name after the initial "s" must be the exact name of the shell variable. You cannot set new shell variables or change the existing values.
- o Command line option. These are Boolean values. An RTE program considers all upper case flags (a single letter preceded by a minus sign) as user options and their values can be referenced with these identifiers. An option identifier name consists of the initial "o" followed by a single upper case letter. If the option was specified on the command line, the option identifier is TRUE; otherwise it is FALSE.
- p Parameter. These are text values. Initial parameters can be specified in a parameter file or on the command line. You can modify existing parameters and create new ones. For more details, see "Chapter 7 - Running RTE Programs" on page 88.
- f File. These are special types of objects that are used to reference various attributes for files. For details, see "5.10 - File Variables" on page 80.
- t Text variable. These can contain text of unlimited (for all practical purposes) length.
- n Numeric variable. The content can be an integer or a double precision floating point number.

- b** Boolean variable. This can contain only the values TRUE or FALSE.
- ta** An array containing text items. The array can contain an unlimited number of text items and the index can be either text or an integer.
- na** An array containing numeric items. The array can contain an unlimited number of numeric items and the index can be either text or an integer.
- ba** An array containing Boolean items. The array can contain an unlimited number of Boolean values (TRUE or FALSE) and the index can be either text or an integer.
- tf** Function returning text.
- nf** Function returning a number.
- bf** Function returning a Boolean value.

Local variables

To avoid name conflicts, it is recommended to use local variables inside functions. This will result in a better memory handling, and allow the use of recursivity.

Local variables are marked as local by using the letter 'l' in the "type" part of a variable name. The idea is just to extend the type part (n,t,b,na,ta,b,f) of the naming convention to be able to define local variables.

Some examples:

- blBoolean,
- nlNumeric,
- tlText,
- balBooleanTable,
- nalNumericTable,
- talTextTable,
- flFileVar.

The scope of local variables is limited to blocs function endfunction.

9.1.3 Special Types

9.1.3.1 Database Entry

Database entries must be defined before any statement lists. The identifier can contain upper case and lower case letters, digits, and an underscore. The first character cannot be a digit. The definition is required because the RTE compiler must know the types and lengths of the database fields at compile time. For a complete description, see "Chapter 6 -Database Interface" on page 82.

9.1.3.2 TIME

This type can be used as an argument to the time function. The database and file system interfaces contain this type. Assignments to the time fields of the data base entries are made in text format.

There is no variable type TIME.

9.2 Appendix B: Regular Expressions

9.2.1 Regular Expression Usage

RTE supports a limited form of regular expression notation. Regular expressions are used in line statements to specify line matching rules and in split functions to specify the portions of a text expression that are to be extracted.

A regular expression must be enclosed in quotation (' ') marks so that RTE will handle them correctly. Regular expressions are constant expressions (not text strings), and they cannot be assigned to any variable.

Regular expressions in RTE are defined in a similar way as in the Unix command editor.



Regular expressions can be used in Unix only.

9.2.2 Regular Expression Definition

A regular expression (RE) specifies a set of character strings. A member of this set of strings is said to be matched by the RE. The REs allowed by RTE are constructed as follows.

1. The following one-character REs match a single character:
 - 1.1. An ordinary character (not one of those discussed in 1.2 below) is a one-character RE that matches itself.
 - 1.2. A backslash (\) followed by any special character is a one-character RE that matches the special character itself. The special characters are:
 - . * [and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets ([]); see 4 below).
 - ^ (caret or circumflex), which is special at the beginning of an entire RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 below).
 - c. \$ (dollar sign), which is special at the end of an entire RE (see 3.2 below).
 - d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the g command, below.)
 - 1.3. A period (.) is a one-character RE that matches any character except newline.
 - 1.4. A non-empty string of characters enclosed in square brackets ([]) is a one-character RE that matches any one character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character except newline and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (-) can be used to indicate a range of consecutive ASCII characters; for example, [0-9] is equivalent to [0123456789]. The - loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); for example, []a-f] matches either a right square bracket (]) or one of the letters a through f, inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

2. The following rules can be used to construct REs from one-character REs:
 - 2.1. A one-character RE is an RE that matches whatever the one-character RE matches.
 - 2.2. A one-character RE followed by an asterisk (*) is an RE that matches zero or more occurrences of the one-character RE. If there is any choice, the longest leftmost string that permits a match is chosen.
 - 2.3. A one-character RE followed by `\{m\}`, `\{m,\}`, or `\{m,n\}` is an RE that matches a range of occurrences of the one-character RE. The values of m and n must be non-negative integers less than 256; `\{m\}` matches exactly m occurrences; `\{m,\}` matches at least m occurrences; `\{m,n\}` matches any number of occurrences between m and n, inclusive. Whenever a choice exists, the RE matches as many occurrences as possible.
 - 2.4. The concatenation of REs is an RE that matches the concatenation of the strings matched by each component of the RE.
 - 2.5. An RE enclosed between the character sequences `\<` and `\>` is an RE that matches whatever the unadorned RE matches.
 - 2.6. The expression `\n` matches the same string of characters as was matched by an expression enclosed between `\<` and `\>` earlier in the same RE. Here, n is a digit; the subexpression specified is that beginning with the nth occurrence of `\<`, counting from the left. For example, the expression `^\<.*\>\1$` matches a line consisting of two repeated appearances of the same string.
3. Finally, an entire RE may be constrained to match only an initial segment or final segment of a line (or both).
 - 3.1. A circumflex (^) at the beginning of an entire RE constrains that RE to match an initial segment of a line.
 - 3.2. A dollar sign (\$) at the end of an entire RE constrains that RE to match a final segment of a line.

The construction `^ entire RE$` constrains the entire RE to match the entire line.

9.2.2.1 Examples

Using REs with the Line Statement

To match a line containing product codes in the form shown below (two digits, one upper case letter, three digits and one lower case letter):

- 12M456I
- 45J897k

use the following regular expression:

```
line '[0-9]{2}[A-Z][0-9]{3}[a-z]'
```

To match a line containing just a tag in the form shown below (the text string TAG at the beginning of the line, followed by a numeric portion containing at least 1 and up to 4 digits, followed by the # mark and nothing other than a newline character):

- TAG145#
- TAG12#

use the following line statement:

```
line '^TAG[0-9]{1,4}#$',
```

Using REs with the Split Function

The split function is a string function that can be used to split a string to substrings. Normal use would be something like:

```
split ("1,2,3,4", taNumbers, ",")
```

If a string cannot be split with just the separator character, a regular expression can be used instead of the separator character.

To split the product code in the earlier example into its components, we can use a regular expression, as shown below:

```
split (tCode, taCode, \
'([0-9]{2})$0([A-Z])$1([0-9]{3})$2([a-k]) $3')
```

After this function has been executed with the example code shown above, the text table taCode contains the following strings:

```
taCode[0] = "12"
taCode[1] = "M"
taCode[2] = "456"
taCode[3] = "1"
```

The content of the taCode table is formed such that the string matched by the RE is split into substrings, formed of the substrings that match the RE enclosed in () parentheses. The notation \$n in the RE assigns the substring to the nth component in the taCode table (n can be from 0 to 9).

9.3 Appendix C: Message Storage Format

9.3.1 Message World

The messages used in TradeXpress are described in regular text files in a human readable format. The message definition file contains only a list of segments and groups in the message; the content for each segment is specified in a separate file. This provides for consistency and allows for a more compact presentation.

A message description is a starting point for all the work that is required to generate an EDI message from application data, and vice-versa. A message description defines what is included in a message, what is mandatory and what is conditional, and how many times items are allowed to repeat. A message description is not tied to any application area, translation type, or direction. Instead, it defines the framework for later, more detailed specifications.

The TradeXpress Message Manager obeys all the definitions set in the message description. This means, for example, that you cannot use the Message Manager to change a mandatory segment to a conditional one. If you must actually alter a message description, rather than just defining a subset of it, you must edit the description manually.

It is possible to define a completely new message from scratch, or to change existing ones. This is technically a straightforward task, but the use of standard messages is strongly recommended.

Message descriptions usually represent standard messages that are distributed by a controlling agency, such as the UN or ANSI. Standard messages often contain only a few mandatory segments and a wide selection of conditional segments, allowing various business areas and interest groups to tailor subsets of the message to suit their specific needs.

9.3.2 Message Description Files

The message description file contains identifiers, a segment list, and a section that describes the use of the segments. The rest of this appendix explains these items in the order in which they appear in the file.

9.3.2.1 Identifiers

There are a few mandatory identifiers that are presented as name-value pairs. Since the file name by itself is not sufficient to uniquely identify the message, the identification strings are stored in predefined identifiers:

Identifier	Description
MESSAGE	The name of the message, e.g. "CUSRES."
VERSION	The version number, e.g. "2."
RELEASE	Release for the message, e.g. "912."
AGENCY	Controlling agency for the message, e.g. "UN."

The name and value are separated by an equal sign.

Users can also include free form comments and set their own identifiers. Comments are written to a line that starts after the keyword COMMENT and expands an unlimited number of lines up to a line that

starts with the keyword ENDCOMMENT. User-defined identifiers are in name-value-pair format. Mandatory and user-defined identifiers can be used in RTE programs.

The file can also contain explanatory comments, which start from a “#” character and continue to the end of the line. As the message is read in, these comments are ignored. They are not usable in RTE and cannot be seen in the Message Manager.

9.3.2.2 Message Structure

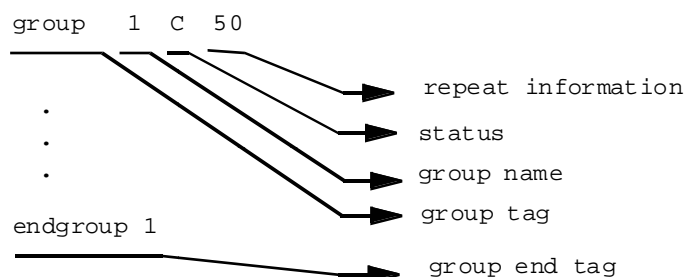
The message structure is described by specifying all included segments and groups on separate lines in the order in which they appear in the message. These lines start immediately following the line that contains the keyword SEGMENTS.

That line can optionally contain a directory where the segment files reside. If there is no directory name, then the segments are assumed to be in directory “segs” relative to the directory where the message description resides.

9.3.2.3 Segment Line

name	Gives the unique name for the segment. This name is used as such in the RTE files.
status	The status can be either mandatory (M), conditional (C), or floating (F). EDIFACT messages contain only mandatory and conditional segments, whereas ANSI12 messages can contain floating segments as well.
repeat information	Specifies how many times the segment can or must repeat in the message.

Group Line



Each segment group is started with the keyword group, which is followed by the name (usually a number) and the status and repeating information. There is always a matching endgroup keyword for each group. The groups are usually indented a few spaces to make the file more readable for human readers. In the case of segment files, the indentations are used for the reading program as well (see Segment Description, below).

Message Usage

Optionally, there can be a usage description for each segment and group. Usage texts start from the keyword USAGE and extend to the end of the file. Each segment is identified with the name and enclosing group (if any). A group is identified by its name preceded by a lower case “g.” The identification tag starts with an exclamation mark followed by the name enclosed in square brackets.

```
! [g3]
Segment Group 3: PCI-GIN
A group of segments identifying markings and labels on individual shipping or
```

```
packing units subject to customs action
! [PCI 3]
Package identification
A segment identifying markings and labels on individual shipping or packing
units.
```

This usage can be viewed with the Message Manager and it is also included in the documents produced. The usage texts are not usable from RTE.

Segment Description

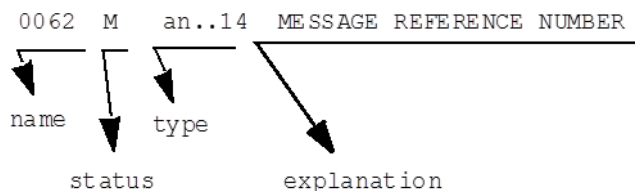
All segments are described in separate text files. The files are named after the segment names; for example, the segment named UNH resides in a file named UNH.

On the first meaningful line, the file contains a long name for the segment. Starting from the second meaningful line, there is a short explanation of the intended use of the segment. This explanation ends when the keyword `ELEMENTS` is found. Comments start with a `"#"` sign and extend to the end of the line.

From the line following the keyword **ELEMENTS**, there is a list of all the elements included in the segment. Elements can be of three different types: simple, composite, and component. These types are explained below.

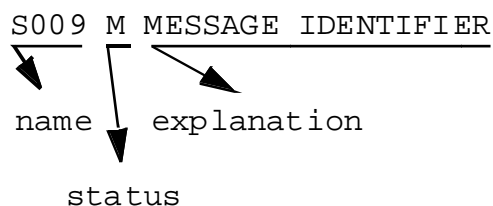
Simple Data Element

A simple data element contains one piece of information. The specification starts from the first column and contains the name, status, type, and an explanation.



Composite Data Element

Composite data elements are constructed of one or more component data elements. The specification giving the name and status for the entire composite starts from the first column and contains the name, status, and an explanation. The name of the element does not always reveal whether it is a simple or a composite (in EDIFACT there is a clear distinction, however). A specification for a composite data element has no type specification (e.g., an..14) and this is used here to distinguish the two types.



