Software for Advanced Traffic Controllers

DARCY BULLOCK AND CHRIS HENDRICKSON

A systematic approach to traffic engineering software development could provide significant advantages with regard to software capability, flexibility, and maintenance. Improved traffic controllers will likely be essential for many proposed intelligent vehicle highway system applications. A computable language called TCBLKS (Traffic Control Blocks) that could provide the foundation for constructing real-time traffic engineering software is introduced. This computable language is designed to be configured by a graphical user interface that does not require extensive software engineering training to use yet provides more flexibility and capability than is possible by simply changing program parameters. The model is based on the function block metaphor commonly used for constructing robust and efficient real-time industrial control systems. Adapting this model to the transportation sector permits traffic control applications to be programmed by (a) selecting preprogrammed function blocks from a standard library, (b) configuring block parameters, and (c) connecting blocks to other blocks in the strategy. The software model was implemented in C on an advanced traffic controller platform and demonstrated in real time for applications such as signalized intersection control and ramp metering. In addition, this same software model was used to control the ramp meters along a segment of westbound Highway 50 during a demonstration in Sacramento, California, in November 1992.

Twenty years ago, traffic controllers underwent a technical revolution in the switch from electromechanical systems to solid-state microprocessor systems. With the computing technology available 2 decades ago, the most cost-effective approach for software development was to construct specialized, embedded systems tailored to the traffic control industry. Traffic control logic was programmed using assembly language programs that could read and write bits associated with external sensors and actuators. Initially, these microprocessor-based controllers did little more than their mechanical predecessors. Over time, transportation engineers realized that more and more features could be implemented on solid-state controllers, and they upgraded their software accordingly.

Today there is another turning point in traffic control technology. The advances in microprocessor technology over the past decade have dropped the average cost of computing by roughly an order of magnitude every 5 years (*1*). In contrast, the average cost of a Caltrans Type 170 controller has decreased only 15 percent since 1987 and 27 percent since 1982 (California Department of Transportation, personal communication, July 1992). Off-the-shelf, field-hardened, and affordable equipment is available that rivals the computing power of mainframes from 20 years ago. If computing costs continue to decline in this manner, it will no longer be cost-effective for proprietary transportation computers to compete with mass-produced industrial hardware. Migration to these more powerful computers will allow traffic engineers to make a fundamental change in software development practices. Memory capacity and processor limitations will not impose significant constraints on applications. Instead, traffic engineers can focus on developing an efficient architecture for building systems that are more effective, easier to install, and easier to maintain.

This change should have numerous benefits. Since the inception of the microprocessor-based traffic controller, the software engineering effort devoted to constructing traffic control software has been less than ideal (*2,3*). In general, the state of the practice with current microprocessor software is to write software in assembly language without an operating system, permanently install the software on a chip (i.e., burn it into a ROM), and empirically test programs to see if they work. Provided a suitable programming model can be developed, it is now possible to engineer software for greater capability, flexibility, and usefulness. However, no substantial model has been proposed. Such models are crucial for the evolution of an engineering discipline from a solely craft-based practice (*4*).

Selecting an appropriate software model is particularly important because the development of transportation control systems is multidisciplinary, requiring the interaction of transportation engineers, electrical engineers, software engineers, and government officials. In the past, the coupling between traffic engineering concepts and field implementation has been weak. This paper presents a software model that is designed to address professional communication gaps and the need for more capable and maintainable software. It is based on the "function block metaphor" that is widely used in industrial control systems. This model provides the capability for nonprogrammers to develop intuitive control software by drawing graphic diagrams on a computer screen and filling in menus. This model is based on a formal, real-time scheduling algorithm that allows the correctness and feasibility of strategies to be formally verified. It has been so successful in the industrial sector that many companies have imposed rules restricting development of custom software and require application developers to use "canned software" applications consistent with the function block model.

The following section introduces as background the rationale and characteristics of an advanced traffic controller hardware platform. The subsequent sections describe the model of traffic control software developed in this study. A final section discusses the status of the proposed software model.

D. Bullock, Department of Civil Engineering, Louisiana State University, Baton Rouge, La. 70803. C. Hendrickson, Carnegie Institute of Technology and Department of Civil Engineering, Carnegie Mellon University, Pittsburgh, Pa. 15213.

ADVANCED TRAFFIC CONTROLLER HARDWARE

Most of the current generation of traffic controllers used in the United States are based on two different families of controllers. This section summarizes these competing efforts and describes some hardware modernization efforts under way.

One family of controllers, referred to as NEMA units, are built with connectors that conform to standard mechanical and electrical connectors. The philosophy of this standard is that manufacturers will compete on the basis of the hardware and software they provide inside the controller. In theory, an agency can migrate to another manufacturer's controller by unplugging the old one and plugging in the new one using standard connectors. Because of additional proprietary sockets added to the NEMA TS1 units and nonstandard communication protocols, this interchangeability is not realized in practice. The 1988 NEMA TS1 standard recently has been updated (NEMA TS2 Type 1 and NEMA TS2 Type 2) to address deficiencies of the NEMA TS1 standard and incorporate an alphanumeric display for interaction with the controller. Because the software on all NEMA controllers remains proprietary and cannot be ported by the customer, engineers and technicians must still learn new software to reconstruct timing and phasing plans on new NEMA controllers.

