

# Programmer's Manual

## RT-Expert™

Copyright ©1996-2005

**BellHawk Systems Corporation**

45 River Street  
Millbury, MA 01527  
508-865-8070  
support@BellHawk.com  
<http://www.BellHawk.com>

## Table of Contents

<b>INTRODUCTION</b>	<b>3</b>
<b>2. ELEMENTAL DATA TYPES IN DSL</b>	<b>5</b>
<b>3. FORMAL STRUCTURE OF A RULE SUBROUTINE</b>	<b>6</b>
<b>4. VARIABLES, DATA TYPES, AND CONSTANTS</b>	<b>8</b>
4.1 DECLARATION OF VARIABLES	8
<i>Reserved Keywords in DSL</i>	8
4.2 CONSTANTS	9
4.3 ATTRIBUTES	12
<b>5. FORMAL STRUCTURE OF RULES AND STATEMENTS</b>	<b>13</b>
5.1 RULES	13
5.2 ARITHMETIC AND OTHER OPERATORS	16
<i>Numeric Operators:</i>	16
<i>Boolean Operators:</i>	16
<i>String/Character Operators:</i>	16
<i>Symbolic Operators:</i>	17
<i>Time Operators:</i>	17
5.3 STATEMENTS	17
5.3.1 <i>Assignment Statements</i>	17
5.3.2 <i>Print Statements</i>	18
5.3.3 <i>Subroutine Statements</i>	19
5.3.4 <i>FOR and WHILE Statements</i>	20
5.3.5 <i>Pragma Statements</i>	20
5.3.6 <i>Initialization Statements</i>	22
5.3.7 <i>Other Statements</i>	22
<b>6. CALLING C SUBROUTINES FROM DSL</b>	<b>22</b>
<b>7. CONTROLLING THE FIRING OF RULES</b>	<b>24</b>
<b>8. DEBUGGING, VALIDATION, AND TESTING</b>	<b>27</b>
<b>APPENDIX A - FUNCTIONS AND PROCEDURES DECLARED IN RTEPERT.AFS</b>	<b>28</b>
<b>APPENDIX B - DSL ROUTINES FOR USER SUBROUTINES</b>	<b>29</b>
ROUTINES FOR HANDLING UNDEFINED VALUES IN DSL SUBROUTINES	29
<b>APPENDIX C - UTILITIES FOR PROGRAMMING WITH RT-EXPERT</b>	<b>31</b>
THE DELDSLCS.COM UTILITY	31
THE AFS2H FILE CONVERSION UTILITY	31
<b>INDEX</b>	<b>33</b>

## Introduction

RT-Expert enables expert systems rules to be easily integrated into information processing and decision support systems such as:

- On-line advisory systems
- Scheduling and planning systems
- Diagnostic and warning systems
- Real-time simulation and modeling systems
- On-line data collection, processing, and display systems.
- Automated test systems
- Smart process control systems
- Geographic information systems

RT-Expert consists of a compiler that converts modules written in expert systems rules into C callable subroutines and a C library that contains the rules execution mechanism.

With RT-Expert, users can break down their rules into multiple subroutines to facilitate ease of integration. When an application calls upon an RT-Expert generated subroutine, a mechanism is automatically invoked to control the firing of the rules. Rules are triggered by changes to data and also by the passage of time. The mechanism also supports goal directed chaining as well as depth first searches. Because rules are compiled into C code, execution is much faster than with inference engines that interpret the rules.

Rules based programming, used in expert systems, reduces the size of decision code for an application by a factor of up to 10:1. Previously, this technology was difficult to integrate into an application because of the complexity of interfacing with expert systems shells and inference engines. Now, with RT-Expert, the interface is as simple as calling a C subroutine.

The language used to write rule subroutines is called Decision Support Language™, referred to here as DSL. In DSL if...then...else.. rules are used to describe the event driven actions of a rule subroutine. DSL rules are triggered by changes to data and by the passage of time. The rules may call upon procedures written in C.

Rule subroutines written in DSL are referred to as RTSUBs. The DSL compiler translates each RTSUB into a set of C code subroutines. These are then compiled using a standard C compiler and linked with the RT-Expert library containing the rules execution mechanism.

This manual assumes that users have already read the previous section, Tutorial on writing RT-Expert RTSUBs in Decision Support Language rules.

An example RTSUB is shown below:

```
RTSUB temperature_control(temperature: ATTRIB FLOAT;
                        control: ATTRIB SYMBOLIC;
                        heater: ATTRIB SYMBOLIC;
                        lo_limit,hi_limit: IN FLOAT) IS
DECLARE
  RUN, STOP, ON, OFF ARE SYMBOLIC CONSTANT;
INIT
  PRAGMA RULE_TRIGGER IS NEW_DATA;
```

```

BEGIN -- Start of RTSUB rules
  IF control = RUN AND heater = undefined THEN heater := ON;
  IF control = undefined OR
     control = STOP THEN
    heater := OFF;
  IF temperature > hi_limit AND
     heater = ON AND
     time_now - heater'time > 3 AND
     control = RUN THEN
    heater := OFF;
  IF temperature < lo_limit AND
     heater = OFF AND
     time_now - heater'time > 3 AND
     control = RUN THEN
    heater := ON;
                                -- only change the value of heater if the heater
                                -- ON,OFF value has not been modified within the last
                                -- three seconds.
END;

```

At its simplest, an RTSUB consists of a declaration of its parameters and its rules. The first four lines in the above example declare the parameters passed when the RTSUB is called. The lines between the BEGIN and END specify the rules for the RTSUB. Statements are terminated by a semicolon and comments start with -- and run to the end of the line.

An RTSUB deals in two types of variable, attributed (ATTRIB) and non-attributed (NONATTRIB). NONATTRIB variables or records are the same as C variables or structs. ATTRIB variables or records have an additional attribute structure following the data structure in memory:

```

typedef struct
{
    long dsl_importance;    /* Importance of the data on a 1
                           * (least) to 10 (most) important
                           * scale of the data.
                           */
    af_time dsl_time;      /* In standard system seconds plus
                           * milliseconds. 'af_time' struct is
                           * long aft_secs plus short aft_ms.
                           */
} dsl_attr_type;

```

Attributed parameters are passed as pointers to record structs that have the additional attribute structure at the end. They are both inputs and outputs to the RTSUB. If an attributed variable is changed by the rules then the record struct passed in is changed, as is the dsl\_time. Non-attributed parameters can be passed by value (IN) or by pointer (OUT or INOUT). Rule execution is only triggered by changes to attributed variables. Note that IN parameters are only copied on entering the RTSUB and OUT parameters are only copied on leaving. INOUT parameters are copied both ways.

Unlike a C subroutine in which if...then...else... statements are executed sequentially, the rules in an RTSUB are triggered by changes to the value of any attributed variable. By default, all attributed variables are assumed to have changed whenever an RTSUB is called. Alternately, if the PRAGMA RULE\_TRIGGER is set to NEW\_DATA (as in the above example) then the time attribute of the variable is compared with the time\_last\_fired attribute of all rules that depend on

the variable. These rules are only triggered if their `time_last_fired` is less recent than the `time` attribute of the variable.

A rule is triggered by a change to any attributed variable in its antecedent (between the IF and the THEN). When the rule is triggered, the rule's antecedent will be evaluated. If the antecedent is true then the THEN clause will be evaluated. Otherwise, if the rule has an ELSE consequent, then this will be evaluated. If an attributed variable is changed as a result of the THEN clause or ELSE clause execution then its `dsl_time` attribute will be automatically updated. Changing the attributed variable may cause other rules to be executed.

Thus an RTSUB may have many rules, but only a few of these may be executed as a result of a change to one of its attributed parameters. The order of the rules in an RTSUB is unimportant except as far as deciding the order in which to execute rules if it is possible to execute more than one rule at any time. As there is no "linkage" code connecting one rule to the next, it is very easy to add or change rules within the RTSUB.

Rules that contain time-dependent antecedent clauses, such as those that contain reference to the current time (`time_now`), are evaluated whenever the RTSUB is called. In the above example, this would happen for the rules in which the current time is being compared to the time at which the heater variable was last changed (`heater.time`, the time attribute of the variable heater).

Non-attributed variables such as `lo_limit` and `hi_limit` are used to set parameters for the rules but do not trigger rules.

Symbolic, numeric, and temporal reasoning can be freely intermixed in DSL rules. In the above example, the attributed variables `control` and `heater` are symbolic and the variable named `temperature` is numeric. The variables are strongly data typed. This is so that the rules compiler can detect as many errors as possible.