Component Data Element

Component data elements are listed on the lines immediately following the specification for the governing composite data element. The lines for component data elements must be indented with at least one space to distinguish them from simple data elements.

```
S009 M MESSAGE IDENTIFIER
    0065 M an..6 Message type identifier
    0052 M an..3 Message type version number
    0054 M an..3 Message type release number
    0051 M an..2 Controlling agency
    0057 C an..6 Association assigned code
```

NAME	The name for the element. Cannot contain spaces.
STATUS	Status can be either mandatory (M) or conditional (C).
TYPE	This field specifies the data type and the dimensions. Data type can be one of the following: <ul style="list-style-type: none"> • a: alphabetic. All extraneous spaces at the end of the string are stripped. • an: alphanumeric. All extraneous spaces at the end of the string are stripped. • n: numeric. Can include a decimal separator (which can be defined in the character set; usually a comma or period) and a sign (+ or -). There must be one digit both in front of the decimal separator and after it. The decimal separator and the sign are not counted in the total amount of characters. All leading zeroes are compressed.
AN	Alphanumeric. No space compression is done.
Nn	Numeric, where "n" specifies the fixed number of decimals. No decimal separator is allowed. No compression is done for leading zeroes. Sign characters are not counted in the total amount of characters.
R	Decimal number. A period is the only allowed decimal separator and it can appear as the last character as well (for example, 3.). Sign characters and decimal separators are not counted in the total amount of characters.
ID	Identifier. At the lower level, this is the same as AN, but at the message level, these can contain only values listed as separate code lists.
DT	Date. The format is YYMMDD, where YY must be from 00 to 99, MM takes values from 01 to 12, and DD can be from 01 to 31.
TM	Time. The format is HHMM[S..S], where HH ranges from 00 to 23, and MM from 00 to 59. There is no fixed length for seconds.

The length of the element is specified immediately after the content type specifier. If there is only a number following the specifier, the element is taken to be fixed length. Two periods before the length specifier indicate that the element is variable length and the number indicates the maximum length.

Note that the examples here represent parts of the UNH segment in EDIFACT, and there are some additional ways to distinguish between different types of elements. All composite data elements in service segments start with a capital "S," in regular segments they start with a capital "C," and in both cases three digits follow the initial letter. Names for simple and component data elements are always four digits. The explanation for composite and simple data elements is often written with all capital

letters, and for components with lower case letters. However, since the TradeXpress translator supports other grammars besides EDIFACT, these naming conventions cannot be relied on.

```
MESSAGE HEADER
#Release:(88.1) 90.2
To head, identify and specify a message.
#- - - - -
ELEMENTS
#- - - - -
0062 M an..14 MESSAGE REFERENCE NUMBER
S009 M MESSAGE IDENTIFIER
    0065 M an..6 Message type identifier
    0052 M an..3 Message type version number
    0054 M an..3 Message type release number
    0051 M an..2 Controlling agency
    0057 C an..6 Association assigned code
    0068 C an..35 COMMON ACCESS REFERENCE
S010 C STATUS OF THE TRANSFER
    0070 M n..2 Sequence message transfer number
    0073 C a1 First/last sequence message transfer indication
```

9.4 Appendix D: Character Sets

9.4.1 Character Set Definition

TradeXpress allows users to define their own character sets. The TradeXpress distribution includes all the standard character sets used in the UN/EDIFACT and X12 community. These sets can be used as a template when creating new ones.

Character sets are defined with a description language especially designed for the EDI syntax usage. The definition file is compiled and placed in the character set directory, which is normally \$EDIHOME/charsets.

Character set definitions can be stored either in the **\$EDIHOME/charsets** or **\$HOME/charsets** directory. The TradeXpress system modules will search the user's home directory for character sets first. If a valid character set definition is not found, the system's **charsets** directory is used.

Character sets are handled automatically by the TradeXpress system modules. If the user executes the RTE translator as a stand-alone program, character sets can be defined with command line arguments. The following options are used for character sets:

- -d <directory path>
- -c <character set>

The command line can have as many -d options as required to specify all the possible places for character set definitions. Only one -c option can be given. For example:

```
invoic.rec.911 -cB -d$HOME/charsets -d$EDIHOME/charsets
```

The example above would implement the default character set search policy for the invoic.rec.911 translator.

9.4.2 The Description Language

9.4.2.1 Character Classes

The following table introduces the different character classes for TradeXpress character class definitions.

Character class	Description
seps	separator characters in the following order seps: <cs><es><st><ri><ts>, where <ul style="list-style-type: none"> • cs - component separator • es - element separator • st - segment tag • ri - release indicator • ts - tag separator; if missing equals es
alpha	alphabetic characters
alnum	alphabetic and numeric characters
digit	digit character class

Character class	Description
sign	sign character
decimalpoint	decimal point character
discard	characters to be discarded
locale	allowed character set <UPPERCASE><UPPERCASE>...<lowercase><lowercase>... For example, Scandinavian characters in 7 bit ASCII set: locale: \[\[\]\]\` \ (Notice escape characters) Characters A-Z need not be explicitly defined in the locale description.
id	identification character set (ANSI)
time	characters used to describe the time data type (ANSI)
string	characters used in string data (ANSI)
decimal	decimal character (ANSI)
date	characters used to describe the date data type (ANSI)

Character classes are written as follows:

TAG : CHARACTER CLASS DEFINITION

A character class definition is a set of characters that belong to a class. The following rules apply to character set definition:

\NEWLINE	class definition continues to following line
OCTAL NUMBER	octal number representation of a character
\CHARACTER	escape character for special characters; use also with the minus (-) sign
^CHARACTER	control character representation (^C == 003)
ANY-ANY	range of characters including delimiting characters

9.4.2.2 Compiling Character Sets

Character set definitions must be compiled before they are valid for RTE translators. Compilation works as follows:

```
cclass < definition > target
```

The **cclass** compiler reads the definition file from its standard input and writes the machine readable character set file to its standard output.

Standard practice with TradeXpress is that character set definition files are kept in the **charsets/src** directory and the machine readable forms are kept in the **charsets** directory.

For example, to compile a character set definition for character set B, give the following command.

```
cclass < charsets/src/B > charsets/B
```

9.4.2.3 Sample Character Set

The following character set definition describes the UN/EDIFACT character set A.

```
#
#      UNOA
#
seps: :+'?
alpha: \ A-Z
alnum: \ !"%&'()*+,\-./0-9:;<=>?A-Z
digit: 0-9
sign: \-
decimalpoint:,.
discard:\000\011\012\014\015
whitespace:\011\012\015\040
```

The following character set definition describes the Finnish national character set, including Scandinavian characters.

```
#
#      UNOY
# seps: 037\035\034\000
alpha: \ A-Za-z\133-\136\173-\176
alnum: !"%&'()*+,\-./0-9:;<=>?A-Za-z\
133-\136\173-\176
digit: 0-9
sign: \-
decimalpoint:..
discard:\000\011\012\014\015
whitespace:\011\012\015\040
```

The following character set definition describes the ISO 8859-1 character set; that is, UN/ EDIFACT character set C:

```
#
#      UNOC (8859-1)
#
seps: :+'?
alpha: \ A-Za-z
#
# tab>, < space> -< tilde>, < inverted !> - small y< with diaeresis>
#
alnum: \011\040-\176\241-\377
digit: 0-9
sign: +\-
decimalpoint:,.
#
# NUL, NL, FF, CR
#
discard:\000\012\014\015
#
```

```
# upper- and lowercase special characters
#
# locale:\300-\326\330-\337\340-\366\370-\377
string:\001-\377
integer:0-9
decimal:0-9
#
#< tab>, NL, FF, CR, < space>
#
whitespace:\011\012\015\040
```

9.5 Appendix E: Code Conversion in RTE

The RTE code conversion mechanism is based on code tables stored in ordinary ASCII files that have two or more columns of data. For example, the following file, **codes/owncode**, could be used as a code table.

BUYER	BY	1
PAYER	PY	2
VENDOR	VE	3

The columns can be accessed by means of a special RTE expression that allows the user to define (a) the key column and (b) the column from which the actual data item is picked.

The RTE expression syntax for code conversion has three variants:

```
<code file> {<key>, <col. separator>}
<code file> {<key>, <output col.>,
<col. separator>}
<code file> {<key col.>, <key>, <output col.>, <col. separator>}
```

The first variant assumes that the code file has a two- column layout. **<key>** is used as the key value, and it is searched for from the first column. If a matching value is found, the value from the second column is returned.

```
e2971 := "codes/owncodes" { "BUYER" , " " }
```

The above expression would produce "BY" as its value. The column separator character can be any valid character, with a space or Tab being the most commonly used.

The second variant can be used with multi-column code files. This variant works like the first one, except that the value is taken from the column specified by the second argument, **<output col.>**.

```
e2971 := "codes/owncodes" { "BUYER" , 3, " " }
```

The above expression would produce 1 as its value.

The last variant can be used when the key column is not the first column. This enables, for example, the use of the code conversion table for both coding and decoding values.

```
e2971 := "codes/owncodes" { 2, "VENDOR" , 3, " " }
```

The above expression would produce 3 as its value.



The following technique is useful when writing RTE programs. The idea is that the user should always check the existence of the actual code file in order to produce meaningful error messages and cause the desired results when errors occur.

```
! This piece of RTE code would come before the begin section.
#define OWNCODE (A) fCodeFile.FULLNAME
{ A, " " }
...
! And this would be where code conversion is required.
e2971 := OWNCODE ("BUYER")
```

```
! This piece of RTE code could be in the begin section
fCodeFile := build (sHOME, "codes/owncode")
if not fCodeFile.EXIST then
    log ("Code conversion file for...not found,
        resuming execution.", NL)
    exit (1)
endif
...
! And this would be where code conversion is required
e2971 := fCodeFile.FULLNAME { "BUYER" , " " }
```

Another useful programming technique is to use macro definitions for code conversion expression. This helps reduce the excessive writing of code conversion expressions when writing translator programs.

```
! This piece of RTE code would come before the
! begin section.
#define OWNCODE (A) fCodeFile.FULLNAME { A, " " }
...
! And this would be where code conversion is required.
e2971 := OWNCODE ("BUYER")
```

9.6 Appendix F: Advanced Utilities

9.6.1 Introduction

This appendix provides a thorough explanation of some of the advanced features of the RTE language. These features will not necessarily be used in your daily work, but they provide tools for the advanced use of TradeXpress. This appendix covers the following topics:

- pass-through mode in RTE
- segment look-ahead in the receiving translator
- C programming in RTE
- RTE - coprocess communications
- RTE FTP routine library

The first two topics cover EDI translation. They explain how the user can process very large messages with the minimum processing effort.

The C programming section explains how to interface existing function libraries or API's to the RTE translator. This feature can be used if the customer has ready-made functions for data manipulation, or if the user wants to integrate the translator to a database using an API (application program interface).

The coprocess communications section explains how to interface the translator to the existing system using coprocess communications facilities. This kind of interface can be used with databases or with application interfaces.

The FTP library section explains how users can easily program their own FTP procedures using the TradeXpress FTP library. This library provides tools that help you rapidly develop new routines for sample application interfaces using TCP/IP FTP functionality.

9.6.2 Pass-through Mode in RTE

In many business environments, operative systems tend to process large amounts of data infrequently, instead of processing smaller amounts more often. For example, the invoices containing a week's sales may be sent once a week, on Friday. On the other hand, the data involved in a business transaction is usually repetitive by its nature. Orders, invoices, or dispatch advices, to mention just a few, all consist of an unlimited number of line items.

With this in mind, the TradeXpress RTE translator has been enhanced to perform better with single messages containing huge amounts of data. Performance is increased, and some problems caused by system resource limitations are avoided. Pass-through mode enables the system to process huge messages, as well as speeding up normal processing and releasing a lot of the memory expended during normal operation.

The idea behind pass-through mode is that the message is read into memory segment by segment. In the sending direction, the user decides how many segments reside in memory simultaneously before memory is released. In the receiving direction, only one segment at a time is kept in memory. In other words, a segment flows through the translator without being stored after it has been processed. Because the entire message is never in memory at the same time, only local syntax or message integrity checking can be performed. The user is responsible for doing integrity checking for the whole message, using the appropriate RTE facilities, such as **valid()**.

9.6.2.1 Receiving an EDI Message in Pass-through Mode

Pass-through mode for the receiving RTE translator is indicated in the message statement at the beginning of the translator.

```
message "UN-EDIFACT/91.2/invoic.msg" receiving, passthru
```

In normal (non-pass-through) operations, the RTE translator first evaluates the EDI message, checks the syntax and message structure integrity, and then passes the message segment by segment to user-defined segment statements.

In pass-through mode, the RTE translator checks the syntax of the segment and the validity of the message so far, and passes the segment immediately to the user-defined segment statement. The memory used by the segment and the related data structures is released after the segment statement has been executed.

For the programmer, this means that elements outside the current segment cannot be accessed (that is, elements in the previously processed segments). Of course, the data from the earlier elements can be stored in RTE variables and tables, but it is the programmer's responsibility to take care of resource management after having used the saved values.

Pass-through mode imposes new requirements on the translator in terms of inhouse file building. The conventional method is to build inhouse records immediately after each segment is parsed. If there are errors in the message, the inhouse file constructed is inconsistent and must be discarded in most cases. Thus it is good programming practice to build inhouse file from each message into a buffer or temporary file. After the entire message is processed, the buffer or file is concatenated into standard output.

Example:

```
segment UNH
  put (1, 1, "UNH")
  put (1, 4, e0062:14.14)
  !- - - - -
  ! Save reference number for later use
  !- - - - -
  tRefNo := e0062
endsegment
...
segment UNT
  if e0062 <> tRefNo then
    log ("Inconsistent reference number.", NL)
    exit (1)
  endif
  flush ()
endsegment
```

9.6.2.2 Building an EDI Message in Pass-through Mode

In normal (non-pass-through) operations, the RTE translator builds an EDI message guided by the user-defined segment building statements. After the message is complete, the syntax and validity are checked and the message is released to be output to the interchange construction module.

