

Summary

This document describes the policies guiding software development at Relational Technology Inc. The basic policy is that high quality software can be developed with and supported by the minimum set of standards satisfying the following requirements:

1. Provide a tool for defining the functions and technical feasibility of a particular product.
2. Provide a quality control tool which will assist in determining the quality of the design and implementation of a product.
3. Assist in training new personnel.
4. Serve as a tool useful in maintaining the product.

Our development environment has certain characteristics which influence the type of standards required. These characteristics are described below.

1. Projects are small, usually one person efforts reducing the need for extensive interface documentation within a project.
2. Most projects will interface with the system through QUEL, reducing the complexity of subsystem interface descriptions.
3. Portability is of primary concern.

In this type of environment the requirements can be met with the following standards:

1. A product specification describing the functional operation of the product and the technical approach and overall organization of the implementation.
2. Coding standards to establish uniformity across product implementations.
3. Test procedures for acceptance and regression testing.
4. Portability standards for insuring that the products are independent of any particular environment.

Each of these standards will be described in the following sections.

Product Specification

The product specification describes the functional capabilities of the product, its operation and the technical approach to be used in implementing the product. This document is prepared and reviewed prior to the start of implementation and serves as the "design" against which the implementation will be measured. It is organized as follows:

1. Product Identification

A one page summary of the product or project.

1.1 Name.

1.2 Product Abstract.

2. Product Description

2.1 External specification.

Describes the functional capabilities and general operation of the product.

2.2 Internal specification

Describes the approach to be used in implementing and technical risks associated with the project (if such risks can be identified). This section of the document should include any of the following characteristics which are relevant to the particular product being described.

1. Relationship of the product to other elements of the system from an internal viewpoint. What processes it communicates with and how that communication is carried out is an example of this kind of information.
2. Protection considerations.
3. Synchronization considerations.
4. Overall structure of the program(s) describing the major modules, how they interact, and any major data structures used to communicate among modules.
5. How the program(s) is initiated and terminated if this is at all obscure.

6. Technical risk areas not covered above. Such things as use of a new language or compiler, unfamiliar system services, portability, new types of interactions between processes or algorithms which are not well understood.
7. Test procedure operating instructions.

3. Project Planning and Organization.

- 3.1 Hardware requirements.
- 3.2 Software requirements.
- 3.3 People requirements.
- 3.4 Development schedule.

The object of this document is to get critical information about a development in concrete form before beginning implementation. Such information can help us in scheduling and in reducing the number of "surprises" encountered as well as providing a review tool for improving the designs.

Coding Standards

The coding standards will be the INGRES project coding standards with the enhancements described below:

1. The block of comments at the beginning of each file or procedure should have the following format.

Name of procedure and a one-line description of its function.

More detailed description of the function of the procedure, the algorithm used (if non-obvious) and any major data structures defined within the module.

Arguments required.

Results returned.

Non-obvious side-effects of the procedure. An obvious side-effect would be I/O activity in a procedure named "read". A non-obvious side-effect would be setting a global variable to indicate EOF.

2. Comments should be placed at the beginning of each block of code to describe the function performed by that block of

code. This is especially important in larger modules performing several sub-functions. Some judgement must be exercised so that the code doesn't become too cluttered with comments. Short modules and those that are easily understood without additional comments can be written without this "paragraphing".

The INGRES project coding standards are attached to this document.

Test Procedures

Each product released by the development group must be accompanied by a test procedure and a set of expected outputs for the test procedure. This test procedure should exercise the product to demonstrate the products adherence to the functional specification and the products ability to handle errors generated by users. Usually, the exercisor should be a command script and the expected output a text file such that the exercisor output may be directed to a file and then compared with the expected output by machine.

This type of test procedure will be useful for both acceptance testing and regression testing as the product is modified. If the product is later modified, a new version of the test procedure must be developed to maintain consistency between the product and its test procedure. As bugs are detected in the product appropriate tests should be added to the test procedure to catch such problems in future releases of the product.

