27
7/13/76
250 215

UCID- 17318

Lawrence Livermore Laboratory

METHODOLOGY AND PLANNING FOR A MICROPROCESSOR-ORIENTED
REAL TIME CONTROLLER DESIGN AUTOMATION SYSTEM

M. N. Matelan
R. J. Smith, II

November 4, 1976

This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the laboratory.

Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ABSTRACT

A methodology for reducing the complexity of designing dedicated real time control systems is developed. It is shown that three areas are amenable to automation: 1) the selection and configuration of hardware; 2) the production of software; and 3) the adaptation of a monitor to maintain real time integrity of the entire system.

The concept of hardware binding is introduced, and it is shown that delaying the point in the design cycle where hardware is functionally bound allows a new approach to machine independence. Concepts which allow expression of repetitive control situations are described, and a realization-independent language (CSDL), based on these concepts, is defined. Methods for automatically selecting a time-wise correct monitor are classified, and techniques for specifying the realization capabilities of digital processors are discussed. These concepts and techniques are brought together in a design automation system for the production of a complete controller design from a behavioral description. An example description is traced through the CSD System, producing a software listing and hardware configuration document for an actual microprocessor (the Intel 8080).

This research provides a structured description of the control system design process, allowing a unified perspective in the realization of controllers for applications previously considered to be unrelated. The concepts developed define a new direction in the production of real time control systems.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research and Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

TABLE OF CONTENTS

	<u>Page</u>
1. INTRODUCTION	1
2. THE BASIS FOR A DESIGN AUTOMATION SYSTEM	4
3. CSDL: CONTROL SYSTEM DESIGN LANGUAGE	13
4. A REALIZATION LIBRARY	24
5. TIMING ANALYSIS	31
6. A CSD SYSTEM	46
7. APPLICATION AREAS	60
REFERENCES	66
APPENDIX A. A CSD SYSTEM	68
APPENDIX B. TIMING ANALYZER	69
APPENDIX C. FORMAL SYNTAX OF CSDL	70
APPENDIX D. A CSDL EXAMPLE	77
APPENDIX E. A CSDL REALIZATION LIBRARY	80
APPENDIX F. INTERMEDIATE FORM OF THE EXAMPLE	90
APPENDIX G. CONTROL CODE TO REALIZE THE EXAMPLE	93
APPENDIX H. HARDWARE TO REALIZE THE EXAMPLE	98
APPENDIX I. DISK CONTROLLER EXAMPLE	102

1. INTRODUCTION

The use of digital computers as real time monitor and control devices is an increasingly important aspect of computer technology. Further, many computer system functions (both hardware and software), not traditionally associated with the engineering discipline of process control may be cast in terms of controlling: correctly responding in a time-critical manner to a set of unpredictable but well-defined conditions. The design of systems (composed of both hardware and software elements) capable of exercising control in a real time environment has proven to be a very complex and expensive undertaking. It is the purpose of this design study to develop concepts which reduce that complexity and methods which allow automated production of real time control systems. Much of the work was made possible by the rapid progress in microprocessor technology, although the results are general and apply to all classes of digital computer.

1.1 THE IMPACT OF MICROPROCESSORS

The term microprocessor has not yet been defined in a way acceptable to most of those who use it. There does, however, seem to be a consensus concerning the eventual benefits that microprocessors will bring to nearly every facet of computing. Most definitions of the term included a combination of packaging and processing properties. The most common notion is probably that of a general purpose central processing unit on a single integrated circuit "chip." However, the rapidly changing nature of digital hardware suggests that "microprocessor" be defined in terms which relate the economics of hardware and software, and for this reason it is more stable. Therefore, the term microprocessor will be used in this work to connote general purpose computing hardware so inexpensive that hardware costs and fixed architectures are no longer the primary design constraints. Instead, hardware becomes a design element bending to meet those requirements best suited to a hardware realization.

Using microprocessor technology as a base, it is possible to automate much of controller design, reducing complexity (and thus the cost) of controller production.

1.2 REDUCING THE COMPLEXITY OF REAL TIME SYSTEM DESIGN

The synthesis of digital systems may be viewed as requiring two general tasks:

- 1) the selection and configuration of hardware, and
- 2) the production of software.

The order in which these tasks are typically accomplished reflects the traditional hardware-then-software approach taken when producing a system design. The availability of microprocessors, and the realization that real time control problems are especially suited to them, allows a

new approach to be considered. (The term system, as used here, is meant to include both hardware and software constituents needed to effect control in accordance with requirements of an application.) The approach has as its goal an overall reduction in the complexity associated with producing a dedicated real time control system. Design complexity may be reduced in at least two ways, both of which impact the process of expressing a problem.

First, the removal of all processing hardware considerations from the description of an application would reduce overall complexity. Limiting hardware considerations to those found in the controlled environment forces attention to the problem definition and away from its realization (the controller). Second, complete elimination of the need to explicitly define the concurrent timing requirements of the several control functions would further reduce design complexity. Obviation of multiprogramming considerations allows the designer to partition the problem into functional sub-problems, as if there were available a separate processor for each function. Both of these approaches will be examined.

1.3 HARDWARE BINDING

It has often been observed that programming complexity is reduced by increased levels of machine independence. However, true machine independence is difficult to achieve. One of the main sources of this difficulty is the traditional economic necessity of viewing hardware as fixed and software as changeable. In system design, software production usually begins after the hardware has been selected and at least partially configured.

The reality of plastic hardware suggests that it is no longer necessary or appropriate to fix the hardware configuration at the beginning of the design process. Selection of hardware components may be done at various points in an evolving design. A powerful software concept used to describe the fixing or associating of a symbolic reference to a specific programmatic entity is that of information binding. That point in time (relative to some time reference frame) when a symbol is bound to an entity is often called the binding time of the symbol. No corresponding term which denotes the concept of associating various application requisites to particular hardware elements is in common use. We propose that the phrase hardware binding be used to convey this concept. Thus, the points in the design cycle when various required hardware components are firmly assigned to realize specific functions associated with the application, can be referred to as hardware binding times.

One way to achieve machine independence then, is to delay all hardware binding until after problem definition time (analogous to the program creation time of information binding). Delaying hardware binding until after the problem has been described by the designer means that the application description can be completely machine independent and transportable. Indeed, it will be a description of the problem only, with the actual hardware configuration to be used being deferred to a selection and assignment process later in the design cycle.

1.4 REDUCING TIME-RELATED COMPLEXITY

Much of the power inherent in the use of digital computers as controllers derives from their ability to monitor and respond to more than one application parameter. A single processor involved in controlling more than a single function is said to be multiprogrammed. Multiprogrammed systems require a process to allocate system resources to subfunctions in a way that satisfies some predefined, time-related criterion relative to all subfunctions. The allocation process is usually called a scheduler. In the case of real time control, the scheduler criterion is the strict adherence to the hard deadlines and repetition rates associated with the control tasks.

The necessity of multiprogramming in control programs adds greatly to the difficulty encountered in their production. Removal of the necessity for explicitly programming or selecting a scheduler would significantly reduce the complexity associated with securing a control process from a problem definition. This aspect of producing control software is especially important in connection with microprocessors. It is not likely that they can economically support the "universal" schedulers sometimes supplied with more expensive hardware.

1.5 AUTOMATING ASPECTS OF SYSTEM DESIGN

The research described herein combines the notions of delayed hardware binding time and removal of explicit scheduling considerations to significantly reduce the effort required to develop a control system given a well-defined problem. The power of the combination greatly exceeds the capabilities of methods incorporating only one of these elements. The approach includes three aspects of system design:

- 1) Problem Specification,
- 2) Selection of Hardware, and
- 3) Production of Software.

It is the goal of this research to simplify the first aspect and completely automate the remaining two.

The machine independence afforded by delaying the selection and configuration of hardware until after problem description, coupled with the removal of scheduling considerations means that the designer need only specify the behavior of the proposed control system. This means the designer must express only what the system is to do (i.e., behavioral description) rather than how the final system constituents are to do it (i.e., implementation prescription). Descriptive tools of this sort have been termed very high-level languages.

Once the behavior of the needed controller has been described, a system capable of analyzing the description would be invoked to produce both a complete control program and a configuration specification for the hardware that is to execute the program. The production of software in this way has been called automatic program writing. The selection and configuration of hardware will be termed automatic hardware binding.

2. THE BASIS FOR A DESIGN AUTOMATION SYSTEM

Before a system can be developed to automate controller design, a good understanding of the nature of control is necessary. Several concepts which form a basis for such an understanding (and the system design as well) are discussed below.

Real time applications consist of a group of contingencies which require responses, and a ranking of the group relating their relative criticality. Responses made to contingencies are called tasks. Contingencies sense changes in the controlled (external) environment which are significant in the context of the defining application. Such changes may become known by the controller in one of two ways: determination that a condition has been fulfilled, or an event has occurred. Likewise, the controller (by activating a task) may respond in only one of two ways: causing a condition to exist, or initiating an event. The first way provides information about the external environment that will be referred to as a condition (represented by C), while the second provides information called an event (represented by E).

A better understanding of the general concept of "contingency" is needed if we are to examine its two constituents, condition and event. A contingency is a collection of one or more parameters (p) of the external environment, each of which is defined by three numbers: a current value, a distinguished value, and a bound.

Those parameters which together form a particular contingency descriptor are collectively referred to as a parametric group (P). Each contingency (and as we shall see, each task) is defined by a unique parametric group. The external environment may be viewed as consisting entirely of the parametric groups needed to specify an application.

Parameters are the fundamental units of communication and control between the physical environment and the abstract environment (the control system). At any resolvable point in time, each parameter has a value which reflects its state at that time. A contingency is said to exist for every instant of physical time that the current value of each of its parameters is equal to the corresponding distinguished value. Therefore the distinguished value of a parameter represents some significant distinguished state in the controlled environment which, in concert with other parameters, defines some distinguished occurrence we have called a contingency. Whenever one or more current values of a parametric group do not equal their distinguished values the contingency does not exist. The life span of a single occurrence of a contingency is that collection of contiguous instants of physical time during which all of its parameter values equal their distinguished values (that is, the time span in which the contingency exists). Each instance of existence will have its own life span value. The shortest possible life span in any instantial existence of a contingency (that is, the minimum possible life span) is called the bound of the contingency (symbolically written θ).

There exists an interesting duality between contingency and task which allows the concepts and nomenclature developed to discuss contingency to be applied to the notion of task. Contingencies define circumstances in the external environment which require response. Inversely, tasks define responses which must be made in answer to contingencies in order to maintain the external environment in some state that is valid in the context of the application. Therefore it is convenient to define a task to be a collection of one or more parameters (p) of the external environment each of which is defined by three numbers: a current value, a distinguished value and a bound. Such a collection is said to be the parametric group (P) of the task. At any resolvable point in time, each parameter (sensed by, and controlling some part of the external environment) has a current value which reflects the activity of the parameter. If at any time, all values of parameters which combine to define a task are equal to their distinguished value, that task is said to be active. It is otherwise inactive (that is, not selected by an existent contingency). The least amount of physical time in which a task must remain active in order to effect its intended change in the external environment is the bound (β) of the task. To avoid confusion when discussing features which contingencies and tasks have in common, subscripts will be used in conjunction with each symbol. Contingencies are assigned the subscript one and tasks are assigned the subscript two. This reflects the nature of their relationship: the existence of a contingency causes its corresponding task to become active.

2.1 CONTINGENCY MODE

The two aspects of contingency, condition and event, may be compared and contrasted now that the general concept of contingency has been developed. Since condition and event are but two ways of describing a contingency, it is not surprising that they have much in common. Both are defined by parametric groups. Both exist only when all current values equal corresponding distinguished values. Finally, both have a minimum life span figure (the bound). It is the duration and nature of the bound which determines whether a parametric group is an event or a condition.

Since condition and event describe contingencies from the controller's internal perspective, it is useful to compare them to corresponding concepts related to the external, controlled environment. Conditions are of relatively long duration (that is, they possess large bounds). Each parameter (usually more than one) of a condition definition must have this large bound property. It is useful to compare the concept of large bound to the concept of level found in circuit theory. Assume that a contingency parametric group defines the outputs of circuits sensed by the control system. That is, each parameter of the parametric group is the output of a circuit which reflects some physical attribute of the external (controlled) environment. (It is not necessary that the model be tied to any particular method of sensing the environment, but the goal of modeling computer-based control systems makes analogies to circuit theory especially appropriate.) If these outputs are levels

(continuously defined) rather than pulses (intermittently defined), then the sensed external environment may be expressed in terms of a Moore circuit model. Portions of the environment which present this type of message to the controller are candidates for representation by conditional contingencies. Likewise, tasks whose parametric groups require levels rather than pulses to effect control are considered to be condition-like in nature.

Events are of relatively short duration (that is, they have small bounds). A single parameter possessed of a small bound is sufficient to define its parametric group as an event. As with condition, event too will be related to a concept found in circuit theory: the pulse. Assume that the contingency parametric group discussed above contains one or more parameters which may be meaningfully sensed only as a pulse. Pulse outputs may be expressed in terms of the Mealy circuit model. The significant feature of the Mealy model in this context is the short term validity of pulsed outputs. Portions of the environment which present this type of message to the controller are candidates for event-oriented contingency specification.

All external (physical) information available to the controller is in one of two modes: level or pulse. As shown, levels tend to resemble conditions while pulses resemble events. There are qualitative differences between event and condition which relate them to pulse and level, but they are only tendencies. For example, events usually sense a single parameter, while conditions usually involve several. The dividing line is, however, quantitative: the magnitude of the bound. The bound is the maximum length of time that sensed (or applied) information is considered valid in the worst case. The final determinant of how a level or pulse is realized, then, is whether its bound is large or small. There is no absolute answer for this in terms of the application alone. The application must specify whether a contingency is a level or pulse, but bound values are relative until they are specifically realized. This relativity is significant. The speed of a particular hardware/software realization of a controller is the final arbiter of whether a contingency which seems qualitatively a condition can actually be realized as one. The availability of methods for translating between Mealy and Moore circuits strengthens the argument against the existence of an absolute, realization-independent criterion for distinguishing condition from event. This does not say that the concepts of level and pulse cannot be distinguished absolutely. We are concerned with how best to realize an answer to externally originating contingencies (that is, as conditions or events) regardless of whether they arrive in the form of levels or pulses. Looking ahead to the systemic interpretation of the model developed in Chapter III, events are usually assigned to interrupts while conditions are well suited to a polling approach. In the last analysis, it is the speed of the realization which determines whether interrupts are necessary for a particular contingency. This concept is stated as the principle of realization relativity: one machine's event is a faster machine's condition. The terms "event" and "condition" will be used throughout the remainder of this work to convey concepts useful in categorizing timing and environmental information, as well as concrete attributes of realizations.

A contingency (whether condition or event) is a description of a specific recognizable change. An agent is implied which identifies the change. Such an agent is called a discriminator. The inputs of a discriminator are the current values of the parametric group defining a contingency. A discriminator has a single binary output which is true when all current values of a parametric group equal their corresponding distinguished values (that is, when the contingency exists), and false otherwise. If the discriminator of a contingency is true, a task defined to respond to the contingency must be performed. A typical application will be composed of a number of such contingency-task pairs. A reduction in complexity accrues from viewing each pair separately, as though each one were assigned to a dedicated, independent realization. It is useful, then, to consider the features of a single, generalized contingency-task pair.

2.2 THE PARAMETERS OF CONTROL

Controlling is repetitive by nature. It is a continuous process that is never completed as long as control is to be maintained. This means that a contingency-task pair describes a highly redundant course through time, like a set of beads on a string. This string of contingencies (some followed by tasks, some not, as dictated by the discriminator) spread through time defines a contingency time line. The collection of all time lines (that is all contingencies) of an application forms the time manifold of the application, and ultimately of the realization (the control system).

Consider one cycle of contingency and active task for a single time line (see Figure 1). Contingencies are represented by brackets and tasks by parentheses embedded in the time line. The time line is read from left to right.

There are five numbers which characterize a cycle of a time line. These numbers express three types of limiting condition: δ , the maximum duration; β , the bound or deadline (previously introduced); and ρ , the period. Both contingencies and tasks have duration and bound values. A subscript of one indicates a contingency while two indicates a task. Following this convention:

δ_{1i} is the maximum amount of physical time required by the discriminator of the i -th contingency to produce its output once started

δ_{2i} is the maximum amount of physical time required to complete the i -th task once started by an instantial existence of the i -th contingency

β_{1i} is the maximum amount of physical time allocated to the i -th discriminator to insure no lost information from the external environment.

Figure 1
A Generalized Time Line

β_{2i} is the maximum amount of physical time allocated to the i -th task to insure no information sent to the external environment is ignored.

ρ_i is the period of physical time allowed for any instance of a contingency-task pair.

The values of β and δ have consistent interpretations whether they refer to condition or event oriented contingencies. This is not true of ρ , however. In the case of an event, ρ represents the minimum amount of time which can ever elapse between instances of the event. In the case of a condition, ρ represents the maximum amount of time that can be allowed to elapse before the discriminator must be activated to avoid loss of information. The difference does not warrant separate symbols as the representation is clear in context.

These values (five for each contingency) supply the necessary global information needed to determine whether a monitor exists which can continuously schedule all contingencies to provide temporal congruence with the application. However, it is important to realize that these figures relate only to contingencies and tasks as gross entities. For example, the bound values only give deadlines for when the entire contingency or task must be completed, and say nothing of internal timing constraints. This means that even though a realization may be configured to meet all contingencies, individual operations (such as acquiring a parameter current value) may not be completed once started with sufficient rapidity to meet the demands of an application. Such time critical internal sections of contingencies and tasks are called timed blocks. There may exist several or no timed blocks in a contingency or a task. They are symbolized by γ , regardless of their number. The usual subscripting convention is used (γ_1 for contingencies, γ_2 for tasks).

As previously stated, information may be acquired by the controller in only one of two modes: pulse or level. Since these values define the timing associated with information acquisition, it is not surprising that they also specify the time line mode. ρ may be viewed as that period of physical time during which information is stable; that is, the external environment cannot change to a new state more than once every ρ . β_1 is the amount of physical time the information is valid. Therefore, if $\beta_1 = \rho$ the time line is externally generated as a level, since information is valid throughout its stable period. If $\beta_1 < \rho$ the time line is generated by a pulse, since the valid information exists for less than its period. Note that the definition of β_1 requires that only a single span of valid information can exist every ρ (that is, multiple pulses within a period cannot exist). Finally, $\beta_1 > \rho$ cannot occur since it violates the definitions. A knowledge of the mode gives insight as to the ultimate realization of a time line's contingency, but it does not guarantee that the tendencies toward condition/level and event/pulse realization will always be actualized.

Consider now the information available to a system as a basis for producing an adapted monitor.

Some of the parameters supplied in a behavioral description of a controller have already been introduced, others are new. These parameters (for the i -th contingency/task) are:

- β_{1i} the maximum amount of time an external parameter group is valid.
- β_{2i} the maximum amount of time allowed to complete a task once its contingency is true.
- ρ_i the minimum period between possible valid changes in the external environment.
- γ_{1ij} the maximum amount of time allowed to complete the j -th timed block of the i -th contingency once started.
- γ_{2ij} the maximum amount of time allowed to complete the j -th timed block of the i -th task once started.

Each parameter is supplied by the system user, as each forms a part of the application definition. These values are critical in determining a controller specification.

A second group of parameters, not given as part of the application definition, have values which depend on the realization to be used. These parameters are:

- δ_{1i} the maximum amount of physical time required to execute the i -th contingency realization.
- δ_{2i} the maximum amount of physical time required to execute the i -th task realization.
- σ_{ij} the maximum amount physical time required to execute the j -th timed block of the i -th contingency realization.
- σ_{2ij} the maximum amount of physical time required to execute the j -th timed block of the i -th task realization.
- λ_{1i} the latency of the interface hardware required to support the i -th contingency realization.
- λ_{2i} the latency of the interface hardware required to support the i -th contingency realization.

λ_{1ij} the latency of the interface hardware required to support the j-th timed block of the i-th contingency realization (λ_{1ij} is included in λ_{1i}).

λ_{2ij} the latency of the interface hardware required to support the j-th timed block of the i-th task realization (λ_{2ij} is included in λ_{1i}).

It should be noted that we need not consider the individual constituents of a given λ parameter. This is due to the use of the binder which configures hardware as necessary to realize each contingency or task. For example, λ_{2i} is a composite value of the latencies of all hardware deemed necessary by the binder to realize the i-th task.

Since these parameters are necessary to both resolve the contingency type question, and determine an adapted monitor, a value for each parameter must be specified. Values for each λ_1 and λ_2 are determined by summing latencies of individual hardware elements (directly available from the realization library). However, calculation of the remaining parameters requires a worst case flow analysis of generated software.

The Functional Procedure furnishes the Temporal Procedure with a number of symbolic data groups which complete the quantitative information previously discussed. Realization software consists of the contingency programs (P_{1i}), task programs (P_{2i}), the contingency interface programs (S_{1i}) and task interface programs (S_{2i}) of the application expressed in a format usable by the chosen processor. This format could range from high level languages to machine instructions; assembly language notation will be used. Realization hardware consists of two data groups: the contingency interface hardware (I_{1i}), and the task interface hardware (I_{2i}).

2.3 CONTROLLER TIMING

Timing, T , is the "bridge" relating internal abstract processor time and external physical time actually used by the processor. In the simplest (yet widely applicable) case, T is a table specifying worst case execution times for each instruction executable by the realization processor. Any serial-execution monoprocessor can be specified in this way. For more complex architectures (such as pipelined machines) where run-time dependencies greatly affect execution times, we must use more complex specification techniques; if calculated values are to be representative. A form of the simple table approach will be used in this thesis, since it is sufficiently general for a large class of dedicated processors, including all currently available microprocessors. Therefore, the timing should be thought of as a function which, when given a program sequence, returns the worst case execution time to be expected of that sequence, as performed on a specific processor. The final capability needed to calculate parameter values is a way of determining the worst

timing. In the simplest case, an operational definition may be a set of relations on the parameters discussed above (such as β , ρ and δ). In the most complex cases, operational definitions may require a complicated set of interrelated rules, in addition to the simple parametric relations. Complex monitor strategies are not easily generalized, so they will usually require an algorithmic definition imbedded in the system. In either case, the use of the definition remains the same: the determination of whether or not the functionally correct realization can be temporally controlled by adapting the monitor specified in the operational definition. If so, the adaptation is performed according to the definition. Otherwise, another monitor must be considered. In an actual design automation system, each operational description is implemented as a subprogram.

Now that a structured view of controller design (and a general approach to design automation) is established, it is possible to consider a system that produces controllers. Such a system has three parts

- 1) a realization-independent language to express the behavior of a required controller - CSDL;
- 2) a source of realization information (software and hardware capabilities for various processors) from which controllers are assembled - the library;
- 3) a method of obtaining monitors to control the realization - monitor adaptation methods based on timing analysis.

Each of these parts will be discussed in detail, and then brought together to form a controller design automation system.

3. CSDL: CONTROL SYSTEM DESIGN LANGUAGE

The complexity of specifying a system can vary greatly depending on the choice of description tools; cosmetic adaptations of syntax are not enough. It is counterproductive to fit our thoughts to an unsuitable language instead of using a language adapted to the abstractions of our thoughts. Thus, it is necessary to consider the elements uniquely tied to the nature of control, and how they affect the structure and orientation of a real time control system design language (CSDL).

The use of design languages in computer applications is certainly not new. Many were intended to describe the functional behavior of synchronous systems, while and to describe digital hardware. Simulation has been the primary motivation in their development. However, little work using such techniques for direct implementation has been done. Furthermore, descriptive design approaches have not been used in specifying real time systems. This has been due to the nature of physical time, which makes simulation of interface functions difficult. Assembly language has often been used to prescribe a system implementation, with the resulting program listing becoming the system description. This approach is popular because direct control of the preselected hardware is possible. Lately, the greater human efficiency afforded by high level languages has motivated attempts to add real time features to

them. However, these attempts have not been totally successful. The reason for this lies with a fundamental incompatibility between the nature of most programming languages and control.

Most programming languages are procedural, that is they are time ordered: the order of their statements reflects the order of actual execution. Control, on the other hand, while highly time dependent, is not inherently time ordered. A natural way to decompose a control requirement is by time line. That is, controlling (except in the trivial single time line case) involves simultaneity of multiple (and usually simple) control procedures. Many situations are naturally suited to an ordered (and therefore procedural) exposition. The ancient format of the recipe testifies to its appropriateness in cooking, a basically sequential activity. However, simultaneity is not well suited to a procedural format. Simultaneous requirements cannot be ordered, otherwise they are not truly simultaneous. Furthermore, the absolute primacy of timing bounds in real time and the repetitive nature of control make this incompatibility of expression even more significant than for statistically modeled situations such as arise in operating systems. This explains why the constructs that have been added to procedural programming languages to express concurrency and parallelism are difficult to use, and the situations they prescribe difficult to envision. They force the user to express non-procedural (simultaneous) situations in a language specifically intended to express procedural (time-ordered) computations.

The requirement to express both procedural and non-procedural components of a design in the same format is the root cause of the difficulty in expressing control problems. Indeed, it is one of the root causes of difficulty experienced in expressing concurrency of sequential processes in other areas of computer science (such as operating system design). Concurrency in a monoprocessor environment requires a monitor of some kind. The use of sequential notation to express concurrency "buries" the monitor by spreading its components through the various modules of a program.

In order to eliminate the problem of implied concurrent control from the design description language, we must partition it. One section of the language is devoted to expressing simultaneity; it consists of non-procedural, or static, specifications. A separate section contains only those specifications well suited to a procedural, or time-ordered, exposition (such as tasks). The division of the simultaneous and sequential portions of a description is the major structural feature of the language. Given this framework, it remains to determine structural details.

