

Tutorial

RT-Expert™

Copyright© 1996-2005

BellHawk Systems Corporation

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

Table of Contents

WHY USE “EXPERT SYSTEMS” RULES? _____ 2

WRITING IN DECLARATIVE RULES _____ 3

WRITING RULE SUBROUTINES IN DECISION SUPPORT LANGUAGE _____ 6

Why Use "Expert Systems" Rules?

Real-time business decisions are becoming increasingly complex in many applications such as customer service, repair and parts depots, insurance claims processing, billing, call-centers and help desks. These decisions have to be made by people or by automated self-service websites. The traditional approach has been to embed these rules as program code that has to be maintained by programmers. The problem with this is that the rules have to be continuously changed as sales and marketing campaigns, product and service offerings change.



For large organizations, it is economically acceptable to have a staff of programmers whose sole responsibility it is to change the code to reflect the changes in required business rules. For smaller and mid-sized organizations, this is not economically feasible. The alternative is to have the people who are knowledgeable about the needed business rules maintain the rules in English-like "IF...THEN...ELSE" rules. Then an IT staff member can integrate these rules through an automated procedure into the appropriate business application after validating that the rules function correctly.

By separating the rules specification and maintenance process from the programming process, RT-Expert enables changes to business rules to be made quickly and efficiently by the people who are knowledgeable about the rules. With an appropriately structured business application, the rules can be run in a separate process so that they can be dynamically changed without bringing down the multi-user application or Webserver.

RT-Expert is a tool for embedding expert systems rules into business applications. It takes expert systems rules written in an English-like language called Decision Support Language and converts these into procedures written in C/C++ that can be integrated within business applications, either by directly calling these subroutines, by linking with a DLL, or by executing a process that can be called from a database trigger and operates on the data written into the database.

The "IF...THEN...ELSE" expert-systems rules can reason about numeric values, symbolic values, and the times at which events occur. They are ideal for building systems to give advice to people who are called upon to make complex decisions in real-time. Because they are written in declarative "IF...THEN...ELSE" rules, the rules-sets are much easier to maintain than conventional code.

RT-Expert has the capability whereby the rule-sets can be abstracted into separate text files that can be maintained by end users. In this methodology, all the software linkage and variable definition is performed by programmers but the rules themselves can be defined and maintained by end-users as their business processes change.

Maintenance of the rules by business area "domain" specialists is facilitated by the powerful Macro pre-processor for RT-Expert. This facilitates the development of application-specific

languages that are easy for end–users to maintain. RT–Expert enables application–specific terms and variables to be defined that enable the people maintaining the rules to work within a familiar framework.

The rules can be maintained using any text editor or word processor, such as Microsoft Word, so that no special training is required except to master the domain–specific syntax established for the specific set of rules being maintained. This can reduce the training for rules maintainers to an hour or two of hands–on training at most.

With RT–Expert, multiple sets of rules can be embedded in a given application. This enables different people, with different domain expertise, to maintain the rules sets. Thus one person may set the rules about what options are available for a set of products and what items are pre–requisites. Another may set the rules for option pricing, including volume discounts.

RT–Expert can handle many hundreds or thousands of rules. These are executed rapidly as they are converted into executable binary code instead of being interpreted, as in many expert systems.

RT–Expert automatically handles the interaction between rules in a rules set. Thus a variable set in one rule may cause other rules to trigger without the need to organize the rules into any specific sequential order. This makes it easy to manage sets of rules as each rule can be changed as an independent entity without worrying about its effects on other rules.

RT–Expert can integrate the rules into a recursive test setup that runs the rules against a set of test data (which may be prior operational data). This is so that new rule sets can be run against existing test data to make sure that they work correctly before being deployed.

Writing in Declarative Rules

Programming in a declarative rules language such as DSL is very different from programming in a sequential language such as C/C++. It is much more object-oriented. Each rule is a semi-autonomous object that is triggered by changes to its antecedent data and changes data as a result of execution of statements in its consequent. The order of rule execution is driven by the flow of data and not by the sequence of program statements. This section provides an introduction to programming in DSL rules.

DSL rules act on variables and set variables as a result of their actions. For example:

```
if temperature > 70 then heater := ON else heater := OFF
```

In this example, *temperature* and *heater* are DSL variables. Unlike variables in languages such as C/C++ which only have values, variables in DSL also have a set of associated attributes.

These attributes are:

- time - Time when the value last changed.
- importance - Importance of the data/information contained by the variable.

The DSL rules execution mechanism keeps track of whether the variable is currently defined (valid) or not. It also automatically keeps track of the time at which the variable was last changed.

In DSL, when we simply use a variable name, such as *temperature* or *heater*, we are referring to its value. All other attributes are referenced in the form:

```
variable_name'attribute_name
```

For example, *temperature'time* is the time at which the value of the variable *temperature* was last changed.

Variables can be numeric (floating point or integer) or symbolic. An example of a symbolic variable is a variable *state* which takes symbolic values ON and OFF. Variables can also be strings, characters, boolean variables, and records. The professional edition of RT-Expert also supports Arrays and Lists of variables.

DSL rules can be used in real-time systems in which the data on which the production rules act changes over time. Initially, values for all the variables may not be available. Also, values for the data variables may be available at random times.

The basic syntax of a rule is:

```
if...then...else...;
```

A simple rule might be

```
if t > 100 then a := 1;
```

The left hand side of the rule (between the if and then) is called the antecedent and can consist of one or more predicate (conditional) clauses. Rules always end with a semi-colon. The else alternate action clause is optional.

The symbol `:=` denotes an assignment. Comments begin with `—` and continue until the end of the line.

Rule based languages are different from conventional programming languages in that rules are only executed when new values for variables in their antecedents are available.

For example consider a subroutine with the following rules:

```
1.      if p < 150 then b := 2;
2.      if x > 100 then a := 1;
3.      if (a = 1) and (b > 0) then alarm := 1;
4.      if v > 12.5 then b := 1;
```

Let us assume that all the variables are parameters to the subroutine. That is, that their values and attributes are passed between the calling C/C++ code and this rule subroutine.