The pass-through mode for the sending RTE translator is always available without any mode settings. The pass-through-enhanced RTE translator builds the message into memory until the user performs the flush function to output the message built so far. The memory used by the message structure, segments, and related data structures is released after the flush has been executed.

For the programmer, this means that the elements built before the last flush cannot be accessed. The data from the earlier elements can be stored in RTE variables and tables, but the programmer is responsible for dealing with resource management after having used the saved values.

The user is responsible for validity checking of the message portions to be flushed. The user must check for the validity of the segments and segment groups, as well as the appearance of the mandatory segments outside any groups (segments in the 0- and 1-level of the message). Here are the steps you would follow to use pass-through mode when constructing an effective translator for an INVOIC message:

1. Build the header segments and the beginning section of the message and validate each segment/group built.
2. Build the line item group.
3. In the appropriate segment of the line item group, validate the line item group and execute the flush function.
4. Build the trailer part of the message, validate each segment/group built, and flush the message.

The user will probably encounter difficulties while performing the validation and flushing procedures, because the EDI message to be processed contains mostly optional segments. In many cases, the only mandatory segment in a segment group is its leading segment. Thus, the checks must be performed with the segment building operation that is always performed. The best places for validation and flushing are usually the service segments (for example, EDIFACT UNS, UNT) and the leading segments of the groups.

Example:

```
base "syslog.cfg" LOG
begin
    ...
    ! intermediate file for message part flushing
    fTmpFile := LOG.temp
endbegin
line (1:"BGM")
    !- - - - -
    ! Initiate message variables
    !- - - - -
    nERRORS:= 0
    bUNH:= FALSE
    bBGM:= FALSE
    bUNS:= FALSE
    bUNT:= FALSE
```



```

nMSG++
segment UNH
    e0062 := build (nMSG)
    eS009.0065:= "INVOIC"
    eS009.0052 := "1"
    eS009.0054 := "911"
endsegment
nUNH := bfValidSegment()

segment BGM
    e1004:= pick (1, 4, 35)
    eC507.2005:= "22"
    eC507.2380 := time ("%Y%m%d")
    eC507.2379 := "102"
endsegment
bBGM := bfValidSegment ()
endline
...
line (1:"LIN")
    !- - - - -
    ! Output previous group (and header section segments)
    !- - - - -
    if bNewGroup then
        if valid (GROUP g21) then
            flush (MESSAGE) >>fTmpFile
        else
            log ("Errors found.", NL)
            edierrordump (MESSAGE)
            nERRORS++
        endif
    endif

    segment LIN g21
        e1233 := pick (1, 4, 3)
        e1082 := pick (1, 7, 8)
    endsegment
    bNewGroup := TRUE
endline

line (1:"RFF")
    segment RFF g21
        eC506.1153 := "AAK"
        eC506.1154 := pick (1, 15, 10)
    endsegment
endline

line (1:"UNS")

```

```

!- - - - -
! Output the last group
!- - - - -
if valid (GROUP g21) then
    flush (MESSAGE) >>fTmpFile
else
    log ("Errors found.", NL)
    edierrordump (MESSAGE)
    nERRORS++
endif
segment UNS
    e0081 := pick (1, 4, 1)
endsegment
bUNS := bfValidSegment()
endline
...

line (1: "UNT")

    segment UNT
        e0074 := pick (1, 4, 8)
        e0062 := build (nMSG)
    endsegment
    bUNT := bfValidSegment()

!- - - - -
! Check the presence of mandatory 0- and 1-level segments
!- - - - -
if bUNH and bBGM and bUNS and bUNT then
    flush (MESSAGE) >>fTmpFile
else
    log ("Errors found.", NL)
    edierrordump (MESSAGE)
    nERRORS++
endif
endline

```

9.6.2.3 Error detection within Pass-through mode

Use

The segment errors are stored in the two overall tables: `tPASSTHRUerrordump` (message in clear), and `tPASSTHRUerrorlist` (message CONTROL coded).

Before processing each segment, the following function is called: `nfValidPASSTHRU(segment_name, Err)`

where:

- `segment_name` is the name of the current segment (or "" if the error is not related to a segment)

- Err is set to:
 - “0” if there is no error at message level,
 - “1”, otherwise.

In this function, the user is able to:

- Print the errors
`log(tPASSTHRUerrordump)` and `log(tPASSTHRUerrorlist)`
- Validate the segment
`if valid(SEGMENT)....`
- Escape from the message in progress
`nextmessage`
- Allow normal segment processing
`return(0)`
- Cancel the segment processing
`return(1)`
- And run all process available in a function related to a segment...

Example

```
message "UN-EDIFACT\93\draft\orders_d_tst.msg" receiving, passthru

begin
endbegin

segment UNH
    tUNH_0062 := e0062
    tUNH_S009_0065 := eS009.0065
endsegment

segment BGM
    tBGM_1004:= e1004
endsegment

segment DTM
    tDTM_eC507_2005 := eC507.2005
endsegment

segment LIN g25
    tLIN_C212_7140 := eC212.7140
endsegment

segment UNS
endsegment

segment UNT
endsegment
```

```

end
endend

function nfValidPASSTHRU(tSeg,nErr)

    log("Valid : tSeg=",tSeg," nErr=",nErr,NL)

    if nErr <> 0 then
        log(tPASSTHRUerrordump)
        log(tPASSTHRUerrorlist)
    endif

    if not valid(SEGMENT) then
        nErr := 1
        edierrordump(SEGMENT)
        edierrorlist(SEGMENT)
    endif

    if nErr <> 0 then
        nextmessage /* skip the message if erroneous segment */
    endif

    return(nErr) /* skip the segment if erroneous else process it */

endfunction

```

Detected errors

The errors detected in passthru mode (resulting to a call of nfValidPASSTHRU with nErr=1) are:

- Missing mandatory group or segment
- Number of repetitions of a group or segment greater than the maximum specified

The other case of errors (for example, incorrect data type) must be tested by the function valid(SEGMENT), as indicated in the example above.



The syntax control level depends on the user configuration ("User info / Settings /RTE startup defaults")

9.6.3 Segment Look-ahead in the Receiving Translator

The RTE translator passes segments to user-defined segment statements in the same order as segments occur in the message (tree traversal order). This order may sometimes cause extra work when the message translator is built, which makes it useful to be able to "look ahead" at segments while receiving an EDI message.

The statement **peeksegment** enables you to access segments before they are processed in the user statements. It replaces the previous **bfPeeksegment** function described below.

Syntax	peeksegment(text array, [[group/occurrence, spec.], name, number)
--------	---

Limitations	This statement cannot be used in EDI or AnyToAny syntax with passthru mode.
-------------	---

9.6.3.1 RTE Function

peeksegment, bfPeeksegment



The peeksegment statement replaces the formerly-used bfPeeksegment function.

Segment look-ahead function in receiving translators is available in both modes (standard and passthru).



The passthru mode is not supported in Syntax to Syntax with passthru mode.

The bfPeekSegment function fetches a segment specified by the group and the repeat information. If the segment is found, the contents of the elements are written in the array such that the array index is the element name. The function returns TRUE if successful; otherwise FALSE.

```
bfPeekSegment(text array, [group/occurrence. spec.], name, number)
```

text array	place for the element content
group/occurrence spec.	group name and group occurrence number pairs for the entire path from 0-level down to nearest group level
name	segment name to be accessed
number	occurrence of the segment

The text array that is used to store the element contents is emptied every time the bfPeekSegment function is called.

The example below shows a look-ahead for the first occurrence of the DTM segment in group 5 repetition 2 under group 4 repetition 3.

```
bfPeekSegment(taPEEK, "4", 3, "5", 2, "DTM", 1)
```



bfPeekSegment takes a variable number of arguments, depending on the position (depth) of the segment to be accessed.

Example

The following example will skip all INVOIC messages having zero total amount in the MOA segment.

```
!
! Message Manager 0.0
!
! DATE: Mon Jun 13 11:45:40 1999
!
message "UN-EDIFACT/91.2/invoic.msg" receiving

segment UNH
  nUNH := nGLB
  put(nUNH, 1, "UNH_#")
```

```

    put(nUNH + 0, 6, e0062:14)
    put(nUNH + 0, 20, eS009.0065:6)
    put(nUNH + 0, 26, eS009.0052:3)
    put(nUNH + 0, 29, eS009.0054:3)
    put(nUNH + 0, 32, eS009.0051:2)
    put(nUNH + 0, 34, eS009.0057:6)
    put(nUNH + 0, 40, e0068:35)
    put(nUNH + 0, 75, eS010.0070:4)
    put(nUNH + 0, 79, eS010.0073:1)
    nGLB := nUNH + 0 + 1

    if not bfPeekSegment(taPEEK, "15", 1, "MOA", 1) then
        log("The peeked segment was not there", NL)
    else
        if number (taPEEK["C516.5004"]) = 0 then
            log("Total amount is 0 skipping.", NL)
        else
            log("That much money, wow!!!", NL)
        endif
    endif
endsegment

segment BGM
    nBGM := nGLB
    put(nBGM, 1, "BGM_#")
    put(nBGM + 0, 6, eC002.1001:3)
    .
    .
    .
endsegment

segment UNT
    nUNT := nGLB
    put(nUNT, 1, "UNT_#")
    put(nUNT + 0, 6, e0074:8)
    put(nUNT + 0, 14, e0062:14)
    nGLB := nUNT + 0 + 1
    flush (0, 0, NL)
endsegment

end
.
.
.
endend

```



For more details about using peeksegment, bfPeeksegment for RTE traslator with XML syntax, refer to the Technical Reference User's Guide, Chapter Syntaxes, section XML.

9.6.4 C Programming in RTE

TradeXpress supports the use of the C programming language within the RTE programs. This feature is called RTE In-line C. It is a valuable tool, especially for advanced RTE programmers familiar with C. The In-line C facility lets you insert your own C-written modules, functions, and libraries directly into the RTE code. Furthermore, all the C run-time functions are at your disposal, in addition to RTE built-in functions, when you are coding translators and other RTE programs.

In-line C statements are passed directly to the C compiler without any modification by the RTE compiler. In addition, the C preprocessor, a facility that existed in the earlier versions of RTE, can be used to define symbols and macros, include code from other files, and perform conditional compilation.

9.6.4.1 In-line C Block Statement

The In-line C block statement provides access to the description area of an RTE program. Users can use this statement to define their own C functions and data structures. Syntax:

```
inline<nl>
    <C source code>
endinline<nl>
```

In-line C block statements can be located in any statement list or function part in the RTE file. The order of appearance of the In-line C functions and their calls is not significant to RTE; a function can be called before it is presented in the RTE file. And there are no limitations to the naming of In-line C functions, as long as the calls are made as In-line C statements, described below. The functions presented in the In-line C blocks can also be called as normal RTE statements.

In this case, the function names and return value types must meet the RTE requirements. The name of a function begins with **tf**, **nf**, or **bf**, corresponding to the return value types text, numeric, and Boolean, respectively. The associated C variable types are presented as follows:

Function name	RTE variable	C type variable Type
tf<name>	Text	char *
nf<name>	Numeric	double
bf<name>	Boolean	int

Example:

```
inline
    bfPrintHello ()
    {
        printf ("Greetings from \Inline\n");
    }
    double nfCounter (double x, double y)
    {
        return (x + y);
    }
endinline
...
```

```
end
    bfPrintHello()
    nSum := nfCounter(1,2)
endend
```

9.6.4.2 In-line C Statements

In-line C statements provide access to the RTE program user statement area. Users can define their own C statements, such as function calls and data structures, in this way. In-line C statements are also useful when the user attaches API functions such as database access directly to the RTE program.

Syntax:

```
inline "<C source code %N >",<0>, %1, ..., %n<n1>
```

where %N represents a replaceable, ordinal macro variable.

These variables can be located anywhere in the C source code and they are replaced with arguments from the argument list. Example: %0 is replaced with the first argument, %1 with the second argument, etc. There is no limit to the number of variables.

Example 1:

```
inline
    bfPrintHello (char *greetings)
    {
        printf ("Greetings\n\t%s \n" greetings);
    }
endinline
...
end
    bfPrintHello("from Paris.")
endend
```

Example 2:

```
begin
    inline "SQL_Connect(%0, %1, %2);", \"srv\", ARGV[1], ARGV[2]
endbegin
...
line "1:DATA"
    split (taData, pick(1,1,EOL), ",")
    inline "SQL_Insert (%0, %1);", \taData[1], number (taData[2])
endline
...
end
    inline "SQL_Disconnect();"
endend
```


9.6.5 RTE Coprocess Communications

Coprocesses are normal Unix processes that are executed within a parent process, and their input and output data is fed or processed by the parent process. Coprocesses are usually used to perform tasks that are not programmed into the parent process, such as data communication and database access.



RTE coprocess communication is not available in the NT environment.

Coprocess communication within RTE is implemented as a set of functions that can accomplish the following:

- start the coprocess
- write input for the coprocess
- read output from the coprocess
- match text from the coprocess output stream.

In addition, the RTE switch statement has a new extension for matching and expecting coprocess output.

Coprocess communication dedicated functions are different from the RTE runtime functions in the sense that by default, they reside in their own library, called `libcopro.a`. If the default actions of the functions do not meet the user's needs, then the functions can be written like RTE functions at the end of the RTE program.

9.6.5.1 Coprocess Communication Basics

Coprocess is a Unix process that is executed by another process as a child process. Characteristic of the coprocess is that it is fed by data from the parent process, and/or the parent process reads data from the coprocess. This conversation with the parent process distinguishes the coprocess from the plain child process.