3.1 STRUCTURE OF THE LANGUAGE

The basic format for the design language is similar to that used in common programming languages. While such similarity is not necessary (as other formats are possible), it does reduce the problems of generating a realization in a practical design automation system. For example, each design description is divided into sections, each of which is defined by groups of statements. (The use of sections enforces the

basic structure of the language.) There is, however, a fundamental difference: intent. The intent of a programming language is the specification of a program which will ultimately control the operation of a machine. Even if the language is machine-independent, the purpose of the language is clear, and limits expression to software concepts. The intent of a design description language is the behavioral specification of a system without regard for its ultimate realization. The fact that the automation system developed here is oriented toward a computer-based realization cannot be allowed to influence the language design. The language must avoid any preconceived modes of thought based on realizations. This allows it to attain a high degree of portability in system description. Since realization is ignored, the language describes control requirements as amenable to an implementation which uses yet unknown technologies as it is to the protean microprocessor. The language, then, is the most general element of the design automation system, based as it is on an abstraction of real time control (the time line). Furthermore, it is the most important element of the system with regard to reducing design complexity, since it is through the language that control designers must express their thoughts.

Consider now, the four sections of CSDL. The first section of a design description is also the simplest. It is the design identification. It is here that the designer (or designers) are named, along with the project and a brief description. Revision information as well as other documentary remarks are collected here. This section identifies the following design, but contains no design information. It is included to encourage (and to some degree enforce) good documentation practice.

The controller and the controlled come together at an interface. Since the design description language is intended to specify the behavior of a system from the controller's side of the interface, the elements under control may be viewed as defining the controller's environment. The environment is a source and sink of information which must be controlled in strictly bounded physical time. The static aspects of the controller's environment are listed in the second section of the language, the environment. (The temporal aspects of the environment are more clearly understood when defined in a separate section.)

The third section of the language specifies the simultaneity of a design. Recall that an application description is formed of one or more contingencies, each associated with a task to be performed whenever the contingency exists. The string of such occurrences (and their various task invocations) describes a course through time (a time line). A controller may be thought of as regulating the behavior of one or more external activities such that they conform in time to a designer's time line definitions. The simultaneity of a design, then, stems from the need to specify several concurrent time lines. Therefore, we can use the requirements for defining a time line as the basis for determining simultaneity. There are two aspects involved in defining a time line: 1) relating a contingency to its task, and 2) bounding both contingency and task with respect to physical time.

Relating a contingency and a task is straightforward. When a contingency exists, its task must be performed. The circumstances under which a contingency is said to exist are completely defined by its

parametric group, P_1 . Likewise, each task is completely defined by its parametric group, P_2 . Therefore, specifying a (P_1, P_2) pair for each time line statically relates each task to its enabling contingency. That group of such pairs which defines all time lines needed to describe a design is referred to as its contingency list, the third section.

Contingency lists, in addition to allowing a natural way of relating contingencies and tasks, are non-procedural. This means that the statements (pairs) of the list do not require an ordering to establish control flow; each pair stands alone. Whenever a contingency is true, its task must be initiated. Since control of logical flow (and the way it is expressed) is central to the complexity experienced in defining procedures, its removal (and thus the use of "nonprocedures") usually reduces complexity.

Specification of a time line could be very difficult were it not for its inherent redundancy. The redundancy of a generalized time line is determined by the values of five parameters: β_1 , β_2 , δ_1 , δ_2 , and ρ . These parameters define a single worst-case cycle of a time line allowing a law of redundancy to be formulated which requires simply a nonterminating repetition of such cycles. Therefore, each time line is defined temporally by these five numbers. Fortunately, only β_1 , β_2 and ρ need be stated in the nonprocedural section, since δ_1 and δ_2 are realization-dependent, and must be determined by the system.

The fourth, and final, section of a design description is the procedure section. It is here that design specifications best expressed in sequential notation are recorded. As we have seen, timing and relationship of contingency to task are naturally suited to a static expression since they supply all information needed to determine the timing resources required of a time line. Therefore, a group of such time line descriptions defines what must be done simultaneously. However, these descriptions do not express what functions must be performed for each time line. This is done in the procedure section.

A contingency is defined by its parametric groups consisting of parameters in the environment. A contingency is said to exist when all its parameters equal their distinguished values. In many cases this could be expressed statically. However, the power of procedures with their decision capability makes them attractive for defining exactly what is meant by a distinguished value, without loss of generality. In addition to defining when a contingency is true logically, the procedure must also define internal timing requirements which must be met for the definition to coincide with physical time. These requirements are the contingency's timed blocks, γ_1 . The format of procedures used to define contingencies is the function, seen in common programming languages. A function is a procedure which computes a single value for use in the evaluation of an expression. The result of a function used to define a contingency must be either true or false, indicating whether or not the contingent event currently exists.

Tasks are also defined by parametric groups. However, tasks must cause change in the environment, rather than sample it. Tasks must force external parameters to assume their distinguished values. Sequential description is often the most natural way to express such tasks, especially since timed blocks (γ_2) may be used to specify internal deadlines. A procedure used to define a task is called simply a task. A task is identical in intent with the notion of subroutine, in that no value is explicitly returned. Both task and function procedures may be referenced within any procedure to facilitate modularization.

3.2 SPECIFICATION OF CSDL

Now that the fundamental structure of CSDL (and the reasons for that structure) have been developed, it remains to formally specify the language.

The primary aim of this research (and of much of structured programming research) is a reduction in complexity. The study of natural language indicates that two important aspects of efficient transmission of information between human beings are clarity of formulation and conciseness. A third aspect affecting the complexity of a design is completeness. These aspects conflict in such a way that it is necessary to rank their importance in specifying CSDL. Above all, a design must be well-defined (and therefore complete to the extent of eliminating ambiguity). Second, the language must be as simple as possible to make clear the intent as well as the behavior of the design. Finally, problem descriptions expressed in the language should be concise when this requirement does not adversely impact completeness and clarity.

Backus Naur Form is used to define CSDL. A section of this chapter is devoted to each of the sections of the language. However, it is instructive to first define its fundamental structure:

```
<control-system-design>::=<id-section><environment-section>  
      <contingency-list><procedures-section>
```

Consider the order in which the sections occur. Identification is first for obvious reasons. A description of the environment in which the controller is to function is next, forcing the designer to consider interface structure before behavior. Then the concurrent needs of the application are expressed in terms of contingencies and tasks communicating with the environment through the interface. Finally, procedures defining the exact behavior of each time line are expressed in the procedures section.

Note that this organization is compatible with the structured HIPO (hierarchy, input, process, output) design methodology. The inputs and outputs to which the controller has access are specified in the environment section. Processes are described in the procedures section. The top of a top-down specified hierarchy is expressed in the relationships of contingency to task in the contingency list. This list is essentially an overview of the control situation. It contains each function of the

control problem significant enough to warrant a separate (and therefore concurrent) time line. Every contingency and task is a module of a temporal hierarchy, as well as a logical one. Organizing the language in these sections strongly motivates the designer to find a natural high level temporal and logical structure for each control problem. It is through structuring that simplicity is approached. It is not our purpose here to define a complete version of CSDL, but to develop the language to a point that makes clear its structure. Furthermore, the language outlined will be oriented toward microprocessors used as universal logic elements (as discussed in the introduction), in order to focus on a concrete example.

The syntax of CSDL is defined in Appendix C using BNF. The reader is encouraged to use the definition in conjunction with the following descriptive text.

3.2.1 IDENTIFICATION SECTION

The identification section will not be discussed in detail, although this should not indicate a lack of importance. Thorough documentation is not a luxury, but a necessity, if productivity is valued.

3.2.2 ENVIRONMENT SECTION

The environment section details the format and characteristics of all data which must at sometime pass between the application environment and the controller. Such a description, from the controller's point of view, constitutes its interface to the controlled process. The term interface as used here has a broader meaning than is commonly ascribed to it. Our interface encompasses the usual direct transfer of data through communications registers and latches. This information may be considered a definition (at any particular time) of the current state of the controlled process. However, in many control situations, the past history of the process is also important. Therefore, the interface (and thus the controller's environment description) may include past information, recorded in counters, flags, and registers of the controller. This information is to be considered an encoded state history of the controlled process.

Now that the kinds of data to be found in the environment have been determined (data transmission and state history), we must isolate those characteristics needed to fully specify them. These characteristics vary significantly from those used in defining data in ordinary computer languages. The necessity of maintaining realization-independence is the source of this variation. While most languages can make use of implicit hardware features (such as word size), CSDL cannot. Therefore, a simple rule must be followed in writing CSDL data descriptions: assume nothing. If nothing can be assumed about data formats, what characteristics must be defined? They are those properties which are essentially invariant over a design specification: type, organization, precision and technology.

The concept of type in CSDL is broader than that usually seen in computer languages. Type usually refers to a data storage template, such as integer or real. The concept of type in CSDL gives no hint as to how the data is maintained. It instead refers to how the data may be used (similar to the class concept). The number of types that could be defined is endless, however we shall define only those needed to demonstrate the CSDL approach. The types allowed for data transmission are input only, output only and duplex (input and output). The types allowed for state history specification are binary and arithmetic. Binary data are fixed precision binary numbers, usually manipulated by logical operators. Binary data encompass both strings of bits and logical (or Boolean) data defined to be a single bit (single bit precision). Logical binary data take on values of true (1) or false (0). Real arithmetic types and operators could be easily added to the CSDL if required.

Binary data is distinguished from arithmetic data to reflect the essence of digital control. The environment presents the controller a world view composed of discrete signals which can be only true or false (on or off). Aggregating these signals and assigning them meaning is a human endeavor, done in the task and contingency definitions. The environment specification must, however, allow the designer to express the interface as it exists (that is, as a collection of digital signals). Arithmetic data are binary or decimal numbers used for computing. No operation is allowed that assumes anything about the ultimate realization of data storage or internal format.

Organization is in two forms: scalar and vector. Vector data may be multi-dimensional. Nothing in the semantics of CSDL implies that vector elements are stored sequentially.

Precision refers to the range of values a datum may assume. Logical, input, output and duplex data are always a fixed number of binary digits. The environment of a controller ultimately must supply a digital representation of its process. (For purposes of this study we assume that analog information must be converted to bits before it reaches the interface, thereby making a binary format natural for data transmission. The A/D-D/A converter design could, however, also be automated.) Arithmetic data is defined as having a fixed maximum integer and fractional part, expressed in decimal orders. Maximum values are required for true realization independence. Zero fractional places define an integer. Note that this does not necessarily imply that such numbers will be realized differently from real numbers. Precision is considered a true part of design definition rather than a way of allowing user-defined storage optimization.

Finally, data transmission types must have their technology specified. That is, the electrical characteristics of the signal appearing at the interface from the environment must be defined. (For example, a common technology is transistor/transistor logic, or TTL.) A realization cannot be generated to mate with the environment without a knowledge of the signal characteristics. Note that only data transmission types require a technology specification, since they alone directly connect with the controlled process. We shall use predefined technologies in this material, although a feature allowing definition of new electrical interface conventions would be an attractive extension.

Notice that, in addition to the numeric declaration types allowed, the user may define text. Text is a string of bits which are most conveniently associated with characters. Text is specified by a string of characters delimited by apostrophes (the apostrophe may be included by the traditional device of repeating it for each occurrence). The text string must then be associated with an identifier and a defining bit representation; this is accomplished by a code declaration. It is not necessary to do this in ordinary languages since their results are to be executed on specific machines; an appendix containing the bit representation of each character is usually available. However, if we are to remain realization-independent, use of text must be defined to match the requirements of the environment, and not some predefined processor. The tedious nature of this requirement may be mitigated by the inclusion in the language of often used codes given standard identifiers such as ASCII7 (for seven bit ASCII text).

3.2.3 CONTINGENCY LIST

A contingency list is an unordered list of contingencies; it is an enumeration of all relevant conditions and events which may eventuate at the controller's interface paired with responses (tasks) appropriate to the application. The relational aspect leads directly to the dominant statement type of this section, the WHEN-DO. This statement directly states that when a contingency exists, do the appropriate task. Included in the statement are the necessary timing bound definitions to establish a time line. These are β_1 , β_2 , ρ and rank.

An extended form of the WHEN-DO allows contingency existence to further depend on the states of other contingencies in the list. The WHEN-DO-UNLESS statement allows its contingency to exist unless some other set of circumstances prevail. The WHEN-DO-IF statement allows its contingency to exist only if some set circumstances also exist. These circumstances are termed a qualification. Qualifications may include any expression which can be evaluated to a boolean result (that is, true or false). Contingency identifiers may be used; they are true when the contingency exists. Even task identifiers may be addressed; they are true any time the task is active or scheduled. The use of qualifications allows a realization-independent expression of mutual exclusion through the use of complementary WHEN-DO statements (simplified here):

WHEN READ-READY DO READ UNLESS WRITE

•

•

•

WHEN WRITE-READY DO WRITE UNLESS READ

This form of expressing mutual exclusion is used to separate technique from concept. Techniques have no place in a realization independent language. The example above expresses the concept of mutual exclusion, while use of a semaphore (for instance) would have represented a particular technique to realize the concept, since it is subtly biased in favor of a dynamic monitor strategy. However, a phase locked monitor strategy can also guarantee mutual exclusion, by means of its intrinsic ordering.

Note that the contingency list may contain statements other than the WHEN-DO. These statements all directly reference time. Although this could be accomplished by a WHEN-DO construct, more natural statements such as EVERY-<period>-DO are allowed.

The timing specification allows a default value to be assigned to all timing elements but ρ (defaults are to ρ).

The simple DO was included to allow noncritical background tasks to be scheduled on a "time-available" basis, therefore no timing information is required. Its rank defaults to below the lowest time-critical rank.

3.2.4 PROCEDURES SECTION

The last section of a design specification defines the procedural (sequential) requirements of a design specified in the contingency list. Functions and tasks other than those directly named as contingency elements may also be defined, allowing modularization.

Practically all computer languages are procedural in nature, and any one (barring realization dependencies) could express the procedural requirements of a time line with the exception of timed block specification. However, the intent of this language is the expression of designs and not the prescription of programs. This difference in intent, and the recent progress in the study of language design, indicates (if not requires) that a new language be defined. Furthermore, the fact that existing languages and CSDL can be shown logically equivalent does not guarantee that they possess equivalent psychological efficiency. We must, as far as possible, evaluate language elements in light of their ability to allow simplicity of expression, since the overriding concern in designing CSDL is the reduction of complexity. Procedural CSDL (that is, the subset of CSDL used in the procedure section) is referred to as PCSDL. It is divided into four major statement types: control, data declaration, timing and action.

3.2.4.1 CONTROL

It has been observed that choice of sequencing control mechanisms greatly affects the complexity of the process being described. It is further observed that control should be kept simple. However, there are two ways to view simplicity. One view is that there should be few mechanisms. The other is that there may be many, but that each be individually simple. PCSDL strives to achieve both goals: a few simple mechanisms. This approach is not generally taken. There are currently several languages which have many features for expressing control flow, but it has been shown that only three (sequence, repetition and selection) are needed. The rationale seems to be the reduction in the total number of symbols needed to express a situation. The PCSDL approach would produce a more verbose description, but each part would retain simplicity due to the lack of more capable (some say tricky) constructions. (The number of keywords is reduced as a side benefit.)

Care must be taken to avoid limiting thought from a too parsimonious selection. For example, the DO<statements>UNTIL<condition> mechanism is not used since it can be replaced in all cases by the WHILE <condition> DO<statements> without loss of clarity. However, a less baroque form of the ALGOL FOR loop is included. Although the WHILE mechanism can replace it, the FOR construction seems a more natural way to express progressions, which often occur in real time control.

There are no GOTO statements in PCSDL. It is a common misconception that omitting the GOTO guarantees a structured program. It does, however tend to cause a more orderly approach. This is reason enough to eliminate it, although there is another reason specifically due to the use for which PCSDL is intended. As implied earlier, it is necessary to determine the maximum execution time of each existence of a time line (and therefore of its defining procedures). The use of such an unstructured mechanism as the GOTO makes that task difficult if not impossible. Similar reasoning makes a block structured language desirable. In a block structured language, a block of statements may be substituted for a simple statement. This modular organization is not only easier for humans to understand, it also allows simple, redundant methods in calculating maximum execution time. These timing considerations require that each loop mechanism be bounded. No action in a procedure may be unbounded in time, otherwise the entire procedure would be unbounded, making timing analysis impossible. Therefore, both the WHILE and FOR mechanisms must be accompanied by an explicit upper bound on the number of loops possible.

Removing the GOTO does have a disadvantage: exiting a nested structure can be complex. For this reason, labels are used to establish statement groups which may be directly exited from any level. No BEGIN-END construction is included; instead, explicit terminators (such as FI for IF and OD for DO) are used exclusively for their improved readability.

CSDL uses the two most common control structures for explicit modularization, the function and the task (or subroutine). They are identical in use and intent to their counterparts in most common languages (such as ALGOL), although their argument lists are more structured. Arguments must be grouped into input variables and output variables. The input variables must be stated first, followed by the output variables (delimited by a colon). This forces the designer to consider the role of each variable (similar to the HIPO concept) and allows more consistency testing (for example, a constant in an output list). Function types must be declared.

3.2.4.2 DATA DECLARATION

As in the case of control structures, an orderly approach to data also reduces complexity. Data is defined in declarations specifying their use and format, such as those seen in the environment. Procedures are excluded from declaring transmission data types (input, output and duplex). Transmission data define the environment as seen from the controller; if they are defined at all, they must be defined in the environment section.

Declarations in procedures may include any number of binary, decimal, text and code definitions. However, they must be located (as an unordered group) following the procedure introductory statement (task or function) but before any other noncomment statement type. There may be but one such group in each procedure. This forces data attributes to be considered before their use, and simplifies compilation and hardware binding. All identifiers (including function names) must appear in a declaration to establish their characteristics. While this is bothersome, the rewards in error-checking capability (due to redundancy) and explicit attention to data requirements can be significant in a widely used system.

Finally, we must consider the impact of realization-independence on data type use. Programming languages usually assume a host processor. Therefore, type usually describes a storage template for data. This cannot be in CSDL. Nothing may be assumed about data or how it will be stored in a yet-unknown processor. This situation impacts data manipulation. It is possible in most languages to take explicit advantage of a knowledge of the processor to be used. For example, logical operations can be performed on floating point variables to cause some desirable alteration. This is strictly prohibited by CSDL, since no realized format information is available at design specification time.

3.2.4.3 TIMING

CSDL must also be able to define timing if real time designs are to be described. Most of the timing information resides in the contingency list, but one major aspect is left to the procedure section: the timed blocks. Timed blocks are groups of procedural action that must occur within a fixed time. This is accomplished by the IN-NI construct, used to bracket timed blocks and express the time bounds (for example, IN 10 MS: X ← 0 NI;). We must also have a mechanism to mark time (delay); the WAIT statement is provided for this purpose. Finally a way is needed to delay until some event or condition has occurred, in order to trigger various sections of a procedure (thus, the WAIT UNTIL). Noting that mutual exclusion and concurrency may be specified in the contingency list completes in CSDL the capabilities thought necessary to describe concurrent sequential processes.

3.2.4.4 ACTION

Now that constructions for controlling flow, data and time are defined, it remains only to define the action statements. Action statements cause alteration and exchange of data (such as assignment and transmission). The details of such statements should vary as the orientation of the user; only the control mechanisms of procedure flow need remain constant. For instance, in process control design it would be convenient to have available "built-in" differentiation and integration facilities. For purposes of this discussion, the area of logic replacement by microprocessors has been selected to illustrate such application-area dependent concepts. Therefore, the action statements defined here resemble those of register transfer languages, allowing such features as direct reference to particular bits of a binary value (n:m). Action

statements involve only expression evaluation, assignment and transmission to keep exposition of the concepts simple. More capable, but complex, statements could be added to the language developed herein. However, care should be exercised to avoid the confusion a wealth of operators can cause.

An example design description incorporating many of these constructs is found in Appendix D. The example is used in conjunction with a specific realization description to illustrate the generation of a controller.

4. A REALIZATION LIBRARY

A design expressed in CSDL is still a rather abstract entity. The intent of this chapter is to demonstrate the feasibility of that abstraction in concrete terms. Methods of specifying all elements needed to construct (from a CSDL description) an actual controller using current technology will be discussed. These specifications will then be collected to form a library of realization elements (Appendix E). The library will be used by a demonstration system in Chapter VI to generate a controller design to realize the example description of Appendix D.

4.1 BASIC LIBRARY ORGANIZATION

Because a processing unit is to be the central component of each controller design, the library is organized in sections (called volumes) each dedicated to a different processor. Each volume is divided into two parts reflecting the model: the functional realization components and the monitor components.

Before we can investigate the library further, however, we need a general concept of how the library is to be used.

4.1.1 LIBRARY REFERENCE

The concept of generating code for different target processors from a high-level language is not new. Consider the task as being divided into three basic parts: lexical/syntactic analysis, semantic interpretation and code generation. First the input must be parsed according to CSDL grammar (Appendix C) to determine what is to be done by the controller (syntactic analysis). Then, this specification is tested, then converted into an internal representation convenient for manipulation (semantic interpretation). Finally (ignoring optimization), actual translation into a form acceptable by the target processor is performed. There are a number of well known techniques for constructing syntax analyzers, therefore this aspect will not be discussed in detail. However, the requirement to generate hardware specifications as well as code, coupled with the data requirements for timing analysis, requires extensions to known techniques for the remaining tasks. These extensions are reflected in the form of the intermediate code generated during semantic interpretation.

4.2 THE INTERMEDIATE CODE

Since we wish to generate hardware and software for a number of different processors described by the library, we must concentrate all realization (and therefore processor) dependencies in the code generation. This requires that the intermediate code passed to the code generator contain all necessary processor-independent information that may be required by the several machines to realize any CSDL action or requirement. These actions and requirements are called CSDL primitives.

The form of the intermediate code is based on the intermediate output of a typical compiler's syntax analyzer, called an n-tuple. An n-tuple consists of an operator (defining the primitive) and a number of operands. Operands define information needed by the code generator, and may be variable names, numbers, or any data that may be needed. The use of each operand depends on the primitive which supplies it. This n-tuple is the first part of the three-part intermediate code format.

The second part is the selection list. In general, much information is associated with each operand of the n-tuple. Such information is usually contained in internal tables and is referenced by pointer from the symbol table. The information may be used to select a code generation definition if there exist several ways of realizing a primitive. Software selection information may include such common items as precision, and items unique to CSDL such as maximum time delay possible for various realizations of the WAIT statement. Hardware selection information include items such as maximum number of data lines (discrete signals) and technology. For example, an input line to the controller declared as a TTL technology in CSDL, may only be realized using that technology if the interface is to be compatible.

The third section of the intermediate code is the attribute list. The attribute list is given values by the code generator after a specific method of realizing the defining primitive has been selected. Conventional code generators, once having generated code, are finished. However, the timing analyzer and hardware binder of the system require various types of information about the generated code. The information returned by the generator falls into two categories, software and hardware. Software primitives must return three fundamental values associated with their realization: storage, time, and external references. Storage is the number of basic units of memory required to realize the primitive. Time is the number of basic time units (clock cycles) required to execute the code in the worst case. External references define the number of total accesses to information sources external to the process (such as memory) that are made by the code in the worst case. (A count of external references allows the calculation of time delays required to match fast processors to slower, cheaper memory.) Storage values allow sufficient memory to be specified by the hardware binder, while time and external reference information together allow procedure execution times (δ) to be calculated.

Hardware primitives must return two fundamental values: latency and power. Latency is the maximum amount of time required by the hardware to perform its function. Consider three examples. The latency of the

processor defines the clock period; this number multiplied by the time attribute of a software primitive gives the intrinsic time required to execute the code associated with that primitive. The latency of the memory chosen by the binder gives the delay (if any) caused by slow memory access; this number multiplied by the external references of a software primitive gives the actual delay for the code realizing the primitive. The sum of the intrinsic time and the delay for external accesses gives the actual total execution time of the generated code. This value is then used by the timing analyzer. Finally, the latency of data transmission hardware primitive (to perform a SENSE or ISSUE statement) gives the interface timing delay (λ) caused by the hardware. The total execution time of such primitives must be increased by the amount of this delay.

Now that a format has been defined for specifying machine-independent primitives to realize CSDL statements (and retrieve their machine-dependent attributes), consider what those primitives must be. The primitives needed for Realization Set software generation are well known; they are the same as, or similar to, those used by various designers of table-driven compilers. The common "three address" intermediate form for arithmetic expression representation is used. It is in hardware and monitor descriptions that the use of primitives is unique.

There are only seven primitives which can be produced by the binder in order to define hardware needed to realize a controller (although each of these may cause more hardware definition as side effects). These seven are divided into two categories: controller and interface. The controller primitives are processor and memory. The processor primitive defines the processor hardware to be used and returns the clock period for timing analysis. The memory primitive defines the memory resources needed and returns the external-reference time delay value and basic unit of storage (in bits). The fact that the system is to generate both software and hardware mitigates the problems encountered by designers of table-driven compilers in isolating all machine dependencies in the code generator.

There are only two ways communication can occur between the controller and the environment, a condition or an event. Further, communication may pass in either direction (sense or issue). Therefore, there are four possible interface primitives for communication: sense condition, sense event, issue condition, and issue event. There exist four corresponding software primitives with identical names (S in the model) which contain code sequences to access data through the hardware specified for each communication requirement. Each communication hardware primitive returns its latency figure (λ). The last hardware primitive is rank, which describes any hardware needed to enforce a user-defined fixed priority.

Monitor primitives are generated only after timing analysis has successfully isolated a suitable monitor strategy. A basic assumption about monitor organizations is made to simplify their description. Monitors are assumed to be table driven, with the table directly following the body of the monitor. Therefore there are three monitor primitives. The first is simply called monitor, and consists of the monitor body. Its selection list contains the name of the monitor desired. The monitor

body returns attributes of storage and worst case monitor overhead time. Second is the table entry primitive, which defines a contingency/task pair to be executed, returning the attribute storage. Finally there is the table end primitive which specifies the end of the table, and any final processing that might be required.