Initially all the variables will be undefined. If *x* is set to 150 and the subroutine is called then rule 2 would execute and *a* would be set to 1. As *a* is in the antecedent of rule 3, an attempt would be made to execute rule 3, but this would fail because *b* is undefined at this point in time. If the above four rules were the only rules in the rule subroutine, then execution would cease and the subroutine would return. If *v* were set to 15 and the rules subroutine called again then rule 4 would be executed. This would be followed by the execution of rule 3 before the subroutine returned again.

Note how different this is from sequential languages such as C/C++. There may be dozens of rules in a module but only a few may be executed in response to any stimulus, which is very efficient. Also (in general) rules do not have to be put in a specific order to ensure correct execution, so maintenance of the rule base is easier.

How do we know when data in a variable has been changed? This is done by comparing the time at which a rule was last fired with the times at which the variables in its antecedent were last changed. If an antecedent variable has changed after the rule was last fired, then the antecedent is re-evaluated. The rules mechanism automatically updates the time attribute of a variable whenever its value is changed.

By default, a rule subroutine assumes that all its parameters have changed every time it is called. This can sometimes result in unnecessary rule firing as some rules will be conditioned upon ATTRIB parameters that have not changed. A rule subroutine does not have to assume that all its parameters have changed every time it is called. A run-time option called a PRAGMA can be invoked so that rules are only triggered if the parameters in their antecedents change. In this case, if a C/C++ application changes the value of a variable that is a parameter before calling a rule subroutine, the application also has to set the time attribute for the variable. This is so that the rules mechanism knows which rules to fire as a result of the change (by comparing the time for the last firing of a rule with the time attributes of its antecedent variables). The benefit of this approach is that the rules subroutine can be executed repeatedly (such as in a loop) and only those rules that are affected by changes to specific parameters are executed.

To most C/C++ programmers, this event driven nature of rules is the most difficult aspect to master. A rule subroutine may be conditioned upon a number of events that are passed as parameter variables. The rule subroutine may only act on those parameters that were changed since the rule subroutine was last called. Also the internal variables within the rule subroutine retain their values between calls, thus giving the rule subroutine memory of its prior state. This allows state transition dependent systems (such as user/GUI interaction) to be easily coded but requires the programmer to think in terms of data flow rather than control flow.

Some people, especially those trained in disciplines such as process control, find it easier to think in terms of the decision or inference graph formed by the rules. This is like a state transition diagram with the variables containing the states and the rules forming conditional arcs between the variable states.

If multiple rules could be executed as a result of a change to a variable then rule importance is used to decide which rule to execute first. When a variable is changed the importance of all affected rules is computed. The most important rule's antecedent is evaluated. If the antecedent is true then the rule's consequent is evaluated. If the antecedent is false then the alternate (else) consequent is executed if one is present, otherwise the rule's importance is set to zero until it is triggered again. Then the most important rule is selected again and the cycle repeated.

Duplicate predicate clauses are only evaluated once. Also predicate clauses are normally (there are a few special case exceptions) evaluated only when a variable they contain changes. Thus evaluation time for rule antecedents is minimized.

By default, the rule importance is assigned according to "lexical" order. That is the rule execution importance is the same as the order in which the rules are written, with the highest importance being given to the first rule and the lowest to the last. After firing, the importance of a rule is set to zero and other rules may have their importances set as a result of consequent evaluation. The most important rule is then chosen and execution is continued in this manner until there are no more rules with a non-zero importance. At this point the rule subroutine returns to the calling program.

Through the use of a run-time pragma, rule importances can be assigned so that the most recently triggered rule is fired next. This is called recency ordering and is mostly used for building automated planning systems. Sometimes it is necessary to ensure that the rules handling the most important data fire first. In this case “data” driven order pragma is used. In this case, the rules inherit their importance from the most important variable in their antecedent. This can be used in conjunction with goal directed chaining, in which the order of rule firing is determined by goal variables in the antecedents of rules.

Writing Rule Subroutines in Decision Support Language

We will now use an example program to explain the syntax of subroutines written in DSL rules and how these are integrated into an application. Our example concerns a temperature controller that controls the temperature of a chamber. The temperature controller has as inputs a RUN,STOP control and the temperature of the chamber. Its output is the ON,OFF state of the heater that controls the temperature of the chamber.

Let us first consider the rules for the temperature controller itself which are shown below. In our example, these are contained in the file temp_ctl.dsl. Note that rule subroutines, called RTSUBs, are named. The file containing the RTSUB is given the same name as the RTSUB and must have a .dsl extension. There can only be one RTSUB in each file.

The C/C++ file created from the .dsl file has the same name as the .dsl file with a .c extension.

In the file temp_ctl.dsl:

```

1  RTSUB temp_ctl(temperature: ATTRIB FLOAT;
2      contrl:ATTRIB SYMBOLIC;
3      heater: ATTRIB SYMBOLIC;
4      lo_limit,hi_limit: IN FLOAT) IS
5  DECLARE
6      RUN, STOP, ON, OFF ARE SYMBOLIC CONSTANT;
7  INIT
8      PRAGMA RULE_TRIGGER IS NEW_DATA;
9  BEGIN - Start of RTSUB rules
10     IF control =RUN AND heater =undefined THEN heater :=ON;
11     IF control =undefined OR control =STOP THEN heater :=OFF;
12     IF temperature > hi_limit AND heater = ON AND
13         time_now - heater'time > 3 seconds AND
14         control = RUN THEN heater := OFF;
15     IF temperature < lo_limit AND heater = OFF AND
16         time_now - heater'time > 3 seconds AND
17         control = RUN THEN heater := ON;
18 END;
```

An explanation of the lines follows:

Line 1: An RTSUB begins with the keyword RTSUB followed by the name of the RTSUB, in this case temp_ctl. This is followed by the parameters for the rule subroutine in the order that they will be called from C/C++ enclosed in parentheses.

The name of each parameter is followed by a colon, the parameter type, and its data type. An ATTRIB data type says that the parameter is a pointer to a record that contains the time the data was last changed and its importance. ATTRIB data types are the only ones that can trigger rules when their data is changed. In the case of the *temperature* parameter the data type is FLOAT which is a 32 bit floating point

number. The data types can be also be records.

In this example the *temperature* attributed variable is used to provide the current temperature of the chamber to the rules.