When working with coprocesses, you must always consider one-way versus two-way communications. One-way communications (either writing to the coprocess or reading from the coprocess) are comparatively straightforward. When writing to the coprocess, you must make sure that the coprocess is ready for input, and when reading from the coprocess, you must take care of timeouts while waiting for input.

Two-way traffic is more complicated because of the need for synchronization (for example, in deadlock situations). There is a wide range of literature that deals with this subject. When starting different programs as a coprocess, you must also consider the startup environment. For example, some programs require a properly defined terminal environment (`tty`) or certain environment variables to be set before execution.

RTE Functions for Coprocess Communications

RTE provides a set of functions to handle the above- mentioned aspects of coprocess communications. There are functions for the startup environment setting, data input and output to and from the coprocess, and synchronization.

When calls are made to the coprocess functions, they are read from libraries and perform default actions. When specific tailoring, exception handling, or logging is needed, you can define your own

actions for any of these functions. These are user-provided functions that the user can create and add to the end of the RTE file, similar to the other, ordinary user functions.

nPID := nfLaunchCoProcess (tCOMMAND)

Starts a coprocess given in tCOMMAND, with **/bin/sh -c tCOMMAND**. If tCOMMAND is empty, just **/bin/sh** is started. Returns the pid of the started process, or 0 in case of failure.

The process is started on a pseudo terminal (pty), which has 256 columns and 25 lines. The coprocess is given its own session if the underlying system supports it, or a process group if not. The variables **TERM=dumb**, **LINES=25**, and **COLUMNS=256** are inserted into the environment of the coprocess. The pty is initialized to contain no-echo, no-postprocess, 8bit no-parity, no-xon/xoff, canonical input. The control characters are the common ones, intr ^C, quit ^\, erase ^H, kill ^U, eof ^D, and eol ^J. Before the command is executed, a call is made to function **bfCoProcessStartup()**. This function can be tailored by the user to either set different tty modes (with **system("stty ...")** for example), or execute coprocesses differently.

nRV := nfCloseCoProcess (nPID)

Closes the pty of the coprocess and waits for the process to exit. Returns the exit status of the process: 0 if no coprocess was found or -1 on system error. Even though the process may already have finished, this function is still needed to free memory that has been used for managing the coprocess (the exit status is returned).

nHIT := nfExpectCoProcess (nPID, nSECONDS, tTOKEN0, tTOKEN1, ..., EMPTY)

Wait until any of the given tokens appears on the input, or nSECONDS elapse. nSECONDS equal to 0 means an immediate return, and any negative value means an infinite waiting time. If a token is found, its order number on the argument list is returned; for example, the appearance of tTOKEN1 would return 1. If no match is found in the given time, -1 is returned. If the process dies unexpectedly, -2 is returned. -1 is returned if the usersupplied function **bfCoProcessNoMatch()** returns **TRUE** for any line. Each call of this function flushes the read-queue up to the last match.

The pattern matching algorithm for tokens is simple. The characters ^ and \$ are special if they appear at the beginning or end, respectively. They can be used to anchor the token to BOL and/or EOL. Other tokens can be found anywhere on a line. Lines end with a newline character that should be added to a token when necessary. Prompts that do not end in newline are best found with "^csh % \$". Complete lines require "^line\n\$".

tTTY := tfCoProcessTty (nPID)

Returns the name of the pseudo terminal where the coprocess runs.

tTXT := tfReadCoProcess (nPID)

Returns one line from the coprocess, waiting for one to appear if no lines were queued already.

tTXT := tfCollectCoProcess (nPID)

Returns one queued line from the coprocess and does not wait for more. Lines accumulated between expects can be picked up with this function if it is called repeatedly until it returns **EMPTY**.

bOK := bfCoProcessOkay (nPID)

Returns **TRUE** if the coprocess can be found and no errors have been detected during the process.

bOK := bfKillCoProcess (nPID, nSIGNAL)

Sends the **nSIGNAL** to the process given in **nPID**. Returns **TRUE** if the system call kill was successful.

bOK := bfWriteCoProcess (nPID, tTXT)

Appends the data given in the text to the write queue of the coprocess, and tries to write some amount to pty. Returns **TRUE** if the coprocess was found and no errors have been detected.

bOK := bfFlushCoProcess (nPID)

Flushes (forgets) the read and write queues associated with the coprocess, and clears any error condition. Returns **TRUE** if the coprocess was found.

bOK := bfDrainCoProcess (nPID)

Waits until all output has been written to the pseudo terminal of the coprocess from the write queue. Returns **TRUE** if the process was found and the output completed without errors.

nOLD := nfDebugCoProcess (nHOW)

Activation of debugging output. Any generated output is logged to **stderr**. The default state is all debug output off. **nOLD** returns the old debug state, and **nHOW** sets the new one. Available bits for **nHOW**:

- input (data read from pty)
- output (data written to pty)
- expect matches (token that matched in expect)

The following functions can be supplied by the user and are called by the other functions in the coprocess library.

Boolean bfCoProcessStartup (tTTY)

Called in the new process just before the command is to be executed. This is the proper place, for example, to alter the terminal settings with the command **system("stty")**. The environment can also be changed here without disturbing the parent's environment. If this command returns **TRUE**, the command given to **nfLaunchCoProcess()** is executed. If it returns **FALSE**, this is taken to indicate an error, and **nfLaunchCoProcess()** fails.

Boolean bfCoProcessInput (nPID, tTXT, nTXTLEN)

Called before a complete line from the coprocess is appended to a read queue. **tTXT** contains the line whose length is **nTXTLEN**. This function should return **FALSE** if the line is to be ignored. Can be used to filter uninteresting lines, or to process input line-by-line during **nfExpectCoProcess()** or **tfReadCoProcess()**.

Boolean bfCoProcessNoMatch (nPID, tTXT, nTXTLEN)

Called after a complete line has been tested against all tokens in **nfExpectCoProcess()**. **tTXT** contains the line whose length is **nTXTLEN**. This function should return **FALSE** if expect is to be continued. If it returns **TRUE**, expect terminates and returns -1. This could also be useful for processing input lines during expect.

Boolean bfCoProcessExit (nPID, nEXITSTATUS)

This is called whenever the coprocess exits unexpectedly, for example during **fExpectCoProcess()**.

Boolean bfCoProcessError (nPID, tSYSCALL, nERRNO, nRETVAL)

This is called after an io error is detected. The **tSYSCALL** can have the values "read" or "write". **nERRNO** is the system errno variable and **nRETVAL** is the return value from the failed system call. If this function returns **FALSE**, the error is ignored. These functions are mainly used for debugging or very special cases, and require the use of the C language in RTE.

int coprocess_expect_match (char *token, char *line, int linelen)

This function determines whether an expect token matches with an input. The token is the user-supplied argument in **nfExpectCoProcess()**. Arguments **line** and **linelen** contain the input string that is to be matched and its length. A line can be terminated with a newline character, in which case it is a complete line. Otherwise the line may not yet be complete.

The function should return zero if no match was detected, or the number of bytes this match removes from the line. One special case for this function is to support regular expressions when matching input.

The default function supports only these special cases:

- ^text matches text at the beginning of the line
- text\$ matches text at the end of the line

These two can be mixed, in which case only exact lines will match. An example of a token that would match a prompt from program foo would be "^foo> ".

Note that \$ does not imply a new line, so complete lines should be matched with ^line\n\$. This requires knowledge of the terminal modes. In normal use, the terminal is configured to produce only a newline character, but programs could change the modes so that a carriage return is also appended to output lines.

int coprocess_input_cleanup(char *text, int len)

Function to clean up text that has been read from the terminal. The function should modify the text as appropriate and return the modified length of the line. The original length (len) should not be exceeded.

int coprocess_debug_flags;

Void coprocess_debug_input (char *text, int len) flag 1

Void coprocess_debug_output (char *text, int len) flag 2

Void coprocess_debug_match (char *token) flag 4

These functions are called after a match is found in **nfExpectCoProcess()**, after reading some amount of data from the terminal, and after writing data to the terminal, respectively. The default functions log to **stderr**, if the corresponding bit is set in the flag integer. By default, nothing is logged.

9.6.5.2 RTE Extension for Coprocess Communications

The RTE language has an extension of the switch statement for use by the coprocess library.

```
switch expect <PID> <time-out>
    case <string>:
        <RTE statements>
    ...
endswitch
```

A parameter timeout of 0 means an immediate return and any negative value means infinite waiting.

This example shows the wait for the string **Password:** at the beginning of the input line from the coprocess that is executing the FTP file transfer program:

```
switch expect nPID 5
  case "^Password:":
    bfFlushCoProcess (nPID)
    inline "sleep (1);"
    bfWriteCoProcess (nPID, tPasswd)
  case "^55":
    bfErrorProcessing ()
endswitch
```

Example 1: FTP Session Driven by the RTE Translator

```
begin

  tCommand := "ftp -n"
  tHost := "open localhost 10021\n"
  tUser := "user xxxx\n"
  tPasswd := "yyyy\n"
  tQuit := "quit\n"
  nPID := nfLaunchCoProcess(tCommand)

  if nPID = 0 then
    log ("Cannot execute:: ", tCommand, NL)
    exit (1)
  endif

  nfExpectCoProcess(nPID, -1, "^ftp>", EMPTY)
  bfWriteCoProcess(nPID, tHost)
  nfExpectCoProcess(nPID, -1, "^ftp>", EMPTY)
  bfWriteCoProcess(nPID, tUser)
  nfExpectCoProcess(nPID, -1, "^Password:", EMPTY)
  bfWriteCoProcess(nPID, tPasswd)
  nfExpectCoProcess(nPID, -1, "^ftp>", EMPTY)
  bfWriteCoProcess(nPID, "ls\n")
  nfExpectCoProcess(nPID, -1, "^ftp>", EMPTY)
  tLine := tfCollectCoProcess(nPID)

  while tLine <> EMPTY do
    if compare (tLine, "ftp>") then
      break
    endif
    print (tLine)
    tLine := tfCollectCoProcess(nPID)
  endwhile

  bfWriteCoProcess(nPID, tQuit)
  nfCloseCoProcess(nPID)
endbegin
```

```
function bfCoProcessInput (nPid, tLine, tLen)
    log ("FTP:: ", tLine, NL)
    return TRUE
endfunction

=====>
$ ftp -n
ftp> open localhost
Connected to localhost.
220 mymmeli FTP server (Version 1.7.109.2 Mon Aug 3 22:28:13 GMT
1992) ready.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> user xxxx
331 Password required for solid.
Password:
230 User xxxx logged in.
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 52
-rw-r--r-- 1 solid users 377 Feb 23 10:37 .login
-rw-r--r-- 1 solid users 551 Mar 6 14:15 .profile
drwxr-xr-x 2 solid users 1024 Mar 10 17:56 base
drwxr-xr-x 2 solid users 1024 Feb 27 10:34 bin
drwxr-xr-x 5 solid users 1024 May 4 11:41 src
drwxr-xr-x 3 solid users 1052 Mar 20 18:00 work
226 Transfer complete.
ftp> quit
221 Goodbye.
```

Example 2: SQL Interpreter Session Driven by the RTE Translator

```
begin
    tCommand := "exec solsql \"tcpip mymmeli1313\""
    tUser := "xxxx\n"
    Passwd := "yyyyy\n"
    Quit := "quit;\n"
    nPID := nfLaunchCoProcess(tCOMMAND)

    if nPID = 0 then
        log ("Cannot execute:: ", tCommand, NL)
        exit (1)
    endif
```

```

if oX then
    ! Debug input, output and matches.
    nfDebugCoProcess(7)
endif

switch expect nPID -1
    case "^Username:":
        bfFlushCoProcess(nPID)
        bfWriteCoProcess(nPID, tUser)
endswitch

switch expect nPID 5
    case "^Password:":
        bfFlushCoProcess(nPID)
        inline "sleep(1);"
        bfWriteCoProcess(nPID, tPasswd)
endswitch

switch expect nPID -1
    case "^SOLID SQL Editor":
        bfFlushCoProcess(nPID)
        break
endswitch

bfWriteCoProcess(nPID, "SELECT * FROM CARS;\n")
switch expect nPID -1
    case " rows fetched.\n":
        break
endswitch

while bfGetLine() do
    print (tLine)
endwhile

bfWriteCoProcess(nPID, tQuit)
nEXIT := nfCloseCoProcess(nPID)
endbegin

function bfCoProcessInput (nPID, tLine)
    if oD then
        log ("INPUT:: ", tLine, NL)
    endif
    return TRUE
endfunction

```

```
function bfGetLine ()
    tLine := tfCollectCoProcess(nPID)
    if tLine <> EMPTY then
        return TRUE
    else
        return FALSE
    endif
endfunction

=====>
select * from cars
MODELCOLORPRICE
-----
Volvo 850Yellow250000
Peugeot 405Red150000
Lada 1200White25000
Fiat UnoBlack50000
4 rows fetched
```

9.6.6 RTE FTP Client Routine Library

TradeXpress's basic facilities provide FTP send and FTP fetch functionality for the most common file transfer purposes. These routines have parameters to meet users's needs, from file type definition to acknowledgement facilities. However, there are a number of FTP file transfer situations which these predefined routines cannot handle; for example, specialized login procedures and file transfer actions where several files are transferred during a single session.

The RTE FTP Client Routine Library has all atomic FTP client commands as RTE functions. These functions are available from RTE programs (such as EDI translators or any general purpose RTE programs). These functions contain three basic features:

- timeout processing
- log information processing
- user-defined error detection

9.6.6.1 Compiling RTE Source Code with the FTP Client Routine Library

An RTE program using the FTP Client Routine Library must be compiled using the following mktr command:

```
$ mktr ownftp.rte -lnetlib (Unix)
%EDIHOME%\lib\libnetlib.lib (Windows NT)
```

The FTP Client Routine Library includes a component that manages the password encryption and decryption.