Appendix F lists primitives that might typically be generated for the CSDL example of Appendix D. Each primitive is intended to cause some software or hardware description to be included in the final controller design from those available in the library (Appendix E). The list is organized in three parts. The name of the primitive appears first (software primitives preceded by S., hardware by H.). This is followed by the operand list and selection list (if any) separated by a colon. Finally, the attribute list returned by the code generator appears to the right. Given this method of expressing realization-independent hardware and software requirements based on a CSDL description, we must now develop a method for describing the realizations to be used.

4.3 LIBRARY FORMAT

The library is organized in volumes, one volume for each processor. Volumes are ordered according to preferred use with regard to some metric (such as cost or power consumption). This allows selection of more expensive systems, for example, only after less costly realizations have been tried. Each volume is divided into a functional realization part and a temporal realization (or monitor part). Each of these parts contains software and hardware descriptions applicable to the processor for which the volume is defined.

Each description is maintained in the library in a skeletal form called a macro. Macros contain an outline (or template) of code or hardware needed to realize the various primitives which may be generated in an intermediate code sequence. The operands of the primitive (the actual parameters) are substituted for placeholders (formal parameters) in the corresponding skeleton to tailor the outlined code to the specific needs of the generating application. The new concept of delayed hardware binding requires that our design automation system generate a support hardware configuration in addition to the conventional requirement for code generation.

A macro definition consists of an introductory statement specifying the macro name and associated parameters, followed by the body of the macro. There are two distinct methods for defining macros: there may be a few, complicated ones which process a number of similar primitives, or there may be many simple ones. The first approach is compact, but requires a relatively capable macro definition language to perform conditional expansion (based on operand mode, for example). The second approach is easily understood. For the purposes of this study, a macro definition language with some extended features will be defined, but it will be used in the context of simple macros. This allows the capability of the first approach to be demonstrated while permitting the simplicity of the second approach to ease understanding of the illustrative examples. Consider now the statements of a macro definition language.

Macros are introduced by the SKEL declarative. The SKEL statement is divided into four parts: the skeleton identifier (or name), the operand list (or formal parameter list), the selection list, and the attribute list. (This is, of course, the same format as that of the intermediate code primitives which invoke expansion of the macros). The identifier field must begin with either an S. (for software macro) or H. (for hardware macro) followed by the name of the macro. This is done for readability and because it is often the case that an action requires both software and hardware components, making identical names convenient. The operand list is a group of formal parameter names, separated by commas, which are used in macro expansion to indicate positions for actual parameter replacement. The selection list supplies values needed to select a particular macro which has several representations. The attribute list defines values required for further processing, as described above. The library of Appendix E contains a number of example macros. For the reader's convenience, two comment statements (introduced by the COM declarative) appear directly following each SKEL statement. The first comment describes the purpose of the primitive that the macro defines, while the second describes the purpose of each parameter in each of the three lists. Note that the lists in each SKEL statement are enclosed in parentheses, and separated by colons. A macro definition continues until the next SKEL declaration is encountered.

In order to reduce overall macro bulk, and to promote clarity through modularization, macros may reference other macros. This takes two forms. The CALL-macro statement causes the called macro to be included in the design definition at the point in the calling macro where the call statement appears. It is identical in intent with a subroutine invocation. The INCL-macro (include) statement also causes the called macro to be included in the design definition, but not as part of the calling macro. This capability allows, for example, software macros to cause needed hardware (such as a part definition for a SENSE primitive) to be included in the design without being interspersed in the generated code; the same is true for hardware macros which require ancillary support software. Both the INCL and CALL statements may pass expansion arguments to called macros.

A capability for making simple arithmetic calculations is often needed to avoid repetition of very similar, but not identical, macros. Such calculations (in the form of parenthesized arithmetic expressions) are allowed only following a CALC declaration. There is no continuation device, therefore long calculations must be partitioned. Identifiers receiving values in CALC statements are global; they are known by all other macros to allow inter-macro communication.

A simple method for conditional expansion of macros is possible with the IF statement. This statement has the form IF <logical-expression> SKIP <lines>. The "logical-expression" is an expression involving only relation operators and defined parameters; the result must be either true or false. "Lines" specifies the number of statements directly following the IF statement which are not to be processed during the current macro expansion if the associated logical expression evaluates to true.

A TEMPLATE statement is used to determine the actual appearance of generated output. Such details as label placement (by column count) and comment delimiters (if any) are specified. The first data field of a template must be an S (to define the software generation format) or an H (to define the hardware format). If the template statement is omitted, generated output will appear as it does in the skeletons. Templates allow the specification of processor peculiarities without affecting the generality of the syntax analyzer. Only one software template and one hardware template are allowed in each volume.

The most important part of a skeleton definition is the actual code or hardware skeleton itself. The generic term used to refer to both software and hardware skeletal outlines is text. Hardware is defined in H-text blocks and software in S-text blocks. The collection of H-text blocks for a particular processor, H, corresponds to the mapping Δ_H in the CSD model (the Hardware Binding Transformation). Likewise, the collection of S-text blocks corresponds to Δ_H (the Algorithm Translation Transformation). H-text blocks are introduced by a BEGIN HTEXT statement, while S-text blocks begin directly following a BEGIN STEXT statement. Both types of blocks are terminated by an END TEXT statement. Actual parameter values are substituted in text blocks wherever a formal parameter is found bracketed by the symbols < and >.

H-text and S-text blocks are processed identically (according to their associated H or S templates), but the results of the processing (expansion) are directed to two separate text streams. All S-text blocks are, after expansion, copied to the software listing. All H-text blocks are added to the hardware configuration listing. Together, these listings define a complete control system design to realize the control behavior specified in the CSDL input.

Finally, we define a set of system functions to ease common calculations and aid in formatting. Each function returns a single value which replaces the function invocation. A function is specified for invocation by enclosing its name and argument list (if any) in a pair of back-slashes (\). A number of useful functions of this sort have been discussed [Matelan, 1973], although only those used in the library example will be defined:

\FORMAT (variable, type, precision)\ - The format function allows a specific representation of the value of a variable to be specified. For example, FORMAT (EVPNT,B,3) returns a three bit (binary) equivalent of the value of EVPNT. (Default format is decimal.)

\PREC (variable)\ - PREC returns the precision of the variable.

\IDSEC (line)\ - IDSEC returns the specified line of text of the identification section (for documentation purposes).

\STATUS (n)\ - STATUS returns various status information, for example:
n = 1 : fundamental units of memory used;
n = 2 : total number of interface ports defined;
n = 3 : total number of events defined.

4.4 AN EXAMPLE LIBRARY

Specifications for various elements to realize an actual controller, using the macro approach described here, are given in the realization library of Appendix F. The technology used in this example is that of microprocessors. Specifically, the Intel 8080 eight-bit microprocessor and its support hardware are used, because it has been widely exploited in control applications and because it is representative of current microprocessor capabilities [Intel, 1975]. (The library format is, however, a general one capable of defining the realization capacities of a number of other processors.)

All of the software skeletons in the example are given in a version of the 8080 assembly language. This was done for several reasons. Although slow, assembly level code generation removes the bookkeeping problems of generating machine code. It also provides another level of documentation, functionally prescribing the controller in terms of its realization. It should be realized that other output formats could have been used.

Since the concept of generating control processing hardware is new, a method for organizing hardware text blocks was needed. The approach, again for clarity, is the use of descriptive text indicating the use of each device, the actual device used, and the connections which must be made between the device and the rest of the system. Connections are specified between each device connection pin and the signal set to which each pin must belong. (A signal set is a collection of circuit nodes which are electrically common.) Pins are assigned to signal sets by equal signs; device nomenclature for each pin is included in square brackets for convenience. Collections of lines considered in aggregate (such as buses) may be specified by a single name followed by pin numbers enclosed in parentheses and separated by a colon (for example DB(1:3) represents the first three parallel signals of a data bus).

The text format used in defining hardware is somewhat informal for ease of reading, although all information needed to actually assemble a working device is specified. For example, memory-device-select signals result from decoding the address bus. This is stated simply as DECODE <signal-set>VALUE<number> which defines a standard decoder whose output is true when the signal-set value is equal to the enabling value. A more formal approach would be the specification of decoder gates in a register transfer language such as AHPL.

Before moving to a discussion of a system using the generation techniques of this chapter, the reader is encouraged to trace the expansion of several of the primitives listed in Appendix F in conjunction with the library of Appendix E. Appendices G and H are the resultant text streams for software and hardware realizations, respectively. Macros are expanded in an order corresponding to the primitive list. The @ symbol is used to distinguished system-generated identifiers and labels.

5. TIMING ANALYSIS

Given a way to express a design and way to specify components to functionally realize the design expressed, it remains to determine methods of timing analysis.

The purpose of a monitor is to control the execution of contingency and task programs in such a way that no incoming data is ignored and no response is lost. Therefore, a schedule, or ordering of execution, is required. Codd, in his work on multiprogram scheduling, divides schedulers into two modes: the static mode and the dynamic mode [Codd, 1960]. In the static mode, schedule generation is done prior to, and separated from, its execution. In dynamic mode, the schedule is determined during execution, based on past performance and current, real time demand. The static mode requires a backlog of unscheduled programs in order to generate its schedule.

5.1 MONITOR TAXONOMY

Control, by its nature, is cyclic. It may be viewed as defining a perpetual, consistent and well-defined backlog of "programs" to be executed, therefore making a static (or pre-scheduled) approach possible. Such a schedule fixes the relative order in which contingencies are executed. Therefore, each time line maintains a constant phase relationship with all other time lines of an application. Monitors which take advantage of the redundancy and regularity inherent in control problems by using static scheduling will be termed phase locked monitors. On the other hand, scheduling may be done at execution time; such monitors are called dynamic monitors. There is a third general monitor type: the hybrid monitor. These monitors combine certain features of phase locked and dynamic monitors.

Several examples of each general monitor class are given in this section. These examples, while representative, are by no means exhaustive. There are a large number of possibly useful monitor strategies known, and no doubt more will be invented. We are here concerned, however, with determining the general characteristics and relative usefulness of each monitor type, so that they may be ranked and their selection automated. Each monitor type is not equally useful in controlling all applications. Monitors need to be ranked according to their relative applicability to various types of control application. Such a ranking (and the rationale used in its determination) is defined following the general discussion.

It is advantageous, before discussing each monitor type in detail, to define those constraints and concepts which all types share. These constraints are called feasibility tests on the realization. They must apply to all time lines; a single test failed by a single time line is sufficient to make a realization infeasible, and therefore uncontrollable by any class of monitor.

The relationship of β_1 to ρ in defining level and pulse gives, by definition:

$$\forall i, \quad \beta_{1i} \leq \rho_i \quad [5]$$

Any contingency definition not so related is incorrectly specified in the application definition.

It is necessary that the physical time required to execute contingencies and tasks be within the bounds set by the application, therefore:

$$\forall i, \quad \delta_{1i} \leq \beta_{1i} \quad [6]$$

$$\forall i, \quad \delta_{2i} \leq \beta_{2i} \quad [7]$$

Each of the timed blocks must also execute within its defined limits:

$$\forall i, j, \quad \gamma_{1ij} \leq \sigma_{1ij} \quad [8]$$

$$\forall i, j, \quad \gamma_{2ij} \leq \sigma_{2ij} \quad [9]$$

It is also necessary that timed blocks be consistently defined with respect to their containing contingencies and tasks:

$$\forall i, \quad \sum_{\text{all } j} \gamma_{1ij} \leq \beta_{1i} \quad [10]$$

$$\forall i, \quad \sum_{\text{all } j} \gamma_{2ij} \leq \beta_{2i} \quad [11]$$

This is again a test of proper application definition. Lastly, we must ensure that a contingency and its task can be completed before new information arrives:

$$\forall i, \quad \delta_{1i} + \omega_{1i} + \delta_{2i} + \omega_{2i} \leq \rho_i \quad [12]$$

Finally, the concept of priority must be more carefully defined. We must differentiate between "priority" as viewed by the user defining an application and "priority" as an ordering used in a scheduler to allocate execution resources. The first type of priority will be referred to as extrinsic priority. It is part of the application definition, and therefore it is a fixed ordering. The meaning attached to extrinsic priority in hard real time environments differs from that usually given in operating systems contexts. Real time extrinsic priority defines an order such that a contingency of higher priority must execute even if it violates the constraints of a lower priority contingency. An example of allowable extrinsic priorities in real time applications is background

programs which have a preferred completion time but cannot be allowed to interfere with time critical contingencies. In general, time constrained contingencies cannot be given an extrinsic priority ordering, since they must all meet their deadlines all the time. Intrinsic priority refers to the relative time criticality of contingencies while executing, but does not rank importance. In hard real time, all deadlines have equal importance. Otherwise, a statistical description of timing requirements would suffice.

5.2 PHASE LOCKED MONITORS

The constant phase relationship of each time line with all others of an application is the main feature of phase locked monitor operation. This is shown in Figure 2, a directed graph of a generalized three contingency control system. Each circular node represents a contingency program and each square node represents a task. Tasks are related to their contingencies by the number of the node. Each arc is labeled with the value of the maximum time spent in the node it leaves. The path described by moving from node to node until the starting point is reached is called a cycle. The time required to make a cycle, the cycle time, is the sum of the values of the labeled arcs traversed. Whenever a contingency is true (exists), the path leading to its task is taken. Whenever a contingency is false, the path to the next contingency is taken. Therefore, the minimum cycle time is always the sum of the arcs between contingency nodes (i.e., when no contingencies exist). The maximum cycle time is always the sum of the arcs between all nodes (i.e., when all contingencies exist).

Figure 3 shows an example time line for each of the contingencies of Figure 2. Various combinations of true and false contingencies are shown for each time line. It is clear that no combination of contingencies and tasks can cause the order of contingency execution to change; they are locked in phase. It is also clear that interrupts cannot be allowed in a phase locked monitor implementation since their random existence could cause execution to lose synchronization. Therefore, phase locked monitors may be considered pure polling implementations.

This means that the phase locked approach may only be used when all contingencies of an application present levels to the controller, that is:

$$V_i, \quad \beta_{1i} = \rho_i \quad [13]$$

Since phase-locked monitors are pre-scheduled, they do not depend directly on physical time, so no accurate, real time source is needed. If a single cycle of a schedule can be found which meets worst case timing constraints, then the timing analysis is finished, since the ordering of that cycle will correctly schedule all other possible cycles. Inductively, if the monitor is started in a correct cycle it must continue in phase indefinitely. If the ordering found is sufficient in the worst case, then it is sufficient in all cases, since the worst case

Figure 2

A Three Contingency Control System

Figure 3

Time Lines Associated with Figure 2

occurs when all tasks are active (the maximum cycle time is required). If some task is not active, then the period between its containing contingencies can only shorten and (since phase lock is guaranteed), bounds cannot be exceeded.

It remains to determine criteria for finding a sufficient ordering for phase locked monitors. These criteria are divided into two categories: global timing and local timing. Global timing criteria define the applicability of a monitor to an application in a gross way. They must determine whether or not the total time available in a worst case cycle is sufficient to execute all contingencies, tasks, and their associated overhead. Local timing involves determining a schedule which resolves local time line conflicts such that all individual timing constraints are met. This is approached by generating trial schedules and choosing the first one which satisfies all constraints. This division allows the relatively inexpensive reasonableness tests specific to a particular monitor strategy to be made (that is, gross timing tests), before the more involved local tests are made to generate and evaluate trial schedules.

Example monitors will be described and their criteria given to clarify the use of the phase locked monitor approach. The first example is a simple, but widely applicable, monitor designated P1 which is often implicitly used in the literature. Figure 2 shows a situation controllable by P1. Each contingency node appears once in each cycle. Each task, when active, is executed to completion before the next contingency is examined. Consider, then, the timing criteria of P1 which define this situation.

Consider first the global criterion of the P1 monitor. Let $\hat{\rho}$ be given the value of the largest ρ_i defined by the application. This is called the worst case cycle length. Let α_i be the worst case cost in time resources that can ever be needed by a single instance of the i -th contingency. Therefore:

$$\alpha_i = \delta_{1i} + \omega_{1i} + \delta_{2i} + \omega_{2i} \quad [14]$$

Note that equation 3.14 guarantees that all α_i will be less than or equal to ρ_i , (it is assumed that all feasibility tests have been met).

We may now normalize each worst case time cost for each time line to the time required during the longest possible period by multiplying by the normalization factor $(\hat{\rho}/\rho_i)$. Let $\hat{\alpha}$ be the maximum cost that can possibly be billed to the monitor in a worst case cycle (that is, $\hat{\rho}$ in duration). $\hat{\alpha}$ is found by summing the normalized contributions of each time line:

$$\hat{\alpha} = \sum_{\text{all } i} \frac{\hat{\rho}}{\rho_i} \cdot \alpha_i \quad [15]$$

Given these values, it is possible to define a relation for determining whether the gross timing criterion is met for the application under control of the P1 monitor:

$$\hat{\alpha} \leq \hat{\rho}. \quad [16]$$

(It should be noted that, since P1 is the simplest possible monitor, relation 16 also defines the weakest global timing bound for all other monitors regardless of classification. If a monitor fails relation 16, it will surely fail when overhead is introduced.)

If the relation is not met, the time required for worst case operation is greater than one worst case cycle, which of course is a physical impossibility. If the relation is met, it may be possible to use the P1 monitor. The local timing criteria will determine whether the time lines of the application can be arranged with respect to one another so that all timing requirements are met.

An incrementally generated trial schedule for the P1 monitor may be used to make local timing tests. Choose a time line as the basis. Then choose a second time line and determine whether the combination violates any individual timing constraint. If not, a consolidated time line may be found from the first two time lines. A third time line is chosen and combined with the composite time line. Tests for local constraint violations are again made. If there are none, a new composite time line is formed of the first three. The procedure is followed until a new time line cannot be accommodated (the procedure fails) or all time lines have been incorporated (the trial schedule is satisfactory). The simplicity of P1 shows that all time lines may be included in the final composite whenever the sum of all α_i is less than or equal to the shortest repetition interval (ρ_i). A valid schedule for P1 may be executed by an equally simple monitor. The phase locked monitor progresses through the schedule, initiating execution of each contingency entry in turn and each task entry if its corresponding contingency was true. At the end of the schedule, the monitor returns to the first entry, endlessly repeating the cycle. Note that since the minimum cycle time for contingencies may cause their execution more than once in a repetition interval, a completed task should force the input information for its contingency to an invalid state to avoid multiple task executions. This is not necessary, however, if the application calls for repeated task execution while the contingency exists.

Dedicated control applications amenable to a P1 monitor strategy are common in random logic replacement applications. However, more complex applications may require a more complex strategy, such as P2, the second phase locked monitor example. It is often the case that the repetition rates (ρ) of an application's contingencies are not all long enough to allow a simple cycle containing one execution of each contingency. One or more contingencies may require several invocations during an interval when another may require only one. The approach used in securing a P2 monitor schedule for such an application is the determination of a

single cycle in which an integral number of invocations for each contingency are performed. This is possible since we may allow more contingency invocations per repetition interval than needed (as in the case of P1 monitors), but not less than one per interval. Before effort is expended determining such a schedule, gross timing tests should be made.

The schedule cycle for a P2 monitor is determined by that contingency with the longest ρ . This contingency need occur only once in the cycle. Let $\hat{\rho}$ equal this greatest ρ . The smallest number of complete contingency invocations needed in this cycle (for the i -th contingency) is given by

$$\epsilon_i = \left\lceil \hat{\rho} / \rho_i \right\rceil \quad [17]$$

Therefore, the total time expended executing all contingencies (and their tasks) in the worst case is:

$$\hat{\alpha} = \sum_{\text{all } i} \epsilon_i \cdot (\delta_{1i} + \omega_{1i} + \delta_{2i} + \omega_{2i}) \quad [18]$$

Again, if $\alpha \leq \hat{\rho}$ then it may be possible to arrange all required contingencies in a cycle whose length is less than or equal to $\hat{\rho}$; if $\hat{\alpha} > \hat{\rho}$ it is not possible.

If the global criterion is satisfied, the local tests may be made. Once again, we shall attempt to incrementally generate a schedule. Order all contingencies so that if the i -th contingency precedes the j -th contingency, $\rho_i \leq \rho_j$. We shall build a composite time line one worst case cycle in length, beginning with the most critical contingency (that is, the one with the shortest ρ), to the least critical (longest ρ) contingency. We must arrange ϵ_1 contingency-task invocations on the trial time line in such a way that no interval ρ_1 exists without at least one invocation. Next, ϵ_2 invocations of the second most critical contingency must be fitted into the time remaining from the first contingency. This process continues until there are no contingencies left to be integrated (which produces a P2 schedule), or until there is not enough time to accommodate one of the remaining contingencies (a local failure). The order in which each contingency is added to the composite time line defines its intrinsic priority. The order used (most critical to least critical) is called rate monotonic. It can be shown optimum in the sense that if a rate monotonic intrinsic priority ordering cannot schedule a composite time line, then no other fixed ordering exists which can.

Each contingency on the trial time line may be thought of as a color-coded bead on a looped string ρ in length, which may be moved in either direction (see Figure 4). Each bead of a contingency (each

Figure 4
Determining a P2 Schedule

invocation) has a length proportional to its α and it is bounded so that it may not be pushed more than its p time distance in either direction from its nearest left and right invocations (same colored beads). We must find an ordering (schedule) which arranges all beads within their limits. The relative unimportance of utilization means that any such schedule which satisfies the global and local constraints is usable. Therefore, passing these local tests also produces a schedule.

In situations where a large number of contingency invocations would be necessary (due to wide differences in p values making so ϵ large), a partitioning scheme could be used. For instance, a partition of less critical contingencies might be invoked at long intervals rather than continuously. Partitions could also be based on contingencies with similar values. These partitions may then be tested separately for their applicability to the PI monitor strategy. Such sub-strategies might then be invoked according to their composite ϵ value.

It is possible to make several general remarks about phase locked monitors based on the examples. First, the monitor is very simple. It has but to endlessly select the next contingency from its schedule and execute it. Therefore, monitor overhead is small. Second, if a schedule can be found, it can be guaranteed to meet all timing constraints since it meets worst case constraints. Finally, rank is not a consideration, since the existence of a phase locked schedule guarantees service, further reducing hardware and monitor overhead. Consider the elements of rank, priority and precedence. Extrinsic priorities are necessary only if there is a lack of resources; it would not be a consideration if (for instance) a processor were available for each contingency. If a schedule exists, it meets all timing requirements for all contingencies, so that ordering them to resolve conflicts (that is, assigning priorities) is not necessary. Precedence relations between contingencies can be guaranteed by creating a schedule in which they hold. Mutual exclusion can be guaranteed without active mechanisms (such as semaphores) for the same reason.

5.3 DYNAMIC MONITORS

Phase locked monitors are pre-scheduled, therefore their schedules are fixed. Their operation coincides with external requirements, but it is not driven by them. This mode of operation is reversed in dynamic monitors. A dynamic monitor incrementally generates the next element of its schedule in real time according to the current state of the controlled environment. The general characteristics of dynamic monitors will be discussed so that they may be compared with other types.

The DI monitor is based on the Relative Urgency Algorithm (sometimes called the Earliest Deadline Algorithm): that active task which has been requested and which has the earliest deadline (most relatively urgent constraint) is to be executed. This approach has been shown to be optimal where overhead is not considered.

Two implementations are possible, both of which use a pure interrupt orientation. If all contingencies can be processed as interrupts, the scheduler can reevaluate the relative urgency of each active task at

each interrupt and switch execution, if necessary. Whenever a task completes, a fixed priority list is used to start a new pending task until the next interrupt. This scheduler defines dynamic monitor D1.

A second implementation, D2, is needed if one or more contingencies are not amenable to an interrupt realization. An interrupt generating, decrementing clock may be used to realize conditions as interrupts. Initially, that contingency (C1) whose deadline is nearest at the current time is chosen for execution. The next most urgent contingency (C2) is chosen and the time in the future when it will become the most critical task is used to preset the counter. Both contingency and counter are started together. When the counter reaches zero, the current contingency (C1) is suspended, the next most critical contingency (C3) is used to preset the counter, and the currently most urgent contingency (C2) and the counter are started together. This procedure, endlessly followed while control is required, will even realize applications with no external interrupts. External interrupts may also be accommodated since they may be viewed as defining, and then initiating, the most urgent task at the time of their arrival. This approach, therefore, is characterized by a dynamic priority scheme based on current physical time and elapsed time since task initiation. Other dynamic approaches (such as use of fixed priorities) are possible but they can be shown to provide performance only equal to or poorer than D1 and D2.

The gross criterion for the dynamic case is given by the relation of equation 16 for the phase locked case. This is a very weak bound for a dynamic monitor since the overhead for interrupts caused by counter time-out is not included, nor is overhead needed to calculate and set the counter following each interrupt. There is no way to predetermine the number of interrupts that will be needed in the worst case, so it is not possible to include the overhead required to perform them.

Finally, there is no local criterion that can be used to guarantee that an application will always be correctly controlled by a dynamic monitor. Dertouzos' proof, which shows D1 optimal, states only that it will properly control an application if it can be properly controlled, ignoring all overhead. The more costly and more frequently invoked monitor overhead code can invalidate the proof.

5.4 HYBRID MONITORS

Hybrid monitors combine characteristics found in the preceding two types. For example, phase locked monitors are not capable of controlling pulsed contingencies. Monitor H1 extends the phase locked class to include pulses (interrupts).

Phase locked monitors are restricted to contingencies for which $\beta_1 = \rho$ (that is, levels). Monitor H1 can control a number of applications which define contingencies for which $\beta_1 < \rho$ (pulses) while retaining the attributes of the phase locked class, such as extrinsic priority independence. H1 controls level contingencies just as P1 does, with a pre-defined schedule. Pulse contingencies are controlled by augmenting the

schedule so that each pulse contingency appears to behave as a level, and thus may be pre-scheduled with the level contingencies. The technique is based on realizing each pulse as an interrupt. All such interrupts are required to have the same intrinsic priority, therefore none will be ignored. Rather than initiating the task associated with the interrupt, however, a flag for the contingency is set by the monitor to be polled later by the phase locked schedule (such as the one defined for the PI monitor). In order to determine the feasibility of this approach to a particular application, additional timing tests must be made before PI global and local tests will be meaningful. Therefore a way of modeling interrupts as conditions is needed.