Line 2: The control variable is SYMBOLIC. In this example it can take the values RUN or STOP. SYMBOLIC variables are 16 bit integers that contain numbers used to represent symbolic variables. Instead of enumerating the values at compile time, RT-Expert converts a string such as “RUN” to a symbolic variable by calling `af_sym(“RUN”)`. This function places the string in a hash table and returns a unique short integer that is the position of the string in the table. If another subroutine also calls `af_sym()` with the same string, this string is looked up in the hash table and the same value is returned. This avoids problems with separately compiled subroutines using different values for symbolic constants. Usually the symbolic constant strings are converted to short integers once, at initialization time. Thereafter comparison of symbolic entities is done using integer comparison which is much faster than the alternative of comparing text strings.

In this example control starts and stops the temperature controller

Line 3: In this example, heater can take the values of ON and OFF. It is an output from the rules as it is set based upon the temperature and control inputs. It is also an input, as the rules are dependent on the prior setting of heater and the time at which it was last changed. It is important to note that RTSUBs are intended to be called repeatedly. Unlike most C/C++ subroutines, RTSUBs remember their prior state and rule execution is driven by changes to parameters.

Line 4: Not all parameters are intended to trigger rules. Some are simply used to introduce constants in the rules. Those passed by value are simply referred to as IN parameters followed by their data type. This line also shows how the specification of multiple parameters of the same type can be combined. In this case `lo_limit` and `hi_limit` are two floating point numbers representing lower and upper set points for the controller.

Note that the specifications of groups of parameters are separated by semicolons. The parameters are terminated by a close parenthesis followed by the keyword `IS`.

Line 5: Beginning of the declaration section. In this section we declare symbolic constants, other variables to be used within the RTSUB, and also any functions or procedures used by the RTSUB.

Line 6: Declaration of the symbolic constants to be used by the RTSUB. This is so the compiler will not confuse these names with variable names. Each symbolic constant is converted into a 16 bit integer using `af_sym()` as described above (in the description for line 2).

Line 7: Beginning of the initialization (INIT) section of the RTSUB. Statements in this section are executed sequentially when the RTSUB is initialized.

Line 8: PRAGMAS are used to control the execution of the rules execution mechanism. This pragma specifies that the mechanism will only trigger rules whose last execution time is before that of any variable in their antecedent. If this is not specified, then it is assumed that all ATTRIB parameters have been changed since the last call to the

RTSUB and all rules affected by any of the parameters are executed. The default is convenient in that the programmer does not have to set-up the time attribute of a parameter before calling the RTSUB. It can, however, result in the unnecessary repeat execution of certain rules. In this case, the time attributes of temperature and control are established in other RTSUBs and passed through to the `temp_ctl()` RTSUB. Thus we can take advantage of this pragma to eliminate unnecessary rule firing.

Line 9: BEGIN signals the start of the action rules. Comments begin with `—` and run to the end of line.

Line 10: This is a top-level rule. Execution of top-level rules is triggered if a variable in their antecedent changes. Rules can also be nested within the consequents of other rules. Nested rules are executed sequentially.

Note that in the first predicate clause we are doing symbolic reasoning. In the second clause, we are reasoning about whether heater contains a valid value. In DSL generated code, undefined data is represented by “magic” numbers. This enables the compiler to reason about actions to take if certain data is undefined. RTSUBs make less assumptions about their calling state than do most C/C++ subroutines. This RTSUB could be called with all of its attributed parameters in their undefined initial state.

Note that in DSL, unlike C/C++, a single `=` sign is used in a test for equality and that `:=` is used for assignment.

When an assignment is performed, not only is the value of the variable set but so is the time last changed.

Rules can occupy multiple lines. They are terminated with a `;`.

This rule says that if we do not yet have a heater setting and the control is in the RUN state then turn the heater on.

Line 11: If the control setting is unknown, or the control is in the stop state, turn the heater off.

Line 12: This is the start of a multi-line rule. Note that we are mixing symbolic and numeric reasoning.

Line 13: This predicate clause is doing temporal reasoning. The variable `time_now` is the current time. The attribute `time` of the variable `heater` is the time at which the variable was last changed. This clause ensures that if we have just turned the heater on we will wait 3 seconds minimum before we turn it off again. This is to prevent minor fluctuations in temperature from causing the heater to be rapidly turned on and off.

Line 14: Note that `heater` appears both in the antecedent and consequent of this rule. The rule execution mechanism inhibits rules such as this from self re-triggering, otherwise we could have an endless loop.

Lines 15-17: Another multi-line rule, similar to the last.

Line All RTSUBs are terminated with `END;`

18:

This temp_ctl.dsl RTSUB file will be converted by the RTX compiler into the file temp_ctl.c for compilation by a standard C/C++ compiler. The temp_ctl.c file will contain two callable subroutines:

init_temp_ctl() must be called to initialize the RTSUB before use

temp_ctl() - the action rules converted to C/C++ language.

In the initialization procedure, all attributed variables (including parameters to the subroutine) are initialized to undefined. Then the initialization (INIT) section of the RTSUB is run (if it contains one). Both subroutines are called with the parameters specified for the RTSUB.

The subroutines assume that the user has allocated memory for the ATTRIB parameters, including the time and importance attributes, before calling these subroutines. Also the subroutines assume that these structures are declared such that data stored in them is retained from call to call.

The RTX compiler also generates a file temp_ctl.h that contains prototypes for the subroutines from the .dsl file as well as type declarations for any record types declared before the RTSUB in the .dsl file.

A sample main program to call this RTSUB is shown below. This program also calls three other RTSUBs which are described later. The main program listing is followed by a detailed description.

