# A Language Independent Superstructure for Implementing Real-Time Control Systems

Anthony J. Barbera, M. L. Fitzgerald, James S. Albus, and Leonard S. Haynes

National Bureau of Standards, Washington, D.C. 20234

## ABSTRACT

It is the purpose of the system superstructure described in this paper to create an environment that eases the user's development of software. Techniques and software tools are described that help organize a system into a very structured and modular framework that is conducive to interfacing, upgrading, partitioning onto multiple computer systems, and debugging. A system dictionary is described that together with the modular superstructure allows the creation of a highly interactive enviromnent where single programs or any level of aggregation of programs can be executed and where any variable or aggregation of variables can be examined, traced, displayed or modified.

## Introduction

The Industrial Systems Division of the National Bureau of Standards has been working for over a decade on real-time control systems. The primary application of this work is in the real-time sensory-interactive control of robots. References [1] through [3] document this work and describe the application of the underlying structure to the control of an automated manufacturing system being developed at NBS.

This paper presents a broader view of this work. It describes the implementation techniques and developed software tools used to format the large number of computer programs into an organized framework. This system superstructure has been found to aid considerably in the user interactions in all aspects of software design, implementation, debugging and maintenance. The Real-time Control System ( RCS ) presently used for sensory-interactive robot control at NBS has been implemented with this superstructure including all of the described tools.

This paper is partitioned into three sections. Section I describes the high level concepts of the system superstructure. Section II describes implementation techniques and software tools used to impress this superstructure onto a software system. Section III describes some specific examples from RCS to illustrate the use of this superstructure with real-time control software.

## Section I. Concepts of a System Superstructure

Previous experience at NBS in the development of large system software such as real-time control systems has indicated a need for higher level structuring and greater ease of user interaction than provided by most programming systems. Even the application of structured programming techniques [4], while keeping the individual routines in a manageable form, can easily lead to a large complex hierarchical program structure of many hundreds of subroutines [5,6]. The very size of this structure together with the practical problems of recompiling and relinking new or modified routines, editing diagnostics in and out of routines, etc. makes this system awkward to track, comprehend, debug and maintain. Hence the phenomenon that once a large system is finally made relatively bug-free, it is sealed up and no-one dares to venture in and modify the code for fear of creating errors that might be difficult or impossible to track down.

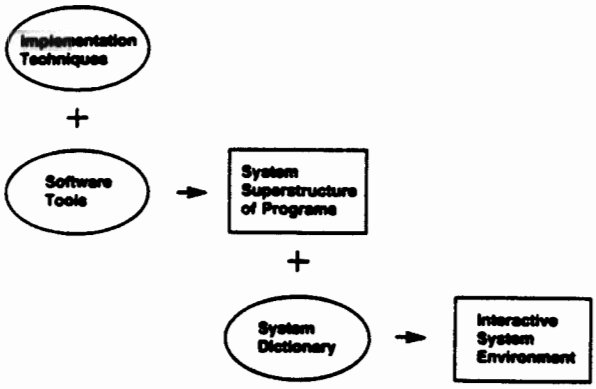
This phenomenon is closely linked to the difficulty of the human programmer to deal with very much information at any given time. Individual programs, thanks to the discipline of structured programming, process only small well-bounded sets of information and are therefore easily understood. Systems made up of hundreds of these well-structured routines, however, represent vast amounts of inter-related information that can quickly exceed human limitations of information management [6]. These limitations relate to the apparent inability of people to manage more than seven pieces of information at any time [7].

This basic problem is addressed here by a rigorous higher level structuring technique or style to format large software systems. This paper presents the concept of a superstructure to organize sets of programs in such a manner as to always allow the user to focus on any level of detail in a complex system and only be required to deal with not more than seven pieces or chunks of information. Therefore, rather than allowing the uncontrolled growth of numbers of routines into large hierarchies, this superstructure, through its implementation techniques and software tools, controls system growth and creates bounded structures of manageable size and provides an organized framework for their interactions.

Program testing is another area that becomes more difficult as system size increases. Testing of individual routines involves considerable overhead in generating test calling routines and test data or I/O routines for user specification of test data. There is also the additional burden of larger amounts of time required for each recompile and relink of the larger sets of routines. To deal with this situation, a system dictionary has been implemented that uses a single set of generic tools to create a highly interactive user interface to all components of the system at any level of detail. The use of this system dictionary in support of additional diagnostic graphic displays and communication mechanisms will be described.

## Section II. Implementation

Figure 1 summarizes the relationship between the component techniques and tools and the creation of the system superstructure and interactive system environment. This section describes in some detail these implementation techniques, software tools and system dictionary and discusses the rationale for their use.




The Described System Consists of a Superstructure for Program Organization and an Interactive Environment for User Access. These Two Features Are a Function of Implementation Techniques, Software Tools and a System Dictionary

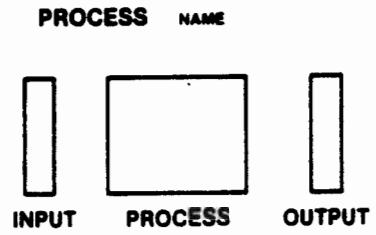
FIGURE 1

### A. Implementation Techniques

Three basic techniques are used to aid the user in structuring a large software system. The goal is to partition the processing into comprehensible units of a standard format and then assemble those units into an organized framework. This superstructure causes the processing to be always identifiable as a collection of simple bounded units that can be examined at any level of detail with never more than a small number of parameters to be considered. The techniques used include the structuring of a well-bounded set of programs into a unit module, a standardized processing sequence format, and a generic higher level clustering of the unit modules to accomplish large complex functions. The following three subsections elaborate on these techniques.