First, all pulse contingencies must have their parameters altered, to facilitate the phase locked timing tests. The time allowed to complete the test is set to the period, in order that it appear to be a level: $\beta_1 = \rho$. It is further necessary that β_2 be equal to ρ in the application definition. This is not a severe constraint, since it simply states that the contingency's task must be completed before the next pulse (interrupt) arrives. The contingency's maximum execution time, δ_1 (i.e., time to test for contingency), becomes the time needed to post the contingency's flag after an interrupt. Let ω_1 be the overhead to set the flag and λ_1 be the latency of interrupt response. Therefore, for events, δ_1 becomes $\omega_1 + \lambda_1$. Remember, however, that the contingency description (being procedural) may contain other operations besides acquiring environmental data. Therefore, the time needed to execute this remaining portion of the contingency is added to δ_2 (the time to perform the task). This is possible, since the task will only be executed when the contingency is true, and the contingency can only be true after an interrupt. The time to perform the contingency code (δ_1), altered to reflect data acquisition by interrupt, is called δ'_1 . (That is, references to data to be acquired by interrupt are realized by code which polls the contingency's flag: δ'_1 is δ_1 reduced or lengthened as needed to correspond to this recognition technique.) Thus, for events under hybrid timing tests, δ_2 becomes the sum of the original δ_2 and δ'_1 .

Consider, then, the time costs of such a situation. Let ρ' be equal to the longest repetition interval (longest ρ_i for all i). The time consumed in this interval by condition-oriented contingencies is:

$$\forall i | \beta_i = \rho_i, \quad \alpha_C = \sum \frac{\rho'}{\rho_i} (\delta_{1i} + \omega_{1i} + \delta_{2i} + \omega_{2i}) \quad , \quad [19]$$

the time consumed by events is:

$$\forall i \mid \beta_i < \rho_i, \quad \alpha_E = \sum \frac{\rho'_i}{\rho_i} (\delta'_{1i} + \omega_{1i} + \delta_{2i} + \omega_{2i}), \quad [20]$$

and the time required of worst case interrupt acknowledgement is:

$$\forall i \mid \beta_i < \rho_i, \quad \alpha_I = \sum \frac{\rho'_i}{\rho_i} (\omega_S + \lambda_I) = \sum \frac{\rho'_i}{\rho_i} \delta_{1i} \quad [21]$$

We may now define parameters required to perform the global and local timing tests defined for monitor P1 above. These are $\hat{\alpha}$ and $\hat{\rho}$. We therefore calculate them, adjusted for the presence of interrupts:

$$\hat{\alpha} = \alpha_C + \alpha_E \quad [22]$$

and

$$\hat{\rho} = \rho' - \alpha_I \quad [23]$$

These adjusted values, together with the altered contingency parameters, may be directly used to determine a P1 trial schedule, as discussed above. Furthermore, a new monitor called H2 may be analogously defined in which the adjusted value set is directly used to determine a P2 trial schedule. In fact, this approach may be used to define a hybrid counterpart to any phase locked monitor strategy.

5.5 OTHER MONITOR STRATEGIES

It is likely that improved monitor strategies and even wholly new monitor types will be developed in the future. Therefore, it is useful to consider integrating a new monitor into a design automation system. There are two classes of attributes involved: the necessary and the desirable.

It is necessary that each monitor strategy be accompanied by well defined criteria which, when applied to any application, will indicate whether the monitor can be adapted to the application. For example, the global and local timing tests associated with monitor P1 together specify such criteria for monitor P1. Once a strategy is determined to be applicable, an implementable method for adapting the monitor to the application must be available. The local timing test for monitor P2 satisfies this requirement for P2 since passing it guarantees that the trial schedule satisfying the test is valid and adapts the generalized phase locked monitor to the application. The feasibility criteria and adaptation method may be intertwined (as for P2) or defined separately.

Finally, certain monitor attributes could be deemed desirable, if not strictly necessary. We should like to employ monitors which display efficiency and compactness (two usually conflicting attributes). Also, the efficiency of performing feasibility tests and adapting the monitor is important in a system implementation. However, the most important (if not strictly required) attribute is that of generality.

Since an actual system must store each monitor and provide testing and adaptation of each type, it is desirable that each included monitor be widely applicable. Monitors which can rarely be adapted should obviously be avoided. However, there is a more elusive situation which must also be avoided. If there exists a monitor strategy which can adapt all applications adaptable by another monitor strategy, the first is said to temporally cover the second. Monitor P2 can control, as a subclass of all applications for which it is adaptable, all applications controllable by monitor P1. Therefore, P2 temporally covers P1, written $P1 \subseteq P2$, and shown schematically in Figure 5. It is also possible that several monitors might together cover some monitor: $M1 \subseteq M2 \cup M3$. It is a waste of resources to support a monitor which is temporally covered.

5.6 ORGANIZATION OF THE TIMING ANALYZER

The classification scheme thus developed divides monitor strategies into three basic types: phase locked, dynamic and hybrid. No attempt was made in this development to rank the types according to their usefulness, since applications can be postulated for each of the possible rankings. However, we will now orient the discussion toward a specific type of control application, the replacement of combinatorial logic, in order that it reflect current technology. Therefore, we will define a ranking of the monitor strategy classes based on two issues: the use of interrupts and the need for extrinsic priority.

Controllers based on current computer technology have two basic modes of accepting information from the environment: polling and interrupt. The processor controls the timing of communications in a polling environment; it decides when the controlled process is to be sampled. However, the opposite is true with an interrupt scheme. The environment controls the points in time when communication must occur. Polling is to be preferred over an interrupt strategy in our ranking for four reasons. First, interrupt methods usually impose much stricter timing constraints than do polling approaches. (Indeed, problems resulting from such tight timing requirements have motivated discussion of the abandonment of the interrupt in general purpose computers.) Second, the action of an interrupt closely resembles the execution of a random GOTO statement. The problems of allowing the GOTO have been widely discussed. Similar difficulties caused by a branch in program flow triggered by an unpredictable external stimulus can only be increased. Third, the interface hardware requirements for interrupts (due to their timing) can be highly variable. Polling approaches are more amenable to the standardization desirable in a system. Finally, most activities required of representative controllers have been slow when compared with the speed of the processor. The same situation is expected to hold true for microprocessors used for logic replacement.

Figure 5

$$P1 \subset P2$$

Any strategy which requires a specification of the relative importance of each contingency of a design is also to be avoided. Such extrinsic priority requirements are counter to the purpose of a controller. The applicability of the time-line approach showed that control has a concurrent nature. Extrinsic priorities would attempt to order that concurrency, a contradiction in terms. Therefore, we will include such strategies only after unordered methods have proven inadequate.

Consider now Appendix B, a timing analyzer oriented toward the class of problem represented by the example. Input to the analyzer is a functional realization of the application. Each of the three classifications is tested in turn, first for global acceptability and then in local terms. If no strategy is found that satisfies both the global and local timing needs of the application, a failure flag is set so that the next volume may be accessed. (Note that in each module shown to test global timing, there may be several actual strategies of that general type involved. For example, the hybrid global timing module would attempt to adapt all the strategies of the hybrid type defined for the current volume, such as H1, H2, and so on.)

The phase locked monitor classification is used first for attempted monitor adaptation. This class is ranked highest because it is a pure polling approach which does not require extrinsic priorities. Pure polling methods tend to require less specialized hardware. (Since they are pre-scheduled, they require no real time clock hardware, for example.) Polling monitors tend to be rather simple looping programs with low overhead. Finally, since their schedules do not depend on the unpredictability of the environment (being scheduled prior to execution) their performance can be guaranteed to conform to the behavior described for the application.

Hybrid monitor strategies are used for attempted monitor adaptation if a phase locked solution cannot be found. Hybrid strategies allow the use of interrupts in a very structured manner. These strategies force interrupts to post a flag in the monitor which is later polled by a phase locked strategy. This allows the system to meet timing requirements caused by events in the environment which occur too quickly for a straightforward polling approach. We therefore gain some of the advantages of the phase locked approach. For instance, no extrinsic priorities are necessary, and the use of phase locked polling of interrupt flags allows guaranteed pre-scheduling. However, the acquisition of interrupt signals may require special ancillary hardware and interfacing techniques. Any mix of polled or interrupting contingencies is allowed, including pure-event schedules, as long as the phase relationship of interrupt flag polling is maintained.

If there exist no phase locked or hybrid strategies adaptable to the application, a dynamic approach may be tried. Dynamic approaches depend on conditions that prevail from instant to instant in physical time. Therefore, they cannot be prescheduled and their capability to correctly control cannot be guaranteed (with known strategies). All contingencies are realized as interrupts, even those suited to polling. A physical-time clock is needed, as well as interrupt interface hardware. Another drawback is that (in the most common dynamic monitor strategies)

a knowledge of extrinsic priority is necessary. Fortunately, it is expected that the majority of applications will be well adapted to polling.

Methods are now available to allow discussion a complete control system design (CSD) automation facility.

6. A CSD SYSTEM

A basic CSD System is summarized in the flow chart of Appendix A. Consider each module of the system. A system to completely automate controller design requires a repository of information defining those elements from which controllers may be assembled. This source of software capabilities, hardware descriptions and monitor strategies is called the realization library. In our system, the library is shown as the input to a module called the librarian which produces a common repository of information (file) called the formatted library. System efficiency is increased by maintaining a version of the realization information formatted to be directly usable by those system modules which access the library.

Note that the librarian is shown as a logically separate component of the system. Library maintenance (that is, direct attention to realization) is not a part of the system seen by the user, since a major goal of the system is to keep it realization-independent as viewed by a user. It is intended that the library be maintained by personnel most knowledgeable in the capabilities of processors found to be best-suited to the problems addressed by users of the system. This also allows the maintenance of libraries oriented toward differing control disciplines, tailoring the generated controllers to suit their use.

6.1 PRODUCING A FUNCTIONAL CONTROLLER

The input to the system, supplied by the user, is a behavioral description of the controller expressed the Control System Design Language (CSDL). Such a description is the only input required of, or allowed from, the user of a CSD system. The remaining modules of the system have as their joint function the design of a complete controller which behaves as the user has indicated. The input is acquired and converted into the intermediate form by the input analyzer module. This module performs a lexical scan of the input, parses it according to CSDL syntax, builds a symbol table, and produces a stream of intermediate code which reflects the requirements of the input. We will not spend a great deal of time on the input analyzer because each of its duties has been widely treated in the literature. Lexical scan, parsing and symbol table production are commonly referred to as the "front-end" in compiler texts. The production of an n-tuple intermediate code for use in table driven compilers (or to direct fixed code generators) is also not new. Therefore, standard techniques may be used to generate software oriented primitives as intermediate code.

Before leaving the input analyzer, however, we must make note of two major extensions, unique to a CSD System, which must be made to

standard methods. We cannot determine whether a contingency or task will be realized as an interrupt or a polling-type until timing analysis has occurred. However, timing cannot be analyzed without maximum execution times having been determined, which requires the procedure to be realized as either an event or condition. The technique which most clearly deals with this dilemma is realization-pair generation. In the abstract, this means realizing each procedure as both an event and a condition. However, the model shows that the only primitives affected by changing realizations from one form to the other are the transmission primitives, sense and issue. Therefore, only one realization of each procedure need be generated, except that each transmission primitive must be defined twice, once as a condition and once as an event. This allows the timing analyzer to make its selection without regenerating the intermediate code.

The second extension to a standard compiler front-end needed for a CSD System is an application-timing table, similar in organization to a symbol table. In this table are stored the names of contingencies with their respective tasks (thereby relating them), and all directly and indirectly supplied timing values from the problem definition (ρ , β_1 , β_2 , γ_1 and γ_2). This table is used during timing analysis.

If no errors are detected during analysis of the input, the system has available a stream of machine-independent software primitives corresponding to the behavior of the described controller. At this point in the system, it is necessary to select a trial processor for a first attempt at realization. The order in which processors are selected is determined by the position of their defining volumes in the library. Each time a trial processor is found unsuitable, return to this point in the system is made so that the next volume (and therefore, the next processor) may be tried. Volumes should be ordered to reflect preferred processors; ordering may be based on economic, technical or other criteria.

Once a trial processor volume has been selected a trial generated code stream may be produced. This process returns the attributes of each primitive in the intermediate code stream; that is, the execution time and storage requirements associated with realizing the machine-independent description with the software of a particular processor are made available. Note that if the realization-pair approach is used, a dual set of such values is available (one for a condition realization and one for an event realization).

From the library, execution time figures are in terms of the total number of basic processor cycles required (usually a count of clock pulses) called TIME, and the total number of references to external sources which may have a different fundamental access time than the processor (such as accesses to slower memory) called EXT. These time figures reflect the abstract time required of the processor to execute each procedure. However, physical time constraints must be met in an actual controller. Therefore, the hardware (processor and memory) required to support the generated code must be bound. This is done by the hardware binder.

The hardware binder may generate only two primitives: H.PROCESSOR which returns the fundamental clock period (t_H) and H.MEMORY which returns the fundamental latency (if any) of the memory used (x_H).

Furthermore, each of the transmission primitives which generated input/output code also generated an "include" statement for any support hardware needed to interface the code to the controlled process. The binder, then, also has access to the latency figures for each transmission primitive of each procedure (λ).

Each of these time constants (t_H , x_H and λ) refers to actual hardware, and therefore defines periods of physical time needed by the timing analyzer to compute realization-dependent values, such as δ (the maximum execution time of a procedure). Consider how δ may be computed. The model indicates that the time required of each CSDL statement is necessary (that is, τ_H must be applied). Call this time, for an arbitrary statement s , t_s . Once s is realized on a particular processor, its primitive will supply the number of clock cycles (TIME) and external references (EXT) made. After the binder expands its processor and memory primitives, and also retrieves the necessary latency figures for any necessary support hardware (λ), the time to execute an arbitrary statement is given by the general formula:

$$t_s = \text{TIME} \cdot t_H + \text{EXT} \cdot x_H + \lambda. \quad [24]$$

Given the time required for each CSDL statement, and the bounds on loop constructions required of CSDL descriptions, known techniques [Knuth, 1968] may be used to find the maximum time-wise path through any procedure (corresponding to the max function of the model).

RELATING FUNCTIONALITY TO TIME

Successful completion of hardware binding has as its result a functionally acceptable realization of the CSDL description. The timing and execution characteristics thus made available are then used by the timing analyzer.

The timing analysis module has two distinct duties. It must first determine a method of realization for each procedure (that is, as a condition or event). This is accomplished by performing the feasibility tests (equations 5 through 12). These tests are based on information reflecting a realization with the current processor. This data is collected in a realization-timing table. This table must eventually be reconciled with behavioral needs expressed in the application-timing table. An initial attempt is made to realize each procedure as a condition. Any procedure which cannot pass its feasibility tests is then realized as an event. If any procedure still cannot pass the feasibility requirements, the entire controller is said to be infeasible with respect to

the current processor. This requires abandonment of the processor, and an attempt will be made with the next processor (that is, the next volume is accessed).

After a feasible functional realization has been determined, actual timing analysis can be performed. The purpose of timing analysis is the determination of a monitor strategy which can control execution of the generated software on the bound hardware in such a way that the given timing constraints are always met in the worst case. If a strategy cannot be found that is temporally acceptable in all cases, the trial processor under consideration cannot be made to behave according to the application. In this case, the next processor (volume) is accessed and used in a new trial realization.

Once a strategy is found that will serve, an actual monitor must be generated which is executable by the controller processor. The monitor generator must access the monitor skeletons of the current volume and produce all software and ancillary support hardware (such as an interrupt clock) needed to realize the correct strategy. We assume a table driven basic form for each monitor. Under this assumption, only three monitor primitives are needed: one to allow the monitor body to be defined, one to allow entries in the table to be made, and one to define the end of the table. Other formats are possible. The processor dependent monitor thus generated is a complete controller design.

The final module of the CSD System is the formatter. The task of automatically designing our controller has, by the time the formatter is invoked, been done. However, the output (the design) must still be used to direct (whether manually or automatically) the construction and programming of the actual controller. Therefore, the formatter is included to organize the design in the most useful format and actually expand library macros.

Returned attributes supply all necessary information for the design process, therefore there is no need to carry along expanded versions of the library macros during design evaluation. The design is maintained in intermediate code format for convenience; changes to the developing design thus take the form of insertions and deletions performed on the intermediate code. This also allows the binding times of system parameters (such as total memory) to be delayed, for use in actual macro expansion. The formatter also directs the insertion of comments from the CSDL input in the output to ease relating the abstract requirement to its realization and to provide further documentation. The final design consists of a software listing and a hardware configuration listing.

6.2 AN EXAMPLE OF AUTOMATIC CONTROLLER DESIGN

In order to demonstrate the use of a CSD system, a three-contingency example will be traced through the design process, culminating in a complete controller specification. The example (Appendix D) is neither simple nor complex; its contingencies were chosen to be representative of practical control situations while allowing the basic operation of

the system to be observed. Appendix E is a realization library sufficient to allow automated design of the example. A single representative volume (for the Intel 8080 microprocessor) forms the library. This library is assumed to be available to the CSD System.

6.2.1 INPUT ANALYSIS

The first action of the system is the acquisition of the controller description by the input analyzer. The identification section is retained for output documentation. The environment section is processed to obtain symbol table entries for global variables. Associated with each of these identifiers is its type, organization, precision and technology (see the Environment Section of CSDL).

The next section read must be the contingency list. The information in this list is used to build the application-timing table, shown in Table 1. This table is maintained in rate monotonic order (that is, from most critical to least critical timing requirements). Therefore, time-lines are listed in order of ascending values of ρ . The first two elements of the table are the identifiers of each contingency and its task, to relate them. The values of ρ , β_1 , β_2 , η and π follow. Note that default values have been assigned to several of these parameters. η and π are equal throughout since the application requires three concurrent time lines. There were no timed blocks in any of the contingencies (γ_i), therefore their values are zero. However, there is a timed block requiring the output pulse of the task labeled Serialize (γ_{21}) to be completed within 500 microseconds. The final column is the calculated value of ϵ_i (equation 17) needed by the timing analyzer. The figures of Table 1 constitute the timing requirements which must be met by any controller realization produced by the system.

The last section of the design description is the procedures section. It should be remembered that there need be no procedures section included. All contingencies and tasks not defined in this last section are assumed resident in a general procedure section of the library. These procedures will be used if not overridden by those supplied with the application. Therefore, it is possible for all procedures to be library resident. For the purposes of this example, however, all task and contingency specifications are found in the procedures section.

It is from the information in the procedures section that the input analyzer produces the intermediate code stream used to generate realizations. Appendix F shows the intermediate code stream as it appears after timing analysis, with all generated and included primitive and subroutine invocations. Consider the stream generated by the input analyzer. First, only software primitives (beginning S.) are generated directly by the input analyzer, although some of these primitives cause hardware primitives to be included in the stream (this is done mainly by transmission primitives requiring interface hardware). Second, all transmission primitives (SENSE and ISSUE) are generated in pairs, so the timing

Time* Line	P_{1i}	P_{2i}	ρ_i	β_{1i}	β_{2i}	γ_{1i}	γ_{2i}	η_i	π_i	ϵ_i
i=1	Data ready	Serialize	1.3 μ s	1.3 s	1.3 μ s	0	500 μ s	1	1	3
i=2	Alarm	Alert	2 μ s	50 s	2 μ s	0	0	1	1	2
i=3	Every 5 μ s	Encode	4 μ s	4 s	4 μ s	0	0	1	1	1

*Rate Monotonic Order

Table 1
Application-Timing Table

analyzer may choose the best realization method (condition or event). For example, the input (sense) primitive generated in contingency Alarm is shown realized as an event (S.SENSEVENT). However, before the timing analyzer chose this realization method, it needed a realization of both condition and event methods. Therefore the input analyzer originally generated a condition-oriented sense primitive (S.SENSECOND) and an event-oriented sense primitive (S.SENSEVENT), from which the choice is made. The timing analyzer expects every conditional transmission primitive to be directly followed by a primitive for an event implementation.

The first primitive always generated by the input analyzer is S.MAIN, which can be used for initialization (or be empty). The volume one (8080) definition of S.MAIN calls a subroutine named S.HEADING to define the software listing heading (demonstrating the CALL library capability). Following the system primitives are the primitives which define the software for each of the contingencies and tasks, output in the order they appear in the procedures section. The exception to this is the primitive to realize the contingency requesting the task called Encode. The every-period statement causes a built-in primitive to be generated which acts as the contingency code for the task. In essence, it makes the contingency always true for the task, and the timing analyzer must then schedule it to meet its period requirements.

After the primitives of the last procedure are generated, the input analyzer generates global memory allocation instructions. Also note that the last few hardware primitives in Appendix F were generated by INCL (include) statements in the various transmission primitives to define interface hardware. The values, such as precision, found in the selection list are supplied by the symbol table. The attribute list of each realized primitive appears on the left. These values are used to determine values needed in calculating execution time parameters needed by the timing analyzer.

6.2.2 HARDWARE BINDING

After the software is generated, the actual binding of hardware is performed. The binder always generates two hardware primitives, H.PROCESSOR and H.MEMORY. The processor primitive must generate a processor description defining signal sets for each of its connection pins and any other support hardware needed (such as the clock generator and the bus driver circuits). It must also return a value for the fundamental clock period used (t_H). The memory primitive must generate sufficient storage for the software and return a value for external access latency of the memory used (x_H). A memory type slower than the 8080 processor is purposely used in the library to supply a non-zero external delay value for the example. The delay is implemented in the hardware by including in the memory specification a device which causes the processor to idle one clock period, allowing the slower memory to settle. The latency figure (x_H) thus allows the time calculated for each primitive operation (by equation 24) to reflect this necessary hardware delay. A zero

value of x_H would have indicated a memory technology capable of supplying information at the same rate the processor requests it. The values returned, in the example, for processor clock period and external latency are both 500 nanoseconds.

The binder also retrieves latency values of interface hardware primitives included by earlier software primitives (for example S.SENSEVENT caused the inclusion of H.SENSEVENT). Since such included primitives may call or include others, the binder must also cause these new primitives or subroutines to be added to the intermediate code stream, and collect their latency figures (if any). This is illustrated by the inclusion of S.EVENTMARK by H.SENSEVENT, which specifies interrupt hardware and returns the event-oriented latency (that is, the time required to process an interrupt). H.SENSEVENT then causes S.EVENTMARK to be included which defines the "trap" software for the event (interrupt). This complete definition of an 8080 interrupt method by the single system primitive S.SENSEVENT (eventually causing inclusion of the interface hardware and trap processing software) is an especially good example of the power of delayed hardware binding. The necessity of generating software for fixed hardware (which has made compilers untransportable) is obviated. Automatic hardware binding allows hardware and software to be defined together, promoting flexibility. This approach makes possible the use of a single set of realization-independent primitives, leaving definition of the actual realization methods to the library writer.

Note finally that each included or called hardware specification returns a power figure, used to calculate total power requirements of the controller for power supply analysis; the controller produced in the example consumes 6.5 watts.

6.2.3 TIMING ANALYSIS

Now that a functionally congruent controller design is available, the determination of an adapted monitor may be attempted. This is accomplished by the timing analyzer. The timing analyzer has two primary tasks, determination of transmission mode (event or condition) for each contingency and task, and adaptation of a monitor strategy to the application. Recall that functional analysis modules did not have sufficient information to determine transmission mode and therefore used the realization-pair technique which supplies two realizations of each transmission primitive, one oriented toward events, the other toward conditions. The system must determine which approach to use, since only one can appear in the final design. As shown above, it is desirable to realize as many transmission primitives as possible as conditions, rather than events. Therefore, the approach will be to realize all such primitives as conditions and determine whether or not a completely condition-oriented solution is feasible. If any of the primitives cause an infeasible solution, those are then realized as events and the tests for feasibility are again applied. If the solution is again infeasible, the current processor cannot be used, and the next library volume must be accessed. The information for these feasibility tests is collected in the available application-timing table, and in the realization-timing table.

A realization-timing table for the example with all of its transmission primitives realized as conditions, appears in Table 2. The rows of this table correspond (by time-line) to the rows of the application-timing table (Table 1). The columns define parameters needed to perform the feasibility tests. The total execution time of the contingency (δ_1) and its task (δ_2) are calculated by equation 1 for the current processor using the returned attributes of the primitives involved and maximum time-path analysis. The actual times to execute timed blocks (σ_1 for contingencies, σ_2 for tasks) are easily obtained as subpaths of δ_1 and δ_2 . Monitor overhead ω (equal to the sum of ω_1 and ω_2) is shown for the P2 monitor in the library example (this figure changes as various monitors are used for trial adaptation). The total time required for a worst case instance of a contingency/task pair is included for the reader's convenience (this is the sum of δ_1 , δ_2 and ω). The latency figures for interface hardware (both for the entire contingency or task and for embedded time blocks) are next. These figures are incorporated in each associated δ value. Finally, the storage required for each contingency and task for the current trial realization is included.

The information calculated to build this table, coupled with the data in the application-timing table, are sufficient to perform the feasibility tests. These tests are defined by equations 6 through 12. In performing the calculations we find that all of these tests are satisfied except one. The contingency of the second time-line (Alarm) fails to meet the criterion of equation 6; the time to execute this contingency (125.5 microseconds) is greater than the time allowed by the application description (50 microseconds). This single test failure makes the pure-polling approach reflected in Table 2 infeasible.

The next step is the alteration of the communication mode of the infeasible contingency to an event type (all others remaining condition types). This means that this contingency will be realized as an interrupt. The fact that, of two nearly identical CSDL behavioral descriptions (Alarm and Dataready), one is realized as an event and one as a condition, illustrates the impact of differing timing requirements. Values calculated for such a realization appear in the altered realization-timing table seen in Table 3. Again using the feasibility equations, we find that this table does reflect a feasible realization. It remains to adapt a suitable monitor to this realization.

