CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Vehicle-Based Control Computer Systems

David M. Auslander

University of California, Berkeley

California PATH Research Report
UCB-ITS-PRR-95-3

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

January 1995

ISSN 1055-1425

Vehicle-Based Control Computer Systems

Final Report - September 1, 1994
PATH Project MOU-61

David M. Auslander
Mechanical Engineering Department
University of California
Berkeley, CA 94720
510-642-4930(office), 510-643-5599(fax)
email: dma @ euler. berkeley .edu

Abstract

This report is in two parts: the first part (starting at Section 1) describes a design and implementation methodology for real time software suitable for control of mechanical systems such as vehicles. This method provides for a design description of the system, a means of capturing the system structure in such a way as to modularize the software writing job, and a portable implementation method. The second part (starting at Section 13) is concerned with a particular problem in mechanical system control: estimating velocity when the event rate from a digital encoder is lower than the controller's sample time. Methods are explored for performing this estimation that achieve best accuracy overall, but can also avoid the problem of spurious estimates when the velocity reverses.

Table of Contents

1 Part I: Design and Implementation of Real Time Software for Mechanical Control

Use of an explicit, formalized design layer is critical to the development of reliable real time software. This section describes such a method, which is a candidate design method for systems of modest complexity. The context for the use of the method is described first, then its structure is described, the associated software, and examples.

2 Mechanical System Control: Mechatronics

Mechanical system control is undergoing a revolution in which the primary determinant of system function is becoming the control software. This revolution is enabled by developments occurring in electronic and computer technology. The developments in electronics have made it possible to energetically isolate the four components making up a controlled mechanical system:

- The target system
- Measurement
- Computation
- Actuation

Once isolated from the instruments on one side and the actuators on the other, computation could be implemented using the most effective computing medium, independent of any needs to pass power through the computational element. That medium has been the digital computer, and the medium of expression for digital computers is software.

This ability to isolate is recent. Watt's famous speed governor, for example, combined the instrument, computation and actuation into the flyball element. Its computational capability was severely limited by the necessity that it pass power from the power shaft back to the steam valve. Other such examples, where computation is tied to measurement and/or actuation, include automotive carburetors, mastered cam grinders, tracer lathes, DC motor commutation, timing belts used in a variety of machines to coordinate motions, rapid transit car couplings (which are only present to maintain distance; cars are individually powered), and myriads of machines that use linkages, gears, cams, etc., to produce desired motions. Many such systems are being redesigned with software based controllers to maintain these relationships, with associated improvements in performance, productivity (due to much shorter times needed to change products), and reliability.

The term *mechatronics*, attributed to Yasakawa Electric in the early 1970s, was coined to describe the new kind of mechanical system that could be created when electronics took on the decision-making function formerly performed by mechanical components. The phenomenal improvement in cost/performance of computers since that time has led to a shift from electronics to software as the primary decision-making medium. With that in mind, and with the understanding that decision-making media are likely to change again, the following definition broadens the concept of mechatronics while keeping the original spirit of the term:

The application of complex decision-making to the operation of physical systems.

With this context, the compucentric nature of modern mechanical system design becomes clearer. Computational capbilities and limitations must be considered at all stages of the design and implementation process. In particular, the effectiveness of the final production system will depend very heavily on the quality of the real time software that controls the machine.

3 Real Time Software

Real time software differs from conventional software in that its results must not only be numerically and logically correct, they must also be delivered at the correct time. A design corollary following from this definition, is that real time software must embody the concept of ***duration,*** which, again, is not part of conventional software. The real time software used in most mechanical system control is also ***safety-critical.*** Software malfunction can result in serious injury and/or significant property damage, In discussing software-related accidents which resulted in deaths and serious injuries from clinical use of a radiation therapy machine (Therac-25), Leveson and Turner (1993) established a set of software design principles, "...that apparently were violated with the Therac-25...." Those are:

- Documentation should not be an afterthought.
- Software quality assurance practices and standards should be established.
- Designs should be kept simple.
- Ways to get information about errors -- for example, software audit trails -- should be designed into the software from the beginning.
- The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate.

In particular, it was determined that a number of these problems were associated with ***asynchronous*** operations, which while uncommon in conventional software, are the heart and soul of real time software. Asynchronous operations arise from preemptive, prioritized execution of software modules, and from the interaction of the software with the physical system under control.

Because of preemptive execution, it becomes impossible to predict when groups of program statements will execute relative to other groups, as it is in synchronous software. Errors that depend on particular execution order will only show up on a statistical basis, not predictably. Thus, the technique of repeating execution until the source of an error is found, which is so effective for synchronous (conventional) software, will not work with for this class of real time error.

In a like manner, there are errors that depend on the coincidence of certain sections of program code and events taking place in the physical system. Again, because these systems are not strictly synchronized, they can only have statistical characterization.

3.1 Engineering Design / Computational Performance

Too often, the only detailed documentation of real time software is the program itself. Furthermore, the program is usually designed and written for a specific real time target environment. The unfortunate consequence of this is that the engineering aspects of the design problem become inextricably intertwined with the computation aspects. This situation relates directly to the first design principle listed above, “documentation should not be an afterthought.” If the system

engineering is separated from the computational technology, then the documentation will have to exist independent of the program; otherwise, documentation can be viewed as an additional burden.

The following definitions will be used to separate these roles as they appear in the proposed methodology:

> System engineering: Detailed specification of the relationship between the control software and the mechanical system.
>
> Computational technology: A combination of computational hardware and system software that enables application software based on the engineering specification to meet its operational specifications.

Using these definitions, the engineering specification describes ***how*** the system works; the computational specification determines its *performance*. As a result of this separation, if a suitable paradigm is adopted to describe the engineering specification, a much broader discussion and examination can be brought to bear because, the ***details*** of the engineering can be discussed by project participants familiar with the problem, not just those familiar with computing languages and real time programming conventions. This provides for meaningful design review of projects that are software intensive.

3.2 Software Portability

In mechanical system control, portability has consequences both for product lifetime and for the design/development cycle. The mechanical part of the system can have a commercial lifetime of anywhere from 5 to 20 years. On the other hand, the computational technology used for its control only has a lifetime of 3 to 5 years. To remain competitive, new versions of the product need to use new computers to take advantage of the ever-increasing computational capability. Doing this cost effectively requires software that will "port" easily from the original target processor to new ones.

Software portability seriously affects the design/implementation cycle as well. Early stages of the software tend to be simulations, done to test hypotheses and to substitute for hardware not yet built. Later stages use laboratory prototypes, then pilot prototypes, then, finally, the actual production system. If software can't migrate from step-to-step in this process, the whole process can become greatly elongated as new software must be created for each step, and there are significant possibilities for introducing new bugs at each stage.

Portability is complicated by the real time constraints. If real time software environments are used as aids in meeting those constraints (kernels, schedulers, real time operating systems), software written for one environment can require substantial rewriting to run in another. Crossing the full spectrum from simulation to production traverses environments in which program structure itself is incompatible. The proposed methodology provides a layer of abstraction one higher than the real time environments, so offers a means of bypassing these incompatibility problems.

Further portability challenges arise from the operator interface section of the program, and the inter-program communication for those solutions implemented with multiple processors. These subjects will be discussed briefly here, but remain in need of substantial further work.

3.3 Control System Organization

A two-level organization is used on both the engineering and computational sides of the control software design. On the engineering side, a job is organized into ***tusks*** and ***states***, and, on the computational side, into processes and ***threads.***

The breakdown of the engineering specification into tasks and then states is a subjective process, requiring considerable engineering judgement. This stage is the primary creative design step. Tasks represent ***units*** of ***work,*** and, roughly speaking, can be viewed as cells in a matrix that breaks the control system down along a time-scale dimension and a physical component dimension. For example, along a column representing a motor-driven axis, there could be separate tasks to handle a high speed input (such as an incremental encoder), an actuation output, a servo control, and, perhaps, a motion profiler. Still higher in that column, there could be a coalescence as tasks handled coordinated motion of several axes.

Tasks, in general, are active simultaneously. They are used to describe the parallelism inherent in the physical system. Internally, however, tasks are organized in a strictly sequential manner into states. States describe specific activities within the task; only one state can be active at a time. The primary reason for this distinction between the nature of tasks and states is that sequential activity is a part of many mechanical systems. Even for systems that do not seem to be fundamentally sequential, such as process or web systems, the sequential breakdown within tasks seems to work quite well. During normal operation, tasks tend to stay in only one or two states. However, during startup and shutdown, the task/state structure describes the operation very well. The sequential state structure also serves as one test for appropriate task definition. If tasks are aggregated to much, they will end up with parallel operations which cannot be described effectively by states. Breaking the task into several smaller tasks solves the problem.

On the computational side, ***processes*** describe computational entities that do not share an address space (this includes independent processors). ***Threads,*** are computational entities that share an address space but can execute asynchronously. A thread can contain one or more tasks; a process can contain one or more threads.

The organization into processes and threads is purely for performance purposes. As is noted below, there is no theoretical necessity for such organization at all. The organization serves only to meet performance specifications when the chosen processor is not sufficiently fast to meet the specs with a single-thread structure. It does this by shifting processor resources from low priority to high priority tasks. As a further note, the portability requirements enumerated above imply that several different computational organizations will be used in the course of a development project.

4 State Transition Logic

The task organizational structured described in this paper is an adaptation of state transition logic (Auslander, 1993a). This adaptation provides for implementation of the structural principles enumerated above. Documentation development is integrated with the software specification procedure, the software produced is inherently modular, and audit trail information can be produced automatically. By extending the transition logic paradigm to cover the entire realm of real time requirements (Auslander, 1993b), two important results are achieved:

1) The discipline associated with following a formal design paradigm is extended to the "low level" as well as high level software tasks.

2) It becomes possible to produce portable code, that is, code which can be generated and compiled to run in a number of different real time environments without changing the user-written portion of the code.

State transition logic is formally defined within finite automata theory (Kohavi, 1970). As used in the design of synchronous sequential circuits, it becomes a formal, mathematical statement of system operation from which direct design of circuits can be done (Sandige, 1990, or any basic text on logic design). When used as a software specification tool, state transition logic takes on a more subjective cast; the transition logic specifies overall structure, but specific code must be written by the user (Domfeld et al, 1980, Bastieans and Van Campenhout, 1993).

The state transition logic concept has been further specialized to mechanical system control through the specification of a functional structure for each state. This structure specifies the organization of code at the state level so that it corresponds closely with the needs of control systems.

The use of transition logic has also been based on the very successful applications of programmable logic controllers (PLCs). These devices, in their simplest form, implement Boolean logic equations, which are scanned continuously. The programming is done using ladder logic, a form of writing Boolean equations that mimics relay implementation of logic. In basing real time software design on transition logic, each state takes on the role of a PLC, greatly extending the scope of problems that can be tackled with the PLC paradigm.

Depending on the nature of the problem being solved, other formulations have been proposed. For example, the language SIGNAL (Le Guemic et al, 1986) was invented for problems which have signal processing as a core activity. Benveniste and Le Guemic (1990) generalize the usage to hybrid dynamical systems.

4.1 States and Transitions

State specifies the particular aspect of its activity that a task is engaged in at any moment. It is the aspect of the design formalism that expresses ***duration.*** States are strictly sequential; each task is ***in*** one state at a time. Typical activities associated with states are:

- Moving - a cutting tool moving to position to start a cut, a carriage bringing a part into place, a vehicle holding a velocity.

- Waiting - for a switch closure, for a process variable to cross a threshold, for an operator action, for a specified time.
- Processing - thermal or chemical processes, material coating in webs.
- Computing - where to go, how long to wait, results of a complex measurement.
- Measuring - size of a part, location of a registration mark, object features from vision input, proximity.

Each state must be associated with a well-defined activity. When that activity ends, a ***transition*** to a new activity takes place. There can be any number of transitions to or from a state. Each transition is associated with a specific condition. For example, the condition for leaving a ***moving*** state could be that the end of the motion was reached, that a measurement indicated that further motion was not necessary, that an exception condition such as stall or excessively large motion error occurred, etc.

4.2 Transition Logic Diagrams

State transition logic can be represented in diagrammatic form. Conventionally, states have been shown with circles, and transitions with curved arrows from one state to another. Each transition is labelled with the conditions that specify that transition. This format is inconvenient for computer-based graphics, so a modified form, shown in **Figure 1,** is used.

This diagram shows a fragment of the transition logic for a task that controls the movement of a materials handling vehicle. The vehicle moves from one position to another, picking up parts in one position and dropping them off at another. The states are shown with rectangles; a description of the state is given inside the rectangle. The transitions are shown with arrows and the transition conditions are shown inside rounded rectangles. The dashed lines provide an unambiguous indication of which transition the condition is attached to. The first "move-to" state shows a typical normal transition as well as an error transition, in this case based on a time-out condition,

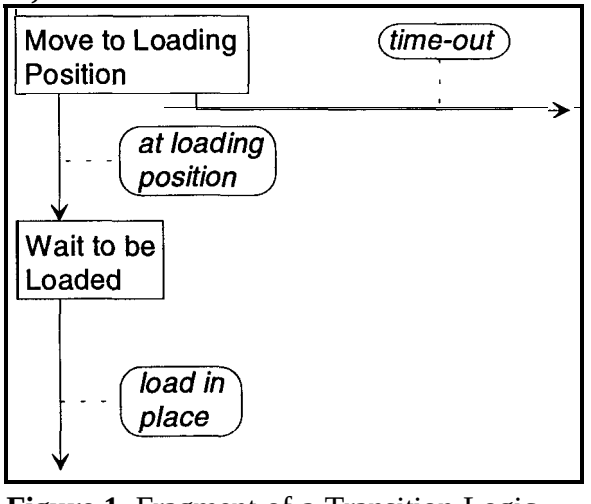

Figure 1. Fragment of a Transition Logic Diagram

Although these diagrams are not essential in using transition logic, they are an excellent visualization tool. If a task is compact enough to fit a logic diagram on a single page, the graphical description makes its function much easier to grasp.

4.3 Block/Non-Block

A major feature of PLCs contributing to their success as control components has been that the logic function is continually scanned. The programmer does not deal with program flow control, as must be done when using conventional programming languages. As long as the ladder is active, it is scanned repeatedly, so the user only has to be concerned with the fundamental performance issue of whether the scan rate is adequate for the particular control application.

Transition logic design is based on this same scanning principle for execution of state-related code. In order to achieve greater functional flexibility than is possible with ladder diagrams, however, standard sequential languages are used for coding. To implement a scanning structure with algorithmic languages requires the restriction that only ***non-blocking*** code can be used. Non-blocking code is a section of program that has predictable execution time; execution time for blocking code cannot be predicted. Examples of blocking code in the C language include, for example, the *scanf()* function call used to get keyboard input from the user. Scanf only returns when the requested input values have been typed; if the user goes out for coffee, the function simply waits. Likewise, the commonly used construction to wait for an external event such as a switch closure,

```
while(inbit(bitno) == 0) ;
```

is also blocking. If the event never happens, the while loop remains hung.

The restriction to non-blocking code, however, does not cause any loss of generality. Quite the contrary, the transition logic structure is capable of encoding any kind of desired waiting situations, as shown in the example given above. By encoding the "wait" at the transition logic level rather than at the code level, system operations are documented in a medium that any engineeer involved in the project can understand without having to understand the intricacies of the program.

4.4 State-Related Code

The transition logic metaphor encourages the use of modular software by associating most of the user-written code with states. In addition, a formal structure of functions is established for this state-related code. Two goals of modular code writing are thus fulfilled:

a) Sections of code are directly connected to identifiable mechanical system operations,
b) Individual functions are kept short and easily understood.

For each state, the following functions are defined:

Entry function: Executed once on entry to the state.
Action function: Executed on every scan of the state.