The user must write the function bfExit(tErrorMsg) in the RTE program. This function is called automatically when an error is encountered. It is possible to call the function manually as well. This function should not return.

Parameters used:

- pFTP.HOST - string
- pFTP.USER - string
- pFTP.PASSWORD - string
- pFTP.PASSWORD.ENCRIPTED-string (overrides PASSWORD)
- pFTP.ACCOUNT - string
- pFTP.SITE - string (\n line separator)
- pFTP.XFER.TYPE - IMAGE,ASCII, a.s.o.
- pFTP.BINARY - "1" or not
- pFTP.USE_PASV - "1" or not
- pFTP.XFER.WATCHDOG - (numeric) string, minutes

9.6.6.2 RTE Functions for FTP Communication

General functions for FTP

Generally, all Boolean functions return FALSE when any error occurs, but the severity of the error depends on the function. Text functions return EMPTY on error.

```
taFTP_ALL
taFTP_OK
taFTP_ERROR
```

RTE arrays containing user-defined errors and notes. OK contains patterns that should be accepted as okay; ERROR contains patterns that should be rejected as error responses.

Function	Description
bfFTP_watchdog(nMinutes)	Either sets the timeout (in minutes) or resets it (when the given time is zero). If any activity lasts longer than the timeout, transfer is aborted.
bfFTP_initialize()	Initializes the system, rewinds log history, and loads user-defined errors.
bfFTP_open(tHost, tUser, tPassWord, tAccount)	Makes an implicit call to bfFTP_initialize if this has not already been done. Opens a control connection to a given host. If the account is non-empty, it is sent immediately after logging in and remembered. When an activity requires an account, the same account is used again. If necessary, the remote server port can be given after the host name as a service name or a port number, separated with a ":"; for example, host.name:ftp or 1.2.3.4:2002.
bfFTP_close()	Closes the current connection.
tfFTP_logtext()	Returns and rewinds the log history collected so far.

Function	Description
bfFTP_setlogging(nLevel)	Sets the detail of logging. Levels above 2 log the session continuously into the LOGGING stream.
bfFTP_ascii()	Sets the active transfer type, which can be one of the following: <ul style="list-style-type: none"> • IMAGE, image, i, I • binary, BINARY, bin, BIN • ASCII, ascii, ASC, asc, a, A • EBCDIC, ebcdic, e, E
bfFTP_binary()	
bfFTP_ebcdic()	
bfFTP_type(tType)	
tfFTP_command(tCmd)	Sends the given command and returns the response.
tfFTP_site(tCmd)	Sends the given command prepend with "SITE" and returns the response.
tfFTP_pwd()	Returns the current directory.
tfFTP_lastreply()	Returns the last response received.
nfFTP_size(tRemotefile)	Returns the size of a given file. May not be supported by all FTP servers. Return value is -1 if the size could not be determined (if the file is nonexistent, for example).
tfFTP_modtime(tRemotefile)	Returns the modification time of the given file. EMPTY on error. The FTP server should format the response in the format yyyymmddHHMMSS in UTC time. This can vary with nonstandard servers.
bfFTP_del(tRemotefile)	Deletes the given file.
bfFTP_rename(tFrom, tTo)	Renames tFrom to tTo .
bfFTP_cd(tRemotedir)	Changes the directory. If tRemotedir is "..", then CDUP is used instead of CWD.
bfFTP_putas(tLocal, tRemote)	Stores local file tLocal as tRemote .
bfFTP_putas_uniq(tLocal, tRemote)	Stores the local file tLocal as tRemote. In addition, the FTP server constructs a unique file name by appending a running number to the remote name. This should prevent file overwriting, but again, nonstandard FTP servers may cause problems. The actual remote file name can be collected, after the put has completed, with the call tfFTP_lastuniq().
bfFTP_appendas (tLocal, tRemote)	Appends local file TLocal to tRemote. If tRemote does not exist, it is created.
tfFTP_lastuniq()	Returns the last unique file name received from the FTP server.
bfFTP_getas(tRemote, tLocal)	Retrieves the given remote file to the local file.
bfFTP_put(tFilename) ;	A put and get that use the base name of the given file as the other name. When putting local file /etc/passwd, the remote name is

Function	Description
bfFTP_get(tFilename)	passwd; and when getting remote file /etc/ passwd, the local file is passwd.
bfFTP_dir(tRemote, taArray) ; bfFTP_nlist(tRemote, taArray)	Long or short directory listing. The array is first cleared, then filled line-by-line from the output from the command (LIST or NLST).
tfFTP_clean_multiline(tCommandSequence)	Replaces every “\n” in sequence with a real new line.
tfFTP_user_reject_check(tResponse)	Can be used to detect user-defined errors. First the OK patterns are searched for a match to screen out valid responses, and then the ERROR patterns are searched. Returns EMPTY when the response is OK, and an error string if a match is found in ERRORS.
bfLogIn()	Loads user-defined errors. They are searched from \$FTPERR, \$HOME/ftperr, and finally \$EDIHOME/lib/ftperr. Logs in if not already logged in. Retries twice to connect. Exits if the connection cannot be opened. Runs the site command sequence from pFTP.SITE.
bfLogInTSI(sBase, tType, nIndex)	Needs to replace bfLogIn() if pFTP.PASSWORD is encrypted (e.g.: pFTP.PASSWORD={tsi}1egdSgBs9AsUy) <ul style="list-style-type: none"> • sBase : Path environment variable to the database where the password is stored. (e.g.: sPARTNER) • tType : Database type (partner or edisend). • nIndex : Encrypted password entry of the database index. The behaviour is the same as with bfLogIn, plus the password decryption.
bfLogOut()	Closes the connection if necessary and resets directory-changing variables.
bfCWD(tDir)	Changes the remote directory. An empty tDir is taken as the initial login directory. Never fails; bfExit() is called if chdir does not succeed. Keeps the current remote directory in tFtpCurDir , and logs in when necessary.
bfSizeOK(tRemote, tLocal)	Checks that the local and remote files have the same size. Diagnostic is always logged in case of a mismatch, but an error is returned only when using IMAGE transfer.
bfSetSafetyTimeout()	Sets the timeout in pFTP.XFER.WATCHDOG .
bfDoSiteCommands(tCmds)	Text in pFTP.SITE is cleaned and sent line-by-line to the server.

Example: A Simple FTP Send Program

```

!=====
! pFTP.ACK.DIR - remote directory for the acknowledge flag files
!!
FTP.SITE=UMASK 022\nIDLE 3333
! FTP.REMOTENAME=remfil
! FTP.REMOTEDIR=remdir
! FTP.ACCOUNT=account
! FTP.PASSWORD=password
! FTP.USER=testuser
! FTP.HOST=host
! FTP.DIRNAME=dirname
!=====
!-----
! Generic rte-routines for manipulating FTP.
!-----

begin

    ! -----
    ! The remote_file_ is constructed in transport, from remote_name_ and
    ! counters. We should do that here, but...
    ! -----
    if pFTP.REMOTEFILE = EMPTY then
        pFTP.REMOTEFILE := pFTP.REMOTENAME
    endif

    if ARGV[1] = EMPTY then
        log ("Need filename to send\n")
        exit (3)
    endif

    fFILE := ARGV[1]
    if fFILE.EXIST = FALSE then
        log ("Cannot open file \"", ARGV[1], "\" (does not exist)\n")
        exit (3)
    endif

    if fFILE.READ = FALSE then
        log ("Cannot open file \"", ARGV[1], "\"\n")
        exit (3)
    endif

```

```

!-----
! Log in
! if pFTP.PASSWORD is not crypted -> bfLogIn()
! if pFTP.PASSWORD is crypted -> bfLogInTSI(sPARTNER,"partner",100)
!-----
bfLogIn()

!-----
! Send datafile (and flag file, if necessary), close and exit. Simple...
!-----
bfSendFiles()
bfFTP_close()
bfExit(EMPTY)
endbegin

function bfSendFiles ()
!-----
! Transfer directly to final filename.
!-----
tRem := build(pFTP.REMOTEFILE, pFTP.SPECIAL)
if not bfFTP_putas(fFILE.FULLNAME, tRem) then
    bfExit("Send failed")
endif
endfunction

!-----
! Exit program. If errortext is nonempty, there was an error.
! Log session on errors or if debugging.
!-----

function bfExit (tErrorText)
    if tErrorText <> EMPTY then
        log("INFO: ", tErrorText, NL)
    endif

    if tErrorText <> EMPTY or LOGLEVEL > 0 then
        log("\n*** FTP Session log follows: \n", tFTP_logtext(),
            "\n*** End of session log\n")
    endif

    if tErrorText <> EMPTY then
        exit(1)
    endif
    exit(0)
endfunction

```

Specific FTPS RTE functions

Creating a RTE program for ftpfetch and ftpsend fitted to secure transfer of data over a FTP session is possible:

Switch to secure FTP by just adding a simple function call to an initialisation function at the very beginning of your RTE. And after this call all subsequent transfers will be encrypted automatically. The only thing you have to remember is a single function name if you want to switch to secure FTP over SSL.

Hereafter a description of the FUNCTION:

bfSet_security_ssl

This function allows to link the C code with the RTE code so that a user can decide to start secure FTP communications. After this function is run successfully, all subsequent calls will be encrypted

Hereunder, the INPUT descriptions:

- **tPriv_key_file**
Name of the file containing the User's private key (it can contain prefix such as : tx: or ldap: or file:)
- **tPriv_key_pwd**
Name of the file containing the private key's password (or passphrase)
- **tCacert_fichier**
Name of the file containing the trusted certificates list (Server certificate) (it can contain prefix such as : tx: or ldap: or file:)
- **tClientcert_fichier**
Name of the file containing the client certificate (user certificate) (it can contain prefix such as : tx: or ldap: or file:)
- **tSecBase**
True if security base has been disabled, false otherwise.
This option allows to shortcut the security database if necessary, but it is recommended not to use it (it is useful mostly for problems diagnostic), thus you should put "FALSE" (upper or lowercase).
- **tClient_Auth_Server**
True if the client Must authenticate the server, false otherwise. This option allows to state whether the client should check the server's certificate or not.
- **tPartner_Name**
Name of partner used to find a default server certificate when tCacert_fichier field is empty. Notice that this field is not mandatory if tCacert_fichier is not empty, but if tCacert_fichier is empty, tPartner_Name MUST be filled to retrieve the Server's certificate.

```
begin
/*
We suppose here that the necessary parameters are given to the program when program
is launched. In other words the program will probably be called with a parameter
file given as follows : ./my_ssl_test -p <param_filename>
That's the reason why we have access to all necessary arguments for the function
"bfSet_security_ssl" since the beginning of the RTE program.
```

```

*/
bRet := bfSet_security_ssl(FTP.SSL.PRIVATE_KEY_FILE,
pFTP.SSL.PRIVATE_KEY_PASSWORD, pFTP.SSL.CACERT_FILE,
pFTP.SSL.CLIENTCERT_FILE, pFTP.SSL.SECURITY_BASE_DISABLED,
pFTP.SSL.SERVER_AUTH, pFTP.SSL.PARTNER_NAME )
if bRet then
    log("+++ ERROR: SSL Security initialization failed", NL )
    bfExit("Can't proceed, exiting")
else
    log("SSL Security setting....[SUCCESS]\n")
endif
endbegin

```

Specific SFTP RTE functions

TradeXpress users may use these functions to write or develop directly and dynamically RTE procedures to build their data transfer services, by relaying on the SFTP protocol. The RTE library for SFTP communication in TradeXpress offers primitive functions whose description is listed hereafter.



Most of these SFTP functions are SFTP equivalent to existing FTP functions.

Opening an closing a SFTP connection

These functions are used to open, establish, close a SFTP connection on a remote workstation

Function	Description
bfSFTP_Open (tHost, tUser, tPassword, tAccount)	Opens a SFTP connection
bfSFTP_Close ()	Closes an established connection
bfSFTP_Initialize ()	Initializes the system, rewinds log history, and loads user-defined errors.
bfSFTP_LogIn ()	Logs in if not already logged in.
bfSFTP_LogOut ()	Closes the connection if necessary.
bfSFTP_LogInPwd(tPassword)	Allows to log in to an opened SFTP connection
bfSFTP_LogInTSl (sBase, tType, nIndex)	Refer to the description of the bfLogInTSl(sBase, tType, nIndex) function on page 171.
bfSFTP_Open_Ex (tHost, tUser, tPrivFile, tPubFile, tPassphrase, tAccount)	Opens a SFTP connection using the public and private key files

Handling operations on the remote workstation

Users who need to handle and receive extra information from a file or a directory may use the following functions.


Function	Description
tfSFTP_pwd ()	Refer to the description of the tfFTP_pwd() function on page 170.
bfSFTP_del (tRemoteFile)	Removes a remote file
bfSFTP_rename (tFrom, tTo)	Renames a remote file (tFrom to tTo)
bfSFTP_CWD (tRemoteDir)	Refer to the description of the bfCWD(tDir) function on page 171.
bfSFTP_SizeOk (tRemote, tLocal)	Refer to the description of the bfSizeOK(tRemote, tLocal) function on page 171.
nfSFTP_size ()	Refer to the description of the nfFTP_size(tRemotefile) function on page 170.
tfSFTP_modtime (tRemoteFile)	Refer to the description of the tfFTP_modtime(tRemotefile) function on page 170.