The order of trial monitor adaptations follows Figure 2. The phase locked strategy is used first. However, since phase locked strategies are of a pure-polling nature, none of them can be adapted to our application since the realization now includes an interrupt. This is detected in the phase locked global timing analysis which then causes the next strategy type to be tried; this is the hybrid strategy type.

The approach used in hybrid monitors consists of forcing each interrupt to post a flag (meeting the timing constraint of the signal), which is then examined in a polled manner. The polling procedure used

Time Line	δ_{1i}	δ_{2i}	σ_{1i}	σ_{2i}	ω	Total	λ_{1i}	λ_{2i}	λ_{1i1}	λ_{2i1}	Store ₁	Store ₂
i=1	129.5	444	0	343	75.5	639	.5	1	0	1	46	87
i=2	125.5	156	0	0	75.5	357.5	.5	1	0	0	47	58
i=3	21.5	336	0	0	75.5	423	0	1	0	0	7	125

All time values in microseconds
Storage in basic Intel 8080 units (8-bit bytes)

Table 2
Infeasible Realization-Timing Table

Time Line	δ_{1i}	δ_{2i}	σ_{1i}	σ_{2i}	ω	Total	λ_{1i}	λ_{2i}	λ_{1i1}	λ_{2i1}	Store ₁	Store ₂
i=1	129.5	444	0	343	75.5	639	.5	1	0	1	46	87
i=2*	29.5	297	0	0	75.5	402	4	1	0	0	42	58
i=3	21.5	336	0	0	75.5	423	0	1	0	0	7	125

All time values in microseconds

Storage in basic Intel 8080 units (8-bit bytes)

*Altered approach

Table 3

Feasible Realization-Timing Table

is the phase locked method, allowing the same CSD system modules to be used for trial adaptation of both phase locked and hybrid monitors. The simplest approach to common use of the phase locked tests is redefinition of several key hybrid parameters which are then supplied to the phase locked test modules as though they were originally of that type. This transformation follows the method outlined for hybrid strategies. (Note that altering these parameters for timing analysis does not affect the hardware and software modes selected during feasibility testing; it is the returned attributes of the realization which supply the data used for alteration.)

Consider the parameter alterations needed. First, for contingencies to be realized as events (interrupts in the case of current processor technology), the time to complete the contingency (β_1) is set equal to the period (ρ), making the contingency appear to be a condition. Each affected contingency's execution time (δ_1) becomes the time needed to respond to, and post, the occurrence of its interrupt. This is equal to the sum of the overhead to set the flag (ω_1) and the latency of interface hardware realizing the interrupt (λ_1). These values are supplied as attributes of the primitives S.EVENTMARK ($\omega_1 = 25.5$ microseconds, after application of equation 6.1) and H.SENSEVENT ($\lambda_1 = 4$ microseconds). Therefore, δ_{12} becomes 29.5 microseconds. The time to execute the procedure defining the contingency (less the time, δ'_1 , needed to access the environmental data since this is done by the interrupt) added to the time needed to perform the task (δ_2) becomes the new δ_2 . Since software is already generated to perform the contingency, δ_2 is simply the sum of the current δ_2 and δ_1 , or 297 microseconds. The contingency and task codes are combined since the task cannot be activated without the interrupt occurring. Finally, $\hat{\rho}$ (the longest ρ) must be reduced by α_1 (the total time spent responding to interrupts in the worst case). Since there is only one time-line realized as an interrupt, we compute (by equation 3.22) $\alpha_1 = 4 + 2 \cdot 29.5 = 59$ microseconds. This gives $\hat{\rho}$ a value of 3941 microseconds. $\hat{\alpha}$ is then calculated (by equation 22) to be 3135 microseconds. Given these values, we may now attempt to generate a phase locked monitor.

The first phase locked monitor strategy discussed was P1. However, we also saw in that chapter that P1 $\subset$ P2, therefore P1 is not included in the library, making the P2 strategy the first one tried. The global timing criterion for P2 is the same as for P1, $\hat{\alpha}$ must not be greater than $\hat{\rho}$ (equation 16), meaning simply that the time required to execute contingencies and tasks in the worst case cannot exceed the time allowed for their execution. This requirement is met by the values of $\hat{\rho}$ and $\hat{\alpha}$ calculated above.

An attempt to meet local timing criteria may be made now that the global P2 timing criterion is satisfied. This is done by incrementally

generating a trial schedule by adding the required number of contingency/task pairs (ϵ_i) to a growing composite time line. We know from the global test that the sum of these instances will be less than or equal to $\hat{p}$. As each instance is added to the composite, we must verify that no pair is more than its p_i value from another instance of the pair.

The required number of instances is added in rate monotonic order (shown in Figure 1). (This search method is used here only because it so clearly demonstrates the necessity of maintaining acceptable time distances for each instance of each pair in the worst case cycle.) Figure 6 shows a feasible P2 worst-case schedule for our example application. The example was formulated to cause the trial schedule to closely resemble the P2 example schedule (Figure 4) allowing the reader to more easily relate the method to its use.

6.2.4 MONITOR GENERATION

Production of a successful trial schedule not only gives evidence that a P2 monitor strategy will serve, it also supplies a schedule to use with such a strategy. As indicated by Figure 1, monitor generation may now begin.

The monitor generator may generate software primitives to define the three parts of a table-driven monitor: the monitor body (S.MONITOR), table entries (S.TABENT), and a table ending (S.TABEND). The monitor body to be generated is the P2 strategy of volume one (that is, the P2 monitor expressly designed for use with an 8080 processor). It should be remembered that we actually performed processing for the H2 monitor strategy, but intentional similarity allows the library to contain only phase locked monitors (thus shortening it). Substitution of a phase locked strategy for a corresponding hybrid strategy is possible because the interrupt transmission primitives that are generated supply all hardware and software needed to cause simple posting of an event mark to be polled later. Therefore, a pure polling strategy may be used for H2, and the one chosen is the already available phase locked approach, P2.

After the monitor body is defined, a primitive invocation is generated in the intermediate code stream for each instance of the P2 schedule determined by the timing analyzer. This is done by simply generating an S.TABENT primitive (whose arguments are the contingency and task identifiers) for each pair in the schedule, moving left to right (that is, in the direction of increasing time). Finally, a table end primitive is generated to mark the end of the schedule. Notice that the volume one P2 S.TABENT primitive generates a subsidiary code section named S.TABACC which contains actual access code, allowing the repeated table entries to use as little storage as possible. Any of these primitives could have generated hardware primitives to include support hardware, although phase locked strategies do not generally require it. (Dynamic strategies, not needed for the example controller, usually require a count-down clock, for example.) All primitives finally generated to realize the example are shown in Appendix F.

DATAREADY/SERIALIZE $\rho_1 = 1.3$ milliseconds

ALARM/ALERT $\rho_2 = 2$ milliseconds

"EVERY"/ENCODE $\rho_3 = 4$ milliseconds

time frame $\leq \hat{\rho} = \rho_3 - \alpha_1 = 3974$ microseconds

Figure 6
A Feasible P2 Schedule

6.2.5 FORMATTING THE OUTPUT

Output formatting is performed by the system, after a systemically congruent design is defined by the intermediate code stream (Appendix F). The output formatter orders the contents of the intermediate code stream, software followed by hardware, according to desired documentation standards (such as monitor code following procedure-code, for instance). It then expands the STEXT and HTEXT blocks indicated by the primitives. As expansion continues, comments and other documentary text are interspersed in the output from associated CSDL input, producing continuity between the behavior description and the realization prescription.

The last appendices form an overview of the design automation process. The CSDL description (Appendix D) causes generation of intermediate code (Appendix F) from information in the realization library (Appendix E). This intermediate code is then expanded from the library, resulting in a complete control program listing (Appendix G), followed by a configuration listing of all hardware needed to execute the software and interface the processor (Appendix H).

7. APPLICATION AREAS

Although the major use of dedicated real time control systems has been in process control, many diverse areas of computer science could also benefit from a hard real time approach to traditional problems. Brief examples from operating system design, computer architecture and logic design indicate the applicability of these ideas.

The technique of layering in operating system design, from an outer (user-directed) layer to an inner (machine-directed) layer, has many advocates. It has been noted that to work cleanly, the layers must be partitioned dynamically as well as functionally. The lowest layers in an operating system contain many more functions relating to physical timing constraints than do outer layers. Lower levels have been characterized by the short term nature of the policies which guide their implementation; duration of actions at these levels are usually discussed quantitatively in terms of the speed of the host hardware. The functions of these lower levels lend themselves to representation as hard real time control sequences.

The availability of microprocessors for dedication to selected support (lowest layer) functions of an operating system would have a great impact on the organization of the remainder of the system. The elimination of expressly programmed multifunction scheduling and timing could enhance transportability. An essentially machine independent operating system design could instead rely on "black box" control subsystems to mate the upper levels to the host hardware.

In the area of computer architecture, microprocessors are being aggregated to form distributed processor computing systems called multi-micro-processors. Two important problems which must be solved when using this approach are the production of local control programs to

synchronize the system, and the configuration of each node (consisting of microprocessor hardware) so that intercommunication is possible. The collection of valid interactions allowed in the aggregation may be viewed as a communication protocol observed by each node. A hard real time control approach to the management of networks of processors appears to be very appropriate.

While the effects of microprocessors on conventional process control, operating systems and computer architecture are evolutionary (being an extension of the use of computers in a more dedicated way), their effect on logic designers is revolutionary. The basis for the rapid acceptance of the microprocessor as a logic design element is economic. Many different logic functions can be simulated by a general purpose computer--making it, in essence, a universal logic element capable of performing a number of functions simultaneously. A universal logic element produced in high-volume can economically replace many lower volume special purpose items.

The universality of microprocessors in logic replacement stems from their adaptability. However, they must be programmed before they can simulate any specific logic element they are to replace. The necessity of producing a program has often cost the designer more than was gained in flexibility. Much of the difficulty in using microprocessors is caused by viewing them as small computers. This causes designers to design around the processor rather than design with it, another legacy of the high cost era of computational hardware.

A way to avoid concentration of effort on the intricacies of the processor at the expense of other parts of the design, is to consider the microprocessor a "black box" universal element. This could be done by specifying the inputs, outputs, timing and external function of the element. This view can be accommodated by approaching the universal logic design element as if it were a dedicated real time controller.

Each of these applications is based on the same concept: automatically configuring computational systems to behave as needed without requiring the user to understand the ultimate realization. It is a step forward in broadening the use of computers, allowing them to be removed from the realm of the computer expert and applied in other fields.

7.1 POTENTIAL IMPACT

The trend toward automation in system design is steadily increasing, prompting consideration of the role that the CSD approach might have in the future. For instance, complete realization automation allows collection of detailed statistics reflecting the frequency with which various processors and interfacing schemes are used. Such information could be used to specify new hardware and software modules, better suited to the kinds of controllers commonly generated. This, then, is a form of optimization which tends to make library resident capabilities conform more closely to those most often needed. Utilization statistics could also promote commonality of realization for diverse applications. For

example, the use of standard bus conventions would allow standardized documentation and test procedures, reducing maintenance time and product cost.

A common goal of design automation is the complete automatic production of a device to fulfill an expressed need. A first step toward this goal is the coupling of a CSD system with a conventional design automation system, oriented toward automatic placement and routing of electronic components on printed circuits boards (DASLL, for example [Smith and Matelan, 1975]). Such a combination would provide not only software and hardware descriptions, but also the finished product: a functioning controller from a CSDL behavioral description.

Finally, the concept of delayed hardware binding can be extended to encompass general purpose computing system design. Rather than producing a dedicated hardware configuration, a CSD system could produce a configuration specification consisting of switching commands to define the organization of a form of "skeleton" processing system. A skeleton processor may take the form of a very basic machine which has access to a pool of computation resources, such as memories and logic units. These resources may be configured automatically through "cross-bar" switch closures to "flesh out" the skeletal processor. The techniques of this paper could be used to generate code appropriate to a given problem, and then generate switching orders (analogous to a hardware configuration) to make the skeleton processor appear specifically oriented toward that particular specialized code. This suggests in turn a general methodology for dynamically altering the computational features of a processor to best suit the current task (for example a business application followed by a scientific one) [Matelan, February 1975]. Finally, the use of timing ranges rather than the strict timing bounds discussed in this paper would allow the CSD approach to address statistically modeled control situations. Methods of specifying and operating on "fuzzy sets" could be integrated into the model, allowing upper and lower bounds to define worst case timing.

8. IMPLEMENTATION PLANNING

There does not exist, at present, a CSD system; the example was manually processed. Therefore, each module seen in Appendices A and B must be designed and programmed before actual controller automation can be achieved. This section discusses the anticipated level of effort required to obtain a CSD system.

A general purpose design automation system at Lawrence Livermore Laboratory, DASLL, can serve as a development environment for a CSD system [Smith and Matelan, 1975]. Several of its modules and its overall design make it well suited to CSD development. Furthermore, the developed system could remain a part of the DASLL production system for later use, or be easily partitioned as a stand-alone system. Therefore, realization of the proposed facility will be discussed in terms of an initial DASLL implementation.

Figure 7 illustrates the structure of one possible DASLL-based implementation. The DASLL supervisor module is used to control submodule activation and inter-module communication. The DASLL user interface and error handling modules simplify CSDL data acquisition. However, the most useful DASLL module is the general purpose data base and its management code [Matelan and Smith, 1975]. The data base structure may easily support the realization library information. The remaining modules must be developed.

Table 4 gives for each module two estimated levels of development needed for two distinctly different CSD systems. (Levels of effort are given in approximate documented card images for a standard high level language, such as ANS FORTRAN IV.) The first column reflects effort required to obtain a CSD system embodying the capabilities discussed in this proposal. Such a system, while capable of useful work, would be essentially a developmental one. It is estimated that such a system would require 24 to 30 man-months of effort by experienced senior software engineers.

The second column reflects a production quality CSD system capable of automating controller design for all but the most time-wise demanding applications. Such a system could be built with the developmental system as a basis, and might require approximately 48 to 60 additional man-months of effort.

Figure 7

Structural Overview of a DASLL Implementation of the CSD System

	DEVELOPMENTAL	PRODUCTION
Supervisor	available	available
User Interface	available	available
Librarian	1000	2000
Input Analyzer	1500	2500
Software Generator	1000	1500
Hardware Binder	1000	2000
Timing Analyzer	2000	5000
Monitor Generator	500	1000
Output Formatter	500	500
Optimizer	not required	4000

Table 4
Levels of Effort

REFERENCES

- Intel Corporation, Intel 8080 Microcomputer Systems User's Guide, Santa Clara, California (September 1975).
- M. N. Matelan, "A Compiler-compiler System for Avionic Test Languages," Proceedings of ASSC 73 Conference, IEEE Aerospace and Electronic Systems Society, Arlington, Texas (November 1973) pp. 17-24.
- M. N. Matelan, "DELTA - The Uses of Microprocessors in the Distributed Control of Electronic Testing," Joint Conference on 'Advances in Automatic Testing Technology,' University of Birmingham, Birmingham, England (February 1975), pp. 19-26.
- M. N. Matelan, "MPACT...Microprocessor Application to Control-Firmware Translator," ACM Special Interest Group on Design Automation Newsletter, Vol. 5, No. 1 (March 1975) pp. 13-41.
- M. N. Matelan, "Parallelism in Automatic Testing," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, (August 1975) pp. 100-104.
- M. N. Matelan, "A Data Base Design for Digital Design Automation," Proceedings of the Workshop on Data-Bases for Interactive Design, University of Waterloo, Ontario, Canada (September 1975; with R. J. Smith).
- M. N. Matelan, "Automating the Design of Microprocessor-based Real Time Control Systems," Proceedings of the 13th Design Automation Conference, ACM-IEEE (June 1976) pp. 462-469.
- M. N. Matelan, "A Model of Real Time Control System Production," ACM Special Interest Group on Design Automation, Vol. 6, No. 2 (June 1976).
- M. N. Matelan, Automating the Design of Dedicated Real Time Control Systems, UCRL-78651, Lawrence Livermore Laboratory (August 21, 1976).
- R. J. Smith, II and M. N. Matelan, "Software Organization for a Multi-purpose, Digital, Design Automation System," Proceeding of the Fall IEEE Computer Conference, Washington, D. C. (1975) pp. 253-255.

APPENDICES

APPENDIX A
A CSD SYSTEM

APPENDIX B
TIMING ANALYZER

APPENDIX C
FORMAL SYNTAX OF CSDL

SYMBOLS

001. <empty>::=<the null string of symbols>
002. <letter>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N
|O|P|Q|R|S|T|U|V|W|X|Y|Z
003. <decimal-digit>::=0|1|2|3|4|5|6|7|8|9
004. <binary-digit>::=0|1
005. <logical-value>::=<binary-digit>
006. <underscore>::=_
007. <character>::=<letter>|<decimal-digit>|<underscore>

STRINGS

008. <identifier>::=<letter>|<identifier><letter>
|<identifier><digit>|<identifier><underscore>
009. <string>::=<any sequence of symbols not containing ' or ">
|<empty>
010. <binary-integer>::=<binary-digit>
|<binary-digit><binary-integer>
011. <decimal-integer>::=<decimal-digit>
|<decimal-digit><decimal-integer>
012. <unsigned-number>::=<decimal-integer>
|<decimal-integer>.<decimal-integer>|<binary-integer>B
|<binary-integer>.<binary-integer>B
013. <number>::=<unsigned-number>|+<unsigned-number>
|-<unsigned-number>
014. <comment>::="<string>"

OPERATORS

015. <arithmetic-operator>::=+|-|*|÷|↑
016. <relational-operator>::=≤|=|≥|≠|>|<
017. <logical-operator>::=∨|∧|¬|⊃|≡

VARIABLES

- 018. <variable-identifier>::=<identifier>
- 019. <simple-variable>::=<variable-identifier>
- 020. <subscript-expression>::=<arithmetic-expression>
- 021. <subscript-list>::=<subscript-expression>
|<subscript-list>,<subscript-expression>
- 022. <array-identifier>::=<identifier>
- 023. <subscripted-variable>::=<array-identifier>[<subscript-list>]
- 024. <variable>::=<simple-variable>|<subscripted-variable>
|<variable>(<bit-field>)
- 025. <bit-field>::=<decimal-integer>|<decimal integer>:<decimal-integer>
- 026. <label>::=<identifier>

ARITHMETIC EXPRESSIONS

- 027. <adding-operator>::=+|-
- 028. <multiplying-operator>::=*|÷
- 029. <primary>::=<unsigned-number>|<variable>
|<function-designator>|(<arithmetic-expression>)
- 030. <factor>::=<primary>|<factor>↑<primary>
- 031. <term>::=<factor>|<term><multiplying-operator><factor>
- 032. <arithmetic-expression>::=<term>|<adding-operator><term>
|<arithmetic-expression><adding-operator><term>

LOGICAL EXPRESSIONS

- 033. <relation>::=<arithmetic-expression><relational operator>
|<arithmetic expression>
- 034. <Boolean-primary>::=<logical-value>|<variable>
|<function-designator>|<relation>
|(<Boolean-expression>)
- 035. <Boolean secondary>::=<Boolean-primary>
|¬<Boolean-primary>
- 036. <Boolean-factor>::=<Boolean-secondary>
|<Boolean-factor>^<Boolean-secondary>

037. <Boolean-term>::=<Boolean-factor>
|<Boolean-term>v<Boolean-factor>
038. <implication>::=<Boolean-term>
|<implication>D<Boolean-term>
039. <Boolean-expression>::=<implication>
|<Boolean-expression>E<implication>

CONTROL STATEMENTS

040. <if-then>::=IF<Boolean-expression>THEN<statement-group>FI
041. <while-do>::=WHILE<Boolean-expression>:<maximum-loop-count>
DO<statement-group>OD
042. <for-loop>::=FOR<index>FROM<arithmetic-expression>
TO<arithmetic-expression>:<maximum-loop-count>
DO<statement-group>OD
043. <exit>::=EXIT<label>;|EXIT<proc-identifier>
044. <perform-task>::=PERFORM<task-designator>
045. <maximum-loop-count>::=<decimal-integer>
046. <index>::=<simple-variable>
047. <control-statement>::=<if-then>|<while-do>
|<for-loop>|<exit>|<perform-task>

ACTION STATEMENTS

048. <left-part-list>::=<variable>x|<left-part-list><variable>x
049. <assignment-statement>::=<left-part-list><arithmetic-expression>
|<left-part-list><Boolean-expression>
050. <data-input>::=SENSE(<variable>)
051. <data-output>::=ISSUE(<variable>)
052. <action-statement>::=<assignment-statement>
|<data-input>|<data-output>

PROCEDURE TIMING

053. <time-measure>::=H|S|MS|US|NS
054. <period>::=<decimal-integer><time-measure>

- 055. <time>::=<time-list><time-measure>
- 056. <time-list>::=<decimal-integer>|<decimal-integer>.<time-list>
- 057. <timed-block>::=IN<period>:<statement-group>NI
- 058. <wait>::=WAIT<period>|WAIT<arithmetic-expression>:<period>
- 059. <wait-until>::=WAIT UNTIL<Boolean-expression>:<period>
- 060. <proc-timing-statement>::=<timed-block>|<wait>|<wait-until>

PROCEDURE STRUCTURE

- 061. <statement>::=<control-statement>|<action-statement>
|<proc-timing-statement>|<label>:<statement>
- 062. <statement-group>::=<statement>|<statement>;<statement-group>

DECLARATIONS

- 063. <proc-declaration>::=<binary-spec>|<arithmetic-spec>
|<test-spec>|<code-spec>
- 064. <proc-declaration-group>::=<proc-declaration>
|<proc-declaration><proc-declaration-group>
- 065. <declaration>::=<proc-declaration>|<input-spec>
|<output-spec>|<duplex-spec>
- 066. <declaration-group>::=<declaration>
|<declaration><declaration group>
- 067. <duplex-spec>::=DUPLICATE<transmission-body>;
- 068. <binary-spec>::=BINARY<binary-body>;
- 069. <arithmetic-spec>::=ARITHMETIC<arithmetic-body>;
- 070. <transmission-body>::=<transmission-decl>
|<transmission-decl>:<transmission-body>
- 071. <transmission-decl>::=<identifier>,<binary precision>,<technology>
- 072. <binary body>::=<binary-decl>|<binary-decl>:<binary-body>
- 073. <binary-decl>::=<identifier><structure><binary precision>,<initial-value>
- 074. <arithmetic-body>::=<arithmetic-decl>
|<arithmetic-decl>:<arithmetic-body>
- 075. <arithmetic-decl>::=<identifier><structure><decimal-precision>,
<initial-value>

- 076. <structure>::=(<decimal-integer>)|(<decimal-integer>,<decimal-integer>)
|<empty>
- 077. <binary precision>::=<decimal-integer>
- 078. <decimal precision>::=<decimal-integer>
|<decimal-integer>.<decimal-integer>
- 079. <initial-value>::=<number>
- 080. <technology>::=TTL|ECL|IIL
- 081. <text-spec>::=TEXT<code-id>,<text-body>
- 082. <text-body>::=<text-decl>|<text-decl><text-body>
- 083. <text-decl>::=<identifier><structure>'string';
- 084. <code-spec>::=CODE<code-id>,<binary-precision>,<code-body>
- 085. <code-body>::=<character-representation>
|<character-representation>,<code-body>
- 086. <character-representation>::=<character>:<binary-integer>
- 087. <code-id>::=<identifier>|<predefined-codes>
- 088. <predefined-codes>::=ASCII6|ASCII7|EBCDIC|BCD

PROCEDURE DESIGNATORS

- 089. <procedure-identifier>::=<identifier>
- 090. <actual-input-parameter>::=<string>|<arithmetic-expression>
|<logical-expression>|<procedure-identifier>|<empty>
- 091. <actual-output-parameter>::=<variable>
- 092. <actual-input-list>::=<actual-input-parameter>
|<actual-input-parameter>,<actual-input-list>
- 093. <actual-output-list>::=<actual-output-parameter>
|<actual-output-parameter>,<actual-output-list>
- 094. <procedure-designator>::=<procedure-identifier>
|<procedure identifier>(<actual-input-list>:
<actual-output-list>)
- 095. <function-designator>::=<procedure-designator>
- 096. <task-designator>::=<procedure-designator>

PROCEDURES SECTION

097. $\langle \text{formal-input-parameter} \rangle ::= \langle \text{simple-variable} \rangle$
098. $\langle \text{formal-output-parameter} \rangle ::= \langle \text{simple-variable} \rangle$
099. $\langle \text{formal-input-list} \rangle ::= \langle \text{formal-input-parameter} \rangle$
| $\langle \text{format input-parameter} \rangle, \langle \text{formal-input-list} \rangle$ | $\langle \text{empty} \rangle$
100. $\langle \text{formal-output-list} \rangle ::= \langle \text{formal-output-parameter} \rangle$
| $\langle \text{formal-output-parameter} \rangle, \langle \text{formal-output-list} \rangle$ | $\langle \text{empty} \rangle$
101. $\langle \text{formal-parameter-list} \rangle ::= (\langle \text{formal-input-list} \rangle : \langle \text{formal-output-list} \rangle)$
| $\langle \text{empty} \rangle$
102. $\langle \text{procedure} \rangle ::= \langle \text{task} \rangle | \langle \text{function} \rangle$
103. $\langle \text{task} \rangle ::= \text{TASK} \langle \text{procedure-identifier} \rangle \langle \text{formal-parameter-list} \rangle;$
 $\langle \text{proc-declaration-body} \rangle \langle \text{statement-group} \rangle;$
 $\text{EXIT} \langle \text{procedure-identifier} \rangle$
104. $\langle \text{function} \rangle ::= \text{FUNCTION} \langle \text{procedure-identifier} \rangle;$
 $\langle \text{formal-parameter-list} \rangle;$
 $\langle \text{proc-declaration-body} \rangle \langle \text{statement-group} \rangle;$
 $\text{EXIT} \langle \text{procedure-identifier} \rangle$
105. $\langle \text{proc-group} \rangle ::= \langle \text{procedure} \rangle | \langle \text{procedure} \rangle ; \langle \text{proc-group} \rangle$
106. $\langle \text{procedure-section} \rangle ::= \text{PROCEDURES} ; \langle \text{proc-group} \rangle | \langle \text{empty} \rangle$

CONTINGENCY LIST

107. $\langle \rho \rangle ::= \langle \text{period} \rangle$
108. $\langle \beta_1 \rangle ::= \langle \text{period} \rangle$
109. $\langle \beta_2 \rangle ::= \langle \text{period} \rangle$
110. $\langle \text{rank} \rangle ::= \langle \eta \rangle | \langle \eta \rangle . \langle \pi \rangle$
111. $\langle \eta \rangle ::= \langle \text{period} \rangle$
112. $\langle \pi \rangle ::= \langle \text{period} \rangle$
113. $\langle \text{qualification} \rangle ::= \text{UNLESS} \langle \text{Boolean-expression} \rangle$
| $\text{IF} \langle \text{Boolean-expression} \rangle$ | $\langle \text{empty} \rangle$
114. $\langle \text{episode-timing} \rangle ::= \langle \rho \rangle | \langle \rho \rangle, \langle \beta_1 \rangle | \langle \rho \rangle, \langle \beta_1 \rangle, \langle \beta_2 \rangle$
| $\langle \rho \rangle, \langle \beta_1 \rangle, \langle \beta_2 \rangle, \langle \text{rank} \rangle$
115. $\langle \text{when-do} \rangle ::= \text{WHEN} \langle \text{function-designator} \rangle ; \langle \text{episode-timing} \rangle$
 $\text{DO} \langle \text{task-list} \rangle \langle \text{qualification} \rangle$

- 116. <simple-do>::=DO<task-list><rank><qualification>
- 117. <every>::=EVERY<p>DO<task-list><qualification>
- 118. <at-time>::=AT<time>DO<task-list><qualification>
- 119. <task-list>::=<task-designator>
|<task designator>THEN<task-list>
- 120. <contingency-defn>::=<when-do>|<simple-do>|<every>|<at-time>
- 121. <list-body>::=<contingency-defn>|<contingency-defn>;<list-body>
- 122. <contingency-list>::=CONTINGENCY LIST;<list-body>

ENVIRONMENT SECTION

- 123. <environment-section>::=ENVIRONMENT;<declaration-group>

IDENTIFICATION SECTION

- 124. <name>::=<string>
- 125. <date>::=<string>
- 126. <description>::=<string>
- 127. <id-section>::=IDENTIFICATION;DESIGNER:<name>;DATE:<date>;
PROJECT:<description>;

CONTROL SYSTEM DESIGN DESCRIPTION

- 128. <control-system-design>::=<id-section><environment-section>
<contingency-list><procedures-section>

APPENDIX D

A CSDL EXAMPLE

While this three contingency example is straightforward, it demonstrates the CSDL approach in three concrete application areas. The first contingency (ALARM) detects an alarm pulse and initiates its task (ALERT). The task must set a single warning light if a temperature reading is low, and set two lights otherwise. This is an example of discrete logic replacement [Rony, Larsen and Titus; 1976] in a field such as chemical process automation [Fisher, 1974].