For each transition from the state, the following pair of functions is defined:

Test function: Test the condition for transition; returns TRUE or FALSE.
Exitfunction: Executed if the associated transition is taken.

This structure enforces programming discipline down to the lowest programming level. All of these functions must be non-blocking, so, ***test*** functions, for example, never wait for transition conditions. They make the test, then return a logic value. Relating code to design-level documentation is also enforced with this structure. Transition logic documentation for each task identifies states in terms of what the mechanical system is doing. Not only is the code relevant to that state immediately identified as well, the code is further broken into its constituent parts.

4.5 State Scanning: The Execution Cycle

The state scan is shown in **Figure 2.** In addition to establishing the execution order for the state-related functions, it also provides the basis for parallel operation of tasks.

Each pass through the cycle executes one scan for one task. If this is the first time the scan has been executed for the current state, the ***entry*** function is executed. The ***action*** function is always executed. Then, the first ***transition test*** function is executed. If it returns TRUE to indicate that the transition should be taken, the associated ***exit*** function is executed and a new state is established for the next scan. Otherwise, subsequent ***test*** functions are executed in a similar manner. The first ***test*** function returning TRUE terminates the sequence. Thus, if more than one transition became TRUE at the same time, the one associated with the earliest ***test*** function would be recognized.

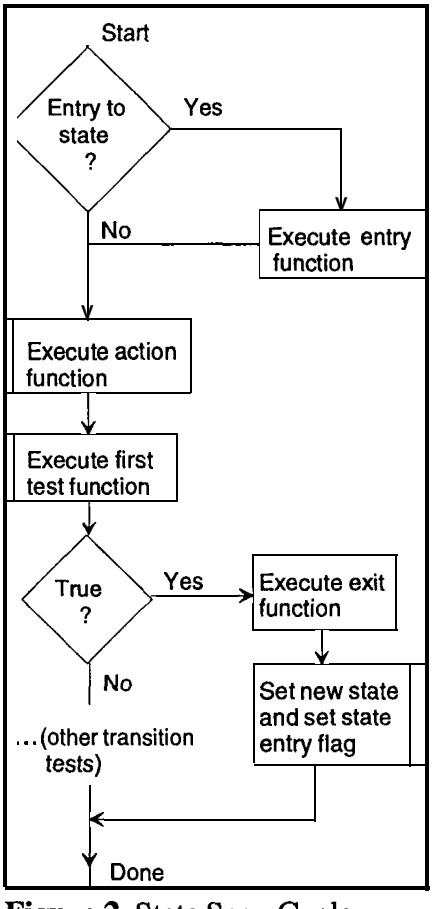

Figure 2. State Scan Cycle

Behind the execution details, there must be a data base of task information. Each task must have a data table specifying its structural information, that is, all of the states and transitions, task parameters such as priority, sample time, etc., and the transient information such as present state and status.

4.6 Task Concurrency: Universal Real Time Solution

Tasks, as noted above, must operate concurrently. This structure provides for parallel operation of tasks, even in the absence of any specific multitasking operating system or scheduler. Because all of the state functions are non-blocking, the scan cycle itself is non-blocking. It can, therefore, be used to scan each active task in succession. After finishing with all of the tasks, the first task is scanned again. This guarantees fully parallel operation of all tasks. This method of scheduling, ***cooperative multitasking,*** will be an adequate solution if the total scan time for all tasks is short enough to meet the system timing constraints. If not, a faster computer must be used, or other scheduling solutions must be found. These will be discussed below.

The methodology discussed thus far therefore presents a universal real time solution. It is capable of solving all real time problems, without any special real time constructs or operating systems, *if*

a fast enough computer is available. All of the usual real time facilities, semaphores, task synchronization, event-based scheduling, etc., can all be implemented using the formalism of transition logic, with all code non-blocking. If a fast enough computer is not practical, use of preemptive scheduling based on interrupts can be implemented. To do this, the transition logic paradigm must be extended to include designation of task-type, and library code must be included to allow integration of transition logic based code with a variety of real time and/or multiprocessor environments.

5 Example: Pulse-Width Modulation (PWM)

Pulse-width modulation is widely used as an actuation function where there is need for a digital output to the actuating device, but continuous variation in the actuator output is desired. Anytime the actuator plus arget system constitute a low-pass filter, PWM can perform the function of a digital-to-analog converter by exploiting the termporal properties of the low-pass filtering. The PWM signal is usually a retangular-wave of fixed frequency, with variable duty-cycle (i.e., ratio of on-time to cycle time). The logic diagram in **Figure** 3 shows a task to implement PWM.

The task has four states, as shown, and will produce an effective low frequency PWM from software. The maximum frequency depends on the means of measuring time that is utilized, and the timing latencies encountered in running the task. It would be suitable for actuation of a heater or, perhaps, a large motor, but would be too slow for a modest sized motor.

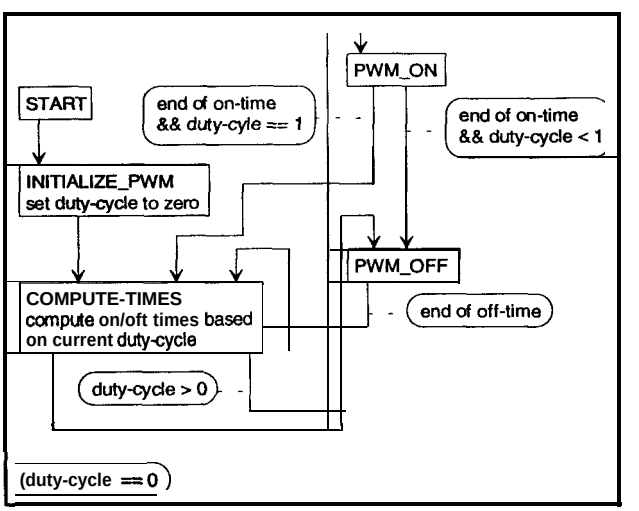

Figure 3. PWM Task

The two main tasks, PWM_ON and PWM_OFF, turn the output on (or off) on entry, and then just wait for the end of the specified time interval. COMPUTE-TIMES is active once per cycle to find the appropriate on and Off times in the event that the duty cycle has been changed. The transition conditions take account of the two special cases -- duty cycles of 0 (never on) and 1 (always on) in addition to normal operation.

This example shows the use of transition logic for a task that is quite low level computationally. Unlike a conventional implementation of such a task, however, the details are readily apparent from the transition logic and reference to the code (not shown here) need only be made to confirm that it accurately translates the transition logic.

6 Transition Logic Design Tool

Manual programming of systems based on transition logic is possible, that is, programming from a transition logic diagram or tabular description with no software development tools other than a compiler, editor and linker. There are two major disadvantages to this, however. First, it can be very

tedious because of the large numbers of functions that must be created (and named) and the data tables that must be made to do the bookkeeping the scan execution cycle needs. Second, there is always the temptation, particularly when one is in a hurry, to make changes directly in the program without updating the transition logic. This latter is especially pernicious because it leads to documentation that misrepresents system operation.

The design tool described here aids in the full program production by keeping track of tasks and transition logic, organizing the writing of state-related code, and generating C files that can be compiled to operate in a variety of environments, including multi-processor systems. It also produces system documentation, with the transition logic listed in tabular rather than graphical format. A schematic of the design and implementation cycle using this software tool is shown in **Figure 4.** It currently runs in MS Windows, but is written using the XVT portable library, so, potentially, can be recompiled for most common operating systems.

6.1 Task Specification

The software design tool (referred to as TRANLOG) keeps track of a single project consisting of any number of tasks. Each task is defined by:

Name
Description (one line)
Task type
Priority
Initial status
Initial state
Data items (if any)
Transition logic structure
Task-related code

The ***tusk type*** and ***priority*** are not necessary for all execution environments, but are included so portability will be maintained. The types of tasks will be defined in the Multitasking section, below. As the program executes, a task can be active, inactive, waiting, or executing. The ***initial status*** determines whether the task will be active or inactive when the program first starts. Similarly, the ***initial state*** determines which state will be entered first each time the task is activated (if a task is de-activated, re-activating it is the same as starting it for the first time). The ***data items*** are related to task type and can specify such infomation as input ports, sample time, etc.

The details of the transition logic structure are described below. The ***tusk-refuted code*** consists of code items that support the states. These include functions used by several states, variables that are common across the task, defined constants, etc.

There is no formal structure describing the relationships of the tasks. It is programming convention, however, to designate one task as the ***muster tusk*** to make sure the system is properly (and safely) initialized. In defining the tasks, only the master task has its initial status set to active; all other tasks are initially inactive. The master task can then activate other tasks in an order the guarantees proper

system start-up. The master task is usually responsible for de-activating and re-activating tasks when necessary as the system proceeds through various operating modes.

6.2 State and Transition Specification

Specification of a state requires a name for the state and a short description. Transitions are defined between already defined states, so it is usually easiest to define all of the states in a task before defining the transitions. Other than associated code, neither states nor transitions require any further information.

The code for states and transitions can be entered while defining the transition logic, or can be entered or edited using an external editor (usually the program development environment that is part of the compiler). A merge facility is supplied so that changes made outside of the transition logic development tool can be captured.

6.3 Task and Common Code

In addition to the entry, action, test and exit functions themselves, it is usually necessary to add some code that is common to sections of the system. There are three places to do ***this.*** the ***tusk code*** section is for code that is specific to a single task. It is most often used to define ***static*** variables that are used across many state functions. Because the non-blocking code of transition logic uses functions that return immediately, information that must be retained within or across states must be declared as ***static,*** which is normally done in the ***tusk code*** section. Functions that are only used in a single task can also be defined as part of ***tusk code.***

Because the control program created is portable, the tasks can execute in many different environments, ranging from cooperative multitasking in which all tasks execute in the same computational thread, to each task on an independent process, connected, for example, through a network. Thus, in general, tasks will not share the same computational address space so a separate definition is required for any code that will be common to more than one task. The ***common code*** and ***common header*** sections are provided for this purpose. When the code is linked, the common code and header sections are linked with every computationally independent component. These sections are most commonly used to define functions that are needed in more than one task. The function definitions themselves are put in the ***common code*** section, and the function prototypes are in the ***common header*** section.

6.4 Code Generation

Code generation produces a separate source file for each task, files for the common code and common header, and an additional file with internal control information. The target language is C, although it could be modified for any other language with sufficient syntax to support the task and state descriptions. The major jobs done by the code generator are naming all of the state functions and producing data objects that describe each of the tasks and each of the states. These data objects are linked lists containing all of the topological and status information associated with tasks and states. They include the transition information, pointers to relevant functions, etc.

The code generated at this step is still completely portable. The compile/link operation customizes the program for the particular environment it will run in. The usual procedure is to provide ***make*** files for whatever environments are supported.

6.5 Compile, Edit and Merge

The compile, edit, link sequence is shown in **Figure** 4. As noted above, it is usually desirable to enter all of the states before entering the transitions. The initial code entry is often done while using the transition logic tool, as shown in the figure, although it is not mandatory. After generating code and attempting a compile, it is possible to edit the source code directly, for example, using the interactive development environment of the compiler. This is practically a necessity; it is much too awkward to have to go back to the transition logic tool each time an edit is necessary.

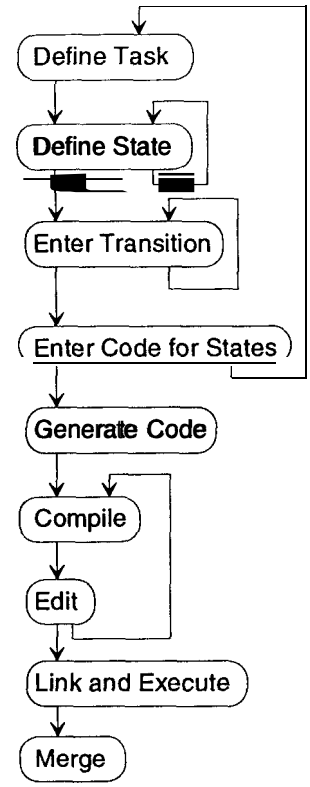

Figure 4. Design and Implementation Sequence

Once this editing is done, however, it is necessary to capture the editing back into the transition logic source file. This is done with the merge step, which is part of the transition logic tool. It copies all user-generated code in the C language source files back into the transition logic source file. The only rules for assuring that this is done successfully are that code is only entered in marked locations in the C source files, and that the names of the files are not changed.

The transition logic tool can be invoked at any time to change or add to the task or state structure. New code is then generated to build a test program.

6.6 Execution Shell

The execution shell is the key to portability. It uses conditional compilation to build programs for any real time environment that is supported. For the simplest environments, it contains all of the necessary scheduling algorithms, giving a completely self-contained program. To interface with other environments, appropriate code structure and library calls are set up to operate in the selected environment. By using this approach, it is possible to keep the user-written code completely portable -- no changes at all are required to recompile for a new operating environment.

The compilation and linking procedure is normally controlled using a MAKE facility associated with the target compiler. At present, creation of MAKE files is not automated.

6.7 Documentation

Either working or archival documentation can be produced from the transition logic tool. Working documentation includes a list (text) description of the entire system. For each task, all of the task parameters are listed (name, type, prioirity, initial status,etc.), as well as the task code (the code that is placed at the top of the task file). Then, for each state in the task, all of the transitions from that state are listed along with the description of the reason for the transition. The archival form of the

documentation adds all of the user-written code to the information given in the working documentation.

The working documentation can be used while developing the program, since the code is easily accessible from other routes, for example, using the compiler's editor. The archival documentation contains sufficient information to completely reconstruct the program, even if all of the source files were to disappear. It is, however, harder to read than the working form.

7 Task Organization

The primary form of organization within tasks is the transition logic model. In addition, however, there is need for code that is common within a task or across some or all tasks, which is handled by task-related coded and common code. The file organization of one file per tasks is exploited in defining this code. Inter-task organization includes issues of data exchange among tasks and the control of how and when to activate and deactivate tasks. The former, data exchange, is handled formally, while the latter is handled informally.

7.1 Task-Related Code

This code has been referred to in the description of the transition logic tool itself. It is code that must be within the scope of all parts of the task. Because tasks are complete in a single file, this code can be placed at the top of that file. The most common use of ***this*** code area is to define ***static*** variables. The nature of transition logic coding encourages the use of static variables. If, for example, a variable is initialized in an *entry* function and then acted on in ***an action*** function, it must be declared as static. For variables that ***are local*** to a single ***action*** function, ***static*** class may still have to be used because the function is continually scanned, although the variable need not be visible to the rest of the file. Functions that are used only by a single task can also be defined in the task-code section.

7.2 Common Code

Code that has to be visible to more than just one task goes into ***common*** code. This is, in fact, two sections. One section is common *header* and the other ***common*** code. The common header is brought into every task file with an ***include*** statement; the common code is linked to every independent thread.

The ***common code*** section is most commonly used to define functions of general utility, so they will be available to all tasks. The ***common header*** is used mostly for prototypes of those functions and ***defines.*** Note that global variables (*externs*) are not permitted, so this section is not used for global definitions (see the Communication section, below).

7.3 Task Process Configuration

How tasks are distributed among processes is a design decision made by the user. The configuration information must be entered to the transition logic design tool. Any number of configurations can be defined, but one of them must be selected when code is generated. The code generation process puts configuration information into the C source files so it can be used by the execution shell (or shells for a multiprocessor configuration). The default configuration places all tasks in a single process. No user action is required to pick this configuration.

7.4 Intertask Communication

Tasks in a control system exist as part of the solution to a job. As such, tasks will necessarily need to exchange information. The design of a means of implementing that exchange is most severely affected by the need for portability of the user-written source code. In addition, there is the fundamental need to protect the integrity of the data, which can be at risk whenever asynchronous computing components exist. Because of these concerns, intertask data communication is treated formally, through the transition logic design tool. Details are given in Section 8.6.