A second family of controllers, referred to as the Caltrans Type 170 controllers, is built to provide both standard connectors and portable software. The philosophy of this standard is to develop a very precise specification for a traffic control microcomputer. Manufacturers are selected periodically on the basis of competitive bidding. This standard has been successful for the past 20 years. Minor modifications have been introduced over time, including a second serial port, additional memory, and different ROM sizes, but the essential features are unchanged. The distinguishing feature of the 170 controller remains the program module—an insertable card with a ROM that stores the traffic control program. This module can be removed from one manufacturer's 170 controller and inserted into another manufacturer's controller, and the software will run without modification. This decoupling of the hardware and software procurement is very desirable, particularly when equipment purchases are going to be staged over several years. Instead of relying on embedded user interfaces, as in the NEMA controllers, the 170s are typically configured by connecting a small computer such as a personal computer to a serial port and downloading the strategy. Alternatively, binary configurations can be keyed in on a hexadecimal keypad. A modernization of the Type 170 has been undertaken by New York state and is called the Type 179. This controller provides more powerful computing and employs a real-time operating system. However, it has not been widely adopted. As a result, it has been difficult to develop a pool of competitive vendors.

In view of the microprocessor and software engineering developments in the past 2 decades, these standards are beginning to age (*3*). First, the software is written entirely in assembly language. The complex nature of assembly language development precludes all but the largest cities and states from maintaining a software staff for making software configuration changes other than changing parameters in a given configuration. Second, no operating system is employed (except for some 179 software). Routine chores, such as task scheduling, memory management, and semaphores, must be recoded. The "home-grown" executives that have evolved preclude sharing of new control strategies. Third, the hardware constraints (slow processors and limited memory) can only be addressed by a revised standard that would require rewriting large quantities of assembly language applications. Finally, it is unclear how much longer it will be cost-effective for the transportation community to manufacture specialized computers.

These shortcomings and requirements for improved software development tools, faster processors, expandable I/O, and more memory have led the California Department of Transportation to investigate the use of modular industrial computers for applications ill suited to the Caltrans 170s (*5*). This proposed platform, called advanced traffic controller (ATC), is based on a 3U VME bus, a 680x0 processor, and an OS-9 operating system. This computer is used extensively in the military and commercial sectors and provides an economical, off-the-shelf hardware platform for the ATC. Although a rich set of development tools, including operating systems, compilers, and debuggers, are available for this platform, the low-level nature of the tools renders them inappropriate for everyday use by traffic engineers. This is analogous to a desktop computer that has only a language compiler. For a desktop computer to be truly useful to an engineer, application software such as a spreadsheet or CAD package must be available. Because of this ATC software void, a general-purpose application program (software model) is required to enable traffic engineers to develop real-time traffic control strategies.

COMPUTABLE LANGUAGE FOR TRAFFIC ENGINEERING

The motivation for developing a computable language is to provide a high-level configuration tool that does not require extensive software engineering training to use, yet provides more flexibility than just changing program parameters. The model underlying the language proposed in this paper is based on function block programming in which the function blocks specialized to traffic engineering are graphically assembled and downloaded to a field controller. In the function block programming paradigm, a user develops applications by selecting and connecting predefined software modules called blocks. The blocks represent parameterized programs prepared in a uniform manner, which permits them to be interconnected with other blocks. Connections between blocks serve as communication links for particular variables such as detector states, approach volumes, or phase timing. Selection of blocks may require definition of parameters, such as execution frequency, minimum and maximum green extensions, and filter times. Function block programming lends itself readily to graphical displays in which blocks are represented pictorially as a box with a title, indicating the program associated with a particular block, and a name, providing a symbolic means of referring to elements of a specific block. Figure 1 shows an example of a simplified semiactuated signal with presence detectors on the east and west approaches. Con-

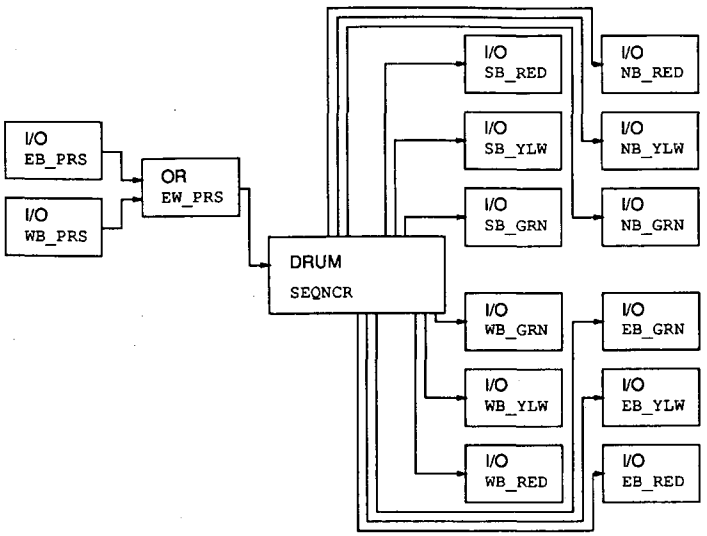

FIGURE 1 Semiactuated intersection control strategy using five function block types.