Portability

Enhancing the portability of products which are intended to run in multiple environments can be accomplished by isolating the product from the environment. This is achieved by insisting that the product access resources provided by the environment through a set of well defined abstractions. INGRES and its associated products must access environment supplied resources through routines in the library GUTIL. If new environment resources are required appropriate abstractions must be defined and placed in GUTIL. The library VMS contains routines needed to access VMS capabilities from C. These routines should be used only in GUTIL since they are environment specific.

A few examples will make this discussion less abstract. A environment resource might be the ability to obtain a list of all the files in a specified directory. This resource should not be used in any code outside of GUTIL since it depends on the particular form of the implementation of that resource in each environment. Further, the file names which are produced by this facility are also dependent on the environment in which the system is run. Thus, these names should not be used outside GUTIL. An INGRES abstraction defining the form of the filenames used in the INGRES system must be produced and the routines placed in GUTIL for converting from environment names to INGRES names. Other examples of environment services which cannot be used outside CUTIL are spawning processes,

input/output, interrupt processing, interprocess communications, special facilities of the compiler or O/S.

Other things will impact portability (such as how the linker will handle externals). Such problems cannot be solved by creating an abstraction. In this case the problem should be handled by structuring code in such a way that it will work in the largest subset of environments which can be anticipated. For example, only defining global symbols once so it will work on linkers which allow both a single definition and ones which allow multiple definitions. Our conventions in this case will be to place all global variables used by routines in a directory in a file called "globals.h" while the initializations of those variables (causing a definition of the variable) will be placed in a file called "globals.c". Variables which are global to the functions in a file but local to that file, will not have to follow this rule.

Other factors influencing portability include:

1. Command line interpretation and the use of upper/lower case character sets.
2. Compiler differences.
3. Non-portable language constructs (compiler and machine dependent).

As we determine how to handle these classes of problems, this section of the document will be expanded.

Apr 25 11:00 1977 /mnt/ingres/doc/other/example.c Page 1

```
# include      "/usr/sys/param.h"
```

```
/*
```

```
** DEMO PROGRAM
```

```
**
```

```
** This hunk of code does virtually nothing of use. Its main  
** purpose is to demonstrate the "official" ingres coding style.
```

```
**
```

```
** This demonstrates comments. There should be a block comment  
** at the beginning of every file and/or procedure to explain  
** the function of the code. Important information to put here  
** includes the parameters of the routines, any options that the  
** user may specify, etc.
```

```
**
```

```
** The first line of the comment should be a one-line description  
** of what's going on. The remainder should be more detailed.  
** Blank lines should separate major points in the comments. In  
** general, ask yourself the question, "If I didn't know what this  
** code was, what it was for, how it fit in, etc., and if I didn't  
** even have the documentation for it, would these comments be  
** enough for me?"
```

```
**
```

```
** Some general guidelines for the code:
```

```
**
```

```
** - Commas and semicolons should always be followed by a space.  
**   Binary operators should be surrounded on both sides by  
**   spaces. Unary operators should be in direct contact  
**   with the object that they act on, except for "sizeof",  
**   which should be separated by one space.
```

```
**
```

```
** - Two statements should never go on the same line. This includes  
** such things as an if and the associated conditionally  
** executed statement.
```

```
** In cases such as this, the second half of the if  
** should be indented one tab stop more than the if. For  
** example, use:
```

```
**         if (cond)  
**             statement;  
**  
** never:  
**         if (cond) statement;  
**  
** or:  
**         if (cond)  
**             statement;
```

```
**
```

```
** - Braces ({} ) should (almost) always be on a line by them-  
** selves. Exceptions are closing a do, and terminating  
** a struct definition or variable initialization. Braces  
** should start at the same indent as the statement with  
** which they bind, and code inside the braces should be  
** indented one stop further. For example, use:
```

```
**         while (cond)  
**         {  
**             code  
**         }  
**  
** and never:  
**         while (cond)
```

```
**
```

```

**          (
**          code
**          )
**
**      or:
**          while (cond) {
**              code
**          }
**
**      or:
**          while (cond)
**          {
**              code
**          }
**
**      or anything else in that line. Braces which match
**      should always be at the same tab stop.