7.5 The Master Task

A means is needed to activate and deactivate tasks. In particular, virtually all jobs have the problem of orderly task activation during start-up so that, for example, power supplies are turned on before the components they supply power to, and controllers are initialized before setpoints are changed. Activation and deactivation are done with function calls and so are embedded in the code structure rather than being treated formally.

As a convention for handling this chore, a single task is designated ***as*** the ***master*** task. In specifying the tasks, only the master task is given an initial status of active; all other tasks are initially inactive. The master task can then have a state structure that turns the other tasks on in a safe manner, including suitable delays and checks to make sure that conditions are proper as each task is activated. Likewise, the master task can orchestrate shutdown so safe conditions are maintained. In between, it can also be responsible for deactivating tasks that are not needed, although it is not always clear whether deactivating is significantly more efficient than giving the task an idle state that waits for the event that triggers the task's next activity.

In relatively simple jobs, it makes sense to have the master task control the operator interface as well, since the context changes in the operator interface follow major changes in task activity very closely.

It should be noted that the this structure is not enforced by any system formalism. It is only a suggested mechanism for task organization.

8 Multitasking Performance: The Real World

The "universal real time solution" as outlined above depends on adequate computational speed to juggle all of the simultaneous activities without regard for the importance or computational characteristics of any of the individual tasks. The real world is not often so kindly! Taking acount of these real world concerns, however, only requires modest additions to the design structure as it has been developed thus far. These additions allow for a stronger connection between the engineering system design and the computational technology so that system performance specifications can be met.

8.1 Priority-Based Scheduling - Resource Shifting

The essence of living within a finite computing budget is to make sure enough of the computational resource budget is expended on the highest priority tasks. The first addition to the design structure, therefore, is to associate priorities with the tasks. The job of the computational technology, then, is to shift resources away from the low priority tasks and towards the high priority tasks.

How that is to be done depends on the criteria for success. There are at least two ways of measuring performance,

- progress towards a computational goal
- execution within a certain window

These lead to different computational strategies and to fundamental task classifications.

8.2 Continuous/Intermittent Tasks

These two performance measures suggest a division of tasks according to performance type. ***Continuous*** tasks measure success by progress. They will absorb as much computing resource as they are given. ***Intermittent*** tasks respond to explicit events and have a limited computing function to perform for each event occurrence. Their success is measured by whether the response to the event occurs within a specified time of the event itself.

The universal solution uses only continuous tasks. The nature of a task's response to events is embedded in its transition structure, so no distinction needed to be made in task classification. Using the continuous/intermittent classification, it will be necessary to specify the event to which an intermittent task responds explicit, that is, part of the transition logic design tool, rather than implicit. Thus, there will be a further subdivision within the intermittent tasks; no subdivision of continuous tasks is needed.

Continuous tasks encompass that portion of a control job that is most like conventional computing in the sense that the goal is to deliver a result as soon as possible rather than at a specified time. They thus form the lower priority end of the real time spectrum. Priority distinctions within the continuous tasks can be handled by, on average, giving more resource to the higher priority tasks. Within the "universal" approach, which is still applicable to the continuous tasks, priorities of continuous-type tasks can be recognized by giving high priority tasks more scans than lower priority tasks.

A different approach is required for the intermittent tasks. In the simplest computing environment, that is, a single computing thread, intermittent tasks are distinguished from continuous tasks by two properties,

1) They are only given computing resource (i.e., transition logic scans) when the triggering event has occurred, and,

2) Once they are triggered they are given as many scans as are necessary to complete their activity.

This change alone is sometimes enough to meet performance specifications for a job that could not meet specifications using all continuous tasks (the universal approach). In particular, if the problem is that once triggered, the high priority task does not gets its output produced within the specified window, this will help meet specification because as an intermittent task it will be given all of the computing resource until it completes the response to the event. If, on the other hand, the problem

is one of ***latency,*** that is, the task does not always start soon enough after the triggering event to meet the specification, then other remedies will be required.

8.3 Cooperative Multitasking Modes

"Multitasking" is a generic term in computing referring to the ability to do more than a single activity at a time. It is usually applied to single-processor configurations. In that context, since digital computers are by their nature only capable of doing one thing at a time, the simultaneity comes from rapid (with respect to the observer) switching from one activity to another, rather than truly simultaneous operation. In the context of real time control software, this definition must be further refined to reflect both the engineering and the computational categories defined above.

First, to map from the engineering system description to the computational implementation, the engineering ***task*** is the indivisable computing unit. Thus, each computing thread will contain at least one task.

Within any thread that contains more than one task, some form of ***scheduling*** must be implemented to share the threads computational resource among its tasks. Two forms of scheduling have been implemented for multiple tasks inside a single computing thread, ***sequential,*** and ***recursive.*** Since they exist in one thread, these schedulers are referred to ***as cooperative multitasking schedulers.*** The term "cooperative" is used to describe the fact that the scheduler can never preempt a running program, but can only perform its function when tasks voluntarily return control back to the scheduler.

The success of cooperative schedulers depends on the voluntary return of control, which in standard programming languages is difficult to design consistently. Cooperative schedulers can be used very effectively in the transition logic context, however, because the state functions (entry, action, test, exit) are all specified as nonblocking. Thus, there is always return of control at the end of each state scan which doesn't require any special consideration on the programmer's part.

The simplest of these has already been described in **Figure 2,** and in the subsequent description of the "universal real time solution." By successively applying the scan logic to each task in the thread, all of the tasks are given a share of the computing resource. Following the rules descirbed above, intermittent tasks are given as many scans as they need to complete their current activity, while continuous tasks are given a fixed number of scans for each complete cycle of the scheduler. In the transition logic context, this is called a ***sequential*** scheduler, since it addresses each task in sequence.

In order to know when an intermittent task should run, the scheduler must keep track of the events to which the tasks are attached. The most common such event is the end of a specified time period. Events are often also based on changes in external signals or on internally generated conditions.

As noted above, intermittent tasks often have a latency specification. Use of the sequential scheduler can lead to longer latencies than can be tolerated because all other tasks must be scanned in succession before the scheduler will return to a given task. A ***recursive*** scheduler can greatly reduce the latency, particularly if there are a large number of tasks. Instead of scanning tasks sequentially,

the scheduler itself is called alter every scan. When called, it examines the task list to see if any tasks that are higher priority than the currently executing task are ready to execute. If so, the highest priority task is started, while the task that had been running waits. The latency for the highest priority task is thus reduced to no more than the worst case scan time of any single task, rather than the sum of scan times for all of the tasks.

This is a recursion because the scheduler is “calling itself’ -- it is initially invoked to start task execution, while running the task it invokes itself again to check to see if higher priority taks need to be run. If any do, they are started, but the scheduler again invokes itself to continue performing priority checks. As the higher priority tasks complete, they return control to the scheduler thereby backing out of the recursion.

The recursive scheduler can achieve the resource shifting needed to help a high priority task meet its real time performance constraints, but it takes significantly more overhead than the sequential scheduler since it is run more often. The net effect is thus to improve the effectiveness of the high priority tasks at the expense of the total amount of computing resource that is available for “productive” use.

8.4 Preemptive Multitasking Modes

If the latency specifications still cannot be met, a preemptive solution will be needed. Such a situation could arise, for example, if the action functions of one or more low priority tasks required substantial amounts of computing time. This does not violate the “nonblocking” condition, which states only that computing time in any of the state functions must be predictable, not that it must be “short.” In other cases, it might be necessary to violate the nonblocking condition. Though certainly not desirable, violations could arise due to the need to use software not under the control of the real time programmer. Examples of these might be operator interface software or mathematical software with nondeterministic iterations.

Preemptive scheduling makes use of the computer’s interrupt facility, which can, in response to an external event, temporarily halt the execution of whatever code is currently running and start some other code running. Since the interrupt facility is part of the computer’s hardware, this preemption takes place on a much faster time scale than can be achieved by the recursive, cooperative scheduler and so, can meet much stricter latency requirements.

Preemption is used to establish a new computing thread. When an interrupt occurs, the current thread is suspended and a new thread is started. These threads share a common memory space, but are otherwise independent entities. At the simplest level, the interrupt mechanism itself is a scheduler. Each interrupt-scheduled thread would thus contain one task. However, interrupt-scheduled threads can contain more than one task. The tasks inside the interrupt-scheduled thread must be scheduled by a cooperative scheduler.

Interrupt-scheduled threads must only contain intermittent tasks. Because the hardware interrupt establishes a priority level ahead of any software-based priorities, if the tasks in an interrupt thread were to ask for “continuous” CPU resource, all other tasks would be locked out. Thus, all of the

continuous tasks must be left in the non-interrupt domain; that is, they get to run whenever no interrupts are active. Threads that are scheduled directly by interrupts must normally be quite short. Because the interrupt mechanism on most computers is constructed in such a way that all other interrupts of the same or lower priority are locked out until the present interrupt is finished, it becomes essential to keep this time as short as possible.

Interrupts can also be used as a mechanism for running schedulers. When this is done, the restriction that only short threads should be run can be relaxed. This is possible because, while the scheduler runs as interrupt-based code, it resets the interrupt mechanism before allowing any user-written code to run. This allows access for all interrupts while the user-written task is running.

Schedulers of this sort come in two flavors:

1) Those that operate through the function-call mechanism,
2) Those that manipulate the execution context directly.

The most obvious difference between these, particularly in the transition logic environment, is that for the first type of scheduler, the continuous tasks are operated using cooperative scheduling, while the second type can use time-sliced scheduling. The difference is that in cooperative scheduling, the continuous tasks are given access to CPU resource according to the number of transition logic scans they use whereas in time-sliced scheduling, the continuous tasks get CPU resource according to the amount of computing time they use. This can make a performance difference in those cases for which there are significant differences in execution time of specific state functions. Only time-sliced scheduling can handle a case where the nonblocking restriction is violated.

The scheduler that operates via the function-call mechanism is identical to the recursive scheduler described above. The only difference is that instead of being called from the transition logic scanner, it is called from an interrupt. It is thus the same code, but `run` as a reentrant rather than recursive scheduler. However, to avoid two different names for the same scheduler, it will be called a ***recursive-interrupt*** scheduler.

There are other differences between these scheduling mechanisms as well, particularly relating to how and when tasks can suspend themselves, but these differences are less important in the transition logic environment than they are in other computing environments.

8.5 Task Types

In recognition of the various scheduling methods outlined above, every task must have a designated task type. The task types are the user's indications of ***preferred*** task type. Depending on the scheduling environment that is actually used, the task may be scheduled using a simpler mechanism than was requested. The distinction between ***continuous*** and ***intermittent*** tasks, however, will be maintained in all implemented environments.

The types of tasks that are defined are:

- Hard timer interrupt
- Soft timer interrupt
- One-shot timer
- Digital interrupt
- Sample-time
- Logic event
- Continuous

The first four are all interrupt-scheduled, if an interrupt environment is defined. That means that they operate at interrupt priority and can only be preempted by other, higher priority, interrupts. The one-shot timer differs from the other timer-types because it does not automatically reset the timer and run again after it has run once. The timer must be explicitly reset each time. If no interrupt is present, they will be cooperatively multitasked, using either the sequential or recursive scheduler. As interrupt-scheduled tasks, they should be short, relinquishing the CPU quickly after activation.

The sample-time and logic-event types are generally of lower priority, and, given the appropriate execution environment, should be controlled by an interrupt-driven scheduler, of either of the types defined above. Because the tasks execute only after the interrupt system's priority has been reset, they are preemptable by any direct interrupt-driven tasks, and by the interrupt driven scheduler, which can determine at any time if higher priority tasks should be run.

The continuous tasks always operate at lowest priority. They can be either scan or time-slice scheduled.

8.6 Supported Environments

At present, environments of all of the types described above are supported. All of these run on PC-type of computers (X86 architecture, MS/PCDOS operating system, Microsoft C compilers). A version for QNX (a commercially available real time operating system) is currently being developed.

The simulation and calibrated-time modes are the least affected by the target environment. Although these are currently targeted for PCs, it would take little or no work to target them for other general purpose computing environments. Both sequential and recursive schedulers are implemented in the simulation environment. The calibrated-time environment is the same as the simulation environment, except that the simulation step size is chosen so that simulated time matches real time (at least on average).

The single-thread, real time environment also uses the same scheduling as the simulation mode, either sequential or recursive. It differs from calibrated-time, however, by using an external clock to measure time instead of a calibration. It thus gives accurate time at all times, instead of just on average. This mode is also easily ported to any target environment for which a fine enough time granularity is available.

The direct interrupt environment is supported through a locally written package called XIGNAL. This package sets up interrupt hardware and associates the desired interrupt function with the

hardware interrupt. In this mode, all timer and event related tasks are executed tasks are attached to the interrupt. Since there is only one clock available on a PC, all of the time-type tasks are connected to the same clock. Thus, regardless of priority, once a time-type task starts running, it will run to completion before any other timed task can run. There is a somewhat arbitrary decision to be made in the implementation of this mode. That is, whether the sample-time and event based tasks should be connected to the interrupt or left in the cooperative domain. If any of them have long execution times, they could interfere with the higher priority direct interrupt tasks. On the other hand, they have a better chance of meeting time constraints as interrupt-driven tasks.

Since in the direct interrupt mode no reentrance is allowed to the running tasks, it is probable that the interrupt facilities supplied with the compiler would work as well as XIGNAL, and perhaps more efficiently. This has not yet been tested.

Adding the recursive scheduler to the interrupt mode (which then makes it a reentrant scheduler) allows the separation of the sample-time and event tasks from the direct interrupt tasks. Because the recursive scheduler only requires the additional ability to reset the interrupt system's priority, it does not pose much more difficulty for porting than the direct interrupt mode itself. This mode, does, however, require that the interrupt package permit reentrant interrupts. Most interrupts associated with compilers do not permit reentrance so are not suitable for this mode.

Full context-switching scheduling is implemented with another locally written package, CLOTHO. In most control applications, the major operational distinction of this scheduler is that it provides for time-slice scheduling of the continuous tasks. This permits the use of ill-behaved tasks (i.e., blocking) or tasks which, while not actually blocking, have unacceptably long execution times.

The following table summarizes the supported environments and gives the abbreviations used in further references to them.

Name	***Description***
Seq-Sim	Sequential scheduler, simulation environment (no real time at all)
Ret-Sim	Recursive scheduler, simulation environment
Seq-Calib	Sequential scheduler, time by calibration of step size, single thread
Ret-Calib	Recursive scheduler, time by calibration of step size, single thread
Int	Direct interrupt scheduling for time and event tasks, number of threads depends on hardware configuration; single thread, sequential scheduler for continuous tasks

Name	*Description*
Ret-Int	Direct interrupt scheduling for timer and digital IO tasks, interrupt-driven recursive (reentrant) scheduling for sample time and event tasks. Each recursively scheduled task has a unique thread. Sequential scheduling for continuous tasks.
Slice	Direct interrupt and recursive-interrupt scheduling for time and event tasks, time-sliced scheduling for continuous tasks

9 Intertask Communication

Communication among tasks is inherent to control software. The design concerns in structuring the communication facilities are data integrity, communication efficiency, and software portability. These must be addressed for data that is explicitly defined by the user/programmer, and data that is integral to the task model, including task status information, state information, task parameters, etc.