#### 1. Use of well-bounded function as building block

This technique has much in common with the goals of structured programming, namely, the partitioning of a system into small subunits. Each of these subunits has been set up to encompass a well-bounded computation - a simple function that is totally contained within a routine of a few conditional tests and a small amount of computation. The difference here is that these modules are cleanly delineated from each other by the specification of their input and output data interfaces (Figure 2). These modules might call subroutines to complete their calculations but their entire function is totally defined within a small set of these subroutine calls, and are clearly separated from all other modules by a defined input and output data structure.




Functionally Bounded, Named Module
Functionally Bounded Modules are Independently Executable

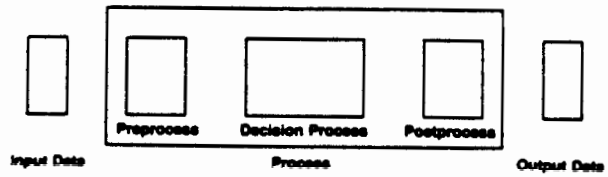
FIGURE 2

If "A" requires data that "B" produces, then "B" executes, generates its output buffer. "A" then executes and uses that data in its execution. "A" does not call "B" midway through its processingin order to pick up any required data. That is, the entire input set required by "A" must be available when "A" executes. If "B" cannot execute independently from "A", then "B" is not a well-defined unit module and must be included as part of "A".

These unit modules become the fundamental building blocks for the whole system. The larger functions of which they are the component units are built, not by assembling these units into a normal hierarchical calling relationship, but rather by executing lists, or sets of these separate well-defined component units. Each unit module can always be executed stand-alone, with its input and output data retrieved from and stored into a common memory area. This unit module is a function of only a small number of variables. Its processing involves only a small number of decisions and/or calculations. As such, it is easily comprehensible.

## 2. Use of a standard processing format

Each of the above unit modules performs some processing to calculate an output data set as a function of its input data set. All processing is on symbolic variables. No numeric constants are permitted. This processing can be partitioned into a standard format of: preprocess, decision-process, and postprocess (Figure 3).




A Standard Processing Format of Preprocess, Decision Process and Postprocess is Used for Each Functional Module

FIGURE 3

The preprocessing is used to condition and reduce the number of variables into a convenient representation for the major decision-processing of the function being calculated within the module.

The decision-processing is the algorithm or procedure that represents the function being calculated. It should clearly identify the relevant parameters and explicitly represent the various test conditions required. The resultant output procedures for each of these tested states are also clearly identified within this section of processing.

The postprocessing is used to cleanup the processing of the function. It performs such tasks as reformatting data into a prescribed output structure, or updating internal variables such as counters, or saving certain values for the next calculation of the function.

The following is a simple example in pseudo-code used to illustrate this format.

Preprocess

```
scaled_value = ( new_value- thresh_val ) * scalar
scaled_diff  =  scaled_value - old_scale_value
```

Decision-Process

```
if  scaled_diff (EQ) test_value then call OUT_1
if  scaled_diff (LT) test_value then call OUT_2
if  scaled_diff (GT) test_value then call OUT_3
```

Postprocess

```
old_scaled_value = scaled_value
```

The function of this unit module is the calculation of output 1, 2, or 3 depending on whether a particular value is equal to, less than or greater than the test value.

The preprocessing is used to create the real variable of interest (scaled_diff) from the input variables new_val, thresh_val, scale_factor, and the state-variable old_scaled_val.

The information is now in a form for a clean specification of the test conditions where the only information of concern is how the value of the scaled_diff compares to that of test_value. Both the test conditions and output calculations are contained in the decision-processing section. It will be noted that the unit module can make subroutine calls to appropriate output procedures. However, the unit module is the largest structural component that calls subroutines. How larger systems are built from these units is discussed in the following subsection.

In the postprocessing section of the module, the scaled_value is saved in old_scaled_value for use in the next calculation of the preprocessing section.

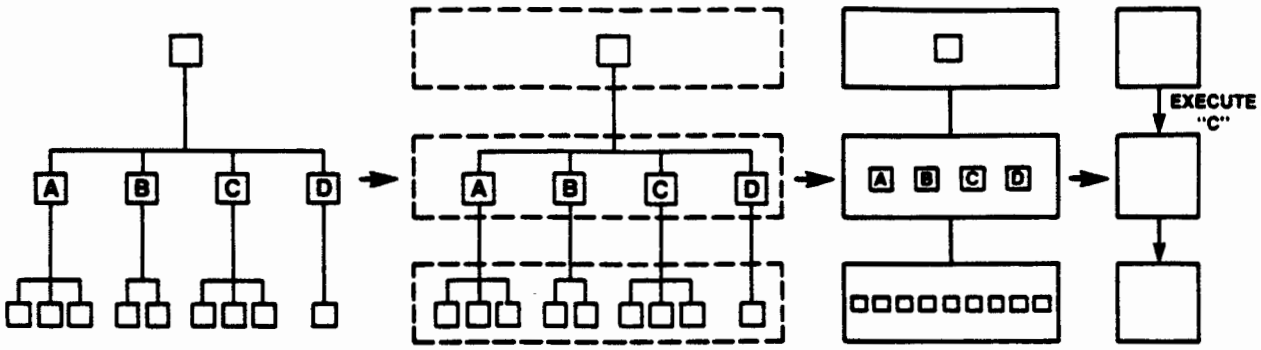
This partitioning of the processing allows for the isolation of the major functional calculations in a clear format in the decision-processing section. Preprocessing evaluates, scales, reduces and transforms the input data into a more appropriate set of variables for this decision-processing. Postprocessing picks up the extraneous, but necessary, additional processing to keep this processing from cluttering or confusing the main functional calculation of the module.

When used as a standard processing format, this technique does much to ease the user's interaction and comprehension of each unit module. This standard processing format can be applied to the larger system components (levels) as well. This will be described in the following section.

## 3. Use of superstructure to organize system

As mentioned above, the unit module is the largest component that issues subroutine calls. It is a very limited, well controlled program hierarchy of a small manageable size performing a well-bounded function. But, it is only one component of a much larger system. To organize unit modules into a larger framework, a superstructure is proposed.