In main program file my_main.c:

```

1  #include <rtx.h>
2  #include "temp_ctl.h"
3  #include "set_heat.h"
4  #include "get_temp.h"
5  #include "get_ctl.h"
6
7  main()
8  {
9  static dsl_FLOAT temperature;
10 static dsl_SYMBOL control;
11 static dsl_SYMBOL heater;
12 float lo_limit,hi_limit;
13 lo_limit = 90.0; hi_limit = 110.0;
14 init_temp_ctl(&temperature,&control,&heater,lo_limit,hi_limit);
15 init_set_heater(&heater);
16 init_get_temperature(&temperature);
17 init_get_ctl(&control);
18 while(1)
19     {
20         get_ctl(&control);
21         get_temperature(&temperature);
22         temp_ctl(&temperature,&control,&heater,lo_limit,hi_limit);
23         set_heater(&heater);
24     }
25 }
```

An explanation of this main program follows:

- Line 1: The include file `rtx.h` includes all the declarations of standard procedures and data structures needed to interface with an RTSUB.
- Line 2: The include file `temp_ctl.h` is generated by the RTX compiler from the `temp_ctl.dsl` file. It contains prototypes for the application callable C/C++ subroutines created by the compiler and for any ATTRIB record data types. The definitions of ATTRIB record structures for elemental data types such as floating point numbers are contained in `dsl_usr.h` which is included in `rtx.h`.
- Lines 3-5: The generated include files for the other RTSUBs.
- Lines 9-11: Declarations of the record structures for the three ATTRIB parameters passed between the RTSUB subroutines and the main program. The declaration of `dsl_FLOAT` is:

```
typedef struct
{
float dsl_data; /* value of variable */
dsl_attr_type dsl_attr; /* attributes of variable
                        (time last modified and importance).*/
} dsl_FLOAT;
```

The declaration of `dsl_SYMBOL` is:

```
typedef struct
{
SYMBOL dsl_data; /* value of variable
                  (time last modified and importance).*/
} dsl_SYMBOL;
```

Other elemental data types are likewise defined in `dsl_usr.h`.

The type `SYMBOL` is synonymous with a short 16 bit integer.

Time is kept in seconds and milliseconds:

```
typedef struct
{
long aft_secs; /* Standard time in seconds. With
               * most C/C++ run-time libraries this is
               * expressed in seconds since 00:00
               * hours on Monday January 1, 1970.
               */
short aft_ms; /* Milliseconds after the second.
              */
} af_time;
```

The time attribute in the record should be set whenever the value of a variable is changed. This is done automatically by an RTSUB but has to be programmed if the variable value is changed in C/C++ code. The data importance is only used in data and goal directed chaining. If used, it should be set on a 1 to 10 scale with 1 being the least important and 10 being the most important. The default value assumed is 5.

The time attribute can be set to the current time by calling `aft_get(ptime)`, where `ptime` is a pointer to an `af_time` struct.

The data structures are declared as static so that their data is retained between calls to the RTSUBs. In this example, because the RTSUBs are called from `main()`, they do not strictly need to be declared static. If an RTSUB is called from a subroutine,

however, it is essential that these be declared as static because C/C++ subroutines use the stack for variable data storage which is dynamic (the default).

The data structures for the parameters must be declared, thereby allocating memory for them, before the initialization procedure for the RTSUB is called. This is because the initialization procedure writes into these structures.

- Line 12: Declaration of non-attributed parameters to be passed to the RTSUB.
- Line 13: Initializing these non-attributed parameters. It is as well to do this before calling the initialization procedures for the RTSUBs. This is because these parameters may be used in the INIT section of an RTSUB.
- Line 14: Here we are calling the initialization procedure of an RTSUB. This must be done before the RTSUB rules subroutine is called. The RTX compiler automatically generates the initialization procedure for each RTSUB. This procedure:
1. Sets up the lists of predicate clauses and rules affected by changes to each variable.
 2. Looks up the numerical equivalent of all symbolic constants using `af_sym()`.
 3. Initializes each value field of all attributed parameters and any other attributed variables used by the RTSUB as undefined. It does this by setting the field to a magic number according to its type. It also sets the time attribute to the current time and sets the importance field to 5 (the default importance).
 4. Runs the INIT section of the RTSUB if it has one.

The name of the initialization procedure is always the name of the RTSUB subroutine (the name following the keyword RTSUB) with the prefix "init_".

The initialization procedure is called with the same parameters as specified for the RTSUB. Any parameter specified as an OUT parameter will not be modified unless so specified in the INIT section of the RTSUB. It is always a good idea to declare these OUT parameters so as to allocate memory for them before calling the initialization procedure. Even if your RTSUB does not refer to the OUT parameter in its INIT section, someone may come along later and change this.

ATTRIB parameters are passed by address, that is with a pointer to a block of memory containing the data and its attributes. ATTRIB parameter values and attributes are modified by the rules execution mechanism as it runs. IN parameters are passed by value and cannot be changed by the RTSUB. OUT parameters are passed by address. These point to regular C/C++ variables or structs that can be modified by the execution of the rules.

OUT parameters usually only appear in the consequents of rules. If an input parameter is to be passed by address then it must be declared as INOUT. Such an INOUT parameter can appear in both the antecedent and consequent of a rule.

- Lines Initializing other RTSUBs. Note that these RTSUBs share attributed parameters.

15-17: Thus the temp_ctl() RTSUB. The attributed variable heater will be passed to the set_heater() RTSUB.

Order is important in calling the initializing procedures if their RTSUBs contain INIT sections. This is because the initialization procedure always initializes the parameter to undefined. Thus it was important to call init_temp_ctl() before init_get_temperature(). This latter RTSUB initializes the temperature to the starting ambient value in its INIT section. If its initializing procedure were called first, then the temperature value would be set to undefined again when init_temp_ctl() was called.

Line 18: We are going to execute the RTSUBs in an infinite loop. The RTSUB get_ctl() causes the program to be terminated upon typing ESCAPE on the keyboard.

Lines 20-23: The call to execute an RTSUB's rules is made by calling a C/C++ subroutine with the same name and the same parameters as declared for the RTSUB. The attributed parameters are visible at the calling level and their values and attributes may be modified between calls.