9.1 Data Integrity

Even when all tasks reside in a single process, they can still be separated into asynchronously executing (lightweight) threads. Since these tasks share teh adress space, normal C language syntax such as function calls or external variables, could be used for data transfer across tasks, if any form of preemptive scheduling is used, there is a risk of data corruption. To maintain data integrity, restrictions of ***mutual exclusion*** must be observed. Mutual exclusion must be invoked whenever the information exchange involves separate threads to prevent the data exchange from being interrupted by a preemptive context switch, leading to the possibility of part of the data exchange being completed before the switch, and then, in the new context, changing that data, thereby leading to corrupted data when the data exchange completes after the preemption is complete. There are ways to prevent this, most commonly by disabling the interrupts during a data exchange, but the user must know when and how to apply them. Furthermore, these methods will fail any time tasks are divided into more than one process.

9.2 Communication Across Processes

One attractive way to meet the constraints of several high priority tasks is to use independent processors, one for each such task. This guarantees immediate attention to the highest priority task in each processor, and, rather than the resource shifting associated with real time scheduling, this adds additional resource to meet these requirements. The tradeoff in using multiple processors is that data exchange must go through some external medium to get from one task to another. This has consequences of both overhead to service the medium, and delay for the actual transfer of information. These costs must be balanced against the savings due to less scheduling overhead and the potential increase in total processing resource.

If an application is compiled and linked to run in an environment with independent processes (including the case of multiprocessors) then conventional programming methods for interchange of information among functions (through arguments, return values, global variables) will not work

because the independent processes do not share memory address spaces. Since a major goal of this methodology is to retain code portability, standard data exchange mechanisms are not suitable.

9.3 Communication Media

Common communication media include shared external memory (for multiple processors that share a bus, for example), networks of all sorts, and point-to-point connections, usually via serial channels, but possibly using parallel channels. Each of these media has different software interface protocols. Again, if portability is to be presemved, the details of these protocols must be hidden from the user. Selection of different mechanisms will have significantly different performance and cost, as well as different distance and environmental constraints. which must be factored into the final system design.

At present, support is provided for point-to-point serial connections and for Ethernet connections using TC/PIP protocols.

9.4 Communication Structures

As a result of these factors, the definition of intertask communication has been placed in the transition logic tool, at the same level as task definition. This is important for the portability to various means of inter-process communication and to assure maintenance of data integrity. The model chosen for inter-task communication is that of pseudo-shared memory. Variables set in one task are accessed in other tasks as if they were local variables. The variables involved are specified using two lists for each task -- one list for those variables that are set in the task and available for use by other tasks, the second list for variables set in other tasks and used in this task. Each variable must be set in one and only one task. This avoids conflicts in usage in which the same variable could be set in two places at nearly the same time, but one of the actions would override the other. Since this is a race condition, there is no way to predict what value the variable would have.

An important implementation factor with this method is that when tasks share a process, the communication overhead must be kept to a minimum. Tasks are grouped together specifically to minimize communication overhead, so this must be recognized when the inter-task communication structure is set up.

10 Example: Stepping Motor Control

An example of the application of these techniques is given by a stepping motor control program. It is a relatively simple example, but contains enough complexity and hierarchy to show the basics of the methodology.

10.1 Tasks

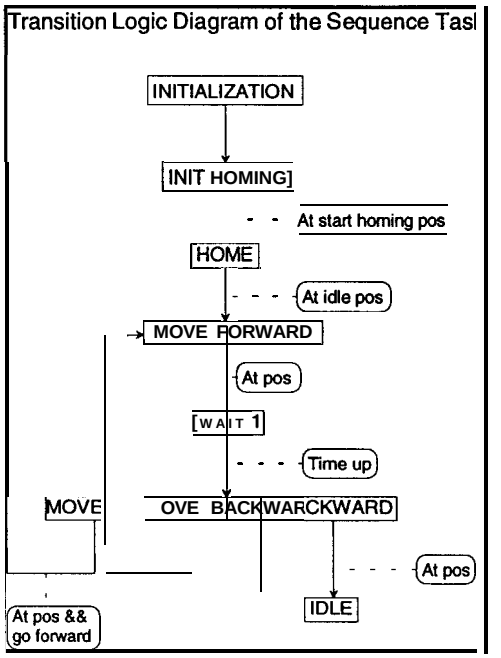

Figure 5. Transition Logic for the ***Sequence*** Task

The purpose of the control system is to make sequences of moves using a stepper motor. It is organized into three tasks:

1. Sequence control to specify when the moves should be made and to where.
2. Step generation to resolve differences between actual and desired motor position by generating steps at a specified rate.
3. A master task to control start-up and shutdown of the other tasks and to control the operator interface.

The sequence task, **Figure** 5, begins by performing whatever system initialization is necessary, then requests that the stepping motor find its home position. It then enters a loop in containing a forward move, a timed wait and a backward move. This loop is intended to be typical of the kinds of motion that might be commanded in manufacturing or assembly machinery. The actual sequence of events is easily modified by defining a new set of states for that section of the task. The sequence terminates when the ***go-forward*** flag is no longer on. Each of the states in this loop has transition conditions relevant to its activity: the motion states complete when the motor reaches its target position and the ***wait*** state completes when time is up.

The step generation task, **Figure 6,** operates by interpreting commands that are sent to it. The master task, **Figure** 7, controls the start-up of the other tasks, and, in this case, also implements the operator interface.

10.2 Documentation

The documentation information produced by the transition logic processor follows:

Project: STEPSEQ

Task: Master-opint; Initial State: Initialize
Type: Continuous, Priority: 0 Chnl: 0 Data value: 0 **Init** status: 2
State: Initialize [Initialize I/O, variables, etc.]
to: **StartTasks** if [Initialization done]
State: **StartTasks** [Start up sequence and position tasks]
to: Operate if [Tasks successfully started]
State: Operate **[Normal** system operation]
to: Shutdown if [Operator request for shutdown]
State: Shutdown [Home motor, turn off control]

Task: Sequence; Initial State: Initialize
Type: Continuous, Priority: 0 **Chnl:** 0 Data value: 0 **Init** status: 0
State: Initialize [Home motor]
to: **InitHoming** if [Start sequence]
State: **MoveForward** [Move to forward position]
to: Wait1 if [Done with forward move]
State: Wait1 [Wait specified time]
to: **MoveBackward** if [Done waiting]

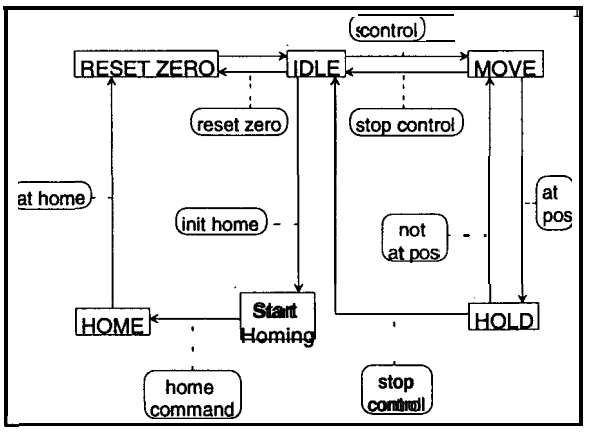

Figure 6. Transition Logic for the ***Position*** Task

State: MoveBackward [Move back to home1
to: Idle if [Completed N sequences]
to: MoveForward if [Do another sequence]
State: Idle [Waiting for next command]
State: InitHoming [First homing step]
to: Home if [Unconditional]
State: Home [Complete homing]
to: MoveForward if [Start sequencing]

Task: Position; Initial State: ResetZero
Type: Soft timer interrupt, Priority: 10 Chnl: 0 Data value: 50 Init status: 0
State: ResetZero [Establish a new 'zero' position]
to: Idle if [Done resetting]
State: Move [Move until motor is at setpoint]
to: Hold if [At desired position]
to: Idle if [Stop control]
State: Hold [Keep motor at setpoint]
to: Idle if [Stop control]
to: Move if [Change in desired position]
State: Idle [Wait for command]
to: ResetZero if [Set new zero]
to: Move if [Start control]
to: StartHoming if [Start homing command]
State: StartHoming [Prepare for homing]
to: Home if [Complete homing command]
State: Home [Complete homing]
to: ResetZero if [Done homing]

CONFIGURATION INFORMATION
Configuration #0: SingleProcess
Process - All-Tasks, Task List:
Master-opint
Sequence
Position

End of Documentation==========

In addition to giving a tabular description of the transition structure, it gives some task information that is not shown on the diagram. In particular, it gives the task type, data associated with the task, the initial state, and the status of the task when the program starts. For the stepping motor system, this information is:

Master: continuous task; initial state: Initialize; initial status: active

Sequence: continuous task; intial state: Initialize; initial status: inactive

Position: soft-timer interrupt task; initial state: ResetZero; initial status: inactive; priority: 10; sample time (data value): 50 ms

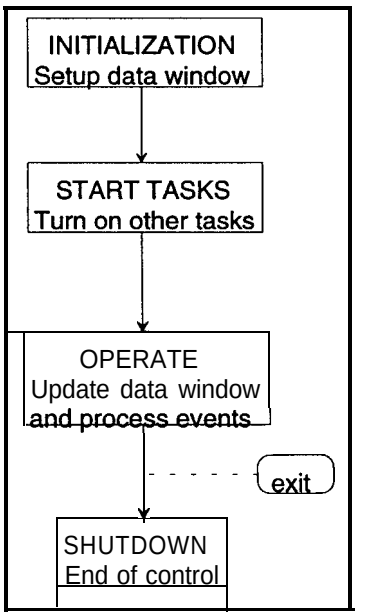

Figure 7. Transition Logic for the ***Master*** Task

The information at the end of the documentation, configuration information, is used to indicate which processes the tasks are assigned to. In this case, all of the tasks are assigned to a single process, which is the default configuration.

10.3 Sample of Code for "Position" Task, Move State

The complete code for the program is a combination of automatically generated code and user-written code. A selection of the code from the stepping motor control code follows. The sections that are user-written are identified by the surrounding comments indicating "Editable..." code. The state and function type (entry, action, etc.) is also identified. All other code has been automatically generated. The code as shown is set up for simulation -- the defined symbol STEP-SIM is used for this. Additional defined symbols can be added to adapt the code for operation of actual stepping motors.

```
void EMove_Position(void)
{
/*@tl_mark..52. Editable Entry func, state: Move */
/*@tl_mark..52. End editable Entry func, state: Move */
}
void AMove_Position(void)
{
/*@tl_mark..53. Editable Action func, state: Move */
step-desired = step-desired-next;     /* Update setpoint */
#ifdef STEP-SIM
if(step_actual c step_desired)step_actual++;
else if(step_actual > step_desired)step_actual--;
#endif
SuspendTask(THIS_TASK);
/*@tl_mark..53. End editable Action func, state: Move */
}
int T1Move_Position(void)
{
/*@tl_mark..54. Editable Test func, state: Move -to- Hold */
if(step_actual == step-desired)retum( 1);
else return(O);
/*@tl_mark..54. End editable Test func, state: Move -to- Hold */
}
void X1Move_Position(void)
{
/*@tl_mark..55. Editable Exit func, state: Move -to- Hold */
/*@tl_mark..55. End editable Exit func, state: Move -to- Hold */
}
int T2Move_Position(void)

/*@tl_mark..56. Editable Test func, state: Move -to- Idle */
if(step_command == STEPCOM_STOP_CONTROL)return( 1);
else return(O);
/*@tl_mark..56. End editable Test func, state: Move -to- Idle */
}
void X2Move_Position(void)
{
/*@tl_mark..57. Editable Exit func, state: Move -to- Idle */
/*@tl_mark..57. End editable Exit func, state: Move -to- Idle */
```

}

10.4 Operating Information

The operating record of transitions in all tasks is automatically generated as an audit trail. The audit trail is set to save the most recent transitions (up to the limit of available memory), so if anything unusual happens, the events leading up to the problem are automatically saved. The audit trail shown below is for a typical run of the stepping motor control program. The first column gives the time in milliseconds, next the task name is given, then the transition that has just taken place. The audit trail show the start-up activites, then the homing, and finally the sequence of move-forward, wait, move-backward stepping motor motions that are the control goal.

0.000000e+000 Master-opint Initialize-to-StartTasks
1.000000e+000 Master-opint StartTasks-to-Operate
2.000000e+000 Sequence Initialize-to-InitHoming
5.1 00000e+001 Position ResetZero-to-Idle
5.200000e+001 Position Idle-to-StartHoming
5.400000e+001 Sequence InitHoming-to-Home
1 .010000e+002 Position StartHoming-to-Home
1.020000e+002 Position Home-to-ResetZero
1.030000e+002 Position ResetZero-to-Idle
1.060000e+002 Sequence Home-to-MoveForward
1.5 10000e+002 Position Idle-to-Move
1.001510e+005 Position Move-to-Hold
1.001530e+005 Sequence MoveForward-to-Wait1
1.301550e+005 Sequence Waitl-to-MoveBackward
1.302000e+005 Position Hold-to-Move
2.302000e+005 Position Move-to-Hold
2.302020e+005 Sequence MoveBackward-to-MoveForward
2.302510e+005 Position Hold-to-Move
3.302510e+005 Position Move-to-Hold
3.302530e+005 Sequence MoveForward-to-Wait1
3.602550e+005 Sequence Waitl-to-MoveBackward
3.603000e+005 Position Hold-to-Move

11 Real Time Performance

Engineering success of a control project depends on the ability of the control system to meet all of the relevant constraints, including economic constraints, and to optimize whatever facet(s) of the performance space is most important for the particular project. A major asset of the methodology proposed here is that the focus on portability means that performance related decisions can be held in abeyance until the basic control system is completely functional and can be evaluated in its entirety in a variety of candidate configurations. The primary definition of real time performance is:

The right result at the right time

This is defined on a task-by-task basis,

For intermittent tasks:

I. Never miss a scheduled execution slot
2. Execute within specified tolerance of defined slot

For continuous tasks:

Average progress over an extended time must meet specification

Each task has its own specifications and tolerances. In general, tolerances are looser for lower priority tasks.

11.1 Sample Problem for Performance Evaluation

A very simple job has been defined to demonstrate performance evaluation. All of the tasks do the same thing -- count (except for master task). The results are thus easily assessed!

Although each task has the same structure, the computational load and real time characteristics are varied by using different task types, and varying the numerical values associated with each task.

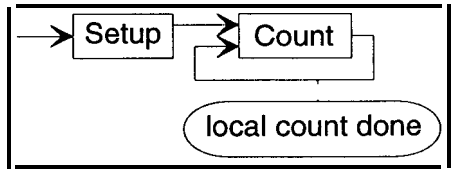

Figure 8. Counting Task

Each task has three variables:

local count - set in entry function, counts down to zero for exit
action count - iteration limit for a counting loop in the action function
task count - starts at zero, increments by one on each scan

The transition shown is a self-transition, however, because it is an explicit transition it causes the ***entry*** function to run. The balance between the sizes of the local count and the action count affects the "visibility" of the task to a cooperative scheduler. As described above, cooperative schedulers run only between transition logic scans. Thus, increasing the computing load by increasing the action count will be invisible to a cooperative scheduler, while increasing it by increasing the local count can be visible. The affect of local count will depend on the type of task and the type of scheduler. For continuous tasks, which in cooperative mode get a fixed number of scans each time they are activated, the local count will ahve no affect on performance. On the other hand, intermittent tasks run until they self-suspend on each activation, so the local count will affect the computing time used on each activation.

Only the local count is reset in the entry function, while the task count is initialized in the Setup state. Thus the overall task performance is measured by the task count, which continues to count up as long as the task remains active. For intermittent tasks, it indicates whether any scheduled execution slots have been missed. For continuous tasks, it gives the average progress of the task.

11.2 Counting Task Functions

The state functions from Task1 are shown below. All of the others are the same, with only numeric differences.