Variables can contain undefined values. These are indicated by "magic" numbers. The rules are able to reason about undefined values. Thus the antecedent clause:

```
IF temp1 >= temp2 THEN ....
```

can only be true if both `temp1` and `temp2` are defined (not magic numbers).

Once called, RTSUBs execute until there are no more rules to fire. Then they return to their caller. RTSUBs may be called multiple times, either in a time loop or whenever an attributed parameter changes. A change in an attributed parameter is indicated by changing the time attribute field before calling the RTSUB again.

## 2. Elemental Data Types in DSL

DSL Type	Description	C data type
short/SHORT	16 bit integer	short
long/LONG	32 bit integer	long
integer/INTEGER	16 bit integer	short
byte/BYTE	8 bit unsigned integer	unsigned char
ushort/USHORT	16 bit unsigned integer	unsigned short
ulong/ULONG	32 bit unsigned integer	unsigned long
float/FLOAT	32 bit floating point number	float

double/DOUBLE	64 bit floating point number	double
boolean/BOOLEAN	16 bit integer (0 or 1)	short
char/CHAR	character	char
string/STRING	null terminated string	char *
symbolic/SYMBOLIC	symbolic type	short
goal/GOAL	symbolic goal variable	short
abstime/ABSTIME	absolute time	af_time
reltime/RELTIME	relative time in seconds	float

Note that the exact representation of some of these types is machine/compiler/operating system dependent.

DSL is case-sensitive for user-defined words (types, variables, functions, etc.), but case-insensitive for compiler keywords (such as IF, THEN, BEGIN, END, INIT, etc.). In the code examples that are in the manuals, we use upper-case words for compiler keywords and lower-case words for words defined in the example.

### 3. Formal Structure of a Rule Subroutine

Every Rule Subroutine (RTSUB) has the format:

```

declaration of parameter data types

RTSUB rtsub_name(formal parameter list) IS
DECLARE
    declaration of additional variables and data types
INIT
    initialization of variables
BEGIN
    action rules . . .
END;
```

Both the INIT and DECLARE sections are optional.

All variables used within the RTSUB must be declared as a parameter or within the DECLARE section or within a specification file (.afs, the equivalent of a .h file for DSL) included in this section. Variables can only be declared once and only in the declaration section. The declaration section cannot include any rules or assignment statements. Specification file declarations can also precede the line containing the RTSUB keyword.

The initialization section, following the keyword INIT, is used to give initial values to variables using assignment statements. It is also used to call subroutines that may initialize data shared by a number of subroutines. The initialization section for each RTSUB is executed by calling `init_rtsub_name(formal parameter list)`, where `rtsub_name` is the name given to the RTSUB in the line beginning: `RTSUB rtsub_name(parameter list)`. The `init_rtsub_name()` initialization subroutine is generated by the compiler and is called with the same formal parameter list as is specified for the RTSUB. When called, this initialization subroutine first initializes all attributed variables to undefined (using magic numbers). This includes both those passed as ATTRIB parameters and those specified in the declaration section. Then it sequentially executes the statements specified in the INIT section of the RTSUB.

Pragma statements (used to set run-time rule execution parameters) are usually executed in the INIT section. They can be executed in the action rules section but for many pragmas (such as

changing the method for determining rule execution order) the transient effects are unpredictable.

All the action rules are included between the BEGIN and END statements. Their formal structure is described in Section 5.

All data types, C procedures and C functions must be declared before they are used. For frequently used data types and subroutines, it is much more convenient to place the declarations in a file and then to include this file at the beginning of the RTSUB file with the following syntax:

```
SPECIFICATION datatype.afs
```

or:

```
SPEC datatype.afs
```

The extension .afs is used for these DSL include files.

Note that specification files can also be included before the RTSUB line, to specify types used in parameters to the RTSUB. This should be done when the same type is used in more than one RTSUB - if not, the type will be declared in the .h generated from each RTSUB, and if they are both included by the same C source file, then the C compiler will issue a duplicate declaration error. To accommodate this, specify the type in a specification file, and include this file before the RTSUB line. Then, include the .h file with the same base name as the specification file in the C file (make sure to use 'afs2h' in your makefile to automatically generate the .h file from the specification file - see Appendix C for more information on 'afs2h').

It is also possible to put common action rules for a class of RTSUBs in a separate file and include that file in the rule section of the RTSUB with the following syntax:

```
RULES init.rul;
```

The DSL compiler expects to find the rules in the file init.rul. All rule files must have a .rul extension.

Note that specification and rule statements in DSL, like include files in C, can be nested. Nesting is not, however, recommended from a configuration control viewpoint and may result in a system that is incompatible with many configuration control systems.

The file names (datatype.afs and init.rul) can include full directory paths. This is particularly useful in constructing sets of rules that the end user can modify. In this case, the application is created as a "wrapper", not seen by the user, that calls upon rule sets that are in a separate directory and can be modified by the user. The application programmer specifies the declaration and initialization section of any RTSUB to be user modified but leaves some (or all) of the action rules to be modified by the end user. These rules are placed in .rul files in a user accessible directory. The user can modify the rules using a standard text editor and then simply run MAKE to recompile and link the executable program process. In this way, limited end user modification is made simple and straightforward with minimal end user training.

In order to make programming more convenient, RT-Expert comes with a standard specification file:

```
rtexpert.afs
```

This specification file contains prototypes for useful functions and procedures that are supplied with the RT-Expert library. It also contains the prototypes for most common math functions such as sine or cosine. A detailed listing of the function prototypes is found in Appendix A. DSL does not automatically include this specification file so as to maximize compilation speed if it is not required. As such, users must include this file in SPEC statements in the declaration section of the RTSUB.

Each RTSUB must be contained in a separate file that has a .dsl extension. If the name of the file is compute.dsl then the DSL compiler will create a source file called compute.c and an include file called compute.h. Programmers should be careful not to put the RTSUB in the same file name as a corresponding C file. It is easy to destroy a file of C subroutines called test.c by creating an RTSUB in a file named test.dsl and then running RTX to compile test.dsl into test.c!

The C file created from an RTSUB named *rsub\_name* contains two callable subroutines:

- An initialization procedure named *init\_rsub\_name*
- A rule execution procedure name *rsub\_name*

In addition the DSL compiler creates many functions that are called by these subroutines. The initialization subroutine should be run during the start-up of the application. The rule execution procedure should be run whenever there is new data to be processed and at the time period required for time dependent rule re-evaluation.

## 4. Variables, Data Types, and Constants

### 4.1 Declaration of Variables

Variables can have names up to 32 characters in length and must start with an alphabetic character. Names can include any alphanumeric characters. The only "special" character allowed is \_ (underbar). Variable names cannot be the same as any of the reserved keywords shown below:

#### Reserved Keywords in DSL

ABSTIME	AND	ARE	ATTRIB	BEGIN
BOOLEAN	BREAK	BYTE	CHANGED	CHAR
COMPILER	CONSTANT	CONTINUE	CURRENT	DAY(S)
DDL	DECLARE	DEF	DEFINED	DO
DOUBLE	DYNAMIC	ELSE	EMPTY	END
FALSE	FLOAT	FOR	FUNCTION	GOAL
HAS	HAS_VALUE	HOURL(S)	IF	IMPORTANCE
IN	INIT	INOUT	INT	INTEGER
IS	KNOWN	LONG	MINUTE(S)	MSEC(S)
MOD	MONTH(S)	NAME	NEWLINE	NONATTRIB
NOT	NULL	OF	OR	OUT
PRAGMA	PRINC	PRINT	PROCEDURE	RTSUB
RECORD	RELTIME	RETURN	SECOND(S)	SHORT
STEP	STRING	SYMBOL	SYMBOLIC	THEN

TIME	TIME_NOW	TNOW	TO	TRUE
TYPE	ULONG	UNDEFINED	UNKNOWN	UNSIGNED
USHORT	WEEK(S)	WHILE	YEAR(S)	

All variables have types that must be declared, as in:

```
abc, xyz are FLOAT;
```

which declares two variables abc and xyz as floating point numbers.

The allowed elemental data types were given in Section 2. Both upper and lower case type names are allowed. These are reserved words.

The data type of variables can be declared using IS and ARE forms as in:

```
ax IS FLOAT;
t1,t2 ARE ABSTIME;
```

or they can be declared using the standard Ada format of

```
ax: FLOAT
t1,t2: ABSTIME;
```

Record data types can be declared by (for example):

```
TYPE xxx IS
BEGIN
    yy: FLOAT;
    xx: SHORT;
END;
```

A record data type can contain fields that are any of the elemental data types, and it can also contain other records.

Variables which are specific instances of a record type can be declared as in:

```
a : rec_type;
b IS rec_type;
c, d ARE rec_type;
```

The declaration of the type must appear in the declaration section of the RTSUB (or in an included SPEC file) prior to its variable declaration.