nections, or data flows, are shown as link connections between the boxes. A typical function block program resembles an activity-on-node (PERT) project management scheduling network.

Function block programming is different from the modular design taught in introductory programming classes because the end user never encounters any procedural code. All interaction with hardware devices, protocol conversions, buffers, timing demands, and error recovery are embedded in a parameterized function block graphical icon that can be configured by a traffic engineer using a function block editor. The blocks available within the function block editor are prepared by software engineers in a standardized manner, which permits seamless interconnection and implementation. This set of blocks is called TCBLKS, an acronym for traffic control blocks.

TRAFFIC ENGINEERING FUNCTION BLOCK PROGRAMMING MODEL

The previous section introduced the function block programming model. This section addresses three areas: (a) traffic engineering task vocabulary, (b) configuration of function block strategies, and (c) function block language structure.

Traffic Engineering Task Vocabulary

The set of building blocks available in the block library constitutes the "vocabulary" for users to assemble applications. Table 1 summarizes the 40 blocks that have been developed. This library includes signal sequencing blocks, signal filters, logic functions, interfaces to external sensors and actuators, archival functions, and various algorithmic blocks. The intent of establishing a definition of these control blocks is to provide a vocabulary that can be assembled by a traffic engineer (in a sketch or diagram) to define the required software. This concept is used extensively in the chemical and process engineering fields so that there is an almost one-to-one correspondence between the process and instrumentation diagram (P&ID) developed by the chemical engineer and a function block strategy constructed by the control system contractor. The same continuity is sought for traffic engineering.

Because this model is only in the prototype stage, the blocks described in Table 1 currently fall short of providing a comprehensive set of building blocks. To support the growth of this model, new blocks can be created and included in the block library as long as the new blocks conform to standard block definition and operation practices. Thus, applications such as a dynamic signal control algorithm such as OPAC (*6*) could be included in a single-function block. In general, this model supports blocks of varying execution complexity ranging from simple logic gates to complex blocks supervising several ramp meters. For example, the simple blocks, such as mathematical computations and digital logic, are necessary for incorporating minor operational changes typically required by peculiar geometric or policy constraints. In contrast, the complex blocks such as ramp metering or intersection control algorithms can provide rapid and reliable task-level programming.

Configuration of Function Block Strategies

An advantage afforded by the function block programming model and advocated in this paper is the ability to easily "program" or configure robust traffic control software without an extensive software engineering background. A typical configuration tool can operate like a simple vector drawing package commonly found on desktop or notebook computers. Instead of manipulating shapes and lines, it manipulates function blocks. A block program is developed by assembling a strategy composed of predefined blocks that provide common traffic engineering operations. The mechanics of constructing such a strategy can be viewed in three steps.

TABLE 1 Function Block Summary

AND, OR, XOR, NOT: These blocks perform the essential boolean logic operations on their input(s).

DlyOn, DlyOff, OneShot: These digital blocks perform digital logic timing operations. The DlyOn block delays a transition from low to high for a specified time. Alternatively, the DlyOff block delays a high to low transition for a specified time period. The OneShot provides a pulse generating mechanism for transitions from low to high.

D-Shift: provides a 16 bit shift register for transient storage of digital states.

D-UI: provides an operator with simple on/off and pulse operations for user interfaces.

Match: provides basic decoding functionality.

Timer: measures the duration of digital events.

Counter: can be used for counting lo-high transitions.

FF-RS, FF-D, FF-JK: These blocks provide discrete implementations of clocked RS, D, and JK flip flops. A T flip flop can be constructed from the JK flip flop.

Drum: provides state sequencing subject to minimum and maximum durations with the capability of back stepping.

Rate: calculates the filtered rate of an incoming digital pulse train.

Add, Mult, Div: These block provide basic mathematical operations.

Mavg, A-Shft: Both blocks implement a circular queue. The Mavg block uses the queue to compute the moving average of a time series. The A-Shft block provides a mechanism for introducing a time delay (lag).

A-Latch: latches an analog value when a digital pulse is received.

A-SWITCH: selects between two analog signals based on the state of a digital input.

A-UI: provides an operator with a mechanism to enter an analog value for a user interface.

Filter: provides a simple discrete approximation for a first order analog filter.