A normal procedure for dealing with large software systems is to develop a hierarchy of calling routines in a top-down structured approach. Each layer in this hierarchy partitions out finer and finer component procedures. The left hand side of Figure 4 illustrates such a program hierarchy where each box represents a module or subroutine. The highest level routine has the choice of calling subroutine A, B, C, or D depending on the values of its inputs. Each of the subroutines A, B, C, and D perform a similar evaluation and make calls to their appropriate subroutines.




**Equivalence of Normal Hierarcial Structured Program Set To Their Format in the Described Structure**

FIGURE 4

This nesting can repeat a large number of times for a large software system, such as a real-time control system. The resultant structure becomes suprisingly awkward to deal with especially if modifications are to be made. Even though this hierarchy consists of well-structured routines designed in a top-down fashion, tested each step along the way, the final structure is complex. This complexity is largely a result the very size of the system.

The proposed superstructure is a technique for the horizontal partitioning of this program hierarchy into a number of levels. These levels are small manageable groupings of a number of unit modules These levels are totally separated from one another and become independent entities. Subroutine calls cannot be made from one level to the next, as in the programming hierarchy. Rather, the name of the subroutine to be called is encoded as part of the output data structure of one level in a manner very similar to the use of stubs as a testing technique during the top-down design. But here, the calling link is never made. All information that passes between levels is in data interface structures of defined input and output buffers for each level. Each level behaves as a separate processing unit of the input, process, output type shown in Figure 2.

The level is a much larger structure than the unit module shown in Figure 2, but its processing is formatted in the same manner (Figure 5). Preprocessing in a level consists in executing a list of preprocessing modules. Their outputs are the necessary set of variables (less than seven) for the major decision-processing of the function of this level. Postprocessing consists in executing lists of postprocessing modules.

In the example program hierarchy illustrated on the left side of Figure 4, the highest level module uses its set of input variables to determine which of the major subroutines, A, B, C, or D, should be executed next. It calls the appropriate one and processing of the function continues with that subroutine evaluating additional input variables to determine which of its subroutines should be called.

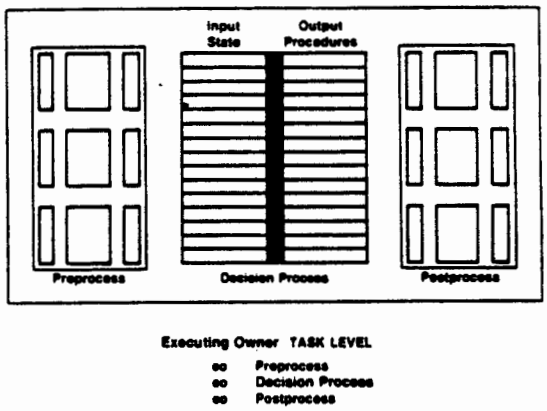



FIGURE 5

The proposed superstructure, however, clusters the subroutines, A, B, C, and D, as unit modules within a single level. This reorganization separates out the major decision processing of each routine from any additional processing. It collects all the variable conditioning into a set of preprocessing modules and all the extraneaous processing into a set of postprocessing modules. Only the major decision-processing and functional output calculations remain in each of the modules A, B, C, or D.

As an example, the highest level module evaluates its set of inputs and and determines that module C should be called to continue the processing. This request or command is encoded as part of this level's output data which becomes the input data to the next lower level. This lower level first

executes its list of preprocessing modules and conditions the appropriate test variables. The request in the input data from the level above to "execute C" causes this level to pick "C" from its pool of possible modules and execute it. This involves evaluating the set of test conditions of module "C" using the variables set up by preprocessing and to decide on the appropriate output procedures. Part of the processing in the selected output procedure is to decide on the appropriate module from the next lower level to be executed and to encode its name in the output data.

To better understand the effect of this structuring on the overall system processing, this process will be examined from a slightly different point of view. The entire functional processing can be described as a very large decision tree of the same basic shape as the programming hierarchy but containing many more branches at each level. The complete evaluation of the function comes by determining a path from the top to the bottom of the tree where each node is a test on an input variable and the path or branch chosen depends on the present value of that variable.

Each module shown on the left side of Figure 4 represents the testing of several variables to determine which path to follow to continue processing. When the high level module chooses subroutine C, it means that it has tested its input variables and based on their values, the solution of the function can be found in that part of the decision tree encompassed under C. Or, conversely, the solution will not be found in the remainder of the tree under A, B, and D.

Subroutine C then evaluates its small set of additional input variables and narrows the solution paths to those found in that part of the tree contained under a particular one of its subroutines.

Viewing the function in this manner, one finds that the levels of the superstructure partition this large decision tree horizontally. The uppermost level evaluates a defined limited set of variables and, in this example, chooses one of four regions (A, B, C, or D) of the decision tree to continue down. It encodes this choice in its output data. This message is part of the input data of the next lower level. It is a token that marks the path in the decision tree at which to pick up and continue the processing. This next lower level has four mutually exclusive starting points (A, B, C, or D) from which to continue down the tree. But within this level, these four modules are totally separate and independent and can be represented as such.

Part of the input data to each level, therefore, specifies a particular module which is a small subset of the decision tree to execute. The remainder of the input data is the set of variables or their precursors to be tested within this small region of the overall decision tree. The path through this region for the present state is determined and this point in the overall tree is encoded in a data structure to be sent to the next lower level which uses this information as the starting point in its region of the decision tree to continue processing.

This processing continues down through all of the levels until the entire tree has been evaluated and the function determined as one complete path.