Transferring files or data

These functions are used to send files from the local workstation to the remote workstation and receive files from the remote workstation to the local workstation

Function	Description
bfSFTP_putas (tLocal, tRemote)	Sends a file from the local workstation to the remote workstation
bfSFTP_getas (tRemote, tLocal)	Receives a file from the remote workstation to the local workstation
bfSFTP_nlist (tRemote, taArray)	Retrieves the list of files from a directory. Refer to the description of the bfFTP_dir(tRemote, taArray) ; function on page 171.

Managing Logs

Function	Description
tfSFTP_lastreply ()	Returns the last error message from the SFTP server in case of error only.  This function differs from the tfFTP_lastreply () which returns the last received answering from the FTP server in case of error or success.
bfSFTP_logtext ()	Returns the log history collected so far.
bfSFTP_setlogging (nLevel)	Refer to the description of the bfFTP_setlogging(nLevel) function on page 170.

Error management

Function	Description
taSFTP_ALL	Refer to the description of the FTP error arrays .
taSFTP_OK	
taSFTP_ERROR	

SFTP test program

This section describes the RTE programs used to test the RTE SFTP library. These programs are given here as examples.

Usable parameters

Use the following parameter to log in and to handle operations on the SFTP server.

- For ClientSendSFTP program

Parameter	Description
FTP.ACK.DIR	remote directory for acknowledge flag files
FTP.REMOTEFILE	remote file
FTP.REMOTEDIR	remote directory
FTP.HOST	host name
FTP.USER	login user
FTP.PASSWORD	login password
FTP.SSH.PRIVATE_KEY_REALPATH	private key file path
FTP.SSH.PUBLIC_KEY_REALPATH	public key file path
FTP.SSH.PRIVATE_KEY_PASSPHRASE	private key pass phrase
FTP.DIRNAME	directory name
RECIPIENT	index name of database PARTNER

- For ClientFetchSFTP program

Parameter	Description
FTP.ACK.DIR	remote directory for acknowledge flag files
FTP.LOCALFILE	local file
FTP.REMOTEDIR	remote directory
FTP.HOST	host name
FTP.USER	login user
FTP.PASSWORD	login password
FTP.SSH.PRIVATE_KEY_REALPATH	private key file path
FTP.SSH.PUBLIC_KEY_REALPATH	public key file path

Parameter	Description
FTP.SSH.PRIVATE_KEY_PASSPHRASE	private key pass phrase
FTP.DIRNAME	directory name
FTP.FILEMASK	file mask
CONNECTION	index name of database EDISEND

Windows environnement

- Connection in a « username and password » mode

```
ClientSendSFTP.exe
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.PASSWORD=edimanu
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP.exe
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.PASSWORD=edimanu
FTP.REMOTEDIR=dao
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Connection in a « public and private keys » mode

```
ClientSendSFTP.exe
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.SSH.PRIVATE_KEY_REALPATH=c:\demo\id_rsa
FTP.SSH.PUBLIC_KEY_REALPATH=c:\demo\id_rsa.pub
FTP.SSH.PRIVATE_KEY_PASSPHRASE=pass1
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP.exe
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.SSH.PRIVATE_KEY_REALPATH=c:\demo\id_rsa
FTP.SSH.PUBLIC_KEY_REALPATH=c:\demo\id_rsa.pub
FTP.SSH.PRIVATE_KEY_PASSPHRASE=pass1
FTP.REMOTEDIR=dao
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Connection "using a PARTNER database entry"

```
ClientSendSFTP.exe
RECIPIENT=SFTPAixpress
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP.exe
CONNECTION=SendSFTPAixpress
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Compilation: use the following command line to compile the rte files.

```
mktr ClientFetchSFTP.rte      %EDIHOME%\lib\libnetlib_sftp.lib
%EDIHOME%\lib\libcurl.lib      %EDIHOME%\lib\libssh2.lib
%EDIHOME%\lib\static_sslseay32.lib %EDIHOME%\lib\static_libseay32.lib
%EDIHOME%\lib\static_zlib.lib  gdi32.lib winmm.lib
```

Linux and other Unix environnements

- Connection in a « username and password » mode

```
ClientSendSFTP
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.PASSWORD=edimanu
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.PASSWORD=edimanu
FTP.REMOTEDIR=dao
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Connection in a « public and private keys » mode

```
ClientSendSFTP
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.SSH.PRIVATE_KEY_REALPATH=c:\demo\id_rsa
FTP.SSH.PUBLIC_KEY_REALPATH=c:\demo\id_rsa.pub
FTP.SSH.PRIVATE_KEY_PASSPHRASE=pass1
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP
FTP.HOST=aixpress
FTP.USER=edimanu
FTP.SSH.PRIVATE_KEY_REALPATH=c:\demo\id_rsa
FTP.SSH.PUBLIC_KEY_REALPATH=c:\demo\id_rsa.pub
FTP.SSH.PRIVATE_KEY_PASSPHRASE=pass1
FTP.REMOTEDIR=dao
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Connection « using a PARTNER database entry »

```
ClientSendSFTP
RECIPIENT=SFTPAixpress
FTP.REMOTEFILE=WAB_04.xml WAB.xml
ClientFetchSFTP
CONNECTION=SendSFTPAixpress
FTP.LOCALFILE=WAB_04.xml WAB.xml
```

- Compilation: use the following command line to compile the rte files.

```
mktr ClientSendSFTP.rte $EDIHOME/lib/libnetlib_sftp.a -L/usr/lib
$EDIHOME/lib/libcurl.a $EDIHOME/lib/libssh2.a $EDIHOME/lib/libz.a
$EDIHOME/lib/libssl.a $EDIHOME/lib/libcrypto.a -ldl

mktr ClientFetchSFTP.rte $EDIHOME/lib/libnetlib_sftp.a -L/usr/lib
$EDIHOME/lib/libcurl.a $EDIHOME/lib/libssh2.a $EDIHOME/lib/libz.a
$EDIHOME/lib/libssl.a $EDIHOME/lib/libcrypto.a -ldl
```

9.7 Appendix G: RTE SQL Access Library

9.7.1 Introduction

TradeXpress has a new facility that lets you access the relational databases using SQL syntax. Access is provided directly from the RTE translators. SQL access is provided by a set of RTE runtime functions, which access relational databases using SQL syntax. They also enable automatic database variable binding with the RTE variables.

The RTE SQL access library and its functions are designed in such a way that they can be used seamlessly with any commercially recognized relational database that provides SQL access. Depending on the underlying database system, the access can be local or networked.

The RTE SQL library gives access to a Oracle database in native mode (OCI), or to any other SQL-type database via an ODBC or JDBC connector.

An RTE translator using RTE SQL access becomes a database client program, and its behavior, resource requirements, and database access license issues are like those of any general purpose database access client within the database environment that is being used.

9.7.1.1 Requirements

RTE translators that use RTE SQL access require, at a minimum, the database provider's runtime environment (Oracle7 / Oracle8) to be installed and configured.

9.7.1.2 Compilation

An RTE translator using the RTE SQL access library is compiled like normal RTE translators. The linking phase requires special libraries for database access functions. The ccargs file (**EDIHOME/lib/ccargs**) can be used with TradeXpress. Here are examples of the ccargs file on different platforms:

```
Windows NT:
libruntime.lib          liblogsystem.lib          libedi.lib
Z:\orant\oci73\lib\msvc\ociw32.lib libessqlora.lib libodbsys.lib
wssock32.lib
SUN SOLARIS
-L/u/tradexpress4.0/lib -L/usr/local/oracle7/lib -lruntime -llogsystem
-ledi -lcIntsh -lm -lsocket -lnsl -lessqlora -lodbsys -lruntime
IBM AIX:
-L/u/tradexpress4.0/lib -L/home/oracle7/lib -llogsystem -ledi -lcIntsh
-lm -lessqlora -lodbsys -lruntime
```

9.7.1.3 Execution Environment

The database clients using shared libraries often require special user environment settings to be able to locate and access database libraries and resources correctly. The following environment settings can be used with the TradeXpress and Oracle 7 / Oracle 8 databases. These settings are found in the oraenv file located at **\$EDIHOME/lib/oraenv**.

```
ORACLE_HOME=/usr/local/oracle7
export ORACLE_HOME
PATH=$PATH:$ORACLE_HOME/bin
NLS_LANG=American_America. WE8ISO8859P1
ORACLE_BASE=$ORACLE_HOME
ORACLE_SID=ES
ORA_NLS=$ORACLE_HOME/ocommon/nls/admin/data
ORACLE_DOC=$ORACLE_HOME/doc
NLS_ADMIN=$ORACLE_HOME/network/admin
ORACLE_TERM=$TERM
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ORACLE_HOME lib
export PATH NLS_LANG ORACLE_BASE ORACLE_SID ORA_NLS ORACLE_DOC
NLS_ADMIN
ORACLE_TERM LD_LIBRARY_PATH
```


9.7.2 RTE SQL Access Library Reference List

9.7.2.1 bfSqlOracle ("DBID", "User", "Passwd", "DBLink")

Function	Establishes connection to Oracle database.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Used with other functions to identify connection. Value can be any string. • User: Oracle user ID. If the operating system authentication is used, set this to <code>"/"</code>. • Passwd: Oracle user's password. If the operating system authentication is used, leave this empty; i.e., <code>""</code>. • DBLink: Database connection ID (SQLNet V2 connect descriptor).
Return value	TRUE, if succeeds.

9.7.2.2 bfSqlSet ("DBID", "SqlID", "SQLString")

Function	Creates new or modifies existing SQL command.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Value can be any string. • SQLString: SQL command string. RTE numeric and text variables can be used as part of the SQL command. For example: <div data-bbox="485 1939 1382 2009" data-label="Text"> <pre>"UPDATE table_x SET field1 = :tTclVariable WHERE field2 >= :nTclVariable2"</pre> </div>

Return value	TRUE, if succeeds.  If the SQL command is an Oracle database administration command (for example, CREATE TABLE), it is executed immediately.
--------------	--

9.7.2.3 tfSqlGetField("DBID","SqlID","Field")

Function	Retrieves the value of the fetched SQL field as a text string.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function. • Field: SQL field name. Fields are the result of an SQL SELECT command. Value of the field is available after successful bfSqlOpen and bfSqlFetch commands.
Return value	Value of fetched field as an RTE text string.

9.7.2.4 nfSqlGetField("DBID", "SqlID","Field")

Function	Retrieves the value of the fetched SQL field as a numeric string.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function. • Field: SQL field name. Field or fields are the result of an SQL SELECT command. Value of the field is available after successful bfSqlOpen and bfSqlFetch commands.
Return value	Value of the fetched field as an RTE numeric value.

9.7.2.5 bfSqlExec("DBID", "SqlID")

Function	Executes an SQL modify command (INSERT, UPDATE, or DELETE).
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function.
Return value	TRUE, if succeeds, otherwise FALSE.

9.7.2.6 bfSqlOpen("DBID", SqlID")

Function	Executes an SQL SELECT command. Rows will be selected from the database for further fetching.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function.

Return value	TRUE, if succeeds, otherwise FALSE.
--------------	-------------------------------------

9.7.2.7 bfSqlFetch("DBID", "SqlID")

Function	Fetches the next row, produced by the previous bfSqlOpen. Can be called repeatedly until the last row is fetched.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function.
Return value	TRUE, if succeeds. FALSE, if there are no rows left.

9.7.2.8 bfSqlFetchArray("DBID", "SqlID", taVar)

Function	Fetches the next row, produced by the previous bfSqlOpen, directly to an existing RTE text array variable. Can be called repeatedly until the last row is fetched.
Parameters	<ul style="list-style-type: none"> • DBID: Database ID. Created with the bfSqlOracle function. • SqlID: SQL command identification. Created with the bfSqlSetNew function. • taVar: RTE text array that will receive fetched fields as name-value pairs. The name of the field is the index of the array variable, and the fetched value is the value of the array variable.
Return value	TRUE, if succeeds. FALSE, if there are no rows left.

9.7.2.9 bfSqlFree("DBID", SqlID")

Function	Frees the SQL command from memory.
Parameters	<ul style="list-style-type: none"> • "DBID" Database ID. Created with the bfSqlOracle function. • "SqlID" SQL command identification. Created with the bfSqlSetNew function.
Return value	TRUE, if succeeds, otherwise FALSE.

9.7.2.10 bfSqlClose("DBID")

Function	Closes the database connection. Frees all open SQL commands and closes the database connection.
Parameters	DBID : database connection ID. Created with the bfSqlOracle function.
Return value	TRUE, if succeeds, otherwise FALSE.

9.7.3 ODBC interface



Available for TradeXpress 4.4 and higher

9.7.3.1 Installation

Requirements: ODBC interface need working ODBC connector to the related database (see NT manuals to setup ODBC connection).

Libessqlodbc.lib must exists in \$EDIHOME/lib directory.

9.7.3.2 Use

Odbc interface is very similar with Oracle interface the only difference is that function "bfSqlOracle" is replaced by "bfSqlOdbc" that use exactly the same parameters (see "9.7.2.1 - bfSqlOracle ("DBID", "User", "Passwd", "DBLink")" on page 181).

9.7.3.3 Compilation

To compile your Rte program use the following command line:

```
mktr -Codb32.lib test_odbc.rte libessqlodbc.lib
```

9.7.4 JDBC Interface



Available for TradeXpress 4.4 and higher

9.7.4.1 Installation

Requirements :

JDBC interface needs a working Java environment (JDK 1.2 to 1.5.0) and a JDBC driver installed for the related database, depending on this driver, you may also need a working database client (if not "Thin" driver).

No installation required, as the setup is included when installing TradeXpress (according to you licence)

Configuration

Rename the file "options.xmlpl" in "options.txt" in \$EDIHOME/lib directory. This file contains locations to used classes, libraries and drivers. Following, an example of "options.txt":

```
[OPTION]
-Djava.compiler = NONE
# Specify the path to jdbc interface programs and the path to jdbc
driver
-Djava.class.path=/usr/TradeXpress/lib; /usr/jdbc/classes12.jar
# for debugging information
-verbose:class,gc,jni
[DRIVER]
# jdbc driver name
oracle.jdbc.driver.OracleDriver
```




"#" puts the line into comment.

Update then the environment variable `$LD_LIBRARY_PATH` (PATH on Windows OS, `SHLIB_PATH` on HP) to your `libjvm.a` (or `.lib`, `.so`) emplacement.

It is possible to have access to different JDBC databases from two RTE in the same environment. It is even possible on the same RTE.

You have to:

- Own all the jar available for all the users, usually in `$EDIHOME/lib`.
- Update the classpath inside the `$EDIHOME/lib/options.txt` to point to the jar files and `$EDIHOME/lib` where you can find `RTE_JDBC.class` used for the connection between JDBC and RTE.

Sun is maintaining a driver database at <http://developers.sun.com/product/jdbc/drivers>

Under the `$EDIHOME/lib/options.txt` you have to list all the classes you want to be able to use. For example, to use oracle and SQLServer you will have:

```
oracle.jdbc.driver.OracleDriver
com.mysql.jdbc.Driver
#org.postgresql.Driver
#com.microsoft.jdbc.sqlserver.SQLServerDriver
```

Uncomment only the ones which are used, otherwise we will try to setup the others too.

9.7.4.2 Use

Jdbc interface is very similar to Oracle interface the only difference is that the function "`bfSqlOracle`" is replaced by "`bfSqlJdbc`" that uses exactly the same parameters (see "9.7.2.1 - `bfSqlOracle`" ("DBID", "User", "Passwd", "DBLink")" on page 181).