## 4.2 Constants

Numeric constants can take the following forms:

```
SHORT:          359, -23
LONG:           -378459 123456789
FLOAT:          3.5 -4.639 3.5e9 -1.0e-5
DOUBLE:         2.31e-40 123456789.9876
```

The float and double constants are expressed as decimal numbers with optional exponents that are within the range of IEEE single or double floating point formats respectively.

There are two boolean constants:

```
TRUE and FALSE
```

Numerically TRUE is 1 and FALSE is 0 after translation to C.

String constants are always enclosed in double quotes such as:

```
"This is a string"
```

Character constants are enclosed in single quotes such as:

```
'a'
```

The keyword `NULL` is a special string identifier which is used to refer to a null C pointer. Thus, a string can be compared to `NULL` as in the following:

```
s := get_param(param_name);
if s = NULL then PRINT "Warning, No Applicable Parameters";
```

Many C routines that handle strings return a null pointer under some conditions. Thus, DSL allows you to compare a string to `NULL` and to assign the value `NULL` to a string. Care should be taken using this value, because `NULL` is also the `UNDEFINED` value for a string. A rule that checks if `s = DEFINED` will not fire because `NULL` is the `UNDEFINED` value.

It is often very useful to be able to refer to the special characters such as escape, carriage return, or the bell character. Thus, DSL has predefined character constants to refer to these control characters:

'NULL'	represents the null character
'FF'	represents the form feed character
'LF'	represents the line feed character
'CR'	represents the carriage return character
'TAB'	represents the tab character
'ESC'	represents the escape character
'BELL'	represents the bell character

Note that DSL automatically translates these character constants into the correct machine or language specific representation. Also note that `'NULL'` and `NULL` are very different things - `'NULL'` is a character constant which represents the ASCII NUL character (`'\0'` in C), where `NULL` (without the single quotes) represents a `NULL` string pointer (a string variable that has not yet been allocated any storage).

DSL supports `SYMBOLIC` constants and variables. Symbolic constants can take values such as:

```
ON OFF RUN STOP
```

and are used in symbolic reasoning. Symbolic constants must begin with a letter and must not be a keyword or the name of a variable or other declared entity. A distinction is made between declaring a symbolic variable as:

```
runstate is SYMBOLIC;
```

and

```
runstate is (ON, OFF, RUN);
```

In the latter case, any attempt to set `runstate` to a value other than the specified constant values will result in a compile-time error (there is, however, no run-time checking to verify this, for the sake of efficiency). This latter statement also declares the symbolic constants `ON`, `OFF`, and `RUN`.

Unlike other constants, symbolic constants have to be declared before use. They can be declared implicitly in the form shown above or they may be declared without association with a specific symbolic variable. For example:

```
BROKEN is SYMBOLIC CONSTANT;
ON, OFF are SYMBOLIC CONSTANT;
```

The reason that symbolic constants must be declared is to allow the DSL compiler to perform validation of its input. Consider the assignment statement:

```
runstate := control;
```

In this example, suppose that `runstate` has been declared as a symbolic variable. If control were not declared as a symbolic constant then it would not be clear whether the user meant control to be a symbolic constant or a variable.

The rules execution mechanism in RT-Expert maintains a hash table of symbolic constants. The C subroutine `af_sym("name")` looks for "name" in the hash table and if it finds it, returns a unique index. If it does not find it, it inserts "name" into the hash table and returns the index. The C subroutine `af_desym(index)` returns a pointer to the name string. The advantage of this approach is that if a symbolic variable is set to a value in one RTSUB and then this value is tested in another, all the index values used for comparison are guaranteed to be the same in all the RTSUBs and in the rest of the application code provided that this mechanism is used consistently.

The use of unique hash table index values for comparing symbolic constants is much faster than comparing strings. The symbolic constants are looked up once (when the `init_rtsub_name()` initialization procedure is called) and thereafter are referenced as short integer variables containing the indices.

There are two types of time constants and variables in DSL: those concerned with relative time and those concerned with absolute time. Relative time variables are declared as in:

```
dt1, dt2 are RELTIME;
```

Absolute time variables are declared as in:

```
time_1 is ABSTIME;
```

Relative time variables contain a floating point number representing a relative time in seconds. They can be assigned to any numeric constant, in which case the constant is assumed to be in seconds. Thus:

```
dt1 := 3.25;
```

would set the relative time variable to 3.25 seconds. Note that time is maintained to a granularity of 1 millisecond and that finer granularity for constants has no practical effect.

Relative time value units can also be specified as in:

```
dt2 := 7 years + 3 months + 2 days;
```

The keywords `year(s)`, `month(s)`, `week(s)`, and `day(s)` cause the preceding integer constant to be multiplied by the appropriate number of seconds. The compiler takes the number of seconds in a non-leap year and divides it down into the appropriate number of units (365 days, 12 months).

The other available time keywords are:

```
msec[s]    _ milliseconds
second[s]
minute[s]
hour[s]
```

Note that the `[s]` to make time units plural is optional. Both singular and plural keywords are defined for all units.

Absolute Time (ABSTIME) type variables store time in an `af_time` structure as:

```
typedef struct
{
    long  aft_secs; /* Seconds part of time representation (as
                   * returned by time() function in C).
                   */
```

```

    short aft_ms;      /* Milliseconds after the seconds (in
                       * range 0-999).
                       */
} af_time;

```

The number of seconds is the standard used by the C run-time library in use. In most cases, standard Unix time is used which is seconds since 00:00 hours Monday January 1, 1970. To set the absolute time on an attributed variable, it is best to use the time manipulation subroutines listed in Appendix B. These avoid the user having to know which "standard" is being used for absolute time.

The only absolute time constant is a date in the form [mm/dd/yy]. This is converted to the number of seconds at 00:00 hours on that date since 00:00 hours on Monday January 1, 1970 (UNIX standard time). A desired number of hours, minutes, and seconds can then be added to this to form a specific time as in:

```
time_1 := [10/06/93] + 13 hours + 30 minutes; -- 1:30 PM
```

The reserved words TNOW or TIME\_NOW refer to the current absolute time.

### 4.3 Attributes

By default, Activation Framework maintains a set of attributes for every variable. These are the time a variable was last changed and its current importance. If a user wishes to maintain other attributes, such as the time an individual field of a record variable was changed, then these can be maintained by the user as part of the variable's record structure.

To minimize the space taken for variables such as counters, indices, and temporary variables, DSL allows non-attributed variables to be declared in the form:

```
counter IS NONATTRIB SHORT;
```

The keyword NONATTRIB preceding the data type tells DSL not to generate code to maintain attributes for this variable. A change to a non attributed variable does not trigger any rules, even if the variable appears in the antecedent of a top level rule. Non-attributed variables can be any simple variable or a record. The declaration of non-attributed variables is the same as for regular attributed variables except for the keyword NONATTRIB.

The only exception to this is a non-attributed parameter in the call to an RTSUB. It is specified as IN, OUT, or INOUT which implies that it is NONATTRIB. For attributed variables declared in the declaration section of an RTSUB the ATTRIB prefix for the data type is not used as ATTRIB is the default.

Because NONATTRIB variables do not have attributes and do not trigger rules, rules containing these variables in their antecedent part are treated differently. For more on rule triggering involving NONATTRIB variables, see the sections on rule execution and on pragma statements.

A change to an element of an attributed record only triggers those rules that are affected by the change to that element. DSL does not, however, maintain attributes for each element of a record to save memory space.

When a variable is referenced as in xxx, this refers to the value of that variable. When this is followed by a quote mark, the compiler expects an attribute name to follow as in xxx'time.

The DSL compiler emits code to generate the following attributes when they are required:

```
'name          A string containing the name of the variable.
'def           A boolean (true or false) specifying whether the
              variable is defined or undefined.
```

The following attributes are maintained only at the top level for each attributed variable, and not for fields of a record. When a field of a record variable is changed, for example, the time of the entire variable is updated to reflect the last time the variable was changed.

```
'time          The time any part of the variable was last changed.
```

However, it is not updated when only the variable's attributes are changed.

```
'importance    The importance of a variable.
```

The pseudo-variable *time\_now* and constants have an importance of 5, which is also the default importance of variables. If PRAGMA IMPORTANCE is set to INHERIT, then whenever a variable is assigned a new value, it is also assigned the maximum importance of any of the variables on the right hand side of the expression.

Importance is the only attribute that can be directly modified by the user (the others can only be modified indirectly).