Let us now examine the other RTSUBs used with this example. The first is a subroutine to get new temperature data every second:

In file get_temp.dsl:

```

1  RTSUB get_temperature(temperature: ATTRIB FLOAT) IS
2  DECLARE
3      DYNAMIC FUNCTION get_input RETURN FLOAT;
4  INIT
5      temperature := get_input;
6  BEGIN
7      IF time_now - temperature'time > 1 second THEN
8          BEGIN
9              temperature := get_input;
10             PRINC "temperature is ", temperature, " at ",time_now;
11         END;
12     END;
```

The get_temperature() RTSUB gets the temperature from a sensor every second by calling a subroutine get_input() which may get its input from an A/D converter connected to a temperature probe inside the chamber. An explanation follows of those lines not previously explained:

Line 3: All C/C++ functions and procedures called by the rules must be declared before they are used. In this case, the function get_input() has no parameters and returns a floating point value. Its C/C++ equivalent is:

```
float get_input(void);
```

The form of declaration used for C/C++ functions and procedures is borrowed from the Ada language. That is parameters are declared as IN, OUT, or INOUT (which we met previously). Thus the C/C++ function:

```
float sin(float x)
```

is declared as

```
FUNCTION sin(x: IN FLOAT) RETURNS FLOAT;
```

Procedures do not return any value, thus

```
PROCEDURE xyz(a:IN SHORT;b:OUT FLOAT);
```

is equivalent to the C/C++ procedure:

```
void xyz(short a, float *b);
```

Note that IN parameters are passed by value and OUT parameters are passed by address. In an RTSUB, if an attributed variable is used as an IN to a C/C++ subroutine then only the value part is passed. If it is used as an OUT or INOUT then a pointer to its data structure is passed.

OUT and INOUT parameters are assumed to change during a call to a C/C++ subroutine, as are return values. The rules mechanism will automatically update the time attributes of the affected variables and trigger the appropriate rules for execution as a result.

The function is declared as DYNAMIC so that it will be re-evaluated every time that a statement containing this procedure is executed. The RTX compiler generates optimized code so that a subroutine is only called if one of its arguments changes. In this example, the value returned from `get_input()` will change even without changes to its (non-existent) arguments. Such subroutines must be declared as dynamic so that the RTX compiler will generate code that re-evaluates the subroutine every time its return value is needed. While this is only necessary if the function is called in the predicate part of a rule, it is still a good idea to declare it as such.

- Line 5: If a function call is placed in the INIT section, as in this case, then it will be executed when the “init_” initialization subroutine is called. In this case we are getting the initial temperature of the chamber. Note that when there are no parameters for a subroutine call, the parentheses are omitted (as they are in the declaration).
- Line 7: Here we have a time dependent top-level rule. It will be evaluated every time the RTSUB is called because *time_now* is presumed to have changed. The antecedent will only be true if the current time is more than one second later than the time at which we last took a temperature value (temperature’time). If less than a second has elapsed, the antecedent is false and, there being no further rules to execute, the RTSUB returns. Hence the nested block which is the consequent of the rule will only be executed once a second.
- Line 8: Here we have the beginning of a nested statement block. Statements within a nested block are executed sequentially. This includes rules nested within the block. Thus only top-level rules are data driven. These nested rules can contain blocks that contain other nested rules and so forth. This enables RTSUBs to integrate both event driven and step by step reasoning.
- Line 9: Here we are calling the C/C++ subroutine to get a new value of temperature. The rules execution mechanism will automatically update the *time* attribute so that the antecedent will ensure that new data is only collected once a second.
- Line 10: Here we are printing out the value of the temperature and the time at which it was taken. Because RT-Expert knows about the format of data used in an RTSUB, it is able to perform much of the formatting of a print statement automatically. The DSL language contains two forms of print statement: PRINT and PRINC. The only

difference is that PRINC automatically appends a carriage return to the end of the line. Print statements are explained in detail in the users manual. In this case the time will be printed in absolute time format which is the date followed by hours:minutes:seconds.milliseconds.

Line 11: A nested block starts with BEGIN and terminates with END.

The temperature value obtained by the get_temperature() RTSUB will be passed to the temp_ctl() RTSUB through the main program (see lines 21 and 22 of the listing for main()). Because the temp_ctl() RTSUB has the pragma set to only trigger rules if ATTRIB parameters have changed, the rules that are dependent on the temperature variable will only be fired when this variable changes. As we will see below, the *control* variable only changes when specified by the user. Also the *heater* variable only changes if *temperature* or *control* change. Thus, most of the time, the temp_ctl() RTSUB will return without executing any rules as *temperature* and *control* have not changed. Normally it will only execute its rules once a second, when the *temperature* variable changes.

Again, note that this is very different from C/C++ code where execution is controlled by the value of variables. With rules it is the fact that variables have changed that is a major controlling factor. This data driven nature of rules makes it possible to write very efficient decision routines in very few lines of code.

In our example get_temp() gets the temperature, temp_ctl() sets the heater to an ON,OFF state and set_heat() calls a subroutine to actually turn the heater on and off. A listing of set_heat.dsl follows:

In file set_heat.dsl:

```
1 RTSUB set_heater(heater: ATTRIB SYMBOL) IS
2 DECLARE
3     PROCEDURE set_output(val:SHORT);
4     ON,OFF ARE SYMBOLIC CONSTANT;
5 INIT
6     PRAGMA RULE_TRIGGER IS NEW_DATA;
7 BEGIN
8     IF heater HAS CHANGED THEN PRINC "heater ",heater;
9     IF heater = ON THEN set_output(1);
10    IF heater = OFF THEN set_output(0);
11 END;
```

An explanation of lines that have not already been explained follows:

- Line 3: Defines the function that will be used to set the switch to turn the heater ON or OFF. Note that this has a single input parameter. If a parameter type [IN,OUT,INOUT] is not specified, then it is assumed to be IN.
- Line 6: Rule triggering only on new data is invoked so that we will not keep turning the heater ON if it is already ON and the same for the OFF state.
- Line 8: Note the construct HAS CHANGED. HAS CHANGED causes the rule to be fired whenever the variable preceding the keywords HAS CHANGED is changed. Note that if the variable *heater* changes to undefined, then the print statement will cause the value of *heater* to be printed as “<undefined>”. Note that, with the use of the pragma

on line 6, this could have been shortened to simply:

```
PRINC "heater ",heater;
```

A top-level (not nested) print statement will be triggered whenever any variable that it is printing changes. Because the pragma inhibits rule execution unless heater changes, then printout will occur when heater changes. Without the pragma, however, this statement would print out the value of heater whenever the subroutine was called.