```
bfSqlJdbc("DBID", "User", "Passwd", "DbLink")
```

Function: Establishes a connection to a JDBC database

Parameters:

- **DBID** : Database ID. Used with other functions to identify connection. Value can be any string.
- **User**: Database user ID.
- **Passwd**: Database user's password.
- **DBLink**: Database connection ID.

You need to use a specific URL to connect to your database.

This all depend on the database used. Here are some examples:

```
jdbc:odbc:maBase;CacheSize=30;ExtensionCase=LOWER
jdbc:mysql://localhost/maBase
jdbc:oracle:oci8@:maBase
jdbc:oracle:thin@://localhost:8000:maBase
jdbc:sybase:Tds:localhost:5020/maBase
jdbc:microsoft:sqlserver://
```

```
fulbert:1433;DatabaseName=synchro;SelectMethod=Cursor
```

For example, to initialise a connection to a mysql database, you will use the following:

```
bfSqlInit()  
bfSqlJdbc("DBID", "user", "passwd", "jdbc:mysql://localhost/maBase")
```

9.7.4.3 Compilation

To compile your Rte program use the following command line :

Unix

```
mktr -C ljvm -C L/path/to/libjvm.a sample.rte -lessqljdbc
```

Windows

```
mktr.exe source.rte libessqljdbc.lib [path-to-]\jvm.lib
```

The jvm.lib is usually supplied with Oracle (for instance: C:\oracle\ora92\jdk\lib\jvm.lib), or the Java sdk (for instance: C:\Sun\AppServer\jdk\lib\jvm.lib).

9.7.4.4 Example of an RTE Program with SQL Access

```
begin  
    !-----  
    !-----OPENING CONNECTION  
    bfSqlInit()  
    if sfSqlOracle( "DB", "/", "", "sohvi_ES-tcp" ) = FALSE then  
        print( "bfSqlSet: Failed", NL )  
    endif  
  
    !-----  
    !-----SELECT  
    if bfSqlSet( "DB", "SQL", "SELECT IN_INDEX,EX_STATUS FROM ESD_SYSLOG" ) = FALSE  
then  
        print ( "bfSqlSet: Failed", NL )  
    endif  
  
    if bfSqlOpen( "DB", "SQL" ) = FALSE then  
        print( "bfSqlOpen: Failed", NL )  
    endif  
  
    while bfSqlFetch( "DB", "SQL" ) = TRUE do  
        print( nfSqlGetField( "DB", "SQL", "IN_INDEX" ), " ",  
            tfSqlGetField ( "DB", "SQL", "EX_STATUS" ), NL )  
    endwhile
```

```

!-----
!-----DROP TABLE
if bfSqlSet( "DB", "SQL", "DROP TABLE MY_TABLE" ) = FALSE then
    print( "bfSqlSet: DROP TABLE Failed", \ NL )
endif

!-----
!-----CREATE TABLE
    if bfSqlSet( "DB", "SQL", "CREATE TABLE MY_TABLE (F_TEXT VARCHAR(50),
F_NUM NUMBER(10) )" ) = FALSE then
        print( "bfSqlSet: CREATE TABLE Failed", NL )
    endif

!-----
!-----INSERT TABLE
tText := "SOME TEXT"
nNum := 1234
if bfSqlSet( "DB", "INSERT", " INSERT INTO MY_TABLE (F_TEXT,F_NUMVALUES)
(:tText, :nNum)" ) = FALSE then
    print ( "bfSqlSet: INSERT TABLE Failed", NL )
endif

if bfSqlExec( "DB", "INSERT" ) = FALSE then
    print( "bfSqlExec: INSERT TABLE Failed", NL )
endif

if bfSqlCommit( "DB", "INSERT" ) = FALSE then
    print( "bfSqlCommit: INSERT TABLE Failed", NL )
endif

if bfSqlSet( "DB", "SELECT", "SELECT F_TEXT, F_NUM FROM MY_TABLE" ) = FALSE
then
    print( "bfSqlSet: SELECT1 TABLE Failed", NL )
endif

if bfSqlOpen( "DB", "SELECT" ) = FALSE then
    print( "bfSqlOpen: SELECT1 TABLE Failed", NL )
endif

while bfSqlFetch( "DB", "SELECT" ) = TRUE then
    print( tfSqlGetField( "DB", "SELECT", "F_TEXT" ), " ",
nfSqlGetField ( "DB", "SELECT", F_NUM" ), NL )
endwhile

```

```

!-----
!-----UPDATE TABLE
tText := "CHANGED"
nNum := 5678
if bfSqlSet( "DB", "SQL", "UPDATE MY_TABLE SET F_TEXT = :tText, F_NUM = :nNum"
) = FALSE then
    print( "bfSqlSet: UPDATE TABLE Failed", NL )
endif

if bfSqlExec( "DB", "SQL" ) = FALSE then
    print( "bfSqlExec: UPDATE TABLE Failed", NL )
endif

if bfSqlCommit( "DB", "SQL" ) = FALSE then
    print( "bfSqlCommit: UPDATE TABLE Failed", NL )
endif

if bfSqlOpen( "DB", "SELECT" ) = FALSE then
    print( "bfSqlOpen: SELECT2 TABLE Failed", NL )
endif

while bfSqlFetch( "DB", "SELECT" ) = TRUE do
    print( "tfSqlGetField( "DB", "SELECT", "F_TEXT" ), " ",
    nfSqlGetField ( "DB", "SELECT", "F_NUM" ), NL
endwhile

!-----
!-----INSERT MORE ROWS
tText := "Third"
nNum := 3333
if bfSqlExec( "DB", "INSERT" ) = FALSE then
    print( "bfSqlExec: INSERT TABLE Failed", NL )
endif

if bfSqlCommit( "DB", "INSERT" ) = FALSE then
    print( "bfSqlCommit: INSERT TABLE Failed", NL )
endif

tText := "Second"
nNum := 2222
if bfSqlExec( "DB", "INSERT" ) = FALSE then
    print( "bfSqlExec: INSERT TABLE Failed", NL )
endif

if bfSqlCommit( "DB", "INSERT" ) = FALSE then
    print( "bfSqlCommit: INSERT TABLE Failed", NL )
endif

!-----

```

```

!-----FETCH ARRAY
if bfSqlOpen( "DB", "SELECT" ) = FALSE then
    print( "bfSqlOpen: SELECT3 TABLE Failed", NL )
endif

while bfSqlFetchArray( "DB", "SELECT" ) = TRUE do
    print( taDummy )
endwhile

if bfSqlSet( "DB", "SQL", "SELECT * FROM ESD_SYSLOG" ) = FALSE then
    print( "bfSqlSet: Failed", NL )
endif

if bfSqlOpen( "DB", "SQL" ) = FALSE then
    print( "bfSqlOpen: SELECT4 TABLE Failed", NL )
endif

while bfSqlFetchArray( "DB", "SQL", taDummy ) = TRUE do
    while tIndex in taDummy do
        print( tIndex, "=", taDummy[tindex], NL )
    endwhile
endwhile

!-----CLOSING CONNECTION
if bfSqlFree( "DB", "SQL" ) = FALSE then
    print( "bfSqlFree: Failed", NL )
endif

! bfSqlDump()
if bfSqlClose( "DB" ) = FALSE then
    print( "bfSqlClose: Failed", NL )
endif
endbegin

```

9.8 Appendix H: XML to XML translator

9.8.1 Principles

9.8.1.1 Input and output messages

Traditionally, while building an RTE translator, you used to indicate the direction and grammar your syntax translator deals with:

message "... " receiving	to translate an XML document to a flat text format.
message "... " building	to translate a flat text file into XML.

If you want to build an XML message from another XML file using this method, you will have to develop two different translators and use an intermediary format within a flat file.

The XML to XML approach is a little different: the RTE is now able to translate directly a document from XML to XML. Thus, both message building and message receiving directives will appear in only one translator source code. All along the RTE source file, the actions will be marked either as “incoming” or “outgoing”.

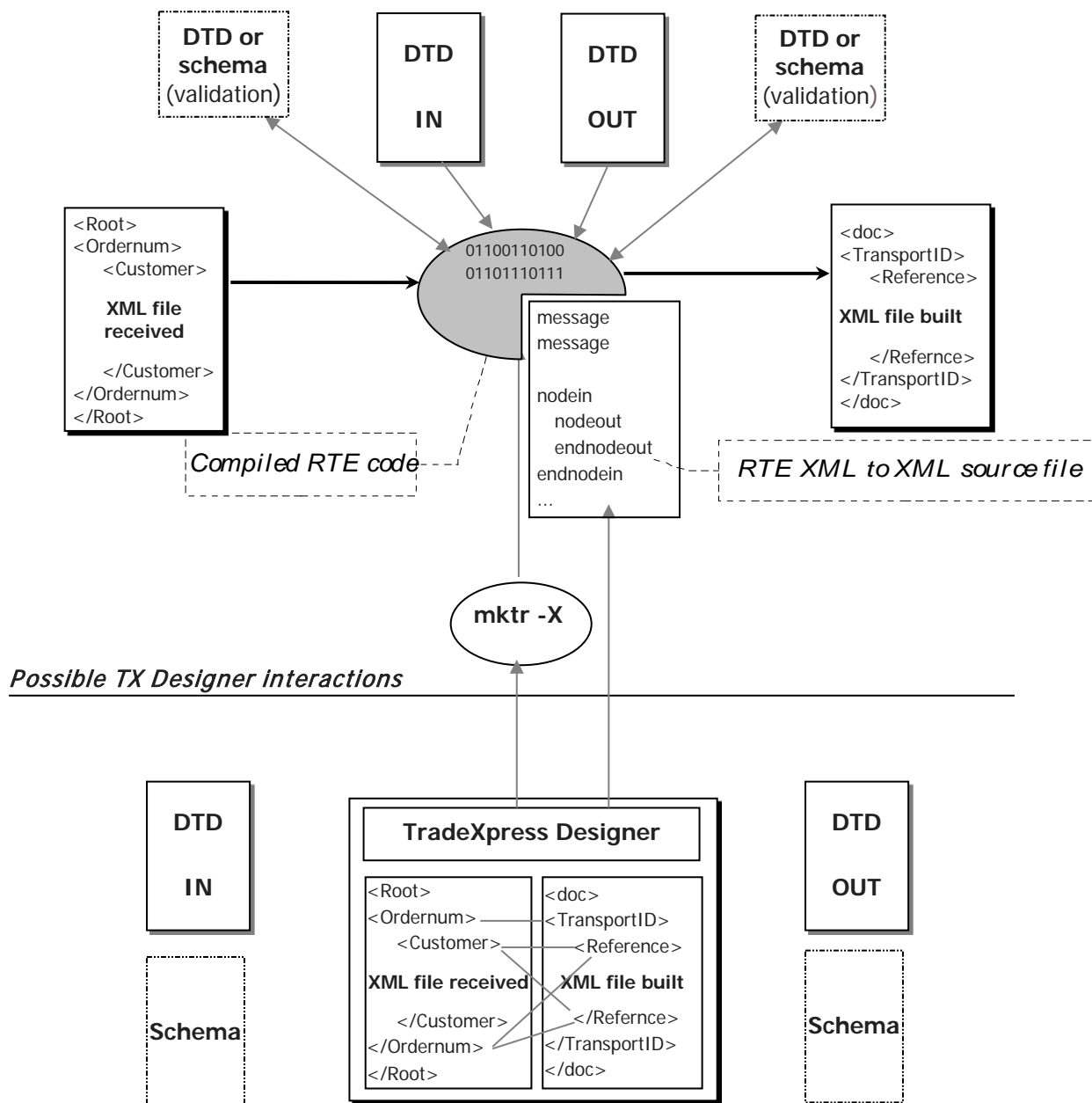
As an example, the “MESSAGE” keyword will be post-fixed with its direction: “MESSAGE_IN” or “MESSAGE_OUT”.



XML compiling process requires a DTD in order to prepare the parsing. The validation process is independent and uses its own DTD or schema declared using the schema keywords.

9.8.1.2 Translation process

The schema below describes the whole XML to XML message process.



9.8.1.3 How the translation works

Such as the actual translation in simple building mode (*e.g. flat file to XML*), the translation process follows the input message. Instead of processing incoming message line by line, it parses XML message node by node. Your translator specifies each action you want to execute to build your new XML tree.

9.8.2 Syntax

XML to XML syntax is very close to other RTE translators. All your practice regarding RTE programming will remain unchanged. The only differences you have to notice regarding the syntax are:

- Header statements
- Segments are now called `nodes` and indicate its direction (in or out)
- MESSAGE keyword is now post-fixed with “_IN” or “_OUT”

9.8.2.1 Header statements

Message receiving, message building

The message statement is a TradeXpress internal operation. It is used for both building and receiving translations and is absolutely required.

The message statements will also determine the translation mode to use: by calling two message definitions, it automatically switches from normal mode to “syntax to syntax” mode.

Example:

```
message “incoming.dtd” receiving, passthru
message “outgoing.dtd” building
```



passthru keyword is absolutely required by XML to XML translators.

schema keyword

schema header function is only used by XML translators to specify which grammar file will be used by the Xerces parser for validation purposes.

In the same way than for “message,” the validation directive in XML to XML mode, must specify which is the receiving grammar file and which is the building one. This directive is only required while validating messages.

Example:

```
schema “incoming.dtd” validating , receiving
schema “outgoing.dtd” validating , building
```

9.8.2.2 Nodes

To avoid confusions with the old RTE syntax (including `segmentii` for example) and to be closed to XML vocabulary, the `segment` directive has been replaced by “**node**” keyword on which we added the direction “in” or “out”. Thus, we have four specific keywords:

- `nodein`
- `endnodein`
- `nodeout`
- `endnodeout`



The `segment` keyword is not allowed in XML to XML translators.

nodein

This keyword indicates we are going to process a node from the incoming message. The syntax is the same as those used for `segment` statement.

```
nodein Sdata gGA
...
endnodein
```

Exactly as usual, you can access current node elements and attributes using 'eE' and 'eA' variables:

```
nodein Sdata gGA
    tElementData := eEdata
    tAttributeData := eAattribute1
endnodein
```

nodeout

The syntax is strictly the same than previous, but “`nodeout`” replaces `nodein`.

```
...
    nodeout Sdata gGA
    ...
    endnodeout
...
```

node statements rules

The `nodein` statement is only allowed when standing alone (not included in another statement).

As a contrary, `nodeout` can be included in all other statements except itself:

- `begin / endbegin`
- `nodein / endnodein`
- `end / endend`

9.8.2.3 Printing or validating

message

The `message` keyword is mainly used for validation (e.g. `valid(MESSAGE)`) or message printing (e.g. `print(MESSAGE)`).

This common use will not really change, but you have to indicate which message you want to use:

- `MESSAGE_IN` stands for incoming message.
- `MESSAGE_OUT` stands for built message.

Consequently, the `message` keyword is not allowed anymore in syntax to syntax mode.

segments

It is also the same regarding segment processing: `SEGMENT` keyword use:

- `SEGMENT_IN` for segment from incoming message,
- `SEGMENT_OUT` for segment from built message.

9.8.3 Building an XML to XML translator

Step 1 – Check your XML documents

First of all, check and validate your XML documents (dtd, schema, XML message). Many errors encountered while processing XML documents are due to unsuitable XML documents.

Step 2 – Analyze your data sources

All along the translator writing, it is recommended to follow the incoming XML file order. Highlight on this file all data sources for each outgoing node. Then you could be able to establish a complete matching between sources and targeted data.

Step 3 – Initialize code

Place your DTDs and schemes in the directory of your choice and write your message directives (and schema for validation purposes) including the path to your directory containing dtd and xsd files.

If you are using TradeXpress Designer, this task is automatically carried out within references features.

- `message "/path/to/incoming.dtd" receiving, passthru`
- `message "/path/to/outgoing.dtd" building`
- `schema "/path/to/incoming.xsd" validating, receiving`



The order for these declarations does not have any importance

Step 4 – Begin your translation

Using the source and target data match, and following the incoming message order, create each source node. Therefore, you will be able to store data into variables, and then use their values later.

Remember that you can include an `nodeout` in each statement. Thus, you can choose the right time to build your nodeout.

Step 5 – Check your code and search for redundancies

As you have ever seen, nodeout statements can be included into your functions.

To conclude, notice that XML to XML mode uses TradeXpress XML module technology: please refer to the “9.6.2- Pass-through Mode in RTE” section, on page 150, to detect possible errors.

Step 5 – Compilation

There is no difference between XML translator compiling and XML to XML. Use the -X option just as usual:

```
mktr -X <rte file>.rte
```

9.9 Appendix I: RTE LDAP Library

LDAP is a protocol based on the X500 standard allowing access to a directory; we usually call it an LDAP directory.

This section is about the implementation of the RTE LDAP library function enabling to use this protocol.

9.9.1 Prerequisites

Before using the functions allowing access to a LDAP directory, it is necessary to install an LDAP client.

You need five libraries to compile an RTE program using Ldap:

- liblutil.a (.lib on Windows)
- libldap.la (.lib)
- liblber.la (.lib)
- libsasl.la (.lib)
- libconn_ldap.a (libconn_ldap.lib)

Except for libconn_ldap, the libraries are supplied with the Ldap client or in some cases are parts of the Operating System. On Windows these libraries are installed in the `$EDIHOME/lib` directory, however it is better to use the libraries supplied with the Ldap client if any.

On Unixes check that the `$LD_LIBRARY_PATH` environment variable definitely points at the directory or directories which contain these libraries.

9.9.2 Compiling your RTE program

On UNIX:

```
mktr test_ldap.rte -lldap -llber -lsasl -lconn_ldap
```

On Windows:

```
mktr test_ldap.rte libldap.lib liblber.lib libsasl.lib libconn_ldap.lib  
/link/NODEFAULTLIB:LIBCMT
```

9.9.3 RTE Library function

The functions available are the same as those already existing to access the databases. Here is the list of the functions available.

9.9.3.1 bfLdapInit

Syntax	bfLdapInit (tLdap_host, nLdap_port, tLdap_Login, tLdap_password);
Description	Ends the connection to the ldap server.
Parameters	None
Returns	TRUE if successful, otherwise FALSE.

9.9.3.2 bfLdapAdd

Syntax	bfLdapAdd (taData)
Description	Adds an input in the Ldap base.
Parameters	A table containing the element and its properties to be inserted.
Returns	TRUE if successful, FALSE in case of failure, with an error message to the standard output. The failure is not blocking and do not stop the execution of the rest of the program.

9.9.3.3 bfLdapDelete

Syntax	bfLdapDelete (taData)
Description	Removes an entry in the Ldap base.
Parameters	A table containing the element to be deleted.
Returns	TRUE if successful, FALSE in case of failure, with an error message to the standard output. The failure is not blocking and do not stop the execution of the rest of the program.

9.9.3.4 bfLdapSearch

Syntax	bfLdapSearch (tBase_parameter, tFilter_parameter)
Description	Search of elements belonging to a Ldap base.
Parameters	<ul style="list-style-type: none"> tBase_parameter: search root. tFilter_parameter: search filter. <p>For further details regarding search filters syntax, see documentation provided with your LDAP client.</p>
Returns	An error may occur if the tBase_parameter parameter is erroneous. In this case, the value -1 is returned. Otherwise, the number of results found is returned.

9.9.3.5 bfLdapFetchArray

Syntax	bfLdapFetchArray (taResult)
Description	Selects the next line in the response table, and return the value found.
Parameters	taResult : Rte table in which the various returned elements will be stored.
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.6 bfLdapFetch

Syntax	bfLdapFetch ()
Description	Selects the next line in the responses table.
Parameters	None
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.7 bfLdapGetField

Syntax	bfLdapGetField (taResult)
Description	Returns the current line in the response table.
Parameters	taResult : table containing the various retrieved values.
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.8 bfLdapModify

Syntax	bfLdapModify (taData)
Description	Modification of an entry of the Ldap base.
Parameters	taData : table containing the modified element.
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.9 nflDapgetAttList

Syntax	nfLdapGetAttlist(naRteResult)
Description	Retrieve attributes list of the current entry.
Parameters	naRteResult: after successful execution, will contains all attributes names and for each, values count (naRteResult["AttributeName"] = values count)
Returns	-1 in case of error, with an error message to the standard output, otherwise returns the attributes count.

9.9.3.10 bfLdapGetFirstAttVal

Syntax	bfLdapGetFirstAttVal (taRteResult, tAttributeName)
--------	--

Description	Retrieve the first value of current entry attribute.
Parameters	<ul style="list-style-type: none"> • <code>taRteResult</code>: after successful execution, index "<code>tAttributeName</code>" will contains first value of the attribute "<code>tAttributeName</code>" • <code>tAttributeName</code>: name of the attribute
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.11 `bfLdapGetNextAttVal`

Syntax	<code>bfLdapGetNextAttVal (taRteResult, tAttributeName)</code>
Description	Retrieve the next value of current entry attribute.
Parameters	<ul style="list-style-type: none"> • <code>taRteResult</code>: after successful execution, index "<code>tAttributeName</code>" will contains next value of the attribute "<code>tAttributeName</code>". • <code>tAttributeName</code>: name of the attribute.
Returns	FALSE in case of error, with an error message to the standard output, otherwise TRUE.

9.9.3.12 `nfLdapGetFirstAttBinVal`

Syntax	<code>nfLdapGetFirstAttBinVal (tBinaryFile, tAttributeName)</code>
Description	Retrieve the first value of current entry attribute in binary format.
Parameters	<ul style="list-style-type: none"> • <code>tBinaryFile</code>: after successful execution, binary attribute will be stored in "<code>tBinaryFile</code>" • <code>tAttributeName</code>: name of the attribute.
Returns	-1 in case of error, with an error message to the standard output, otherwise size of attribute.

9.9.3.13 `nfLdapGetNextAttBinVal`

Syntax	<code>nfLdapGetNextAttBinVal (tBinaryFile, tAttributeName)</code>
Description	Retrieve the next value of current entry attribute in binary format.
Parameters	<ul style="list-style-type: none"> • <code>tBinaryFile</code>: after successful execution, binary attribute will be stored in "<code>tBinaryFile</code>" • <code>tAttributeName</code>: name of the attribute.
Returns	-1 in case of error, with an error message to the standard output, otherwise size of attribute.

9.9.4 Example of RTE file using all the functions available

```
! Compile this file with command line:
! mktr test_ldap.rte -lldap -llber -lconn_ldap

begin

    !=== General parameters ===

    tLdap_Host := "localhost"
    nLdap_Port := 389
    tLdap_login := "cn=name, o=society, c=country"
    tLdap_Passwd := "password"

    !=== Action: Connexion to ldap database ===

    if bfLdapInit (tLdap_Host, nLdap_Port, tLdap_login, tLdap_Passwd) = TRUE then
        print (">>>> Connexion effectuée.", NL)
    endif

    !=== Action: Add an entry ===

    taData["dn"] := "cn=name, o=society, c=country"
    taData["cn"] := "name"
    taData["mail"] := "e-mail"
    taData["objectClass"] := "person"

    if bfLdapAdd (taData) = TRUE then
        print ("Add: success.", NL)
    else
        print ("Add: failure.", NL)
    endif

    remove (taData)

    !=== Action: Look for added entry (Fetch + GetField) ===

    tBase := "o=society, c=country"
    tFilter := "objectClass=person"
    nNbResult := nfLdapSearch (tBase, tFilter)
    print ("Search returned ", nNbResult, " results.", NL)

    if nNbResult <> -1 then
        while bfLdapFetch (taResult) do
            bfLdapGetField (taResult)
            print ("Result", NL)
            while tIndex in taResult do
                print ("\t", tIndex, ": ", taResult[tIndex], NL)
            endwhile
        endwhile
    endif
end
```

```

        endwhile
    endif

    remove (taResult)

    !=== Action: Look for added entry (FetchArray) ===

    tBase := "o=society, c=country"
    tFilter := "objectClass=person"

    if bfLdapSearch (tBase, tFilter) = TRUE then
        print (">>>> Search finished", NL)
    endif

    while bfLdapFetchArray (taResult) do
        print ("Result:", NL)
        while tIndex in taResult do
            print ("\t", tIndex, ": ", taResult[tIndex], NL)
        endwhile
    endwhile

    remove (taResult)

    !=== Action: Remove added entry ===

    taData["dn"] := "cn=name, o=society, c=country"

    if bfLdapDelete (taData) = TRUE then
        print ("Delete: success.", NL)
    else
        print ("Delete: failure.", NL)
    endif

    !=== Action: Close connexion ===

    bfLdapClose ()

endbegin

```




Headquarters

6, rue du Moulin de Lezennes - Immeuble le Verdi
BP 10215 59654 Villeneuve d'Ascq
Tel: +33 (0) 3 20 41 48 00

Paris

69/71, rue Beaubourg
75003 Paris
Tel: +33 (0) 1 77 45 41 80

In France, Generix Group also has offices in Rennes and Clermont -Ferrand.

Generix Group is present abroad thanks to its subsidiaries
(Belgium, Brazil, Spain, Italy, Portugal).