Test: compares an input against a set of absolute Hi and Lo bounds or relative to another signal. The results of these comparisons are digital points other blocks can connect to. It is useful for implementing conditional logic.

Sel-H, Sel-L, Sel-M: High, low and middle selector blocks. The first two blocks have two inputs, the middle selector requires 3 inputs.

RMSB: provides supervisory rate selection of a ramp metering rate based upon one upstream volume sensor and up to six downstream occupancy values.

LOOKUP: provides an interpolated lookup table for defining non-linear transformations

D-Coll, A-Coll: monitors up to eight inputs (Analog or Digital) and records their state to a file. A background spooler is set up so this file can reside on any OS-9 file device. These devices include hard disks, floppy disks, RAM disks and non volatile disks.

RMDI, RMDO: used to read digital inputs (DI) or write digital outputs (DO) on a 170 running ramp metering software.

RMRI, RMRO: used to read register inputs (RI) and write register outputs (RO) on a 170 running ramp metering software.

VMS: contains up to 8 prioritized ASCII messages that can be displayed on a variable message sign by a digital event.

1. Select: Blocks providing the requisite device interfaces, signal processing, control computations, cycle phasings, or data collection features are selected and placed on the drawing area.

2. Configure: Parameters defining a program block's operation, such as the number of phases or a loop detector's I/O port, are configured for each block. This procedure is performed by selecting a block with the mouse and choosing the "configure parameters" option. Of course, each block is instantiated with a full set of default values that may be acceptable, in which case this operation can be omitted for many blocks.

3. Connect: The blocks are connected by clicking on a block, selecting a particular block output connection, clicking on another block, and selecting a particular block input connection. Basic error checking is performed to prevent sockets with various data types from being connected. For example, it would be invalid to connect the state of a loop detector to the socket determining the cycle length for a traffic light drum sequencer.

These steps are intended only to give the reader an idea of how the function block model could be configured. In practice, these steps will likely be intertwined as a strategy is developed and edited incrementally. Past strategies would typically serve as templates for new applications. Also, a number of diagnostic, reporting, drawing, scaling, and annotating tools are necessary to round out the features of the configuration tool.

Language Structure

Table 1 provides a summary of some preprogrammed blocks that can used to develop block strategies. This section details the basic architecture of those blocks and how they can be assembled. Abstractly, a function block is a vector consisting of the following elements (Figure 2):

- Input sockets are used either to retrieve data from other blocks or are assigned constant values. Input sockets are actually references to memory locations from which the block reads values. The values stored in those locations can be changed either by another block's output socket or by an operator manually inserting a value. These sockets represent the destination half of a data flow connection.
- Local storage stores block parameters and interim calculations.
- Output sockets are used to store block output values and can be connected to other blocks. Output sockets are actually references to memory locations to which the block will write output data. The values written to those locations can be read by another block's input socket or by an operator examining sockets. These sockets represent the source half of a data flow connection.
- A block algorithm periodically reads the values associated with the input sockets, performs calculations, manipulates local storage, and then updates the output sockets.

Although blocks may have several input or output sockets, it is not required that they all be connected. In fact, input

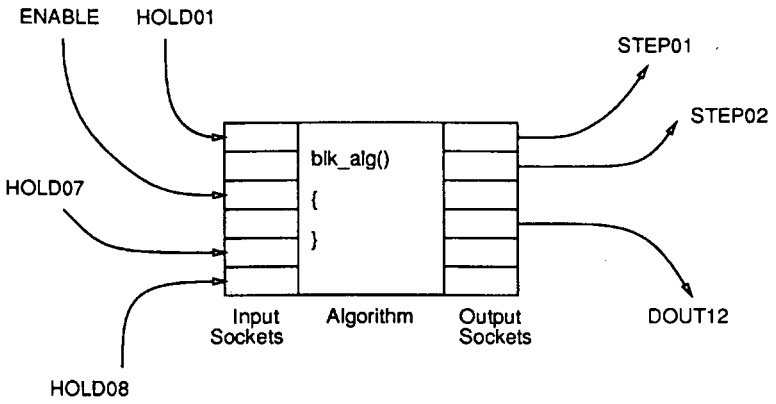

FIGURE 2 Function block components.

sockets can be assigned either constant values (Figure 3, Socket 3) or connected to another block's output socket (Figure 3, Socket 2) during configuration. Similarly, output sockets can be left dangling (Figure 3, Socket 1) or connected to input sockets (Figure 3, Socket 2) on other blocks. The only restriction on connecting blocks is that one input cannot be connected to more than one output socket (Figure 4).