Task Code, common to the whole task:

```
/* Task code area -- statics, functions, etc. for this task */
static int task_count,local_count;
int  getcl(void)
        {
        return(task_count);
```

State "Setup", ***entry function (all others are empty):***

```
/* @tl_mark..4. Editable Entry func, state: Setup */
task-count = 0;
```

State "Count", ***entry, action, test,*** and ***exit*** functions:

```
void ECount_Task1(void)
{
/* @tl_mark..8. Editable Entry func, state: Count */
local-count = 4; /* Number of scans before suspension *I

/* @tl mark..8. End editable Entry func, state: Count */
}
void ACount_Task1(void)
{
int i,action_count = 10;

/* @tl_mark..9. Editable Action func, state: Count */
local-count--;
for(i = 0; i < action-count; i++)task_count++;
/* @tl_mark..9. End editable Action func, state: Count */
}
int T1Count_Task1(void)
{
/* W-mark.. 10. Editable Test func, state: Count -to- Count */
if(local_count <= 0)return( 1);
else  return(O);

/* @tl_mark..10. End editable Test func, state: Count -to- Count */
}
void  XlCountTaskl(void)
{
/* @tl_mark..11. Editable Exit func, state: Count -to- Count */
SuspendTask(THIS_TASK);
  /* Task will be suspended when scan is done */

/* @tl_mark.. 11. End editable Exit func, state: Count -to- Count */
}
```

11.3 Model of a Simple Control Job

A sample of what could be the task structure for a simple control job is shown in the table, below. It has time-based intermittent tasks plus continuous tasks --

2 soft timer interrupt tasks
1 sample time task
3 continuous tasks (one of which is the master task)

Task	*Type*	*Priority*	*Typical Sample Time*
Master	continuous	--	--
Task1	soft timer interrupt	10	2 ms
Task2	soft timer interrupt	8	4 ms
Task3	sample time	5	10 ms
Task 4	continuous	--	--
Task 5	continuous	--	--

11.4 Simulation Mode

Simulation mode will be examined in some detail, before summary information is presented for several real time environments. Simulated operation is easy to control, and thus generates information more easily than other modes. It is also a critical, and often neglected step in the development of real time software. Because time is completely artificial in simulation mode, all of the debugging mechanisms of conventional programming can be used, inserted output statements, rerunning to reproduce an error, etc. When this process is complete, the basic logic of the program has been checked out and verified, thereby greatly reducing the debugging effort required in actual real time operation, where debugging is much more difficult because of asynchronous operation of program and physical components. To implement simulation mode, a computational model is required of the control object. There is a significant cost to creating this model, but, overall, it is very effective towards keeping total job cost down (and predictable), and minimizing elapsed time to complete the job.

Time, in the simulated domain, is incremented by a specified at after each state scan. This time increment represents how much simulated (computing) time each transition scan takes. Smaller time increments simulate faster computers.

A sample run is shown below, for Δt = 0.5 ms and desired run time = 100 ms --

```
C:\PROG>perfevl
Transition Logic Runtime System

Enter tick time (ms): 0.5
time: 116.000000, task_count1 ... 5: 52 91 44 13 13
```

The first three tasks are timer tasks -- they must run in all scheduled time slots to meet their basic specification. In this case, the correct values are: 200, 175, 40, whereas the actual results show 52, 91, 44. Therefore 0.5 ms is too large a step size (simulated tick time). Thus, the simulated computer is "too slow." The average computing time taken for each state scan, 0.5ms is too much to allow the job to meet its timing specifications. Note that the final time is 116 ms instead of 100 ms as specified. This is because the master task is a continuous task, and thus of low priority. That was the closest time to 100 that it got to run.

11.5 Audit Trace File

The transition trace produces a record of the most recent transitions for use as an audit trail. For this case, the a section of the audit trace is shown below; it has been annotated with "* 1" for a relevant set of lines relating to task 1 and "*2" for lines relating to task 2. Task 1 has a specified sample time of 2 ms and task 2 has a specified sample time of 4 ms.

```
 Time            Task  Transition
0.000000e+000  Master  StartTasks-to-Run
5.000000e-001 Task4 Setup-to-Count
1  .OOOOOOe+OOO  Task5  Setup-to-Count
3.000000e+000 Task1 Setup-to-Count * 1
5.000000e+000 Task1 Count-to-Count *1
5.500000e+000 Task2 Setup-to-Count *2
9.000000e+000 Task2 Count-to-Count *2
1.250000e+001 Task1 Count-to-Count *1
1.600000e+001 Task2 Count-to-Count *2
1.650000e+001 Task3 Setup-to-Count
1.850000e+001 Task3 Count-to-Count
1.950000e+001 Task4 Count-to-Count
```

The three marked occurrence of task 1 are at times of 3, 5, and 12.5 ms. The first two occurrences are separated by the proper interval, but the last is much too late, confirming the information from the total count output.

Making at smaller simulates a faster computer. Several runs at smaller sample times are given below:

```
At =
0.2     time: 105.599533, task_count1 ... 5: 168 182 40 44 44
0.1     time: 102.499008, task-count1 ... 5: 204 175 40 200 200
0.05    time: 101.001564, task-count1 ... 5: 200 175 40 533 533
0.02    time: 100.395164, task_count1 ... 5: 200 175 40 1533 1533
```

While 0.2 ms is also too large, 0.1 gives satisfactory results for the intermittent tasks (1, 2, and 3). Once the basic real time constraint of the intermittent tasks has been met, the activity of the continuous tasks can be examined. As noted above, as the sample time gets smaller, the total activity of the intermittent tasks remains constant, but the activity of the continuous tasks (4 and 5) increases from 200 total counts to 1533. This shows that more of the computational resource is devoted to the lower priority continuous tasks as less is needed for the intermittent tasks.

For a sample time of 0.05 ms, here is a section of the audit trail:

```
6.864959e+001 Task2 Count-to-Count
6.904961e+001 Task4 Count-to-Count
6.949964e+001 Task4 Count-to-Count
6.994967e+001 Task4 Count-to-Count
6.999967e+001 Task5 Count-to-Count
7.019968e+001 Task1 Count-to-Count
7.039970e+001 Task3 Count-to-Count
7.079972e+001 Task4 Count-to-Count
7.124975e+001 Task4 Count-to-Count
7.169978e+001 Task4 Count-to-Count
7.224981e+001 Task1 Count-to-Count
7.259983e+001 Task2 Count-to-Count
7.269984e+001 Task4 Count-to-Count
...
```

These show proper execution of the intermittent tasks, Task1 -- 70.2ms then 72.2ms (2 ms sample time) and Task2 -- 68.6ms then 72.6ms (4 ms sample time).

Note that tasks 4 & 5 run at time slots not used by the time-based tasks. Task 5 does transitions less frequently because its local count is higher than that of task 4. However, the total task count is the same for tasks 4 and 5 because continuous tasks get the same number of scans, regardless of whether transitions take place or not. Time-based tasks, on the other hand, get as many scans as they need on each invocation.

11.6 Counting: Performance in Several Environments

The table below shows a typical performance experiment. In this case, the action count is changed in the continuous tasks, and the overall system performance is measured.

The schedulers used are those described in Section 8.5. The first four are all single thread implementations, while the last two are multithread. The numbers on the left give the local count/action count used for each of the trials (the local count is not being varied in this experiment). Varying the action count simulates that kind of situation that arises when a substantial computing function that is not easily broken into states, such as an FFT, is done within a task.

The base case has a minimal computing load in all tasks. The total task counts are scaled to 1.0, where 1.0 represents the maximum count achieved for that task. In the first (base) case, the intermittent tasks all meet their specifications (all results are 1.0). The task counts for tasks 4 and 5, then, show the relative scheduling efficiency; most of the computing time goes to running the scheduler because the computing within tasks is so minimal. The sample times for the calibrated time schedulers (Seq-Calib and Ret-Calib) are the step sizes needed to match real time in an average sense. All other step sizes are the largest that will work. In general, the recursive schedulers are less efficient, which is reasonable since the scheduler gets run between every state scan.

As the action counts of the continuous tasks are increased, the single thread (cooperative) schedulers show the first failures. The simpler schedulers (Seq-X) fail first because they have the longest

latency, that is, the longest time between finishing a task and the next time the task is checked. The recursive schedulers (Ret-X) lower the latency, since they check every task after each transition scan -- they do better than the sequential schedulers in that they can tolerate higher action counts in the continuous tasks before they fail. On the other hand, they are less efficient than the sequential schedulers as can be seen from the scaled task counts for the continuous tasks (4 and 5) for cases where all of the intermittent tasks meet their specifications. For the base case, for example, the real time recursive scheduler (Ret-RT) has an index of 0.216 while the real time sequential scheduler (Seq-RT) has an index of 0.722.

No failures for either of the multithread schedulers (Int and Rec Int) are observed for this experiment. That is because the intermittent tasks are operated from the computer's interrupt facility, and thus preempt the continuous tasks at the hardware level. Therefore, regardless of the action-function load (or any other load) in the continuous tasks, they cannot interfere with any of the timed tasks.

		ΔT (ms)	Task Counts (Scaled)				
			Task 1	Task2	Task 3	Task 4	Task 5
4/1 7/1 4/1 3/1 12/1	Seq Calib	0.094	1.	1.	1.	0.257	0.025
	Rec Calib	0.154	1.	1.	1.	0.065	0.065
	Seq RT	2.0	1.	1.	1.	0.722	0.722
	Rec RT	2.0	1.	1.	1.	0.216	0.216
	Int	2.0	1.	1.	1.	1.	1.
	Rec Int	2.0	1.	1.	1.	0.151	0.151
4/1 7/1 4/1 3/800 12/1000	Seq Calib	0.174	0.92	1.	1.	0.599	0.599
	Rec Calib	0.180	1.	1.	1.	0.210	0.210
	Seq RT	2.0	1.	1.	1.	0.958	0.958
	Rec RT	2.0	1.	1.	1.	0.659	0.659
	Int	2.0	1.	1.	1.	1.	1.
	Rec Int	2.0	1.	1.	1.	0.485	0.485
4/1 7/1 4/1 3/1500 12/1600	Seq Calib	0.210	0.67	1.	1.	0.730	0.730
	Rec Calib	0.185	1.	1.	1.	0.248	0.248
	Seq RT	2.0	0.95	1.	1.	1.	1.
	Rec RT	2.0	1.	1.	1.	0.730	0.730
	Int	2.0	1.	1.	1.	1.	1.
	Rec Int	2.0	1	1	1	0.562	0.562

		AT (ms)	Task Counts (Scaled)				
			Task 1	Task2	Task 3	Task 4	Task 5
4/1 7/1 4/1 3/3200 12/3400	Seq Calib	0.300	0.41	0.79	1.	0.813	0.813
	Rec Calib	0.192	1.	1.	0.83	0.236	0.236
	Seq RT	2.0	0.51	1.	1.	1.	1.
	Rec RT	2.0	1.	1.	1.	0.732	0.732
	Int	2.0	1.	1.	1.	0.927	.927
	Rec Int	2.0	1.	1.	1.	0.545	0.545

12 References

Auslander, D. M., M. Lemkin, A-C Huang "Control of Complex Mechanical Systems," Proceedings of 1993 IFAC Congress, Sydney, Australia, 1993(a).

Auslander, D. M., "Unified Real Time Task Notation for Mechanical System Control," Proceedings of the ASME Winter Annual Meeting, New Orleans, LA, 1993(b).

Bastiaens, K. , J.M. Van Campenhout, "A Visual Real-Time Programming Language," ***Control Eng. Practice,*** Vol 1, No. 1, pp 59-63, Feb., 1993.

Benveniste, A, P. Le Guemic, "Hybrid Dynamical Systems Theory and the Signal Language," ***IEEE Transactions on Automatic Control,*** Vol. 35, No. 5, May 1990.

Domfeld, D.A., D.M. Auslander, P. Sagues, "Programming and Optimization of Multi-Microprocessor Controlled Manufacturing Processes," ***Mechanical Engineering,*** Vol 102, No. 13, 34-41, 1980.

Kohavi, Z., ***Switching and Finite Automata Theory,*** chap. 9, McGraw-Hill, New York, 1970.

Le Guemic, P., A. Benveniste, P. Boutnai, T. Gautier, "SIGNAL - A Data Flow-Oriented Language for Signal Processing," ***IEEE Transactions on Acoustics, Speech, and Signal Processing,*** Vol ASSP-34, No. 2, April 1986.

Leveson, N.G., C.S. Turner, "An Investigation of the Therac-25 Accidents," ***IEEE Computer,*** July, 1993.

Sandige, R.S., ***Modern Digital Design,*** p. 452, McGraw-Hill, New York, 1990.

13 Part II: Velocity Measurement From Widely Spaced Encoder Pulses

The need for velocity measurements oftentimes arises when feedback control of a system is desired. Velocity feedback can result in lower overshoot, better damping, and faster rise times when implemented in a system previously having only position feedback. In some instances, such as an inverted pendulum, systems may be even impossible to stabilize without velocity feedback. In many of today's systems a analog velocity measurement is not available because of cost or practicality considerations. In this case it is necessary to infer the system's velocity through whatever means the control engineer has at their disposal. Because of their wide dynamic range (limited only by computing considerations) and the reliability of using a digital signal, often the only information available is a discretized position measurement as supplied by an incremental encoder or equivalent device. This report reviews current technology in velocity estimation of a continuous time system using only digital position data. Next the problem of velocity estimation of slow and reversing systems is described and new methods to deal with these problems are proposed. Extensions of this new method are to be explored and refined.

Encoder signals are "sparse" when the pulse period becomes greater than the sampling period of the controller. In any system that ever stops or reverses direction, there must always be such an operating range. Because the upper limit of encoder density is limited by processing power, even systems that normally run a slowly varying velocity and rarely or never stop may have a valid velocity range for which the encoder signals become sparse. A major problem, which will be addressed here, is that whenever there is a velocity reversal, the actual point of reversal can never be known and can introduce substantial errors into the velocity estimate.

These problems are relevant to vehicle control problems because of the prevalence of digital feedback. The system proposed for lateral control based on magnetic markers implanted in the road is such a system, and will, in general, have a pulse period that is long with respect to controller sampling times. Other vehicle components in the drive train, including engine speed measurement, anti-lock braking, and traction control also will be likely to use pulsatile signals.

13.1 Problem Formulation

The problem of velocity estimation from discrete position measurements can be formulated as follows: ***Given an evenly spaced real sequence $\{x_i\}$, consider a nondecreasing sequence $\{t_k\}$ of positive real numbers such that $x_i=f(t_i)$ for some smooth real valued function $f(.)$. Find the first derivative of some C^1 function $g:\mathbb{R}\to\mathbb{R}$ with $f(t_k)=g(t_k)$ for all $k\in\mathbb{N}$ and $\sup|g'(t)|<\infty$ for all $t\in\mathbb{R}_+$*** (See figure 1). In this problem statement, the function $f:\mathbb{R}\to\mathbb{R}$ is the position trajectory of the plant and it is unknown in general. The only information available is the sequence $\{x_i\}$ which represent the encoder pulses with $|x_{i+1}-x_i|=$constant. The sequence $\{t_k\}$ of times at which the encoder pulses come out is not necessarily known. The problem is to find the first derivative of f(t) from the sequence $\{x_i\}$. Instead of estimating the velocity from the sequence $\{x_i\}$ on which the time derivative is not defined, some differentiable function $g:\mathbb{R}\to\mathbb{R}$ is found to approximate the function f(.), and the time derivative can be evaluated from g. In solving this problem, a Nyquist theorem like assumption should be observed:

- ***The maximum frequency off(.) should be at most*** $2\inf_i\{\frac{1}{t_{i+1}-t_i}\}$

If the above condition is not satisfied then it becomes impossible to reconstruct the true velocity using any velocity estimation algorithm due to a lack of information about the trajectory.