```

- Declarations should always have at least one tab between the declaration part and the variable list part, but never any tabs within the declaration part or the variable list part. For example, in the line:

```

**      register int    i, j;
**
**      There is a tab between the "int" and the "i", but a
**      space between "register" and "int", since together
**      these make up the declaration part.

```

- There should always be a space following a keyword (i.e., for, if, do, while, switch, and return), but never between a function and the paren preceding its arguments. For example, use:

```

**      if (i == 0)
**          exit();
**
**      never:
**      if(i == 0)
**          exit ();

```

- Every case in a switch statement (including default) should be preceded by a blank line. The actual word "case" or "default" should have the indent of the switch statement plus two spaces. It should be followed by a space (not a tab) and the case constant. Multiple case labels on a single block of code should be on separate lines, but they should not be separated by blank lines. The switch statement should in general be used in place of such constructs as:

```

**      if (i == 1)
**          code1;
**
**      else
**          if (i == 34)
**              code2;
**          else
**              if (i == -1643)
**                  code3;

```

which can be more succinctly stated as:

```

**      switch (i)
**      {
**
**          case 1:

```

```
**          code1;
**          break;
**
**          case 34:
**              code2;
**              break;
**
**          case -1643:
**              code3;
**              break;
**
**      }
```

In point of fact the equivalent switch will compile extremely efficiently. (Note that if you had some instance where you could not use a case, e.g., checking for $i < 5$, else check for $j > 3$, else whatever, then the above ("if") code is in the correct style. However, an acceptable alternate structure is to consider "else if" as a primitive. Hence:

```
if (i < 5)
    code1;
else if (j > 3)
    code2;
else
    code3;
```

is acceptable.

- Do statements must always be of the form:

```
do
{
    code;
} while (cond);
```

even if "code" is only one line. This is done so that it is clear that the "while" is with a do, rather than a standalone "while" which is used for the side effects of evaluation of the condition.

- Defined constants (defined with the # define feature) must be entirely upper case. The exceptions to this are compilation flags, which begin with a lower case "x", and some sub-types for parser symbols. In any case, the majority of the symbol is upper case.

- Global variables should begin with an upper case letter and be otherwise all lower case. Local symbols should be entirely lower case. Procedure names are all lower case. The only exception to this is the trace routine "tTf". You should avoid user non-local symbols (globals or # define'd symbols) which are one character only; it is impossible to distinguish them.