The task of the second contingency is automatically initiated every fixed period to accomplish a parallel data conversion for driving a display, which is a common system requirement. Finally, the third contingency must sense a change in data in a parallel buffer, and cause the data to be output in a bit-serial mode. Such tasks are common in communications and automation controllers; they are most often performed by special purpose hardware [Larsen and Rony, April 1975].

IDENTIFICATION:

DESIGNER: NICK MATELAN
DATE: 7-4-76
PROJECT: MIXED-APPLICATION CSDL DEMONSTRATION

ENVIRONMENT:

INPUT ABORT,1,TTL; BUFFER,8,TTL;
 TEMP,8,TTL;

OUTPUT DISPLAY,4,TTL; SERIAL,1,TTL;
 WARN,2,TTL;

BINARY OLDBUF,8,63;

CONTINGENCY LIST:

"CONDITIONALLY ALERT OPERATOR TO ALARM SITUATION"
 WHEN ALARM : 2MS, 50US DO ALERT;

"CONVERT TEMPERATURE INFORMATION TO DISPLAY FORMAT"
 EVERY 4 MS DO ENCODE;

"CONVERT NEW PARALLEL DATA TO SERIAL PULSE TRAIN"
 WHEN DATA_READY : 1300US DO SERIALIZE;

PROCEDURES:

FUNCTION "TEST FOR" DATA_READY;

 "TESTS BUFFER FOR NEW DATA TO SERIALIZE"
 BINARY DATA_READY,1;

 SENSE (BUFFER) "FETCH CURRENT VALUE";

 "TEST VALUE TO LAST VALUE, PERFORM TASK IF DIFFERENT"
 IF BUFFER ≠ OLDBUF THEN DATA_READY ← 1 FI;

EXIT DATA_READY "DONE";

FUNCTION ALARM;

 "DETECT FATAL ERROR SIGNAL"
 BINARY ALARM,1;

 SENSE (ABORT) "ACCESS SIGNAL";

 "IF SIGNAL HIGH, ALERT OPERATOR"
 IF ABORT = 1 THEN ALARM ← 1 FI;

EXIT ALARM;

TASK SERIALIZE:

```
"SERIALIZE PARALLEL DATA IN BUFFER"
ARITHMETIC BITPSN.2; "DEFINE BIT POSITION POINTER"

"LOOP THROUGH BUFFER"
FOR BITPSN FROM 1 TO 8 : 8
DO "CONVERSION"
  IN 500 US:
    "OUTPUT PULSE"
    SERIAL ← BUFFER(BITPSN); "LINE HIGH IF DATA IS ONE"
    ISSUE (SERIAL);          "LEADING EDGE OUT"
    WAIT 100US;              "SET PULSE WIDTH"
    SERIAL ← 0;
    ISSUE (SERIAL)           "BRING LINE DOWN"
  NI;
OD;

OLDBUF ← BUFFER; "SAVE VALUE SENT"
```

EXIT SERIALIZE;

TASK ALERT "OPERATOR";

```
"ALERT OPERATOR TO ERROR IN CONTROLLED PROCESS"

WARN ← 1; "PRESET ALERT LAMP-DRIVER LINE WARN(1)"

"IF TEMPERATURE IS OVER LIMIT, SET SECOND LAMP-DRIVER LINE"
  SENSE (TEMP); "GET CURRENT TEMPERATURE"
  IF TEMP > 40 THEN WARN ← 3; "SET SECOND LAMP"

ISSUE (WARN); "SEND WARN VALUES TO LAMPS"
```

EXIT ALERT;

TASK ENCODE;

```
"ENCODE RANGE OF TEMPERATURE (A, B, OR C)"
" TO DRIVE LED DISPLAY"
CODE LEDCODE, 4, A:1001, B:1010, C:1011;
TEXT LETTER(3), LEDCODE 'ABC';

DSPLAY ← LETTER(1); "INITIALIZE LOWEST RANGE"

  SENSE (TEMP); "GET TEMPERATURE"

"TEST REMAINING RANGES"
  IF TEMP > 20 THEN DSPLAY ← LETTER(2) FI;
  IF TEMP > 30 THEN DSPLAY ← LETTER(3) FI;
  IF TEMP > 35 THEN PERFORM ALERT FI;

ISSUE (DSPLAY); "DISPLAY RANGE VALUE"
```

EXIT ENCODE;

APPENDIX E

A CSDL REALIZATION LIBRARY

VOLUME 1

```
COM *****
COM ***** VOLUME ONE - INTEL 8080 CPU *****
COM *****
```

```
COM FORMAT INFORMATION
TEMPLATE S LABEL,1,5,: OPCODE,8,11, OPERAND,13,29, COMMENT,30,70,:
COMLINE,2,70,:
```

```
COM *****
COM ** FUNCTIONAL REALIZATIONS - SOFTWARE SECTION **
COM *****
```

```
SKEL S.MAIN ( : : 0,0,0)
COM PRIMITIVE TO DEFINE CONTROLLER SETUP AND INITIALIZATION
COM LIST = EMPTY : EMPTY : STORAGE, TIME, EXT
CALC EVNT = -1
CALC PORTPNT = -1
CALL S.HEADING (\STATUS(3)\)
```

```
SKEL S.HEADING (TOTEVNT : : 14,0,0)
COM SUBROUTINE TO DEFINE SOFTWARE HEADING AND DEFINE INTERRUPT AREA
COM LIST = TOTAL-NUMBER-OF-EVENTS : EMPTY : STORAGE, TIME, EXT
CALC BEGIN = 8 * (TOTEVNT-1)
BEGIN STEXT
```

```
;
; + INTEL 8080 CONTROL SYSTEM +
;
; \IDSEC(1)\
; \IDSEC(2)\
; \IDSEC(3)\
;
; OVERVIEW:
; \CONTSEC\
;
;
```

```
ORG <BEGIN> ; BEGIN CODE AFTER INTERRUPT AREA
JMP @SPVSR ; FORCE MONITOR EXECUTION
@STAK DS 8 ; DEFINE STACK AREA
LXI SP,@STAK+8 ; INITIALIZE STACK POINTER
ENDTEXT
```

```
SKEL S.SENSEVENT (SIGNAM : 1,8 : 0,40,11)
COM PRIMITIVE TO DETECT EVENT-TYPE INPUT
```



```
COM LIST = SOURCE : MAX-INPUT-LINES, MAX-EVENTS : STORAGE, TIME, EXT
CALC EVPNT = EVPNT + 1
CALC EVADDR = EVPNT * 8
BEGIN STEXT
; VALUE ASSIGNED TO <SIGNAM> BY INTERRUPT <EVPNT>
ENDTEXT
INCL H.SENSEVENT (SIGNAM,EVPNT : 1)
```

```
SKEL S.EVENTMARK (SIGNAM,EVADDR :: 7,40,11)
COM PRIMITIVE TO ESTABLISH INTERRUPT HANDLER FOR EVENT
COM LIST = SOURCE, TRAP-ADDR : EMPTY : STORAGE, TIME, EXT
BEGIN STEXT
;
      ORG <EVADDR>          ; INTERRUPT TRAP FOR <SIGNAM>
      MVI B,1              ; POST INTERRUPT
      LHLD <SIGNAM>         ; FLAG FOR
      MOV M,B              ; SIGNAL <SIGNAM>
      RET                  ; RESUME PROCESSING
ENDTEXT
```

```
SKEL S.SENSECOND (SIGNAM : 8,128 : 5,23,4)
COM PRIMITIVE TO DETECT CONDITION-TYPE INPUT
COM LIST = SOURCE : MAX-INPUT-LINES, MAX-CONDS : STORAGE, TIME, EXT
CALC PORTPNT = PORTPNT + 1
BEGIN STEXT
      IN <PORTPNT>          ; SENSE ENVIRONMENTAL DATA
      STA <SIGNAM>         ; AND STORE IN <SIGNAM>
ENDTEXT
INCL H.SENSECOND (SIGNAM,PORTPNT)
```

```
SKEL S.ISSUEVENT (SIGNAM : 8,128 : 5,23,4)
COM PRIMITIVE TO SEND EVENT-TYPE OUTPUT
COM LIST = SINK : MAX-OUTPUT-LINES, MAX-EVENTS : STORAGE, TIME, EXT
COM USE SAME TECHNIQUE AS CONDITIONAL ISSUE
INCL S.ISSUECOND (SIGNAM : 8)
```

```
SKEL S.ISSUECOND (SIGNAM : 8,128 : 5,23,4)
COM PRIMITIVE TO SEND EVENT-TYPE OUTPUT
COM LIST = SINK : MAX-OUTPUT-LINES, MAX-EVENTS : STORAGE, TIME, EXT
CALC PORTPNT = PORTPNT + 1
BEGIN STEXT
      LDA <SIGNAM>          ; ISSUE CONTROL
      OUT <PORTPNT>        ; DATA THROUGH SIGNAL <SIGNAM>
ENDTEXT
INCL H.ISSUECOND (SIGNAM,PORTNAM : 8)
```

```
SKEL S.FIXEDWAIT (TIME : 2550,10 : 9,9)
COM PRIMITIVE TO CAUSE FIXED-TIME DELAY
COM LIST = TIME-DELAY : MAX-DELAY,RESOLUTION : STORAGE, EXT
CALC CNT = TIME / 10 - 1
BEGIN STEXT
      MVI A,0              ; WAIT <TIME> US
      MVI A,CNT            ; LOAD LOOP CNT
      DCR C                ; DELAY
      DCR A                ; DECREMENT LOOP COUNT
      JNZ $-2              ; LOOP UNTIL TIME IS UP
ENDTEXT
```

```
SKEL S.PROC (NAM :: 1,4,1)
COM PRIMITIVE TO DEFINE PROCEDURE ENTRY POINT
```

```
COM LIST = PROC-NAM : EMPTY : STORAGE, TIME, EXT
BEGIN STEXT
;
; PROCEDURE <NAM>
@<NAM>: NOP ; ENTRY POINT FOR <NAM>
ENDTEXT
```

```
SKEL S.EVERY (NAM :: 7,34,9)
COM PRIMITIVE TO DEFINE DUMMY FUNCTION FOR EVERY-PERIOD STATEMENT
COM LIST = PROC-NAM : EMPTY : STORAGE, TIME, EXT
BEGIN STEXT
; DUMMY PROCEDURE FOR EVERY-PERIOD TYPE CONTINGENCY
@<NAM>: NOP ; DUMMY FUNCTION ENTRY POINT
MVI A,1 ; FORCE FUNCTION <VAL>
STA <NAM> ; TO TRUE VALUE (1)
RET ; RETURN TO MONITOR
ENDTEXT
```

```
SKEL S.EXITPROC (NAM,CONT :: 6,30,8)
COM PRIMITIVE TO CLOSE PROC (AND RESET ASSOCIATED CONTINGENCY)
COM LIST = PROC-NAM, CONT-NAM : EMPTY : STORAGE, TIME, EXT
IF <CONT> = 0 SKIP 4
BEGIN STEXT
MVI A,0 ; RESET CONTINGENCY <CONT>
STA <CONT> ; FOLLOWING COMPLETED TASK
ENDTEXT
BEGIN STEXT
RET ; RETURN TO MONITOR, EXIT <NAM>
ENDTEXT
```

```
SKEL S.PERFORM (NAME :: 3,17,5)
COM PRIMITIVE TO INVOKE A PROCEDURE
COM LIST = PROC-NAME : EMPTY : STORAGE, TIME, EXT
BEGIN STEXT
CALL @<NAME> ; PERFORM PROCEDURE <NAME>
ENDTEXT
```

```
SKEL S.ASSIGN (VAR,DATA : 8,8 : 6,26,6)
COM PRIMITIVE TO ASSIGN VALUE OF VARIABLE TO VARIABLE
COM LIST = VAR, DATA-VAR : VAR-PREC, DATA-PREC : STORAGE, TIME, EXT
BEGIN STEXT
LDA <DATA> ; ASSIGN <DATA>
STA <VAR> ; TO <VAR>
ENDTEXT
```

```
SKEL S.ASSIGNCONS (VAR,DATA : 8,8 : 5,20,5)
COM PRIMITIVE TO ASSIGN VALUE OF CONSTANT TO VARIABLE
COM LIST = VAR, DATA-CONS : VAR-PREC, DATA-PREC : STORAGE, TIME, EXT
BEGIN STEXT
MVI A,<DATA> ; ASSIGN CONSTANT <DATA>
STA <VAR> ; TO <VAR>
ENDTEXT
```

```
SKEL S.COMS (NAM,VAL : 8 : 1,0,0)
COM PRIMITIVE TO DEFINE DATA
COM LIST = DATA-NAME, VALUE : VALUE-PREC : STORAGE, TIME, EXT
BEGIN STEXT
<NAM>: DB <VAL>
ENDTEXT
```

```

SKEL S.JMPF (VAL,LOC : 8 : 8,30,8)
COM PRIMITIVE TO BRANCH ON FALSE CONDITION
COM LIST = VALUE, JUMP-LOC : VALUE-PREC : STORAGE, TIME, EXT
BEGIN STEXT
    LDA    <VAL>                ; FETCH <VAL> AND
    CPI    0                    ; TEST FOR FALSE (0)
    JZ     <LOC>                ; BRANCH TO <LOC> IF FALSE
ENDTEXT

```

```

SKEL S.LOC (LOC :: 1,4,1)
COM PRIMITIVE TO DEFINE A LABEL (LOCATION)
COM LIST = LABEL-NAME : EMPTY : STORAGE, TIME, EXT
BEGIN STEXT
<LOC>: NOP                    ; DEFINE LOCATION <LOC>
ENDTEXT

```

```

SKEL S.EQ (RSLT,ARG1,ARG2 : 8,8,8 : 20,98,22)
COM PRIMITIVE TO SET RESULT TRUE IF ARG1 EQUALS ARG2, ELSE FALSE
COM LIST = RSLT,ARG1,ARG2 : RSLT-PREC,ARG1-PREC,ARG2-PREC
COM                                     : STORAGE, TIME, EXT
BEGIN STEXT
    LHLD   <RSLT>                ; PRESET <RSLT>
    MVI    M,1                  ; TO TRUE (1)
    LDA    <ARG1>                ; FETCH 1ST ARG <ARG1>
    LHLD   <ARG2>                ; COMPARE
    CMP    M                    ; TO 2ND ARG <ARG2>
    JZ     $+6                  ; SET <RSLT>
    LHLD   <RSLT>                ; TO FALSE (0)
    MVI    M,0                  ; IF ARGS ARE NOT EQUAL
ENDTEXT

```

```

SKEL S.NE (RSLT,ARG1 ARG2 : 8,8,8 : 20,98,22)
COM PRIMITIVE TO SET RESULT TRUE IF ARG1 NOT EQUAL ARG2, ELSE FALSE
COM LIST = RSLT,ARG1,ARG2 : RSLT-PREC,ARG1-PREC,ARG2-PREC
COM                                     : STORAGE, TIME, EXT
BEGIN STEXT
    LHLD   <RSLT>                ; PRESET <RSLT>
    MVI    M,1                  ; TO TRUE (1)
    LDA    <ARG1>                ; FETCH 1ST ARG <ARG1>
    LHLD   <ARG2>                ; COMPARE
    CMP    M                    ; TO 2ND ARG <ARG2>
    JNZ    $+6                  ; SET <RSLT>
    LHLD   <RSLT>                ; TO FALSE (0)
    MVI    M,0                  ; IF ARGS ARE EQUAL
ENDTEXT

```

```

SKEL S.GT (RSLT,ARG1 ARG2 : 8,8,8 : 20,98,22)
COM PRIMITIVE TO SET RESULT TRUE IF ARG1 GREATER THAN ARG2, ELSE FALSE
COM LIST = RSLT,ARG1,ARG2 : RSLT-PREC,ARG1-PREC,ARG2-PREC
COM                                     : STORAGE, TIME, EXT
BEGIN STEXT
    LHLD   <RSLT>                ; PRESET <RSLT>
    MVI    M,1                  ; TO TRUE (1)
    LDA    <ARG1>                ; FETCH 1ST ARG <ARG1>
    LHLD   <ARG2>                ; COMPARE
    CMP    M                    ; TO 2ND ARG <ARG2>
    JNC    $+6                  ; SET <RSLT>
    LHLD   <RSLT>                ; TO FALSE (0)
    MVI    M,0                  ; IF ARG1 NOT GREATER THAN ARG2
ENDTEXT

```

SKEL S.FORCONS (INDEX,LWR,UPR,SLAB,ELAB : 8,8,8,255 : 17,67,17)
 COM PRIMITIVE TO SET UP LOOP WITH CONSTANT BOUNDS
 COM LIST = INDEX,LWRBND,LWRBND,UPRBND,STRT-LABEL,END-LABEL :
 COM INDEX-PREC,LWRBND-PREC,UPRBND-PREC,MAX-LOOP-CNT
 COM : STORAGE, TIME, EXT

BEGIN STEXT
 MVI A,<LWR> ; PRESET INDEX <INDEX>
 STA <INDEX> ; TO LOWER BOUND <LWR>
 <SLAB>: LDA <INDEX> ; LOOP, FETCH INDEX
 MVI H,<UPR> ; COMPARE
 CMP H ; INDEX
 JZ \$+6 ; TO
 JNC <ELAB>+3 ; UPPER BOUND <UPR>
 ENDTXT

SKEL S.FOREND (INDEX,SLAB,ELAB : 8,255 : 7,36,7)
 COM PRIMITIVE TO TERMINATE FOR-LOOP
 COM LIST = INDEX,STRT-LABEL,END-LABEL : INDEX-PREC,MAX-LOOP-CNT
 COM : STORAGE, TIME, EXT

BEGIN STEXT
 LHLD <INDEX> ; INCREMENT
 INR M ; INDEX <INDEX>
 <ELAB>: JMP <SLAB> ; RETURN TO LOOP START <SLAB>
 ENDTXT

SKEL S.MASK (RSLT,DATA,I : 8,8,8 : 28,289,55)
 COM PRIMITIVE TO SET RESULT TO VALUE OF I-TH BIT OF DATA
 COM LIST = RSLT,DATA,BITPOS : RSLT-PREC,DATA-PREC,BITPOS-PREC
 COM : STORAGE, TIME, EXT

BEGIN STEXT
 LDA <DATA> ; FETCH DATA FROM <DATA>
 LHLD <I> ; FETCH THE
 MOV C,M ; BIT POSITION FROM <I>
 JMP \$+2 ; PRESET BIT COUNTER
 RRC ; ROTATE DATA RIGHT
 DCR C ; DECREMENT BIT COUNTER
 JNZ \$-2 ; TEST FOR CORRECT BIT
 ANI 1 ; MASK TO GET RIGHTMOST BIT
 STA <RSLT> ; SAVE VALUE IN <RSLT>
 ENDTXT

SKEL S.STOP (: 0,0,0)
 COM PRIMITIVE TO END SOFTWARE
 COM LIST = EMPTY : EMPTY : STORAGE, TIME, EXT
 BEGIN STEXT
 END ; SOFTWARE COMPLETE
 ENDTXT