The BOOLEAN expressions:

```
x = defined      evaluates to:    x'def
x = undefined    evaluates to:    not x'def
```

By default, all variables have an importance value of 5. The importance can be set by:

```
x'importance := 3;
```

This is usually done in a nested rule group. The importance of the data item can be used to affect the order of rule execution.

The variable TNOW or TIME\_NOW has an implicit importance of 5. Thus time dependent rules are always at least this important when data directed chaining is being used.

## 5. Formal Structure of Rules and Statements

### 5.1 Rules

A rule takes the general form:

```
IF antecedent THEN consequent ELSE alternate consequent;
```

where the ELSE clause is optional.

At the top (non-nested) level, a rule can also take the form:

```
statement;
```

which is short for:

```
IF any of the variables which affect the execution of the statement HAS CHANGED
THEN statement;
```

The consequent or alternate consequent of a rule can be a statement or a block of statements as in:

```
if antecedent then
begin
```

```

    statement1;
    statement2;
    statement3;
end;
else
begin
    statement1;
    statement2;
    statement3;
end;

```

Statements can be simple assignment statements, they can be IF...THEN...ELSE... rules, they can be subroutine calls, print statements, pragma statements, or a number of other special constructs described in the Section 5.3. Rules can be nested to any level.

Note that only top level (non-nested) rules and statements are triggered by changes to variables (or by time passing) and that all nested statements are executed sequentially.

The antecedent of a rule is constructed from a sequence of predicate clauses linked by AND or OR. Parentheses can be used to group boolean expressions and NOT can be used to negate them. For example:

```
IF a > 3 AND NOT (b < 5 AND c = RUNNING) THEN .....
```

Parentheses can be nested to an arbitrary level.

The boolean expressions, in the predicate clauses, are formed using the following operators:

```

/=          - not equal (same as the != operator in C)
=           - test for equivalence (same as the == operator in C)
< > <= >=  - less than, greater than, etc. (same as C)

```

The greater than, less than, etc. group does not apply to symbolic data types. Within a boolean expression, the variables and constants used must be of compatible data types. They can be:

- a) All numeric - fixed or floating point.
- b) All boolean
- c) All symbolic
- d) All strings or characters

or both sides of the expression must evaluate to absolute times or relative times. Absolute and relative times cannot be directly compared.

Thus:

```

tnow - x'time > 12.5    is valid
tnow > 55              is not valid

```

In performing a comparison of strings and characters, if both sides of the expression are strings then the string compare library routine is used. If both sides are characters then a character compare is used. If the left hand side is a string and the right hand side (RHS) is a character, then the RHS is converted to a string before comparison. If the LHS is a character then it is compared with the first character of the RHS string.

Fields of records can be used in comparison using the dot convention. For example:

```
IF x.field1 > 100 THEN .....
```

Here we are referring to the field named field1 in the data type declaration used to declare the record x. This field data type must be appropriate to the boolean expression in which it is used.

Note that top level rules containing field references (as above) in their predicate clauses are only triggered when that specific field of the antecedent variable changes or if the entire record changes.

Sometimes it is desired to re-evaluate a rule because time has changed. For example:

```
if TIME CHANGED and foo(bar) = TRUE then .....
```

Here the predicate expression `TIME CHANGED` causes the rule to be re-evaluated whenever the `RTSUB` is called. This can be used to cause the function `foo()` to be called every time (provided it is declared as `DYNAMIC` - see Section 7) and the action of the rule conditioned upon its return value. In such a situation, the subroutine `foo()` is typically checking on the occurrence of some asynchronous event such as the availability of data from another computer program.

Assignments are not permitted in predicate clauses, with one important exception. If a C subroutine is called in a predicate clause, and it has `OUT` parameters, these `OUT` parameters will be assumed to change value during the call. This will then cause other rules, in whose antecedents these parameters appear, to be triggered as a result.

An optimization of DSL causes predicates of rules to only be evaluated if a part of the predicate has changed or if it hasn't yet been evaluated. (Where a 'part' of the predicate is a variable in the predicate, or a parameter to a function call in the predicate, etc.) Otherwise, the previous value is used. This is so that computationally expensive function calls and mathematical calculations are only done when necessary. `NONATTRIB` variables, however, cannot trigger rules because they do not have the rule chains or attributes of regular variables. By default, predicates containing non-attributed variables are always re-evaluated. This can be changed through use of the `REEVAL_NONATTRIB` compiler pragma. (See Section 5.3.5 for an explanation of `PRAGMA`'s)

The DSL language in which `RTSUB`s are expressed can be used as a Program Design Language. As such some features have been added to the language to make predicate clauses more readable.

Instead of writing

```
IF x = 5 THEN .....
```

This can be expressed as:

```
IF x HAS_VALUE 5
```

This can sometimes be used to make rules easier to read.

The keywords `HAS`, and `HAS_VALUE` are syntactically equivalent to '='.

`HAS` is useful with symbolic variables as in:

```
if cat has mouse then .....
```

A simple variable such as a number, a symbolic, or a string can be tested for being defined by:

```
if y = defined then .....
```

and for being undefined by:

```
if y = undefined then ....
```

Where `defined` and `undefined` are keywords. They can also be tested to see whether they have changed (including to and from undefined state) by:

```
if y HAS CHANGED then ....
```

Records are more complicated because they can have some fields that are defined and some that are undefined. Hence these data elements can be undefined or partially or fully defined.

A record *x* can be tested for being defined by:

```
if x = defined then .....
```

In this case, whether *x* has to be fully or partially defined is set by the pragma `RECORD_DEF`.

The default of the pragma for records can be over-ridden by:

```
if x = fully defined then ....
if x = partially defined then .....
```

## 5.2 Arithmetic and Other Operators

Programmers can specify arithmetic and other operations to be performed in predicate and consequent clauses. The operations are, for the most part, based on those used in the Ada language. They follow normal left to right parsing rules and parentheses can be used to resolve any possible ambiguities. This section describes the various types of possible operations.

### Numeric Operators:

#### Arithmetic Operators:

<code>+ - * /</code>	- plus, minus, multiply, divide are translated to the equivalent C language operations
<code>mod</code>	- modulus operation on integer numbers
<code>**</code>	- <code>k mod j</code> produces remainder after dividing <code>j</code> by <code>k</code>
	- performs integer or floating point exponentiation
	- depending on mantissa and exponent types

#### Typical expression:

```
((b + c)/(a * b)) ** d
```

#### Boolean Operators:

<code>&lt; &lt;=</code>	- less than, less than or equal
<code>= /=</code>	- equal, not equal
<code>&gt; &gt;=</code>	- greater than, greater than or equal

#### Boolean Operators:

`BOOLEAN and BOOLEAN`

`BOOLEAN or BOOLEAN`

`not BOOLEAN`

### String/Character Operators:

#### Concatenation Operators:

<code>"CAT" + 'S'</code>	becomes	<code>"CATS"</code>
<code>"BIRD" + "DOG"</code>	becomes	<code>"BIRDDOG"</code>
<code>'S' + "CAT"</code>	becomes	<code>"SCAT"</code>
<code>'S' + 'M'</code>	becomes	<code>"SM"</code>

**Boolean Operators:**

= < <= > >= /=

- If the LHS is a string, a string compare is done. If the LHS is a character, a character compare is done. Library routines are called to make the necessary conversions.

**Symbolic Operators:**

Only operators are tests for equality

= /= - equal, not equal

**Time Operators:**

Arithmetic Operators:

ABSTIME - ABSTIME yields RELTIME

ABSTIME + RELTIME yields ABSTIME

ABSTIME - RELTIME yields ABSTIME

Any arithmetic operation involving one or more relative times and no absolute times is treated as a numeric operation that yields a RELTIME

Boolean Operators:

> < >= <= /=

- require ABSTIME on both sides of operation or a combination of RELTIME and NUMERIC on both sides.

**5.3 Statements****5.3.1 Assignment Statements**

Assignment statements appear on the Right Hand Side (RHS) of rules either in the then or else clauses. They can also appear as top level rules with no if...then... clause.

Assignment statements that are top level 'rules' are triggered for execution whenever any right hand side variable changes. Thus

```
a := b + c;
```

would be executed if b or c changed.

Statements inside a nested block are executed sequentially.

For all assignment statements, except boolean assignments, if any variable on the right hand side is undefined then the variable on the left hand side becomes undefined. In the case of a boolean assignment, the right hand side may be defined even if it contains undefined variables. For example:

```
c := a or b;
```

If a is true and b is undefined the result is still true and c is defined.

Examples of arithmetic assignment statements are:

```
x := 2 * y + 3;
y := 25.9e-3*exp(z)+c**3;
```

Essentially any Ada language arithmetic expression is permissible in DSL.