Note that the symbolic variable will be printed as ON or OFF. That is, the string equivalent to its contents will be retrieved from the hash table by calling `af_desym()` before printing.

The final RTSUB in our example is used to control the running of our program from the keyboard. Note that this code is platform specific and will only work as a Console Application under Win32 where you can access the keyboard directly. It takes in an escape character to terminate execution. If the user types R or r then control is set to RUN. If the user types S or s then control is set to STOP. A listing of `get_ctl()` follows:

In file `get_ctl.dsl`:

```
1 RTSUB get_ctl(control: ATTRIB SYMBOL) IS
2 DECLARE
3     RUN, STOP ARE SYMBOLIC CONSTANT;
4     c IS CHAR;
5     DYNAMIC TIME FUNCTION kbhit RETURN INT;
6     DYNAMIC FUNCTION getch RETURN CHAR;
7     PROCEDURE exit(errcode: IN INT);
8 BEGIN
9     IF kbhit /= 0 THEN c := getch;
10    IF c = ESC THEN exit(0);
11    IF c = 'r' OR c = 'R' THEN control := RUN;
12    IF c = 's' OR c = 'S' THEN control := STOP;
13    IF c HAS CHANGED THEN PRINC "Control is ", control;
14 END;
```

A description of lines that have not previously explained follows:

Line 4: The variable `c` is an attributed variable that is internal to the RTSUB. Setting the value of the character `c` causes rules to fire (lines 10, 11, 12, and 13).

Line 5: The function `kbhit()` is a standard library routine valid in Console Applications that returns the number of characters in the keyboard input buffer. It is declared DYNAMIC so that RTX knows that its value can change from one invocation to the next, even though its parameters do not change. By default, RTX generates code to avoid re-evaluating any rule predicate clause if there are no changes to any values referred to in the predicate clause. This optimization does not work if a subroutine in the predicate clause returns a different value each time it is called. I/O interface routines fall in this category as do functions such as random number generators.

The function `kbhit()` is also declared as a TIME function. This causes `kbhit()` to trigger any rule it is in the predicate clause of every time the RTSUB is called (i.e. when time changes).

Lines Definition of subroutines from the standard C/C++ library.

6-7:

Line 9: Because `kbhit` is declared as dynamic and time dependent, it is executed every time `get_ctl()` is called. Note also that not equal is `/=` not `!=` as in C/C++. If there is a character in the keystroke buffer then this will be retrieved using `getch()` and set the variable `c`.

Lines 10-12: These will be triggered when `c` changes. Note that DSL has predefined constants for characters such as 'ESC' for escape (A list of these predefined special characters can be found in Section 4.2 in the Programmer's Manual).

Line 13: This will be triggered when `c` changes, printing out the symbolic constant RUN or STOP. If we had simply stated:

```
PRINC "Control is ", control;
```

This would be triggered whenever `control` changes. While this might appear to be what we want, there is an undesired side effect. By default, all ATTRIB variables will be assumed to have changed whenever the subroutine is called. In this case, the rule execution mechanism will assume that `control` has been externally changed whenever this RTSUB is called and will print out a new value of `control`. As this RTSUB will be called in a loop it will be called many times when `control` is not changed by keyboard input.

By conditioning the printout on the character `c` we will only get printout when `c` is changed.

Because this keyboard handling was done in rules and not in C/C++, all the attribute management was handled by the rules mechanism. Thus rules in `temp_ctl()` that are dependent on `control` will only be fired as a result when `control` changes.

Print statements in DSL end up at run-time formatting the text to be printed in strings. The run-time mechanism calls `af_text(char * string_pointer)` to print the string. This procedure is provided in source form so that users can change the output. By default, `af_text()` uses `puts()` to output the string. If the user has integrated the rules with graphics, then this text may well be re-directed to a graphics window for display. This can easily be done by replacing the default `af_text()` with one supplied by the user. For example, a replacement may write output to an edit box in a Windows application.

The final step in putting our example program together is to create a makefile for use with a make utility. A example makefile follows with explanation:

In Makefile:

```
1. # Copyright (c) 1994-96 All Rights Reserved
2. # BellHawk Corporation
3. #
4. # Directories
5. #
6. # This should be set to the directory where you installed RT-Expert
7. RTX = d:\rtexpert
8. # The include directory - where to find the RT-Expert '*.h' files.
```

```

9.   RTXINC = $(RTX)\include
10.  #
11.  # Implicit rules
12.  #
13.  .SUFFIXES: .dsl .afs
14.
15.  .dsl.c:
16.      $(RTX)\bin\rtx1 -I$(RTXINC) $<
17.      $(RTX)\bin\rtx2
18.
19.  .dsl.h:
20.      $(RTX)\bin\rtx1 -I$(RTXINC) $<
21.      $(RTX)\bin\rtx2
22.
23.  .afs.h:
24.      $(RTX)\bin\afs2h $<
25.
26.  #
27.  # Macros - define directories, compiler options, etc.
28.  #
29.
30.  # Uncomment the following only for debugging
31.  #DEBUG = /Zi
32.  # Uncomment the following only for optimized execution
33.  DEBUG = /O
34.
35.  CFLAGS = $(DEBUG) /I$(RTXINC) /DWIN32 /Zp4 /DCONSOLE
36.
37.  #
38.  # Explicit rules - specific to this program
39.  #
40.
41.  OBJS=my_main.obj temp_ctl.obj set_heat.obj get_temp.obj get_ctl.obj
42.
43.  all: stock.exe
44.
45.  clean:
46.      del *.exe
47.      del *.obj
48.      del *.lst
49.      deldslcs
50.
51.  my_main.exe: $(OBJS)
52.      link @<<
53.      /DEBUG /OUT:my_main.exe /SUBSYSTEM:console /INCREMENTAL:no
54.      kernel32.lib
55.      user32.lib
56.      gdi32.lib
57.      /MACHINE:I386
58.      $(OBJS)
59.      $(RTX)\LIB\rtx_mc32.lib
60.      <<
61.
62.  my_main.obj: my_main.c temp_ctl.h set_heat.h get_temp.h get_ctl.h
63.
64.  temp_ctl.obj: temp_ctl.c
65.  temp_ctl.c: temp_ctl.dsl
66.  temp_ctl.h: temp_ctl.dsl
67.
68.  set_heat.obj: set_heat.c
69.  set_heat.c: set_heat.dsl
70.  set_heat.h: set_heat.dsl
71.
72.  get_temp.obj: get_temp.c

```

```
73.  get_temp.c: get_temp.dsl
74.  get_temp.h: get_temp.dsl
75.
76.  get_ctl.obj: get_ctl.c
77.  get_ctl.c: get_ctl.dsl
78.  get_ctl.h: get_ctl.dsl
```

This makefile is for Microsoft Visual C/C++++ Version 2.0 or higher.