COM *****
 COM ** FUNCTIONAL REALIZATIONS - HARDWARE SECTION **
 COM *****

SKEL H.PROCESSOR (: 500NS,20)
 COM PRIMITIVE TO DEFINE CENTRAL PROCESSING UNIT HARDWARE
 COM LIST = EMPTY : EMPTY : LATENCY (CLOCK), POWER

```
BEGIN HTEXT
CENTRAL PROCESSING UNIT
DEVICE: INTEL 8080 8-BIT MICROPROCESSOR
CONNECTIONS:
  PINS 25,26,27,29,30,31,32,33,
    34,35, 1,40,37,38,39,36 = A(0:15)
  PINS 10,9,8,7,3,4,5,6 = D(0:7)
  PIN 21 = HLDA
  PIN 12 = RESET
  PIN 13 = HOLD
  PIN 14 = INT
  PIN 16 = INTE
  PIN 17 = DBIN
  PIN 18 = WR-BAR
  PIN 19 = SYNC
  PIN 22 = PHI1
  PIN 15 = PHI2
  PIN 23 = READY
  PIN 24 = WAIT
  PIN 2  GND
  PIN 20  +5V
  PIN 11  -5V
  PIN 28  +12V
```

```
ENDTEXT
BEGIN HTEXT
```

```
CLOCK GENERATOR (500NS)
DEVICE: INTEL 8224 CLOCK GENERATOR AND DRIVER FOR 8080 CPU
CONNECTIONS:
  PIN 1 [RESET] = RESET
  PIN 4 [READY] = READY
  PIN 5 [SYNC] = SYNC
  PIN 10 [PHI2] = PHI2
  PIN 11 [PHI1] = PHI1
  PIN 14,15 [XTAL(1:2)] = DEVICE: 18 MHZ CRYSTAL
  PIN 16 [VCC] = +5V
  PIN 9 [VDD] = +12
  PIN 8 [GND] = GND
```

```
ENDTEXT
COM INCLUDE BUS DRIVER MORE THAN TWO I/O PORTS
IF \STATUS(2)\ .LE. 2  SKIP 1
  CALL H.BUSDRVR
COM OTHERWISE CONNECT PORTS DIRECTLY
IF \STATUS(2)\ .GT. 2  SKIP 3
BEGIN HTEXT
  D(0:7) = DB(0:7)
ENDTEXT
BEGIN HTEXT
```

```
CPU STATUS LATCH
DEVICE: INTEL 8212 8-BIT I/O PORT
CONNECTIONS:
  PINS 3,5,7,9,16,18,20,22 [DI(0:7)] = D(0:7)
  PIN 4 [DO(1)] = INTA
  PIN 6 [DO(2)] = WO-BAR
  PIN 8 [DO(3)] = STACK
  PIN 10 [DO(4)] = HLTA
  PIN 15 [DO(5)] = OUT
  PIN 17 [DO(6)] = M1
  PIN 19 [DO(7)] = INP
```

```
PIN 21 [DO(8)] = MEMR
ENDTEXT
```

```
SKEL H.BUSDRVR ( :: 0,10)
COM SUBROUTINE TO DEFINE DATA BUS DRIVER HARDWARE
COM LIST = EMPTY : EMPTY : LATENCY, POWER
BEGIN HTEXT
```

```
DATA BUS BUFFER DRIVER / LOWER HALF (4 OF 8 BITS)
DEVICE: INTEL 8216 4-BIT BIDIRECTIONAL BUS DRIVER
CONNECTIONS:
```

```
PINS 3,6,10,13 [DB(0:3)] = DB(0:3)
PIN 2 [DO(0)] = D(0)
PIN 5 [DO(1)] = D(1)
PIN 11 [DO(2)] = D(2)
PIN 14 [DO(3)] = D(3)
PIN 4 [DI(0)] = D(0)
PIN 7 [DI(1)] = D(1)
PIN 9 [DI(2)] = D(2)
PIN 12 [DI(3)] = D(3)
PIN 1 [CS-BAR] = .NOT. HLDA
PIN 15 [DIEN-BAR] = .NOT. DBIN
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND
```

```
DATA BUS BUFFER DRIVER / UPPER HALF (4 OF 8 BITS)
DEVICE: INTEL 8216 4-BIT BIDIRECTIONAL BUS DRIVER
CONNECTIONS:
```

```
PINS 3,6,10,13 [DB(0:3)] = DB(4:7)
PIN 2 [DO(0)] = D(4)
PIN 5 [DO(1)] = D(5)
PIN 11 [DO(2)] = D(6)
PIN 14 [DO(3)] = D(7)
PIN 4 [DI(0)] = D(4)
PIN 7 [DI(1)] = D(5)
PIN 9 [DI(2)] = D(6)
PIN 12 [DI(3)] = D(7)
PIN 1 [CS-BAR] = .NOT. HLDA
PIN 15 [DIEN-BAR] = .NOT. DBIN
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND
```

```
SKEL H.MEMORY ( : 1024 : 500NS,600MW)
COM PRIMITIVE TO DEFINE MEMORY HARDWARE
COM LIST = EMPTY : MAX-MEMORY : TIME-DELAY, POWER
BEGIN HTEXT
```

```
SYSTEM MEMORY
TIME DELAY TO MATCH R111-2 LATENCY TO 8080
DEVICE: TEXAS INSTRUMENTS SN74175 HEX-QUAD D-TYPE FLIP-FLOP WITH CLEAR
CONNECTION:
```

```
PIN 9 [CLOCK] = .NOT. PHI2
PIN 1 [CLEAR] = .NOT. SYNC
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND
PIN 2 [IQ] = Q
PIN 4 [ID] = +5V
READY = Q .AND. .NOT. MEMR
```

```
ENDTEXT
COM INCLUDE INITIAL 256 BYTES
CALL H.RAM256 (CE1,0)
COM INCLUDE ANOTHER 256 BYTES IF NECESSARY
IF ~STATUS(1) \ .LE. 256 SKIP 1
```

```

CALL H.RAM256 (CE2,1)
COM INCLUDE ANOTHER 256 BYTES IF NECESSARY
IF \STATUS(1)\ .LE. 512 SKIP 1
CALL H.RAM256 (CE3,2)
COM INCLUDE ANOTHER 256 BYTES IF NECESSARY
IF \STATUS(1)\ .LE. 768 SKIP 1
CALL H.RAM256 (CE4,3)

SKEL H.RAM256 (CE,SELECT)
COM SUBROUTINE TO DEFINE 256 BYTE PAGE OF RAM
COM LIST = CHIP-ENABLE, CHIP-SELECT-CODE : EMPTY : EMPTY
BEGIN HTEXT
RANDOM ACCESS MEMORY (LOWER HALF OF PAGE <SELECT>)
DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM
CONNECTION:
PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
PIN 9 [OD] = DBIN
PIN 16 [RW] = RW
PIN 15 [CE1-BAR] = <CE>
PIN 10 [CE2-BAR] = GND
PINS 11,12,13,14 [IO(1:4)] = DB(0:3)
<CE> = ( .NOT. ( OUT .AND. WR-BAR ) .OR. MEMR )
        .AND. ( .NOT. ( DECODE A(8:10) VALUE <SELECT> ) )
PIN 18 [VCC] = +5V
PIN 8 [GND] = GND
RANDOM ACCESS MEMORY (UPPER HALF OF PAGE <SELECT>)
DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM
CONNECTION:
PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
PIN 9 [OD] = DBIN
PIN 16 [RW] = RW
PIN 15 [CE1-BAR] = <CE>
PIN 10 [CE2-BAR] = GND
PINS 11,12,13,14 [IO(1:4)] = DB(4:7)
PIN 18 [VCC] = +5V
PIN 8 [GND] = GND
ENDTEXT

SKEL H.SENSEVENT (SIGNAM,EVPNT : 1,8,TTL : 500NS, 500MW)
COM PRIMITIVE TO DEFINE EVENT-TYPE INPUT HARDWARE
COM LIST = SOURCE, PORT-NUM : MAX-INPUT-LINES, MAX-EVENTS,
COM TECHNOLOGY : LATENCY, POWER
BEGIN HTEXT
EVENT-MODE INPUT INTERFACE HARDWARE TO SENSE SIGNAL: <SIGNAM>
DEVICE: INTEL 8212 8-BIT I/O PORT
CONNECTIONS:
NOTE: '1' = +5V, '0' = GND
PINS 3,5,7 [DI(1:3)] = '111'
PINS 20,22 [DI(7:8)] = '11'
PINS 9,16,18 [DI(4:6)] = '\FORMAT(EVPNT,8,3)\
PINS 4,6,0,10,15,17,19,21 [DO(1:8)] = DB(1:8)
PIN 2 [MD] = '1'
PIN 11 [STB] = '0'
PIN 1 [DS1-BAR] = .NOT. ( INTA .AND. <SIGNAM> )
PIN 13 [DS2] = DBIN
OR-TIE <SIGNAM> TO INT
PIN 12 [GND] = GND
PIN 24 [VCC] = +5V
ENDTEXT
INCL S.EVENTMARK (SIGNAM,EVADDR)

```

```

SKEL H.SENSECOND (SIGNAM,PORTPNT : 8,128,TTL : 500NS, 500MW)
COM PRIMITIVE TO DEFINE CONDITION-TYPE INPUT HARDWARE
COM LIST = SOURCE, PORT-NUM : MAX-INPUT-LINES,
COM      MAX-CONDS, TECHNOLOGY : LATENCY, POWER
BEGIN HTEXT
  CONDITION-MODE INPUT INTERFACE HARDWARE TO SENSE SIGNAL: <SIGNAM>
  DEVICE: INTEL 8212 8-BIT I/O PORT
  CONNECTIONS:
    PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = <SIGNAM>, REMAINDER TO GND
    PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = DB(1:8)
    PINS2 [MD] = GND
    PINS11 [STB] = GND
    PINS1 [DS1-BAR] = .NOT. ( DECODE A(0:7) VALUE <PORTPNT> )
    PINS13 [DS2] = INP .AND. DBIN
    PIN 24 [VCC] = +5V
    PIN 12 [GND] = GND
ENDTEXT

```

```

SKEL H.ISSUEEVENT (SIGNAM,PORTPNT : 8,128,TTL : 500NS, 500MW)
COM PRIMITIVE TO DEFINE EVENT-TYPE OUTPUT HARDWARE
COM LIST = SINK, PORT-NUM : MAX-OUTPUT-LINES,
COM      MAX-EVENTS, TECHNOLOGY : LATENCY, POWER
COM      USE SAME TECHNIQUE FOR BOTH CONDITION AND EVENT OUTPUT
INCL H.ISSUECOND (SIGNAM,PORTPNT)

```

```

SKEL H.ISSUECOND (SIGNAM,PORTPNT : 8,128,TTL : 500NS, 500MW)
COM PRIMITIVE TO DEFINE CONDITION-TYPE OUTPUT HARDWARE
COM LIST = SINK, PORT-NUM : MAX-OUTPUT-LINES,
COM      MAX-EVENTS, TECHNOLOGY : LATENCY, POWER
BEGIN HTEXT
  CONDITION-MODE OUTPUT INTERFACE HARDWARE TO ISSUE SIGNAL: <SIGNAM>
  DEVICE: INTEL 8212 8-BIT I/O PORT
  CONNECTIONS:
    PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = DB(1:8)
    PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = <SIGNAM>(1:\PREC(SIGNAM)\)
    PINS 2 [MD] = +5V
    PINS11 [STB] = GND
    PINS 1 [DS1-BAR] = WR-BAR
    PINS13 [DS2] = OUT .AND. ( DECODE A(0:7) VALUE <PORTPNT> )
    PIN 24 [VCC] = +5V
    PIN 12 [GND] = GND
ENDTEXT

```

```

COM      *****
COM      ** MONITOR REALIZATIONS - SOFTWARE SECTION **
COM      *****

```

```

SKEL S.MONITOR ( : P2 : 22,121,30)
COM PRIMITIVE TO DEFINE P2 MONITOR AS CONTROLLER SUPERVISOR
COM LIST = EMPTY : MONITOR-TYPE-NAME : STORAGE, TIME, EXT
BEGIN STEXT
;
; * MONITOR SECTION *
;
@SPVSR: LHLD @TABLE      ; INITIALIZE TABLE POINTER
        SHLD @PNTR      ; TO BEGINNING

```



```

@LOOP:  LHLD  @PNTR          ; MAIN LOOP: GET POINTER
        INX   H             ; INCREMENT
        INX   H             ; THE POINTER
        INX   H             ; BY 3 (BYTES TO BYPASS JMP)
        SHLD  $+2           ; STORE POINTER ADDRESS FOR TABLE ACCESS
        JMP   0             ; BEGIN EXECUTION AT CURRENT TABLE ENTRY
; DATA SECTION
@PNTR:  DW    1             ; TABLE ENTRY ADDRESS POINTER
@TABLE: DW    $-1          ; TABLE HEADER (DEFINE TOP)
ENDTEXT

SKEL S.TABENT (FNC,TASK : P2 : 3,10,3)
COM PRIMITIVE TO ADD ONE ENTRY TO MONITOR TABLE FOR P2 MONITOR
COM LIST = FUNC-NAME, TASK-NAME : MONITOR-TYPE-NAME : STORAGE, TIME, EXT
BEGIN STXT
        JMP   @T<FNC>       ; TEST FOR CONTINGENCY <FNC>
ENDTEXT
INCL S.TABACCP2 (FNC,TASK)

SKEL S.TABEND ( : P2 : 3,0,0)
COM SUBROUTINE TO DEFINE END OF P2 MONITOR TABLE
COM LIST = EMPTY : MONITOR-TYPE-NAME : STORAGE, TIME, EXT
BEGIN STXT
        JMP   @SPVSR        ; START AT TOP OF TABLE
ENDTEXT

SKEL S.TABACCP2 (FNC,TASK :: 14,0,0)
COM SUBROUTINE TO ADD ACTUAL ROUTINE ACCESS FOR CONTINGENCY/TASK PAIR
COM LIST = FUNC-NAME, TASK-NAME : EMPTY : STORAGE, TIME, EXT
BEGIN STXT
;
@T<FNC>:CALL @<FNC>         ; EXECUTE CONTINGENCY CODE <FNC>
        LDA   <FNC>         ; FETCH CONTINGENCY RESULT
        CPI   1             ; AND COMPARE TO TRUE FLAG (1)
        CZ    @<TASK>       ; EXECUTE TASK <TASK> IF TRUE
        JMP   @LOOP         ; RETURN TO MONITOR
ENDTEXT

```

APPENDIX F

INTERMEDIATE FORM OF THE EXAMPLE

GENERATED BY: SYSTEM (INPUT ANALYZER)			
S.MAIN	/ATTR :	0.	0. 0
GENERATED BY: PRIMITIVE S.MAIN			
S.Heading (1 :)	/ATTR :	14.	0. 0
GENERATED FOR: EVERY			
S.EVERY (E000)	/ATTR	7.	34. 9
GENERATED FOR: FUNCTION DATAREADY			
S.PROC (DATAREADY)	/ATTR :	1.	4. 1
S.SENSECOND (BUFFER : 8.4)	/ATTR :	5.	23. 4
S.NE (@T1,BUFFER,OLDBUF : 8.8.8)	/ATTR :	20.	98. 22
S.JMPF (@T1,@L000 : 8)	/ATTR :	8.	30. 8
S.ASSIGNCONS (DATAREADY.1 : 1.1)	/ATTR :	5.	20. 5
S.LOC (@L000)	/ATTR :	1.	4. 1
S.EXITPROC (DATAREADY.0)	/ATTR :	6.	30. 8
GENERATED FOR: FUNCTION ALARM			
S.PROC (ALARM)	/ATTR :	1.	4. 1
S.SENSEVENT (ABORT : 1.1)	/ATTR :	0.	40. 11
S.EQ (@T1,ABORT,@C000 : 8.1.1)	/ATTR :	20.	98. 22
S.JMPF (@T1,@L001 : 8)	/ATTR :	8.	30. 8
S.ASSIGNCONS (ALARM.1 : 1.1)	/ATTR :	5.	20. 5
S.LOC (@L001)	/ATTR :	1.	4. 1
S.EXITPROC (ALARM.0)	/ATTR :	6.	30. 8
S.CONS (@C000.1 : 1.1)	/ATTR :	1.	0. 0
GENERATED FOR: TASK SERIALIZE			
S.PROC (SERIALIZE)	/ATTR :	1.	4. 1
S.FORCONS (BITPSN.1.8,@FS000,@FE000 : 7.1.4.8)	/ATTR :	17.	67. 17
S.MASK (@T1,BUFFER,BITPSN : 8.8.7)	/ATTR :	20.	209. 55
S.ASSIGN (SERIAL,@T1 : 1.8)	/ATTR :	6.	26. 6
S.ISSUECOND (SERIAL : 1.4)	/ATTR :	5.	23. 4
S.FIXEDWAIT (100 : 2550.10)	/ATTR :	9.	9. 0
S.ASSIGNCONS (SERIAL.0 : 1.1)	/ATTR :	5.	20. 5
S.ISSUECOND (SERIAL : 1.4)	/ATTR :	5.	23. 5
S.FOREND (BITPSN.@FS000,@FE000 : 7.8)	/ATTR :	7.	36. 7
S.ASSIGN (OLDBUF,BUFFER : 8.8)	/ATTR :	6.	26. 6
S.EXITPROC (SERIALIZE,DATAREADY)	/ATTR :	6.	30. 8
GENERATED FOR: TASK ALERT			
S.PROC (ALERT)	/ATTR :	1.	4. 1
S.ASSIGNCONS (WARN.1 : 2.1)	/ATTR :	5.	20. 5
S.SENSECOND (TEMP : 8.4)	/ATTR :	5.	23. 4
S.GT (@T1,TEMP,@C001 : 8.8.6)	/ATTR :	20.	98. 22
S.JMPF (@T1,@L002 : 8)	/ATTR :	8.	30. 8
S.ASSIGNCONS (WARN.3 : 2.2)	/ATTR :	5.	20. 5
S.LOC (@L002)	/ATTR :	1.	4. 1
S.ISSUECOND (WARN : 2.4)	/ATTR :	5.	23. 4

S.EXITPROC (ALERT,ALARM)	/ATTR : 6, 30, 8
S.CON\$ (BITPSN,0 : 7)	/ATTR : 1, 0, 0
S.CON\$ (@C001,40 : 6)	/ATTR : 1, 0, 0
GENERATED FOR: TASK ENCODE	
S.PROC (ENCODE)	/ATTR : 1, 4, 1
S.ASSIGNCONS (DSPLAY,9 : 4,4)	/ATTR : 5, 20, 5
S.SENSECOND (TEMP : 8,4)	/ATTR : 5, 23, 4
S.GT (@T1,TEMP,@C002 : 8,8,6)	/ATTR : 20, 98, 22
S.JMPF (@T1,@L003 : 8)	/ATTR : 8, 30, 8
S.ASSIGNCONS (DSPLAY,10 : 4,4)	/ATTR : 5, 20, 5
S.LOC (@L003)	/ATTR : 1, 4, 1
S.GT (@T1,TEMP,@C003 : 8,8,5)	/ATTR : 20, 98, 22
S.JMPF (@T1,@L004 : 8)	/ATTR : 8, 30, 8
S.ASSIGNCONS (DSPLAY,11 : 4,4)	/ATTR : 5, 20, 5
S.LOC (@L004)	/ATTR : 1, 4, 1
S.GT (@T1,TEMP,@C004 : 8,8,5)	/ATTR : 20, 98, 22
S.JMPF (@T1,@L005 : 8)	/ATTR : 8, 30, 8
S.PERFORM (ALERT)	/ATTR : 3, 17, 5
S.LOC (@L005)	/ATTR : 1, 4, 1
S.ISSUECOND (DSPLAY : 4,4)	/ATTR : 5, 23, 4
S.EXITPROC (ENCODE,0)	/ATTR : 6, 30, 8
S.CON\$ (@C002,20 : 5)	/ATTR : 1, 0, 0
S.CON\$ (@C003,30 : 5)	/ATTR : 1, 0, 0
S.CON\$ (@C004,35 : 6)	/ATTR : 1, 0, 0
GENERATED BY: SYSTEM (INPUT ANALYZER)	
S.CON\$ (ABORT,0 : 1)	/ATTR : 1, 0, 0
S.CON\$ (BUFFER,0 : 8)	/ATTR : 1, 0, 0
S.CON\$ (TEMP,0 : 8)	/ATTR : 1, 0, 0
S.CON\$ (DSPLAY,0 : 4)	/ATTR : 1, 0, 0
S.CON\$ (SERIAL,0 : 1)	/ATTR : 1, 0, 0
S.CON\$ (WARN,0 : 2)	/ATTR : 1, 0, 0
S.CON\$ (OLDBUF,63 : 8)	/ATTR : 1, 0, 0
S.CON\$ (@T1,0 : 8)	/ATTR : 1, 0, 0
GENERATED BY: SYSTEM (TIMING ANALYZER)	
S.MONITOR (: P2)	/ATTR : 22, 121, 30
S.TABENT (E000,ENCODE : P2)	/ATTR : 3, 10, 3
S.TABENT (DATAREADY,SERIALIZE : P2)	/ATTR : 3, 10, 3
S.TABENT (ALARM,ALERT : P2)	/ATTR : 3, 10, 3
S.TABENT (DATAREADY,SERIALIZE : P2)	/ATTR : 3, 10, 3
S.TABENT (ALARM,ALERT : P2)	/ATTR : 3, 10, 3
S.TABENT (DATAREADY,SERIALIZE : P2)	/ATTR : 3, 10, 3
S.TABEND (: P2)	/ATTR : 3, 10, 3
GENERATED BY: PRIMITIVE S.TABENT	
S.TABACCP2 (E000,ENCODE)	/ATTR : 14, 0, 0
S.TABACCP2 (DATAREADY,SERIALIZE)	/ATTR : 14, 0, 0
S.TABACCP2 (ALARM,ALERT)	/ATTR : 14, 0, 0
GENERATED BY: SYSTEM (BINDER)	
S.EVENTMARK (ABORT,0)	/ATTR : 7, 40, 11
GENERATED BY: SYSTEM (INPUT ANALYZER)	
S.STOP	/ATTR : 0, 0, 0
GENERATED BY: SYSTEM (BINDER)	
H.PROCESSOR (:)	/ATTR : 500NS, 2W
H.MEMORY (: 510)	/ATTR : 500NS, .1W

GENERATED BY: PRIMITIVE H.PROCESSOR
H.BUSDRVR.(:)

/ATTR : 0, 1W

GENERATED BY: PRIMITIVE H.MEMORY
H.RAM256 (CE1,0)
H.RAM256 (CE2,1)

/ATTR : 0, .2W
/ATTR : 0, .2W

GENERATED BY: SOFTWARE INTERFACE PRIMITIVES

H.SENSECOND (BUFFER,0 : 8,4,TTL)
H.SENSEVENT (ABORT,0 : 1,1,TTL)
H.ISSUECOND (SERIAL,1 : 1,4,TTL)
H.SENSECOND (TEMP,2 : 8,4,TTL)
H.ISSUECOND (WARN,3 : 2,4,TTL)
H.ISSUECOND (DSPLAY,4 : 4,4,TTL)

/ATTR : 500NS, .5W
/ATTR : 4US, .5W
/ATTR : 500NS, .5W
/ATTR : 500NS, .5W
/ATTR : 500NS, .5W
/ATTR : 500NS, .5W

APPENDIX G

CONTROL CODE TO REALIZE THE EXAMPLE

```

;
;      + INTEL 8080 CONTROL SYSTEM +
;
; DESIGNER: NICK MATELAN
; DATE: 7-4-76
; PROJECT: MIXED-APPLICATION CSDL DEMONSTRATION
;
; OVERVIEW:
;   CONDITIONALLY ALERT OPERATOR TO ALARM SITUATION
;   CONVERT TEMPERATURE INFORMATION TO DISPLAY FORMAT
;   CONVERT NEW PARALLEL DATA TO SERIAL PULSE TRAIN
;
;
;      ORG      8          ; BEGIN CODE AFTER INTERRUPT AREA
;      JMP      @SPVSR     ; FORCE MONITOR EXECUTION
@STAK  DS       8          ; DEFINE STACK AREA
;      LXI      SP,@STAK+8 ; INITIALIZE STACK POINTER
;
; DUMMY PROCEDURE FOR EVERY-PERIOD TYPE CONTINGENCY
@E000: NOP                ; DUMMY FUNCTION ENTRY POINT
;      MVI      A,1        ; FORCE FUNCTION E000
;      STA      E000       ; TO TRUE VALUE (1)
;      RET              ; RETURN TO MONITOR
;
; PROCEDURE DATAREADY
@DATAR: NOP                ; ENTRY POINT FOR DATAREADY
;   TESTS BUFFER FOR NEW DATA TO SERIALIZE
;   FETCH CURRENT VALUE
;      IN       0          ; SENSE ENVIRONMENTAL DATA
;      STA      BUFFER     ; AND STORE IN BUFFER
;   TEST VALUE TO LAST VALUE, PERFORM TASK IF DIFFERENT
;      LHL      @T1        ; PRESET @T1
;      MVI      M,1        ; TO TRUE (1)
;      LDA      BUFFER     ; FETCH 1ST ARG BUFFER
;      LHL      OLDBUF     ; COMPARE
;      CMP      M          ; TO 2ND ARG OLDBUF
;      JNZ      $+6        ; SET @T1
;      LHL      @T1        ; TO FALSE
;      MVI      M,0        ; IF ARGS ARE EQUAL
;      LDA      @T1        ; FETCH @T1 AND
;      CPI      0          ; TEST FOR FALSE (0)
;      JZ       @L000      ; BRANCH TO @L000 IF FALSE
;      MVI      A,1        ; ASSIGN 1
;      STA      DATARE     ; TO DATAREADY
@L000: NOP                ; DEFINE LOCATION @L000
;   DONE
;      RET              ; RETURN TO MONITOR, EXIT DATAREADY
;

```

```

; PROCEDURE ALARM
@ALARM: NOP ; ENTRY POINT FOR ALARM
; DETECT FATAL ERROR SIGNAL
; ACCESS SIGNAL
; VALUE ASSIGNED TO ABORT BY INTERRUPT 0
; IF SIGNAL HIGH, ALERT OPERATOR
    LHLD @T1 ; PRESET @T1
    MVI M,1 ; TO TRUE (1)
    LDA ABORT ; FETCH 1ST ARG ABORT
    LHLD @C000 ; COMPARE
    CMP M ; TO 2ND ARG @C000
    JZ $+6 ; SET @T1
    LHLD @T1 ; TO FALSE
    MVI M,0 ; IF ARGS ARE NOT EQUAL
    LDA @T1 ; FETCH @T1 AND
    CPI 0 ; TEST FOR FALSE (0)
    JZ @L001 ; BRANCH TO @L001 IF FALSE
    MVI A,1 ; ASSIGN 1
    STA ALARM ; TO ALARM
@L001: NOP ; DEFINE LOCATION @L001
    RET ; RETURN TO MONITOR, EXIT ALARM
@C000: DB 1
;
; PROCEDURE SERIALIZE
@SERIA: NOP ; ENTRY POINT FOR SERIALIZE
; SERIALIZE PARALLEL DATA IN BUFFER
; LOOP THROUGH BUFFER
    MVI A,1 ; PRESET INDEX BITPSN
    STA BITPSN ; TO LOWER BOUND 1
@FS000: LDA BITPSN ; LOOP, FETCH INDEX
    MVI H,8 ; COMPARE
    CMP H ; INDEX
    JZ $+6 ; TO
    JNC @FE000+3 ; UPPER BOUND 8
;
; CONVERSION
; OUTPUT PULSE
; LINE HIGH IF DATA IS ONE
    LDA BUFFER ; FETCH DATA FROM BUFFER
    LHLD BITPSN ; FETCH THE
    MOV C,M ; BIT POSITION FROM BITPSN
    JMP $+2 ; PRESET BIT COUNTER
    RRC ; ROTATE DATA RIGHT
    DCR C ; DECREMENT BIT COUNTER
    JNZ $-2 ; TEST FOR CORRECT BIT
    ANI 1 ; MASK TO GET RIGHTMOST BIT
    STA @T1 ; SAVE VALUE IN @T1
    LDA @T1 ; ASSIGN @T1
    STA SERIAL ; TO SERIAL
; LEADING EDGE OUT
    LDA SERIAL ; ISSUE CONTROL
    OUT 1 ; DATA THROUGH SIGNAL SERIAL
; SET PULSE WIDTH
    MVI A,0 ; WAIT 100 US
    MVI A,9 ; LOAD LOOP CNT
    DCR C ; DELAY
    DCR A ; DECREMENT LOOP COUNT
    JNZ $-2 ; LOOP UNTIL TIME IS UP
    MVI A,0 ; ASSIGN CONSTANT 0
    STA SERIAL ; TO SERIAL