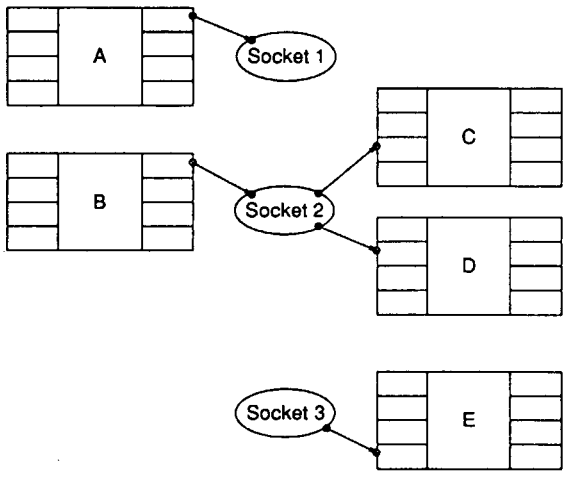

FIGURE 3 Example of valid socket connections.

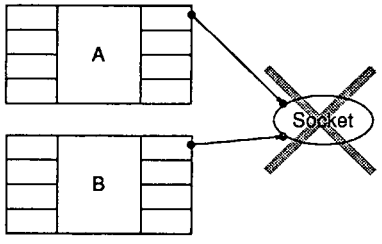

FIGURE 4 Example of invalid socket connection.

IMPLEMENTATION OF TCBLKS

Traffic engineers are likely to be most concerned with the block vocabulary, configuration concepts, and language structure of this traffic control software model. To round out the description of this function block model, a few important implementation concepts are addressed: (a) internal data model for the function blocks, (b) real-time scheduling, (c) capacity considerations, and (d) on-line user interfaces. Our purpose is not to formally define the model but to demonstrate an efficient real-time implementation and to give further insight into the software model. A more extended discussion appears in a previous paper (*7*).

Consider the example strategy shown in Figure 1. This strategy is composed of 16 blocks that describe which sensors should be read, which internal algorithms should be used, and which actuators should be manipulated. Without regard to how often the blocks must be run, this strategy can be described as a topologically sorted list of blocks to be run {EB__PRS, WB__PRS, EW__PRS, SEQNCR, SB__RED, . . . }. Each of these blocks must be represented internally as a data structure with local storage, input sockets, and output sockets. These data structures are different for each block type. For example, the DRUM block has 16 digital output sockets, but the OR block has only one digital output. To provide a structured method for interacting with the various data structures, a master list of blocks called the block table (Figure 5) maintains a list of all the symbolic block names and a code representing the class of blocks. For example, all OR blocks would have a class code of 11 and all I/O blocks would have a class code of 19. This code is used by the software model to determine which table to search to retrieve the data structure defining a block. For example, the table for OR blocks (Figure 5) would contain the data structures defining the EW__PRS block.

Connections between blocks are very important for this model because they provide the mechanism for communication. The connection table (Figure 5) provides a list of all data connections and includes the following information:

- Source socket is a symbolic name identifying the source of a data connection.

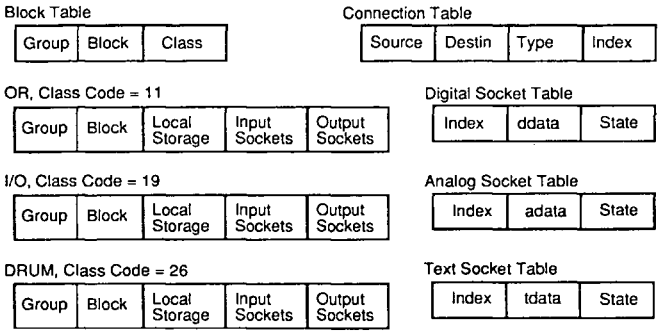

FIGURE 5 Internal model for function blocks.

- Destination socket is a symbolic name identifying the destination of a data connection.
- Socket type indicates which table to look in for the socket. For the data model shown in Figure 5, this could be a reference to the digital, analog, or text socket tables.
- Socket index is used to locate the particular socket in a socket table (digital, analog, or text) identified by the socket type field.

In preceding sections, sockets have been conceptually diagrammed as tightly coupled with the block. However, to improve implementation efficiency, all sockets are stored outside the block and referenced via the connection and socket tables (Figure 5). There is a socket table for each possible data connection type. For example, digital states, analog values, or text messages are stored in the tables shown in Figure 5. When a block is executed, it references its input socket indexes (Figure 5) and retrieves the appropriate information from the socket table. Similarly, after the computations have been performed, it uses the output socket indexes to update the respective output sockets.

Real-Time Scheduling of Function Blocks

Because block processing is not instantaneous, the blocks must be scheduled such that all blocks have an opportunity to run often enough to meet their application requirements. One possible approach would be a round-robin scheduler. The problem with this type of scheduling is that when blocks are added or subtracted the timing characteristics change. This kind of side effect is unacceptable, particularly if interaction with a particular device or evaluation of a traffic signal phase change at regular intervals is necessary. A more sophisticated approach would be to run all the blocks at their fastest required rate (a least-common-denominator approach). This technique would be adequate if sufficient CPU cycles were available for executing all blocks at the fastest required rate. However, in practice, only a few blocks require very frequent service (say 50 Hz) and other blocks require service far less often (say 0.1 or 0.01 Hz).