Note that the generation of the .h files from the .dsl files is explicitly stated. This allows for the re-generation of the .h files if the .dsl file changes.

Also note the inclusion of the runtime library *rtx_mc32.lib* in the project. It is also important to use the specified CFLAGS when compiling your application as the generated code needs them to work.

5. Other Topics

Variables can be declared as numeric data types such as floating point or integer numbers. They can be declared as boolean data types which are true or false or they can be declared as characters or strings of characters. They can also be declared as symbolic data types which have values such as ON or OFF.

Variable declarations for numbers can be of the form

```
temperature is FLOAT;
```

or they can use the form:

```
temperature: FLOAT;
```

Other permissible formats are:

```
x,y: LONG;
```

```
x,y are LONG;
```

```
x,y is LONG;
```

DSL requires the declaration of all variables before use. This is so that the DSL compiler can detect as many user errors as possible. For example, this strong data typing enables DSL to check that a floating point number is not added to a string and that the arguments to a function or procedure are of the correct type.

Some of the types that a variable can be declared as being are:

SHORT	- 16 bit integer*
LONG	- 32 bit integer*
INTEGER	- 16 bit integer*
FLOAT	- 32 bit floating point*
DOUBLE	- 64 bit floating point*
BOOLEAN	- TRUE or FALSE
STRING	- null terminated string of characters
CHAR	- single ASCII character
SYMBOLIC	- symbolic variable
GOAL	- goal variable for goal-directed chaining

Record variable types can be declared as in:

```
TYPE x IS
BEGIN
  y:FLOAT;
  z:SHORT;
END;
```

These declarations must precede the use of a type in a .dsl file. If a parameter of the RTSUB is of a user-defined type then the type declaration must precede the RTSUB itself, as in:

```
TYPE atype IS
BEGIN
  y:FLOAT;
  x:SHORT;
END;
RTSUB r (a: ATTRIB atype) IS
.....
```

In such a case, the C/C++ equivalent of the type definition will be placed in the file generated .h file. If a record type is internal to the RTSUB, then it can be declared in the declaration section of the RTSUB, before its use, as in:

```
RTSUB yyy(.....) IS
DECLARE
  TYPE btype IS
  BEGIN
    y:FLOAT;
```

```

        x:SHORT;
    END;
    b IS btype;

```

In this case the C/C++ type declaration is not generated into the .h file.

In the case where the record is a parameter, the RTX compiler will also generate a record type declaration:

```

typedef struct
{
    atype dsl_data;                /* record structure */
    dsl_attr_type dsl_attr;        /* attributes of variable (time
    * last modified and importance).
    */
} dsl_atype;

```

into the .h file. This declaration can then be used to declare a static variable to hold the record and its attributes.

Variables within a record are referred to using a dot convention. Thus the field x of record a is referenced by a.x. Records can be nested and as many dot fields as necessary can be used.

When rules refer to a specific field of a record in their antecedent, they are only triggered by a change to that field, not to any other field in that record. Thus:

```
IF a.x > 10.5 THEN a.y := 2;
```

will only be triggered when the field a.x is changed and will only trigger rules that have a.y in their antecedent. If a record is an OUT parameter from a subroutine call or is returned from a function then it is assumed that all fields have changed. Similarly, if the whole record is set in an assignment statement, then it is assumed that all fields have changed.

If y.z has subfields, then it is assumed that y.z has changed if any of its subfields change. This applies at any level of record nesting.

If an ATTRIB variable is a parameter to an RTSUB and it is determined to have changed, all fields are assumed to have changed.

All variables can be in one of two states: defined or undefined. If a variable has been assigned a value, either through the execution of a rule or through the arrival of a new value for the variable in a parameter, then it becomes defined. A variable is undefined before it is set. It can become undefined in a statement such as:

```
if y = defined then x:= undefined;
```

Note that defined and undefined are keywords. Also note that the above rule will be evaluated whenever a new value is assigned to y and its antecedent will always be true in such a case.

If a consequent statement such as:

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

is executed, then a will be set to undefined if either b or c is undefined. In general, the left hand side of any consequent statement will be set to undefined if any variable on the right hand side is undefined. The exceptions to this are procedure or function calls which are called with undefined variables set equal to the appropriate “magic” numbers. It is up to the subroutine to determine how to handle undefined variables.

If the subroutine being called cannot handle undefined numbers, then the rule should be constructed in such a way as to preclude this such as in:

```
IF x = defined THEN y := sin(x) ELSE y := undefined;
```

A top level rule such as:

```
IF b HAS CHANGED and c HAS CHANGED THEN a := b + c;
```

will recompute a whenever b or c change and will set a to undefined if either b or c become undefined.

The only form of expression where this propagation of defined and undefined is not followed is in boolean expressions. If x, y, and z are booleans then:

```
z := x OR y;
```

will give z a value of true if either x or y is defined and true. In this case the other variable can be undefined. If both x or y are undefined then z is undefined. DSL correctly handles the evaluation of booleans that may take undefined states.

We may also want to execute a statement if a variable changes, that is takes on a new value or becomes undefined. We can test for this by using the keyword CHANGED as in:

```
IF aa HAS CHANGED THEN .....
```

This same keyword can be used to cause a rule to be re-evaluated whenever the RTSUB is called as in:

```
IF TIME CHANGED THEN .....
```

An antecedent clause is assumed to be false if there is not enough data to evaluate the clause. For example:

```
IF x > y OR c > d THEN f := TRUE;
```

This rule will be evaluated if x, y, c, or d changes. The antecedent will be true if either x and y are defined and x is greater than y or c and d are defined and c is greater than d.