```

```

; BRING LINE DOWN
LDA SERIAL ; ISSUE CONTROL
OUT 1 ; DATA THROUGH SIGNAL SERIAL
LHLD BITPSN ; INCREMENT
INR M ; INDEX BITPSN
@FE000 JMP @FS000 ; RETURN TO LOOP START @FS000
; SAVE VALUE SENT
LDA BUFFER ; ASSIGN BUFFER
STA OLDBUF ; TO OLDBUF
MVI A,0 ; RESET CONTINGENCY DATAREADY
STA DATARE ; FOLLOWING COMPLETED TASK
RET ; RETURN TO MONITOR, EXIT SERIALIZE

; PROCEDURE ALERT
@ALERT: NOP ; ENTRY POINT FOR ALERT
; ALERT OPERATOR TO ERROR IN CONTROLLED PROCESS
; PRESET ALERT LAMP-DRIVER LINE WARN(1)
MVI A,1 ; ASSIGN CONSTANT 1
STA ALERT ; TO WARN
; IF TEMPERATURE IS OVER LIMIT, SET SECOND LAMP DRIVER LINE
; GET CURRENT TEMPERATURE
IN 2 ; SENSE ENVIRONMENTAL DATA
STA TEMP ; AND STORE IN TEMP
; SET SECOND LAMP
LHLD @T1 ; PRESET @T1
MVI M,1 ; TO TRUE (1)
LDA TEMP ; FETCH 1ST ARG TEMP
LHLD @C001 ; COMPARE
CMP M ; TO 2ND ARG @C001
JNC $+6 ; SET @T1
LHLD @T1 ; TO FALSE
MVI M,0 ; IF ARG1 NOT GREATER THAN ARG2
LDA @T1 ; FETCH @T1 AND
CPI 0 ; TEST FOR FALSE(0)
JZ @L002 ; BRANCH TO @L002 IF FALSE
MVI A,3 ; ASSIGN CONSTANT 3
STA WARN ; TO WARN
@L002: NOP ; DEFINE LOCATION @L002
; SEND WARN VALUES TO LAMPS
LDA WARN ; ISSUE CONTROL
OUT 3 ; DATA THROUGH SIGNAL WARN
MVI A,0 ; RESET CONTINGENCY ALARM
STA ALARM ; FOLLOWING COMPLETED TASK
RET ; RETURN TO MONITOR, EXIT ALERT

BITPSN: DB 0
@C001: DB 40

; PROCEDURE ENCODE
@ENCOD: NOP ; ENTRY POINT FOR ENCODE
; ENCODE RANGE OF TEMPERATURE (A, B, OR C)
; TO DRIVE LED DISPLAY
; INITIALIZE LOWEST RANGE
MVI A,9 ; ASSIGN CONSTANT 9
STA DSPLAY ; TO DSPLAY
; GET TEMPERATURE
IN 2 ; SENSE ENVIRONMENTAL DATA
STA TEMP ; AND STORE IN TEMP
; TEST REMAINING RANGES
LHLD @T1 ; PRESET @T1
MVI M,1 ; TO TRUE (1)

```

```

LDA    TEMP                ; FETCH 1ST ARG TEMP
LHLD   @C002              ; COMPARE
CMP     M                  ; TO 2ND ARG @C002
JNC     $+6               ; SET @T1
LHLD   @T1                ; TO FALSE
MVI     M,0               ; IF ARG1 NOT GREATER THAN ARG2
LDA     @T1               ; FETCH @T1 AND
CPI     0                 ; TEST FOR FALSE (0)
JZ      @L003             ; BRANCH TO @L003 IF FALSE
MVI     M,10              ; ASSIGN CONSTANT 10
STA     DSPLAY            ; TO DSPLAY
@L003: NOP                ; DEFINE LOCATION @L003
LHLD   @T1                ; PRESET @T1
MVI     M,1               ; TO TRUE (1)
LDA     TEMP              ; FETCH 1ST ARG TEMP
LHLD   @C003              ; COMPARE
CMP     M                  ; TO 2ND ARG @C003
JNC     $+6               ; SET @T1
LHLD   @T1                ; TO FALSE
MVI     M,0               ; IF ARG1 NOT GREATER THAN ARG2
LDA     @T1               ; FETCH @T1 AND
CPI     0                 ; TEST FOR FALSE (0)
JZ      @L004             ; BRANCH TO @L004 IF FALSE
MVI     M,11              ; ASSIGN CONSTANT 11
STA     DSPLAY            ; TO DSPLAY
@L004: NOP                ; DEFINE LOCATION @L004
LHLD   @T1                ; PRESET @T1
MVI     M,1               ; TO TRUE (1)
LDA     TEMP              ; FETCH 1ST ARG TEMP
LHLD   @C004              ; COMPARE
CMP     M                  ; TO 2ND ARG @C004
JNC     $+6               ; SET @T1
LHLD   @T1                ; TO FALSE
MVI     M,0               ; IF ARG1 NOT GREATER THAN ARG2
LDA     @T1               ; FETCH @T1 AND
CPI     0                 ; TEST FOR FALSE (0)
JZ      @L005             ; BRANCH TO @L005 IF FALSE
CALL    @ALERT            ; PERFORM PROCEDURE ALERT
@L005: NOP                ; DEFINE LOCATION @L005
; DISPLAY RANGE VALUE
LDA     DSPLAY
OUT     4
RET
@C002   DB    20
@C003   DB    30
@C004   DB    35
; GLOBAL VARIABLES
ABORT   DB    0
BUFFER  DB    0
TEMP    DB    0
DSPLAY  DB    0
SERIAL  DB    0
WARN    DB    0
OLDBUF  DB    63
@T1     DB    0
;
; * MONITOR SECTION *
;
@SPVSR: LHLD  @TABLEF      ; INITIALIZE TABLE POINTER
        SHLD  @PNTR        ; TO BEGINNING

```



```

@LOOP    LHLD  @PNTR      ; MAIN LOOP: GET POINTER
          INX   H         ; INCREMENT
          INX   H         ; THE POINTER
          INX   H         ; BY 3 (BYTES TO BYPASS JMP)
          SHLD  $+2       ; STORE POINTER ADDRESS FOR TABLE ACCESS
          JMP   0         ; BEGIN EXECUTION AT CURRENT TABLE ENTRY

; DATA SECTION
@PNTR    DW    1         ; TABLE ENTRY ADDRESS POINTER
@TABLE   DW    $-1       ; TABLE HEADER (DEFINE TOP)
          JMP   @TE000    ; TEST FOR CONTINGENCY E000
          JMP   @TDATA    ; TEST FOR CONTINGENCY DATAREADY
          JMP   @TALAR    ; TEST FOR CONTINGENCY ALARM
          JMP   @TDATA    ; TEST FOR CONTINGENCY DATAREADY
          JMP   @TALAR    ; TEST FOR CONTINGENCY ALARM
          JMP   @TDATA    ; TEST FOR CONTINGENCY DATAREADY
          JMP   @SPVSR    ; START AT TOP OF TABLE

;
@TE000:  CALL  @E000      ; EXECUTE CONTINGENCY CODE E000
          LDA   E000      ; FETCH CONTINGENCY RESULT
          CPI   1         ; AND COMPARE TO TRUE FLAG (1)
          CZ    @ENCOD    ; EXECUTE TASK ENCODE IF TRUE
          JMP   @LOOP     ; RETURN TO MONITOR

;
@TDATA:  CALL  @DATAR     ; EXECUTE CONTINGENCY CODE DATAREADY
          LDA   DATARE    ; FETCH CONTINGENCY RESULT
          CPI   1         ; AND COMPARE TO TRUE FLAG (1)
          CZ    @SERIA    ; EXECUTE TASK SERIALIZE IF TRUE
          JMP   @LOOP     ; RETURN TO MONITOR

;
@TALAR:  CALL  @ALARM     ; EXECUTE CONTINGENCY CODE ALARM
          LDA   ALARM     ; FETCH CONTINGENCY RESULT
          CPI   1         ; AND COMPARE TO TRUE FLAG (1)
          CZ    @ALERT    ; EXECUTE TASK ALERT IF TRUE
          JMP   @LOOP     ; RETURN TO MONITOR

;
          ORG   0         ; INTERRUPT TRAP FOR ABORT
          MVI   B,1       ; POST INTERRUPT
          LHLD  ABORT     ; FLAG FOR
          MOV   M,B       ; SIGNAL ABORT
          RET                ; RESUME PROCESSING
          END              ; SOFTWARE COMPLETE

```

APPENDIX H

HARDWARE TO REALIZE THE EXAMPLE

DEVICE: INTEL 8080 8-BIT MICROPROCESSOR

CONNECTIONS:

PINS 25,26,27,29,30,31,32,33,
34,35, 1,40,37,38,39,36 = A(0:15)
PINS 10,9,8,7,3,4,5,6 = D(0:7)
PIN 21 = HLDA
PIN 12 = RESET
PIN 13 = HOLD
PIN 14 = INT
PIN 16 = INTE
PIN 17 = DEIN
PIN 18 = WR-BAR
PIN 19 = SYNC
PIN 22 = PHI1
PIN 15 = PHI2
PIN 23 = READY
PIN 24 = WAIT
PIN 2 GND
PIN 20 +5V
PIN 11 -5V
PIN 28 +12V

CLOCK GENERATOR (500NS)

DEVICE: INTEL 8224 CLOCK GENERATOR AND DRIVER FOR 8080 CPU

CONNECTIONS:

PIN 1 [RESET] = RESET
PIN 4 [READY] = READY
PIN 5 [SYNC] = SYNC
PIN 10 [PHI2] = PHI2
PIN 11 [PHI1] = PHI1
PIN 14,15 [XTAL(1:2)] = DEVICE: 18 MHZ CRYSTAL
PIN 16 [VCC] = +5V
PIN 9 [VDD] = +12
PIN 8 [GND] = GND

CPU STATUS LATCH

DEVICE: INTEL 8212 8-BIT I/O PORT

CONNECTIONS:

PINS 3,5,7,9,16,18,20,22 [DI(0:7)] = D(0:7)
PIN 4 [DO(1)] = INTA
PIN 6 [DO(2)] = WD-BAR
PIN 8 [DO(3)] = STACK
PIN 10 [DO(4)] = HLTA
PIN 15 [DO(5)] = OUT
PIN 17 [DO(6)] = M1
PIN 19 [DO(7)] = INP
PIN 21 [DO(8)] = MEMR

DATA BUS BUFFER DRIVER / LOWER HALF (4 OF 8 BITS)

DEVICE: INTEL 8216 4-BIT BIDIRECTIONAL BUS DRIVER

CONNECTIONS:

PINS 3,6,10,13 [DB(0:3)] = DB(0:3)
PIN 2 [DO(0)] = D(0)
PIN 5 [DO(1)] = D(1)
PIN 11 [DO(2)] = D(2)
PIN 14 [DO(3)] = D(3)
PIN 4 [DI(0)] = D(0)
PIN 7 [DI(1)] = D(1)
PIN 9 [DI(2)] = D(2)
PIN 12 [DI(3)] = D(3)
PIN 1 [CS-BAR] = .NOT. HLDA
PIN 15 [DIEN-BAR] = .NOT. DBIN
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND

DATA BUS BUFFER DRIVER / UPPER HALF (4 OF 8 BITS)

DEVICE: INTEL 8216 4-BIT BIDIRECTIONAL BUS DRIVER

CONNECTIONS:

PINS 3,6,10,13 [DB(4:7)] = DB(4:7)
PIN 2 [DO(0)] = D(4)
PIN 5 [DO(1)] = D(5)
PIN 11 [DO(2)] = D(6)
PIN 14 [DO(3)] = D(7)
PIN 4 [DI(0)] = D(4)
PIN 7 [DI(1)] = D(5)
PIN 9 [DI(2)] = D(6)
PIN 12 [DI(3)] = D(7)
PIN 1 [CS-BAR] = .NOT. HLDA
PIN 15 [DIEN-BAR] = .NOT. DBIN
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND

SYSTEM MEMORY

TIME DELAY TO MATCH 8111-2 LATENCY TO 8080

DEVICE: TEXAS INSTRUMENTS SN74175 HEX-QUAD D-TYPE FLIP-FLOP WITH CLEAR

CONNECTION:

PIN 9 [CLOCK] = .NOT. PHI2
PIN 1 [CLEAR] = .NOT. SYNC
PIN 16 [VCC] = +5V
PIN 8 [GND] = GND
PIN 2 [1Q] = Q
PIN 4 [1D] = +5V
READY = Q .AND. .NOT. MEMR

RANDOM ACCESS MEMORY (LOWER HALF OF PAGE 0)

DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM

CONNECTION:

PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
PIN 9 [OD] = DBIN
PIN 16 [RW] = RW
PIN 15 [CE1-BAR] = CE1
PIN 10 [CE2-BAR] = GND
PIN 9,11,12,13,14 [IO(1:4)] = DB(0:3)
CE1 = (.NOT. (OUT .AND. WR-BAR) .OR. MEMR)
.AND. (.NOT. (DECODE A(8:10) VALUE 0))
PIN 18 [VCC] = +5V
PIN 8 [GND] = GND

RANDOM ACCESS MEMORY (UPPER HALF OF PAGE 0)

DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM

CONNECTION:

PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
 PIN 9 [OD] = DBIN
 PIN 16 [RW] = RW
 PIN 15 [CE1-BAR] = CE1
 PIN 10 [CE2-BAR] = GND
 PINS 11,12,13,14 [IO(1:4)] = DB(4:7)
 PIN 18 [VCC] = +5V
 PIN 8 [GND] = GND
 RANDOM ACCESS MEMORY (LOWER HALF OF PAGE 1)
 DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM
 CONNECTION:
 PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
 PIN 9 [OD] = DBIN
 PIN 16 [RW] = RW
 PIN 15 [CE1-BAR] = CE2
 PIN 10 [CE2-BAR] = GND
 PINS 11,12,13,14 [IO(1:4)] = DB(0:3)
 CE2 = (.NOT. (OUT .AND. WR-BAR) .OR. MEMR)
 .AND. (.NOT. (DECODE A(8:10) VALUE 1))
 PIN 18 [VCC] = +5V
 PIN 8 [GND] = GND
 RANDOM ACCESS MEMORY (UPPER HALF OF PAGE 1)
 DEVICE: INTEL 8111-2 1024 BIT (256X4) STATIC MOS RAM
 CONNECTION:
 PINS 4,3,2,1,17,5,6,7 [A(0:7)] = A(0:7)
 PIN 9 [OD] = DBIN
 PIN 16 [RW] = RW
 PIN 15 [CE1-BAR] = CE2
 PIN 10 [CE2-BAR] = GND
 PINS 11,12,13,14 [IO(1:4)] = DB(4:7)
 PIN 18 [VCC] = +5V
 PIN 8 [GND] = GND

CONDITION-MODE INPUT INTERFACE HARDWARE TO SENSE SIGNAL: BUFFER
 DEVICE: INTEL 8212 8-BIT I/O PORT
 CONNECTIONS:
 PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = BUFFER, REMAINDER TO GND
 PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = DB(1:8)
 PINS2 [MD] = GND
 PINS11 [STB] = GND
 PINS1 [DS1-BAR] = .NOT. (DECODE A(8:7) VALUE 0)
 PINS13 [DS2] = INP .AND. DBIN
 PIN 24 [VCC] = +5V
 PIN 12 [GND] = GND

EVENT-MODE INPUT INTERFACE HARDWARE TO SENSE SIGNAL: ABORT
 DEVICE: INTEL 8212 8-BIT I/O PORT
 CONNECTIONS:
 NOTE: '1' = +5V, '0' = GND
 PINS 3,5,7 [DI(1:3)] = '111'
 PINS 20,22 [DI(7:8)] = '11'
 PINS 9,16,18 [DI(4:6)] = '000'
 PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = DO(1:8)
 PIN 2 [MD] = '1'
 PIN 11 [STB] = '0'
 PIN 1 [DS1-BAR] = .NOT. (INTA .AND. ABORT)
 PIN 13 [DS2] = DBIN
 OR-TIE ABORT TO INT
 PIN 12 [GND] = GND
 PIN 24 [VCC] = +5V

CONDITION-MODE OUTPUT INTERFACE HARDWARE TO ISSUE SIGNAL: SERIAL

DEVICE: INTEL 8212 8-BIT I/O PORT

CONNECTIONS:

PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = DB(1:8)
PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = SERIAL(1:1)
PINS 2 [MD] = +5V
PINS11 [STB] = GND
PINS 1 [DS1-BAR] = WR-BAR
PINS13 [DS2] = OUT .AND. (DECODE A(0:7) VALUE 1)
PIN 24 [VCC] = +5V
PIN 12 [GND] = GND

CONDITION-MODE INPUT INTERFACE HARDWARE TO SENSE SIGNAL: TEMP

DEVICE: INTEL 8212 8-BIT I/O PORT

CONNECTIONS:

PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = TEMP, REMAINDER TO GND
PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = DB(1:8)
PINS2 [MD] = GND
PINS11 [STB] = GND
PINS1 [DS1-BAR] = .NOT. (DECODE A(0:7) VALUE 2)
PINS13 [DS2] = INP .AND. DBIN
PIN 24 [VCC] = +5V
PIN 12 [GND] = GND

CONDITION-MODE OUTPUT INTERFACE HARDWARE TO ISSUE SIGNAL: WARN

DEVICE: INTEL 8212 8-BIT I/O PORT

CONNECTIONS:

PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = DB(1:8)
PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = WARN(1:2)
PINS 2 [MD] = +5V
PINS11 [STB] = GND
PINS 1 [DS1-BAR] = WR-BAR
PINS13 [DS2] = OUT .AND. (DECODE A(0:7) VALUE 3)
PIN 24 [VCC] = +5V
PIN 12 [GND] = GND

CONDITION-MODE OUTPUT INTERFACE HARDWARE TO ISSUE SIGNAL: DSPLAY

DEVICE: INTEL 8212 8-BIT I/O PORT

CONNECTIONS:

PINS 3,5,7,9,16,18,20,22 [DI(1:8)] = DB(1:8)
PINS 4,6,8,10,15,17,19,21 [DO(1:8)] = DSPLAY(1:4)
PINS 2 [MD] = +5V
PINS11 [STB] = GND
PINS 1 [DS1-BAR] = WR-BAR
PINS13 [DS2] = OUT .AND. (DECODE A(0:7) VALUE 4)
PIN 24 [VCC] = +5V
PIN 12 [GND] = GND

APPENDIX I

DISK CONTROLLER EXAMPLE

IDENTIFICATION:

DESIGNER: NICK MATELAN
DATE: 6-1-76
PROJECT: MEMORY CONTROLLER FOR SHUGART 801 FLOPPY DISK

ENVIRONMENT:

INPUT	READ DATA,1,TTL;	SEP CLOCK,1,TTL;
	SEP DATA,1,TTL;	INDEX,1,TTL;
	SECTOR,1,TTL;	
OUTPUT	WRITE DATA,1,TTL;	WRITE GATE,1,TTL,0;
	WRITE PROTECT,1,TTL;	HEAD SELECT,1,TTL;
	STEP,1,TTL;	DIRECTION,1,TTL;
DUPLX BUFFER(131),8,0,IIL;		
BINARY REGISTER,8;	BIT(8),1;	
VALID DATA,1;	BIT CNT,1;	
ARITHMETIC SECTOR COUNT,3;	SECTOR NUMBER,3;	
CURRENT TRACK,2,0;		
STRUCTURE REGISTER(BIT);	BUFFER(BIT);	

CONTINGENCY LIST:

" THESE CONTINGENCIES ARISE FROM THE USER SIDE OF THE CONTROLLER "

WHEN BUFFER(1) = 1 : 500MS DO READ;

WHEN BUFFER(2) = 2 : 500MS DO WRITE;

" THESE CONTINGENCIES ARISE FROM ACCESSING THE DISK "

WHEN SEP CLOCK = 0 : 30US, 10US DO CLOCK CHECK

WHEN VALID DATA = 0 ! SEP CLOCK = 1 : 30, 10US DO RECORD ONES

PROCEDURES:

TASK READ: "READ 128 BYTES FROM REQUESTED SECTOR TO BUFFER "

```
ARITHMETIC BYTE POINTER,3;
PERFORM POSITION HEAD; "INITIALIZE DISK HARDWARE"
BYTE POINTER ← 4; SECTOR CNT ← 0; "INITIALIZE COUNTERS"
"CANT START UNTIL INDEX MARK"
WAIT UNTIL INDEX = 0 \ 100MS;
"LOCATE CORRECT SECTOR"
VALID SECTOR: WHILE SECTOR CNT ≠ SECTOR NUMBER \ 32
DO SECTOR CNT ← SECTOR CNT + 1;
  WAIT UNTIL SECTOR = 0 \ 30MS
OD;
  "READ (BIT SERIAL) DATA INTO BUFFER"
  WHILE BYTE ADR < 131 \ 128
  BIT CNT ← 0; "START BYTE READ"
  WAIT UNTIL BIT CNT = 8; "WAIT FOR END-OF-BYTE"
  DO (WITHIN 16 MS)
    BUFFER(BYTE POINTER) ← REGISTER; "STORE DATA"
  OD;
  OD;
  BUFFER(3) ← 0; "SET STATUS"
  BUFFER(3) ← 0; "LOWER BUSY FLAG"
EXIT READ;

TASK WRITE; "WRITE 128 BYTES FROM BUFFER TO DISK"
ARITHMETIC BYTE POINTER,3;
PERFORM POSITION HEAD; "INITIALIZE DISK HARDWARE"
BYTE POINTER ← 4; SECTOR CNT ← 0; "INITIALIZE COUNTERS"
"CANT START UNTIL INDEX MARK"
WAIT UNTIL INDEX = 0 \ 100MS;
"LOCATE SECTOR"
WHILE SECTOR CNT ≠ SECTOR NUMBER \ 32
DO SECTOR CNT ← SECTOR CNT + 1;
  WAIT UNTIL SECTOR = 0 \ 30MS;
OD;
"WRITE BIT SERIAL DATA FROM BUFFER"
WHILE BYTE POINTER < 131 \ 128
DO "WRITE SPACER PULSE"(WITHIN 420US)
  WRITE DATA ← 1;
  WAIT 400US;
  WRITE DATA ← 0;
OD;
FOR BIT POINTER FROM 1 TO 8
DO "WRITE CLOCK AND DATA" (WITHIN 4 US)
  WRITE DATA ← 0 "WRITE CLOCK PULSE" ;
```

```
        WAIT 1 US;
        WRITE DATA ← 1;
        WAIT 1 US;

        WRITE DATA ← BUFFER(BYTE POINTER).BIT(BIT POINTER);
        WAIT 1 US;
        WRITE DATA ← 1;
    OD;

    BUFFER(3) ← 0; "LOWER BUSY FLAG"

EXIT WRITE;

TASK CLOCK CHECK: "WAIT FOR CLOCKED DATA PULSE"

    VALID DATA ← 1; "SET FLAG FOR DATA ACCESS"
    BIT CNT ← BIT CNT + 1; "UPDATE DATA COUNTER"

EXIT CLOCK CHECK;

TASK RECORD ONES: "STORE DATA BITS THAT ARE ONE "

    REGISTER.BIT(BIT CNT) ← 1
    VALID DATA ← 0; "RESET FLAG FOR DATA ACCESS"

EXIT RECORD ONES;

TASK POSITION HEAD: "CALCULATES SECTOR AND TRACK, LOWERS HEAD"

    BUFFER(3) ← 1; "SET BUSY FLAG"
    WRITE GATE ← 0; "INHIBIT WRITING"
    BUFFER(1) ← 0; "REMOVE POSTED REQUEST"

    "TEST REQUEST"
    IF BUFFER(2) < 0 ? BUFFER(2) > 76*32
    THEN "SET ERROR FLAG" BUFFER(3) ← 2;
        "AND QUIT" EXIT POSITION HEAD FI;
    "CALCULATE TRACK NUMBER, 76 TRACKS OF 32 SECTORS EACH OF 128 BYTES"
    TRACK = BUFFER(2) / (76 * 32 * 128)
    SECTOR NUMBER = (BUFFER(2) - (TRACK * 32 * 128)) / 128

    "PUT HEAD DOWN"
    HEAD SELECT ← 1;
    STEP ← 1;

    "SEEK TRACK"

    IF TRACK = CURRENT TRACK THEN "DONE" EXIT POSITION HEAD FI;

    IF TRACK < CURRENT TRACK THEN DIRECTION ← 0 FI;
    IF TRACK > CURRENT TRACK THEN DIRECTION ← 1 FI;

    "MOVE HEAD TO TRACK"

    WHILE TRACK ≠ CURRENT TRACK \ 76
    DO WAIT 10 MS;
        STEP ← 0; "START HEAD MOVING"
```



```
WAIT 10US "PULSE WIDTH";  
STEP ← 1;  
WAIT 10MS;"SETTLE TIME";
```

OD:

EXIT POSITION HEAD;

DISTRIBUTION LIST

Internal

R. E. Batzel, L-1
 L. L. Cleland, L-156
 M. P. Ekstrom, L-156
 E. A. Lafranchi, L-151
 H. H. Loomis, L-156 (3)
 W. G. Magnuson, L-156 (10)
 H. C. McDonald, L-161
 R. J. Smith, L-156 (10)
 R. H. Wyman, L-263
 ERD File, L-156 (2)
 TIC, Oak Ridge (27)
 TID (15)

External

M. N. Matelan (5)
 1824 Las Vegas Trail, #272
 Ft. Worth, TX 76101

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research & Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights.

NOTICE

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Energy Research & Development Administration to the exclusion of others that may be suitable.

Printed in the United States of America
 Available from
 National Technical Information Service
 U.S. Department of Commerce
 5285 Port Royal Road
 Springfield, VA 22161
 Price: Printed Copy \$; Microfiche \$2.25

Page Range	Domestic Price	Page Range	Domestic Price
001-025	\$ 3.50	326-350	10.00
026-050	4.00	351-375	10.50
051-075	4.50	376-400	10.75
076-100	5.00	401-425	11.00
101-125	5.50	426-450	11.75
126-150	6.00	451-475	12.00
151-175	6.75	476-500	12.50
176-200	7.50	501-525	12.75
201-225	7.75	526-550	13.00
226-250	8.00	551-575	13.50
251-275	9.00	576-600	13.75
276-300	9.25	601-up	*
301-325	9.75		

*Add \$2.50 for each additional 100-page increment from 601 to 1,000 pages:
 add \$4.50 for each additional 100 page increment over 1,000 pages.

Technical Information Department

LAWRENCE LIVERMORE LABORATORY

University of California | Livermore, California | 94550