The major benefit of this type of structuring is in always keeping the total processing that the human sees as small contained units. In each level, there are a number of small totally independent sets of processing that do a very small tree search using small numbers (seven or less) of variables. The resultant output is the starting point for the continued searching at the next lower level. In this manner, no matter how large or complex the function being evaluated, the user never has to see or deal with more than a very limited tree search in an isolated module. It is the interaction of all of the levels that creates the large complex tree search, but the user's view of the system does not reflect this complexity as it would if the system had been programmed in the typical large hierarchical programming structure that would mimic the complex decision tree to be evaluated.

The horizontal breaking of the system by the levels makes it appear as if the modules are still in the small simple test format with "stubs" being used for all the rest of the system that is not present. But, unlike the normal programming hierarchy where the "stubs" are replaced by the real procedure calls, here, they always remain and are part of the input and output data at each level.

Further benefits gained from this superstructure of levels include the ease of incorporating new branches within the overall decision tree. The rigid partitioning into levels, groups modules concerned with the same general level of the tree structure. This aids considerably in identifying which level is appropriate for the incorporation of additional modules.

This structure can repeat as is appropriate for the complexity of the function. That is, the more complex the system, the more levels that will be required. But, each level is a totally bounded entity, and all levels are similar in the amount of complexity of their processing, which is always kept small. In addition, the levels are similar in the format of the processing within them. This is useful in identifying the proper places in the code for modification. Since each level contains a small set of independent modules, grouped into one of three categories, namely preprocessing, decision-processing or postprocessing, once the user knows what changes need to be made, it can easily be determined where the changes should occur. The superstructure of the system embeds a great deal of information about the processing of the function, and allows the user to consider only the change to be made, rather than having to figure out the linkages with the rest of the system or the relationship of the module with the

other modules. In this way, the superstructure itself can limit the amount of documentation required. Since the relationships of the modules is defined by their position within the structure, documentation need only be concerned with the function performed by the module.

In addition, since each of the modules within a level is an independent function with well defined input and output data sets, the superstructure has created a system in which the various functional modules may be treated as black boxes allowing their interchangeablilty and modification without effecting the rest of the system. As long as two components meet their interface buffer format specification, they may be successfully integrated.

The horizontal partitioning of the system function also lends itself to implementation on a multiprocessor system. Since the linkages between the levels is only through their input and output data sets and not through a direct process to process protocol or calling structure, each of the levels can be implemented to execute on a separate computers (or even different computers). In addition, they may even be written using different programming languages under different operating systems. As long as there is some mechanism for supplying each level with its input and output data sets, through common memory buffers for example, levels may bedeveloped and tested in isolation, and integrated at some later time.

## B. Implementation Tools

Several software tools have been designed and implemented to support the application of the above structuring techniques and to create a highly interactive system development environment. These tools include language extensions that set up a clustering of members under named owners to aid with the superstructure implementation, a system dictionary that is the basis for an interactive user interface, and a common memory for the interface data buffers that additionally simplfies communications and real-time trace diagnostics. The following three subsections elaborate on these tools.

### 1. Owner-Member List Structure

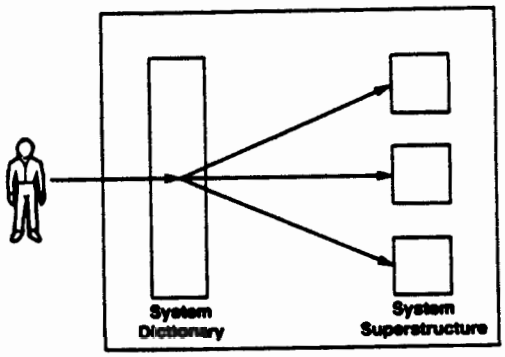
As described above, in this superstructure, the unit module is the largest entity containing a normal programming hierarchy of calling routines. For the larger organizational structures (that is, the levels) aggregation of these unit modules into executable lists is done through a set of language extensions. The extensions are higher level programming structures for establishing owner-member relationships to cause a grouping of a number of related items under a named owner.

There are three basic owner-member extensions that have been implemented. One clusters variables of any type into named buffers (much like records) to organize the input and output data sets of modules or levels. The second clusters unit modules into a named execution list to organize the preprocessing unit modules into a single structure and the postprocessing modules into a single executable structure for each level. Execution of that owner executes all the member unit modules in the sequence specified. The third extension clusters the major decision-processing unit modules of a level under a named owner and causes execution of just one of its members based on the input data from the next higher level.

Thus, these three extensions organize a level into the overall structure of input data, process, output data, and format the processing into a preprocessing block, a decision-processing block and a postprocessing block. Explicit examples of these extensions are given in Section III.

### 2. On-line System Dictionary

The system dictionary is the major structure that has enhanced the interactive nature of the user interface. It is essentially a data-base-like structure maintained in memory for high speed access (Figure 6). The compilers and above mentioned language extension tools have been modified so that as programs are compiled, variables are declared and member-owner relationships established, all of the named items are entered into this dictionary along with information that gives the memory address of each routine, each variable, each owner as well as other data concerning type and interrelationship between entries.




The System Dictionary Provides the User Interface Mechanism For Examining, Modifying and Executing Any Variable or Program in the System in a Totally Interactive Mode

FIGURE 6

This then becomes the major component of a set of powerful interactive tools. Individual routines at any level can be executed stand-alone at any time. Any variable can be interrogated or changed at any time. Any cluster of programs can be executed as well as any cluster of data specified by the member-owner extensions can be interrogated and modified.

Thus, no matter how large or complex the function being calculated, the user always has immediate access to any portion of the system at any level of detail. The system becomes totally interactive to the user without the burden of writing any additional software other than the one-time creation of the system dictionary mechanism and its associated generic tools.

## 3. Common Memory - Communications and Diagnostics

The use of the above superstructure along with the well-bounded function module has created a system of separate entities (levels and modules) that interact only through their input and output data interfaces. To connect any two components, the output data of one must become the input data of the other. This implies that there must be some method to transfer data between these components, especially if one module executes on one computer and the other on a second computer. A communications mechanism has been implemented that uses a common memory area. A copy of the output data is moved to a duplicate buffer area in common memory and from there moved into the input data buffer of the other module. Figure 7 shows a hardware configuration of several microprocessors and a common memory connected on a common bus. A communications process transfers data between modules or levels on the computers through the common memory.