Format conversion is performed across any numeric assignment or operator, such as:

```
long      := float;
float     := short;
double   := float + short;
```

in the same manner as performed by C.

Strings can be concatenated using the + operator as in

```
a := b + c;
```

This will concatenate c to b and put the result in a. All operators must be of type string or character. The following apply when mixing string and character assignments:

```
string    := string;  -- simple assignment
char      := string;  -- put first character of string in char variable
string    := char;    -- create string containing char
```

The fields of a record variable can be accessed in a statement using a dot convention, by (for example):

```
if readings.x_val > 10.5 then readings.in_range := TRUE;
```

Here x\_val is the field name of a previously declared record type. The variable readings is declared to be of this type. If this is a top level rule it will only be triggered by changes to readings.x\_val and not other fields of the variable readings.

The importance of variables can be set in assignment statements such as:

```
status'importance := 3;
```

Note that an importance of 1 is low and 10 is high. An importance of zero can be used, but programmers should be aware that this causes special effects in such situations as data directed chaining (see Section 8).

Variables have an importance level of 5 by default. If PRAGMA IMPORTANCE is set to INHERIT, then the importance of a variable on the left hand side of an assignment statement will be inherited from the maximum importance of any variable on the right hand side of the assignment statement. This is primarily used in conjunction with data or goal directed chaining as described in Section 8.

### 5.3.2 Print Statements

DSL supports a PRINT statement in the form:

```
PRINT a, b, c;
```

This is useful for debugging and for simple user interaction. Each of the variables is printed in a default format according to its data type. The print statement can also include string constants (and other constants), as in:

```
IF name = UNDEFINED THEN
BEGIN
  PRINT "ENTER YOUR NAME : ";
  read_name(name);
  PRINT "HELLO, ", name;
END;
```

This top level rule fires if name becomes undefined. The user is prompted for their name, a procedure is called to read the name, and the user's name is echoed back. Note that DSL handles any type specific issues when printing out variables. String constants are printed out as they appear, and variable data is formatted depending on its type.

The keyword NEWLINE may be used to insert carriage returns into the print statement as follows:

```
PRINT "RED", newline, "BLUE", newline, "GREEN", newline;
```

This statement will put one word on each line, and go to the next line at the end. If you are just typing single lines of text and data, the PRINC statement will insert a carriage return at the end of the line automatically:

```
PRINC "RED", newline, "BLUE", newline, "GREEN";
```

This will behave identical to the previous example.

Print Formatting:

Variables are printed according to their type.

ABSTIMEs are printed out formatted as:

```
<month>/<day>/<year> <hour>:<min>:<sec>:<milli>
```

So 3:42:23 P.M. on February 5, 1992 would be printed as

```
02/05/1992 03:42:23:000
```

Top level PRINT statements are triggered by a change to any variable being printed. Nested PRINT statements are executed sequentially. Any variable that is undefined is printed as <undefined> by the PRINT statement.

### 5.3.3 Subroutine Statements

Procedure calls are another form of statement. For example, if we have a procedure declared as:

```
PROCEDURE foo(a:IN SHORT; b:OUT LONG);
```

Then we can write a top level rule simply as a procedure call:

```
foo(c,d);
```

Top level procedure statements are triggered by a change to any IN variable of the procedure. All OUT variables are assumed to have changed across the call and may trigger rules as a result.

Procedure statements can also be within nested blocks. In this case they are executed sequentially.

It should be noted that undefined variables can be passed to a procedure or function (see Section 6). If the subroutine is only to be called with defined values for its variables then it should be stated in a rule such as:

```
if c = defined then foo(c,d);
```

Note that if there are no parameters to the function or procedure, the parentheses are omitted as in the following:

```
if getchar = 'a' then do_a_stuff;
```

This calls a function getchar and compares its return value to 'a'. If it succeeds, it calls the procedure do\_a\_stuff.

Appendix A of this manual contains the prototypes for some useful C functions that are supplied with RT-Expert. The prototypes for these are declared in the specification file rtexpert.afs.

Appendix B of the manual contains prototypes for C subroutines that may be useful in setting up calls to an RTSUB, such as functions for setting the time attribute of a variable.

### 5.3.4 FOR and WHILE Statements

The FOR statement is used in DSL for executing a single statement or block of statements multiple times. The syntax of the FOR statement is as follows:

```
FOR <loop_var> := start TO end [STEP <integer value>] [DO]
  BEGIN
    Statements
  END;
```

The loop\_var must be declared as a non-attributed integer type (e.g. SHORT, LONG, BYTE, ...). The start and end values can be any numeric expressions and the STEP value may be any positive integer value. At the start of the loop, the loop\_var will be assigned the value start. It is then checked to be in the range from start to end. This includes start being greater than, less than, or equal to end. If the value is still in the range, the loop is executed. At the end of the loop, if there is a STEP value, the variable is modified by that value. If start is less than or equal to end, the loop\_var is incremented by the STEP value, otherwise it is decremented by that value. If no STEP value is provided, a value of 1 is assumed. Note that start and end are re-evaluated every time through the loop to validate the range. Thus, if they contain function calls that return different values, unpredictable results may occur.

The WHILE statement is yet another way of looping. It is similar to the FOR statement, however the WHILE statement loops while a condition is true. The syntax of the WHILE statement is as follows:

```
WHILE <expression> [DO]
  BEGIN
    statements
  END;
```

The expression must be of a boolean type and the statement block may be replaced by a single statement.

If a BREAK statement appears within a FOR or WHILE loop then the looping is terminated. If a CONTINUE statement is executed, then the current iteration only is terminated.

### 5.3.5 Pragma Statements

Pragma statements are mostly used to control the processing of rules. They usually appear in the INIT section of an RTSUB in the form:

```
...
INIT
  pragma order is lexical;
BEGIN...
```

Pragmas are directives provided to the RTX compiler and runtime environment by the user.

Pragmas which affect the compilation of a DSL file are specified as:

```
COMPILER PRAGMA <name> IS <value>;
```

while runtime pragmas are specified as:

```
PRAGMA <name> IS <value>;
```

Compiler pragmas can occur in the init section and rule section of an RTSUB and affect how the rule code is compiled. Currently, the only compiler pragma concerns the handling of NONATTRIB variables in the predicate section of rules. The syntax is as follows:

```
COMPILER PRAGMA REEVAL_NONATTRIB IS [ON | OFF];
```



### 5.3.6 Initialization Statements

Variable values may be initialized by statements in the INIT initialization section of an RTSUB. Examples of initialization are:

```
x := 2.95;
count := 0;
sname := "Test Case";
x := xval(2.95,0);
my_proc(2.95,z);
```

Setting these variables will cause rules to be executed as a result once the RTSUB rule execution subroutine is called by the application. Note that rules are not allowed in the initialization section of an RTSUB.

### 5.3.7 Other Statements

The keyword RETURN can be used as in:

```
IF .... THEN RETURN;
```

This causes the rule subroutine execution to be terminated.

## 6. Calling C Subroutines from DSL

DSL is a strongly data typed language. As such, all user procedures and functions must be specified in the declaration section of the RTSUB before they are used.

C procedures are declared in Ada format:

```
PROCEDURE proc_name(a,b,c:IN FLOAT; vxc:OUT vxc_type);
```

This form is used as the rules execution mechanism has to know which subroutines are input to and which are output from the procedure. All values specified as OUT or INOUT are assumed to have changed across the subroutine call.

The parameters to the procedure are specified in much the same way as for a variable declaration with one exception. The user must specify whether the variable is to be an input or output parameter. In C this corresponds to whether the variable is to be passed by value or by address. The parameter is described by a variable name, a colon, the mode of the parameter, and the type of the parameter. Multiple parameters are separated by a semicolon. If the mode of the variable is omitted, it is assumed to be an input variable and is passed by value in C. Variables which are both input and output should be specified as INOUT. This will call the subroutine with the address of the variable in C. As for variable declarations, it is possible to specify multiple parameters at once by semicolon-separating the names in the parameter list. The above example demonstrates this.

The above example would correspond to the C subroutine declaration:

```
extern void proc_name (float a,float b,float c,vxc_type *vxc);
```

Procedures do not return values. If it is desired to return a value such as in the case of sin(x) then a function definition is used:

```
function sin(x:FLOAT) return FLOAT;
```

The corresponding C function declaration would be:

```
extern float sin(float x);
```

C functions can have input and output call parameters in much the same way as for procedures.

The declaration of the procedure or function type must appear in the variable declaration section of an RTSUB or in a specification file which is discussed in Section 3.

Functions may be used in arithmetic or boolean expressions such as:

```
if sin(x) > 0.5 then .....;
```

Functions may be used in predicate or consequent expressions. Procedures may be used as a statement in the consequent part of the rule or as a statement rule.