- # defines and # includes should have a space after the sharp sign and be followed by a tab. In general, try to make things line up. Use:

```
# define      ARPA          25
# define      MAXFIELDS     18
```

```
**      and not:
**      #define ARPA 25
**      #define MAXFIELDS 18
**      Conditional compilation statements should have as many
**      tabs as are necessary to bring the "ifdef",
**      "ifndef", or "endif" to the tab stop of the surrounding
**      code. The keyword ("ifdef" or "ifndef") should be
**      followed by a space and the conditional compilation
**      variable. Conditional compilation should be used
**      around all trace information, timing code, and code
**      which may vary from version to version of UNIX. See
**      the code below for an example of conditional compila-
**      tion use.
**
** - A blank line should separate the declarations and the code
**   in a procedure. Blank lines should also be used freely
**   between major subsections of your code. The major
**   subsections should also have a comment giving some idea
**   of what is about to occur.
**
** - Use descriptive variable names, particularly for global var-
**   iables. "IEH3462" tells me nothing; nor does "R". On
**   the other hand, "Resultid" tells me quite a lot,
**   including what it might be, where I might go to see
**   how it is initialized, etc. Try not to use variables
**   for multiple purposes.
**
** - It is quite possible to name a file "printr.c" and then
**   put the code for "destroydb" in it. Try to arrange
**   the names of your files so that given the name of a
**   routine, it is fairly easy to figure out which file
**   it is in.
**
** - Sometimes it is really pretty much impossible to avoid doing
**   something which is not immediately obvious. In these
**   cases, put in a comment telling what you are doing and
**   why you are doing it.
**
** - Try to write things that are clear, rather than things which
**   you think are easier to compile. I mean, who really
**   cares? For example, always declare temporary buffers
**   as local, rather than as global. This way you can
**   guarantee that you will never clobber the buffer in
**   another routine accidentally when it still had useful
**   info in it.
**
** Remember, it is easy to write incomprehensible code in
** C. If you really get off on doing this, however, go get
** a job programming in APL.
**
** For efficiency reasons, you should always use register variables
** when possible. A simple and extremely effective tip is to define
** a register variable, and assign an oft-used parameter to it,
** since it is particularly inefficient to reference a parameter.
** Another particularly inefficient operation is referencing arrays
** of structures. When possible, define a register pointer to the
```

```

**      structure, and then say:
**      struct xyz          structure[MAX];
**      register struct xyz *p;
**      ...
**      for (i = 0; i < MAX; i++)
**      {
**          p = &structure[i];
**          p->x = p->y + p->z;
**          (diddle with p->???)
**      }
**
** and not:
**      struct xyz          structure[MAX];
**      ...
**      for (i = 0; i < MAX; i++)
**      {
**          Structure[i].x = Structure[i].y + Structure[i].z;
**          (diddle with Structure[i].???)
**      }
**
** Remember, the nice things about register variables is that they
** make your code smaller and they run faster. It is hard to
** lose with registers. There are three restrictions which you
** should be aware of on register variables, however. First,
** the only types which may be registers are int's, char's,
** and pointers. Second, there may only be three register
** variables per subroutine. Third, you may not take the address
** of a register variable (i.e., you may not say "&i" if i is
** typed as a register variable).
**
*/

```

```

# define      XEQ1          5

struct magic
{
    char      *name;        /* name of symbol */
    int       type;        /* type of symbol, defined in symbol.h */
    int       value;       /* optional value. This is actually
                          * the value if it is type "integer",
                          * a pointer to the value if it is a
                          * string. */
};

struct magic Stuff;

main(argc, argv)
int      argc;
char     *argv[];
{
    register struct magic *r;
    register int          i;
    register int          j;
    int                   timebuf[2];
    int                   status;

    /* Note that in the declarations of argc and argv above, all
     * parameters to any function should be declared, even if they

```

```

    * are of type int (which is the default). */

r = &Stuff)
/* initialize random # generator */
time(timebuf);
srand(timebuf[1]);

/* scan Stuff structure */
for (i = 0; i < XEQ1; i++)
{
    #       ifdef xTRACE
    #       if (tTf(5, 13))
    #           printf("switch on type %d\n", r->reltype);
    #       endif
    switch (r->type)
    {

        case 0:
        case 1:
        case 3:
            /* initialize */
            printf("hi\n");
            break;

        case 2:
            /* end of query */
            printf("bye\n");
            break;

        default:
            /* be sure to print plenty of info on an error;
             * "syserr("bad reltype");" would not have been
             * sufficient */
            syserr("bad type %d", r->type);
    }
}

/* resist the temptation to say "} else {" */
if (i == 5)
{
    i++;
    j = 4;
}
else
    i--;

/* plot the results */
do
{
    i = rand() & 017;
    while (i--)
    {
        printf("*");
    }
    printf("\n");
}

```

```
    } while (j--);  
  
    /* wait for child processes to complete */  
    wait(&status);  
    /* end of run, print termination message and exit */  
    for (i = 0; i < 2; i++)  
        printf("bye ");  
    printf("\n");  
}
```