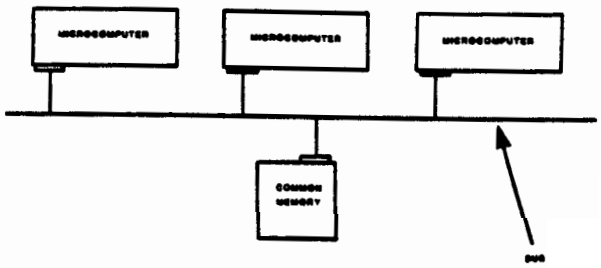
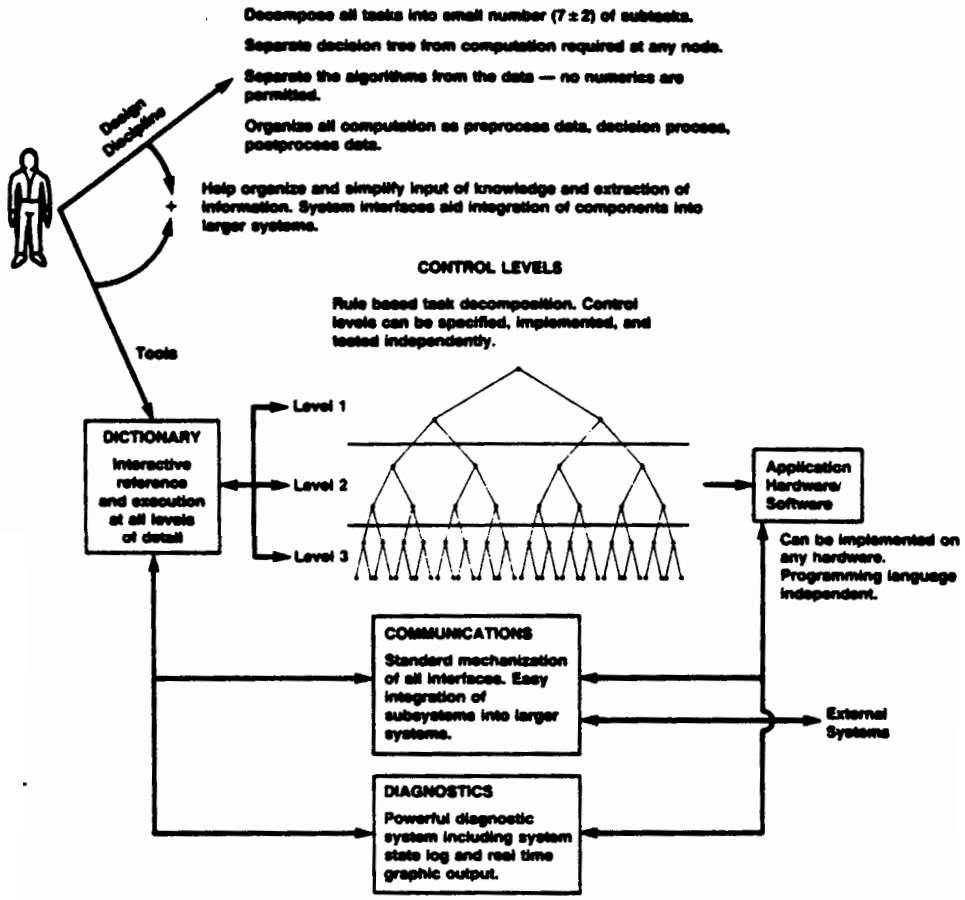



FIGURE 7




RCS Language-Independent Superstructure for Implementing Real-time Control Systems

FIGURE 8

The use of a common memory buffer area as a double buffering mechanism for a communications system offers a number of benefits. Interfacing between separate systems becomes straightforward since the data buffers define natural interfaces. The buffering of these data in common memory becomes a convenient structural mechanism for communications since it eliminates any need for timing between sending and receiving modules. Data moves from the first module to common memory whenever that module is ready to send it, and moves from common memory to the second module whenever it is ready to receive it totally asynchronous with the first modules timing requirements.

The existence of the duplicate copy in common memory makes available to an additional process, such as a diagnostic process, the present values of all of the critical variables in the system. If the communications mechanism is executed as a synchronous process, then there is a period out of every communication cycle (equivalent to a calculation cycle or control cycle) when the data in common memory is stable and not being written over. The diagnostic process can read this data during this time period every interval and provide a real-time trace that can be placed on mass storage for later retrieval or displayed in real-time on a terminal or graphics system.

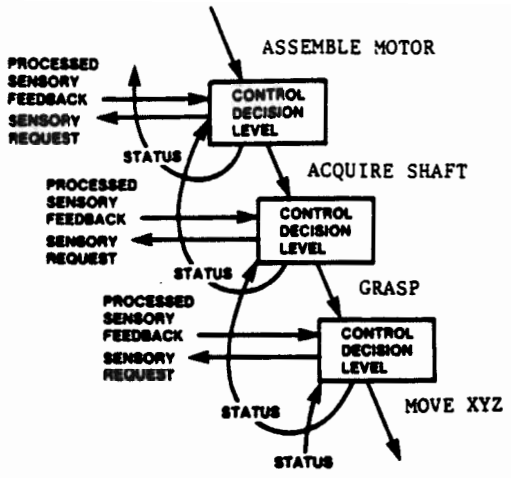
Figure 8 is a summary illustration of the techniques and tools described above. The next section will provide an example using a real-time control system.

## Section III. Examples from RCS

The described superstructure has been used to organize the complex information processing for a robot real-time sensory-interactive control system. A robot task, such as "Assemble Motor", may be commanded to the system, and the resulting outputs which eventually are the drive signals to the actuators of the robots are generated based on sensory data that measure the state of the environment.