Because of the varying timing requirements for different portions of a block strategy, it is desirable to be able to assign a processing period to a group of blocks. To provide this capability and introduce a hierarchical level of abstraction, blocks can be grouped and assigned a name and periodic execution rate (Figure 6). Two additional internal tables are constructed to maintain this information: the group table and the task table (Figure 7). An additional status field in the group table is used to turn on and off the processing for an entire group of blocks. From the user's perspective, a collection of groups assigned to periodic tasks constitutes an application program (Figure 8). In the application shown in Figure 8, the blocks in Groups A, B, and C would be run every T_1 sec. Similarly, the blocks in Groups D and E would be run every T_4 sec. Within each of these groups, the blocks, their type, and their configuration define the semantics of the application program.

To facilitate the orderly start-up and shutdown of a function block strategy, the software starts up in a single threaded mode. It reads the function block strategy, creates all the necessary data structures for execution, initializes all I/O devices, runs all blocks once to initialize them, spawns periodic tasks, and commences the periodic execution shown in Figure 8. The periodic tasks are created according to the period and priorities in the task table (Figure 7). Groups are assigned to these tasks according to the task field in the group table (Figure 7). When the software receives a signal to shut down, it allows the periodic tasks to complete their current cycle (only if block processing was in progress before the shutdown signal was received), returns to single-threaded operation, runs all blocks once (permits files to be closed and I/O to be left in a safe state), and then terminates. The state diagram for this behavior is shown in Figure 9.

Capacity Considerations

The periodic tasks shown in Figure 8 represent only one-half of the software model. In practice, interactions with I/O de-

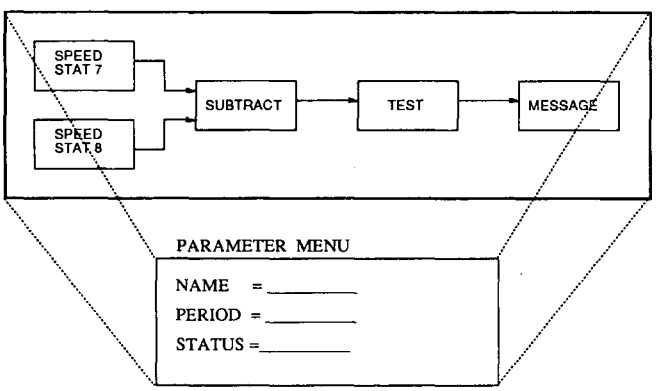

FIGURE 6 Block grouping illustration.

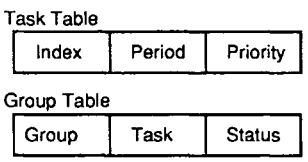

FIGURE 7 Scheduling tables.

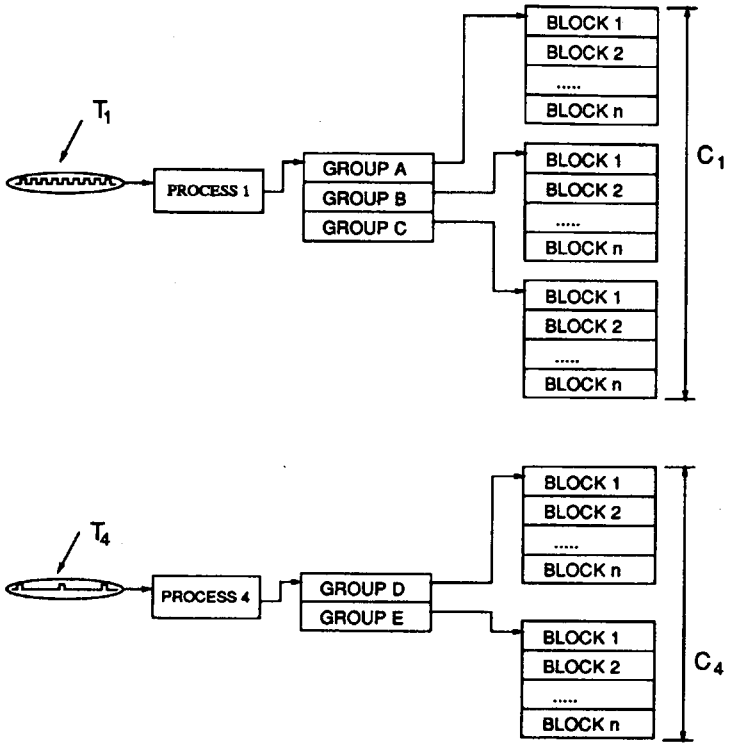

FIGURE 8 Processing of group and block structures by periodic task.