For example:

```
if .....then
begin
  y := cos(theta)/2;
  my_proc(y,z);
end;
```

or

```
my_proc(y,z);
```

The function and procedure calls are translated into calls to corresponding C functions. All input arguments are translated into a call by value. For all output variables, the address of the variable is passed in C. Procedures are of the C type void. Functions are of the C type corresponding to the DSL type specified.

Procedure and function calls may be placed in the initialization section of an RTSUB. This is translated into a call to the corresponding C subroutine in the initialization function of the RTSUB.

An optimization of DSL causes functions in the predicate part of a rule to only be evaluated if the parameters to the function change or if the predicate has not yet been evaluated. However, many functions return different values even though the parameters do not change. A random number generator is a perfect example. This function returns a different value every time it is called. By declaring a function as DYNAMIC, the user indicates to DSL that the function is to be re-evaluated every time it is encountered in a predicate clause. Thus, the prototype for the random number generator would be declared as follows:

```
DYNAMIC FUNCTION my_rand RETURN long;
```

It is important to realize that DSL only optimizes the calls to functions in the predicate part of a rule. Functions in the statement part of rules are always called and the value assigned to the left hand side (LHS).

Some functions and procedures are time sensitive. This means that their output varies with time even if their inputs do not change. Hence rules which use these functions need to be re-evaluated periodically. To cause this re-evaluation to take place, use the TIME CHANGED construct as in:

```
IF TIME CHANGED and foo(bar) > 3 THEN ....
```

This will cause the rule to be re-evaluated whenever the RTSUB is called. If foo() is declared to be a dynamic function then it will be re-evaluated every time the rule is evaluated, not just when the variable bar changes.

Function overloading allows users to declare a function or procedure in DSL which maps to different C routines depending on the type of the parameters passed to the function.

Refer to the following example:

```

FUNCTION abs IS labs (x : LONG) RETURN LONG;
FUNCTION abs IS fabs (x : FLOAT) RETURN FLOAT;
a IS SHORT;
b IS FLOAT;

...
c := abs (a) + abs (b);

```

The above DSL code causes the C routine "labs" to be called for the variable "a" and the C routine "fabs" to be called for the variable "b". This allows DSL users to declare generic routines which call different subroutines in the underlying language depending on the parameters passed.

Programmers need to be aware of several issues relative to the parameters passed to a C subroutine. In C, IN parameters are passed by value and OUT parameters are passed by address.

Strings are represented in C as a pointer to a block of memory. In C, when passing a string to a function it is important to realize that, depending on whether the parameter is passed by value or by address, DSL will either pass the pointer, or the address of the pointer. The important thing to realize is that strings are not automatically allocated by DSL. If a user-defined function is manipulating a string, the string must be an out parameter, and the function must proceed as follows:

1. Check whether the string pointer is a null. If not then a string has been previously allocated for the variable. If the string is to be changed then free the old string by calling free() with the string pointer as an argument.
2. Allocate space for the new string (including the terminating null) by calling malloc(num\_bytes) and place the returned pointer value into the OUT address. Fill the allocated memory space with the desired text, null terminate the string, and return.

Magic numbers are used to mark a variable or a field of a record as being undefined. The undefined value for integer and floating point types is the largest possible number for that type. For example, if a variable of type SHORT has the value 32767, that means it is undefined. String variables with a NULL pointer in C are taken to be undefined. By default, records are undefined if all their elements are undefined (to change this, see the description for PRAGMA RECORD\_DEF). Boolean variables have values of 0 for false, 1 for true, and 2 for undefined. In C, the magic number for an undefined character is 0xFF, which is the old teletype RUBOUT character.

See Appendix B for user-callable routines that handle these magic numbers.

Undefined variables and fields of records are identified to the subroutine by the presence of Magic numbers. This is so that subroutines can manipulate undefined data values. If a subroutine that does not handle Magic numbers is being used then this should be handled in the rule, as in:

```
if x = defined then if sin(x) > 0.3 then .....
```

## 7. Controlling the Firing of Rules

A change in an attributed variable may cause a number of rules to fire. The DSL execution mechanism must choose which rule to execute next. It does this based upon the dynamic importance of the rule. This importance may be selected according to PRAGMA ORDER:

**LEXICAL** - (default) Earlier rules in the set of rules within an RTSUB are more important than later rules. This is the simplest to understand but may not always give the desired results.

**RECENCY** - (depth-first) The rules will fire in a chain. That is, if there are 4 rules to execute as a result of a parameter changing, DSL will pick the top one and execute it. If this results in 2 more rules being triggered, then DSL will pick the topmost of these and execute it. This chain will proceed until no more rules are triggered, then DSL will go back to the last one it left undone and try that one and so forth until no more rules can be executed, when the RTSUB returns to the calling program.

**DATA** - (data driven) The rules inherit their importance from the variables in their antecedent. A rule takes the importance of the most important variable in its antecedent. The effect is to allow important data to rapidly "ripple" through the rules while other data is still being processed.

Goal directed chaining can be used in conjunction with data directed chaining.

In a conventional backward chaining mechanism, goals can be established such as to determine the value of a specified variable. This then forces the rule execution order to be such as to lead to this goal. For example, specifying the goal of determining a value for a variable could lead to the execution of a rule which causes the user to be asked for the value of this variable. The most common use for backward chaining is, in fact, to force the application to ask questions in a reasonable order that a person would comprehend.

In real-time systems it is not usually possible to halt execution of the system until a user provides a value for a variable. Instead, the rule set may call upon a subroutine that handles the input and then proceeds to process other data and rules concurrently. Also, a problem arises in conventional backward chaining in real-time systems. The burden of maintaining goal dependencies for every rule imposes considerable overhead on the system. This could prove to be a major handicap in time-critical situations.

DSL avoids this problem by requiring the user to explicitly state how the rule triggering and backward chaining should be handled. This avoids any unnecessary overhead in goal dependencies and gives the user a great deal of flexibility in how the goal is to be managed.

DSL supports backward chaining through the goal variable. For example, the goal variable

```
y_goal is GOAL;
```

might specify a goal to determine the value of y. The convention is that the goal can either be defined or undefined. In addition, the importance of the goal can be set using the importance attribute.

An example of how the goal structure is used follows:

```
IF y_goal = DEFINED AND x > 5 then y = sin (x);
IF y = DEFINED THEN y_goal := UNDEFINED;
IF y = UNDEFINED then
BEGIN
    y_goal := DEFINED;
    y_goal'importance := 50;
END;
```

The primary use for goal variables is that they can be set by any rule and then used to invoke a complex chain of common actions in response.

A change to any variable causes the triggering of all rules in whose antecedent the variable appears. This is true no matter the type of statement or how deeply nested the statement causing the change. The only exceptions to this are:

1. Does not apply to NONATTRIB variables.
2. A variable set in the consequent of a rule and which also appears in the antecedent of that rule does not cause retriggering of that rule.
3. A change to a field of a record only triggers other rules containing that field in their antecedent or those rules referring to the whole record in their antecedent or those rules referring to a subfield of that record element. (This may occur with nested records.)

To illustrate exception 2, consider the rule

```
if control = OFF and pressure < 350 then control := ON;
```

When a new value of pressure is received, this rule turns the control ON if the control was OFF and the pressure is too low. Because the variable 'control' appears both in the antecedent and the consequent, it might be presumed that this rule would be immediately re-triggered by the change in the variable 'control'. This re-triggering of a rule due to a change to a variable in its own consequent is inhibited, however, to prevent cyclical or inefficient behavior.

To illustrate exception 3, consider the following:

```
if a.field1 = 3 then .....
PRINT a;
if a.field2 = "set" then ....
```

A change to a.field1 would cause the first two rules above to be triggered but not the third.

The antecedent of a rule contains one or more boolean clauses to be evaluated. These may consist of comparisons between variables or function calls. As an optimization, DSL will only evaluate a function call in the antecedent of a rule if the parameters of the function change or it is the first time the function has been called. Otherwise, DSL will return the previously computed value for the function.

An example of this follows:

```
if keycheck = TRUE then
    key := getchar;
```

This rule will call the function keycheck to see if a key has been pressed. Because no parameters change, keycheck will only be called the first time, and will return that value every other time.

In order to force DSL to evaluate the function every time, the function must be declared DYNAMIC. Thus, in the declaration section, the function must be declared as follows:

```
DYNAMIC FUNCTION keycheck return BOOLEAN;
```

This will force DSL to evaluate the function every time the rule is processed.