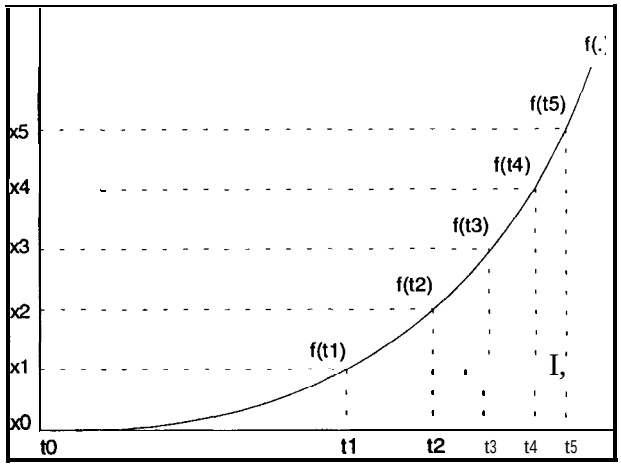

Figure 1

14 Current Technology

Current methods of velocity estimation belong to one of two categories - those which use a system model to estimate velocity and those which do not use a model of the dynamic system. We will first look at estimators which don't employ a system model. A more detailed analysis and results of simulations for the following estimators may be found in [1,2]. References [3,4] provide additional analysis and performance characteristics of these estimators.

14.1 Lines per Period Estimator

The Lines per Period (LPP) estimator is the simplest possible velocity estimator. Samples of the output from the encoder are taken periodically every T_s seconds. Since the sample time remains constant this is also known as a fixed time estimator. The estimate of the velocity is given by (1).

$$\hat{V} = \frac{x(t_1)-x(t_0)}{T_s}, \qquad T_s \; a \; t, - t_0 \tag{1}$$

Note that a problem with this type of velocity estimation is that the actual position the encoder has moved during the sample period is not known exactly but is bounded by +/- 1 encoder count. Because of these errors velocity estimates made with this method are extremely poor at slow speeds, i.e. when the number of counts is comparable to the loss in precision due to sampling. At higher speeds performance improves since the change in position becomes large and roundoff errors introduced by sampling become insignificant.

14.2 Reciprocal Time Estimator

The Reciprocal Time (RT) estimator is very similar to the LPP estimator but varies in that the RT estimator is a ***fixed position*** estimator. Fixed position estimators use time stamped data on the arrival of each encoder pulse. Time stamping, which may be implemented in either software or hardware,

is accomplished by saving the current value of a running clock the instant each pulse occurs. The estimate of the velocity given by the RT estimator which calculates the velocity based on the current and last pulse is:

$$\hat{V} = \frac{x(t_1)-x(t_0)}{T_{s1}-T_{s0}} = \frac{1\ Count}{T_{s1}-T_{s0}} \tag{2}$$

where T_{s1} and T_{s0} are the times at which the current and previous pulses occurred at respectively. This type of estimator works well at slower speeds since the time between pulses may be measured quite precisely. At higher speeds precision is lost since the time between pulses becomes comparable to the precision of the clock used for time stamping. Since encoders and other sensors have systematic as well as random errors in position due to inaccuracies in the quadrature, the numerator in (2) is not truly equal to one. This gives rise to periodic and random errors in the velocity estimate.

14.3 Taylor Series Expansion

If the system is assumed inertial, which makes no assumptions about the structure of the system, the velocity must be a smooth function of time. A Taylor Series Expansion (TSE) of the velocity as a function of time may then be formed.

$$\hat{V}_k = \sum_{i=0}^{\infty} \frac{1}{i!} \hat{V}_{\beta}^{(i)} (t_k - t_{\beta})^i \tag{3}$$

If, furthermore, the following assumptions are made:

$$\hat{V}_{\beta j}^{(i)} \approx \frac{\hat{V}_{\beta j}^{(i-1)} - \hat{V}_{\beta j-1}^{(i-1)}}{T_j}, \qquad \hat{V}_{\beta j}^{(0)} \approx \frac{x_j - x_{j-1}}{T_j}$$
$$(t_k - t_{\beta}) \approx \frac{T_k}{2} \tag{4}$$

we may approximate the velocity at the kth sampling instant by combining (3) & (4) and truncating high order terms to yield the following equation for a Nth order velocity estimator:

$$\hat{V}_k = \sum_{i=0}^{N} \frac{1}{i!} \hat{V}_{\beta k}^{(i)} \left(\frac{T_k}{2}\right)^i \tag{5}$$

This algorithm may be implemented as either a fixed time or fixed position algorithm, fixed position being better for slower speeds and fixed time superior for slower speeds.

14.4 Backward Difference Expansion

The Backward Difference Expansion (BDE) estimator, like the TSE estimator, may be implemented as either a fixed time or a fixed position estimator. For fixed position data, velocity is estimated by expressing time as a function of position (vice versa for fixed time) in a Taylor series expansion around your current value of time from your latest time stamp (or latest position in fixed time data). Explicitly, for fixed position data:

$$t_{k-m} = t_k + (-m)\frac{dt_k}{dx} + \frac{(-m)^2}{2!}\frac{d^2t_k}{dx^2} + \cdots \tag{6}$$

In order to implement a mth order BDE estimator, the mth order Taylor series approximation for t_{k-1} through t_{k-m} must first be calculated. Next the solution dt_k/dx which solves these m simultaneous algebraic equations must be found. The velocity estimate is the reciprocal of this quantity. For example a second order estimator gives us:

$$\begin{aligned} t_{k-1} &= t_k + (-1)\frac{dt_k}{dx} + \frac{(-1)^2}{2!}\frac{d^2t_k}{dx^2} \\ t_{k-2} &= t_k + (-2)\frac{dt_k}{dx} + \frac{(-2)^2}{2!}\frac{d^2t_k}{dx^2} \end{aligned} \tag{7}$$

Solving these we find:

$$\frac{dt_k}{dx} = \frac{3}{2}t_k - 2t_{k-1} + \frac{1}{2}t_{k-2} \quad \rightarrow \quad \hat{V} = \frac{1 \text{ count}}{\frac{3}{2}t_k - 2t_{k-1} + \frac{1}{2}t_{k-2}} \tag{8}$$

14.5 Least Squares Estimator

Perhaps the most promising of all the estimators found in current technology is the Least Squares (LS) velocity estimator reviewed in [1]. The LS estimator assumes that the time at which each encoder pulse occurs may be represented by a polynomial function of position. By taking the derivative of a polynomial fitted to time stamped position data an estimate of the velocity is obtained. The general idea is to look at your last several data points when fitting the polynomial, moving the data used for fitting forward as new data arrives. There are several issues which need to be considered when choosing the order of fit and number of data points used to fit the polynomial, including encoder rate, quadrature error, and complexity of trajectory. The Nth order LS estimator using the past M data points assumes there is a function :

$$t_k = c_0 + c_1 x_k + c_2 x_k^2 + \cdots + c_N x_k^N \tag{9}$$

$$t = Ac, \qquad t\ a \begin{vmatrix} t(x_{k-m}) \\ t(x_{k-m+1}) \\ \\ t(x_k) \end{vmatrix}, \qquad A \triangleq \begin{bmatrix} x_{k-m}^0 & x_{k-m}^1 & \cdots & x_{k-m}^N \\ x_{k-m+1}^0 & x_{k-m+1}^1 & \cdots & x_{k-m+1}^N \\ x_k^0 & x_k^1 & \cdot & x_k^N \end{bmatrix} \qquad c \triangleq \begin{vmatrix} c_0 \\ c_1 \\ \\ c_n \end{vmatrix} \tag{10}$$

Equation (9), for the past M data points, may be more conveniently expressed in matrix form as shown by equation (10). Since M > N for any estimator of interest the system is overdetermined and the least squares estimates of the coefficients are given by (11). The time elapsing between periods is now estimated at the current pulse by (12) from which the velocity is determined by taking the reciprocal of this quantity. Note that this particular implementation is a fixed position

$$\hat{c}=(A\ 'A)-'A\ 't \tag{11}$$

estimator so the matrix A is constant and need not be recalculated. In addition this method is only applicable for situations in which the velocity does not change sign, otherwise $t(x_k)$ is no longer a single valued function and the analysis no longer holds.

$$\frac{d\hat{t}_k}{dx} = \hat{c}_1 + 2\hat{c}_2 x_k + \ldots + N\hat{c}_N x_k^{N-1} \tag{12}$$

A distinct advantage of the LS estimator is if data points are chosen to include an integral number of periods for fitting (i.e. 4 pulses in a quadrature based system) systematic errors caused by periodic fluctuations are filtered out. The amount of filtering is determined by how overdetermined the matrix equation (10) is.

14.6 Observer Based Estimator

An observer based estimator for longitudinal velocity estimation has been designed and simulated using a discrete time approximation to a continuous time model of an automobile in [5]. The estimator attempts to find the forward velocity of an automobile using magnetic sensors which detect passage over magnets embedded in the roadway. The magnetic sensor is sampled regularly for detection of a new count. The discrete time model and observer are updated every time a pulse is encountered from the sensor. Position and a simple LPP velocity estimate are used to provide the innovations necessary for updating the observer. The observer effectively acts as a low pass filter for the errors introduced by sampling. A better observer based algorithm is proposed and discussed under the next section.

15 New Research Areas

The above algorithms looked at several different ways of estimating velocity given discrete position and time data. There are some situations, however, for which the above algorithms prove to be inadequate and do not appear to be addressed in the literature. Systems which change direction, move slowly, or require estimates of velocity more frequently than new position data becomes available are among these areas. A preliminary analysis and results of simulations and actual tests are listed below for each of these problems which need to be addressed.

15.1 Slowly Moving Systems

In systems which are moving very slowly encoder counts occur infrequently. Because of this the velocity between counts may fluctuate without the "knowledge" of the computer implementing the control algorithm. If one of the current technology algorithms for estimating velocity is implemented, a new velocity is calculated only when a new encoder pulse arrives. This estimate of velocity is then held until the next pulse. Note that in many cases the true velocity of the system does not remain constant between samples.

Suppose the true velocity of the system fluctuates in a manner such that it monotonically increases. This implies a pulse will be generated before predicted from the previous velocity estimate. This allows us to update the estimated velocity sooner than expected, however we have no information allowing us to adjust the velocity estimate in between samples to compensate for the acceleration.

If on the other hand the true velocity is monotonically decreasing, then the velocity at the end of the period will be less than the velocity at the beginning of the period. Note that since a new pulse will occur only when new data has arrived the old velocity estimate will not be updated until an even longer period of time than expected. This will wreak havoc in a controller attempting to bring the system to rest since the velocity estimate is high. One way to reduce the problem is shown in Figure 2. We know that if the system continues at the current estimated rate we expect a pulse at time t2, shown by the path (a). If we have reached time t2 and still haven't received a pulse it is necessarily true the current velocity is less than the previous velocity. An upper bound on this velocity is given by (13), shown by path (c). As time passes the upper bound and hence best estimate of velocity decreases according to (13). Path (b) represents a possible trajectory which has the above velocity characteristics.

$$\hat{V} = \frac{1\ Count}{t - t_1}, \qquad t > t_2 \qquad (13)$$

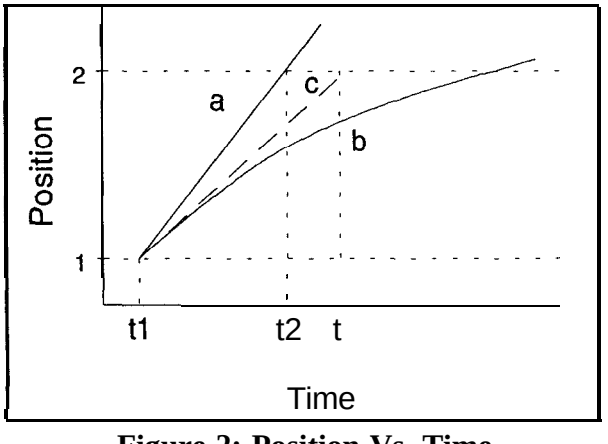

Figure 2: Position Vs. Time

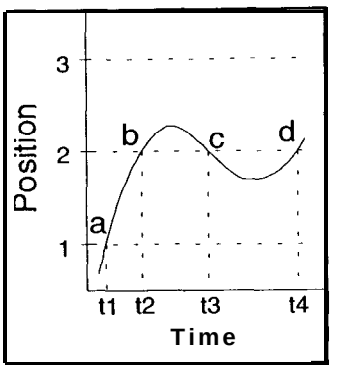

Figure 3: Degenerate Trajectory

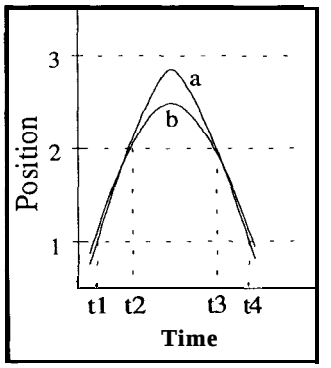

Figure 4: Sensitivity of fit to previous measurements

15.2 Transition Logic based Switching Algorithm

Algorithms in the current technology section all assume a velocity whose sign does not change. In many cases, such as position regulation or tracking, this is an invalid assumption. Since the velocity estimation technique which is most appropriate to use depends on the information available as well as the current operating conditions it is necessary to incorporate a systematic method of choosing the proper estimation algorithm. Figure 3 shows the main problem encountered in velocity estimation under velocity reversal. The position and direction of velocity are supplied by the measuring device. While going from (a) to (b) velocity may be estimated via any method desired. However in going from (b) to (c) the only new information gained is the knowledge that the velocity between (b) and (c) must have become zero for at least one instant in time, since velocity is a continuous function of time. In this case the only safe estimate of the velocity at time t3 is zero. Figure 4 clarifies why this is so. Both curves (a) and (b) in figure 4 have identical time and position data available for measurement at tl through t3. Although it is true we could modify the LS method for use in this situation the peak of the curve will be very sensitive to previous measurements. If the system is chattering, singularities in velocity estimates will occur as the time between pulses approaches zero. This results in very poor and inconsistent estimates of velocity.

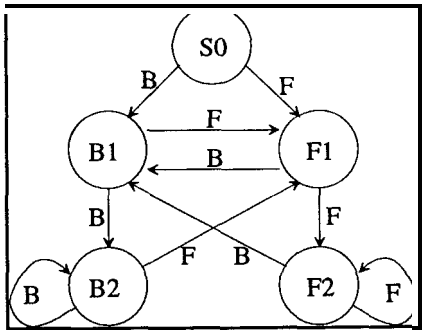

Figure 5: Finite State Machine

A logical solution to the problem of selecting a proper velocity estimate / estimation algorithm is to use a finite state machine to describe the system. One finite state machine which has been implemented successfully in a test system is shown in figure 5. This finite state machine runs inside a hardware interrupt service routine (ISR) triggered by the encoder pulse. The hardware interrupt is

also responsible for time stamping the arrival of new encoder pulses. Actual velocity estimation is performed in a timer ISR which determines the current state of the transition logic diagram and applies rules to generate velocity from the table of time stamped encoder pulses. One set of rules which has been successfully implemented is listed in table 1. This set of rules is based on the least squares method of velocity estimation.

Current State	Velocity Estimation Algorithm
s o	Velocity = 0
F1	Velocity = 0
F2	Velocity = LS Estimate
B1	Velocity = 0
B2	Velocity = LS Estimate

Table 1: Rules used in Velocity Estimation of reversing system

The above set of rules provides good estimates of velocity in the following manner. If we are in state SO, which would correspond to the initial state when the system is first started we have no idea where the true position lies between the encoder pulses (Fig. 6 position (a)) . Therefore best initial value of velocity is zero. If we move one pulse forward the state machine moves to state Fl and a valid timestamp of the position is collected (Fig. 6 position (b)). There is still no way of accurately estimating velocity since the original position is not known exactly. If one more step is taken in the forward direction we move forward to F2 and an estimate of velocity can be made using the current and previous timestamped values (Fig. 6 position (c)). Further steps in the forward direction allow the LS algorithm to use more data in its' estimation of velocity. Table 2 shows the LS estimators used as a function of how many pulses in the same direction are received. Note that the more points used in the LS algorithm the smoother the estimate will be.