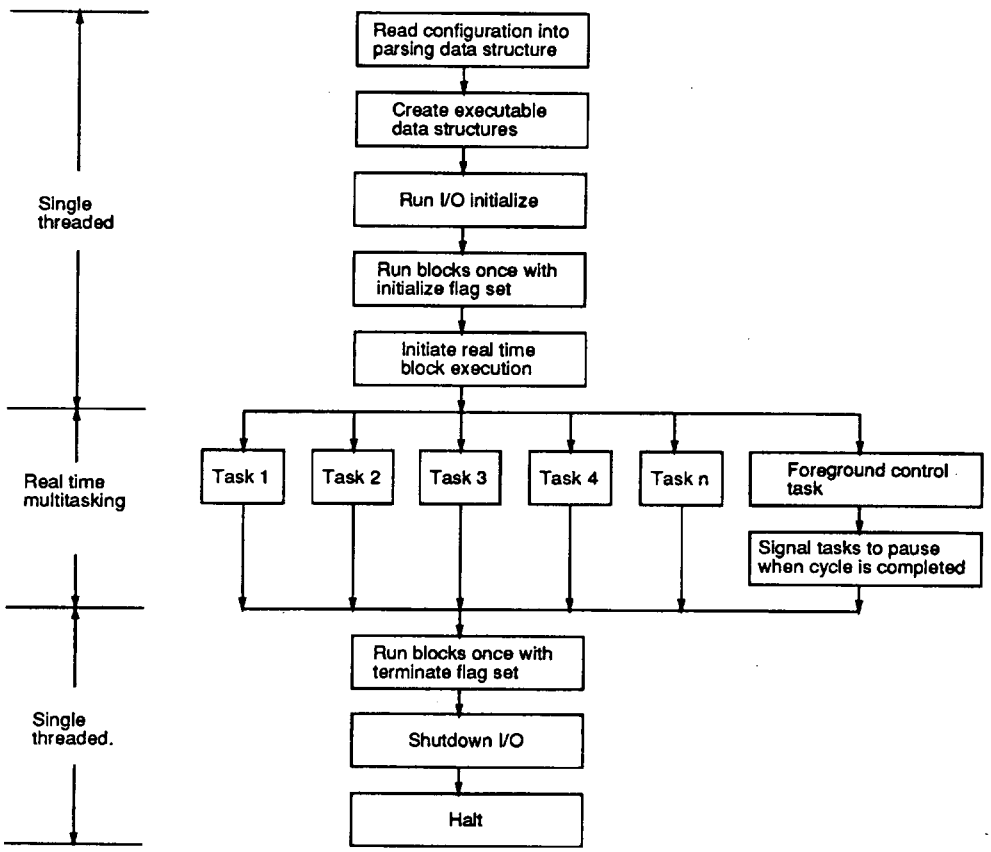

FIGURE 9 Software state diagram.

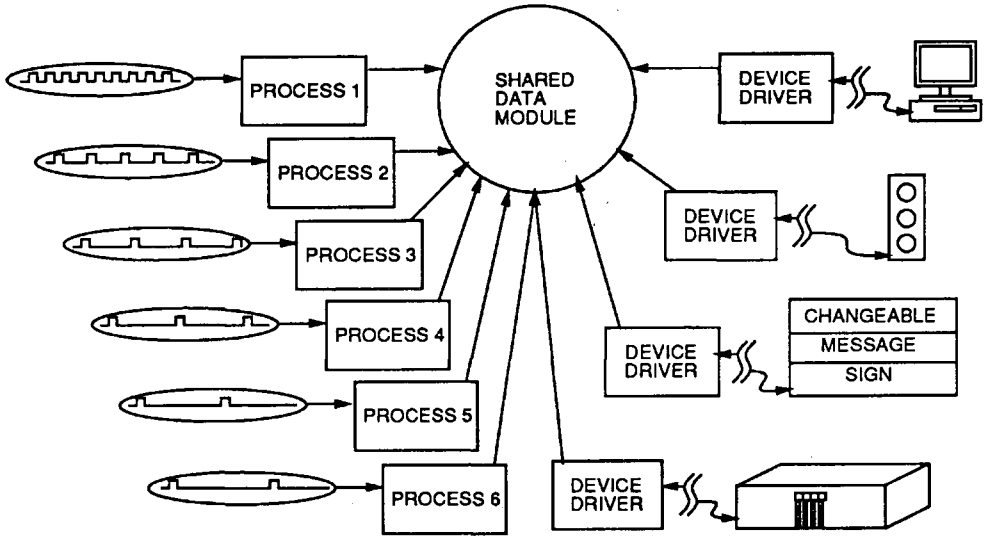

FIGURE 10 Task interaction.

vices such as serial ports, user interfaces, and disk drives have inherent time delays. To permit the processors to work on other duties, the periodic tasks do not directly interact with these devices. Instead, they communicate with asynchronous tasks using internal buffers. Conceptually, this software architecture looks like that shown in Figure 10. The periodic tasks that run the function blocks are shown on the left and the aperiodic tasks interacting with I/O devices are shown on the right. A complex set of tasks is shown in Figure 10. By inspection, it is not obvious whether the software model can respond to all the computational and I/O demands in a timely manner. For example, when monitoring loop detectors it is important that "passage pulses" not be missed. To guarantee that such events are not missed, it is necessary to determine whether the computational demand of the software (Figure 10) exceeds the capability of the processors. This evaluation can be performed using rate monotonic analysis techniques documented previously (*8–10*).