Rules in DSL are data-triggered. Therefore, if a rule has no new data, then it will not be triggered. Unfortunately, this causes many problems for new users of DSL who are unused to this data-driven architecture. Refer to the following example:

```
RTSUB x IS DECLARE
    ...
INIT
    ...
```

```

BEGIN
  PRINC "IN MAIN FUNCTION";
  IF 1 = 1 THEN
    PRINC "IN MAIN FUNCTION";
END;

```

Although you may expect to see the message "IN MAIN FUNCTION" printed twice on the screen, actually nothing will be printed at all. Because rules in DSL are data-driven, these rules will not fire because no attributed variable data is involved. There are several solutions to this problem as shown in the next example:

```

RTSUB x IS
DECLARE
  temp : SHORT;
  PROCEDURE print_statement (x : IN SHORT);
  ...
INIT
  temp := 1;
  ...
BEGIN
  print_statement (temp);
  PRINC "IN MAIN FUNCTION", temp;
  IF temp = 1 THEN
    PRINC "IN MAIN FUNCTION";
END;

```

This example will print "IN MAIN FUNCTION" three times initially, and then whenever temp changes. This is because the printing is now dependent on the dummy variable temp. Although this seems a bit awkward, it is actually an optimization on the part of DSL to only trigger and fire those rules dependent on data that has changed.

Rules can be nested as in:

```

if state = A then
begin
  if x > 300 then y := true;
  if z < 10 then w := false;
end;

```

A top level rule is only executed if there is a change to a variable in its antecedent. The nested rules are treated much like if...then...else statements in a procedural language.

Note that the execution of nested rules is different from top-level rules. If the same condition is true without any data changing in a nested rule, it will be fired every time, while a top-level rule would not be.

The top level rule ordering is set by the pragma in effect at the time. Nested rules are executed in sequential order.

## 8. Debugging, Validation, and Testing

Tracing the order of rule execution and the setting of variables within an RTSUB is done by setting the DEBUG\_RULE and DEBUG\_VARIABLE pragmas to appropriate values.

Both debugging pragmas have a simple and detailed level of tracing. In the case of rules, the SIMPLE setting causes a printout only whenever a rule is fired or a statement executed. The DETAILED setting causes a printout whenever an antecedent clause is evaluated. In the case of

variables, the `SIMPLE` setting only causes a printout whenever the value of a variable changes. In the `DETAILED` case, `printout` gives information about the attributes of the variable.

Tracing rule execution is very useful in the early stages of debugging and for finding complex problems. Many other times it is easier to insert `PRINT` statements in the rules to study the effects of rule execution. As the `PRINT` statements are automatically formatted they are easy to use. For example:

```
PRINT "In state C", x,z,w;
```

will printout the values of `x`, `z`, and `w` even though one is a floating point number, one is a string, and the other is a symbolic variable. The format of the printout is reasonable and the `PRINT` statement indicates if any variable or record field is undefined.

C subroutines called from the rules can be debugged using a standard debugger.

It is not recommended to attempt to debug the C code generated by DSL. This code, while rationally organized, does not bear a 1:1 correlation with the original source code and is difficult for a person to follow. At BellHawk Systems, we debug the rules mechanism by stepping through this generated C code. However, when we have to debug our own rules, we use the debugging pragmas and print statements - we've found that it is much easier to follow in terms of rules that way.

The Decision Support Language is intended as a Program Design Language (PDL) that can be read, understood, and reviewed by domain experts who are not programmers. This review is an important validation step and is aided by incorporating many comments within the code.

The comments are not transferred into the generated C code due to the lack of 1:1 correspondence between the DSL source and output C code. It is not recommended that users attempt to comment the generated code as this is changed whenever the rules are recompiled. (It is like trying to comment assembler code generated from a C program).

## Appendix A - Functions and Procedures Declared in `rtexpert.afs`

If you wish to use these functions in your C programs, you can run `'afs2h'` on `'rtexpert.afs'` to generate `'rtexpert.h'`, which will contain the equivalent C prototypes for these functions.

Function to return the `SYMBOLIC` equivalent of a character string

```
FUNCTION af_sym (str: STRING) RETURN SYMBOL;
```

Function to return the character string associated with a symbol

```
FUNCTION af_desym (sym: SYMBOL) RETURN STRING;
```

Procedure to set the time variable passed to it to zero

```
PROCEDURE aft_nul (tim: OUT ABSTIME);
```

Procedure to convert calendar time to relative time

```
PROCEDURE time_ctor (t_hours, t_mins, t_secs, t_millisecs: SHORT;
                    tim: OUT RELTIME);
```

**Procedure to convert calendar time to absolute time**

```
PROCEDURE time_ctot (t_year, t_month, t_day, t_hours, t_mins, t_secs,
                   t_millisecs: SHORT; tim: OUT ABSTIME);
```

**Procedure to get the current time**

```
PROCEDURE time_get (t_year, t_month, t_day, t_hours, t_mins, t_secs,
                   t_millisecs: OUT SHORT);
```

**Procedure to convert relative time to calendar time**

```
PROCEDURE time_rtoc (tim: RELTIME;
                   t_hours, t_mins, t_secs, t_millisecs: OUT SHORT);
```

**Procedure to convert absolute time to calendar time**

```
PROCEDURE time_ttoc (tim: ABSTIME; t_year, t_month, t_day, t_hours,
                   t_mins, t_secs, t_millisecs: OUT SHORT);
```

**The following are prototypes for commonly used math routines**

```
FUNCTION abs IS labs(n: LONG) return LONG;
FUNCTION abs IS fabs(x: DOUBLE) return DOUBLE;
DYNAMIC FUNCTION random IS af_random(n: LONG) return LONG;
PROCEDURE srandom IS af_srandom (n: LONG);
FUNCTION acos(x: DOUBLE) return DOUBLE;
FUNCTION cos(x: DOUBLE) return DOUBLE;
FUNCTION cosh(x: DOUBLE) return DOUBLE;
FUNCTION asin(x: DOUBLE) return DOUBLE;
FUNCTION sin(x: DOUBLE) return DOUBLE;
FUNCTION sinh(x: DOUBLE) return DOUBLE;
FUNCTION atan(x: DOUBLE) return DOUBLE;
FUNCTION tan(x: DOUBLE) return DOUBLE;
FUNCTION tanh(x: DOUBLE) return DOUBLE;
FUNCTION atan2(x,y: DOUBLE) return DOUBLE;
FUNCTION ceil(x: DOUBLE) return DOUBLE;
FUNCTION floor(x: DOUBLE) return DOUBLE;
FUNCTION fmod(x,y: DOUBLE) return DOUBLE;
FUNCTION log(x: DOUBLE) return DOUBLE;
FUNCTION log10(x: DOUBLE) return DOUBLE;
FUNCTION sqrt(x: DOUBLE) return DOUBLE;
```

**Appendix B - DSL Routines For User Subroutines**

The file `dsl_usr.h` contains all of the types and macros used by the DSL runtime library. It also contains many useful routines and macros for the user.

***Routines for Handling Undefined Values in DSL Subroutines***

The following macros are used by DSL to check if a variable is `DEFINED` based on its type. User routines may call these macros to check for `UNDEFINED` variables.

<b>DSL Type</b>	<b>Macro</b>	<b>Parameter</b>
GOAL	IS_DEF_GOAL	(GOAL g)
SHORT	IS_DEF_SHORT	(short s)
LONG	IS_DEF_LONG	(long l)
BYTE	IS_DEF_BYTE	(unsigned char b)
USHORT	IS_DEF_USHORT	(unsigned short u)
ULONG	IS_DEF_ULONG	(unsigned long U)
FLOAT	IS_DEF_FLOAT	(float f)
DOUBLE	IS_DEF_DOUBLE	(double d)
RELTIME	IS_DEF_REL	(float r)
BOOLEAN	IS_DEF_BOOL	(BOOL b)
CHAR	IS_DEF_CHAR	(char c)
STRING	IS_DEF_STRING	(char *s)
SYMBOL	IS_DEF_SYM	(SYMBOL y)
ABSTIME	IS_DEF_ABS	(af_time a)

Note that no routines exist to check if a record is defined. Because these are complex structures, such comparisons are best left to the rules language. To check the defined status of one of these structures, simply create an RTSUB that takes the structure as an ATTRIB or IN parameter and another ATTRIB or OUT parameter of type BOOLEAN. Then have a rule as follows:

```
return_value := (var = DEFINED);
```

This rule will set the value of return\_value appropriately.

The following macros are used to undefine variables by assigning them to the appropriate UNDEFINED values. Note that because these are macros, the user should not pass a pointer to the routine.