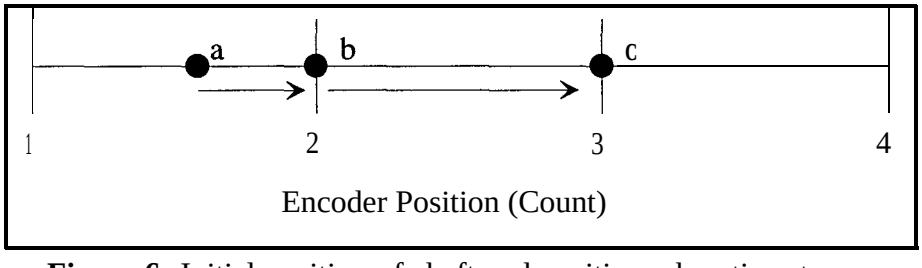

Figure 6: Initial position of shaft and position when timestamps occur

Suppose now a pulse in the reverse direction occurs. Since it is not known where the shaft actually stopped rotating and reversed directions we cannot make any estimate of the velocity. This situation corresponds to (d) in figure 7, i.e. pulses occur at (b) and (d) but (c) is where motion reverses direction. This places us in state B1 where it is known we have one valid time-stamped position

measurement. If a second pulse in the reverse direction follows, corresponding to (e) in figure 7, the state advances to B2 and a velocity estimate may be made.

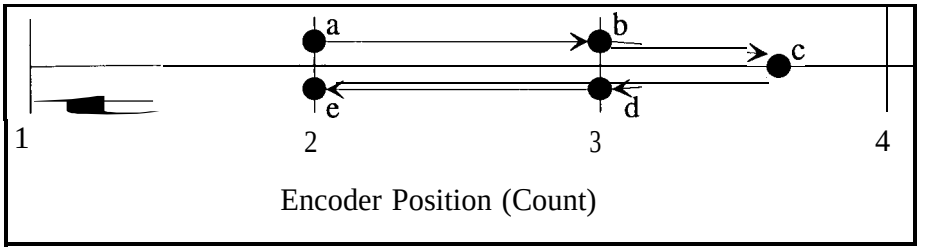

Figure 7: Position in forward motion at timestamps, position of motion reversal, position in reverse motion at timestamps

Number of Pulses in Same Direction	LS Estimate
2	Linear w/ 2 Data Pts. (N=1, M=2)
3	Linear w/ 3 Data Pts. (N=1, M=3)
4+	Quadratic w/ 4 Data Pts.(N=2, M=4)

Table 2: Least Square Fit as a function of number of data points available

15.3 Experimental Results: Motor-Mass System

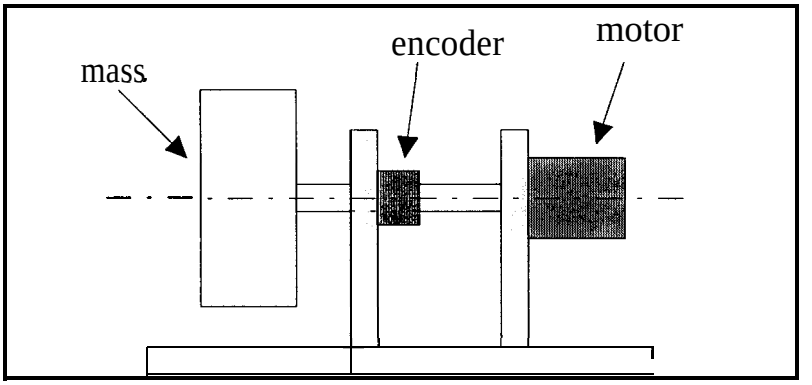

Figure 8. The Motor-Mass System

The above algorithms were combined and tested on an experimental apparatus being designed for classroom use. The apparatus consists of a rotational mass driven by a DC motor and its rotational position is measured by an optical incremental encoder. We constructed a low resolution "pseudo encoder" by dividing the real encoder readings with a constant resolution factor and then rounding the values to the nearest integers. Therefore there are effectively two encoders of different resolutions. We estimated the velocity from the low resolution position. A simple velocity PI controller was used with the low resolution velocity to track a sinusoidal profile. The tracking was compared with the high resolution encoder to check its performance.

In Figure 9 we show the case without using the transition logic algorithm. In the position part of the figure the stairs represent the low resolution positions with resolution 2000 times less than those coming from the real encoder. The relatively high resolution velocity was calculated from the derivative of the high resolution positions. In the velocity part of the figure, the solid line is the reference velocity profile, the star line is the velocity estimated from low resolution position measurements, and the high resolution velocity, the circle line, is the real control result. Obviously

the velocity was poorly estimated, especially when the rotation changed directions or for a long period of time no new position appeared. Those two situations did introduce unexpectedly huge oscillations both to velocity and position.

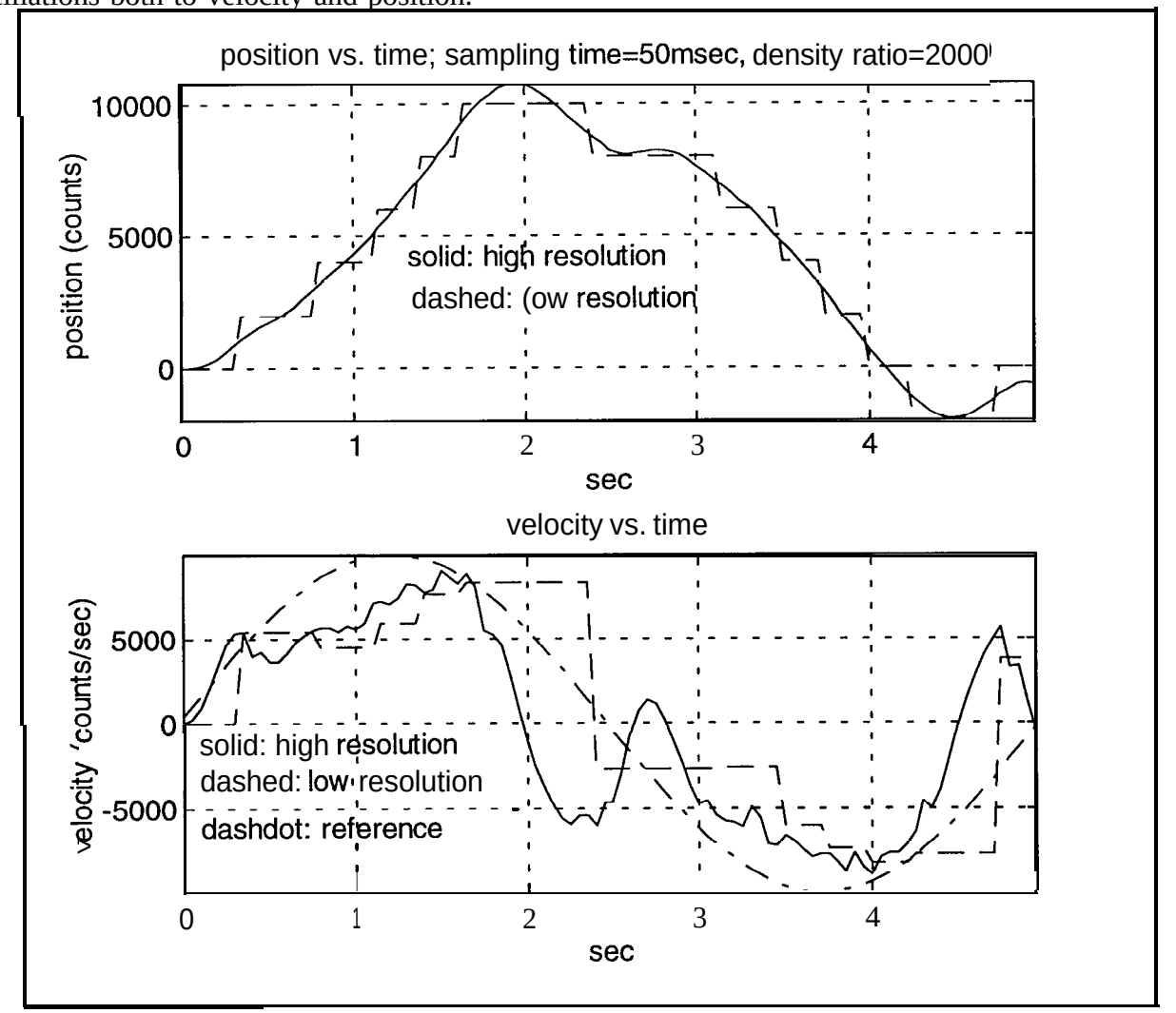

Figure 9. Velocity Estimate and Control: Transition Logic Algorithm Not Used

When the transition logic algorithm was applied, the system performance shown in Figure 10 did improve significantly. There is still some time delay in the estimated velocity because no new position information is available during a sampling period. The transition logic algorithm works reasonably well especially since no system model is needed for this method.

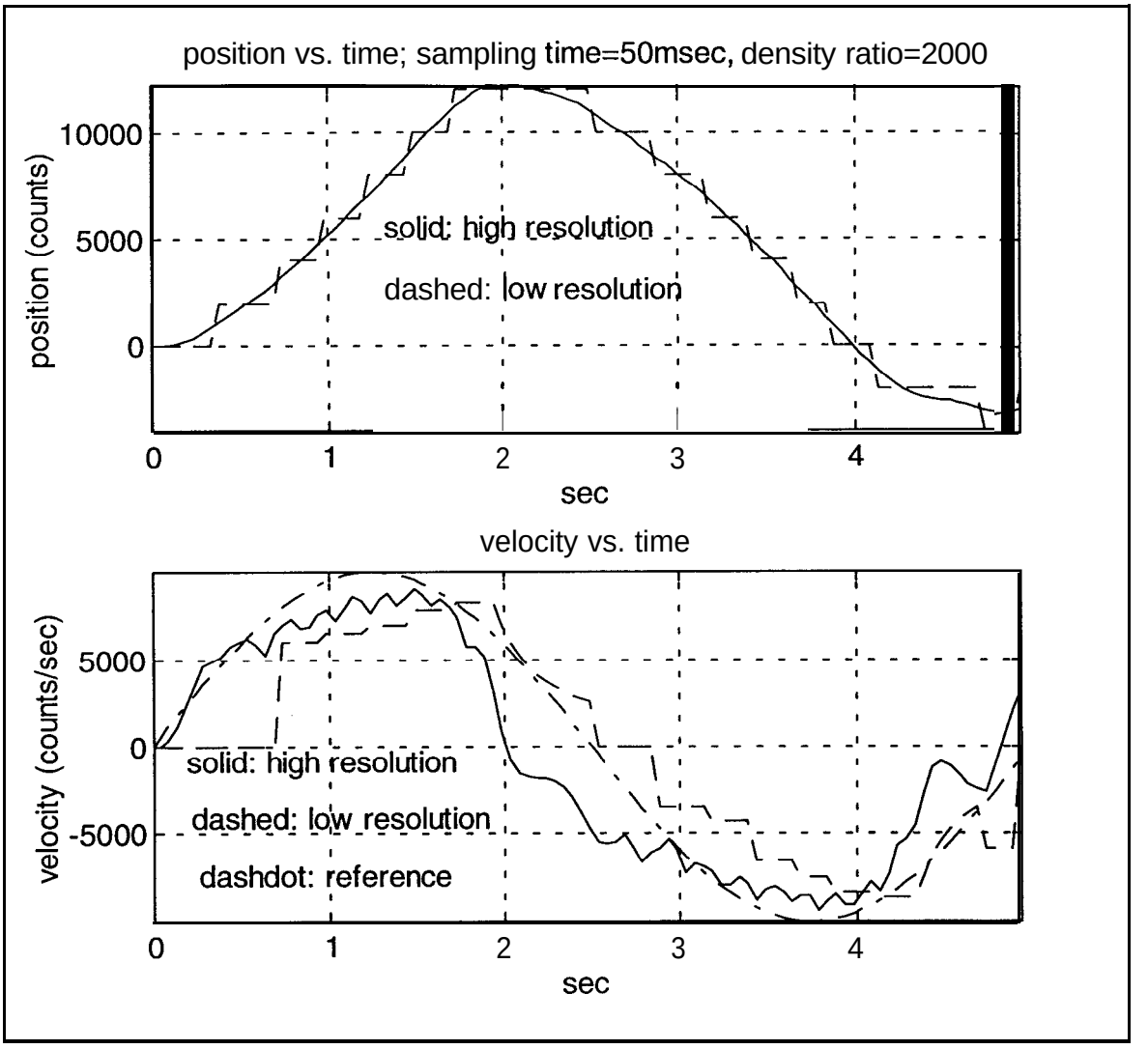

Figure 10. Velocity Estimate and Control Using Transition Logic Algorithm

15.4 Time delayed averaging method

Another algorithm under development is the time delayed averaging method (TDAM). By averaging the past m fixed time estimates of velocity we may smooth out noise due to measurement inaccuracies and sampling artifacts. The TDAM estimator is given by:

$$\frac{d\hat{x}_k}{dt} = \frac{\frac{x_k - x_{k-1}}{T} + \frac{x_{k-1} - x_{k-2}}{T} + \cdots + \frac{x_{k-m-1} - x_{k-m}}{T}}{m} \qquad (14)$$

Collecting terms from equation (14) we find

$$\frac{d\hat{x}_k}{dt} = \frac{x_k - x_{k-m}}{m\ T} \qquad (15)$$

In some sense the estimated velocity is the central derivative at time (k - (m/2))T and therefore there exists a time delay of (m/2)T if the estimate is regarded as the velocity at time kT. One of this method's advantages is its' ability to filter the impulse effects resulting from discrete position measurement at slow speeds, thus smoothing the velocity estimate. In addition this method reduces the impact of small oscillations about encoder pulses on the estimate, achieving an effect similar to the transition logic algorithm. Since the position measurements are taken over a larger time, inaccuracies due to unsymmetries in the quadrature are also filtered out to some degree. A disadvantage to this method is the inherent time delay of (m/2)T which in high acceleration situations may cause problems with stability and tracking.

15.5 Asynchronous Multirate Observer Based Estimator

In the current technology section an attempt at utilizing a model of the system to improve estimates of the velocity was undertaken. Unfortunately this method has several problems. The position sensor is ***sampled periodically*** for the arrival of a new pulse. In order to deal with sampling errors arising from missing the exact time when the pulse occurred, the velocity fed into the estimator was taken as the average of the two previous velocity estimates. This is actually a special case of the TDAM estimator. In order to get a true measurement of the velocity it becomes necessary to collect many samples, thus introducing significant time delays. In addition and perhaps more importantly since the previous estimator is only updated when a new pulse is sensed it becomes impossible to update the controller more frequently than the arrival of pulse rates. This is highly undesirable in situations when the pulse rates are relatively slow, such as longitudinal control of an automobile.