On-Line User Interfaces

The user interface for configuring the block strategy has been described in previous sections. The user interface for instrumentation and monitoring is also very important for development and diagnostic purposes. An interface such as the hex keypad and LED display found on the 170 or the alphanumeric display now being built into NEMA controllers could be used to interact with the ATC software. However, the function block model proposed in this paper provides a more intuitive method for interacting with the run-time control software. The basic concept for developing these "run-time user interfaces" is based on a client-server model in which the client is an operator interface program and the server is the function block processing program. Quite likely, the operator interface would be implemented on a notebook computer that could be plugged into a serial port on the ATC (Figure 11).

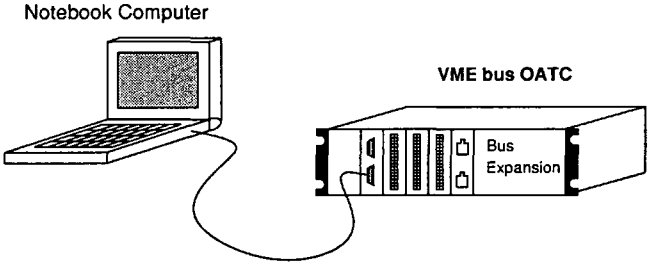

FIGURE 11 Use of a notebook computer to configure and monitor an ATC.

The client operator interface would interact with a strategy via the connection table and the various socket tables (Figure 5).

Interfacing with the controller in this fashion provides two important features. First, the client can symbolically reference any socket. So instead of the current practices on 170 controllers of looking at the word located at a particular hex offset, a symbolic name such as "Main&4th;NB___CNT.AOUT" could be used to read the volume counter on the northbound counter at Main and 4th. Second, the "State" field in the connection table restricts the ability of an operator interface program to write to a socket to only those input sockets not connected to other blocks (Figure 3, Socket 3). Of course, any point could be read by an operator interface, but unpredictable operation would result if an operator was trying to change an output socket that was also being changed by a function block (Figure 3, Sockets 1 or 2).

IMPLEMENTATION

The software model described in this paper has been implemented and tested in real time under simulated conditions for applications such as signalized intersections, ramp metering,

and communication with existing traffic control devices. This software has also been used to implement a bottleneck supervisory control strategy that was field tested along Highway 50 in Sacramento, California. The software communicated with Type 170 ramp meters over leased telephone lines and adjusted metering rates in response conditions at a downstream bottleneck. This demonstration was performed on the proposed Caltrans ATC platform configured with a 16-MHZ 68020 with 4 MB of RAM in November 1992.

ACKNOWLEDGMENTS

This work was supported in part by the California Department of Transportation under subcontract to the University of California Institute of Transportation Studies at Irvine.

REFERENCES

1. A. Rappaport and S. Haleri. The Computerless Computer Company. *Harvard Business Review*, Vol. 69, No. 4, July 1991, pp. 69–80.
2. M. J. Chase and R. J. Hensen. Traffic Control Systems—Past, Present and Future. In *Applications of Advanced Technologies in Transportation Engineering*, ASCE, Feb. 1989, pp. 257–262.
3. D. Bullock and C. Hendrickson. Advanced Software Design and Standards for Traffic Signal Control. *Journal of Transportation Engineering, ASCE*, Vol. 118, No. 3, May 1992, pp. 430–438.
4. M. Shaw. *Prospects for an Engineering Discipline of Software*. Technical Report CMU-CS-90-165. Carnegie Mellon University, Pittsburgh, Pa., Sept. 1990.
5. Quinlan, T. *Evaluation of Computer Hardware and High-Level Language Software for Field Traffic Control*. Technical Report. California Department of Transportation, Sacramento, Dec. 1989.
6. N. G. Gartner. OPAC: A Demand-Responsive Strategy for Traffic Signal Control. In *Transportation Research Record 906*, TRB, National Research Council, Washington, D.C., 1983, pp. 75–81.
7. D. Bullock and C. Hendrickson. *A Model for Roadway Traffic Control Software*. Technical Report. Carnegie Mellon University, Pittsburgh, Pa., Dec. 1992.
8. J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Real Time Systems Symposium*, IEEE Computing Society, Dec. 1989, pp. 166–171.
9. J. Lehoczky, L. Sha, and J. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Real Time Systems Symposium*, IEEE Computing Society, 1987, pp. 261–270.
10. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, Jan. 1973, pp. 46–61.

The views expressed by the authors do not necessarily reflect the individual views or policies of either the California Department of Transportation or the University of California.

Publication of this paper sponsored by Committee on Traffic Signal Systems.