To manage the complexity of this function a Real-time Control System ( RCS ) [8] has been implemented as a number of generic control levels (Figure 9) using the previously described techniques and tools. The structure of the RCS resembles the figure on the far right hand size of Figure 4. Each of the square boxes is an individual level. Each level decomposes the task into simpler subtasks. Each level has a narrowly defined control capability that results in clear identification of the type of sensory processing required at each level, the type of status feedback necessary and the kind of output commands that should be generated. Interaction between levels occurs by encoding the request for a particular action into the output data set of an upper level to be interpreted as input to the lower level when it executes.

An individual control level has its function broken down into preprocessing, decision-processing and postprocessing. The preprocessing




**Hierarchical Control Levels Linked Together to Form a Complex Control System**

FIGURE 9

routines transform input commands, sensory data and status feedback into a form that will be convenient for the decision-processing routines. Each of these preprocessing routines itself has a small set of input variables, performs a well-defined function and generates a limited output.

One of the decision making routines in the level is selected for execution by the output command from the upper level. One of the functions of each decision making routine for a level is to generate the calls to the appropriate output procedures that will encode the command to the next level. That is, it does a partial decomposition of its input task command into the appropriate subtask command for the next level. The command sent is dependent on the input data to this level as well as processed feedback data that represents the state of the system and its environment (Figure 10).

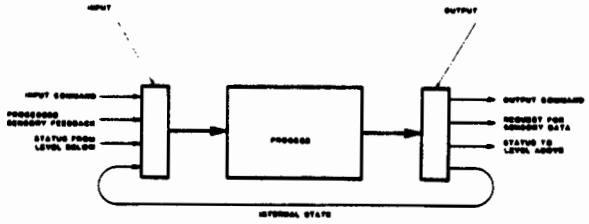


FIGURE 10

Postprocessing routines transform the results from the output procedures into a form convenient for the next level to use, and maintain internal variables such as counters and state variables.

As an example, one level in the RCS coordinates primitive motions of the robot and its gripper. Inputs to this level consist of commands like "RELEASE", "APPROACH" or "GRASP". The functions performed by these commands are encoded in the decision-processing routines in this level. These were represented by the modules A, B, C, and D in the earlier system description. A GRASP command, which is equivalent to "execute C" in the Section II discussion, requires that the gripper close without moving the object. Sensory data to this level are from touch sensors that indicate if contact with an object has been made, force sensors that measure how much force is being applied by the gripper and position sensors that indicate the size of the gripper opening.

The variable owner (VO) that defines the input data set to this level is:

```
    VO   level-input-data

    iv      input-command
3 3 2:a     T#1   ( data from touch sensor 1)
3 3 2:a     T#2   ( data from touch sensor 2)
    iv      grip-position
    iv      grip-force
```

T#1 and T#2 are each 3 by 3 two dimensional arrays (2:a) that contain the data from the nine touch sensors in each finger of the gripper. The integer variable (iv) input-command encodes the command (inthis example, GRASP) from the next higher level.

The preprocessing functions for this level include a routine to threshhold the touch sensor data to determine if contact is made and another that determines the actual gripper opening from the position sensor data. The following example processing module sets the values for the variables "touch-sensor-1" and "touch-sensor-2" to CONTACT or NOCONTACT depending on whether the raw data values from the sensors are greater than a specified threshhold value.

```
routine DETERMINE-CONTACT
   i = 0
   touch-sensor#1 = NOCONTACT
   touch-sensor#2 = NOCONTACT
   repeat
      if T#1[i] (GT) threshhold
         then touch-sensor#1 = CONTACT
      endif
      if T#2[i] (GT) threshhold
         then touch-sensor#2 = CONTACT
      endif
      i = i (+) 1
   until  i (EQ) 9  end-repeat
end-routine
```

This routine has a small set of input values and calculates a well-defined clearly bounded function. It reduces the information contained in 18 words of data into the two pieces of information required to carry out part of the major decision-processing for this level, namely, whether either side of the gripper has contacted the object. This routine is grouped along with the other preprocessing routines for this level in the executing owner (EO) called PREPROCESS-MONITOR. The programs within this owner are loosely coupled and can execute entirely stand alone. They are clustered into the owner to enforce an organization onto their execution. This owner is executed once per control cycle and its execution causes the execution of each of its members in sequence.

```
    EO   PREPROCESS-MONITOR

    p    SCALE-POSITION-DATA
    p    DETERMINE-CONTACT
    p    SET-NEW-COMMAND-FLAG
   ...
```

The interactive user interface by way of the system dictionary allows the entire PREPROCESS-MONITOR to be executed, or each member of that owner to be executed individually. Each of the related variables may be interrogated individually or as clusters through their named owners through any part of the execution.

The decision-processing function of this level uses the processed feedback data to decide which of the possible subcommands of its input command should be issued next. The selected decision-processing module evaluates the main task decomposition of the level, but it is still only required to deal with a small set of variables. The preprocessng has been used to reduce a larger number of simpler input variables into the smaller set of higher-level variables that are the relevant decision variables for determining the next step in the task decomposition at this level.

The output procedures required by the GRASP command may be determined using the algorithm shown in Figure 11. The function of this command is to have the gripper grasp an object without moving the object, regardless of the initial position of the object with respect to the gripper fingers. These output procedures encode the selection of the appropriate branch of the decision tree to be executed by the next lower level as the output command to be sent to that level.