<b>DSL Type</b>	<b>Macro</b>	<b>Parameter</b>
GOAL	UNDEF_GOAL	(GOAL g)
SHORT	UNDEF_SHORT	(short s)
LONG	UNDEF_LONG	(long l)
BYTE	UNDEF_BYTE	(unsigned char b)
USHORT	UNDEF_USHORT	(unsigned short u)
ULONG	UNDEF_ULONG	(unsigned long U)
FLOAT	UNDEF_FLOAT	(float f)
DOUBLE	UNDEF_DOUBLE	(double d)
BOOLEAN	UNDEF_BOOL	(BOOL b)
CHAR	UNDEF_CHAR	(char c)
STRING	UNDEF_STRING	(char *s)
SYMBOL	UNDEF_SYM	(SYMBOL y)
ABSTIME	UNDEF_ABS	(af_time a)
RELTIME	UNDEF_REL	(float r)

Note that no routines exist to undefine a record. This is because these are complex structures that are best left to DSL to manipulate. To undefine one of these structures, simple create an RTSUB that takes the variable as an ATTRIB or INOUT parameter and set it to undefined in a rule as in :

```
IF var = DEFINED THEN var := UNDEFINED;
```

This rule will only fire if the variable is defined.

## Appendix C - Utilities for Programming with RT-Expert

### *The deldslcs.com Utility*

Deldslcs.com is a utility program shipped with RT-Expert. Its purpose is to facilitate cleaning up a development directory. It deletes the .c and .h files associated with every .dsl file in the current directory. It will also delete every .h file with the same base name as every .afs file. It prints out a one line message for each file it deletes. Here is some sample output from the program:

```
Copyright (c) 1996 All Rights Reserved
BellHawk Systems Corporation
Deleting GET_CTL.C because GET_CTL.DSL exists.
Deleting GET_CTL.H because GET_CTL.DSL exists.
Deleting GET_TEMP.C because GET_TEMP.DSL exists.
Deleting GET_TEMP.H because GET_TEMP.DSL exists.
Deleting SET_HEAT.C because SET_HEAT.DSL exists.
Deleting SET_HEAT.H because SET_HEAT.DSL exists.
Deleting TEMP_CTL.C because TEMP_CTL.DSL exists.
Deleting TEMP_CTL.H because TEMP_CTL.DSL exists.
```

This program is handy to use for a 'clean' target in a makefile, for instance:

```
clean:
    del *.exe
    del *.obj
    del *.lst
    deldslcs
```

This will leave only the source files in the current directory after you type 'make clean' (or 'nmake clean' for Microsoft C).

### *The AFS2H File Conversion Utility*

This utility converts DSL Specification files into C include files. This is particularly useful for sharing type declarations between DSL modules and C modules. Because the types are only declared in one place, they stay consistent between DSL and C modules.

Specification files can contain many things that do not belong in C include files. These statements are ignored by afs2h. The understood declaration types and the corresponding emitted code are as follows:

- SPECIFICATION declarations translate into #include directives to include the corresponding generated include file.
- For example, SPECIFICATION standard.afs becomes #include "standard.h"
- Subroutine prototypes are translated into the corresponding C language prototypes.

TYPE declarations cause a number of things to be generated. First, the corresponding C type declaration is emitted. Because variables in DSL have attributes, an attributed type declaration is

also emitted. For a more complete discussion of DSL type declarations see Section 2 of the Programmer's Manual.

Consider the following DSL type declaration :

```
TYPE list_type IS
BEGIN
    a, b ARE LONG;
END;
```

afs2h will generate two C structure declarations as follows:

```
typedef struct
{
    long a;
    long b;
} list_type;
typedef struct
{
    list_type dsl_data;
    dsl_attr_type dsl_attr;
} dsl_list_type;
```

## Index

### A

Absolute \_\_\_\_\_ 11, 14  
 ABSTIME \_\_\_\_\_ 6, 8, 9, 11, 17, 19, 29, 30, 31  
 Ada \_\_\_\_\_ 9, 16, 18, 22  
 AFS2H \_\_\_\_\_ 2, 31  
 arithmetic expression \_\_\_\_\_ 18

### B

boolean 6, 8, 9, 13, 14, 15, 16, 17, 20, 23, 26, 27, 30, 31

### C

C 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,  
 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30,  
 31, 32  
 C code \_\_\_\_\_ 3, 28  
 C subroutines \_\_\_\_\_ 8, 20, 28  
 character \_\_\_\_\_ 6, 8, 10, 14, 17, 18, 24, 29  
 CONTINUE \_\_\_\_\_ 8, 20

### D

debugging \_\_\_\_\_ 18, 21, 28  
 Decision Support Language \_\_\_\_\_ 3, 28  
 declaration \_\_\_\_\_ 4, 6, 7, 8, 9, 12, 15, 22, 23, 27, 31, 32  
 deldslcs.com \_\_\_\_\_ 31  
 DETAILED \_\_\_\_\_ 21, 28  
 DSL 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 15, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 30, 31, 32  
 DYNAMIC \_\_\_\_\_ 8, 15, 23, 27, 29

### F

FOR \_\_\_\_\_ 1, 8, 20  
 FULLY \_\_\_\_\_ 21  
 function 6, 7, 8, 11, 15, 19, 20, 21, 22, 23, 24, 26, 27, 28

### G

Goal directed chaining \_\_\_\_\_ 25

### H

HAS \_\_\_\_\_ 8, 13, 15  
 HAS\_VALUE \_\_\_\_\_ 8, 15

### I

if...then...else \_\_\_\_\_ 3, 4, 27  
 importance \_\_\_\_\_ 4, 8, 12, 13, 18, 21, 22, 25, 26  
 IN parameters \_\_\_\_\_ 4, 21, 24  
 INHERIT \_\_\_\_\_ 13, 18, 21

### L

LEXICAL \_\_\_\_\_ 21, 22, 25

### N

nested rule \_\_\_\_\_ 13, 27, 28  
 NEWLINE \_\_\_\_\_ 8, 19  
 NONATTRIB \_\_\_\_\_ 4, 8, 12, 15, 21, 26  
 Numeric Operators \_\_\_\_\_ 16

### O

operator \_\_\_\_\_ 14, 17, 18  
 operators \_\_\_\_\_ 14, 17, 18  
 OUT parameters \_\_\_\_\_ 4, 15, 24

### P

parameter 4, 5, 6, 7, 12, 15, 19, 21, 22, 23, 24, 25, 26, 27, 30, 31  
 parameters \_\_\_\_\_ 4, 6, 15, 22, 23, 24  
 PARTIALLY \_\_\_\_\_ 21  
 pointer \_\_\_\_\_ 4, 10, 11, 24, 30  
 pragma \_\_\_\_\_ 7, 12, 14, 15, 16, 20, 21, 22, 28  
 PRAGMA ORDER \_\_\_\_\_ 25  
 predicate \_\_\_\_\_ 14, 15, 16, 21, 23  
 PRINC \_\_\_\_\_ 8, 19, 27  
 PRINT \_\_\_\_\_ 8, 10, 18, 19, 26, 28  
 Procedures \_\_\_\_\_ 2, 23, 28  
 Program Design Language \_\_\_\_\_ 15, 28

### R

RECURSIVE \_\_\_\_\_ 21, 22, 25  
 record type \_\_\_\_\_ 9, 18  
 relative \_\_\_\_\_ 6, 11, 14, 17, 24, 29  
 RELTIME \_\_\_\_\_ 6, 9, 11, 17, 29, 30, 31  
 RETURN \_\_\_\_\_ 9, 22, 23, 24, 29  
 RT-Expert \_\_\_\_\_ 1, 2, 3, 7, 8, 11, 20, 31  
 rtexpert.afs \_\_\_\_\_ 2, 8, 20, 28  
 RTSUB 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 20, 21, 22, 23, 24, 25, 27, 28, 30, 31  
 rule 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,  
 Rules 1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24,

### S

SHORT \_\_\_\_\_ 5, 6, 9, 12, 19, 20, 24, 27, 29, 30, 31  
 SIMPLE \_\_\_\_\_ 21, 28  
 statement 4, 6, 7, 8, 10, 12, 13, 14, 17, 18, 19, 20, 22, 23, 26, 27, 28, 32  
 Statements \_\_\_\_\_ 1, 2, 4, 13, 14, 17, 18, 19, 20, 22  
 STEP value \_\_\_\_\_ 20  
 Subroutines \_\_\_\_\_ 2, 22, 30  
 symbol \_\_\_\_\_ 5, 6, 10, 11, 14, 15, 28, 29  
 syntax \_\_\_\_\_ 7, 20, 21

### T

type string \_\_\_\_\_ 18

V

W

Validation \_\_\_\_\_ 2, 28

WHILE \_\_\_\_\_ 1, 9, 20

variable 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32