Antecedent clauses are only evaluated when all their requisite variables contain valid data. For efficiency, they are only re-evaluated whenever a variable in the clause changes. Also the same clause appearing in more than one antecedent is only evaluated once.

A top level rule can be written in the form of a statement. For example:

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

This is equivalent to writing:

```
IF b HAS CHANGED OR c HAS CHANGED THEN a := b + c;
```

These statement rules are triggered by the availability of new data values just like regular rules.

Statements can appear in nested blocks as the consequent of a rule, as in

```
IF aa = TRUE THEN
  BEGIN
    a := b + c;
    z := 2 * a + 3;
    c := undefined;
  END;
```

These statements are executed sequentially if the predicate clause of their top level rule is true. They are not triggered by changes to their right hand side variables.

Rules in DSL can be nested as in:

```
IF w > x THEN
  IF y > z THEN
    f := TRUE;
```

They can also be a part of a statement block as in:

```

IF w > x THEN
  BEGIN
    a := b * c;
    IF y > z THEN f := TRUE;
  END;

```

DSL processes nested rules and statements differently from top-level rules. Statements within a nested BEGIN/END block are processed sequentially. Nested rules are treated much like an IF/THEN statement is treated in a procedural language. That means that the antecedent is evaluated. If it is true, the consequent is executed. If it is false or undefined then the alternate consequent is executed (if present). This is different from a top-level rule where the rule is only evaluated if the data has changed. In a nested rule, the rule is evaluated based only on the value of the data, not the newness of the data.

Statement blocks and rules can be nested to any arbitrary level but only the top level rules are triggered by changes to their antecedent. It is important to remember to include “trigger” variables in the antecedent to a top level rule. For example:

```

IF state = state_1 THEN
  BEGIN
    .....
    zz := aa + bb;
  END;

```

This rule will be evaluated only when *state* changes to *state_1*. This may be what we want, but usually we require that, if we are in *state_1* and *aa* or *bb* changes, then re-evaluate *zz*. In this case we would need to state:

```

IF state = state_1 AND (aa HAS CHANGED OR bb HAS CHANGED) THEN
  BEGIN
    .....
    zz := aa + bb;
  END;

```

If we only wanted the nested block to be evaluated if *aa* and *bb* had values then we would state:

```

IF state = state_1 AND aa = defined AND bb = defined THEN
  BEGIN
    .....
    zz := aa + bb;
  END;

```

Note that keywords in DSL can be in mixed upper and lower case. Variables are case sensitive as are subroutine names and data types.

Users familiar with the Ada language will note the syntactic similarity with Ada. While DSL is functionally very different from an Ada program in the way it executes, Ada syntax has been used for DSL expressions wherever possible. This is because the Ada syntax has been carefully designed to be usable and logically consistent.

Some of the actions of the rule execution mechanism are set by PRAGMA statements. These usually appear in the INIT section of the RTSUB. They control such things as the order of rule execution.

Rules execution order is determined by the dynamic importance the rules hold within an RTSUB. This importance can be determined by a number of methods set by PRAGMA ORDER. These are:

LEXICAL: The rule’s importance is based on lexical order (the earlier rules in the RCO are more important than the later). This is the default.

DATA: The rule's importance is inherited from the importance of its most important data item. This allows important data items to pre-empt less important data items in flowing through the rules.

RECENCY: Rules whose antecedent variables have been most recently changed are given precedence. This gives priority to reactive chains.

The use of DSL rules has considerable advantages over coding the if...then...else... blocks in a procedural language such as C/C++. The problems with using a procedural language for decision making include:

- a) Typically only a small percentage of the rules are affected by an input data variable (such as *temperature* in the example). Using a conventional language, a programmer has to insert additional code so that only those rules for which there is valid data are executed. For example, the rule:

```
IF x > y THEN c := a + b;
```

can only be executed if valid data values are available for x and y. Also, if x is greater than y then c becomes undefined if a or b is undefined. This means that the programmer in a conventional language has to insert considerable code to track which variables contain valid data and which if...then...else... statements can be executed as a result. This is all taken care of automatically by the DSL rules execution mechanism.

Further, it is desirable from an efficiency viewpoint, to only execute a rule when new values are available for variables which affect the left hand side condition of the rule. As soon as there are more than a few rules, the complexity of the code, using conventional methods, increases exponentially with the number of rules. This is made worse by the fact that variables set on the right hand assignment side of one rule often appear on the left hand condition side of a number of other rules. This requires that the programmer add code to force execution of rules which can be executed as a result of the execution of other rules. Also the programmer has to provide code to decide which of a number of possible rules should be executed if the setting of a variable results in the left hand side condition of a number of rules becoming true. All this results in very complex code which is hard to modify and maintain.

- b) The rules in the decision code modules are the most frequently changed part of most systems as they contain the knowledge of the system which evolves over time as the system's usage changes. For example, it may be desirable to change the temperature at which to maintain a room or to make this a function of both the temperature and the humidity. If the rules are written in a procedural language then it is very difficult to make changes as the changes to one rule may affect the code linking this rule to many other rules, and these to many more rules.

It should be noted that the complexity problem is especially severe when the system has to function in real-time. That is, it has to respond in a timely manner to randomly arriving data from multiple sources. DSL solves this problem by generating the code to automatically sequence the execution of the rules.

DSL rules follow similar principles to expert systems rules except that:

1. They are oriented towards the building of real-time or on-line systems

2. They are compiled and not interpreted
3. They are trigger-driven in a data flow manner
4. They are strongly data typed to allow automatic verification by the compiler
5. They are oriented towards reasoning about the times at which events occur
6. They are able to perform computations with variables which are defined only over a limited time.
7. They can be automatically re-triggered by the passage of time.
8. They have alternate action (else) clauses which are needed for efficient interpretation of data.

Unlike expert systems which use large monolithic sets of rules, RT-Expert encourages programmers to break their rules down into objects which represent limited decision domains. These objects are then coded as RTSUBs which can be developed, tested, and maintained as separate entities. Typically the RTSUBs will contain 10 to 30 rules which is a good compromise between ease of development and efficiency of execution. This also allows users to develop libraries of re-usable knowledge modules.