Figure11 has illustrated the decision-processing module denoted by the GRASP command at this control level in the format of an "if-then-else" structure. Close inspection of this figure shows that it is difficult to clearly differentiate the decision paths and to be positive of what conditions relate to what actions when they are expressed in this nested "if-then-else" structure. This is another instance of the problem of dealing with information processing in large hierarchical structures. In order to simplify the statement of the conditional tests of the various input states, a state-table format for programming is recommended and used within RCS. Each line of a state table can be considered a production rule of the type

```
    IF "this input condition "
        THEN "generate this output"  ENDIF
```

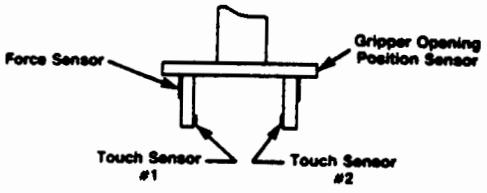
This makes each tested input state and its related output procedures a bounded unit module. The evaluation of an input command like GRASP becomes the process of stepping through the list of test input conditions until one of the lines matches the present state of the input variables. Its output procedures are then executed. In this way, every possible test condition is a stand-alone isolated statement of a possible state the system can be in and the resultant output.

Figure 12 shows the equivalent state table representation of the algorithm for the GRASP command. The five left hand columns are the set of values representing the current input test values. The right hand columns represent the corresponding output procedures that should be executed if the conditions in the input state are met. Every cycle of a control level, the current status of these variables are compared with preprogrammed sets of possible input conditions contained in the lines of the table.

```
IF command = GRASP
    THEN IF touch sensor # 1 = CONTACT
      THEN IF touch sensor # 2 = CONTACT
        THEN IF gripper opening = object size
          THEN IF grip force = specified
            THEN OUT = PAUSE STAT = GRASP-F, OBJ-IN-HAND
            ELSE OUT = CLOSE .1 MM STAT = GRASPEX, FORCE
            ENDIF
          ELSE IF gripper = closed
            THEN OUT = PAUSE STAT = GRASP-F, NO-OBJ-IN-HAND
            ELSE OUT = PAUSE STAT = GRASP-F, NOT-OBJ,
            ENDIF
          ENDIF
          ELSE OUT = CLOSE .1 MM, MOVE – .05 MM, STAT = GRASPEX, CONTACT
          ENDIF
      ELSE IF touch sensor # 2 = CONTACT
        THEN OUT = CLOSE .1 MM, MOVE + .05 MM, STAT = GRASPEX, CONTACT
        ELSE OUT = CLOSE .1 MM,    STAT = GRASPEX, NO-CONTACT
        ENDIF
      ENDIF
ELSE OTHER COMMANDS
```

**Example of program to close robot gripper without moving object to be grasped.**

FIGURE 11




| input command | touch sensor #1 | touch sensor #2 | gripper opening | grip force | OUT | STAT |
|---|---|---|---|---|---|---|
| GRASP | NO CONTACT | NO CONTACT | X | X | CLOSE .1 MM | GRASP-EX |
| GRASP | NO CONTACT | CONTACT | X | X | CLOSE .1 MM MOVE + .05 MM | GRASP-EX CONTACT |
| GRASP | CONTACT | NO CONTACT | X | X | CLOSE .1 MM MOVE – .05 M | GRASP-EX CONTACT |
| GRASP | CONTACT | CONTACT | object size | specified | CLOSE .1 MM | GRASP-EX FORCE |
| GRASP | CONTACT | CONTACT | object size | specified | PAUSE | GRASP-F OBJ-IN-HAND |
| GRASP | CONTACT | CONTACT | obj. size = closed | X | PAUSE | GRASP-F NOT-OBJ |
| GRASP | CONTACT | CONTACT | closed | X | PAUSE | GRASP-F NOT-OBJ-IN-HAND |
| | | | | | | |
| | | | | | | |

**Example of a State Table Which Will Control Robot Gripper to Close Without Moving Object to be Grasped.**

FIGURE 12

This technique makes visible all the relevant conditions of the high level variables and their associated output procedures. It makes it simpler for a user to return to these programs and determine their function since all of the conditions required for a set of outputs are explicitly stated. Additional situations may be added at a later time by adding lines to the state table without concern for altering the affect of the previously entered test conditions.

All the state tables (decision-processing modules) for a control level are clustered under a state-table owner. For each level, the member state-tables are the decision-processing modules required to decompose the input commands they represent into the next subcommand required in order to execute the task. For this level, the following state-table owner (SO) is defined.