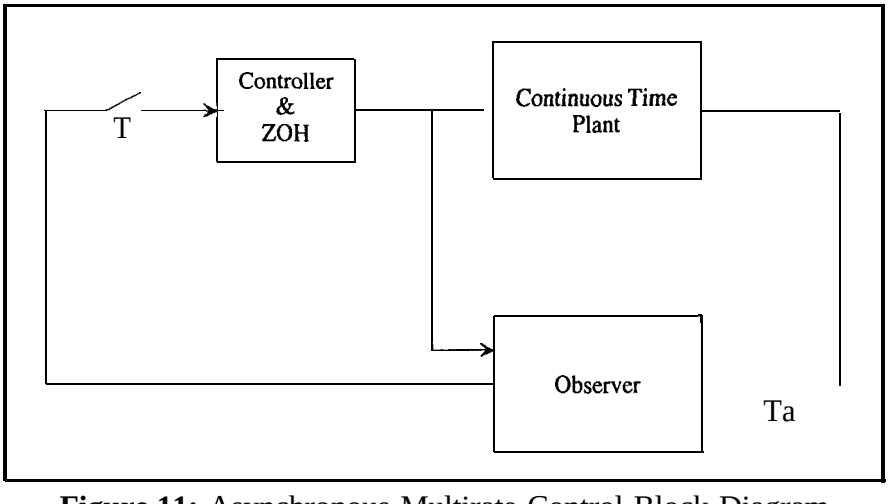

Figure 11: Asynchronous Multirate Control Block Diagram

Instead of viewing the observer / plant / controller as having one update rate it is more convenient to view the system in the multirate paradigm, where different update and sample rates are free to take place. Multirate controller design was initially developed the 1950's. A good explanation of basic multirate control may be found in [6]. Since the early developments a significant amount of research in multirate control has taken place in the area of ***periodic*** multirate systems and how to design controllers to these systems [7,8]. The class of systems for which we are interested in is indeed a multirate system, however unlike previous

systems the output of the system is asynchronous. By choosing the update rate of the observer and controller to occur periodically at a rate faster than the asynchronous output measurements better performance may be expected. Note that even though new sensor data may not be present it is still possible to update the observer based on the values of the control being sent to the plant between output samples. A diagram of an asynchronous multirate system of the type discussed above is shown in figure 11. The update rate of the controller and observer occurs periodically every T seconds, while the outputs from the plant is updated to the observer asynchronously every Ta seconds, where Ta varies from sample to sample.

15.6 Modified Luenberger Observer with Output Estimator

The previous Asynchronous Multirate Observer Based Estimator has two chief drawbacks: implementation difficulty and inefficiency for the control purpose. Owing to a non-periodic output, the plant has to be discretized every period with respect to its previous sampling period and, based on the discrete time model, the observer gains must be renewed also. This consumes a lot of on line computational power. On the other hand if the output signals are very sparse compared to controller, then the control will remain constant for a long period until the next system output signal appears. That seems very inaccurate and inefficient. Could there be some way that the control still be updated every control interval? It may improve the control performance to meet a more complicated control objective. Here we propose a kind of observer shown in Figure 12 without such weak points.

Consider a linear time invariant system with state and output equations

$$\begin{aligned} \frac{dX(t)}{dt} &= AX(t) + Bu(t) \\ \mathbf{y(t)} &= CX(t) \end{aligned} \qquad \textbf{(16)}$$

where y(t) is the position output. The velocity can be derived as

$$v(t) = \frac{dy(t)}{dt} = C\frac{dX(t)}{dt} = CAX(t) + CBu(t) \qquad (17)$$

The discretized system with respect to the constant sampling time of controller T thus can be described by

$$\begin{aligned} X(k+1) &= A_d X(k) + B_d u(k) \\ y(k) &= CX(k) \end{aligned} \qquad \textbf{(18)}$$

where

$$\begin{aligned} B_d &= \int_0^T e^{A(T-\tau)} B\, d\tau \\ A_d &= e^{AT} \end{aligned} \qquad (19)$$

The velocity is thus discretized as

$$v(k) = CAX(k) + CBu(k) \tag{20}$$

The original Luenberger state observer has the form of

$$\begin{aligned} \hat{X}(k+1) &= A_d\hat{X}(k) + B_d u(k) + L(y(k) - C\hat{X}(k)) \\ y(k) &= CX(k) \end{aligned} \tag{21}$$

Now the output is updated asynchronously every Ta seconds, where Ta varies from sample to sample, therefore when updating the controller and observer every T seconds, we must estimate the system output y(k) in advance. Here we assume the velocity varies very little during every sampling period T seconds and the system output may be estimated as

$$\begin{aligned} \hat{y}(k) &= \hat{y}(k-1) + \hat{v}(k-1)\cdot T \qquad if\ y(k) = y(k-1) \\ &= y(k-\delta) + \hat{v}(k-1)\cdot \delta T \qquad if\ y(k) \neq y(k-1) \end{aligned} \tag{22}$$

where y(k-6) = $y((k-\delta)T)$ is the position output renewed within last sampling period T and $0 \le \delta \le 1$. That is if there is no output feedback during last sampling period T, we just guess the current position using the first equality above, otherwise, we can update current position using the latest system position output update value through the second choice of the estimator above. The velocity at the moment kT is thus estimated as

$$\hat{v}(k) = CA\hat{X}(k) + CBu(k) \tag{23}$$

Therefore the modified Luenberger state observer for the asynchronous multirate system becomes

$$\hat{X}(k+1) = A_d\hat{X}(k) + B_d u(k) + L(\hat{y}(k) - C\hat{X}(k)) \tag{24}$$

and we can follow the similar procedure of observer gain selection to the original Luenberger observer to design our observer, i.e., choose L such that the eigenvalues of the matrix $[A_d - LC]$ all lie inside the unit circle. This could approximately make the observer stable. The detailed consideration of stability and convergence of this observer would be an important issue.

15.7 Experimental Results: Motor-Mass System

Using the same setup as the previous experiment, we feed the motor with a known set of input voltages and read the related output positions from the high resolution encoder. After fitting them with an ARX model, the system describing difference equation is

$$y(k) - 1.8575y(k-1) + 0.8577y(k-2) = 46.1054u(k-1)$$

From it, we get the continuous transfer function

$$\frac{Y(s)}{U(s)} = \frac{0.0485s + 1.9895}{s^2+3.0709s+0.0748} \times 10^4$$

If we select x_1 as position and x_2 as velocity, then the system state and output equations become

Let sampling time for the controller and observer output be 50 milliseconds, i.e. T = 0.05 sec.. The

$$\frac{d}{dt}\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -0.0748 & -3.0709 \end{pmatrix}\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1.9895 & 0.0485 \end{pmatrix}\begin{pmatrix} u(t) \\ \dot{u}(t) \end{pmatrix}$$

$$y(t) = [1 \quad 0]\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = x_1(t)$$

system then can be described by the following discrete time states equations,

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \begin{pmatrix} 0.9999 & 0.0463 \\ -0.0035 & -0.8576 \end{pmatrix}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \begin{pmatrix} 23.6430 & 0.5764 \\ 922.1150 & 22.4793 \end{pmatrix}\begin{pmatrix} u(k) \\ \dot{u}(k) \end{pmatrix}$$

$$y(k) = [1 \quad 0]\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} = x_1(k)$$

where we can approximate the derivative of u as

$$\dot{u}(k) = \frac{u(k) - u(k-1)}{T}$$

The state observer and output estimator are then designed as

$$\begin{pmatrix} \hat{x}_1(k+1) \\ \hat{x}_2(k+1) \end{pmatrix} = A_d\begin{pmatrix} \hat{x}_1(k) \\ \hat{x}_2(k) \end{pmatrix} + B_d\begin{pmatrix} u(k) \\ \frac{u(k)-u(k-1)}{T} \end{pmatrix} + L(\hat{y}(k) - \hat{x}_1(k))$$

$$\hat{y}(k) = \hat{y}(k-1) + \hat{x}2(k-1)\cdot T \qquad if \ y(k) = y(k-1)$$
$$= y((k-\delta)T) + \hat{x}2(k-1)\cdot \delta T \qquad if \ y(k) \neq y(k-1)$$

and we can select L as $[1.017, 0.979]^T$ to make the eigenvalues of [A,, - LC] lie at 0.6 ± 0.1 i. The experimental results are shown in the figure below. The reference velocity is 10000 * $\sin(2\pi t/10)$ counts/sec and a velocity PI controller is applied.

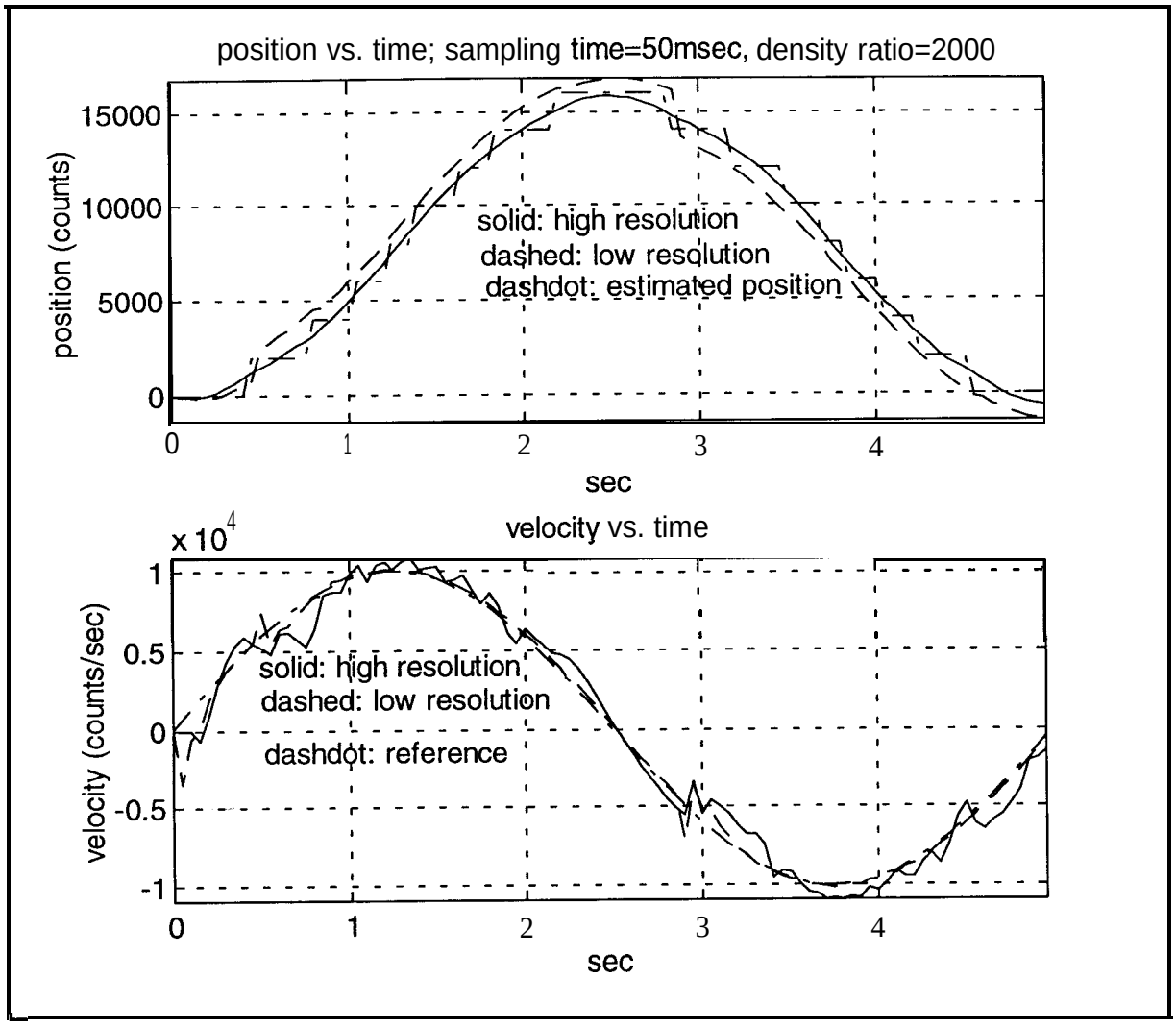

Figure 13. Velocity Estimate and Control Using the Modified Luenberger Observer with an Output Estimator

In the position part of Figure 13, the stairs are the positions from the pseudo low density encoder while the solid line comes from the real high resolution encoder with density ratio 2000: 1. Meanwhile, the dashed line represents the estimated positions, i.e. x_1_hat which are estimated from our modified Luenberger Observer and are corrected when every new low resolution position information comes in.

In the velocity part of Figure 13, the solid line represents the reference velocity which our system is intended to track. The star line is for the velocity values obtained from our modified Luenberger Observer. We feed a velocity PI controller with the difference of the estimated velocity and the reference velocity. The result seems fairly good and is shown by the circle line which represents the high resolution velocity, the derivative of high resolution positions.

Compared to the transition logic based switching algorithm, this method seems to be a better velocity estimator due to the knowledge of a system model information. The main advantage of this method is that the position and velocity can be estimated precisely to a confident extent during the interval without new position information. Therefore we can update the controller every instant and obtain a outstanding control result although just using a low resolution encoder. However, the response still shows a "glitch" in the velocity response when going through zero. This is because the point of turnaround can never be known to the controller. The transition logic method gives information on how much information is available, which can be used in conjunction with this estimator to reduce the estimation errors associated with velocity reversals.

Here we have assumed that a confident system model exists and pay no attention to the robustness of this method. How well the model is identified may be a major factor for the observer to perform successively. Future work for this method would put a lot of emphasis on the robustness issue and also try to create adaptation methodologies to deal with system uncertainty.

16 Conclusions and Future Work

Previous work in velocity estimation from discrete position and time data has resulted in many algorithms by which velocity may be estimated. However, a survey of currently available methods of velocity estimation show they are particularly lacking in their ability to handle two classes of systems. Specifically, previous algorithms considered in this survey do not address the problems of velocity reversals and velocity estimation where position data is sparsely available. In addition, estimation of velocity in between output sampling instants for use in systems where desired update rates are higher than output sampling rates has not been encountered in this survey. Possible solutions to both problems have been proposed: a transition logic based approach for the problem of slow velocity estimation with velocity reversals, an asynchronous multirate observer for higher update rates. In addition, the modified Luenberger observer with a output estimator seems very promising for it can update the control law more efficiently to meet various control objects. Owing to its model based feature, robustness and adaptation methodologies will be an important issue to look at. Further work in this area will concentrate on refining and generating a more complete understanding of these methods.

17 References

[1] R. H. Brown, S. C. Schneider, M. G. Mulligan, "Analysis of algorithms for velocity estimation from discrete position versus time data," ***IEEE Transactions on Industrial Electronics,*** vol. 39, no. 1, pp. 11 - 19, Feb. 1992.

[2] R. H. Brown, S. C. Schneider, "Velocity observations from discrete position encoders," in *Proc. IECON'87, **Thirteenth Annu. IEEE Industrial Electronics Society** Conf.*, Boston, MA, Nov. 1987, pp. 1111- 1118.

[3] K. Saito, K. Kamiyama, T. Ohmae, and T. Matsuda, "A microprocessor-controlled speed regulator with instantaneous speed estimation for motor drives," ***IEEE Transactions on Industrial Electronics,*** vol. 35, no. 1, pp. 95 - 99, Feb. 1988.

[4] J. Tal, "Velocity decoding in digital control systems," in *Proc.* ***Ninth Annual*** *Symp.* ***Incremental Motion Contr.*** *Syst.* ***Devices,*** June 1980, pp. 195-203.

[5] D. W. Love, PATH year end progress report, 1994

[6] R. E. Kalman, J. E. Bertram, "A unified approach to the theory of sampling systems," ***Journal Franklin Institute,*** pp. 405 - 436, May 1959.

[7] B. D. 0. Anderson, "Controller design: moving from theory to practice," ***IEEE Control Systems,*** pp. 16 - 25, Aug. 1993.

[8] P. Colaneri, R. Scattolini, N. Schiavoni, "Stabilization of multirate sampled-data linear systems," ***Automatica,*** vol. 26, no. 2, pp. 377 - 380, Mar. 1990.