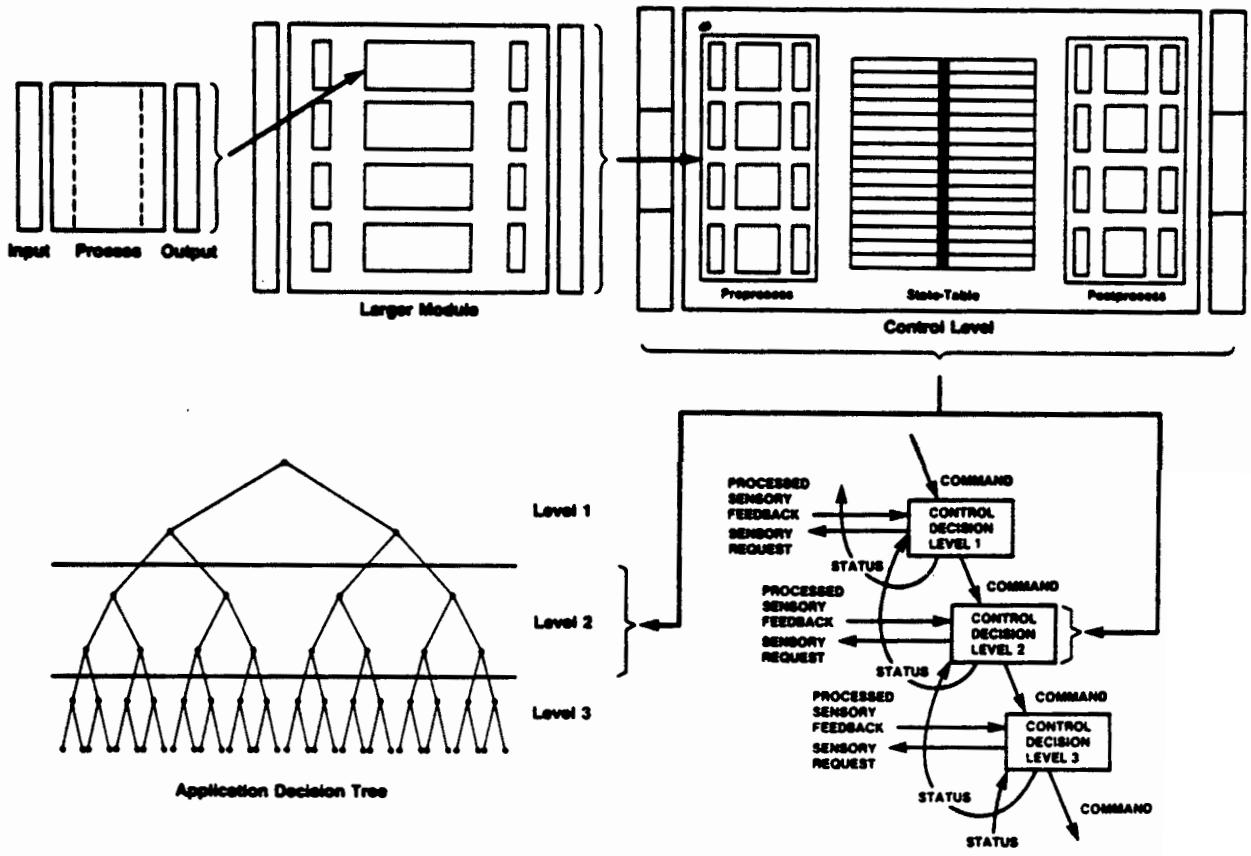
```
SO   LEVEL-STATE-TABLE

 s    RELEASE
 s    APPROACH
 s    GRASP
 s    GOTO
 s    GOTHRU
...
```

Like members in an executing owner, each state table (s) may be executed individually and its outputs inspected at any level of detail. This is again because of the pointers maintained by the system dictionary.

All the postprocessing routines required to complete the calculations for this level are grouped in an executing owner named POST-PROCESS-MONITOR. This owner provides a structure which groups all of the routines that are required in the level to update data formats, maintain internal variables, increment counters etc. It has the same properties as the PREPROCESS-MONITOR in that the whole group of programs may be executed as a whole, or each of its members may be executed alone in order to determine if it is functioning correctly.

Thus, each complete control level performs a partial decomposition of the total complex task for this robot control system. The overall control system is a set of control levels all organized as a superstructure system (Figure 13). All processing within the level is done within modules that represent well-defined clearly bounded functions that generate outputs based on small input data sets. All routines within a level




Hierarchical System Implementation

FIGURE 13

perform their function as a sequence of preprocessing, decision-processing and postprocessing. The level itself has a small well defined input data set and executes its task decomposition function through the use of preprocessing, decision-processing ( in state-table format), and postprocessing program aggregates.

Since the entire system is made up of a number of these similarly structured levels, the user has a reference point to begin any system interrogation; whether to add to or modify the system, or to track down routines that do not seem to do their expected function.

## Conclusion

The described superstructure imposes a rigorous programming style. It lays out a prescribed method of modularizing code, specifying input and output data buffers and restructuring the processing of each module or level into preprocessing, decision-processing and post-processing. Implementation tools have been described that help enforce this format and have been successfully used in the RCS. This rigorous formal style has provided the very large benefit that different people on a programming team, as well as the original programmer, can easily read, modify and debug each other's code. It is simple to identify which particular level contains the relevant processing and within that level whether it is contained in the preprocessing, decision-processing or post-processing section. A quick search down the list in that section isolates the module of interest with its bounding set of input and output data.

Due to the highly interactive environment provided by the system dictionary, the user has been able to very quickly modify any code and test it stand-alone or execute any portion of the system to verify its operation. Data can be interactively examined and modified. The interactiveness has encouraged a much higher degree of testing to verify program correctness because of the ease with which it can be done. The user does not have the burden of writing test calling routines or editing in diagnostic print statements.

The use of this superstructure and the interactive programming environment have been directed at reducing the apparent complexity of large software systems and greatly speeding up the responsiveness of the system so that the user can not only manage all of the information processing, but can also quickly interact with it.

## Acknowledgements

Development of the NBS Real-Time Control System is partially supported by funding from the Navy Manufacturing Technology Program.



## References

1. Barbera, A.J., Fitzgerald, M.L. and Albus, J.S., "Concepts for a Real-Time Sensory-Interactive Control System Architecture," Proceedings of the 14th Southeastern Symposium on System Theory, April 1982.

2. Albus, J.S., McLean, C.R., Barbera, A.J., and Fitzgerald, M.L., "Hierarchical Control for Robots in an Automated Factory", Thirteenth ISIR/Robots 7 Symposium Proceedings, April 1983.

3. Albus, J.S., Barbera, A.J., and Nagel, R.N., "Theory and Practice of Hierarchical Control," Twenty-third IEEE Computer Society International Conference, September 1981.

4. Kernighan, B.W. and Plauger, P.J., "The Elements of Programming Style," McGraw-Hill Publishing Company, New York, 1974.

5. Schneider, G.M., Weingart, S.W., and Perlman, D.M., "An Introduction to Programming and Problem Solving with Pascal," John Wiley & Sons, New York, 1978.

6. Wulf, W.A., "Next Generation of Programming Languages", 10th Anniversary Symposium at the Computer Science Department, Carnegie-Mellon University, 1977.

7. Miller, G.A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", Psychology Review, American Psychology Association, Inc., Vol 63, No. 2, March 1956.

8. Barbera, A.J., Fitzgerald, M.L., Albus, J.S., and Haynes, L.S., "RCS: The NBS Real-Time Control System", Proceedings of Robots/VIII Conference, Detroit, June 1984.